Quintus Prolog User's Manual

by the Intelligent Systems Laboratory

Swedish Institute of Computer Science
PO Box 1263
SE-164 29 Kista, Sweden

Release 3.5
December 2003

Swedish Institute of Computer Science
qpsales@sics.se http://www.sics.se/quintus/

This manual corresponds to:

Quintus Prolog Release 3.5

SICS PO Box 1263
SE-164 29 Kista, Sweden
+46 8 633 1500
http://www.sics.se/quintus/

Authorization Codes:
qpadmin@sics.se

Problem Reports and Product Suggestions:
qpsupport@sics.se

Additional Information:
qpsales@sics.se

Mailing List Subscription:
majordomo@sics.se
with '`subscribe quintus-users`' in the message body

Table of Contents

1 Introduction

1.1 About this Manual

1.1.1 Overview

The purpose of this manual is to provide a complete description of the Quintus Prolog system. It is not necessary to read the entire manual before starting to use Quintus Prolog. On the contrary, it is organized so that you should be able to quickly find out whatever you need to know about the system.

Start by reading Chapter 1. Section 1.2 [int-hig], page 4 provides an overview of all the new functionality and includes references to places where this new functionality is documented. Section 1.3 [int-dir], page 11 provides an overview of the installation hierarchy, which will be helpful in orienting you at many points in the manual.

Familiarity with some form of Prolog programming is assumed, at least at the level of textbooks such as those listed at the end of this chapter. It is the particular syntax, semantics and functionality of Quintus Prolog that is treated here. The material in the first 10 chapters is meant to introduce you to Quintus Prolog, not Prolog in general.

The entire manual is accessible on-line as discussed in Section 8.17 [ref-olh], page 304, and, for Quintus User Interface users, Section 3.6 [qui-hlp], page 68. This facility allows you to access the manual directly from Prolog whenever you need to look something up.

The manual, printed and on-line, is supplemented by various on-line resources including tutorials, demos, and code comments in library packages. See Section 1.3 [int-dir], page 11 for the location of these materials in the Quintus Directory.

1.1.2 Organization of the Manual

The documentation for release 3 consists of 19 chapters (see the Table of Contents).

There is a separate reference entry, following a standard format, for every Prolog built-in predicate, callable function, and command-line tool. They are arranged alphabetically for ease of reference. The other parts of the manual describe how predicates, functions, and tools are used, and how they work together, but the reader will be referred to the “Reference Pages” for a detailed description of each predicate, function, and tool.

While reading the descriptions of functionality provided, you can always find more detail on these routines in the Reference Pages. Since they are listed alphabetically within those major groupings, explicit cross-references to them are not spelled out.

Preceding the reference pages of each category there is a list of predicates, functions, or tools arranged by functional categories to assist the reader in locating unfamiliar names of routines.

At the end of each section, you will be referred to relevant reference pages and libraries when such supporting documentation or packages exist. Reference pages include cross-references to related reference pages and to relevant sections throughout the manual.

1.1.3 Notational Conventions

1.1.3.1 Goal Templates and Mode Annotations

Goal templates such as this are used in reference page synopses and elsewhere:

```
setof(+Template, +*Generator, *Set)
```

Here *Template*, *Generator*, and *Set* are meta-variables, mnemonic names for the arguments. Preceding each meta-variable is a symbol indicating the mode of the argument: whether it is an input or output, and information about its determinacy. These mode annotations are discussed in detail in Section 18.1.2 [mpg-ref-mod], page 986.

> **Please note:** The system of mode annotations used in release 3 has not been applied to the Quintus Prolog Library.

1.1.3.2 Examples

Examples illustrating interactive terminal sessions show what the user types in as well as system output. The operating system prompt is represented as '`% `' in examples. For example:

```
% prolog +
| ?- display(a+b).
+(a,b)
yes
```

1.1.3.3 Operating System Dependencies

The name of certain command line tools and file extensions are operating system dependent. When reading this manual, you may have to substitute the actual name for the one used in the manual. This applies to:

`cc` The C compiler, called `cc` or `gcc` under UNIX and `cl` under Windows.

`ld` The linker, called `ld` under UNIX and `link` under Windows.

‘`.o`’ File extension for object files, ‘`.o`’ under UNIX and ‘`.obj`’ under Windows.

‘`.so`’ File extension for shared object files, ‘`.sl`’ under HPUX, ‘`.so`’ under other UNIX, and ‘`.dll`’ under Windows.

‘`.a`’ File extension for archive files, ‘`.a`’ under UNIX and ‘`.lib`’ under Windows.

1.1.4 Bibliographical Notes

There are now a number of excellent books that teach Prolog. The following six books offer fully comprehensive courses in Prolog.

Programming in Prolog
William Clocksin and Christopher Mellish, Springer Verlag 1987, (third edition), ISBN 0-387-17539-3.

Prolog: A Logical Approach
Tony Dodd, Oxford University Press 1990, ISBN 0-19-853821-9.

Advanced Prolog
Peter Ross, Addison Wesley 1989, ISBN 0-201-17527-4.

Prolog Programming for Artificial Intelligence
Ivan Bratko, Addison Wesley 1990 (second edition), ISBN 0-201-41606-9.

The Art of Prolog, 2nd ed.
Leon Sterling and Ehud Shapiro, MIT Press 1994, ISBN 0-262-19338-8.

Prolog Programming In Depth
Michael Covington, Donald Nute and Andre Vellino, Prentice-Hall, 1996, ISBN 0-13-138645-X.

More advanced texts:

The Craft of Prolog
Richard O’Keefe, MIT Press 1990, ISBN 0-262-15039-5.
An advanced text dedicated to the proposition that elegance is not optional.

The Practice of Prolog
Leon Sterling (ed.), MIT Press 1990, ISBN 0-262-19301-9.
Each chapter presents and explains a particular application program written in Prolog.

Artificial Intelligence: A Modern Approach
Stuart Russell, Peter Norvig, Prentice Hall 1995, ISBN 0-13-103805-2.
A textbook on Artificial Intelligence using Prolog.

Techniques for Prolog Programming
T. Van Lee, John Wiley, 1993.

Computational Intelligence—A Logical Approach
David Poole, Alan Mackworth, Randy Goebel, Oxford University Press, 1998, ISBN 0-195-10270-3.

Natural Language Processing for Prolog Programmers
Michael Covington, Prentice Hall, 1994, ISBN 0-13-629213-5.

From Logic Programming to Prolog
K. Apt. Prentice-Hall, 1997, ISBN 0-132-30368-X.

The author explains the procedural and logical interpretation of Prolog programs, which eases the transition for C programmers.

Tools used in the Quintus Prolog programming environment are documented in these manuals:

GNU Emacs Manual, Version 20
Richard Stallman, Free Software Foundation, 1998.

Introduction to the X Window System
Jones, Oliver, Prentice-Hall, 1988, ISBN 0-13-499997-5.

An introduction to programming with Xlib

X Window System: Programming and Applications With Xt
Young, Douglas A., OSF/Motif Edition, Prentice-Hall, 1990, ISBN 0-13-497074-8.

A basic tutorial on writing programs using the Xt and Motif toolkits.

OSF/Motif Series
(5 volumes) Open Software Foundation, Prentice Hall, 1990. The volumes include Motif Style Guide, Programmer's Guide, Programmer's Reference, User's Guide, and Application Environment Specification (AES) User Environment Volume. Editions of these books, available for Release 1.1 and Release 1.2.

The X Window System Series
(8 volumes), O'Reilly and Associates, 1988, 1989, 1990.

X Window System Toolkit
Asente, Paul J. and Ralph R. Swick, DEC Press, 1990, ISBN 1-55558-051-3.

The X Toolkit bible.

1.2 Highlights of release 3

This section summarizes the new functionality in release 3.

1.2.1 Embeddability

Embeddability means the ability to embed a Prolog sub-program in a program written in some other language or languages. It is described in Section 10.2 [fli-emb], page 365. The specific features of this release that contribute towards embeddability are:

- **Callable Prolog:** In earlier releases, Prolog could call routines written in other languages but the reverse was not possible. Now each language can call the other, allowing much greater freedom in the way that multi-language programs can be organized (see Section 10.4 [fli-ffp], page 413).
- **Terms in C:** Compound terms can be passed to and from C. Routines are provided for testing, unifying, comparing and constructing terms in C. (See Section 10.3.8 [fli-p2f-trm], page 395)
- **Access to C data structures:** Arithmetic evaluation has been extended so that elements of C data structures can be accessed. Also, there is a new built-in predicate `assign/2`, which allows assignment into C data structures (see Chapter 18 [mpg], page 985).
- **Open OS Interface:** The main interfaces between the Prolog system and the operating system are now open. That is, the I/O and memory management interfaces are a set of documented functions, which can be replaced by user-defined functions. Source code is supplied for the default versions of these functions. (See Section 10.2 [fli-emb], page 365)
- **Discontiguous Memory Management:** As in previous releases, the system requests memory from the operating system only as it needs it, and it frees it up again when possible. (This contrasts with many other Prolog implementations in which all needed memory must be pre-allocated.) An important difference with this release is that the allocated memory need not be contiguous. This allows Prolog to better co-exist with other components that share its process' address space.
- **User-defined `main()`:** There is no longer any necessity to use the default `main()` routine. An application may call individual Prolog predicates and may never need to start an interactive Prolog session (See Section 10.2 [fli-emb], page 365).
- **Signal Handling:** Prolog used to trap all signals and then call any signal handler that had been specified by a user. release 3 does not intercept signals that are being handled by the user's code.

1.2.2 QOF Loading and Saving

Since Release 2.5, the Runtime Generator compiler `qpc` has been made available as part of the Development System. `qpc` compiles Prolog files into QOF (Quintus Object Format) files. QOF files can be linked together to make an executable file, which is an extended version of the development system. If you have a Runtime Generator license, you have the alternative of linking your QOF files into a runtime system, which omits development features (the compiler and debugger) and which can be conveniently deployed to different machines since it requires no authorization code in order to be run.

Release 3 introduces the ability to load QOF files directly into a running Prolog system. Loading a QOF file is up to 100 times faster than compiling it from source. File loading has been adapted to take advantage fo this new functionality. For instance, the form

```
| ?- [file].
```

now loads '`file.qof`' if it exists and is more recent than '`file.pl`'; otherwise it compiles '`file.pl`'.

The saved-state produced by `save_program/1` is now a QOF file. This means that it is now portable between different hardware and operating systems as well as between all releases of Quintus Prolog. It can still be executed as if it were an executable file, or it can be loaded into a running Prolog system. Another alternative is to call the QOF-linker `qld` on it to convert it to an executable file. It is also possible to save individual predicates or modules into a QOF file.

Section 8.4 [ref-lod], page 189 and Section 8.5 [ref-sls], page 192 describe QOF loading and QOF saving respectively. See Section 9.1.1.5 [sap-srs-bas-cld], page 339 for how to link QOF files to make an executable file.

1.2.3 QUI: An X-based Development Environment

The Quintus User Interface (QUI) is a new development environment based on the X-Windows system. QUI makes use of windows, buttons and menus to increase programmer productivity, and it includes an internal editor, access to GNU Emacs and access to the on-line manual. See Chapter 3 [qui], page 53.

1.2.4 Source-linked Debugger

A source-linked debugger is provided via QUI as well as via Emacs, allowing you to single-step through your source code. The debugger works by modification of compiled code, so there is no longer any need to distinguish between "consulting" and "compiling"; all code is compiled unless declared dynamic.

The new debugger also helps you to find inefficiencies in your program by

- showing visually the creation of choice points, and
- distinguishing between determinate and nondeterminate exit from a goal.

The debugger is documented in Chapter 6 [dbg], page 113.

1.2.5 Other New Features

- **Exception Handler:** Allows the programmer to specify a recovery action to be taken if an exception occurs during the execution of a particular goal. All errors detected

by the system now cause an exception to be raised; exceptions can also be raised by calling the built-in predicate `raise_exception/1` (see Section 8.19 [ref-ere], page 310).

- **Message Handler:** Allows customization of the text of system messages and of how they are displayed to the user. Major uses are customization or internationalization of error messages, and building user interfaces. (See Section 8.20 [ref-msg], page 325)
- **New I/O:** The new I/O system has substantially faster character I/O and more efficient (buffer-based rather than character-based) user-defined streams. There is now no limit on the number of open streams other than that imposed by the operating system.
- **New Arithmetic:** Standard 32-bit integers and 64-bit floats are supported. Exceptions are raised on overflows. Variables in arithmetic expressions can be bound to expressions (not just numbers).
- **Advice Package:** Allows a developer to associate consistency checks ("advice") to be performed whenever specified predicates are entered or exited. Advice checking can be enabled/disabled selectively or globally during the development process (see Section 6.4 [dbg-adv], page 141).

1.2.6 Compatibility Issues

1.2.6.1 Saved States

`save/[1,2]` are gone. Section 8.5 [ref-sls], page 192 explains why these have been removed. In most cases `save_program/2` can be used in their place, with a little rearrangement of your code.

Foreign code is no longer included in saved-states. When a saved-state is used it needs to be able to find the relevant object files and cause them to be linked in. Since this linking can be slow, it will often be preferable to use `qld` to link your saved-state with its object files into an executable file. See Section 9.1.1.5 [sap-srs-bas-cld], page 339 for how to do this.

1.2.6.2 Error Reporting/Handling

In earlier releases, some errors caused simple failures. For example,

```
| ?- functor(A,B,C).
no
```

This is not logical, since it is easy to choose A, B, C such that `functor(A,B,C)` is true. Generally, built-in predicates should enumerate all their logical solutions or else raise an exception in cases such as this one where enumeration is impractical. Thus you now get:

```
| ?- functor(A,B,C).
! Instantiation error in argument 2 of functor/3
! goal:  functor(_530,_531,_532)
```

Existing code that relies on the old error behavior will need modification to take this into account. The insertion of appropriate `nonvar/1` checks is usually all that is required.

1.2.7 New Built-in Predicates

See the Reference Pages for information on the following new predicates.

`absolute_file_name/3`
: generalization of `absolute_file_name/2`

`add_spypoint/1`
: add a spypoint

`add_advice/3`
: specify an advice action for a particular port of a predicate

`append/3` list concatenation relation

`assign/2` assign a value to a foreign data structure

`at_end_of_file/0`
: test if the current input stream is at end of file

`at_end_of_file/1`
: test if the specified stream is at end of file

`at_end_of_line/0`
: test if the current input stream is at end of line

`at_end_of_line/1`
: test if the specified stream is at end of line

`callable/1`
: test if the a term is syntactically valid as an argument to `call/1`; that is, not a variable, a number or a database reference

`check_advice/0`
: enable advice-checking for all predicates with advice

`check_advice/1`
: enable advice-checking for the specified predicates

`compound/1`
: test if a term is a compound term

`current_advice/3`
: find out what advice exists

`current_spypoint/1`
: find out what spypoints exist

`db_reference/1`
: test if a term is a database reference

`extern/1`
: declare predicate to be callable from C

`ground/1`
: test if a term is ground (contains no unbound variables)

`hash_term/2`
: produce a hash-value corresponding to a term

`initialization/1`
: declare a goal to be called when a file is loaded or when an executable file containing it is run

`load_files/1`
: load source or QOF files

`load_files/2`
: load source or QOF files with specified options

`nocheck_advice/0`
: turn off all advice checking

`nocheck_advice/1`
: turn off advice checking for specified predicates

`on_exception/3`
: execute a goal in the context of an exception handler

`open/4`
: open a file with specified options

`peek_char/1`
: return the next character in the current input stream without consuming it

`peek_char/2`
: return the next character in specified stream without consuming it

`print_message/2`
: print an error, warning, help, silent or informational message

`print_message_lines/3`
: auxiliary routine for message printing

`prompt/3`
: examine or change the prompt for a particular stream

`raise_exception/1`
: raise an exception

`read_term/2`
: read a term from current input stream

`read_term/3`
: read a term from specified stream

`remove_advice/3`
: remove advice for specified port of a predicate

`remove_spypoint/1`
: remove a spypoint

`save_modules/2`
: save a module or modules to QOF

`save_predicates/2`
: save a predicate or predicates to QOF

`save_program/2`
: save the program state and specify goal to be run on start-up

`seek/4` byte-oriented random access to files

`simple/1` opposite of `compound/1`; true of variables, atoms, numbers and database references

`skip_line/0`
: skips characters on current input up to end of line

`skip_line/1`
: skips characters on specified stream up to end of line

`source_file/3`
: relation between source file, predicate and clause number

`volatile/1`
: declare that a predicate should be excluded when saving

`write_term/2`
: write a term to current output

`write_term/3`
: write a term to specified stream

1.2.8 New Hook Predicates

These predicates are called at appropriate times by the Prolog system and are defined by the user.

`user:display_help_file/3`
: define how on-line help is displayed

`user:message_hook/3`
: define how messages are displayed

`user:generate_message_hook/3`
: define the textual form of messages

`user:query_hook/2`
: define or bypass user-interaction

1.2.9 Removed Built-in Predicates

`save/[1,2]` has been removed as discussed above. Their names are still reserved so that we can use them in a future release.

The following predicates were supported only by the interpreter and have now been eliminated. If their functionality is required, it can be achieved by passing an explicit ancestors list to all the predicates that need it as an extra argument.

- `ancestors/1`
- `subgoal_of/1`
- `maxdepth/1`
- `depth/1`

The following predicates that were previously provided only because they are defined in other Prolog systems have now been removed. The user may supply definitions for them if desired. (Many of these just printed error messages in earlier releases.)

- `'LC'/0`
- `'NOLC'/0`
- `current_functor/2`
- `incore/1`
- `load_foreign_files/3`
- `log/0`
- `nolog/0`
- `plsys/1`
- `reinitialize/0`
- `restore/2`
- `revive/2`

1.3 The Quintus Directory

All Quintus products are designed to be installed in a single directory hierarchy. For each product, several different hardware and operating system platforms may be supported within the same directory structure, provided that all platforms are able to access this hierarchy using NFS. Also, multiple versions of each product may co-exist in the same hierarchy.

The Quintus directory, *quintus-directory* (as seen in the figures below), is the root of the whole installation, and is where the entire Quintus hierarchy is installed. The following sections describe the files and directories located directly under the *quintus-directory*: The structure of the Quintus Directory differs slightly between UNIX and Windows. We therefore describe the two cases separately.

1.3.1 Structure of the Quintus Directory under UNIX

Quintus-directory structure under UNIX

'`bin3.5`' Contains one subdirectory for each platform (and operating system) on which Quintus Prolog has been installed. For example, '`sun4-5`' contains the executables installed for a Sun4 running SunOS 5.x. When the manual refers to *runtime-directory*, it is the subdirectories of '`bin3.5`', such as '`sun4-5`', that are referred to. The *runtime-directory* for your platform is the default `runtime_directory` Prolog flag. These directories are generated automatically for each platform during the installation procedure.

'`generic`' a directory containing files shared by different platforms. Subdirectories are described below.

'`editor3.5`'
a directory containing the single directory '`gnu`', which contains '`.el`' and '`.elc`' files for the GNU Emacs interface. See Section 4.1 [ema-ove], page 79.

'`qui3.5`' a directory containing support files for the Quintus User Interface. See Section 3.1 [qui-qui], page 53.

'`proxl3.5`'
a directory for the ProXL Package. See Chapter 16 [pxl], page 749.

'`proxt3.5`'
a directory for the ProXT Package. See Chapter 17 [pxt], page 897.

'`dbi`', '`flex`', etc.
one installation directory exists add-on product installed: Quintus Database Interface, Flex, etc.

'`license3.5`'
a directory containing license files for Quintus Prolog and its add-on products

'`java3.5`' a directory containing Java software components

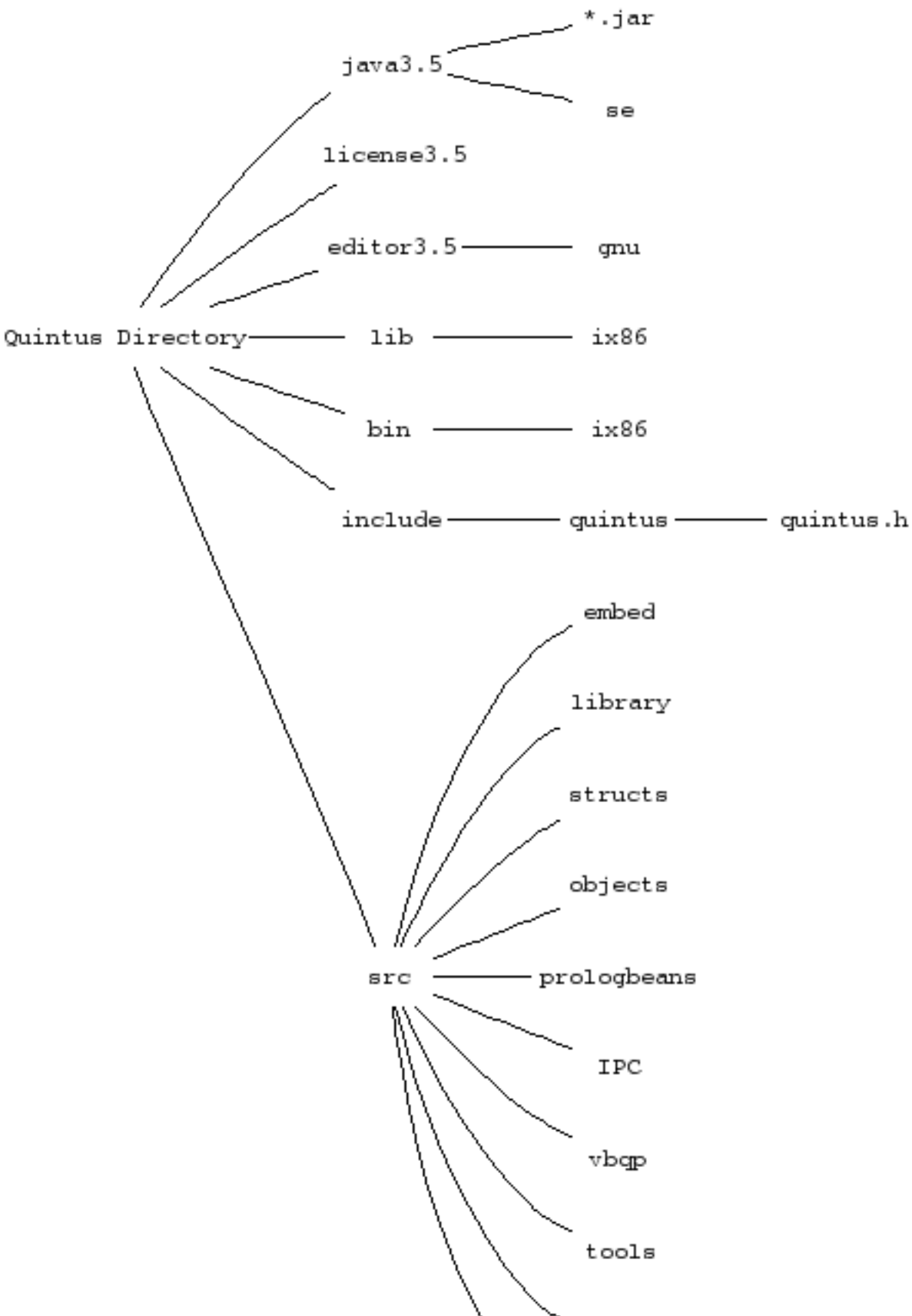
The subdirectories of '`generic`' are the library directory, '`qplib3.5`', and the Quintus information directory, '`q3.5`'.

'`qplib3.5`'
the library directory. This directory contains source and QOF files for the packages in the Quintus Prolog Library and the '`embed`' and '`tools`' directories. The contents of the library directory are detailed in Section 12.1 [lib-bas], page 521.

'`q3.5`' information about Quintus Prolog. Subdirectories:

- '`demo`' demonstration programs
- '`helpsys`' files for the Development System's on-line help-system
- '`man`' the man pages describing the executables found in the binary directory
- '`tutorial`'
  small programs demonstrating aspects of Quintus Prolog

1.3.2 Structure of the Quintus Directory under Windows



Quintus-directory structure under Windows

'`bin`' Contains the single subdirectory '`ix86`'. When the manual refers to *runtime-directory*, it is that subdirectory that is referred to. It is also the value of the `runtime_directory` Prolog flag.

'`lib`' Contains the single subdirectory '`ix86`', which contains import libraries and other files required for building Prolog executables.

'`include`' Contains '`<quintus/quintus.h>`'.

'`src`' a directory containing files shared by different platforms, in particular the library modules. The contents of this directory are detailed in Section 12.1 [lib-bas], page 521. Subdirectories include:

- '`demo`' demonstration programs
- '`helpsys`' files for the Development System's on-line help-system
- '`embed`' see Section 12.1 [lib-bas], page 521.
- '`tools`' see Section 12.1 [lib-bas], page 521.
- '`vbqp`' files for the Visual Basic interface.

'`editor3.5`'
a directory containing the single directory '`gnu`', which contains sq'.el' and '`.elc`' files for the GNU Emacs interface. See Section 4.1 [ema-ove], page 79.

'`dbi`', '`flex`', etc.
one installation directory exists add-on product installed: Quintus Database Interface, Flex, etc.

'`license3.5`'
a directory containing license files for Quintus Prolog and its add-on products

'`java3.5`' a directory containing Java software components

1.3.3 Search Paths

The absolute name of *quintus-directory* is returned by (A) and is, by default, used to set (B):

```
prolog_flag(quintus_directory, QuintusDir).          (A)

file_search_path(quintus, QuintusDir).               (B)
```

See Section 8.10.4.1 [ref-lps-flg-cha], page 246 for discussion of `prolog_flag/2`, and Section 8.6.1.4 [ref-fdi-fsp-pre], page 210 for discussion of predefined `file_search_path/2` facts.

2 User's Guide

2.1 Getting Started

2.1.1 Overview

This chapter describes things that you should know about Quintus Prolog. It assumes only the default interface, the way you can use Prolog on terminals that do not support X-Windows or Emacs.

This section describes how to run and halt Prolog, what you'll see once you've started Prolog, and how to use the on-line help system. Section 2.2 [bas-lod], page 21 describes how to load programs into Prolog. Section 2.3 [bas-run], page 27 discusses various features of the system related to running programs.

2.1.2 Starting Prolog

If you are using Windows, the batch file '*runtime-directory*`\qpvars.bat`' needs to be executed to set up the necessary environment variables. If you are using UNIX, your `PATH` environment variable needs to include the directory containing the Quintus tools.

To start Prolog, type *`prolog`* at the operating system prompt (whether UNIX or Windows):

```
% prolog
```

The system responds by displaying a copyright message followed by the main Prolog prompt, as shown below.

```
Quintus Prolog Release 3.5 (Sun 4, SunOS 5.5)

| ?-
```

The '`| ?- `' is the main Prolog prompt. It indicates that you are at the top level of the Prolog system. At this point, the system is waiting for you to type a goal, such as a command to load a previously created file containing a Prolog program.

If you are using Windows, it is probably more useful to run `qpwin` from the Start Menu. This has the same appearance as the console-based version, except that the output is directed to a window. The properties of this window can be tuned; see Section 20.1.1 [too-too-prolog], page 1476. If you use `qpwin` you do not need to run the '*runtime-directory*`\qpvars.bat`' batch file.

2.1.3 Exiting Prolog

To exit from Prolog, type your end-of-file character at the main Prolog prompt. (The standard end-of-file character is `^d` for UNIX and `^z` for Windows.)

```
| ?- ^D
```

Alternatively, you can exit from Prolog by typing `halt.` followed by a carriage return at the main Prolog prompt:

```
| ?- halt.⟨RET⟩
```

⟨RET⟩ stands for the Return key on your terminal. Note that a period followed by a ⟨RET⟩ must always be typed after a goal. The ⟨RET⟩ will usually not be shown explicitly but will be assumed in the examples that follow.

If all else fails, you can always use a `^c` interrupt followed by an `e` to exit.

2.1.4 The Top-level Prolog Prompt

The prompt '`| ?-`' indicates that Prolog is waiting for a goal to be typed in. For example, you can call built-in predicates like this:

```
| ?- write(hello).
hello
yes

| ?- X is 2+2.

X = 4 ⟨RET⟩

| ?-
```

When Prolog prints a variable binding at the top level like '`X = 4`' in this example, it waits for you to either type a ⟨RET⟩, which brings it back to the top level, or else type a `;`, which causes it to backtrack and look for another solution. In this case, you would get

```
| ?- X is 2+2.

X = 4 ;

no
| ?-
```

because there is only one X for which the goal can be satisfied.

It is always possible to interrupt any Prolog process and return to the top-level Prolog prompt. To do this, type `^c`. The system then displays the message

```
Prolog interruption (h for help)?
```

Type `a` (for abort) and press ⟨RET⟩. The system then displays a message indicating that execution has been aborted, followed by the top-level Prolog prompt.

```
! Execution aborted

| ?-
```

2.1.5 Using the On-line Help System

Quintus Prolog provides an on-line help system, which allows on-line access to this manual. The best ways to access the on-line manual are via QUI or Emacs, but it can also be accessed from the TTY interface. Type *`manual.`* at the main Prolog prompt to access the on-line help system as shown below.

```
| ?- manual.
```

The system then displays the following menu:

```
File: quintus.info,  Node: Top,  Next: int,  Prev: (dir),  Up: (dir)

Quintus Prolog
**************

* Menu:

* {manual(int)}                          Introduction
* {manual(bas)}                          User's Guide
* {manual(qui)}                          The Quintus User Interface
* {manual(ema)}                          The Emacs Interface
* {manual(vb)}                           The Visual Basic Interface
* {manual(dbg)}                          The Debugger
* {manual(glo)}                          Glossary
* {manual(ref)}                          The Prolog Language
* {manual(sap)}                          Creating Executables
* {manual(fli)}                          Foreign Language Interface
* {manual(ipc)}                          Inter-Process Communication
* {manual(lib)}                          Library
* {manual(str)}                          The Structs Package
* {manual(obj)}                          The Objects Package
* {manual(pbn)}                          The PrologBeans Package
* {manual(pxl)}                          The ProXL Package
* {manual(pxt)}                          The ProXT Package
* {manual(mpg)}                          Prolog Reference pages
* {manual(cfu)}                          C Reference Pages
* {manual(too)}                          Command Reference Pages
* {manual(pindex)}                       Predicate Index

* {manual(kindex)}                       Keystroke Index
* {manual(bindex)}                       Book Index

   This manual documents Quintus Prolog Release December 2003.

   Prolog is a simple but powerful programming language devel-
oped at the
University of Marseille, as a practical tool for programming in logic.
From a user's point of view the major attraction of the language is
ease of programming.  Clear, readable, concise programs can be written
quickly with few errors.

{text}
```

To see the table of contents for the Quintus User Interface, type

```
| ?- manual(qui).
```

The table of contents of the QUI part will then be shown as another menu. You can then choose a chapter/section in that part. For example, the second entry in the QUI menu is:

```
* {manual(qui-mai)}                    QUI Main Window
```

So, to find out about the main window of the QUI you would type:

```
| ?- manual(qui-mai).
```

In response, the system displays the appropriate section of the manual on the screen. You can continue typing goals of this form until you reach a file of text that does not begin with a menu.

You can control the way that manual files are written to the screen by setting your environment variable `PAGER` to the name of a program to be used to display a file. If `PAGER` is not set the default is `more(1)`.

If you wish to save an on-line manual section into a file it can be done like this:

```
| ?- tell('qui-mai.manual'), manual(qui-mai), told.
```

To request information about a specific topic, type *`help(Topic).`* at the main Prolog prompt, where *Topic* represents the topic you want information about. The system displays a menu generated by scanning the index of the manual for all entries containing the substring formed by *Topic*.

For example, you might type

```
| ?- help(debug).
```

if you are interested in learning about using the debugger. In response to this, the system will display a menu indicating all the parts of the manual whose index entries contain the substring '`debug`'. Note that you can abbreviate topics; if you type

```
| ?- help(deb).
```

you will get a menu referring to all topics containing with the substring '`deb`'. Thus, the more you abbreviate, the larger the menu you are likely to get. If the menu would only have one entry it is not displayed — that entry is selected automatically.

2.2 Loading Programs into Prolog

2.2.1 Loading a File into Prolog

If you have created a Prolog program and stored it in a file called '`myfile.pl`', you can load that file into Prolog by typing the following:

```
| ?- compile(myfile).
```

This goal has the effect of compiling your file into the Prolog database. A message is displayed showing the absolute filename:

```
% compiling /ufs/joe/myfile.pl...
```

When Prolog finishes compiling a file, it displays the name of the file that was compiled, the amount of time it took to compile the file, and the number of bytes required to store the compiled file in memory. If an earlier version of the file or another file of the same name has been previously compiled during this Prolog session, this last number represents the number of *additional* bytes required to recompile the file, and may be zero (or even negative, if the new version takes up less space than the old).

```
% myfile.pl compiled 2.354 sec 2346 bytes
| ?-
```

As shown above, the main Prolog prompt reappears after the system finishes compiling a file. At this point, you can begin running or testing by typing calls to the predicates that the file defines.

The predicate `compile/1` also accepts a list of files as an argument. For example, to compile three files called '`file1.pl`', '`file2.pl`', and '`file3.pl`', type

```
| ?- compile([file1,file2,file3]).
```

2.2.2 Loading Pre-Compiled (QOF) Files

It is possible to pre-compile files that you use frequently so that they can be rapidly loaded whenever needed. See Section 9.1.1.5 [sap-srs-bas-cld], page 339 for information on how to create such files. The standard naming convention is that the compiled file corresponding to a source file called '`myfile.pl`' is '`myfile.qof`'. If you use this convention, then the command

```
| ?- [myfile].
```

is appropriate: it loads either '`myfile.pl`' or '`myfile.qof`', using the more recent of the two if they both exist. **Please note:** you should not also have a file called just '`myfile`', without any extension, since this will take precedence over the '`.pl`' and the '`.qof`' files.

If you have several files to load, then you can use, for example:

```
| ?- [file1,file2,file3].
```

That is, a list of files typed as a goal is a command to load those files.

2.2.3 Commands in Files

A Prolog source file can contain commands as well as clauses. If you have a program that is spread across many files, it may be useful to create a file containing commands to load each of those files. For example, such a file could look like this:

```
:- compile(file1).
:- compile(file2).
:- compile(file3).
```

When this master-file is compiled, the '`% compiling File...`' and '`% File compiled`' messages for file1, file2 and file3 will be indented by one character. If they in turn cause other files to be loaded, the messages for those files will be indented two characters, and so on.

Notes:

1. The '`:-` ' symbol is placed at the beginning of the line just as it appears in the example above.
2. When a file being compiled contains a command to compile another file, a relative filename in that command is interpreted with reference to the directory that contains the first file. For example, if the file '`/usr/fred/test.pl`' contains the following commands

   ```
   :- compile('../whatsit').
   :- compile('xyz.pl').
   ```

   then the files to be compiled would be '`/usr/whatsit.pl`' (or '`/usr/whatsit`') and '`/usr/fred/xyz.pl`'.

   For example, you can have a file called '`mainfile.pl`' containing

   ```
   :- [file1, file2, file3].
   ```

   and provided that you keep all of these files in the same directory as '`mainfile.pl`', you can compile them all, no matter what your current working directory is, by giving `compile/1` a file specification for '`mainfile.pl`'.

2.2.4 Syntax Errors

If a clause being compiled contains a syntax error, Prolog tells you that a syntax error has been found and displays the clause. For example, suppose you accidentally omitted a closing parenthesis in a clause:

```
| ?- member(X,[a,b,c,d].
```

When you compile a file containing this clause, Prolog compiles all the clauses that precede the clause containing the error. When it reaches this clause, it displays the message:

```
! Syntax error
! between lines 26 and 27
! member(X,[a,b,c,d]
! <<here>>
```

to let you know

1. that the syntax of the clause is incorrect,
2. where the clause is in the file, and
3. at what point in the clause it found the syntax error.

Prolog then ignores the clause and continues loading the rest of the file into the database.

2.2.5 Style Warnings

In addition to checking for syntax errors, Quintus Prolog also has a style checker, which displays warning messages whenever certain stylistic conventions are violated in a program. Whereas syntax error messages indicate clauses that cannot be read into Prolog, style warnings simply indicate possible typing mistakes, or program construction that doesn't follow Quintus Prolog style conventions. The style conventions for Quintus Prolog are listed below. If you adhere to these conventions, you can use the style warnings to catch simple errors very easily.

1. Define all clauses for a given procedure in one file. This is essential; the load predicates do not allow the definition of a procedure to be spread across more than one file unless the procedure is declared multifile — see `multifile/1` for more information on this. If a non-multifile procedure is defined in more than one file, and all the files in which the procedure is defined are compiled, each definition of the procedure in a new file will wipe out any clauses for the procedure that were defined in previous files.
2. Make all clauses for a given procedure contiguous in the source file. This doesn't mean that you should avoid leaving blank space or putting comments between clauses, but simply that clauses for one procedure should not be interspersed with clauses for another procedure.
3. If a variable appears only once in a clause, either write that variable as the single character '`_`' (the void variable), or begin the variable name with the character '`_`'.

If any of these conditions are not met, you will be warned when the file containing the offending clauses is compiled.

If style convention 1 is violated, Prolog displays a message like the one shown below before it compiles each procedure that has been defined in another file that has already been loaded:

```
* Procedure foo/2 is being redefined in a different file -
*    Previous file: /ufs/george/file1
*    New file:      /ufs/george/file2
* Do you really want to redefine it? (y,n,p,s, or ?)
```

If you type *y*, the definition in the file currently being loaded replaces the existing procedure definition. If you type *n*, the existing definition remains intact, and the definition in the file currently being loaded is ignored. If you type *p* (for **p**roceed), the definition in the file currently being loaded replaces the existing definition; furthermore, the remaining procedure definitions in the file '`/ufs/george/file2`' will automatically replace any existing definitions made by the file '`/ufs/george/file1`' without displaying any warning messages. If you type *s* (for **s**uppress), the existing definition remains intact and the definition in the file currently being loaded is ignored; furthermore, the remaining procedure definitions in the file '`/ufs/george/file2`' which attempt to replace definitions made by the file '`/ufs/george/file1`' will be ignored without displaying any warning messages. (These options are particularly useful if you have changed the name or location of a file, since it suppresses the warnings you would otherwise get for every procedure in the file.) If you type `?`, Prolog displays a message that briefly describes each of the options above, and then asks you again if you want to redefine the procedure.

If style convention 2 is violated, you will get a message of the form:

```
* Clauses for foo/2 are not together in the source file
```

This indicates that between some pair of clauses defining procedure `foo/2`, there is a clause for some other procedure. If you followed the style conventions in writing your code, this message would indicate that some clause in your source file had either a mistyped name or the wrong arity, or that the predicate `foo/2` was defined more than once in the file. One other possible cause for this message might be that a period was typed in place of a comma, as in

```
foo(X, Y) :-
   goal1(X, Z),
   goal2(Z).
   goal3(X, Y).
foo([], []).
```

Here the Prolog system will think that you are defining a clause for `goal3/2` between the clauses for `foo/2`, and will issue a style warning.

If style convention 3 is violated, as in

```
check_state(TheState):-
   old_state(TheStaye, X),
   write(TheState),
   write(X).
```

you will get a message of the form:

```
* Singleton variables, clause 1 of check_state/1: TheStaye
```

indicating that in the first clause of procedure `check_state/1`, there is only one occurrence of the variable TheStaye. If that variable is a misspelling, you should correct the source text and recompile. If it was really meant to be a single variable occurrence, replace it with

the anonymous variable '_' or preface it with '_' as in '_TheStaye', and you will no longer get the style warning message.

It is good programming practice to respond immediately to these warnings by correcting the source text. By doing so, you will get the full benefit of the style warning facility in finding many errors painlessly.

By default, all the style warning facilities are turned on. You can turn off any or all of the style warning facilities by typing *no_style_check(X).* at the main Prolog prompt, where *X* represents one of the arguments listed below. To turn on style warning facilities, type *style_check(X).* at the main Prolog prompt, where *X* represents one of the arguments listed below.

Argument	Function
`all`	turns on (or off) all style checking
`single_var`	turns on (or off) checking for single variable occurrences
`discontiguous`	turns on (or off) checking for discontiguous clauses for procedures
`multiple`	turns on (or off) style checking for multiple definitions of same procedures (in different files)

For example, to turn off all the style warning facilities, you would type

```
| ?- no_style_check(all).
```

2.2.6 Saving and Restoring a Program State

2.2.6.1 Basic Information

Once a program has been loaded, its facts and rules are resident in the Prolog database. It is possible to save the current state of the database in its compiled form as a QOF file. This allows you to restore the current database at a later time without having to re-compile your Prolog source files.

The built-in predicate `save_program/1` saves the Prolog state. For example,

```
| ?- save_program(myprog).
```

You can later reload the file '`myprog`' into Prolog using the command

```
| ?- [myprog].
```

A saved program is a special kind of QOF file, which is capable of being run directly from the operating system, as if it were an executable file. To run a saved program from the

command prompt, type the name of the file containing the saved program at the command prompt. For example,

```
% myprog
```

This is equivalent to starting up Prolog and loading '`myprog`'. Under Windows, this only works if the name of the file has the extension '`bat`', e.g. '`myprog.bat`'.

You can also specify a goal to be run when a saved program is started up. This is done by:

```
| ?- save_program(myprog, start).
```

where `start/0` is the predicate to be called.

2.2.7 Using an Initialization File

If you use certain customized features often, you might want to direct the system to load them every time you start up Prolog. You can do this by creating an initialization file called '`prolog.ini`' in your home directory. This file is loaded, if it exists, every time you start up Prolog. It may be a Prolog source file or a QOF file.

> **Please note:** if you wish to start up Prolog or a Prolog saved program without loading your '`prolog.ini`' file you can use the '`+f`' for "fast start" option. That is,
>
> ```
> % prolog +f
> ```
>
> or
>
> ```
> % myprog +f
> ```
>
> will both start up Prolog without loading your '`prolog.ini`' file.

2.3 Running Programs

2.3.1 Overview

This section discusses certain features of Quintus Prolog that you will find helpful to know about when you run your programs.

2.3.2 Interrupting the Execution of a Program

You can interrupt the execution of a Prolog program at any time by typing `^c` (`^c^c` under GNU Emacs). For example, if you submit a query to Prolog and then decide you want to stop (abort) the query, type `^c` to which Prolog will respond by displaying the message

```
Prolog interruption (h for help)?
```

At this point, you can either type `h` to see a list of the options available to you, as shown below, or you can simply type the letter that corresponds to the option you want to select.

If you type `h`, Prolog displays the following list of options:

```
Prolog interrupt options:
  h  help         - this list
  c  continue     - do nothing
  d  debug        - debugger will start leaping
  t  trace        - debugger will start creeping
  a  abort        - abort to the current break level
  q  really abort - abort to the top level
  e  exit         - exit from Prolog
Prolog interruption (h for help)?
```

To select an option, type the letter that corresponds to that option and press ⟨RET⟩. For example, to stop the execution of the current query, type `a` followed by ⟨RET⟩. Prolog will print

```
! Execution aborted
```

and then return to its top level, displaying the main Prolog prompt.

Typing `c` causes the current procedure to continue executing as if nothing had happened. Typing `t` turns on the `trace` option of the debugger (see Section 6.1.5.1 [dbg-bas-con-tdz], page 118). Typing `d` turns on the `debug` option of the debugger (see Section 6.1.1 [dbg-bas-bas], page 113). Typing `a` causes the current query to be aborted and the main Prolog prompt to be redisplayed, as shown above. Typing `e` ends your Prolog session.

2.3.3 Errors, Warnings and Informational Messages

If your program calls a built-in predicate with arguments that are not appropriate for that predicate, the system will display an error message. For example, if your program called the goal

```
| ?- atom_chars(X,a).
```

you would get an error message like this

```
! Type error in argument 2 of atom_chars/2
! list expected, but a found
! goal:  atom_chars(_2016,a)
```

since `atom_chars/2` expects a list of characters (or a variable) as its second argument. The '`!`' prefix to each line signifies that this is an error. The other prefixes that are used are '`*`' for warnings and '`%`' for informational messages.

When an error occurs, your program is abandoned and you are returned to the top level. There is an exception handling mechanism, which can be used to prevent this in specified parts of your program. See Section 8.19.3 [ref-ere-hex], page 312 for more information.

A warning is less serious than an error; it indicates that something might be wrong. It may save you debugging time later to check it right away.

Informational messages are just messages to let you know what the system is doing.

All these messages can be customized if you wish. See Section 8.20 [ref-msg], page 325 for how to do this.

2.3.4 Undefined Predicates

By default, calling an undefined predicate is considered to be an error unless that predicate is known to be dynamic (see Section 2.3.6 [bas-run-dpr], page 30 for an explanation of dynamic predicates). For example,

```
| ?- f(x).
! Existence error in f/1
! procedure user:f/1 does not exist
! goal:  f(x)
```

You can change this behavior to make undefined predicates fail quietly by means of the built-in predicate `unknown/2`. There is also a facility, which allows you to have a predicate of your own called whenever an undefined predicate is called: see `unknown_predicate_handler/3`.

2.3.5 Executing Commands from Prolog

The built-in predicate `unix/1` enables you to execute system commands from within the Prolog environment. With some limitations it works also under Windows.

Under UNIX only, to access a shell (an interactive command interpreter) from within Prolog, call

```
| ?- unix(shell).
```

This command puts you within a command interpreter, from which you can execute any commands you would normally type at a command prompt. To return to Prolog, either type your end-of-file character (default: `^d`), or else type `exit`.

Alternatively, on both UNIX and Windows, you can access the shell and execute a command all at once:

```
| ?- unix(shell(Command)).
```

where *Command* is a Prolog atom representing the command you want to execute. For example, to obtain a listing of the files in your UNIX working directory:

```
| ?- unix(shell(ls)).
```

The same example under Windows would be

```
| ?- unix(shell(dir)).
```

Under UNIX, '`unix(shell).`' and '`unix(shell(`*Command*`)).`' use the command interpreter defined in your `SHELL` environment variable. If you want `sh(1)` instead, use '`unix(system)`' or '`unix(system(`*Command*`)).`'

A special case is made for the common command to change your working directory. To do so, call `unix(cd(`*Directory*`))`, where *Directory* is a Prolog atom naming the directory to change to. For example, to change to a directory named '`/ufs/albert`', you could type:

```
| ?- unix(cd('/ufs/albert')).
```

Notes:

1. The Prolog atom for the directory name '`/ufs/albert`' is surrounded by single quotes because it contains non-alphanumeric characters.
2. Under Windows, you can use either backward '\' or forward slash '/'.
3. This command only affects the current directory while in Prolog; after exiting Prolog, you will be in the directory from which Prolog was invoked.

The command `unix(cd)}` changes to your home directory.

For further information see Section 8.18 [ref-aos], page 307 and the reference page for `unix/1`.

2.3.6 Dynamic Predicates

All predicates in Prolog fall into one of two categories: *static* or *dynamic*. Dynamic predicates can be modified when a program is running; in contrast, static predicates can be modified only by reloading or by `abolish/[1,2]`.

If a predicate is first defined by being loaded from a file, it is static by default. Sometimes, however, it is necessary to add (assert), remove (retract), or inspect (using `clause/[2,3]`) clauses for a predicate while a program is running. In order to do that, you must declare the predicate to be dynamic. A predicate can be made dynamic by specifically declaring it to be so, as described below, or by using one of the assertion predicates. For a list of the assertion predicates, and for more information on using them, refer to Section 8.14.2 [ref-mdb-dsp], page 287.

To make a predicate dynamic, you insert in the file containing the predicate a line, which declares the predicate to be dynamic. The format of the line is

```
:- dynamic name/arity.
```

So, for example, the following declarations make the named predicates dynamic.

```
:- dynamic exchange_rate/3, spouse_of/2, gravitational_constant/1.
```

Notes:

1. The '`:-` ' symbol must appear at the beginning of any line with a dynamic declaration, as shown above.
2. Dynamic declarations can only appear in files; `dynamic/1` cannot be called as a predicate.
3. The line that declares a predicate to be dynamic must occur before any definition of the predicate itself in the file.

2.3.7 Prompts

The prompt '`|:` ' is displayed instead of the '`| ?-` ' prompt if your program requires input from the terminal. The built-in predicate `prompt/2` can be used to change the form of this prompt.

If you are typing a term at any Prolog prompt, and your input is longer than one line, all lines after the first one are indented five spaces. Sometimes this arises unexpectedly because of a typing error. For example, if you type

```
| ?- f('ABC).
     _
```

you will see your cursor positioned where the underscore character appears here. This signifies that you have not completed the input of a term: in this case there was no closing quote. To get back to the top level prompt type a closing quote followed by a period and a ⟨RET⟩. This will give a syntax error after which you can type the correct goal.

2.4 Limits in Quintus Prolog

This section describes the limits pertaining to atoms, functors, predicates, and other structures in Quintus Prolog.

Atoms cannot have more than 65532 characters.

Functors and predicates cannot have arities greater than 255.

There are no limits (apart from memory space) on the number of procedures or clauses allowed.

Prolog floating point numbers have 64 bit precision and conform to the IEEE 754 standard. The range of Prolog integers is -2147483648 (-2^31) to 2147483647 (2^31-1), both inclusive.

The size of a compiled clause is limited to 2^15 (32,768) bytes of compiled code.

There are internal limits on the size of compiled clauses, which are difficult to relate to properties visible to the user. These are 512 "temporary variables" (which only occur in the head goal), and 255 "permanent variables" (non-temporaries, which occur in goals in the body). The compiler will generate warnings if these limits are exceeded. There is no limit on the number of "symbols" (variables, atoms, numbers, or functors) in a compiled clause.

There are no restrictions on the number of variables or symbols in dynamic or interpreted clauses.

Prolog itself has no limit on the number of input/output streams that can be open at any one time. But the underlying Operating System might. For instance, some default configurations of UNIX might allow only 64 streams to be open at one time. Three of these streams are reserved for standard input, standard output, and error output respectively. These three streams are always open. Standard input and output normally refer to your terminal, but can be redirected from outside Prolog by means of operating system facilities. The error stream nearly always refers to the terminal, but can also be redirected.

Virtual memory for Prolog's data areas must come from the low 1 gigabyte of virtual memory. The maximum size of Prolog's data areas is also 1 gigabyte. Prolog expands its data areas as necessary. These areas can be contracted again by calling the built-in predicate `trimcore/0`. This predicate is automatically called on completion of every goal typed at the top level.

2.5 Writing Efficient Programs

2.5.1 Overview

This section gives a number of tips on how to organize your programs for increased efficiency. A lot of clarity and efficiency is gained by sticking to a few basic rules. This list is necessarily very incomplete. The reader is referred to textbooks such as *The Craft of Prolog* by Richard O'Keefe, MIT Press, 1990, a thorough exposition of the elements of Prolog programming style and techniques.

- Don't write code in the first place if there is a library predicate that will do the job.
- Write clauses representing base case before clauses representing recursive cases.
- Input arguments before output arguments in clause heads and goals.
- Use pure data structures instead of database changes.
- Use cuts sparingly, and *only* at proper places. A cut should be placed at the exact point that it is known that the current choice is the correct one: no sooner, no later.
- Make cuts as local in their effect as possible. If a predicate is intended to be determinate, define *it* as such; do not rely on its callers to prevent unintended backtracking.
- Binding output arguments before a cut is a common source of programming errors, so don't do it.
- Replace cuts by if-then-else constructs if the test is simple enough.
- Use disjunctions sparingly, *always* put parentheses around them, *never* put parentheses around the individual disjuncts, and *never* put the '`;`' at the end of a line.
- Write the clauses of a predicate so that they discriminate on the principal functor of the first argument (see below). For maximum efficiency, avoid "defaulty" programming ("catch-all" clauses).
- Don't use lists (`[...]`), "round lists" (`(...)`), or braces (`{...}`) to represent compound terms, or "tuples", of some fixed arity. The name of a compound term comes for free.

2.5.2 The Cut

2.5.2.1 Overview

One of the more difficult things to master when learning Prolog is the proper use of the cut. Often, when beginners find unexpected backtracking occurring in their programs, they try to prevent it by inserting cuts in a rather random fashion. This makes the programs harder to understand and sometimes stops them from working.

During program development, each predicate in a program should be considered *independently* to determine whether or not it should be able to succeed more than once. In most applications, many predicates should at most, succeed only once; that is, they should be *determinate*. Having decided that a predicate should be determinate, it should be verified that, in fact, it is. The debugger can help in verifying that a predicate is determinate (see Section 6.1.3 [dbg-bas-upe], page 115).

2.5.2.2 Making Predicates Determinate

Consider the following predicate, which calculates the factorial of a number:

```
fac(0, 1).
fac(N, X) :-
        N1 is N - 1,
        fac(N1, Y),
        X is N * Y.
```

The factorial of 5 can be found by typing

```
| ?- fac(5, X).

X = 120
```

However, backtracking into the above predicate by typing a semicolon at this point, causes an infinite loop because the system starts attempting to satisfy the goals '`fac(-1, X).`', '`fac(-2, X).`', etc. The problem is that there are two clauses that match the goal '`fac(0, F).`', but the effect of the second clause on backtracking has not been taken into account. There are at least three possible ways of fixing this:

1. Efficient solution: rewrite the first clause as

   ```
   fac(0,1) :- !.
   ```

   Adding the cut essentially makes the first solution the only one for the factorial of 0 and hence solves the immediate problem. This solution is space-efficient because as soon as Prolog encounters the cut, it knows that the predicate is determinate. Thus, when it tries the second clause, it can throw away the information it would otherwise need in order to backtrack to this point. Unfortunately, if this solution is implemented, typing *`fac(-1, X)`* still generates an infinite search.
2. Robust solution: rewrite the second clause as

   ```
   fac(N, X) :-
           N > 0,
           N1 is N - 1,
           fac(N1, Y),
           X is N * Y.
   ```

   This also solves the problem, but it is a more robust solution because this way it is impossible to get into an infinite loop.

   This solution makes the predicate *logically* determinate — there is only one possible clause for any input — but the Prolog system is unable to detect this and must waste space for backtracking information. The space-efficiency point is more important than it may at first seem; if `fac/2` is called from another determinate predicate, and if the cut is omitted, Prolog cannot detect the fact that `fac/2` is determinate. Therefore, it will not be able to detect the fact that the calling predicate is determinate, and space will be wasted for the calling predicate as well as for `fac/2` itself. This argument

applies again if the calling predicate is itself called by a determinate predicate, and so on, so that the cost of an omitted cut can be very high in certain circumstances.

3. Preferred solution: rewrite the entire predicate as the single clause

```
fac(N, X) :-
    (   N > 0 ->
            N1 is N - 1,
            fac(N1, Y),
            X is N * Y
    ;   N =:= 0 ->
            X = 1
    ).
```

This solution is as robust as solution 2, and more efficient than solution 1, since it exploits conditionals with arithmetic tests (see Section 8.2.7 [ref-sem-con], page 186, and Section 2.5.8 [bas-eff-cdi], page 49, for more information on optimization using conditionals).

2.5.2.3 Placement of Cuts

Programs can often be made more readable by the placing of cuts as early as possible in clauses. For example, consider the predicate `p/0` defined by

```
p :- a, b, !, c, d.
p :- e, f.
```

Suppose that `b/0` is a test that determines which clause of `p/0` applies; `a/0` may or may not be a test, but `c/0` and `d/0` are not supposed to fail under any circumstances. A cut is most appropriately placed after the call to `b/0`. If in fact `a/0` is the test and `b/0` is not supposed to fail, then it would be much clearer to move the cut before the call to `b/0`.

A tool to aid in determinacy checking is included in the '`tools`' directory. It is described in depth in Section 2.5.5 [bas-eff-det], page 39.

2.5.2.4 Terminating a Backtracking Loop

Cut is also commonly used in conjunction with the generate-and-test programming paradigm. For example, consider the predicate `find_solution/1` defined by

```
find_solution(X) :-
        candidate_solution(X),
        test_solution(X),
        !.
```

where `candidate_solution/1` generates possible answers on backtracking. The intent is to stop generating candidates as soon as one is found that satisfies `test_solution/1`. If

the cut were omitted, a future failure could cause backtracking into this clause and restart the generation of candidate solutions. A similar example is shown below:

```
process_file(F) :-
        see(F),
        repeat,
            read(X),
            process_and_fail(X),
        !,
        seen.

process_and_fail(end_of_file) :- !.
process_and_fail(X) :-
        process(X),
        fail.
```

The cut in `process_file/1` is another example of terminating a generate-and-test loop. In general, a cut should always be placed after a `repeat/0` so that the backtracking loop is clearly terminated. If the cut were omitted in this case, on later backtracking Prolog might try to read another term after the end of the file had been reached.

The cut in `process_and_fail/1` might be considered unnecessary because, assuming the call shown is the only call to it, the cut in `process_file/1` ensures that backtracking into `process_and_fail/1` can never happen. While this is true, it is also a good safeguard to include a cut in `process_and_fail/1` because someone may unwittingly change `process_file/1` in the future.

2.5.3 Indexing

2.5.3.1 Overview

In Quintus Prolog, predicates are indexed on their first arguments. This means that when a predicate is called with an instantiated first argument, a hash table is used to gain fast access to only those clauses having a first argument with the same primary functor as the one in the predicate call. If the first argument is atomic, only clauses with a matching first argument are accessed. Indexes are maintained automatically by the built-in predicates manipulating the Prolog database (for example, `assert/1`, `retract/1`, and `compile/1`).

Keeping this feature in mind when writing programs can help speed their execution. Some hints for program structuring that will best use the indexing facility are given below. Note that dynamic predicates as well as static predicates are indexed. The programming hints given in this section apply equally to static and dynamic code.

2.5.3.2 Data Tables

The major advantage of indexing is that it provides fast access to tables of data. For example, a table of employee records might be represented as shown below in order to gain fast access to the records by employee name:

```
%  employee(LastName,FirstNames,Department,Salary,DateOfBirth)

employee('Smith', ['John'], sales,       20000, 1-1-59).
employee('Jones', ['Mary'], engineering, 30000, 5-28-56).
...
```

If fast access to the data via department is also desired, the data can be organized little differently. The employee records can be indexed by some unique identifier, such as employee number, and additional tables can be created to facilitate access to this table, as shown in the example below. For example,

```
%  employee(Id,LastName,FirstNames,Department,Salary,DateOfBirth)

employee(1000000, 'Smith', ['John'], sales,       20000, 1-1-59).
employee(1000020, 'Jones', ['Mary'], engineering, 30000, 5-28-56).
...

%  employee_name(LastName,EmpId)

employee_name('Smith', 1000000).
employee_name('Jones', 1000020).
...

%  department_member(Department,EmpId)

department_member(sales,       1000000).
department_member(engineering, 1000020).
...
```

Indexing would now allow fast access to the records of every employee named Smith, and these could then be backtracked through looking for John Smith. For example:

```
| ?- employee_name('Smith', Id),
     employee(Id, 'Smith', ['John'], Dept, Sal, DoB).
```

Similarly, all the members of the engineering department born since 1965 could be efficiently found like this:

```
| ?- department_member(engineering, Id),
     employee(Id, LN, FN, engineering, _, M-D-Y),
     Y > 65.
```

2.5.3.3 Determinacy Detection

The other advantage of indexing is that it often makes possible early detection of determinacy, even if cuts are not included in the program. For example, consider the following simple predicate, which joins two lists together:

```
concat([], L, L).
concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
```

If this predicate is called with an instantiated first argument, the first argument indexing of Quintus Prolog will recognize that the call is determinate — only one of the two clauses for `concat/3` can possibly apply. Thus, the Prolog system knows it does not have to store backtracking information for the call. This significantly reduces memory use and execution time.

Determinacy detection can also reduce the number of cuts in predicates. In the above example, if there was no indexing, a cut would not strictly be needed in the first clause as long as the predicate was always to be called with the first argument instantiated. If the first clause matched, then the second clause could not possibly match; discovery of this fact, however, would be postponed until backtracking. The programmer might thus be tempted to use a cut in the first clause to signal determinacy and recover space for backtracking information as early as possible.

With indexing, if the example predicate is always called with its first argument instantiated, backtracking information is *never* stored. This gives substantial performance improvements over using a cut rather than indexing to force determinacy. At the same time greater flexibility is maintained: the predicate can now be used in a nondeterminate fashion as well, as in

```
| ?- concat(L1, L2, [a,b,c,d]).
```

which will generate on backtracking all the possible partitions of the list [a,b,c,d] on backtracking. If a cut had been used in the first clause, this would not work.

2.5.4 Last Clause Determinacy Detection

Even if the determinacy detection made possible by indexing (see Section 2.5.3.3 [bas-eff-ind-det], page 38) is unavailable to a predicate call, Quintus Prolog still can detect determinacy before determinate exit from the predicate. Space for backtracking information can thus be recovered as early as possible, reducing memory requirements and increasing performance. For instance, the predicate `member/2` (found in the Quintus Prolog library) could be defined by:

```
member(Element, [Element|_]).
member(Element, [_|Rest]) :-
        member(Element, Rest).
```

`member/2` might be called with an instantiated first argument in order to check for membership of the argument in a list, which is passed as a second argument, as in

```
| ?- member(4, [1,2,3,4]).
```

The first arguments of both clauses of `member/2` are variables, so first argument indexing cannot be used. However, determinacy can still be detected before determinate exit from the predicate. This is because on entry to the last clause of a nondeterminate predicate, a call becomes effectively determinate; it can tell that it has no more clauses to backtrack to. Thus, backtracking information is no longer needed, and its space can be reclaimed. In the example, each time a call fails to match the first clause and backtracks to the second (last) clause, backtracking information for the call is automatically deleted.

Because of last clause determinacy detection, a cut is never needed as the first subgoal in the last clause of a predicate. Backtracking information will have been deleted before a cut in the last clause is executed, so the cut will have no effect except to waste time.

Note that last clause determinacy detection is exploited by dynamic code as well as static code in Quintus Prolog.

2.5.5 The Quintus Determinacy Checker

The Quintus determinacy checker can help you spot unwanted nondeterminacy in your programs. This tool examines your program source code and points out places where nondeterminacy may arise. It is not in general possible to find exactly which parts of a program will be nondeterminate without actually running the program, but this tool can find most unwanted nondeterminacy. Unintended nondeterminacy should be eradicated because

1. it may give you wrong answers on backtracking
2. it may cause a lot of memory to be wasted

2.5.5.1 Using the Determinacy Checker

There are two different ways to use the determinacy checker, either as a stand-alone tool, or during compilation. You may use it whichever way fits best with the way you work. Either way, it will discover the same nondeterminacy in your program.

The stand-alone determinacy checker is called `qpdet`, and is run from the shell prompt, specifying the names of the Prolog source files you wish to check. You may omit the '`.pl`' suffix if you like.

```
% qpdet [-r [-d] [-D] [-i ifile] fspec...]
```

The `qpdet` program is placed in the Quintus '`tools`' directory, and is not built by default when Prolog is installed, so you may have to build it (by typing *`make qpdet`* in the '`tools`' directory) first. The tool takes a number of options:

'`-r`' Process files recursively, fully checking the specified files and all the files they load.

'`-d`' Print out declarations that should be added.

'`-D`' Print out all needed declarations.

'`-i` *`ifile`*' An initialization file, which is loaded before processing begins.

The determinacy checker can also be integrated into the compilation process, so that you receive warnings about unwanted nondeterminacy along with warnings about singleton variables or discontinuous clauses. To make this happen, simply insert the line

```
:- load_files(library(detcheck),
              [when(compile_time), if(changed)]).
```

Once this line is added, every time that file is compiled, whether using `qpc` or the compiler in the development system, it will be checked for unwanted nondeterminacy.

2.5.5.2 Declaring Nondeterminacy

Some predicates are intended to be nondeterminate. By declaring intended nondeterminacy, you avoid warnings about predicates you intend to be nondeterminate. Equally importantly, you also inform the determinacy checker about nondeterminate predicates. It uses this information to identify unwanted nondeterminacy.

Nondeterminacy is declared by putting a declaration of the form

```
:- nondet name/arity.
```

in your source file. This is similar to a dynamic or discontiguous declaration. You may have multiple `nondet` declarations, and a single declaration may mention several predicates, separating them by commas.

Similarly, a predicate P/N may be classified as nondeterminate by the checker, whereas in reality it is determinate. This may happen e.g. if P/N calls a dynamic predicate that in reality never has more than one clause. To prevent false alarms asiring from this, you can inform the checker about determinate predicates by declarations of the form:

```
:- det name/arity.
```

If you wish to include `det` and `nondet` declarations in your file and you plan to use the stand-alone determinacy checker, you must include the line

```
:- load_files(library(nondetdecl),
              [when(compile_time), if(changed)]).
```

near the top of each file that contains such declarations. If you use the integrated determinacy checker, you do not need (and should not have) this line.

2.5.5.3 Checker Output

The output of the determinacy checker is quite simple. For each clause containing unexpected nondeterminacy, a single line is printed showing the module, name, arity, and clause number (counting from 1). The form of the information is:

```
* Non-determinate: module:name/arity (clause number)
```

A second line for each nondeterminate clause indicates the cause of the nondeterminacy. The recognized causes are:

- The clause contains a disjunction that is not forced to be determinate with a cut or by ending the clause with a call to `fail/0` or `raise_exception/1`.
- The clause calls a nondeterminate predicate. In this case the predicate is named.
- There is a later clause for the same predicate whose first argument has the same principal functor (or one of the two clauses has a variable for the first argument), and this clause does not contain a cut or end with a call to `fail/0` or `raise_exception/1`. In this case, the clause number of the other clause is mentioned.
- If the predicate is multifile, clause indexing is not considered sufficient to ensure determinacy. This is because other clauses may be added to the predicate in other files, so the determinacy checker cannot be sure it has seen all the clauses for the predicate. It is good practice to include a cut (or fail) in every clause of a multifile predicate.

The determinacy checker also occasionally prints warnings when declarations are made too late in the file or not at all. For example, if you include a `dynamic`, `nondet`, or `discontiguous` declaration for a predicate after some clauses for that predicate, or if you put a `dynamic` or `nondet` declaration for a predicate after a clause that includes a call to that predicate, the determinacy checker may have missed some nondeterminacy in your program. The checker also detects undeclared discontiguous predicates, which may also have undetected nondeterminacy. Finally, the checker looks for goals in your program that indicate that predicates are dynamic; if no `dynamic` declaration for those predicates, you will be warned.

These warnings take the following form:

```
! warning: predicate module:name/arity is property.
!          Some nondeterminacy may have been missed.
!          Add (or move) the directive~n
!              :- property module:name/arity.
!          near the top of this file.
```

2.5.5.4 Example

Here is an example file:

```
:- load_files(library(detcheck),
              [when(compile_time), if(changed)]).

parent(abe, rob).
parent(abe, sam).
parent(betty, rob).
parent(betty, sam).

is_parent(Parent) :- parent(Parent, _).
```

The determinacy checker notices that the first arguments of clauses 1 and 2 have the same principal functor, and similarly for clauses 3 and 4. It reports:

```
* Non-determinate: user:parent/2 (clause 1)
*     Indexing cannot distinguish this from clause 2.
* Non-determinate: user:parent/2 (clause 3)
*     Indexing cannot distinguish this from clause 4.
```

In fact, `parent/2` should be nondeterminate, so we should add the declaration

```
:- nondet parent/2.
```

before the clauses for `parent/2`. If run again after modifying file, the determinacy checker prints:

```
* Non-determinate: user:is_parent/1 (clause 1)
*     This clause calls user:parent/2, which may be nondeterminate.
```

It no longer complains about `parent/2` being nondeterminate, since this is declared. But now it notices that because `parent/2` is nondeterminate, then so is `is_parent/1`.

2.5.5.5 Options

When run from the command line, the determinacy checker has a few options to control its workings.

The '`-r`' option specifies that the checker should recursively check files in such a way that it finds nondeterminacy caused by calls to other nondeterminate predicates, whether they are declared so or not. Also, predicates that appear to determinate will be treated as such, whether declared nondet or not. This option is quite useful when first running the checker on a file, as it will find all predicates that should be either made determinate or declared nondet at once. Without this option, each time a `nondet` declaration is added, the checker may find previously unnoticed nondeterminacy.

For example, if the original example above, without any `nondet` declarations, were checked with the '`-r`' option, the output would be:

```
* Non-determinate: user:parent/2 (clause 1)
*     Indexing cannot distinguish this from clause 2.
* Non-determinate: user:parent/2 (clause 3)
*     Indexing cannot distinguish this from clause 4.
* Non-determinate: user:is_parent/1 (clause 1)
*     Calls nondet predicate user:parent/2.
```

The '`-d`' option causes the tool to print out the needed `nondet` declarations. These can be readily pasted into the source files. Note that it only prints the `nondet` declarations that are not already present in the files. However, these declarations should not be pasted into your code without each one first being checked to see if the reported nondeterminacy is intended.

The '`-D`' option is like '`-d`', except that it prints out all `nondet` declarations that should appear, whether they are already in the file or not. This is useful if you prefer to replace all old `nondet` declarations with new ones.

Your code will probably rely on operator declarations and possibly term expansion. The determinacy checker handles this in much the same way as `qpc(1)`: you must supply an initialization file, using the '`-i` *`ifile`*' option.

2.5.5.6 What is Detected

As mentioned earlier, it is not in general possible to find exactly which places in a program will lead to nondeterminacy. The determinacy checker gives predicates the benefit of the doubt: when it's possible that a predicate will be determinate, it will not be reported. The checker will only report places in your program that will be nondeterminate regardless of which arguments are bound. Despite this, the checker catches most unwanted nondeterminacy in practice.

The determinacy checker looks for the following sources of nondeterminacy:

- multiple clauses that can't be distinguished by the principal functor of the first arguments, and are not made determinate with an explicit cut, `fail/0`, `false/0`, or `raise_exception/1`. First argument indexing is not considered for multifile predicates, because another file may have a clause for this predicate with the same principal functor of its first argument.
- a clause with a disjunction not forced to be determinate by a cut, `fail/0`, `false/0`, or `raise_exception/1` in each arm of the disjunction but the last, or where the whole disjunction is followed by a cut, `fail/0`, `false/0`, or `raise_exception/1`.
- a clause that calls something known to be nondeterminate, other than when it is followed by a cut, `fail/0`, `false/0`, or `raise_exception/1`, or where it appears in the condition of an if-then-else construct. Known nondeterminate predicates include those

declared nondeterminate or dynamic (since they can be modified, dynamic predicates are assumed to be nondeterminate), plus the following built-in predicates:

- `absolute_file_name/3`, when the second argument is a list containing the term `solutions(all)`
- `bagof/3` , when the second argument contains any variables not appearing earlier in the clause (including the clause head).
- `clause/[2,3]`
- `current_op/3`, when any argument contains any variables not appearing earlier in the clause (including the clause head).
- `current_key/2`, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
- `current_predicate/2`, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
- `length/2`, when both arguments are variables not appearing earlier in the clause (including the clause head).
- `predicate_property/2`, when either argument contains any variables not appearing earlier in the clause (including the clause head).
- `recorded/3`
- `repeat/0`
- `retract/1`
- `setof/3`, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
- `source_file/[1,2,3]` when the last argument contains any variables not appearing earlier in the clause (including the clause head).

2.5.6 Last Call Optimization

Another important efficiency feature of Quintus Prolog is last call optimization. This is a space optimization technique, which applies when a predicate is determinate at the point where it is about to call the last goal in the body of a clause. For example,

```
% for(Int, Lower, Upper)
% Lower and Upper should be integers such that Lower =< Upper.
% Int should be uninstantiated; it will be bound successively on
% backtracking to Lower, Lower+1, ... Upper.

for(Int, Int, _Upper).
for(Int, Lower, Upper) :-
        Lower < Upper,
        Next is Lower + 1,
        for(Int, Next, Upper).
```

This predicate is determinate at the point where the recursive call is about to be made, since this is the last clause and the preceding goals (`</2` and `is/2`) are determinate. Thus

last call optimization can be applied; effectively, the stack space being used for the current predicate call is reclaimed before the recursive call is made. This means that this predicate uses only a constant amount of space, no matter how deep the recursion.

2.5.6.1 Accumulating Parameters

To take best advantage of this feature, make sure that goals in recursive predicates are determinate, and whenever possible put the recursive call at the end of the predicate.

This isn't always possible, but often can be done through the use of *accumulating parameters*. An accumulating parameter is an added argument to a predicate that builds up the result as computation proceeds. For example, in our factorial example (see Section 2.5.2.2 [bas-eff-cut-mpd], page 34), the last goal in the body of the recursive case is `is/2`, not the recursive call to `fac/2`.

```
fac(N, X) :-
    (   N > 0 ->
            N1 is N - 1,
            fac(N1, Y),
            X is N * Y
    ;   N =:= 0 ->
            X = 1
    ).
```

This can be corrected by adding another argument to `fac/2` to accumulate the factorial.

```
fac(N, X) :- fac(N, 1, X).

%  fac(+N, +M, -X)
%  X is M * the factorial of N.

fac(N, M, X) :-
    (   N > 0 ->
            N1 is N - 1,
            M1 is N * M,
            fac(N1, M1, X)
    ;   N =:= 0 ->
            X = M
    ).
```

Here we do the multiplication before calling `fac/3` recursively. Note that we supply the base case, 1, at the start of the computation, and that we are multiplying by decreasing numbers. In the earlier version, `fac/2`, we multiply after the recursive call, and so we multiply by increasing numbers. Effectively, the new version builds the result backwards. This is correct because multiplication is associative.

2.5.6.2 Accumulating Lists

This technique becomes much more important when extended to lists, as in this case it can save much building of unneeded lists through unnecessary calls to append sublists together. For example, the naive way to reverse a list is:

```
nreverse([], []).
nreverse([H|T], L) :-
        nreverse(T, L1),
        append(L1, [H], L).
```

This is very wasteful, since each call to `append/3` copies the initial part of the list, and adds one element to it. Fortunately, this can be very easily rewritten to use an accumulating parameter:

```
reverse(L1, L2) :- reverse(L1, [], L2).

%  reverse(+X, +Y, -Z)
%  Z is X reversed, followed by Y
reverse([], Z, Z).
reverse([H|T], L0, L) :-
        reverse(T, [H|L0], L).
```

This version of reverse is many times faster than the naive version, and uses much less memory. The key to understanding the behavior of this predicate is the observation made earlier: using an accumulating parameter, we build the result backwards.

Don't let this confuse you. Building a list forward is easy. For example, a predicate that returns a list L of consecutive numbers from 1 to N could be written in two different ways: counting up and collecting the resulting list forward, or counting down and accumulating the result backward.

```
iota1(N, L) :- iota1(1, N, L).

iota1(N, Max, L) :-
        (   N > Max ->
                L = []
        ;   N1 is N+1,
            L = [N|L1],
            iota1(N1, Max, L1)
        ).
```

or,

```
iota2(N, L) :- iota2(N, [], L).

iota2(N, L0, L) :-
        (   N =< 0 ->
                L = L0
        ;   N1 is N-1,
            iota2(N1, [N|L0], L)
        ).
```

Both versions generate the same results, and neither waste any space. The second version is slightly faster. Choose whichever approach you prefer.

2.5.7 Building and Dismantling Terms

The built-in predicate `=../2` is a clear way of building terms and taking them apart. However, it is almost never the most efficient way. `functor/3` and `arg/3` are generally much more efficient, though less direct. The best blend of efficiency and clarity is to write a clearly-named predicate that implements the desired operation and to use `functor/3` and `arg/3` in that predicate.

Here is an actual example. The task is to reimplement the built-in predicate `==/2`. The first variant uses `=../2` (this symbol is pronounced "univ" for historical reasons). Some Prolog textbooks recommend code similar to this.

```
ident_univ(X, Y) :-
        var(X),                 % If X is a variable,
        !,
        var(Y),                 % so must Y be, and
        samevar(X, Y).          % they must be the same.
ident_univ(X, Y) :-             % If X is not a variable,
        nonvar(Y),              % neither may Y be;
        X =.. [F|L],            % they must have the
        Y =.. [F|M],            % same function symbol F
        ident_list(L, M).       % and identical arguments

ident_list([], []).
ident_list([H1|T1], [H2|T2]) :-
        ident_univ(H1, H2),
        ident_list(T1, T2).

samevar(29, Y) :-               % If binding X to 29
        var(Y),                 % leaves Y unbound,
        !,                      % they were not the same
        fail.                   % variable.
samevar(_, _).                  % Otherwise they were.
```

This code performs the function intended; however, every time it touches a non-variable term of arity *N*, it constructs a list with *N*+1 elements, and if the two terms are identical, these lists are reclaimed only when backtracked over or garbage-collected.

Better code uses `functor/3` and `arg/3`.

```
ident_farg(X, Y) :-
        (   var(X) ->           % If X is a variable,
                var(Y),         % so must Y be, and
                samevar(X, Y)   % they must be the same;
        ;   nonvar(Y),          % otherwise Y must be nonvar
            functor(X, F, N),   % The principal functors of X
            functor(Y, F, N),   % and Y must be identical,
            ident_farg(N, X, Y) % including the last N args.
        ).

ident_farg(0, _, _) :- !.
ident_farg(N, X, Y) :-          % The last N arguments are
        arg(N, X, Xn),          % identical
        arg(N, Y, Yn),          % if the Nth arguments
        ident_farg(Xn, Yn),     % are identical,
        M is N-1,               % and the last N-1 arguments
        ident_farg(M, X, Y).    % are also identical.
```

This approach to walking through terms using `functor/3` and `arg/3` avoids the construction of useless lists.

The pattern shown in the example, in which a predicate of arity *K* calls an auxiliary predicate of the same name of arity *K*+1 (the additional argument denoting the number of items remaining to process), is very common. It is not necessary to use the same name for this auxiliary predicate, but this convention is generally less prone to confusion.

In order to simply find out the principal function symbol of a term, use

```
| ?- the_term_is(Term),
|    functor(Term, FunctionSymbol, _).
```

The use of `=../2`, as in

```
| ?- the_term_is(Term),
|    Term =.. [FunctionSymbol|_].
```

is wasteful, and should generally be avoided. The same remark applies if the arity of a term is desired.

`=../2` should not be used to locate a particular argument of some term. For example, instead of

```
Term =.. [_F,_,ArgTwo|_]
```

you should write

```
arg(2, Term, ArgTwo)
```

It is generally easier to get the explicit number "2" right than to write the correct number of "don't care" variables in the call to `=../2`. Other people reading the program will find the call to `arg/3` a much clearer expression of the program's intent. The program will also be more efficient. Even if several arguments of a term must be located, it is clearer and more efficient to write

```
arg(1, Term, First),
arg(3, Term, Third),
arg(4, Term, Fourth)
```

than to write

```
Term =.. [_,First,_,Third,Fourth|_]
```

Finally, `=../2` should not be used when the functor of the term to be operated on is known (that is, when both the function symbol and the arity are known). For example, to make a new term with the same function symbol and first arguments as another term, but one additional argument, the obvious solution might seem to be to write something like the following:

```
add_date(OldItem, Date, NewItem) :-
        OldItem =.. [item,Type,Ship,Serial],
        NewItem =.. [item,Type,Ship,Serial,Date].
```

However, this could be expressed more clearly and more efficiently as

```
add_date(OldItem, Date, NewItem) :-
        OldItem = item(Type,Ship,Serial),
        NewItem = item(Type,Ship,Serial,Date).
```

or even

```
add_date(item(Type,Ship,Serial),
         Date,
         item(Type,Ship,Serial,Date)
        ).
```

2.5.8 Conditionals and Disjunction

There is an efficiency advantage in using conditionals whose test part consists only of arithmetic comparisons or type tests. Consider the following alternative definitions of the predicate `type_of_character/2`. In the first definition, four clauses are used to group characters on the basis of arithmetic comparisons.

```
type_of_character(Ch, Type) :-
        Ch >= "a", Ch =< "z",
        !,
        Type = lowercase.
type_of_character(Ch, Type) :-
        Ch >= "A", Ch =< "Z",
        !,
        Type = uppercase.
type_of_character(Ch, Type) :-
        Ch >= "0", Ch =< "9",
        !,
        Type = digit.
type_of_character(_Ch, Type) :-
        Type = other.
```

In the second definition, a single clause with a conditional is used. The compiler generates optimized code for the conditional; the second version of `type_of_character/2` runs faster than the first and uses less memory.

```
type_of_character(Ch, Type) :-
        (   Ch >= "a", Ch =< "z" ->
                Type = lowercase
        ;   Ch >= "A", Ch =< "Z" ->
                Type = uppercase
        ;   Ch >= "0", Ch =< "9" ->
                Type = digit
        ;   otherwise ->
                Type = other
        ).
```

Following is a list of builtin predicates that are compiled efficiently in conditionals:

- `atom/1`
- `atomic/1`
- `callable/1`
- `compound/1`
- `db_reference/1`
- `float/1`
- `integer/1`
- `nonvar/1`
- `number/1`
- `simple/1`
- `var/1`
- `</1`
- `=</1`

- `=:=/1`
- `=\=/1`
- `>=/1`
- `>/1`
- `@</1`
- `@=</1`
- `==/1`
- `\==/1`
- `@>=/1`
- `@>/1`

2.5.9 The Quintus Cross-Referencer

2.5.9.1 Introduction

The main purpose of the cross-referencer, `qpxref`, is to find undefined predicates and unreachable code. To this end, it begins by looking for initializations, hooks and `public` directives to start tracing the reachable code from. If an entire application is being checked, it also traces from `user:runtime_entry/1`. If individual module-files are being checked, it also traces from their export lists.

A second function of `qpxref` is to aid in the formation of module statements. `qpxref` can list all of the required `module/2` and `use_module/2` statements by file.

2.5.9.2 Basic Use

The cross-referencer is run from the shell prompt, specifying the names of the Prolog source files you wish to check. You may omit the '`.pl`' suffix if you like.

```
% qpxref [-R] [-v] [-c] [-i ifile] [-w wfile] [-x xfile] [-u ufile
] fspec ...
```

The `qpxref` program is placed in the Quintus '`tools`' directory, and is not built by default when Prolog is installed, so you may have to build it (by typing *`make qpxref`* in the '`tools`' directory) first. The tool takes a number of options, as follows. *File* arguments should be given as atoms or as '`-`', denoting the standard output stream.

'`-R`' Check an application, i.e. follow `user:runtime_entry/1`, as opposed to module declarations.

'`-c`' Generate standard compiler style error messages.

'`-v`' Verbose output. This echoes the names of the files being read.

'`-i ifile`' An initialization file, which is loaded before processing begins.

'`-w wfile`' Warning file. Warnings are written to the standard error stream by default.

'`-x xfile`' Generate a cross-reference file. This is not generated by default.

'`-m mfile`' Generate a file indicating which predicates are imported and which are exported for each file. This is not generated by default.

'`-u ufile`' Generate a file listing all the undefined predicates. This is not generated by default.

2.5.9.3 Practice and Experience

Your code will probably rely on operator declarations and possibly term expansion. The cross-referencer handles this in much the same way as `qpc(1)`: you must supply an initialization file, using the '`-i ifile`' option.

Supply meta-predicate declarations for your meta-predicates. Otherwise, the cross-referencer will not follow the meta-predicates' arguments. Be sure the cross-referencer encounters the meta-predicate declarations *before* it encounters calls to the declared predicates.

The cross-referencer traces from initializations, hooks, predicates declared `public`, and optionally from `user:runtime_entry/1` and module declarations. The way it handles meta-predicates requires that your application load its module-files before its non-module-files.

This cross-referencer was written in order to tear out the copious dead code from the application that the author became responsible for. If you are doing such a thing, the cross-referencer is an invaluable tool. Be sure to save the output from the first run that you get from the cross referencer: this is very useful resource to help you find things that you've accidentally ripped out and that you really needed after all.

There are situations where the cross-referencer does not follow certain predicates. This can happen if the predicate name is constructed on the fly, or if it is retrieved from the database. In this case, add `public` declarations for these. Alternatively, you could create term expansions that are peculiar to the cross-referencer.

3 The Quintus User Interface

3.1 Quintus User Interface

The Quintus User Interface (QUI) is a Motif-based window interface to the Quintus Prolog development system. It is not available in the Windows distribution. It includes the following:

- Main Window: Query interpreter window with history menu
- Debug window: See Section 6.2 [dbg-sld], page 121
- Edit windows
- Interface to external editors, such as GNU Emacs
- Error dialogue windows
- Help window

All these windows facilitate rapid program development by making common Prolog commands available using the mouse. For more information about Motif widgets such as the file browser, see the "OSF/Motif Series" and other references cited in Section 1.1.4 [int-man-bib], page 3.

3.1.1 Starting Up QUI

Before starting up QUI, you must be running an X-windows server process on the machine upon which you want QUI windows to be displayed.

To access the QUI facilities, you must invoke a Quintus Prolog Development System executable that has the QUI libraries included in it. See the Quintus Prolog Development System installation instructions for more details on how to install such an executable.

Assuming a Quintus Prolog Development System executable that includes QUI is installed under the name `qui`, you invoke it by typing,

```
% qui
```

The QUI main window will be displayed on your screen. See the figure below.

To run QUI across local network on a machine called *remote* and have QUI displayed on a machine called *host*, you should first issue the command

```
% xhost +remote
```

on the machine *host*, and do either of the following commands on the machine *remote* when running under `csh` type,

(1) `% qui -display host:0.0`

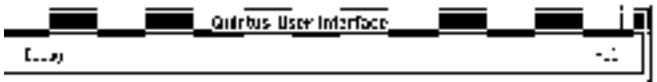
(2) `%` set environment variable `DISPLAY` to *host*`:0.0`
`% qui`

Apart from those X resources that you can specify in environment variables you can set X resources in the various X resource database files that are referenced by this Motif application. See Section 3.7 [qui-ciq], page 72 for more information.

3.1.2 Exiting QUI

To exit QUI, select the '`Quit`' option from the '`File`' menu. Alternatively, type *`halt.`*, *`^D`* (^D is the end of file character), *`^C`* (^C is the interrupt key) and exit option, or type *`end_of_file.`* at the Prolog toplevel prompt '`| ?-`' when you are not inside a break level. In any of these cases, a dialogue window will pop-up asking for confirmation. Choosing the **Exit** button in the dialogue window will exit QUI while choosing the **Cancel** button will return to the '`| ?-`' prompt.

3.2 QUI Main Window



3.2.1 Main Window Menu Bar

3.2.1.1 File Pulldown

Edit... Begin editing a file. The editor will allow you to choose the file to edit. By default, the QUI editor will be used, but you may choose another editor with the `QUINTUS_EDITOR_PATH` environment variable, as explained in Section 3.4 [qui-ied], page 65. The QUI editor is described in Section 3.3 [qui-edi], page 59.

Load... Load a file using `load_files/1`. A Motif file selection dialog will be displayed to allow you to specify the file to load. By default, this button is only enabled when Prolog is at the toplevel prompt. It can be changed through the resource file so that this button is disabled only when Prolog code is running enabling you to reload code while you are debugging (see Section 3.7.2 [qui-ciq-cqr], page 73).

Log –> Write all or part of the history to a file. Putting your mouse cursor near the right end of this menu item will display a submenu with two choices: '`Entire Session...`' and '`From Selection...`'. '`Entire Session...`' will log the entire session to a file, while '`From Selection...`' will only log the current selection. After selecting either of the options, a dialog will be presented in which you can enter the name of the file in which to write the log. Note that you can select a large range conveniently by selecting one end, scrolling to the other end, and then selecting it *while holding down either shift key.*

Quit... Exit Prolog (and QUI). First, a dialogue is displayed asking if you are sure you want to exit Prolog. If you indicate that you are sure, execution is halted and all QUI windows are closed. You will be returned to the operating system shell that invoked QUI.

3.2.1.2 Debug Pulldown

Trace (Creep Initially)
puts the debugger into trace mode

Debug (Leap Initially)
puts the debugger into leap mode

Zip (Zip Initially)
puts the debugger into zip mode

Nodebug turns off the debugger

Selecting one of the first three choices will open the debugger window (if it's not already open). Choosing **Nodebug** will close it. Note that the diamond to the left of one of these

menu items will be darkened, indicating your current debugging mode. See Section 6.1.5.1 [dbg-bas-con-tdz], page 118 for an explanation of these modes.

3.2.1.3 Help Pulldown

See Section 3.6.1 [qui-hlp-hlp], page 69 for more information.

3.2.2 QUI Query History Menu

The query history menu contains all previously entered queries. Initially this menu is empty. Only queries will appear in this menu, and not input submitted to general Prolog I/O predicates. When you single click on an entry in the query history menu, that entry will replace any text that has already been typed into the query interpreter sub-window. When you double click on an entry, that entry will replace any text that has already been typed into the query interpreter sub-window and the query will be submitted to Prolog. A history menu entry will be compressed into a single line even if it consisted of multiple lines when it was submitted.

3.2.3 QUI Query Interpreter Sub-Window

> **Motif caveat:** When the main window originally appears, the insertion point caret is not visible. To make it visible, click the left mouse button anywhere after the Prolog prompt.

3.2.3.1 Prolog Output and Input

The output from Prolog (both `stdout` and `stderr`) is redirected to this window. As output is being redirected to this window, the window is scrolled so that the insertion point is always visible.

Input to Prolog is also received from this window. When Prolog is expecting to receive a term (which also includes a query), the term will not be transmitted to Prolog until its full-stop is followed by a newline. This allows you to edit previously typed lines in a multi-line term before transmitting those lines to Prolog. When Prolog is expecting to receive input that is not a term, a newline will immediately transmit the line just typed.

The text that appears before the current prompt is not editable. You cannot type characters into this window while output is also being redirected into it.

3.2.3.2 Key Bindings

The following key bindings have been added to the existing Motif text widget key bindings in the query interpreter sub-window:

⟨DEL⟩ Deletes the character to the left of the insertion point.

`^C` Interrupts Prolog execution (see Section 3.2.4 [qui-mai-int], page 58 for more information).

`^D` Deletes the next character. If there are no characters on the current line, then an end of file signal is transmitted to Prolog.

`^K` Deletes all characters to the right of the insertion point on the current line.

`^U` Deletes all the characters to the left of the insertion point up to the Prolog prompt on the current line.

`^P` Moves the insertion point to the previous line.

`^N` Moves the insertion point to the next line.

`^A` Moves the insertion point to the beginning of the current line.

`^E` Moves the insertion point to the end of the current line.

`^B` Moves the insertion point one character to the left.

`^F` Moves the insertion point one character to the right.

3.2.4 QUI Interrupt Button

When you select the **Interrupt** button, Prolog execution is suspended. A selection dialogue is then displayed from which you will determine the next action,

Continue Execution is resumed. Execution will continue as if Prolog had never been interrupted.

Abort Execution is immediately aborted (see Section 8.11.1 [ref-iex-int], page 250 for more information about abort).

Trace Execution is continued but traps to the debug window when the next debug port is reached.

Quit Execution is halted and all QUI windows are closed.

Typing `^C` in the main window will also display the interrupt dialogue (see Section 3.2.3.2 [qui-mai-top-key], page 58 for more information).

3.2.5 QUI Next Answer Buttons

When Prolog prints a set of variable bindings for one solution, it waits to find out whether you want to see the bindings for the next solution, if any. When Prolog is in such a wait state the next answer buttons at the bottom of the main window are active.

Next Answer
: Displays the next set of variable bindings, if any, and waits again. If there are no more bindings, you will be returned to query input mode and the next answer buttons will be inactive.

No More Answers
: You will be returned to query input mode and the next answer buttons will be inactive.

Remaining Answers
: Displays all remaining sets of variable bindings; you will be returned to query input mode and the next answer buttons will be inactive. If there is an infinite number of remaining bindings then you must abort execution using the **Interrupt** button or its equivalent (default is `^C`) key binding (see Section 3.2.4 [qui-mai-int], page 58 for more information).

3.2.6 QUI Error Dialogue Window

Each Prolog error message is printed in its own dialogue window. Both types of dialogue window give you the option to either continue or abort the execution. Execution is suspended until you select one of these two options,

Continue Execution is continued following the exception handler that printed the error or warning message (see Section 8.19 [ref-ere], page 310 for more information about exception handling).

Abort Execution is immediately aborted (see Section 8.11.1 [ref-iex-int], page 250 for more information about abort).

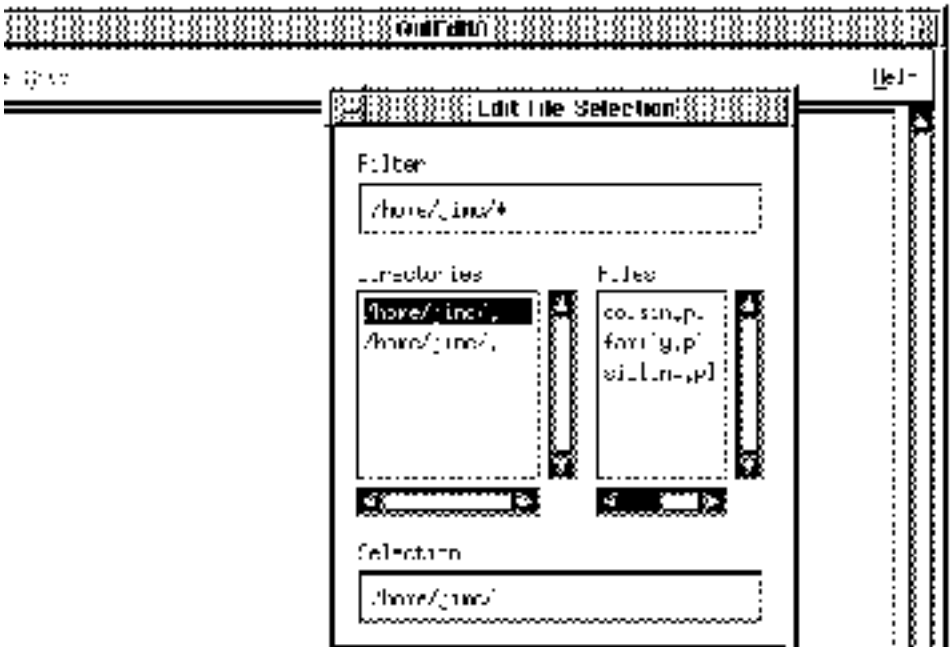
After you select an option, the error or warning message is placed in the Prolog output window so that subsequent inspection of the interactive session will show the context of the message. The *Execution aborted* error message is not displayed in a dialogue window.

3.3 Edit Windows

The name of the file that is loaded into the edit window is displayed in the title bar of the window. An indication of whether or not the file is modified also appears in the title bar. Below the title bar is the menu bar of options that are available in the edit window. Below the menu bar is the text window that contains the loaded file.

3.3.1 Invoking an Edit Window

An edit window is displayed by selecting the **Edit...** option of the **File** pulldown in the main window or the **Edit** option of the **File** pulldown in the debugger main window. See Section 3.2.1.1 [qui-mai-mai-fil], page 56 for more information. The following figure shows an edit window before a file is loaded.


Edit
Help
Edit File Selection
Filter
/home/jmc/*
Directories
/home/jmc/
/home/jmc/
Files
cousin.p
family.p
sibling.pl
Selection
/home/jmc/

Edit Window with File Browser

3.3.2 File Pulldown

Pulling down your File Pulldown menu will give you the options listed below.

Edit... Displays a file browser. The file you select is loaded into the edit window.

Compile The contents of the edit window is saved into the file associated with the edit window if the file has changed since loaded into the window or since last save and that file is compiled by the Prolog system. This item is disabled when compiling file is not permitted (see Section 3.2.1.1 [qui-mai-mai-fil], page 56).

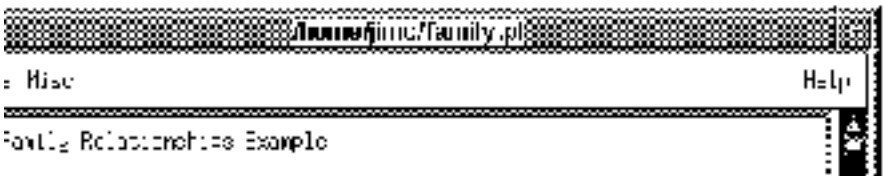
Save The contents of the edit window is saved into the file associated with the edit window.

Save As...
Displays a dialogue that prompts you for the name of a file. The contents of the edit window is saved into the file you specify.

Insert... Displays a file browser. The file you select is copied at the location of the insertion point.

Quit Quits the edit window. If the edit window has modifications that have not been saved, then it displays a dialogue that asks you whether you want to save the modifications.

Once you select **Edit** and load a file the edit window looks like this:



Edit Window with File Loaded

3.3.3 Misc Pulldown

The following choices may be selected by pulling down the Misc Pulldown menu.

Find and Replace...
Displays a dialogue that has the following options,

Find Find the string specified in the box to the right of this button in the direction indicated by the toggles (i.e. **forward** or **backward**). If the **wraparound** option is selected then the search wraps from bottom to top if the direction toggle is **forward** or from top to bottom is the direction toggle is **backward**.

Replace Replace the next occurrence of the string specified in the box to the right of the **Find** button with the string specified in the box to the right of this button in the direction indicated by the toggles (i.e. **forward** or **backward**). If the **wraparound** option is selected then the replacement wraps from bottom to top if the direction toggle is **forward** or from top to bottom is the direction toggle is **backward**.

Replace All
Replace all occurrences of the string specified in the box to the right of the **Find** button with the string specified in the box to the right of the **Replace** button.

Quit Quit the **Find and Replace...** dialogue box.

Go to A Line
Pops up a dialogue allowing you to enter a line number. The insertion point will move to the line you specify.

3.3.4 Help Pulldown

On This Window
Displays information regarding the edit window.

3.3.5 Key Bindings

The following key bindings have been added to the existing Motif text widget key bindings in the edit window,

⟨DEL⟩ Deletes the character to the left of the insertion point.

`^D` Deletes the character to the right of the insertion point.

`^K`	Deletes all characters to the right of the insertion point on the current line.
`^P`	Moves the insertion point to the previous line.
`^N`	Moves the insertion point to the next line.
`^A`	Moves the insertion point to the beginning of the current line.
`^E`	Moves the insertion point to the end of the current line.
`^B`	Moves the insertion point one character to the left.
`^F`	Moves the insertion point one character to the right.
`^L`	Redraws the display of the editor window.
`^H`	Deletes the previous character to the left of the insertion point.
`^J`	Inserts a new line to the right of the insertion points and indents the new line so that it starts from the same column as the current line.
`^M`	Inserts a new line to the right of the insertion point.
`^O`	Inserts a new line to the right of the insertion point without moving the insertion point.
`^T`	Moves the insertion point to the beginning of the file.
`^U`	Moves the insertion point to the previous page.
`^V`	Moves the insertion point to the next page.
`^W`	Deletes the selected region of text.
`^Y`	Yanks back deleted text.
`^X`	Moves the insertion point to the end of the file.

3.4 Interface to External Editors

While using the Quintus User Interface you can also interact with an external editor. You might choose to do this if you are more comfortable or familiar with another editor that QUI supports. Currently QUI only provides an interface to the GNU Emacs editor, in addition to the default QUI editor.

3.4.1 Interface to GNU Emacs

In order to set up GNU Emacs as the editor for QUI, you need to set the shell environment variable `QUINTUS_EDITOR_PATH` to the path of the GNU Emacs executable. You may also need to set the following shell variables:

`QUINTUS_LISP_PATH`
: When QUI invokes the GNU Emacs editor, it instructs GNU Emacs to load initial lisp files from '*quintus-directory*`/editor3.5/gnu`'.

`DISPLAY` If the `DISPLAY` variable is set GNU Emacs will try to create a window on the appropriate display, otherwise it will run in the window that invoked QUI.

3.4.1.1 Invoking GNU Emacs to Edit Files From QUI

There are two ways to invoke GNU Emacs from QUI. The first way is by selecting the **Edit...** item of the **File** pulldown menu in the QUI window. The second way is by selecting the **Edit Source** item of the **File** pulldown menu in the debugger window. In order to use this second method of invoking GNU Emacs, you must already have a file loaded in your Prolog program under QUI and it must be displayed in the debugger source window. Selecting this menu item will invoke GNU Emacs and place the cursor at the position of the file that defines the predicate you are currently debugging. If you have invoked GNU Emacs from the QUI window, a scratch buffer `*qui-emacs*` is displayed on start up. This scratch buffer is needed to set up the communication channels with QUI. Although the buffer is in `qui` mode, (which is the minor mode in which all files with the extension '`.pl`' are placed) you cannot use it as a normal prolog buffer as this buffer is not associated with any file. Note that the `*qui-emacs*` buffer is displayed only if you invoke the editor from the QUI main window. If you invoke the editor from the debugger, the file that defines the predicate you are debugging is displayed, not the `*qui-emacs*` buffer, which is hidden.

3.4.1.2 Key Bindings in "qui" mode

The key bindings are exactly the same as in the Prolog/Emacs interface. You can compile regions, procedures and buffers; you can search for definitions of predicates (find definition) and you can move around clauses.

To use find-definition in QUI, ensure that a gnuemacs process is running, move the cursor into that window, and proceed as in a straight emacs interface (see Section 4.2.2 [ema-emi-key], page 89). It cannot be done directly from the QUI main window, so most likely it will be necessary to type in the procedure name and arity rather than select it.

The only thing that is different is that there are no bindings to repeat previous queries to prolog as the QUI window serves that purpose. For a detailed description of the key bindings see Section 4.2.2 [ema-emi-key], page 89.

3.5 QUI Debug Window

The debug window is displayed either by selecting one of the first three options in the **Debug** pulldown in the menu bar of the main window (see Section 3.2.1.1 [qui-mai-mai-fil], page 56), by selecting the **Trace** button in the interrupt dialogue (see Section 3.2.4 [qui-mai-int], page 58), or by turning on the debugger with the `trace/0`, `debug/0`, or `prolog_flag/3` built-ins (see Section 6.1.5.1 [dbg-bas-con-tdz], page 118).

The debug window is described in the debugger section of this manual set. See Section 6.2 [dbg-sld], page 121 for its description.

3.6 QUI Help Window

The QUI Help Window

3.6.1 Invoking Help

The help window allows you to navigate through all parts of the on-line version of the Quintus manual set. This window is displayed using the **Help** pulldown on the control panel of the QUI Main Window or by calling `help/1` or `manual/[0,1]` (e.g. by typing `help(compile).`) at the top level. Each Prolog help predicate accesses the manual set in a different way. See Section 3.6.3.3 [qui-hlp-hwm-gto], page 71 for more information.

If any requests for help are invoked while the help window is already displayed, then this same help window is loaded with the newly requested portion of the manual set that is requested.

Instead of invoking help through the Prolog help predicates, you can use the **Help** Pulldown in the Control Panel of the QUI Main Window. Each button in this pulldown displays a different part of the manual set:

On Prolog Displays the menu of on-line Quintus manuals.

On QUI Displays the on-line version of this description of the Quintus User Interface.

On This Window
Displays information regarding the main window.

3.6.2 Help Window

The name of the current manual is displayed in the title bar of the window. Below the title bar is the menu bar of options that are available in the help window. Below the menu bar is the name of the current section. Below the name is the current section, either a menu of sub-sections or text.

3.6.2.1 Selecting a Sub-Section from a Menu

When a menu is displayed, you can display one of the sub-sections listed by selecting the appropriate menu item. When you select a line of the menu, the sub-section indicated by the line is displayed.

3.6.2.2 Following Cross-References in Text

When a text section is displayed, it may contain cross-references. Cross-references are underlined. When you double-click on the cross-reference, the section indicated by it is displayed.

3.6.2.3 Selecting a Topic in Text

When a text section is displayed, you can select a new topic by pointing to a word and clicking the left mouse button. That word will then be highlighted. If you click again on the highlighted word then it is sent to Prolog via `help/1`. It is the same as entering the topic through '`index...`' dialogue in the **Goto** pulldown menu.

A recognized topic is a word in the form of [A-Z a-z][A-Z a-z 0-9_-]* possibly followed by a slash preceding an arity specification such as '`/1`' and '`/[1,2]`'. Words consisting of non-alphanumeric characters such as '`@>`' will not be recognized.

3.6.3 Help Window Menu Bar

3.6.3.1 File Pulldown

Quit Selecting this button quits the help window. Any subsequent help command will redisplay the window.

3.6.3.2 Goto Pulldown

Each button in this pulldown allows you to directly display various parts of the manual set.

Next Section
Travel to the following adjacent section of the manual. It is like paging through a paper manual.

Previous Section
Travel to the previous adjacent section of the manual.

Parent When you select this button, the parent (a menu) of the current section is displayed. This button is not available when the current section does not have a parent. The only sections that do not have a parent are the top-most menu of manuals and any menu generated from index entries.

Top When you select this button, the top-most menu of manuals is displayed.

Index... When you select this button, a dialogue that prompts you for an index topic is displayed. When you enter a valid index topic, the help window will generate and display a menu of index entries that refer to that topic. If you enter an invalid index topic, the display will remain as it was. The form of the index topic is the same as the one passed to `help/1`.

Section...
When you select this button, a dialogue that prompts you for a manual section is displayed. When you enter a valid section, the help window will display that

section. If you enter an invalid section, an error dialogue will appear. The form of the section is the same as the one passed to `manual/1`.

3.6.3.3 Invoking Goto Options from Prolog Predicates

Each Prolog help predicate is mapped to a QUI **Goto** pulldown button.

`manual/0` Same as selecting the **Top** button from the **Goto** pulldown.

`manual/1` Same as selecting the **Section** button from the **Goto** pulldown.

`help/1` Same as selecting the **Index** button from the **Goto** pulldown.

3.6.3.4 History Pulldown

This pulldown contains a dynamic list of buttons, which represent the menus and text that have already been displayed in the help window. Initially this history list is empty. As each new section is displayed, the section that it replaces is placed at the top of this list. If a section that is already in the history list is redisplayed, it is not added to the list a second time. The list contains 10 buttons by default. The number of buttons it contains can be modified through the resource file (see Section 3.7 [qui-ciq], page 72). When more than 10 (or whatever other number of buttons you have specified in the resource file) sections have been viewed, the oldest section is removed from the history list. Each history entry that refers to a menu generated from index entry matches is indicated as such so that there is no ambiguity between it and a section entry with the same name. When you select an element from the history list, that section is displayed and the history list is not changed. When you quit the help window, the history list is cleared.

3.6.3.5 Misc Pulldown — Search

Currently the only item in this menu is **Search**. This button is available when the Help Window contains text rather than a menu. It brings up a dialog that enables you to search the text for a selected string.

Selection Type in the search string.

Forward to find the next occurrence of the string. (Or type ⟨RET⟩.)

Backward to search to the beginning of the file.

Wraparound
activates wraparound search.

Cancel The **Search** dialog remains on display, preserving its state, until you select **Cancel**. While it is on display selecting **Search** from the **Misc** pulldown will bring the dialog to the foreground.

When a match is found, the matching text is highlighted. A beep indicates that there are no (more) occurences of the string to be found by one search — forward or back according to which type of search you selected — of the frame. If **Wraparound** is on, a beep will be sounded only if there are no occurences of the string in the frame.

Where does it start searching?

- If a word or pattern is highlighted then the searching starts from that point.
- Otherwise the search begins from the beginning of the text.

Customization is discussed in Section 3.7.2.7 [qui-ciq-cqr-hsr], page 76.

3.7 Customizing and Interfacing with QUI

The resource file and how it is used in customizing QUI is discussed in Section 3.7.1 [qui-ciq-qrf], page 72 and Section 3.7.2 [qui-ciq-cqr], page 73.

Restrictions on developing programs under QUI are the topic of Section 3.7.3 [qui-ciq-dpq], page 76.

3.7.1 The QUI Resource File

By convention, applications developed for the X11 Window System maintain a database of *resources*. This database is built at run time from application defaults and user preferences.

A resource, in the context of X11, is any customizable data that controls the behaviour and appearance of the application. This includes just about everything: colors, fonts, images, text, titles, sizes, positions, flags, etc.

Quintus ships QUI with a default resource file that defines the "standard" QUI look. This file is located in the directory '`quintus-directory/qui3.5`'. This file *must* exist for correct operation of QUI.

We only document some of the resources that QUI uses. The X resource database provides no mechanism for distinguishing between those resources that can be customized and those that can not. Some QUI resources can be customized while modifying others will break QUI. Therefore, if you modify any QUI resource you run the risk of breaking QUI. We suggest that you only change QUI resources for minor enhancements. We reserve the right to change the format and structure of the resource file in future releases.

If you still want to change resources, do *not* modify the default QUI resource file itself. Rather, create the file '`$HOME/Qui3.5`', and add the QUI resources you want to modify to it. Alternatively, you can add QUI resources to your existing X defaults file '`$HOME/.Xdefaults`'.

Adding resources to either of these files will override the resources specified in the default QUI resource file.

See your Motif, Xt and Xlib documentation for more information on the X resource database format and use.

3.7.2 Customizing QUI Resources

The easiest way to find the name of the resource you want to change is to look in the default QUI resource file. Once you know the full name of the resource, you can put a line with the same name and a new value in your personal QUI defaults file in your home directory. Note that case is significant in naming a resource and comment lines start with a '`!`' character.

This section offers a few guidelines on how to identify the QUI resources you may wish to customize.

3.7.2.1 Global Resources

`qui.loadOnlyAtTopLevel:` **True**
: If set to **False**, file loading by QUI menus will be disabled only while Prolog code is running.

`qui*main*dialogTitle:` **Quintus User Interface**
: The main title displayed by QUI on its startup window.

`qui*background:` **wheat**
: The background color used in QUI windows.

`qui*foreground:` **black**
: The foreground color used in QUI windows.

`qui*fontList:` **fixed**
: The font used in most QUI windows, except for the help system [1].

`qui*qui_window.iconName:` **Qui**
: The icon name for QUI

3.7.2.2 Labels and Messages

The text that QUI displays in all of its buttons, menus and dialogs can be customized by changing the appropriate resources.

To change a button or menu label, look for the `labelString` resources. For example:

`qui*main*commands*interrupt.labelString:` **Interrupt**
: Sets the text displayed by the **Interrupt** button.

[1] See Section 3.7.2.7 [qui-ciq-cqr-hsr], page 76 if you want to change the font used by the QUI help system

`qui*main*menuBar*load.labelString:` **Load...**
: Sets the text displayed by the **Load** entry in the **File** menu of the main QUI Window [2].

`qui*qui_window*yesNoProceed*okLabelString:` **Yes**
: Sets the text displayed by the **OK** button in the **Yes/No** confirmation popup dialog.

To change the text of a message, look for the `messageString` resources. For example:

`qui*main*exitWarning*messageString:` **Do you really want to exit Prolog?**
: Sets the message displayed by the **Exit Warning** popup dialog.

3.7.2.3 Menu Entries

To customize a QUI menu entry there are potentially four resources that have to be changed. These are:

- The `labelString` resource that defines the text shown.
- The `mnemonic` resource that defines the single character abbreviation and underlining for the menu entry.
- The `accelerator` resource that binds a key combination to the entry.
- The `acceleratorText` resource that sets the displayed characters to show the accelerator.

For example, the **Frame Up** entry in the debugger **Travel** menu is defined by the following resources

`qui*debugger*frame_up.labelString:` **Frame Up**
: Sets the text of the **Frame Up** menu entry in the QUI debugger

`qui*debugger*travelmenu*frame_up.mnemonic:` **U**
: Sets the single character 'U' as the mnemonic for the **Frame Up** menu entry in the QUI debugger. This character will be shown underlined in the menu.

`qui*debugger*travelmenu*frame_up.accelerator:` **Ctrl<Key>U**
: Binds the `^U` character to the **Frame Up** menu entry action in the QUI debugger. Typing a `^U` character will be equivalent to use the mouse to select the **Frame Up** menu entry.

`qui*debugger*travelmenu*frame_up.acceleratorText:` **Ctrl + U**
: Sets the text to be displayed next to the **Frame Up** menu entry to show the user the accelerator keys for the command.

[2] But see Section 3.7.2.3 [qui-ciq-cqr-men], page 74 before changing QUI menus

3.7.2.4 Key Bindings

All of the QUI text widgets have a set of default key bindings to perform certain operations. These are all `translations` resources.

The value of a translation resource is a set of lines with an escaped new line character at the end of each line, except the last one. Each line associates a mouse/keyboard event with an internal action. For example:

```
qui*prolog*translations: #override\n\
        Ctrl<Key>C:  qpinterrupt() \n\
        Ctrl<Key>D:  qpeof() delete-next-character()\n\
        Ctrl<Key>U:  kill-to-prompt()\n\
        <Key>Delete: delete-previous-character()\n\
        Ctrl<Key>K:  kill-to-end-of-line()\n\
        Ctrl<Key>W:  kill-previous-word()\n\
        Ctrl<Key>P:  previous-line()\n\
        Ctrl<Key>N:  next-line()\n\
        Ctrl<Key>A:  beginning-of-line()\n\
        Ctrl<Key>E:  end-of-line()\n\
        Ctrl<Key>B:  backward-character()\n\
        Ctrl<Key>F:  forward-character()
```

The above sets the default key bindings for the QUI prolog query interpreter window.

3.7.2.5 Editor Resources

All the QUI editor resources have names that starts with `qui*editor`. Resources specific to the QUI editor include

`qui*editor.generateBackup:` **True**
: To prevent the QUI editor from generating backup files set the value to **False**.

`qui*editor.backupSuffix:` **.Bak**
: Sets the suffix that the QUI editor uses to generate backup files.

Many of the editor text messages have a resource name that ends in '`Msg`', for example

`qui*editor.qofFileMsg:` **File is a Quintus Object Format file**
: Sets the message to be displayed when attempting to load a QOF file into the editor.

3.7.2.6 Debugger Resources

All the QUI debugger resources have names that start with `qui*debugger`. Resources specific to the QUI debugger include

`qui*debugger*debugger_main_window.title:` **Quintus Debugger**
: The title at the top of the QUI debugger window.

`qui*debugger*debugger_main_window.geometry:` **583x600**
: Sets the default window size for the QUI debugger.

3.7.2.7 Help System Resources

All the QUI help system resources have names that start with `qui*help_window`. Resources specific to the QUI help system include

`qui*help_window*helpSystem*fontList:`**-*-courier-bold-r-normal–14-***
: Sets the default font for the QUI help system windows.

`qui*help_window*helpSystem*maxHistory:` **10**
: Sets the maximum number of items to be kept in the history pulldown menu of the QUI help window.

`qui*help_window*helpSearDialog.wrapAround:` **False**
: Sets the default value of **Wraparound** in the Search dialog to **on**.

3.7.3 Restrictions on developing programs under QUI

3.7.3.1 Hook Predicates

`message_hook/3`: If this predicate is defined in your program, it must be defined as a multifile predicate.

Also, there are restrictions on how you can use `message_hook/3` under QUI. In particular, error messages (terms with severity `error`) may not be seen by your `message_hook/3` clauses. QUI catches these messages and displays an error dialog.

Since it may be unpredictable whether user-supplied clauses for `message_hook/3` come before or after QUI's `message_hook/3` clauses, it is also recommended that any `message_hook/3` clauses you do supply should fail. See Section 8.20.3.3 [ref-msg-umf-ipm], page 331, as well as the reference page for `message_hook/3`, for more information.

3.7.3.2 Embeddable C Function

`QU_initio()`: QUI has already defined this Prolog embedding I/O initialization function. Users' programs linked with QUI cannot redefine the function.

3.7.3.3 UNIX Signal Handling

UNIX `SIGIO` signal: This signal is used in QUI. No programs developed under QUI should catch this signal.

UNIX `SIGPIPE` signal: The signal handler of this signal is set to `SIG_IGN` in QUI. Resetting the signal handler to `SIG_DEL` may cause QUI to exit unexpectedly.

4 The Emacs Interface

4.1 Overview

4.1.1 Overview

This section describes the Emacs/Quintus Prolog interface and presupposes some knowledge of the Emacs editor. The interface supports GNU Emacs and XEmacs. For information on obtaining these editors, see `http://www.gnu.org` and `http://www.xemacs.org`.

There are three different ways to run Prolog interfaced to Emacs:

1. From the command prompt, invoke a Prolog executable file or saved state with the argument '`+`'. This starts up Emacs and causes it to run Prolog in an Emacs buffer. See Section 4.1.3.3 [ema-ove-upe-epe], page 80 for details.
2. From QUI start an Emacs session. In this case the Prolog top-level interaction is not done in an Emacs buffer because the QUI main window is being used for that purpose. This uses the Emacs Server feature of GNU Emacs and is described in Section 3.4.1 [qui-ied-ige], page 65.
3. From a running Emacs session, cause a Prolog to be started in a buffer. To do this, the Emacs interface code must first be loaded into Emacs—this could be done in an Emacs initialization file. This is described in '*quintus-directory*`/editor3.5/gnu/README`'.

4.1.2 Environment Variables

This section lists the environment variables that can be used to customize your Prolog/Emacs environment. There are up to three environment variables that need to be set before either the Emacs interface can be invoked. They are:

`QUINTUS_EDITOR_PATH`
: the name of the Emacs executable. If this is unset Quintus Prolog will try to invoke `emacs`, which then must be in your path.

`QUINTUS_LISP_PATH`
: the full name of the Emacs-Lisp directories. These directories contain the Lisp code for the GNU Emacs interface, which is supplied with your Quintus Prolog distribution. If using *`prolog +`* or QUI, the `QUINTUS_LISP_PATH` environment variable need be set only if you choose to use a different version of the interface or if the interface has been moved to a different location at your site.

`QUINTUS_PROLOG_PATH`
: the full name of the Quintus Prolog executable. You need to set this variable only when you want to start Quintus Prolog from within GNU Emacs.

It is set automatically if you load qp-setup.el into emacs.

We recommend that you set these environment variables in your shell initialization file ('`.cshrc`' if you use the C shell `csh(1)`).

4.1.3 Using Prolog with the Emacs Editor

4.1.3.1 Overview

The Emacs/Prolog interface is designed to enable you to create a Prolog program in a file outside the Prolog environment and then to move back and forth easily between that file and the Prolog environment. Both the Prolog program and your interaction with Prolog are preserved in edit buffers, which can easily be reviewed and modified. The Emacs process is primary, and Prolog runs as a buffer within it.

In the Prolog window, Prolog programs can be run, and Emacs commands can be used to edit and resubmit previously-entered Prolog commands. In a text window, single procedures, groups of procedures, and entire programs can be edited and quickly reloaded into Prolog without suspending the Prolog process. Additionally, any number of Prolog source files can be loaded into Prolog at once; then, if desired, Emacs can be used to locate a specific procedure in any one of those files. (For more information, see Section 4.1.9 [ema-ove-loc], page 87.)

4.1.3.2 Terminal and Operating System Requirements

Under UNIX, if the `DISPLAY` environment variable is set then GNU emacs will create a window to run in. Otherwise it will run as if on a terminal, in which case it will need to be told what type of terminal it is. This is normally done automatically, at login time, but if it is not you must describe the terminal in a terminal capability database such as `terminfo` or `TERMCAP`. Refer to your UNIX documentation for a description of this facility.

4.1.3.3 Entering Prolog and Emacs

To run Prolog under the Emacs interface, type a command such as

```
% prolog +
```

```
% prolog + command-line-arguments
```

at the operating system prompt. GNU Emacs processes command line arguments in two lots, described in two tables in the *GNU Emacs Manual*. Under the Quintus Prolog GNU Emacs interface, only switches from the first table can be used, and the most commonly

used one is *file-to-be-edited*. (See Section 8.3 [ref-pro], page 186 for full details of starting up Prolog.) Note however that the prolog buffer will not be displayed if the *command-line-arguments* includes files to be edited. In this case the last file specified on the command line is the one displayed. You can however switch to the prolog buffer by invoking the key binding to switch buffers (usually `^x b`) and specifying the prolog buffer name '`*prolog*`'.

Another way to start up the interface is from a QUI menu. See Section 3.4.1 [qui-ied-ige], page 65 for how to do this.

A third alternative is to start up Quintus Prolog from within GNU Emacs by typing ⟨ESC⟩ `x run-prolog` (see Section 4.1.2 [ema-ove-eva], page 79 for a description of how to set the environment variable `QUINTUS_PROLOG_PATH`, which should be set to the filename of a Prolog executable before you invoke this command. In addition you must specify the directories where the Emacs lisp files in the interface live and load them. Refer to '`quintus-directory /editor3.5/gnu/README`' for details.) This should load in a specific set of '`.el`' or '`.elc`' files. These '`.elc`' files are part of the '`editor`' subdirectory of the Quintus distribution. (Section 1.3 [int-dir], page 11 explains the structure of the Quintus directory.)

4.1.3.4 Exiting Emacs

You can exit from the editor in either of two ways: you can stop the current editor job and exit irreversibly, or you can temporarily suspend the current editor job. If you are finished with your session, you will probably want to exit irreversibly, as described in this section. If you want to temporarily halt your session, return to the command prompt, and later be able to resume your session, you should exit as described in the next section.

To exit from the editor irreversibly, type the following:

```
^x ^c
```

If you try to exit while you have a Prolog session running, the system displays the following message at the bottom of the screen:

```
Active processes exist; kill them and exit anyway? (yes or no)
```

To end the Prolog session, type `y` or `yes`, and press ⟨RET⟩. If you don't want to end the Prolog session, type `n` or `no` and press ⟨RET⟩ to abort the exit. Prolog will continue running.

If you try to exit and you have files that have been modified but not saved, you will receive a message at the bottom of the screen to prompt you to save each modified file.

If you want to save the files before you exit, type `n` and press ⟨RET⟩ to abort the exit. Move the cursor to the text window (if it's not already there), and use the `^x ^s` command to save the information from the window into a file. If you want to exit without saving the modified information, type `y` and press ⟨RET⟩, and you will be returned to the main operating system prompt.

4.1.3.5 Suspending an Emacs Session

To exit from the editor by suspending the current editor session, type:

```
^x ^z
```

The system displays the message '`Stopped`' at the bottom of the screen and returns you to the command prompt. However, the Emacs/Prolog job is only suspended (that is, in the background) and you can resume it at any time. To resume your Emacs/Prolog session, type *`fg`* (for "foreground") at the command prompt.

4.1.4 The Source Linked Debugger

The Emacs-based source linked debugger for Quintus Prolog works very much like the QUI debugger (see Section 6.2 [dbg-sld], page 121), with a few significant differences. This document describes the differences.

In order to enable the Emacs-based debugger, execute the Emacs command ⟨ESC⟩ *`x enable-prolog-source-debugger`*; to disable it, type ⟨ESC⟩ *`x disable-prolog-source-debugger`*. If you would like always to use the source-linked debugger when debugging Quintus Prolog code under Emacs, put the following in your '`.emacs`' file:

```
(add-hook 'comint-prolog-hook 'enable-prolog-source-debugger)
```

Alternatively, under Prolog you may load `library(emacsdebug)` and then execute the Prolog goal `emacs_debugger(_,on)` to enable source-linked debugging, `emacs_debugger(_,off)` to disable it, and `emacs_debugger(`*`State`*`,`*`State`*`)`, to see whether it is enabled or not (*State* will be bound to `on` if enabled and `off` if disabled).

The first obvious difference when running the Emacs-based debugger compared to the QUI one is that it doesn't have any buttons or menu to control it. Therefore all commands are keyboard-based. Where possible, the commands are the same as those used in the standard debugger, so most of them should be easy to remember. The most important command in the Emacs-based debugger, as in the standard debugger is the **help** command, invoked by a single *`h`* or `?` character. This command displays the following summary:

```
c  creep          ⟨RET⟩ creep            ⟨SPC⟩ creep
l  leap            +  spy goal/pred      b  break
s  skip            -  nospy goal/pred    a  abort
z  zip             [  frame up           ?  help
n  nonstop         ]  frame down         h  help
q  quasi-skip      |  frame back         =  debugging
r  retry           f  fail               .  edit definition
w  open extra window                     x  close extra window
```

The commands in the first column behave exactly as they do in the QUI debugger. The **spy** and **nospy** commands place a spypoint on the current predicate when at a head port, and on the current goal when at any other port. The **frame up/down/back** commands do exactly what the corresponding QUI debugger commands do, as do **break** and **abort**. The **debugging** command just invokes the standard `debugging/0` built-in predicate, showing the current debugging and leashing modes, as well as listing the currently active spypoints.

The **open/close extra window** commands prompt for a single character to select the "extra" window to display, offering the choices '`b=bindings; s=standard; and a=ancestors`'. The bindings window is probably the most useful of the three.

Finally, the **edit definition** command opens puts the file being debugged in an editor buffer, putting point at the location of the current debugger port (where the arrow is). You may edit and save the file, and then recompile it. It is recommeded that you recompile the whole file rather than just the part you have changed, because the debugger keeps track of the times files are written and compiled, disabling source linking when the file on disk is newer than the code loaded into Prolog.

The graphical arrows of the QUI debugger are simulated by a two-character sequence in the Emacs-based debugger. The Call, Done and determinate Head ports are signified by '`->`'. Exit and nondeterminate Head ports are signified by '`=>`'. Redo and Fail are shown as '`<-`'. The Exception port is indicated by '`<#`'. Finally, where the QUI debugger shows a "hollow" arrow to signify that the currently shown port is not actually the active port but an ancestor of it, the Emacs-based debugger shows '`^>`'.

The Emacs-based debugger currently offers no way to change the leashing; you can do that using the usual Prolog `leash/1` built-in predicate. Similarly, it offers no way to set a spyoint except when debugging a call to the predicate or goal to be spied. Again, the usual `spy/1` and `add_spypoint/1` built-in predicates can accomplish this. Finally, the Emacs-based debugger offers no direct way to set the print format. To change this you must use the `window_format/3` command exported from the `emacs_debug` module:

```
window_format(+Window, -Oldformat, +Newformat)
```

where *Window* is one of: `source`, `bindings`, `ancestors`, or `standard`, and *Newformat* is a list of valid options for the last argument to `write_term/[2,3]`. The default format for all windows is

```
[quoted(true), portrayed(true), max_depth(5)]
```

4.1.5 Accessing the On-line Manual

The help system is largely based on the Info file format, which GNU Emacs uses for on-line documentation. If you are running Prolog under Emacs, type *`manual.`* at the main Prolog prompt to gain access to the on-line help system. Emacs should then locate the Quintus Prolog Info node. If it can't find that node, it will display the information in a special help buffer. See Section 8.17.2 [ref-olh-hfi], page 304 for details.

4.1.6 Loading Programs

4.1.6.1 Basic Information

To load a program from Emacs into Prolog, start up the Emacs/Prolog interface by typing *prolog +* at the main operating system prompt. When the Emacs/Prolog screen appears, type *^x ^f* (for '`find-file`') followed by the name of the file that contains your program. (Alternatively, you can type *prolog +* followed by the name of the file that contains your program.) After you enter the Emacs/Prolog environment, activate the window containing your file.

At this point, you have three options: you can load

- the entire buffer containing your file (⟨ESC⟩ *k b*)
- a portion of the program that you have marked (⟨ESC⟩ *k r*)
- or a single procedure (⟨ESC⟩ *k p*)

Being able to load a designated portion of your program is very convenient if you are running a program and discover that you need to make a few changes to improve the program. You can make your changes and then reload just the changed portions, without reloading the entire program. If you are just beginning a Prolog session, however, you will probably want to load the entire buffer containing your program using ⟨ESC⟩ *k b*.

When you type ⟨ESC⟩ *k*, Emacs displays the following prompt line at the bottom of the screen:

```
compile prolog ... enter p for procedure, r for region or b for buffer
```

In response, you can type *b* to load the entire buffer: or you can type *r* to load a region, or *p* to load a procedure, as described below.

After you indicate how much of your program to load, the cursor moves to the Prolog window, and Prolog displays a message that tells you it is loading the program. As it proceeds, Prolog displays messages to let you know which procedures are being loaded. For example, if your program consisted of procedures for `parts_of/2`, `assembly/2`, and `inventory/2`, Prolog would display the messages:

```
% compiling procedure parts_of/2 in module user
% compiling procedure assembly/2 in module user
% compiling procedure inventory/2 in module user
```

After Prolog has finished loading, it displays a message such as:

```
% compilation completed, 0.083 sec 448 bytes
| ?-
```

At this point, you can begin to run your program (see Section 2.3 [bas-run], page 27).

Please notes:

1. When you load procedures into Prolog, Prolog first removes any previous versions of those procedures from its database, excepting those procedures that have been declared multifile. A multifile declaration indicates that a particular procedure is defined by clauses in more than one file. (For more information on multifile procedures, see Section 9.2.2 [sap-rge-dspn], page 356.) This is why the entirety of a procedure must be loaded at once; otherwise, loading the definition of the second part would wipe out the definition of the first part.
2. The ⟨ESC⟩ *k* commands can also be issued from the debugger prompt, as described in Section 6.1.1 [dbg-bas-bas], page 113 .

4.1.6.2 Loading an Entire Buffer

To load the entire buffer, type ⟨ESC⟩ *k b* (for '`buffer`'). Note that both the *k* and the *b* must be lower case.

4.1.6.3 Loading a Region in a Buffer

To load a portion of your program, mark the region you want to load. To do this, move the cursor to the beginning of the first line you want to load and type `^`⟨SPC⟩ (or `^@` on many terminals). Emacs displays the message

```
Mark set
```

at the bottom of the screen. Move the cursor to the end of the portion of the program you want to load. Then type ⟨ESC⟩ *k r* (for '`region`').

> **Please note:** When you mark the region to be loaded by the ⟨ESC⟩ *k r* command, be sure to include *all* the clauses for any procedures you are loading.

4.1.6.4 Loading a Single Procedure

To load a single procedure, move the cursor to any portion of any line of the procedure. Then type ⟨ESC⟩ *k p* (for '`procedure`').

The ⟨ESC⟩ *k p* facility requires that you use certain syntactic and structural conventions, which are described in Section 4.2.4 [ema-emi-lay], page 91. If you are not sure that your procedures adhere to these conventions, you should use the ⟨ESC⟩ *k r* facility instead, making sure your marked region surrounds all the clauses of the procedures you want to load.

> **Please note:** ⟨ESC⟩ *i* is retained for backward compatibility. Its effect is the same as ⟨ESC⟩ *k*.

4.1.7 Repeating a Query

Often during your Prolog sessions you might find it useful to submit a query, edit it slightly, then resubmit it. For example, if you make a typing error in a query, you generally want to correct the error and resubmit the query, instead of retyping the entire query.

Everything you type during a Prolog session goes into an Emacs buffer, so it is easy to retrieve and copy lines you've already typed. Prolog then redisplays the query.

4.1.7.1 Repeating Queries under Gnu Emacs

For example, suppose you made a typing error, as shown below. If you typed `^x ^e`, Prolog would duplicate the line. You could then correct the typing error using Emacs editing commands and resubmit the query.

```
| ?- parts_of(transmission, X).
no
| ?- ^x^e
```

When you type `^x ^e`, the last input string you typed is displayed in the mini-buffer:

```
 | ?- parts_of(transmission, X).
```

At this point you have two options. You can edit the query in the mini-buffer and place it in the prolog window by hitting ⟨RET⟩; hitting ⟨RET⟩ again submits the query to prolog. Alternatively, you can step through the queries submitted to Prolog by typing ⟨ESC⟩ `p` (⟨ESC⟩ `n` moves you down this list). [1]

Under GNU Emacs it is also possible to search through the goal history for a goal matching a regular expression. To do so, type `^x ^y` and you will be prompted for a regular expression. On entering a regular expression and hitting ⟨RET⟩, the most recently submitted goal matching the regular expression will be displayed in the mini-buffer. You can choose to submit this query to prolog (after editing it in the mini-buffer if you choose to) or locate the next most recent goal matching the pattern.

Alternatively, you can redisplay an earlier input string by moving the cursor to the line you want to copy and then type `^x ^e`.

4.1.8 Displaying Previous Input

If you are running Prolog under the Emacs interface, you can scroll backward through your Prolog session to see previous input by using the Emacs scrolling commands. Similarly, you can use Emacs scrolling commands to scroll forward to the current step in your Prolog session.

[1] You can move around a multi-line query using the `^n` and `^p` keys.

4.1.9 Locating Procedures

The Emacs/Prolog interface provides a facility that enables you to quickly locate procedures in source files once the procedures have been loaded. If you have loaded several files into Prolog at once, it can be helpful to be able to locate a procedure directly without having to search through several files.

While inspecting the definition of `foo/1`,

```
foo(X) :- bar(X).
```

in some Prolog source window, you may want to see the definition of the predicate `bar/1`. To do so put the cursor anywhere on the name '`bar`' and type ⟨ESC⟩ `.` and respond to the Emacs message line,

```
Find: (default bar/1)
```

by pressing ⟨RET⟩. Prolog will then visit the file that defined `bar/1` (this may be the same file that defined `foo/1`) and put the cursor on the first clause of `bar/1` in that file.

While inspecting a previously submitted Prolog query,

```
| ?- bar(X).
```

in the Prolog execution window, you may want to see the definition of the predicate `bar/1`. This is done exactly in the same manner as above.

Alternatively, you can type ⟨ESC⟩ `.` at the main Prolog prompt. The cursor moves to the bottom of the screen, and the system displays the message

```
Find:
```

Type the name of the predicate whose procedure you want to locate followed by a slash and the arity of the predicate; then press ⟨RET⟩. (Recall that the arity is the number of arguments the predicate has.) For example, to locate the procedure for `employee(smith, harold)`, you would type *`employee/2`*, as shown below.

```
Find: employee/2 ⟨RET⟩
```

> **Please note:** You can type the predicate name without typing the arity, and the system will still locate the predicate. If the predicate is defined for more than one arity, the system will simply locate one of the definitions of the predicate. You can then type ⟨ESC⟩ `,` to successively locate the other definition(s).
>
> This will also search other files for additional clauses for a multifile predicate, or will search for a predicate of the same name and arity in a different module.

If Prolog cannot find a procedure of the specified name and arity, it displays a message telling you the procedure is undefined:

```
foo/2 is undefined
```

If the specified predicate is a built-in predicate, Prolog displays a message to that effect:

```
nl/0 is a built-in predicate
```

4.2 The GNU Emacs Interface

4.2.1 Overview

This section presupposes some knowledge of the GNU Emacs editor. The next two sections summarize the features that have been added to GNU Emacs specifically to support Quintus Prolog. It should be noted that these features are not available with the standard GNU Emacs distribution. The Quintus Prolog distribution contains Emacs-Lisp code that constitutes the interface. Users are free to modify this interface in any way, provided they adhere to the copying policies of the Free Software Foundation. Read the file '`COPYING`' in the GNU Emacs distribution for further details.

With GNU Emacs, you can talk to Prolog very much as you would without the GNU Emacs interface. The only difference with GNU Emacs is that control characters issued to Prolog generally have their GNU Emacs meaning rather than any meaning they might have outside of GNU Emacs. The reason for this is that the Prolog window is still an edit buffer, and you are free to move up and down in it and modify its contents using the full range of editing commands. Thus `^d` deletes a character, `^u` may be used to specify an argument for the next command, and so on.

The general philosophy of the Prolog/GNU Emacs interface is that you should not be able to lose your Prolog prompt(by deleting a line, for example). For this reason, a few commands have been slightly modified. There are also a number of additional key bindings, which are described below.

GNU Emacs is a customizable editor. You can use a language called Emacs-Lisp to extend or alter the way it behaves, and in fact this is the way that the Prolog/GNU Emacs interface has been built. If you want to make your own extensions, you may need to know something about the way this interface works; notes to assist you are provided later in this section.

> WARNING: The Prolog/GNU Emacs interface uses the control character '`^]`' for its communication. If this character has been made special by the UNIX command `stty(1)`, the Prolog/GNU Emacs interface will not work. If there is a problem with the interface, you can use *`stty all`* to see all the special character settings and see if '`^]`' is shown. None of this is a problem under Windows or when using GNU Emacs via X Windows under UNIX.

4.2.2 Key Bindings

This section describes the key bindings associated with the Prolog/GNU Emacs interface. For a complete listing of all the key bindings applicable in a particular window, type ⟨ESC⟩ *x describe-bindings* or *^h b*.

The following key bindings apply only in the Prolog window, not in the text window(s):

^c ^d Sends an end-of-file to Prolog. This can be used to exit from a break level or to exit from Prolog altogether (see Section 8.11.1 [ref-iex-int], page 250 for more information on `break/0`). Having exited from Prolog using this command, the only way of start up a new Prolog is by typing ⟨ESC⟩ *x run-prolog*.

^x ^z Suspends Prolog and GNU Emacs.

^x ^e Allows you to edit a query you previously typed to the Prolog prompt and resubmit it. Effectively, it grabs the last query and brings it down to the mini-buffer. There you can edit it if necessary, then move your cursor to the last line of the query and type ⟨RET⟩. This places the query in the prolog window, where you can edit it further, if necessary, and type ⟨RET⟩ to submit the query to Prolog. You can also grab queries other than the most recent one by specifying a prefix argument to this command (using ⟨ESC⟩, or *^u*): 2 to get the second last, 3 to get the third last, and so on. Another way to do this is to move the cursor back to the query you want to copy and type *^x ^e*. This last alternative does not place the query in the mini-buffer but places it directly in the prolog window. In the cases where a query is placed in the mini-buffer, you can step up a list of previously executed queries by typing in ⟨ESC⟩ *p* or down the list by typing in ⟨ESC⟩ *n*. Since the mini-buffer is only one line, a multi-line query can be stepped around by using the conventional *^p*, *^n* key strokes. To obtain the set of bindings that are active within the mini-buffer when you execute the yank-query key sequence type in *^h b*. This method of stepping through a goal history is similar to GNU Emacs method of stepping through a command history.

^x ^y Allows you to edit the most recent query matching a regular expression. You are first prompted for a regular expression; on entering a regular expression and hitting ⟨RET⟩, the most recently submitted goal matching the regular expression is displayed in the mini-buffer. You can choose to submit this query to prolog (after editing it in the minibuffer, if necessary) or locate the next most recent goal matching the given regular expression.

^c ^c Sends an interrupt to the Prolog process (exactly as if you were not running under GNU Emacs).

Please note: to send a numeric argument to a GNU Emacs command, type ⟨ESC⟩ followed by the desired number (for example, ⟨ESC⟩ *1* or ⟨ESC⟩ *12*); then type the command. *^u* also works as an argument prefix, as in "standard" GNU Emacs.

The following key bindings apply in any window:

`^x ^c` Causes an irreversible exit from GNU Emacs and Prolog. You will be prompted to make sure that (1) the Prolog and all other subprocesses should indeed be killed, and (2) any unsaved buffers should indeed be discarded.

`^x ^z` If you are running GNU Emacs from a UNIX terminal window, this suspends (pauses) the GNU Emacs process and returns you to the operating system prompt. If a Prolog program is running, it will continue to run, but you will not see any output from it. You can get your GNU Emacs/Prolog session back by typing *`fg`*. (This pause facility is only available if you are running `csh`; it does not work under `sh`.)

⟨ESC⟩ `.` Finds the source code for a particular procedure. If the cursor is positioned on or before the predicate name part of a goal, you can simply press ⟨RET⟩ to find the clauses for its procedure. Otherwise, in response to the prompt '`Find:` ', you should type the name of the predicate, optionally followed by a '`/`' and its arity. The file containing the procedure for the specified predicate is then visited, and the cursor is positioned at the beginning of the procedure. There are some layout conventions, which must be followed for this facility to work: see Section 4.2.4 [ema-emi-lay], page 91. The facility is also available via the `.` debugger option (see Chapter 6 [dbg], page 113). In QUI, find-definition must be invoked within the GNU Emacs process, not the QUI main window.

⟨ESC⟩ `,` This command can only be used after ⟨ESC⟩ `.`. It successively locates other procedure definition(s) for a predicate. ⟨ESC⟩ `,` will search other files for additional clauses for a multifile predicate, will search for a predicate of the same name and arity in a different module, or will search for predicates with the same name and different arities (in the case where the arity was not specified).

⟨ESC⟩ `x` *`prolog-mode`*
Changes the current buffer to Prolog mode. See Section 4.2.3 [ema-emi-mod], page 91.

⟨ESC⟩ `x` *`library`*
Prompts for a file and locates it in the Quintus Prolog Library directories.

⟨ESC⟩ `x` *`cd`* Prompts for a directory and changes the directory of the Prolog window and of the Prolog process.

⟨ESC⟩ `x` *`enable-prolog-source-debugger`*
Enables the source linked debugger.

⟨ESC⟩ `x` *`disable-prolog-source-debugger`*
Disables the source linked debugger.

The following key bindings apply in any edit window except the Prolog window:

⟨ESC⟩ *`k`* (for "kompile") is used to load procedures from the edit buffer. You are then prompted to choose one of three options; you can compile

1. the procedure in which the cursor is currently positioned (see Section 4.2.4 [ema-emi-lay], page 91, for restrictions on program layout necessary for this to work);

2. the region between the cursor and the mark; or
3. the whole buffer.

⟨ESC⟩ *i* (for interpret) is synonymous to ⟨ESC⟩ *k*; and is there for backward compatibility.

4.2.3 Prolog Mode

Prolog mode applies automatically whenever you are editing a file that ends with the characters '`.pl`'. This mode is useful when you are editing Prolog source code. In Prolog mode:

- Whenever you type a closing parenthesis or bracket, the corresponding opening one is flashed. This bracket matching attempts to be clever about strings in quotes, because normally a bracket written within quotes should not count for matching purposes. Unfortunately, this means that the bracket matching does not work properly when radix notation (for example, 16'100 is hexadecimal 100, or 256 decimal) is used.
- The definition of ⟨LFD⟩ is modified. Immediately after a line with no leading space characters (normally the head of a clause), a ⟨LFD⟩ is equivalent to a ⟨RET⟩ followed by 8 spaces. (This can be overridden by assigning the variable body-predicate-indent some other value.) Otherwise it is equivalent to a ⟨RET⟩ followed by enough tabs and spaces to put the cursor underneath the first non-space character in the previous line. A different set of rules apply for indentation within if-then-else constructs. Refer to '*quintus-directory*`/editor3.5/gnu/README`' for details. Note that the ⟨TAB⟩ key also indents the current line as prolog code. It differs from ⟨LFD⟩ in that a newline is not generated.

4.2.4 Prolog Source Code Layout Restrictions

There are some restrictions on program layout, which are necessary for ⟨ESC⟩ *k p* (compiling a procedure), ⟨ESC⟩ `.` (find-definition) and for indentation in prolog mode to work properly. In order for these commands to function correctly, you must:

1. Group Prolog clauses of the same name and arity together. That is, do not intersperse clauses of one procedure with clauses of another. Normally breaching this restriction will cause a style warning (see Section 2.2.5 [bas-lod-sty], page 24).
2. Start the heads of all Prolog clauses at the beginning of the line; indent any additional lines for those clauses. Within prolog mode the TAB and the ⟨LFD⟩ can be used for indenting the current line as prolog code.
3. If a comment continues onto another line, indent the continuation line(s).
4. Do not write clause definitions that use operators in the heads of the clauses. For example, if you want to define clauses for `+/2`, then write the head of the clause in the form '`+(A, B)`' and not '`A + B`'.

4.2.5 Rebinding Keys in Your Initialization File

You can customize GNU Emacs by defining key bindings and/or Emacs-Lisp functions in a special GNU Emacs initialization file called '`.emacs`', which must be kept in your home directory. To locate it, do `^x ^f` and type '`~/.emacs`' as the filename. (Windows users: the '`~`' character is a abbreviation, inherited from UNIX, for your home directory.)

When GNU Emacs is started, it loads your GNU Emacs initialization file, if you have one, before loading the Emacs-Lisp files defining the editor interface. This means that any key bindings that you make in this file may be overridden by the editor interface package. However, you can tailor the interface, if you wish, by defining one or both of the following "hook functions":

`prolog-startup-hook`
: If a function by this name is defined, it will be called after all the initializations done on invoking Prolog through the editor interface are completed, and before the screen is displayed.

`prolog-mode-hook`
: If a function by this name is defined, it will be called every time Prolog mode is entered. Prolog mode is entered every time you edit a file with a '`.pl`' extension, and can also be entered by using the command ⟨ESC⟩ *x prolog-mode*.

For example, if you don't like incremental search, and you prefer to use ⟨ESC⟩ *e* for moving to the end of a sentence, rather than for enlarging the current window, then you should add the following function definition to your initialization file:

```
(defun
     prolog-startup-hook ()
        (global-set-key  "\C-s" 'search-forward)
        (global-set-key  "\C-r" 'search-reverse)
        (global-set-key  "\ee"  'forward-sentence)
        (global-set-key  "\e\e"
                         'enlarge-window))
```

Note that the command strings could be written as : `"\C-s"` (Backslash `C-s`) for example. This version of `prolog-startup-hook` also binds the useful `enlarge-window` command, which is normally on ⟨ESC⟩ *e* in this interface, to ⟨ESC⟩ ⟨ESC⟩.

If you wish to change key bindings in Prolog mode, you should use `local-set-key` rather than `global-set-key`, because the effect of Prolog mode is local to a particular window. For example, if you don't like the way ⟨LFD⟩ works in Prolog mode, you can define

```
(defun
     prolog-mode-hook ()
        (local-set-key  "\C-j" 'newline-and-indent))
```

This restores the default binding of ⟨LFD⟩.

4.2.6 Programming the Prolog/GNU Emacs Interface

This section describes how the user can program the Quintus Prolog/GNU Emacs interface. In order to make effective use of the interface, the user must be very familiar with GNU Emacs's extension language, Emacs-Lisp. We do not provide support for users' Emacs-Lisp code.

Before deciding to use Emacs-Lisp code in your applications, you should be aware of two potential problems. First, we does not support the GNU Emacs editor interface and as such reserves the right to change the editor environment at any time. This could render your non-Prolog code inoperative or irrelevant. Second, Runtime Systems, which is the usual way of packaging Prolog applications for end users, do not support the GNU Emacs environment. Thus they will not run your Emacs-Lisp code.

With these cautions in mind, if you decide to use Emacs-Lisp code in your applications, these notes should have all of the information that you need to proceed. If your application is very complex, you may also find that you need to understand in some detail how the GNU Emacs interface is built. We supply with the distribution the Emacs-Lisp source code for the interface. You are free to go ahead and change any part of the interface to suit your needs but please be careful to follow the guidelines set by the Free Software Foundation as to how this should be done. These guidelines may be found in any standard distribution of the GNU Emacs editor. You will find the files '`help.el`', '`commands.el`', and '`qprocess.el`' especially useful, although depending upon what you are trying to do you may need information from any of the Emacs-Lisp source files.

4.2.6.1 Submitting Prolog Queries from GNU Emacs

GNU Emacs talks to Prolog by writing an Emacs-Lisp string in the form of a Prolog query (of any kind), terminated by a full-stop and prefixed by a special character code, to Prolog's standard input stream. When Prolog detects one of these prefixed queries in its term input stream, it executes the query as if it had been typed by the user at the top level. The query prefix character is ASCII 29.

As an example, define the function

```
(defun to-prolog ()
    (process-send-string "prolog"
              "\035nl,write(hello),nl.\n")
)
```

Upon invoking this function from GNU Emacs, the results of the query are displayed in the Prolog execution buffer. Notice that the Prolog process is called `prolog`, the query prefix character (ASCII 29) is denoted in an Emacs-Lisp string by the *octal* escape sequence '`\035`', and the full-stop at the end of the query must be a period followed by a newline character (denoted in an Emacs-Lisp string by the escape sequence '`\n`').

To make things simpler, the interface defines a function called `send-prolog`, which, given a query string as its one argument, sends that query to the Prolog process, prefixed by the query prefix character and followed by a full-stop. So the above function can be more clearly written as

```
(defun to-prolog ()
    (send-prolog "nl,write(hello),nl")
)
```

WARNING: GNU Emacs should only send a prefixed query to Prolog when Prolog is waiting for the user to type a term at the terminal. Note that this occurs not only when the user explicitly calls `read/[1,2]` but also when Prolog is at the top-level prompt.

4.2.6.2 Invoking Emacs-Lisp Functions from Prolog

Prolog talks to GNU Emacs by writing a sequence of one or more Emacs-Lisp function calls (including the parentheses) to the standard output stream, where this sequence is delimited by two special character codes. When GNU Emacs detects one of these delimited "packets" (as they are referred to in the Emacs-Lisp code) being written, it executes the function calls that occur between the delimiters. The packet start character is ASCII 30 and the packet end character is ASCII 29.

As an example, define and call the predicate

```
to_emacs :-
        put(30),
        write('(message "hello") (sit-for 50)'),
        put(29).
```

WARNING: Attempting to debug or interrupt (with `^c ^c`) this predicate, thus submitting only a partial packet to GNU Emacs, will cause subsequent output to be considered as a continuation of the current packet and disaster will ensue. If such a situation occurs, try typing the command

```
put(29).
```

to terminate the packet. You may want to consider using a critical region to prevent this problem, see `library(critical)`.

You will notice that after the message ('`hello`') is printed out in the message buffer (also called the minibuffer in GNU Emacs literature) and the "sit-for" period expires, it then disappears. This is a side-effect of the design of the GNU Emacs interface. Any Emacs-Lisp function that is called by Prolog should display messages using the function

```
(&qp-message message-string)
```

where *message-string* must be a single string (this is unlike `message`, which can take multiple strings; use the Emacs-Lisp function `concat` to make a single string out of multiple strings). This string will be displayed in the message buffer after GNU Emacs has processed the current "packet" from Prolog. Therefore, if you redefine the predicate as

```
to_emacs :-
        put(30),
        write('(&qp-message "hello")'),
        put(29).
```

and reinvoke it, you will find that the message remains in the message buffer.

5 The Visual Basic Interface

Quintus Prolog provides an interface that lets you load and call Quintus Prolog programs from Visual Basic.

5.1 Overview

Quintus Prolog provides an easy-to-use one-directional Visual Basic interface that lets you load and call Quintus Prolog programs from Visual Basic but not the other way around. The idea is that Visual Basic is used for creating the user interface while the Prolog program works as a knowledge server in the background.

The control structure of this interface is rather similar to that of the foreign language interface. However, in contrary to that interface, there is currently no way of handling pointers to Prolog terms or queries. The queries to be passed to Prolog have to be given as strings on the Visual Basic side and the terms output by Prolog are received as strings or integers in Visual Basic variables.

The interface provides functions for:

- passing a query to Prolog
- evaluating the Prolog query
- retrieving a value (string or integer) assigned to a variable by the Prolog query
- getting information about the exceptions that have occurred in the Prolog query

5.2 How to Call Prolog from Visual Basic

5.2.1 Opening and Closing a Query

Prolog queries are represented in Visual Basic in textual form, i.e. as a string containing the query, but not followed by a full stop. For example, the following Visual Basic code fragments create valid Prolog queries:

```
'Q1 is a query finding the first "good" element of the list [1,2,3]
Q1 = "member(X,[1,2,3]), good(X)"

'create a Q2 query finding the first "good" element of the list
'[1,2,...,N]:
Q2 = "member(X,["
For i = 1 To N-1
   Q2 = Q2 & i & ","
Next
Q2 = Q2 & N & "]), good(X)"
```

Before executing a query, it has to be explicitly opened, via the `PrologOpenQuery` function, which will return a *query identifier* that can be used for successive retrieval of solutions.

The `PrologCloseQuery` procedure will close the query represented by a query identifier. The use of an invalid query identifier will result in undefined behavior. For example:

```
Dim qid As Long

Q1 = "member(X,[1,2,3]), good(X)"
qid = PrologOpenQuery(Q1)

   ... <execution of the query> ...

PrologCloseQuery(qid)
```

5.2.2 Finding the Solutions of a Query

Prolog queries can be executed with the help of the `PrologNextSolution` function: this retrieves a solution to the open query represented by the query identifier given as the parameter. Returns 1 on success, 0 on failure, -1 on error.

5.2.3 Retrieving Variable Values

After the successful return of `PrologNextSolution`, the values assigned to the variables of the query can be retrieved by specific functions of the interface. There are separate functions for retrieving the variable values in string, quoted string and integer formats.

The `PrologGetLong` function retrieves the integer value of a given variable within a query and assigns it to a variable. That is, the value of the given variable is converted to an integer. Returns 1 on success.

Example: The following code fragment assigns the value 2 to the variable `v`:

```
Dim qid As Long

Q = "member(X,[1,2,3]), X > 1"

qid = PrologOpenQuery(Q)
Call PrologNextSolution(qid)
Call PrologGetLong(qid,"X",v)
```

The `PrologGetString` function retrieves the value of a given variable in a query as a string. That is, the value of the variable is written out using the `write/2` Prolog predicate, and the resulting output is stored in a Visual Basic variable. Retuns 1 on success.

Example: suppose we have the following clause in a Prolog program:

```
capital_of('Sweden'-'Stockholm').
```

The code fragment below assigns the string `"Sweden-Stockholm"` to the variable `capital`:

```
Dim qid As Long

Q = "capital_of(Expr)"

qid = PrologOpenQuery(Q)
If PrologNextSolution(qid) = 1 Then
  Call PrologGetString(qid,"Expr",capital)
End if
Call PrologCloseQuery(qid)
```

The `PrologGetStringQuoted` function is the same as `PrologGetString`, but the conversion uses the `writeq/2` Prolog predicate. Returns 1 on success.

Example: if the function `PrologGetStringQuoted` is used in the code above instead of the `PrologGetString` function, then the value assigned to the variable `capital` is `"'Sweden'-'Stockholm'"`.

The only way of transferring information from Prolog to Visual Basic is by the above three `PrologGet...` functions. This means that, although arbitrary terms can be passed to Visual Basic, there is no support for the transfer of composite data such as lists or structures. We will show examples of how to overcome this limitation later in the manual (see Section 5.4 [vb-ex], page 101).

5.2.4 Evaluating a Query with Side-Effects

If you are only interested in the side-effects of a predicate you can execute it with the `PrologQueryCutFail` function call, which will find the first solution of the Prolog goal provided, cut away the rest of the solutions, and finally fail. This will reclaim the storage used by the call.

Example: this is how a Prolog file can be loaded into the Visual Basic program:

```
ret = PrologQueryCutFail("load_files(myfile)")
```

This code will return 1 if `myfile` was loaded successfully, and -1 otherwise (this may indicate, for example, the `existence_error` exception if the file does not exist).

5.2.5 Handling Exceptions in Visual Basic

If an exception has been raised during Prolog execution, the functions `PrologQueryCutFail` or `PrologNextSolution` return -1. To access the exception term, the procedure `PrologGetException` can be used. This procedure will deposit the exception term in string format into an output parameter, as if written via the `writeq/2` predicate.

Example: when the following code fragment is executed, the message box will display the `domain_error(_1268 is 1+a,2,expression,a)` error string.

```
Dim exc As String

qid = PrologOpenQuery("X is 1+a")
If PrologNextSolution(qid) < 0 Then
   PrologGetException(exc)
   Msg exc,48,"Error"
End if
```

5.3 How to Use the Interface

In this section we describe how to create a Visual Basic program that is to execute Prolog queries.

5.3.1 Setting Up the interface

The Visual Basic - Quintus Prolog interface consists of the following files:

- '`vbqp.dll`' (The Prolog code and Quintus runtime)
- '`vbqp.bas`' (The Visual Basic code)

In order to use the interface, perform the following steps:

- Include the file '`vbqp.bas`' in your Visual Basic project.
- Put '`vbqp.dll`' in a place where DLLs are searched for (for example the same directory as your applications EXE file or the Windows-System directory). Alternatively put it in '*`quintus-dir`*`\bin\ix86\`' and ensure that '*`quintus-dir`*`\bin\ix86\`' is in the PATH environment variable.

- Make the Quintus runtime DLL etc. available. Typically by running '`qpvars.bat`' or by putting '*`quintus-dir`*`\bin\ix86\`' in the `PATH` environment variable.

5.3.2 Initializing the Prolog engine

The Visual Basic interface must be explicitly initialized: you must call `PrologInit()` before calling any other interface function. The `PrologInit()` function loads and initializes the interface. It returns 1 if the initialization was successful, and -1 otherwise.

5.3.3 Deinitializing the Prolog Engine From VB

The Visual Basic interface should be deinitialized: you should call `PrologDeInit()` before exiting e.g. from a `Form_Unload` method.

5.3.4 Loading the Prolog code

Prolog code (source or QOF) can be loaded by submitting normal Prolog load predicates as queries. Note that Quintus uses slashes '`/`' in file names where Windows uses backslash '`\`'. Example:

```
PrologQueryCutFail("load_files('d:/xxx/myfile')")
```

To facilitate the location of Prolog files, two file search paths are predefined:

`app` identifies the directory path of the Visual Basic project or the applications executable.

That is, you can load the file `myfile` located in the same directory as the project/executable, issuing the query:

```
PrologQueryCutFail("load_files(app(myfile))")
```

`vbqp` identifies the directory path of the '`vbqp.dll`' file.

That is, you can use the following query to load the file `myfile` if it is located in the same directory as '`vbqp.dll`':

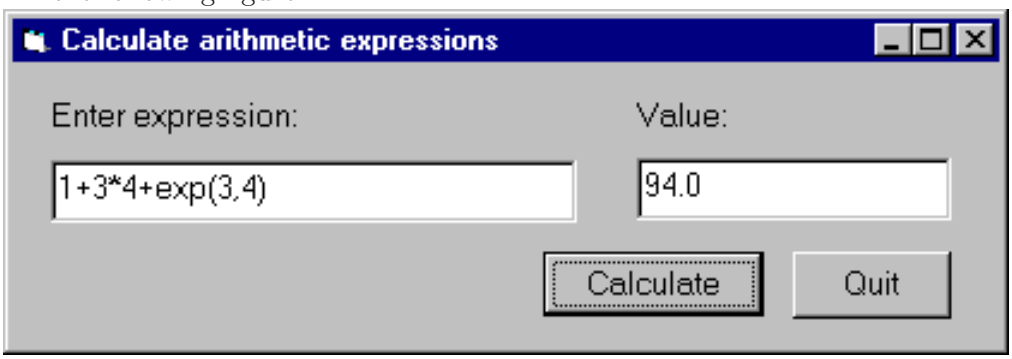
```
PrologQueryCutFail("load_files(vbqp(myfile))")
```

5.4 Examples

The code for the following examples are available in the directory '`examples`' in the '`VBQP`' directory.

5.4.1 Example 1 - Calculator

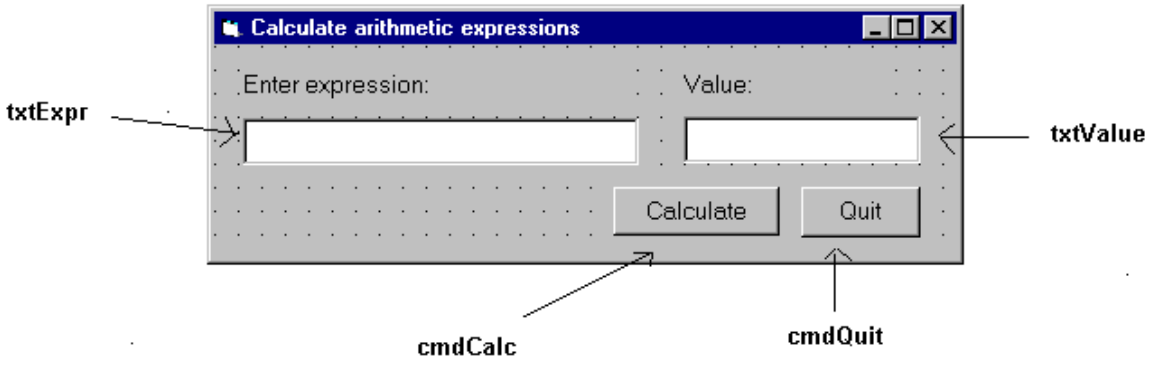
This example contains a simple program that allows you to enter an arithmetic expression (conforming to Prolog syntax) as a string and displays the value of the given expression, as shown in the following figure:



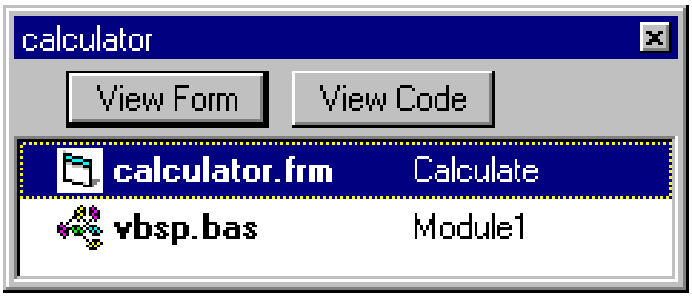
The calculation itself will be done in Prolog.

We now we will go through the steps of developing this program.

1. Start a new project called `calculator`.
2. Add the '`vbsp.bas`' file to the project.
3. Create a form window called `calculator`. Edit it, adding two textboxes `txtExpr` and `txtValue`, and two command buttons, `cmdCalc` and `cmdQuit`:



Save the form window to the '`calculator.frm`' file. Then the project will contain the following two files:



4. Write the Prolog code in the file '`calc.pl`', evaluating the given expression with the `is/2` predicate, and providing a minimal level of exception handling:

```
prolog_calculate(Expr, Value) :-
   on_exception(Exc, Value is Expr, handler(Exc,Value)).

handler(domain_error(_,_,_,_),'Incorrect expression').
handler(Exc,Exc).
```

   Note that this example focuses on a minimal implementation of the problem, more elaborate exception handling will be illustrated in the Train example (see Section 5.4.2 [vb-ex-tr], page 104).

   Compile this file, and deposit the file '`calc`' in the directory where the `calculator.vbp` project is contained.

5. Now you have to write the Visual Basic code in which Quintus Prolog will be called at two points:

   - Initialize Prolog in the `Form_Load` procedure executed when the `calc` form is loaded, calling the `PrologInit()` function and loading the '`calc`' file with the help of the `PrologQueryCutFail(..))` function:

```
Private Sub Form_Load()
    If PrologInit() <> 1 Then GoTo Err
    If PrologQueryCutFail("ensure_loaded(app(calc))") <> 1 Then GoTo Err
    Exit Sub

Err:
    MsgBox "Prolog initialization failed", 48, "Error"
    Unload Me
End Sub
```

   - Do the expression evaluation in the `calculate` procedure activated by the `cmdRun` command button. This procedure will execute the `prolog_calculate(X,Y)` procedure defined in the '`calc.pl`' Prolog file:

```
Public Function calculate(ByVal Expr As String) As String
    Dim qid As Long
    Dim result As String
    Dim ret As Long
    Dim Q As String

    Q = "prolog_calculate(" & Expr & ",Value)"
    qid = PrologOpenQuery(Q)
    If qid = -1 Then GoTo Err ' e.g. syntax error

    ret = PrologNextSolution(qid)
    If ret <> 1 Then GoTo Err ' failed or error

    ret = PrologGetString(qid, "Value", result)
    If ret <> 1 Then GoTo Err
    calculate = result
    Call PrologCloseQuery(qid)

    Exit Function

Err:
    MsgBox "Bad expression", 48, "Error!"
    calculate = ""
End Function
```

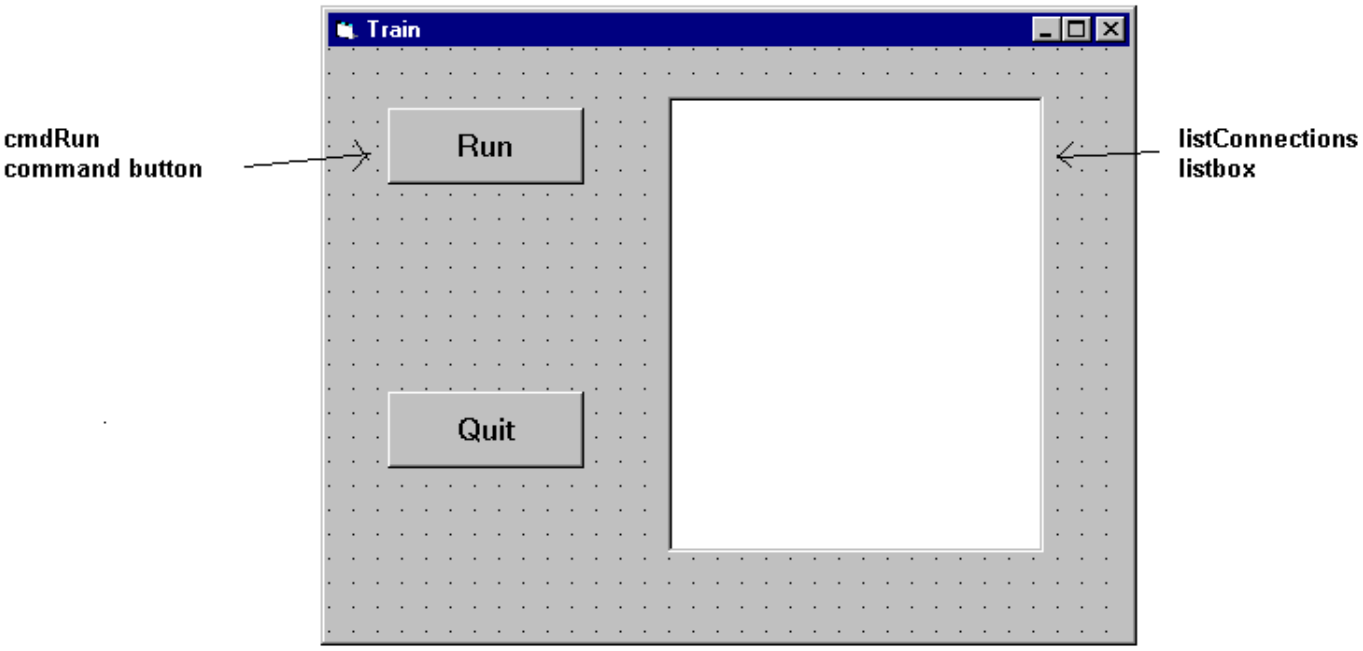
- Deinitialize Prolog in the `Form_Unload` procedure executed when the `calc` form is unloaded, e.g. when the application exits.

```
Private Sub Form_Unload(Cancel As Integer)
    PrologDeInit
End Sub
```

5.4.2 Example 2 - Train

This example provides a Visual Basic user interface to the Prolog program finding train routes between two points.

The Visual Basic program `train` contains the following form window:



Clicking the `cmdRun` command button will display all the available routes between Stockholm and Orebro. These are calculated as solutions of the Prolog query `places('Stockholm','Orebro',Way)`. For each solution, the value assigned to the variable `Way` is retrieved into the Visual Basic variable `result` and is inserted as a new item into the `listConnection` listbox.

The Visual Basic program consists of four parts:

- loading the Prolog code
- opening the query
- a loop generating the solutions, each cycle doing the following
  - requesting the next solution
  - getting the value of the solution variable
  - adding the solution to the listbox
- closing the query

```
Private Sub cmdRun_Click()
    Dim qid As Long
    Dim result As String
    Dim s As String
    Dim rc As Integer

    qid = -1 ' make it safe to PrologCloseQuery(qid) in Err:

    'load the 'train.pl' Prolog file
    rc = PrologQueryCutFail("ensure_loaded(app(train))")
    If rc < 1 Then
        Msg = "ensure_loaded(train)"
        GoTo Err
    End If
    'open the query
    qid = PrologOpenQuery("places('Stockholm','Orebro',Way)")
    If qid = -1 Then
        rc = 0
        Msg = "Open places/3"
        GoTo Err
    End If
    'generate solutions
    Do
        rc = PrologNextSolution(qid)
        If rc = 0 Then Exit Do ' failed
        If rc < 0 Then
            Msg = "places/3"
            GoTo Err
        End If
        If PrologGetString(qid, "Way", result) < 1 Then
            rc = 0
            Msg = "PrologGetString Way"
            GoTo Err
        End If
        listConnections.AddItem result
    Loop While True
    'after all solutions are found, the query is closed
    Call PrologCloseQuery(qid)
    Exit Sub
```

Note that each part does elaborate error checking and passes control to the error display instructions shown below:

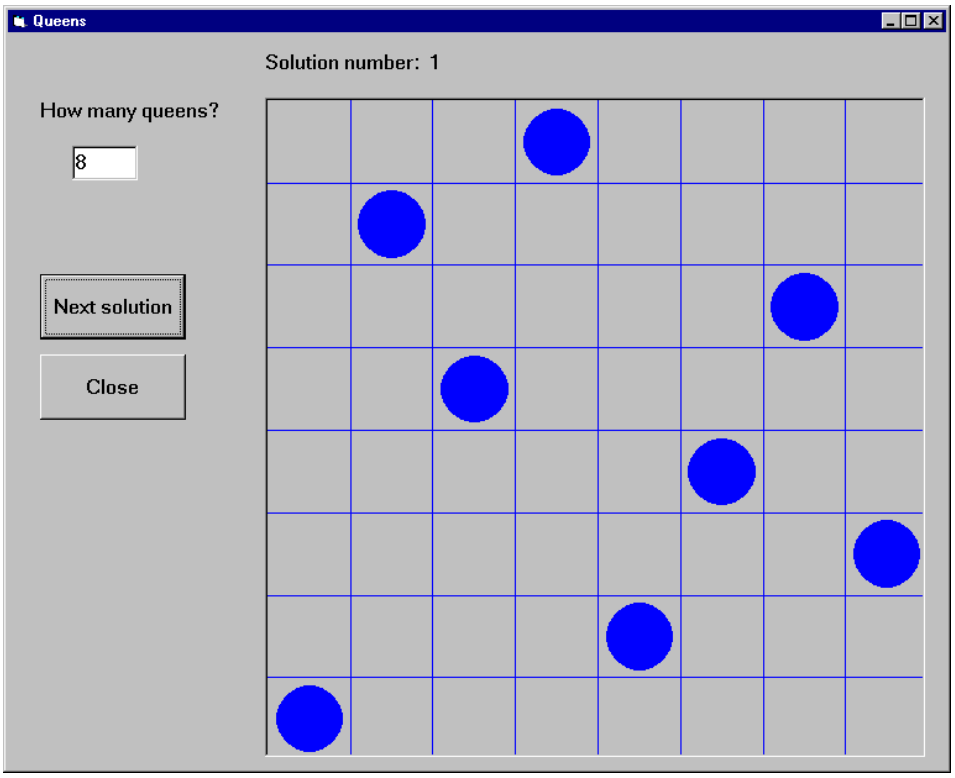
```
Err:
    Call PrologCloseQuery(qid) ' Always close opened queries

    'error message is prepared, adding either the - failed - or
    'the - raised exception - suffix to the Msg string specific
    'to the function called
    If rc = 0 Then
         Msg = Msg + " failed"
    Else
        Call PrologGetException(s)
        Msg = Msg + " raised exception: " + s
    End If
    MsgBox Msg, 48, "Error"
End Sub
```

The Prolog predicate `places/3` is defined in the '`train.pl`' file, as mentioned earlier.

5.4.3 Example 3 - Queens

This example gives a Visual Basic user interface to an N-queens program. The purpose of this example is to show how to handle Prolog lists through the Visual Basic interface. The full source of the example is found in the distribution.

The user interface shown in this example will allow the user to specify the number of queens, and, with the help of the `Next Solution` command button all the solutions of the N-Queens problem will be enumerated. A given solution will be represented in a simple graphical way as a PictureBox, using the basic `Circle` and `Line` methods.



The problem itself will be solved in Prolog, using a `queens(+N,?PositionList)` Prolog predicate, stored in the file '`queens`'.

We now present two solutions, using different techniques for retrieving Prolog lists.

Example 3a - N-Queens, generating a variable list into the Prolog call

The first implementation of the N-Queens problem is based on the technique of generating a given length list of Prolog variables into the Prolog query.

For example, if the N-Queens problem is to be solved for N = 4, i.e. with the query `"queens(4,L)"`, then the problem of retrieving a list from Visual Basic will arise. However, if the query is presented as `"queens(4,[X1,X2,X3,X4])"`, then instead of retrieving the list it is enough to access the `X1,X2,X3,X4` values. Since the number of queens is not fixed in the program, this query has to be generated, and the retrieval of the *Xi* values must be done in a cycle.

This approach can always be applied when the format of the solution is known at the time of calling the query.

We now go over the complete code of the program.

Global declarations used in the program ('`General/declarations`'):

```
Dim nQueens As Long         'number of queens
Dim nSol As Long            'index of solution
Dim nActqid As Long         'actual query identifier
Dim nQueryOpen As Boolean   'there is an open query
```

The initialization of the program will be done when the form window is loaded:

```
Private Sub Form_Load()
    nQueens = 0
    nSol = 1
    nQueryOpen = False

    'initialize Prolog
    If PrologInit() <> 1 Then GoTo Err
    'Load 'queens.pl'
    If PrologQueryCutFail("load_files(app(queens))") <> 1 Then GoTo Err
    Exit Sub

Err:
    MsgBox "Prolog initialization failed", 48, "Error"
    Unload Me
End Sub
```

Deinitialization of the Prolog engine will be done when the form windows is closed, exactly as for the calculator example.

When the number of queens changes (i.e. the value of the text box `textSpecNo` changes), a new query has to be opened, after the previous query, if there has been any, is closed.

```
Private Sub textSpecNo_Change()
    nQueens = Val(textSpecNo)
    nSol = 1

    If nQueryOpen Then PrologCloseQuery (nActqid)

    'create Prolog query in form: queens(4,[X1,X2,X3,X4])

    Q = "queens(" & Str(nQueens) & ", ["
    For i = 1 To nQueens - 1 Step 1
        Q = Q & "X" & i & ","
    Next
    Q = Q & "X" & nQueens & "])"

    nActqid = PrologOpenQuery(Q)
    nQueryOpen = True
End Sub
```

The `Next command` button executes and shows the next solution of the current query:

```
Private Sub cmdNext_Click()
    Dim nPos As Long
    Dim aPos(100) As Long

    If Not nQueryOpen Then
        MsgBox "Specify number of queens first!", 48, ""
        Exit Sub
    End If
    If PrologNextSolution(nActqid) < 1 Then
        MsgBox "No more solutions!", 48, ""
    Else
        For i = 1 To nQueens Step 1
           If PrologGetLong(nActqid, "X" & i, nPos) = 1 Then
               aPos(i - 1) = nPos
           End If
        Next i

        'display nth solution
        txtSolNo = "Solution number: " & Str(nSol)
        Call draw_grid(nQueens)

        nLine = 1
        For Each xElem In aPos
            Call draw_circle(nLine, xElem, nQueens)
            nLine = nLine + 1
        Next

        nSol = nSol + 1
    End If
End Sub
```

Drawing itself is performed by the `draw_grid` and `draw_circle` procedures.

Example 3b - N-Queens, converting the resulting Prolog list to an atom

The second variant of the N-Queens program uses the technique of converting the resulting Prolog list into a string via the `PrologGetString` function, and decomposing it into an array in Visual Basic. Here we show only those parts of the program which have changed with respect to the first version.

In the `textSpecNo_Change` routine the `queens/2` predicate is called with a single variable in its second argument:

```
Q = "queens(" & Str(nQueens) & ",Queens)"
nActqid = PrologOpenQuery(Q)
```

In the `cmdNext_Click` routine the solution list is retrieved into a single string which is then split up along the commas, and deposited into the `aPos` array by the `convert_prolog_list` routine. (`aPos` is now an array of strings, rather than integers.)

Finally, we include the code of the routine for splitting up a Prolog list:

```
Private Sub convert_prolog_list(ByVal inList As String,
                                ByRef inArray() As String)
    'drop brackets
    xList = Mid(inList, 2, Len(inList) - 2)

    i = 0

    startPos = 1
    xList = Mid(xList, startPos)

    Do While xList <> ""
        endPos = InStr(xList, ",")
        If endPos = 0 Then
            xElem = xList
            inArray(i) = xElem
            Exit Do
        End If
        xElem = Mid(xList, 1, endPos - 1)
        inArray(i) = xElem
        i = i + 1
        xList = Mid(xList, endPos + 1)
        startPos = endPos + 1
    Loop
End Sub
```

5.5 Summary of the Interface Functions

In this section you will find a summary of the functions and procedures of the Visual Basic interface:

`Function PrologOpenQuery (ByVal Goal As String) As Long`
: This function will return a query identifier that can be used for successive retrieval of solutions. Returns -1 on error, e.g. a syntax error in the query.

`Sub PrologCloseQuery (ByVal qid As Long)`
: This procedure will close the query represented by a query identifier `qid`. **Please note:** if `qid` is not the *innermost* query (i.e. the one opened last), then all more recently opened queries are closed as well.

`Function PrologNextSolution(ByVal qid As Long) As Integer`

This function retrieves a solution to the open query represented by the query identifier `qid`. Returns 1 on success, 0 on failure, -1 on error. In case of an erroneous execution, the Prolog exception raised can be retrieved with the `PrologGetException` procedure. **Please note:** Several queries may be open at the same time, however, if `qid` is not the *innermost* query, then all more recently opened queries are implicitly closed.

`Function PrologGetLong(ByVal qid As Long, ByVal VarName As String, Value As Long) As Integer`

Retrieves into `Value` the integer value bound to the variable `VarName` of the query identified by `qid`, as an integer. That is, the value of the given variable is converted to an integer. Returns 1 on success, i.e. if the given goal assigned an integer value to the variable, otherwise it returns 0. If an error occurred it returns -1, e.g. if an invalid `qid` was used.

`Function PrologGetString(ByVal qid As Long, Val VarName As String, Value As String) As Integer`

Retrieves into `Value` the string value bound to a variable `VarName` of the query, as a string. That is, the value assigned to the given variable is written out into an internal stream by the `write/2` Prolog predicate, and the characters output to this stream will be transferred to Visual Basic as a string. Retuns 1 on success, i.e. if the given goal assigned a value to the variable, otherwise it returns 0. If an error occurred it returns -1, e.g. if an invalid `qid` was used.

`Function PrologGetStringQuoted(ByVal qid As Long, ByVal VarName As String, Value As String) As Integer`

Same as `PrologGetString`, but conversion uses Prolog `writeq/2`. Returns 1 on success, i.e. if the given goal assigned a value to the variable, otherwise it returns 0. If an error occurred it returns -1, e.g. if an invalid `qid` was used.

`Function PrologQueryCutFail (ByVal Goal As String) As Integer`

This function will try to evaluate the Prolog goal, provided in string format, cut away the rest of the solutions, and finally fail. This will reclaim the storage used by the call. Returns 1 on success, 0 on failure and -1 on error.

`Sub PrologGetException(ByRef Exc As String)`

The exception term is returned in string format into the `Exc` string as if written by the `writeq/2` predicate. If there is no exception available then the empty string is returned.

`Function PrologInit() As Long`

The function loads and initiates the interface. It returns 1 on success, and -1 otherwise.

`Function PrologDeInit() As Long`

The function deinitializes and unloads the interface, and returns any memory used by the interface to the operating system. It returns 1 on success, and -1 otherwise.

6 The Debugger

6.1 Debugging Basics

6.1.1 Introduction

This section explains the basic concepts and terminology used by the Quintus Prolog debugger. Following sections describe the two debuggers available in the Quintus Prolog system: the X Windows-based source linked debugger and the standard debugger.

The main features of the debugging package are:

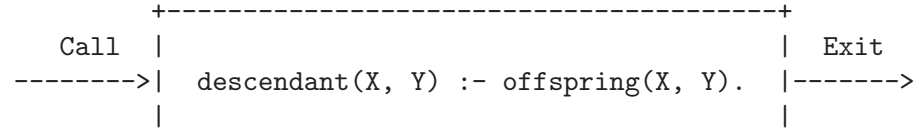
- The "procedure box" control flow model of Prolog execution, which provides a simple way of visualizing control flow, especially during backtracking.
- Full debugging of compiled code. It is not necessary to reload your program, or load it in any special way, in order to debug it.
- The ability to exhaustively trace your program or to selectively set spypoints. Spypoints allow you to specify procedures or goals where the program is to pause so that you can interact.
- A wide choice of control and information options available during debugging.
- Tools to aid in the understanding of your program's performance characteristics.

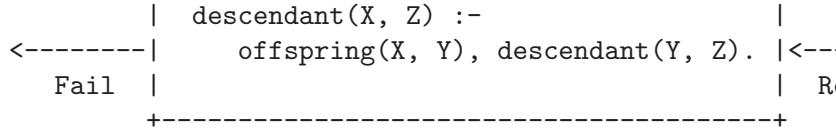
Debugging foreign code is discussed in Section 10.3.11 [fli-p2f-fcr], page 400.

6.1.2 The Procedure Box Control Flow Model

During debugging, the debugger stops at various points in the execution of a goal, allowing you see what arguments the goal is called with, what arguments it returns with, and whether it succeeds or fails. As in other programming languages, key points of interest are procedure entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly begins backtracking. The procedure box model of Prolog execution views program control flow in terms of movement from clause to clause in the procedures of the program. This model provides a basis for the debugging mechanism and enables the user to view program behavior in a consistent way.

Let us look at a representative Prolog procedure:





The first clause states that Y is a descendant of X if Y is an offspring of X, and the second clause states that Z is a descendant of X if Y is an offspring of X and if Z is a descendant of Y. In the diagram a box has been drawn around the whole procedure; labeled arrows indicate the control flow into and out of this box. There are four such arrows, which we shall describe in turn.

Call — This arrow represents an initial invocation of the procedure. When a goal of the form `descendant(X,Y)` must be satisfied, control passes through the Call port of the `descendant/2` box with the intention of matching the head of a component clause and then satisfying any subgoals in the body of that clause. Notice that this is independent of whether such a match is possible; first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for `descendant/2` when meeting a call to `descendant/2` in some other part of the code.

Exit — This arrow represents a successful return from the procedure. This occurs when the initial goal has been unified with the head of one of the component clauses and all subgoals have been satisfied. Control now passes out of the Exit port of the `descendant/2` box. Textually we stop following the code for `descendant/2` and go back to the code we came from.

Redo — This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes into the `descendant/2` box through the Redo port. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down onto another clause and following that if necessary.

Fail — This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if all solutions produced are always rejected by later processing. Control now passes out of the Fail port of the `descendant/2` box and the system continues to backtrack. Textually we move back to the code that called this procedure and keep moving backwards up the code looking for new ways of succeeding.

Note that the box we have drawn around the procedure should really be seen as an *invocation* box, rather than a procedure box. Often, there are many different Calls and Exits to a particular procedure in the control flow, but these are for different invocations. Each invocation box is given a unique integer identifier to prevent confusion.

6.1.3 Understanding Prolog Execution Using The Debugger

The Quintus Prolog debugger extends the procedure box control flow model to add extra information about the details of the execution of a goal, allowing you to better understand how your code behaves, and its efficiency.

Prolog incorporates a *backtracking* mechanism as a basic feature, which allows Prolog programs to efficiently search for multiple solutions to a problem. A goal is *determinate*

if it has only one solution (or none). Often, this is what is desired. When a goal is not determinate, Prolog must keep around information to allow it to backtrack to look for alternate solutions. This extra information is called a *choice point* .

When a goal is intended to be nondeterminate, it may be expected to leave a choice point. However, when a goal that is expected to be determinate leaves a choice point, this may indicate an error in the program. In this case, the goal might succeed with the correct answer, but on backtracking produce wrong answers or not terminate. At the very least, an unnecessary choice point means that memory is being wasted. Quintus Prolog detects many kinds of determinate goals, and either does not allocate a choice point at all, or deallocates it as soon as possible, saving time and memory (see Section 2.5.2.1 [bas-eff-cut-ove], page 33). Sometimes, however, you must help Prolog out by putting a cut

(see Section 2.5.2.2 [bas-eff-cut-mpd], page 34) in your program, or by using an if-then-else (see Section 8.2.7 [ref-sem-con], page 186) construct. The Quintus Prolog debugger helps you find such cases.

Quintus Prolog *index*es on the first argument of a predicate. This means that if the first arguments in the clauses of a predicate are not variables, and the first argument in a call to that predicate is non-variable, then Prolog will go directly to the clause that matches, without even considering those that don't. Note that in order for this indexing to be very efficient, it only looks at the principal functor of a compound term. This means that if the first argument of one clause is a(a) and the first argument of the next clause is a(b), indexing will not be able to distinguish these clauses, so both will need to be tried.

Actually, indexing is more complicated than this. Any clause whose head has a variable as first argument will match any call, so indexing cannot be applied to this clause. Therefore, a predicate can be divided into alternating groups of adjacent indexable and nonindexable clauses. When the first argument of a call is non-variable, Quintus Prolog will skip over any clauses that don't match that argument, within a group of adjacent indexable clauses. Quintus Prolog will then try every clause in a group of adjacent nonindexable clauses, and then again skip nonmatching clause in an indexable group, and so on.

Even more important than time saved by indexing is its effect on determinacy. In effect, indexing makes it possible to ignore clauses *following* the clause being tried, as well as clauses preceding it. If it is possible to ignore all the clauses following the clause being tried, then Prolog will not create a choice point, or if a choice point has already been

allocated for the call, Prolog will de-allocate it. Careful use of indexing can save a great deal time and memory running your program.

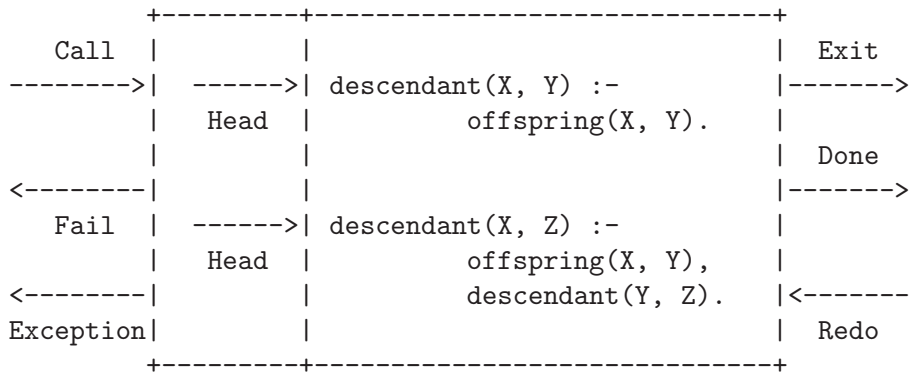
The Quintus Prolog debugger helps you understand these efficiency concerns, and also Quintus Prolog's exception handling, by extending the box model with three extra ports. These ports are described below.

Done — This is just like the Exit port, but signifies a determinate exit. This will help you to find goals that are nondeterminate and shouldn't be.

Head — This port shows the clauses' heads that will be tried for unification. At the Head port, you will see which clause is about to be tried, and which clause will be tried next if this clause fails. Note that the Head port is shown *after* indexing is done, so it helps you understand how indexing is working for you. And since it shows what clause will be tried next, it also helps you understand how unexpected nondeterminacy may be creeping into your program.

Exception — The Exception port signifies an exception being raised while executing a goal.

Here's a more complete picture of the invocation box, including the extended ports.



6.1.4 Traveling Between Ports

The Quintus Prolog debugger provides a rich set of commands to move between ports (that is, execute your program in a controlled way).

6.1.4.1 Basic Traveling Commands

creep — Causes the debugger to single-step to the very next port.

skip — Causes the debugger to skip, or ignore, the internal details of a procedure's execution. The debugger starts showing goals again when execution returns to the invocation's Done, Exit, Fail, or Exception port. If the debugger is already at the invocation's Done, Exit, Fail, or Exception port, then skipping is

meaningless, and the debugger will just creep. Note that skipping is very fast, running at nearly full compiled speed.

One important point about skipping: if the goal you skip over is not determinate, and you later redo (backtrack) into the goal, you will not be able to see the redos into goals that were skipped over. This is because the debugger does not keep any information about the goals that have been skipped over, in order to achieve much greater speed. You will, however, be able to see any new calls that are executed in the process of trying to redo the goal.

nonstop Turns off the debugger for the rest of the execution of the top-level goal. When the execution of this goal is completed, the debugger returns to its current mode. This option does *not* turn the debugger off; to turn the debugger off, you must type "nodebug." at the main Prolog prompt. Like skip, nonstop causes the debugger to run at nearly full compiled speed.

6.1.4.2 Spypoints

Spypoints allow you to indicate where to stop on a per-predicate or even per-goal basis. For example, you might find that you want to run until some particular predicate is called. In this case, you would set a spypoint on that predicate, using `spy/1`. Such spypoints are turned off with `nospy/1`.

It may be desirable to stop when you get to a particular call from one predicate to another. This can be done with the built-in predicate `add_spypoint/1`. These spypoints can be removed with `remove_spypoint/1`.

To examine spypoints, use `current_spypoint/1`. Spypoints are also included in the output of `debugging/0`. Finally, spypoints can be removed all at once with `nospyall/0`.

The debuggers also have commands for setting spypoints, which are easier to use than these predicates.

6.1.4.3 Traveling Commands Sensitive to Spypoints

The basic traveling commands listed above all ignore spypoints; the following commands take advantage of spypoints.

leap Causes the debugger to resume running your program, stopping only when the next spypoint is reached, or when the program terminates. Leaping can be used to follow the program's execution at a higher level than exhaustive tracing through creeping. This is done by setting spypoints on a set of pertinent procedures or calls, then following the control flow through these by leaping from one to the next.

quasi-skip Causes the debugger to resume running your program, stopping when the next spypoint is reached. It also ensures that the debugger will stop at the current

invocation's Done, Exit, Fail, or Exception port, when one is reached. Thus quasi-skip is a combination of leaping and skipping. You may use it to travel to the next spypoint while also marking a place to stop when execution returns there.

zip — Is just like leap, except that the debugger does not keep any debugging information while looking for a spypoint, so it runs at nearly full compiled speed. It does mean that the debugger will be unable to show you the ancestors between the invocation you zipped from and the invocation you stopped at.

Zipping gives up some information in exchange for greatly increased speed. This is not always desirable, but sometimes is very helpful. A good use for zipping might be to run through a time-consuming initial part of a computation that is known to work properly, and stop at the beginning of a part that has a bug. From that point, you might use leaping, creeping, and skipping to locate the bug.

6.1.4.4 Commands That Change The Flow Of Control

The debugger also has these commands that alter the flow of control of your program.

retry — This can be used at any of the seven ports (although at the Call port it has no effect). Control is transferred back to the Call port of the box. It allows you to restart an invocation when, for example, you find yourself leaving with some incorrect result. The state of execution is exactly the same as when you originally called the procedure, except that clauses that have been changed by the database modification predicates will not be changed back to their original state.

fail — This is similar to Retry except that it transfers control to the Fail port of the current box. It places your execution in a situation in which it is about to backtrack out of the current invocation, having failed the goal.

6.1.5 Debugger Concepts

6.1.5.1 Trace Mode, Debug Mode, And Zip Mode

The debugger has three modes of operation: trace mode, debug mode, and zip mode. While trace mode is in force, every time you type a goal at the top level, the debugger starts single-stepping (creeping) immediately. In contrast, while debug mode is in force, the debugger does nothing until a call is made to a procedure or goal on which you have placed a spypoint. That is, it starts by leaping. Similarly, when in zip mode, the debugger begins by zipping.

The debugger mode (trace, debug, or zip) determines the first procedure call that will be traced after a goal is typed at top level. There is no other difference among the three

modes. Whenever the debugger prompts you for input, you have a number of options, including those of creeping (single-stepping) leaping (jumping to the next spypoint), and zipping (jumping to the next spypoint without keeping debugging information).

The debugger mode can be set by using

- `prolog_flag/3` (see Section 8.10.1 [ref-lps-ove], page 245)
- `trace/0`
- `debug/0`
- `spy/1` also starts the debug mode if the debugger was off.
- `^c` interrupts: *`d`* for debug mode, and *`t`* for trace mode.
- the Debug menu of the QUI main window (see Section 3.2.1.2 [qui-mai-mai-deb], page 56)

6.1.5.2 Leashing

The purpose of leashing is to allow you to speed up single-stepping (creeping) through a program by telling the debugger that it does not always need to wait for user interaction at every port.

The leashing mode only applies to procedures that do not have spypoints on them, and it determines which ports of such procedures are leashed. By default, all ports are leashed. On arrival at a leashed port, the debugger will stop to allow you to look at the execution state and decide what to do next. At unleashed ports, the goal is displayed but program execution does not stop to allow user interaction.

At any time, there is a *leashing* mode in force, which determines at which of the seven ports of a procedure box (Call, Exit, Redo, Fail, Done, Head, and Exception) the debugger will stop and wait for a command. By default, the debugger will stop at every port, but sometimes you may wish to reduce the number of times you have to issue commands. For example, it is often convenient only to have to interact at the Call, Redo, and Exception ports.

To set the leashing mode, that is, to specify ports for leashing, call `leash/1`. The leashing mode can also be set from the options menu of the source linked debugger (see Section 6.2 [dbg-sld], page 121).

> **Please note:** Spypoints are not affected by leashing; the debugger will always stop at every port for a procedure or call on which there is a spypoint.

6.1.5.3 Locked Predicates

To allow users to produce code that cannot be debugged by others (for security reasons), Quintus Prolog 3.0 supports the concept of a *locked predicate*. A locked predicate will be

treated by the debugger as opaque: users will not be able to creep into it. The debugger will behave in much the same way for locked predicates as it does for built-ins.

Predicates may be locked by specifying the '`-h`' switch to `qpc` when compiling a file. See the documentation for `qpc`, Section 9.1.2 [sap-srs-qpc], page 341.

6.1.5.4 Unknown Procedures

The built-in predicate `prolog_flag/3` or `unknown/2` can be used to determine or set Prolog's behavior when it comes across an undefined predicate. By default, unknown procedures raise an exception.

Procedures that are known to be dynamic fail when there are no clauses for them.

When an undefined predicate is called and the undefined predicate behavior is set to `error` rather than `fail`, `unknown_predicate_handler/3` is called in module `user`. By defining this predicate, you can (selectively) trap calls to undefined predicates in a program.

6.1.5.5 Current Debugging State

Information about the current debugging state includes the following:

- the top-level state of the debugger
- the type of leashing in force
- the action to be taken on undefined predicates
- all the current spypoints

This information can be displayed by calling `debugging/0`.

6.1.6 Summary of Predicates

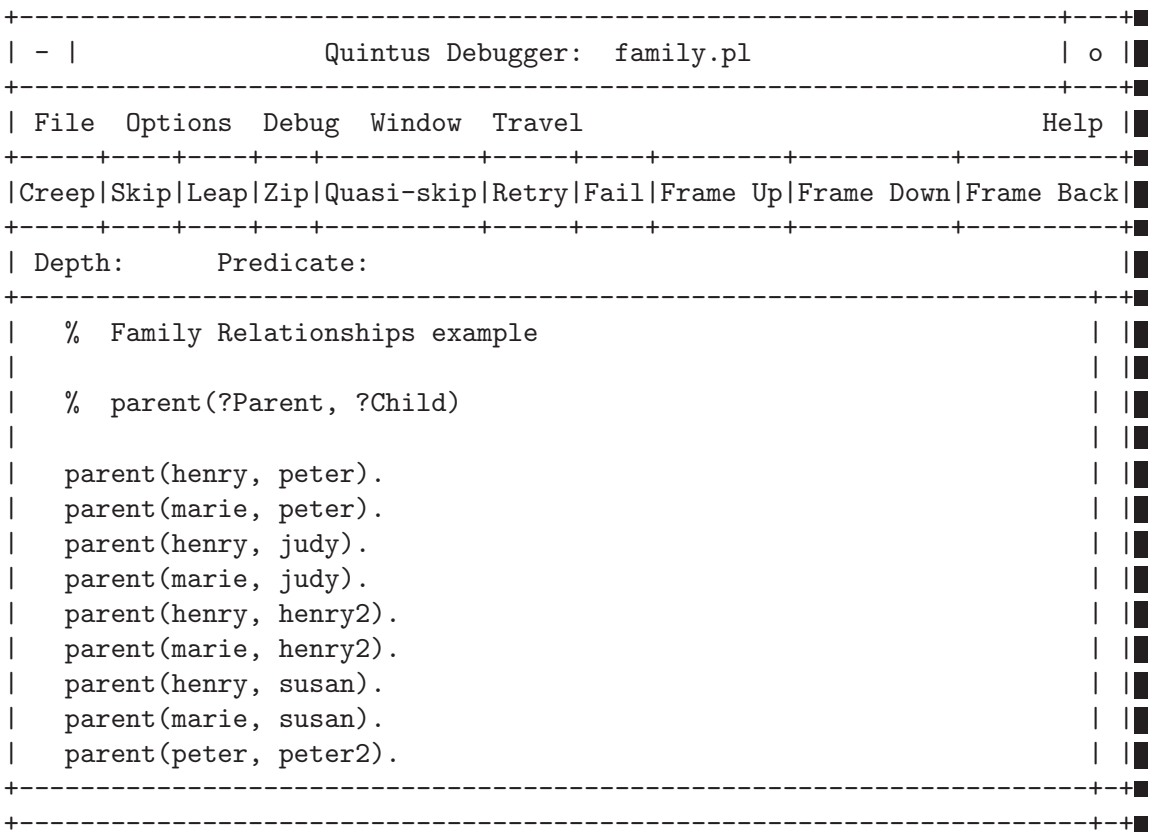
- `add_spypoint/1`
- `current_spypoint/1`
- `debug/0`
- `debugging/0`
- `leash/1`
- `nodebug/0`
- `nospy/1`
- `nospyall/0`
- `notrace/0`
- `remove_spypoint/1`
- `spy/1`

- `trace/0`
- `unknown/2`
- `unknown_predicate_handler/3`

6.2 The Source Linked Debugger

6.2.1 Introduction

Quintus Prolog's source linked debugger allows you to see the source for the code you are running as you step through the debugging. It also provides convenient, window-based ways of debugging your code, and provides several optional continuously-updated views of your program's execution state as debugging proceeds.



This picture shows what the source-linked debugger looks like. At the top of the picture is the debugger window's title bar, which shows the name of the file currently being shown in the source code window. Below this is a menu bar showing names of the available menus. Next is a row of buttons that are used to travel between ports while debugging. Below this is a status panel that shows you useful information about the current debugging state.

Finally, a scrolling window shows you where in your source code your current execution state is. All of these parts of the debugger are explained below.

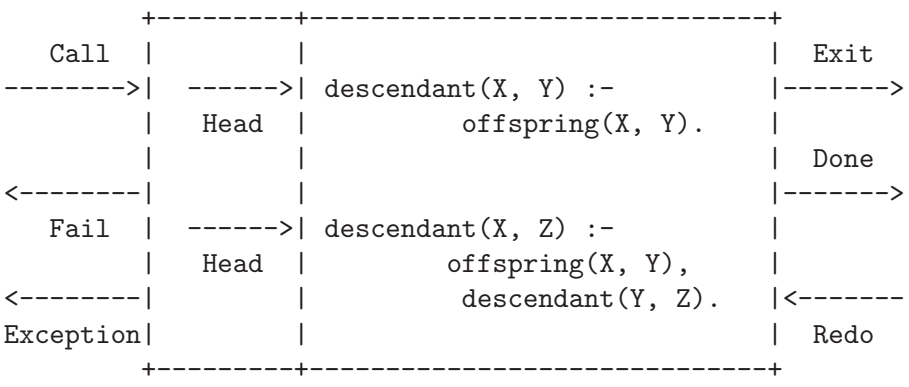
The source-linked debugger is displayed either by selecting one of the first three options in the **Debug** pulldown in the menu bar of the QUI main window (see Section 3.2.1.1 [qui-mai-mai-fil], page 56), by selecting the **Trace** button in the interrupt dialogue (see Section 3.2.4 [qui-mai-int], page 58), or by turning on the debugger with the `trace/0`, `debug/0`, or `prolog_flag/3` built-ins (see Section 6.1.5.1 [dbg-bas-con-tdz], page 118).

An alternative way of accessing the source-linked debugger is via the Emacs interface (see Section 4.1.4 [ema-ove-sld], page 82).

6.2.2 Showing Your Place In The Source Code

The source linked debugger shows your place in your source code by positioning an arrow near the goal being executed or clause to be tried. The position of the arrow and the direction in which it points reflect which debugger port you are at (see Section 6.1.2 [dbg-bas-pbx], page 113).

To help you visualize how this works, we repeat the picture of the procedure box from Section 6.1.3 [dbg-bas-upe], page 115:



6.2.2.1 The Call Port

The call port is shown by an arrow to the left of a goal, pointing toward it. This indicates the arrival at a goal.

```
descendant(X, Y) :-
   ---> parent(X, Y).
descendant(X, Z) :-
        parent(X, Y),
        descendant(Y, Z).
```

The Call Port

6.2.2.2 The Exit And Done Ports

The exit and done ports are shown by an arrow to the right of a goal, pointing away from it. This indicates the successful completion of a goal. The exit port is distinguished from the done port by having a forked tail; this is meant to reflect the fact that this is only one of possibly many solutions to this goal. The done port signifies a determinate exit. This will help you find goals that are nondeterminate and shouldn't be.

```
descendant(X, Y) :-
        parent(X, Y). ===>
descendant(X, Z) :-
        parent(X, Y),
        descendant(Y, Z).
```

The Exit Port

```
descendant(X, Y) :-
        parent(X, Y). --->
descendant(X, Z) :-
        parent(X, Y),
        descendant(Y, Z).
```

The Done Port

6.2.2.3 The Redo Port

The redo port is shown by an arrow to the right of the goal, pointing toward it. This indicates that an alternate solution to a completed goal is being sought.

```
descendant(X, Y) :-
        parent(X, Y). <---
descendant(X, Z) :-
        parent(X, Y),
        descendant(Y, Z).
```

The Redo Port

6.2.2.4 The Fail Port

The fail port is shown by an arrow to the left of the goal, pointing away from it. This indicates that no (more) solutions to this goal can be found.

```
descendant(X, Y) :-
   <--- parent(X, Y).
descendant(X, Z) :-
        parent(X, Y),
        descendant(Y, Z).
```

The Fail Port

6.2.2.5 The Head Port

The head port is shown as an arrow to the left of the clause, pointing toward it. This indicates which clause is about to be tried. If there are other clauses to be tried after this one, The tail of the arrow will be forked (suggesting that this is only one of possibly many clauses to be tried) and a smaller arrow will indicate the next clause to be tried. Note that indexing may mean that this is not the textually following clause.

```
     parent(henry, peter).
     parent(marie, peter).
===> parent(henry, judy).
     parent(marie, judy).
  -> parent(henry, henry2).
     parent(marie, henry2).
     parent(henry, susan).
     parent(marie, susan).
```

Nondeterminate Head Port

```
     parent(henry, peter).
     parent(marie, peter).
     parent(henry, judy).
     parent(marie, judy).
     parent(henry, henry2).
     parent(marie, henry2).
---> parent(henry, susan).
     parent(marie, susan).
```

Determinate Head Port

6.2.2.6 The Exception Port

The exception port is shown as an arrow to the left of the goal, pointing away from it. The tail of this arrow is broken, suggesting that something may be wrong with the program or data.

```
descendant(X, Y, 1) :-
        parent(X, Y).
descendant(X, Z, N) :-
        parent(X, Y),
   <- - descendant(Y, Z, N1),
        N1 is N+1.
```

The Exception Port

6.2.3 When Source Linking Is Not Possible

Sometimes the debugger cannot find the source code for a predicate. This will happen when there is no source code, or when the correspondence between the compiled code and source code cannot be determined. For example, a dynamic predicate does not necessarily have source code, and so the debugger currently cannot show source. Similarly, a meta-call (executing a term with `call/1`) does not have any source code. The debugger also often cannot find the source code of clauses produced by `term_expansion/2`. Predicates that are compiled from `user` do not have source either!

When source linking is not possible, the debugger will show as much of the clause as it knows in place of the source file, with the appropriate arrows. At a head port, it will show the goal being called, followed by '`:- ...`' indicating that this goal will be matched with the head of a clause. For the head of a predicate `descendent/2`, it might look like this:

```
===> dynamic_pred(_743) :- ...
```

At a call port whose source code cannot be shown, the debugger will show '`... :-`' followed by the goal, indicating that this goal is in some unknown clause. The arrow will be as appropriate for that port. The call port for a call to `descendant/2` might look like this:

```
... :- ---> descendant(peter, _749).
```

6.2.4 Traveling Between Ports

The source linked debugger provides a set of buttons to allow you to move from port to port while debugging. The debugger's travel commands are described in Section 6.1.4 [dbg-bas-tra], page 116; these buttons are labeled with the names of the commands, so using them should be straightforward.

```
+-----+----+----+---+----------+-----+----+-     +----------+
|Creep|Skip|Leap|Zip|Quasi-skip|Retry|Fail|  ... |Frame Back|
+-----+----+----+---+----------+-----+----+-     +----------+
```

The Traveling Buttons

When framed up, the skip, retry, and fail buttons operate relative to the invocation shown. Framing up is explained in Section 6.2.5 [dbg-sld-anc], page 125.

6.2.5 Seeing Ancestor Frames

The source linked debugger also provides a set of buttons to allow you to view ancestor invocation frames.

```
-+--------+----------+----------+
 |Frame Up|Frame Down|Frame Back|
```

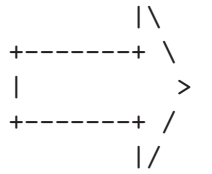
The Framing Buttons

The **Frame Up** button will show you the invocation before the one you are viewing. That means that the arrow will point to the place the invocation you are currently viewing was called from. Repeatedly hitting the Frame Up button will cause you to continue to traverse up through parent invocations, eventually stopping at the goal you typed at the top level prompt.

The **Frame Down** button may be used after you have framed up to take you back down toward the current invocation frame.

The **Frame Back** button will immediately take you back to the current invocation frame. This may also be used when you have framed up or scrolled the source window, and want to instantly scroll back to show the current invocation frame, or even if you are viewing another source file in the source window and want to get back.

In order to remind you when the goal you are viewing is not the current invocation frame, the arrow shown in the source window is hollow, or "ghosted", rather than the usual solid arrow.

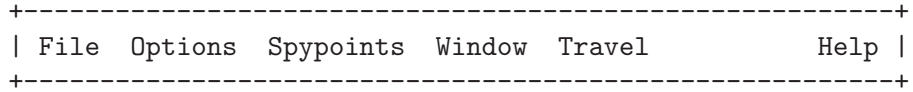


Ghost Arrow Shows Ancestor Frame

When you have framed up, certain travel buttons are interpreted relative to the frame you are currently viewing. **Skip** will skip over the invocation you are viewing (this is very handy if you have accidentally crept into a procedure you don't want to debug). **Redo** and **Fail** will take you to the call and fail ports of the selected invocation, respectively. All other travel buttons are disabled when you have framed up. You must either frame down or frame back in order to travel from the current invocation.

6.2.6 Debugger Menus

The source linked debugger has many options and commands that are invoked by menus. These are the menus available in the debugger window:

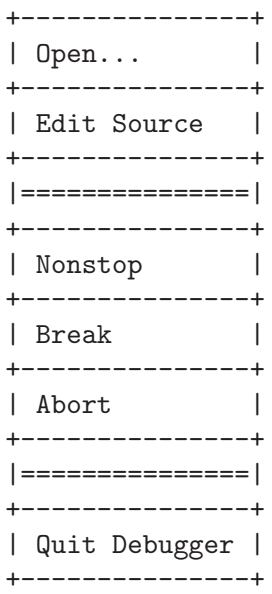


Debugger Menus

Below is a description of each menu and what it is used for.

6.2.6.1 The File Menu

The File menu contains commands that affect the file that is being debugged, and the debugger as a whole. The file menu is selected by clicking its button in the dubber window. The commands in the File menu are:



Selecting The File Menu

The **Open** command allows you to view another file in the source debugger window. This gives you a convenient way to set spypoints. When you select Open, the debugger pops up a dialogue, which allows you to select the file to view. Note that only files that have been loaded into Prolog can be viewed. The Frame Back button, described above, provides a convenient way to return to the current debugging invocation frame.

The **Edit Source** command provides a quick and convenient way to begin editing the file in the debugger window. A QUI editor window is opened on the file currently shown in the debugger window.

The **Nonstop** command turns off the debugger for the rest of the execution of the top-level goal. When the execution of this goal is completed, the debugger returns to its current mode (trace, debug, or zip). This option does *not* turn the debugger off; to turn the debugger off, you must use the Quit Debugger option, or type `nodebug.` at the main Prolog prompt.

The **Break** command calls the built-in predicate `break/0`, thus suspending the execution so far and putting you at the equivalent of a new Prolog top level. (See the description of `break/0` in Section 8.11.1 [ref-iex-int], page 250.) The new execution is separate from the suspended one, and invocation numbers will start again from 1. The debugger is turned off, and the debugger window is closed, when the break level is entered, although the spypoints and leashing of the suspended level are retained. When you end the break (by typing the end-of-file character), execution will resume and you will be prompted once again at the

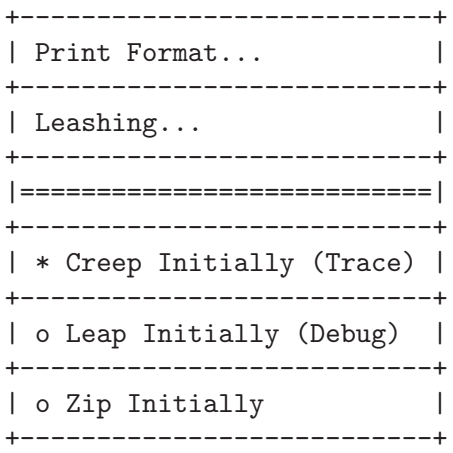
port that you left. Changes to leashing or to spypoints will remain in effect after the break has finished.

The **Abort** command aborts (abandons) the current execution. All the execution states built so far are destroyed, and execution restarts at the top level (or current break level).

The **Quit Debugger** command turns off the debugger altogether, just like the `nodebug/0` command (see Section 18.3.111 [mpg-ref-nodebug], page 1195). The debugger window is also closed at this time, since it is only open when debugging is on.

6.2.6.2 The Options Menu

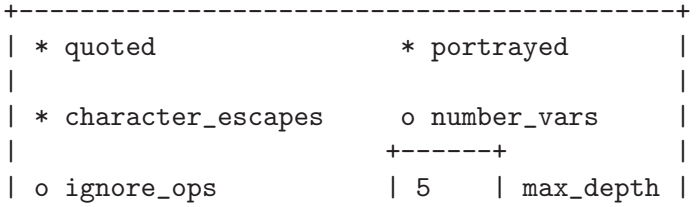
The Options menu allows you to change various aspects of the behavior of the source linked debugger. The Options menu is selected by clicking its button in the debugger window as shown.

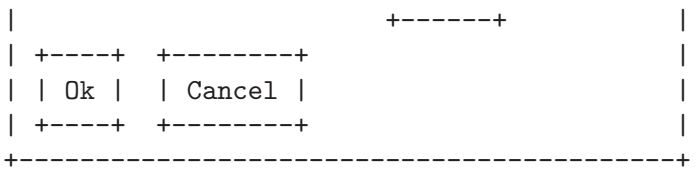


Selecting The Options Menu

Beginning with the bottom part of the menu, the **Creep**, **Leap**, and **Zip Initially** toggles allow you to set the current debugging mode (see Section 6.1.5.1 [dbg-bas-con-tdz], page 118).

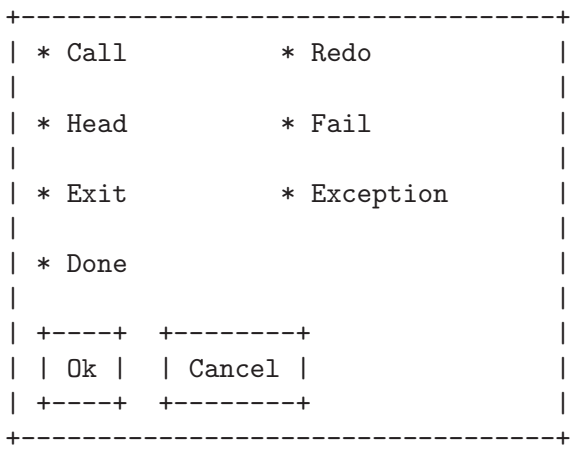
The **Print Format** button brings up a dialogue that will allow you to change the way goals will be printed when source linkage is impossible (see Section 6.2.3 [dbg-sld-whe], page 125). This dialogue has a toggle button for each true/false `write_term/[2,3]` option (see Reference page), and a place to enter a print depth limit. Leaving the print depth empty (or setting it to 0) will mean that there is no depth limit; the goal will be printed in full.





The Print Format Dialogue

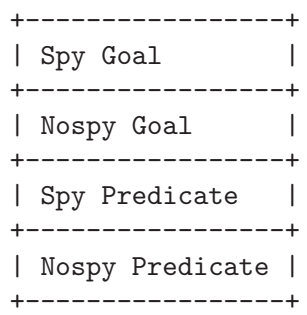
The **Leashing** button brings up a dialogue that lets you set your leashing mode (see Section 6.1.5.2 [dbg-bas-con-lea], page 119).



The Leashing Dialogue

6.2.6.3 The Spypoint Menu

The Spypoint menu allows you to set spypoints in your code by selecting the goal to be spied with the mouse, thereby highlighting the goal, and then selecting a spy command (see Section 6.1.4.2 [dbg-bas-tra-spy], page 117 for an explanation of spypoints). You may add or remove spypoints from goals or predicates this way.



Selecting The Spypoints Menu

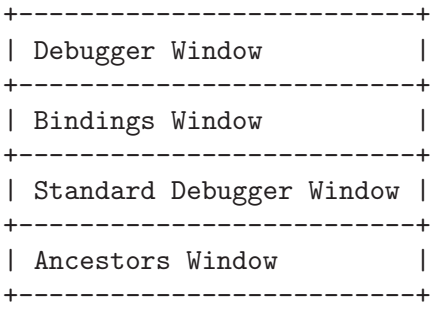
Selecting the **Spy Goal** command will place a spypoint on the currently selected goal, and selecting **Nospy Goal** will remove the spypoint. Selecting the **Spy Predicate** command will place a spypoint on the predicate for which a goal (or clause head) is currently selected,

and **Nospy Predicate** will remove the spypoint. The difference between goal and predicate spypoints is that a spypoint on a predicate will stop the debugger regardless of how that predicate is called, while a goal spypoint will only stop when the predicate is called from that particular goal.

When a spypoint is placed on a goal, a small stop-sign is placed before that goal in the debugger window, indicating that it is spied. Similarly, when a predicate is spied, a stop-sign is placed before the first clause for that predicate.

6.2.6.4 The Window Menu

The Window menu allows you to pop up various windows, which provide useful information not present in the debugger window. Selecting a window from this menu will open the specified window, or, if it is already open, cause it to pop to the front of any windows that might be covering it. This menu is available in all debugger windows.

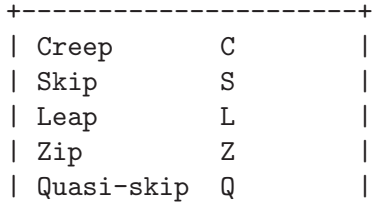


The Windows Menu

Debugger Window refers to the main debugger window, and may be used when the debugger window is covered by other windows to bring it to the front. The other windows, the **Variable Bindings Window**, **Standard Debugger Window**, and **Ancestors Window** are discussed in depth below. (see Section 6.2.8 [dbg-sld-oth], page 132)

6.2.6.5 The Travel Menu

The **Travel** Menu provides all the same commands as in the travel and framing panels, as described in Section 6.2.4 [dbg-sld-tra], page 125. They are provided in a menu for those who are more comfortable using menus, and also to document the keyboard shortcuts for the traveling and framing commands.



```
| Retry       R        |
| Fail        F        |
+----------------------+
| Frame Up    Ctrl + U |
| Frame Down  Ctrl + D |
| Frame Back  Ctrl + B |
+----------------------+
```

The Travel Menu

6.2.6.6 The Help Menu

The **On This Window** item opens up the help window viewing the documentation on this window. This menu is available in all debugger windows.

```
+----------------+
| On This Window |
+----------------+
```

The Help Menu

6.2.7 The Status Panel

The status panel is located just below the buttons in the debugger window, and shows the invocation depth of the current arrow, as well as information about the called predicate.

```
+-------------------------------------------+
| Depth: 1    Predicate:  descendant/2      |
+-------------------------------------------+
```

The Status Panel

Note that the depth shown is the depth of the frame currently being shown, so if you use the Frame Up button to show ancestors (see Section 6.2.5 [dbg-sld-anc], page 125), the depth will reflect the frame being shown.

The Predicate field first shows the module (if other than user), name, and arity of the predicate being called. Second is shown information about the predicate and the port, which is only shown when the there is something unusual or interesting to be noted about the predicate or port. The information about the predicate is one of the following:

`built_in` for built-in predicates; or

`locked` for locked predicates; or

`undefined`
for undefined predicates; or

`foreign` for foreign predicates (defined in another programming language); or

`dynamic` for dynamic predicates; or

`multifile`
: for multifile predicates

Nothing is shown for ordinary user-defined static (compiled) procedures defined all in a single file. The debugger also gives information that may help you understand why it has stopped where it has. This information may be one of the following:

`skipped` if you previously skipped (or quasi-skipped) from this invocation; or

`spied` if there is a spypoint on this goal or predicate

6.2.8 Other Windows

Of course, you will want to know more than just which predicate is being called or which clause is about to be tried. Other useful information includes the bindings of the variables in the goal being executed, the history of your debugging session, and the ancestors (the call stack) of the current goal. This information is available in separate windows, which you may bring up using the window menu, as described in Section 6.2.6.4 [dbg-sld-men-wme], page 130.

6.2.8.1 The Variable Bindings Window

The **variable binding window** shows the current bindings of the variables that appear in source code for the goal being called. This information is displayed in a format similar to that used when Prolog returns to the top level and shows the results of a user-typed goal.

For example, the variable bindings window might show the following:

```
+---+-------------------------------------------------------------+
| - |             Quintus Debugger Variable Bindings              |
+---+-------------------------------------------------------------+
| File  Options  Window                                     Help |
+---------------------------------------------------------------+
| X = henry                                                   | |
| Y = _749                                                    | |
|                                                             | |
|                                                             | |
+-------------------------------------------------------------+-+
+---------------------------------------------------------------+-+
```

The Variable Bindings Window

Unlike the other windows provided by the debugger, the variable bindings window changes its display depending on which frame is being shown in the main debugger window. This allows you to examine the current bindings for the arguments to any ancestor goal. See Section 6.2.5 [dbg-sld-anc], page 125 for more information on the debugger's framing commands.

6.2.8.2 The Standard Debugger Window

The *standard debugger window* shows what would be shown by the standard debugger (see Section 6.3 [dbg-sdb], page 134). This gives you a history of the debugging session from the time you opened the Standard Debugger window that you may scroll back through later to review what has happened. The standard debugger window might look like this:

```
+---+-------------------------------------------------------------+
| - |                 Quintus Standard Debugger                   |
+---+-------------------------------------------------------------+
| File  Options  Window                                      Help |
+-------------------------------------------------------------+
|   (2) 1 Exit: parent(henry,peter)                             | |
|   (1) 0 Exit: descendant(henry,peter)                         | |
|   (1) 0 Redo: descendant(henry,peter)                         | |
|   (2) 1 Redo: parent(henry,peter)                             | |
|                                                               | |
+-------------------------------------------------------------+-+
+-------------------------------------------------------------+-+
```

The Standard Debugger Window

6.2.8.3 The Ancestors Window

The **ancestors window** shows the current invocation and all its ancestors, and is continuously updated. It can be a very powerful tool in debugging, as it lets you quickly see how variable bindings propagate to ancestor goals. The ancestors window might look like this:

```
+---+-------------------------------------------------------------+
| - |               Quintus Debugger Ancestor List                |
+---+-------------------------------------------------------------+
| File  Options  Window                                      Help |
+-------------------------------------------------------------+
|   (8) 3 : parent(peter2,_749)                                 | |
|   (7) 2 : descendant(peter2,_749)                             | |
|   (4) 1 : descendant(peter,_749)                              | |
|   (1) 0 : descendant(henry,_749)                              | |
|                                                               | |
+-------------------------------------------------------------+-+
+-------------------------------------------------------------+-+
```

The Ancestors Window

Since the ancestors window is updated every time the debugger stops, when there are very many ancestors, you may notice some slowdown in the debugger. In this case, you may wish to close the ancestors window, and only open it when you really need to examine the

ancestors. Usually, though, the slowdown caused by having the ancestors window open is small.

6.2.8.4 Menus For These Windows

All of these extra windows have the same menus, with the same options. These menus are as follows:

```
+-------------------------------------------------------------------+
| File  Options  Window                                       Help  |
+-------------------------------------------------------------------+
```

Extra Window Menus

The File menu contains only a close command, which simply closes that window:

```
+-------+
| Close |
+-------+
```

The Extra Window File Menu

The Options menu contains only a Print Format command:

```
+-----------------+
| Print Format... |
+-----------------+
```

The Extra Window Options Menu

Selecting **Print Format** will pop up a dialogue, which allows you to control the printout of the information in that window. The operation of this dialogue is explained in Section 6.2.6.2 [dbg-sld-men-opt], page 128.

Note that changing the print format of the variable bindings window or the ancestors window will cause that window to be updated immediately, to reflect the new print format. However, changing the print format of the standard debugger window will only change the format of subsequent entries in the window; lines already written will not be changed.

6.3 The Standard Debugger

The standard debugger is a traditional terminal-based (as opposed to window-based) debugger. Section 6.1.1 [dbg-bas-bas], page 113 describes the basic debugger facilities; this section only describes the features of the standard debugger.

6.3.1 Format of Debugging Messages

After you turn on the debugger and type the goal you want to debug, the system begins to show the steps of the procedure's execution. As the system passes through each port of a procedure, it displays a debugging message on your terminal.

A sample debugging message and an explanation of its symbols are shown below.

```
** (23) 0 Call (dynamic): mymodule:foo(hello,there,_123) ?
```

'`**`' The first two characters indicate whether this is a spypoint and whether this port is being entered after a '`skip`'. The possible combinations are:

- '`**`' This is a spypoint. (`foo/3` has been spied.)
- '`*>`' This is a spypoint; you are returning from a '`skip`'.
- '` >`' This is not a spypoint; you are returning from a '`skip`'.
- '` `' This is not a spypoint. (For this condition, two blank spaces are displayed at the left of the message.)

'`(23)`' The number in parentheses is the unique invocation identifier. This identifies a particular call to the procedure. This number can be used to correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter is reset to 1 whenever a new goal is typed at the top level, and is also reset when retries (see Section 6.1.4.4 [dbg-bas-tra-ccf], page 118) are performed.

'`0`' The next number is the current depth — that is, the number of direct ancestors this goal has. The ancestors can be printed using the *g* debugging option. The depth increases as procedures are called and decreases as procedures return. There may be many goals at the same depth, which is why the unique invocation identifier is also provided. If the depth is shown as 0, this spypoint is on a compiled procedure and no depth or ancestor information is available. The depth is reset to 0 when a compiled procedure is executed, and begins growing again from there afterward.

'`Call`' The next word shows the particular port: Head, Call, Exit, Done, Redo, Fail, or Exception.

'`(dynamic)`'
The next parenthesized item, if present, indicates when there is something unusual about the predicate. The possibilities are:

- `built_in` for built-in predicates; or
- `locked` for locked predicates; or
- `undefined` for undefined predicates; or

`foreign` for foreign predicates (defined in another programming language); or

`dynamic` for dynamic predicates; or

`multifile`
for multifile predicates.

'`mymodule`'
If the procedure currently being debugged was loaded into a module other than `user`, the module name will be displayed here, followed by a colon.

'`foo(hello,there,_123)`'
The goal is then printed so that its current instantiation state can be seen. (At Redo ports, the instantiation state shown is the same as at the previous Exit.) This is done using `print/1` so that all goals displayed by the debugger can be "pretty printed", if the user wishes, using `portray/1` clauses. The debugger also maintains a print depth limit and will only show terms nested down to this depth. The system initially uses a limit of 10, but this can be changed using the `<` debugging option.

'`?`' The final '`?`' is the prompt indicating that you should type in one of the option codes (see next section). If this particular port is unleashed, there will be no prompt and the debugger will continue to the next port.

Please note: Since the system does not allow the placing of spypoints on built-in predicates, the only way to show the execution of built-in predicates typed at the main Prolog prompt is to select trace mode. For example, if you typed *`write(foo).`* at the main Prolog prompt with the debugger in debug mode, the system would simply display the word '`foo`' without tracing the execution of the predicate `write/1`. However, if a built-in predicate such as `write/1` were called from within a program, the execution of the predicate would be shown in any case that the execution of the procedure containing it would be shown. There are a few basic built-in predicates for which information is not displayed because it is more convenient not to trace them. These are: `true/0`, `otherwise/0`, `false/0`, `fail/0`, `=/2`, `!/0` (cut), `;/2` (or), `,/2` (and), and `->/2` (local cut).

6.3.1.1 Format of Head Port Messages

The message shown at a Head port is slightly different than the messages at other ports. Rather than optionally showing something unusual about the predicate (`dynamic` in the above example), the Head port shows you which clause of the predicate is being, and which, if any, will be tried next. For example, a Head port might be printed as follows:

```
(23) 0 Head [2->4]: mymodule:foo(hello,there,_123) ?
```

Here the '`[2->4]`' means that clause 2 for predicate `mymodule:foo/3` is about to be tried. If this clause's head should fail to unify, or if a goal in the body of the clause fails, clause 4 will be tried next. Clause 3 will be skipped, due to indexing (see Section 2.5.3 [bas-eff-ind], page 36). If clause 2 were the last clause to be tried, it would be shown as '`[2]`'. And in

the case where clause 2 is being tried, and all the following clauses are indexable, but none can match the goal, the debugger will show '`[2->fail]`', suggesting that a choice point is being left, even though no more solutions will be possible (see Section 2.5.3 [bas-eff-ind], page 36 for an explanation of indexable clauses, and how indexing works).

6.3.1.2 Format of Exception Port Messages

The Exception port shows the exception term describing the exception that has been raised. This is printed on a separate line before the line printed for the port itself. For example:

```
! type_error(_2069 is a,2,'arithmetic expression',a)
   (2) 1 Exception (built_in): _2050 is a ?
```

For exception term formats or more information about the exception handling system of Quintus Prolog, see Section 8.19 [ref-ere], page 310.

6.3.2 Options Available during Debugging

6.3.2.1 Introduction

This section describes the options that you can select in response to the '`? `' prompt, which is displayed after the debugger prints out a goal. The options are one-letter mnemonics, some of which optionally can be followed by a decimal integer. Any layout characters are ignored up to the next newline.

The most important option to remember is *h* (for help). When you type *h*, the following list of available options is displayed:

```
Debugging options:

<cr>   creep      p      print         r [i]  retry i      @    command
 c     creep      w      write         f [i]  fail i       b    break
 l     leap       d      display                           a    abort
 s [i] skip i                                             h    help
 z     zip        g [n]  n ancestors   +      spy pred     ?    help
 n     nonstop    < [n]  set depth     -      nospy pred   =    debugging
 q     quasi-skip .      find defn     e      raise_exception
```

These options provide a number of different functions, which fall into the following classes:

Basic control
: he basic ways of continuing with the execution

Printing
: howing the goal, or its ancestors, in various ways

Advanced control
: ffecting control flow

Environment
: hanging spypoints, executing commands, breaking and aborting, access to source code

Help
: howing the debugger state and listing options

Each of the options is described below.

6.3.2.2 Basic Control Options

These commands are described in depth in Section 6.1.4 [dbg-bas-tra], page 116 and Section 6.1.4.2 [dbg-bas-tra-spy], page 117.

⟨RET⟩
: (the Return key) — This is the same as the `c` (creep) option but is reduced to a single keystroke for convenience.

`c`
: creep — This causes the debugger to single-step to the next port and display its goal.

`l`
: leap — This causes the debugger to resume running your program, stopping only when the next spypoint is reached, or when the program terminates.

`s`
: skip — At a Call, Redo, or Head port, this skips over the entire execution of the procedure. At any other port, this is equivalent to the `c` (creep) option.

`s i`
: skip over ancestor invocation — If you specify an invocation identifier (see Section 6.3.1 [dbg-sdb-dme], page 135) of an ancestor invocation after the skip command, it will skip over that invocation. This means that that ancestor goal will be completed without stopping.

`z`
: zip — This causes the debugger to run until the next spypoint is reached, just like leaping, except that no debugging information is kept, so execution is *much* faster. See Section 6.1.4.3 [dbg-bas-tra-tss], page 117 for more information on the trade-offs between speed and information when using this option.

`q`
: quasi-skip — This causes the debugger to run until the next spypoint is reached, or until this invocation is completed. This option combines the behavior of leaping and skipping. It also ensures that the debugger will stop at the Exit, Done, or Fail port of the current invocation, as soon as it is reached.

6.3.2.3 Printing Options

`p`
: print — This option prints the current goal using `print/1` and the current debugger print depth limit. The depth limit prevents very large structures from being shown in their entirety; it can be changed with the `<` (set depth) option (see below).

`w` write — This option writes the current goal on the terminal using `write/1`. This may be useful if your "pretty print" routine (`portray/1`) is not doing what you want. The `write` option has no depth limit.

`d` display — This option displays the current goal on the terminal using `display/1`. This shows the goal in prefix notation. The `display` option has no depth limit.

`g` ancestors — This option prints the list of ancestors of the current goal (that is, all goals that are between the current goal and the top-level goal in the calling sequence). Each ancestor goal is printed using `print/1` with the current debugger depth limit. Goals shown in the ancestor list are always accessible to invocations for the `r` (retry) option.

`g n` ancestors —

This is a version of the `g` option, which prints only *n* ancestors. That is, the last *n* ancestors will be printed counting back from the current goal.

`< n` set depth — This option sets the debugger print depth limit to *n*. This limit determines the depth to which goals are printed when they are shown by the debugger. The depth limit is also used when showing the ancestor list. If *n* is 0, or is omitted, the debugger will then use no limit when printing goals. The default limit is 10.

6.3.2.4 Advanced Control Options

These options are described in greater depth in Section 6.1.4.4 [dbg-bas-tra-ccf], page 118.

`r` retry — This option restarts the current invocation. This may be useful when, for example, you find yourself leaving with some incorrect result. When a retry is performed, the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that execution has returned to the state it was in at the time of the original call. The message

```
% Retrying goal
```

is displayed to indicate where this occurred, in case you wish to follow these numbers later.

`r i` retry previous invocation — If you supply an integer after the retry command, then this is taken as specifying an invocation number and the system tries to get you to the Call port, not of the current box, but of the invocation box you have specified. This must be an ancestor invocation.

`f` fail — This is similar to Retry except that it transfers control to the Fail port of the current box. This places your execution in a situation in which it is about to backtrack out of the current invocation, having failed the goal.

`f i` fail previous invocation — This is similar to `r i` except that it transfers control to the Fail port of the invocation specified (which must be an ancestor).

6.3.2.5 Environment Options

`n` nonstop — This option turns off the debugger for the rest of the execution of the top-level goal. When the execution of this goal is completed, the debugger returns to its current mode (trace, debug, or zip). This option does *not* turn the debugger off; to turn the debugger off, you must type:

```
| ?- call nodebug
```

`+` spy this — This option places a spypoint on the procedure currently being shown.

`-` nospy this — This option removes any spypoint on the procedure currently being shown.

`@` command — This option prompts for a single Prolog goal, which is executed as a command without any variable results being shown. The command is run as a new execution, with the current execution suspended, but without any debugging. This is particularly useful for quickly changing debugging parameters without entering a break level.

`b` break — This option calls the built-in predicate `break/0`, thus suspending the execution so far and putting you at the top level of the interpreter. (See the description of `break/0` in Section 8.11.1 [ref-iex-int], page 250.) The new execution is separate from the suspended one, and invocation numbers will start again from 1. The debugger is turned off when the break level is entered, although the spypoints and leashing of the suspended level are retained. When you end the break (by typing the end-of-file character), execution will resume and you will be prompted once again at the port that you left. Changes to leashing or to spypoints will remain in effect after the break has finished.

`a` abort — This option aborts (abandons) the current execution. All the execution states built so far are destroyed, and execution restarts at the top level of the interpreter.

`.` find-definition — This works only under the editor interface. The source code corresponding to the current procedure call is found and displayed in the text window.

`,` find-more-definition — This works only under the Emacs interfaces, and can be used after doing a find-definition, in the case where there is more than one possible definition. This is useful in several cases:

1. where a predicate is multifile — you can find all the files that have clauses for that predicate;
2. where you specify a name but no arity — you can find all the definitions of predicates with that name and with different arities;
3. where you have the same name/arity defining predicates in different modules.

6.3.2.6 Help Options

`h`	help — This option displays the table of options given above.
`?`	help — This is equivalent to the `h` option.
`=`	debugging — This option shows the current state of the debugger, the spypoints that have been set, the leashed ports, and the behavior on undefined procedures.

6.4 The Advice Facility

The "advice" facility for program debugging and maintenance makes it possible to associate user-specified goals with any of the ports of a predicate (see Section 6.1.2 [dbg-bas-pbx], page 113). Advice may not change the relation computed by the advised predicate, but it may switch on or off debugging facilities, or can check that calls are well-formed and that results make sense. Any user-defined predicate may be advised; however, it is not currently possible to put advice on built-ins.

6.4.1 Use of Advice Predicates

Whether a predicate has advice associated with it is independent of whether advice on that predicate is being "checked". Advice may be thought of as a generalized spypoint, where the user specifies the action to be performed at each debugger port. Distinct built-in predicates are provided for modifying advice and influencing whether advice is being checked. Advice is associated with a predicate using `add_advice/3` and deleted using `remove_advice/3`. You can determine what advice is outstanding by using `current_advice/3`. Advice on a predicate is retained if that predicate is abolished, as is likely to happen during program development. It also survives that predicate being saved to a QOF file, and is reinstated when that QOF file is loaded.

Advice checking can be enabled one predicate at a time using `check_advice/1` and disabled one predicate at a time with `nocheck_advice/1`. These are analogous to `spy/1` and `nospy/1`, and are similarly defined as prefix operators of precedence 900. As a convenience, `check_advice/0` is provided to enable checking for all predicates that currently have advice associated with them. `nocheck_advice/0` can also be used to turn off all advice checking easily. It is possible to use `check_advice/1` to turn advice checking on for a predicate that does not currently have any advice, although doing so will have no effect until advice is added to the predicate. As `check_advice/0` only affects predicates that currently have advice, it cannot be used for this purpose. Like the advice itself, advice checking survives a checked predicate being abolished and redefined.

Note that whether a particular call to an advised predicate is checked is determined once the call is made. If advice checking is enabled after a predicate has been entered but before it reaches a subsequent advised port of the predicate, advice checking will not be done.

Advice on any given port is checked before any debugger interaction for that port. Taking advice checking as well as debugger interaction into account in the procedure box model of Prolog execution (see Section 6.1.2 [dbg-bas-pbx], page 113), we have

```
                          +-----------+
Call -> Advice -> Spy -> |           | Advice -> Spy -> Exit ->
                          | procedure |
                          |    box    |
Fail <- Spy <- Advice <- |           | <- Spy <- Advice <- Redo
                          +-----------+
```

This is necessary so that advice actions can control the debugger.

It is possible (and often advisable) to write an application where advice is always present on appropriate predicates, but that advice is not checked by default. When required, advice checking can be turned on and the application monitored or debugged using the advice. This makes it unnecessary to add advice each time you want it only having to remove it when you don't want it any longer. Instead, the application's advice can persist, with modifications as desired, through development. It can also be retained once development is completed, providing diagnostics that can be turned on in case of problems.

Predicate properties (see `predicate_property/2`) have been added to allow you to determine what predicates have outstanding advice, or have advice checking enabled. When a predicate currently has advice associated with it, it has the property `has_advice`. If advice is being checked for that predicate, it has the property `checking_advice`. These properties are independent of each other, and are independent of whether the advised predicate is currently defined or not. For example:

```
| ?- check_advice(foo/0).
* You have no clauses for user:foo/0
% Advice checking enabled on user:foo/0

yes
| ?- predicate_property(foo, P).

P = checking_advice ;

no
| ?- add_advice(foo, call, (write(hi),nl)).
% Advice placed on user:foo/0

yes
| ?- predicate_property(foo, P).

P = has_advice ;

P = checking_advice ;

no
```

6.4.2 Performance

There is no performance penalty associated with advice that is not being checked. When checking advice, there is no loss of performance for predicates that do not have advice associated with them or whose advice is not currently being checked. The cost in space of advice is roughly equivalent to the cost of an asserted fact the size of the `add_advice/3` call. The memory cost for checking advice on a predicate is the same as that of putting a spypoint on it.

For details see the reference pages for the predicates listed below.

6.4.3 Summary of Predicates

- `add_advice/3`
- `check_advice/0`
- `check_advice/1`
- `current_advice/3`
- `nocheck_advice/0`
- `nocheck_advice/1`
- `remove_advice/3`

6.5 The Profiler

The profiling facility makes it possible to analyse the execution of a program and determine where most time was spent, possibly revealing sources of inefficiently. As well as time spent, the profiler maintains information on number of calls, choice points and redos for each predicate. It also records the callers of each predicate, thus building an extensive execution profile.

Note that the profiler shares some low level internal resources with the debugger in the development system and therefore debugging is disallowed when profiling and vice versa.

The profiler is not available under Windows.

6.5.1 Use of the Profiler

The execution of a goal can be profiled using `profile/1`, which takes the goal as its argument. This will turn on the profiler, recording for each invoked predicate the number of calls, choice points created, redos tried and the predicate's callers. Counts accumulated from any previous executions of the profiler will be reset. The goal is then executed with timing information additionally monitored.

Once the goal has completed execution, the results of the execution profile can be seen by calling the `show_profile_results/2` builtin predicate. The first argument to `show_profile_results/2` is the *display mode* and is one of the atoms: `by_time`, `by_calls`, `by_choice_points` or `by_redos`. This determines how the output is sorted and what the percent figure that is printed relates to. For example, if the argument is `by_time` then the results are sorted according to the time spent in the predicates (in descending order) and the percentage figure is the proporation of total execution time spent executing that predicate.

The second argument gives the number of procedures to show information about, thus a value of 10 means that the top ten predicates are printed. The predicate `show_profile_results/1` is equivalent to `show_profile_results/2` with second argument given a value of 10 and similarly `show_profile_results/0` is equivalent to `show_profile_results/1` with the argument `by_time`.

The following example illustrates the output of `show_profile_results/2`:

```
| ?- [chat].
% loading file /opt/quintus/generic/q3.5/demo/chat/chat.qof
% chat.qof loaded in module user, 0.620 sec 1,520 bytes

yes
| ?- profile(hi(questions)).
% The profiler is switched on

...

yes
[profile]
| ?- show_profile_results(by_time, 3).
Proc             Calls ChPts Redos Time  % Caller(proc,cl#,cll#,%)
user:setof/3     227   0     0     2.04  34.0
                                   user:satisfy/1,6,1 152 61.0
                                   user:seto/3,1,1 48 20.0
                                   user:satisfy/1,7,1 27 17.0
user:satisfy/1   35738 36782 14112 0.32  5.3
                                   user:satisfy/1,1,2 13857 43.0
                                   user:satisfy/1,2,1 12137 31.0
                                   user:satisfy/1,1,1 7315 18.0
                                   user:satisfy/1,3,1 1155 6.0
user:inv_map_l/5 4732  4732  3115  0.20  3.3
                                   user:inv_map_l/5,2,1 3115 60.0
                                   user:inv_map/4,5,1 1617 40.0

yes
[profile]
| ?- show_profile_results(by_calls, 3).
Proc             Calls ChPts Redos Time  % Caller(proc,cl#,cll#,%)
user:satisfy/1   35738 36782 14112 0.32  15.3
                                   user:satisfy/1,1,2 13857 43.0
                                   user:satisfy/1,2,1 12137 31.0
                                   user:satisfy/1,1,1 7315 18.0
                                   user:satisfy/1,3,1 1155 6.0
                                   user:satisfy/1,4,1 1044 0.0
                                   user:holds/2,1,1 3 0.0
user:database/1      13616 0     0     0.06  5.8
                                   user:satisfy/1,14,1 13616 100.0
```

The output lists the predicate name, the number of calls to that predicate, number of times a choice point was created, the number of the times the predicate was retried on backtracking and the time (in seconds) spent executing that predicate. The percentage figure depends on the display mode. In the example above 15.3% of the goal calls were to `satisfy/1`, but only 5.3% of the execution time was spent in `satisfy/1`.

Then follows the list of callers, showing for each caller the predicate name and arity, the clause number and the call number within that clause of the call (see Section 18.3.18 [mpg-ref-add_spypoint], page 1038), followed by the number of calls made by this caller and the percentage of execution time attributed to this caller.

Notice in the example that more callers are shown for `satisfy/1` when the profile results are listed `by_calls` than `by_time`. Callers that do not register any time are not listed in the output when displayed `by_time`. Callers are omitted in a similar way for other display modes when the relevant count is zero.

The profiler can be turned off with the `noprofile/0` predicate.

6.5.2 Customized Output

The predicate `get_profile_results/4` returns the profiling information as a list of terms, to enable the customized display of profiling results. The first two arguments of `get_profile_results/4` are the same as for `show_profile_results/2`, the third argument returns a list of `proc/6` terms described below and the final argument returns a total that depends on the display mode given by the first argument — for example, if the display mode is `by_time` then this is the total execution time.

The `proc/6` term `proc(`*`Name`*`,`*`Ncalls`*`,`*`Nchpts`*`,`*`Nredos`*`,`*`Time`*`,`*`Callers`*`)` gives profiling information about one profiled predicate, where *Name* gives the module, name and arity of the predicate; *Ncalls*, *Nchpts*, *Nredos*, *Time* give call, choice point and redo counts and the execution time in milliseconds. The `Callers` argument is a list of `calledby/5` terms of the form `calledby(`*`Time`*`,`*`Calls`*`,`*`Name`*`,`*`ClauseNo`*`,`*`CallNo`*`)` where *Time* is the percentage of time attributed to this caller, *Calls* is the number of calls made by this caller and *Name*, *ClauseNo*, *CallNo* identify the actual caller. For example:

```
| ?- get_profile_results(by_time,3,List,Total).

List = [proc(user:setof/3,227,0,0,1980,
                         [calledby(61,152,user:satisfy/1,6,1),
                          calledby(20,27,user:satisfy/1,7,1),
                          calledby(18,48,user:seto/3,1,1)]),
        proc(user:satisfy/1,35738,36782,14112,260,
                         [calledby(69,13857,user:satisfy/1,1,2),
                          calledby(15,12137,user:satisfy/1,2,1)]),
        proc(user:write/1,2814,0,0,240,
                         [calledby(33,481,user:reply/1,3,1),
                          calledby(25,608,user:replies/1,3,1),
                          calledby(16,562,user:out/1,2,1),
                          calledby(8,203,user:reply/1,2,5),
                          calledby(8,34,user:replies/1,2,3)])],
Total = 6040

[profile]
| ?-
```

6.5.3 Performance

There is a performance penalty of about 20% associated with running a program in profile mode. Data structures needed to maintain profiling information are created on demand the first time a profiled goal is called, so this may affect first-run statistics if the run is relatively short. In this case you may wish to profile the same goal at least a second time to verify the results.

6.5.4 Summary of Predicates

- `profile/0`
- `profile/1`
- `noprofile/0`
- `show_profile_results/0`
- `show_profile_results/1`
- `show_profile_results/2`
- `get_profile_results/4`

7 Glossary

Glossary

abolish To abolish a *predicate* is to *retract* all the predicate's *clauses* and to remove all information about it from the Prolog system, to make it as if that predicate had never existed. *Built-in predicates* cannot be abolished, but user-defined ones always can be, even when *static*.

absolute filename
A name of a file giving the absolute location of that file. Under UNIX and Windows, Quintus Prolog considers filenames beginning with '`/`' or '`~`' absolute. Under Windows, filenames beginning with a letter followed by '`:`' are also considered absolute. All other filenames are considered *relative*.

alphanumeric
An alphanumeric character is any of the lowercase characters from '`a`' to '`z`', the uppercase characters from '`A`' to '`Z`', or the numerals from '`0`' to '`9`'.

ancestor An ancestor of a *goal* is any goal that the system is trying to solve when it calls that goal. The most distant ancestor is the goal that was typed at the top-level prompt.

anonymous
An anonymous *variable* is one that has no unique name, and whose value is therefore inaccessible. An anonymous variable is denoted by an underscore (`_`).

archive file
A file containing an *object code* library that can be statically linked into programs. Sometimes called *static library*. Archive files have an operating system dependent extension, which is:

Windows: '`.lib`' By convention, Quintus Prolog uses the suffix '`s.lib`' (e.g., '`libqps.lib`') since a '`.lib`' is also used for Windows DLL import libraries.

UNIX: samp`{.a}`

argument See *predicate*, *structure*, and *arity*.

arity The arity of a *structure* is its number of *argument*s. For example, the structure `customer(jones, 85)` has an arity of 2.

atom A character sequence used to uniquely denote some entity in the problem domain. A number is *not* an atom. Examples of legal atoms are:

```
hello   *   '#$%'   'New York'   'don''t'
```

Please note: An atom may not start with a capital letter or underscore unless that atom is enclosed in single quotes. Character sequences that include spaces must also be enclosed in single quotes. To include a single quote in an atom, print it twice in succession for each single quote that is to appear. See the Section 8.1.2.4 [ref-syn-trm-ato], page 160 for a complete definition of an atom.

atomic term
: Synonym for *simple term* or *constant*.

backtracking
: The process of reviewing the *goal*s that have been satisfied and attempting to resatisfy these goals by finding alternative solutions.

binding
: The process of assigning a value to a *variable*; used in *unification*.

body
: The body of a *clause* consists of the part of a Prolog clause following the '`:-` ' symbol.

buffer
: A temporary workspace in Emacs that contains a file being edited.

built-in predicate
: A *predicate* that comes with the system and that does not have to be explicitly *consult*ed or *compile*d before it is used.

clause
: A *fact* or a *rule*. A rule comprises a *head* and a *body*. A fact consists of a head only, and is equivalent to a rule with the body `true`.

command
: An instruction for the Prolog system to perform an action involving *side-effects*. If the command is written preceded by a '`:-` ', it will be executed as a *directive*.

compile
: Load a program (or a portion thereof) into Prolog through the compiler. Compiled code runs more quickly than interpreted code, but you cannot debug compiled code in as much detail as interpreted code.

compound term
: See *structure*.

connective
: a logical *term*, or a symbol thereof, that relates components in such a way that the truth or falsity of the resulting statement is determined by the truth or falsity of the components. For example,

```
:-      ;      ,
```

stand for the connectives '`if`', '`or`', and '`and`'.

constant
: An integer (for example: 1, 20, -10), a floating-point number (for example: 12.35), or an *atom* (for example: `'New York'`). Constants are also known as *simple term*s, and are recognized by the Prolog predicate `atomic/1`.

consult
: Load a program (or a portion thereof) into Prolog through the interpreter. Interpreted code runs more slowly than compiled code, but you can debug interpreted code in more detail than compiled code.

creep
: What the debugger does in *trace* mode, also known as single-stepping. It goes to the next *port* of a *procedure* box and prints the *goal*, then prompts you for input. See Section 6.1.1 [dbg-bas-bas], page 113.

cross-reference
: A notation in the text of the manual, pointing to another section of the manual containing related information. In the on-line manual, these are of the form `{`*manual*`(`*Tag*`)}`, as in "see `{`manual(int-man-ove)`}`." Typing the text between the braces into the Prolog system will cause the text of the referenced section to be displayed.

cursor The point on the screen at which typed characters appear. This is usually highlighted by a line or rectangle the size of one space, which may or may not blink.

cut Written as '`!`'. A *built-in predicate* that succeeds when encountered; if *backtracking* should later return to the cut, the *goal* that matched the *head* of the *clause* containing the cut fails immediately.

database The Prolog database comprises all of the *clause*s that have been *load*ed into the Prolog system via `compile/1`, `consult/1`, or that have been asserted, excepting those clauses that have been removed by `retract/1` or `abolish/[1,2]`

debug A mode of *program* execution in which the debugger stops to print the current *goal* only at *procedures* that have *spypoints* set on them (see *trace*).

determinate
A *procedure* is determinate if it can supply only one answer.

directive A directive is a *command* preceded by the prefix operator '`:-` ', whose intuitive meaning is "execute this as a command, but do not print out any variable bindings."

disjunction
A series of *goal*s connected by the *connective* "or" (that is, a series of goals whose principal *operator* is '`|`' or '`;`').

dynamic predicate
A predicate that can be modified while a *program* is running. A *predicate* must explicitly be declared to be dynamic or it must be added to the *database* via one of the assertion predicates.

environment variable
A variable known to the command interpreter environment. Most programs can be controlled to some extent by environment variables. The syntax for setting environment variables in command interpreter dependent. For example, (A) would be appropriate for `csh(1)` and `tcsh(1)`; (B) for `sh(1)`, `bash(1)`, and `ksh(1)`; (C) for Windows `cmd.exe`. Under Windows, it is often preferable to set environment variables globally in the System control panel.

```
% setenv PROLOGINITSIZE 2M                    (A)
% export PROLOGINITSIZE=2M                    (B)
% SET PROLOGINITSIZE=2M                       (C)
```

export A *module* exports a *procedure* by making that procedure *public*, so that other modules can *import* it.

fact (Also called a *unit clause*.) A *clause* with no *conditions*—that is, with an empty *body*. A fact is a statement that a relationship exists between its *arguments*. Some examples, with possible interpretations, are:

```
king(louis, france).   % Louis was king of France.
have_beaks(birds).     % Birds have beaks.
employee(nancy, data_processing, 55000).
                       % Nancy is an employee in the
                       % data processing department.
```

An integer number assigned to a file when it is opened, and then used as a unique identifier in I/O operations.

first-order logic
: A system of logic in which the values of *variables* may range over the data items in the domain. In Prolog these data items are *terms*. For comparison, in zero-order logic (also known as propositional logic) there are no variables, and in second-order logic the values of variables are allowed to range both over data items and over functions and relations.

functor
: The name and *arity* of a *structure*. For example, the structure `foo(a, b)` is said to have "the functor foo of arity two", which is generally written `foo/2`.

garbage collection
: The freeing up of space for computation by making the space occupied by *terms* that are no longer available for use by the Prolog system.

goal
: A *procedure* call. When called, it will either succeed or fail. A goal typed at the top level is called a *query*.

head
: The head of a *clause* is the single *goal* that will be satisfied if the *conditions* in the *body* (if any) are true; the part of a *rule* before the '`:-` ' symbol. The head of a *list* is the first element of the list.

home directory
: Your default directory upon login. Under UNIX, this is the value of the `HOME` environment variable. Under Windows, it is the directory specified by the environment variables `HOMEDRIVE` and `HOMEPATH`. You can ask Quintus Prolog what it considers to be you home directory by typing

```
| ?- absolute_file_name(~, HomeDir).
```

Horn clause
: See *clause*.

import
: *Public procedures* in a *module* can be imported by other modules. Once a procedure has been imported by a module, it can be called as if it were defined in that module.

  There are two kinds of importation: procedure-importation, in which only specified procedures are imported from a module; and module-importation, in which all the predicates made public by a module are imported.

instantiation
: A *variable* is instantiated if it is *bound* to a non-variable *term*; that is, to an *atomic term* (see *constant*) or a *compound term*.

interpret
: Load a *program* or set of *clauses* into Prolog through the interpreter (also known as *consulting*). Interpreted code runs much more slowly than *compiled* code, but more extensive facilities are available for debugging interpreted code.

leap
: What the debugger does in *debug* mode. The debugger shows only the *ports* of *procedures* that have *spypoints* on them. It then prompts you for input, at which time you may leap again to the next spypoint. See Section 6.1.4.2 [dbg-bas-tra-spy], page 117.

leashing Determines how frequently the debugger will stop and prompt you for input when you are *tracing*. A *port* at which the debugger stops is called a "leashed port."

list A list is written as a set of zero or more *term*s between square brackets. If there are no terms in a list, it is said to be empty, and is written as []. In this first set of examples, all members of each list are explicitly stated.

```
[aa, bb,cc]  [X, Y]  [Name]  [[x, y], z]
```

In the second set of examples, only the first several members of each list are explicitly stated, while the rest of the list is represented by a *variable* on the right-hand side of the "rest of" operator, '`|`':

```
[X | Y]  [a, b, c | Y]  [[x, y] | Rest]
```

'`|`' is also known as the "list constructor." The first element of the list to the left of '`|`' is called the *head* of the list. The rest of the list, including the variable following '`|`' (which represents a list of any length), is called the *tail* of the list.

For example,

list	**head**	**tail**
`[X | Y]`	`X`	`Y`
`[a, b, c | y]`	`a`	`[b, c | y]`
`[[X, Y] | Rest]`	`[X, Y]`	`Rest`

load To *compile* or *consult* a Prolog *clause* or set of clauses.

meta-predicate

A meta-predicate is one that calls one or more of its *arguments*; more generally, any *predicate* that needs to assume some *module* in order to operate is called a meta-predicate.

A meta-predicate declaration is a *term* in a *module-file* that is associated with a given *functor*, sharing its *name* and *arity*, but having each of its *arguments* replaced either by one of the mode annotations '`+`', '`-`', '`*`', '`+-`', '`+*`', or by '`:`' or a non-negative integer. '`:`' or a non-negative integer signifies that the corresponding argument requires module name expansion.

mode line The information line at the bottom of each Emacs *window* that is one line long and the width of the screen; often shown in reverse video. The mode line at the bottom of the Prolog window says "Quintus Prolog" plus other information such as the state of the debugger if it is activated. The mode line of the text window(s) states the *buffer*name, the filename, the editor mode ("Prolog" for a file ending in '`.pl`'), and the percentage of the file that precedes the *cursor*.

module A module is a set of *procedures* in a *module-file*. Some procedures in a module are *public*. The default module is `user`.

module-file

A module-file is a file that is headed with a *module* declaration of the form

```
:- module(ModuleName, PublicPredList).
```

which must appear as the first *term* in the file.

multifile predicate
: A *predicate* whose definition is to be spread over more than one file. Such a predicate must be preceded by an explicit multifile declaration in the first file containing *clauses* for it.

name clash
: A name clash occurs when a *module* attempts to define or *import* a *procedure* that it has already defined or imported.

object code
: The machine-executable, as opposed to the human-readable, representation of a *program*.

object file
: A file containing *object code*. Object files have an operating system dependent extension, which is:

  Windows: '`.obj`'

  UNIX: samp{.o}

operator
: A notational convenience that allows you to express any *compound term* in a different format. For example, if "likes" in

```
| ?- likes(sue, cider).
```

  is declared an infix operator, the query above could be written:

```
| ?- sue likes cider.
```

  An operator does not have to be associated with a *predicate*. However, certain *built-in predicates* are declared as operators. For example,

```
| ?- =..(X, Y).
```

  can be written as

```
| ?- X =.. Y.
```

  because `=..` has been declared an infix operator.

  Those predicates that correspond to built-in operators are written using infix notation in the list of built-in predicates at the beginning of the part that contains the reference pages.

  Some built-in operators do *not* correspond to built-in predicates; for example, arithmetic operators. Section 8.1.5.4 [ref-syn-ops-bop], page 169 contains a list of built-in operators.

parent
: The *parent* of the current *goal* is a goal that, in its attempt to obtain a successful solution to itself, is calling the current goal.

port
: One of the four key points of interest in the execution of a Prolog *procedure*. There are four ports: the Call port, representing the initial invocation of the procedure; the Exit Port, representing a successful return from the procedure; the Redo port, representing reinvocation of the procedure through *backtracking* ; and the Fail port, representing an unsuccessful return due to the failure of the initial *goal* of the procedure.

precedence

A number associated with each Prolog *operator*, which is used to disambiguate the structure of the term represented by an expression containing a number of operators. Operators of lower precedence are applied before those of higher precedence; the operator with the highest precedence is considered the principal *functor* of the expression. To disambiguate operators of the same precedence, the associativity type is also necessary. See the syntax chapter (Section 8.1 [ref-syn], page 159).

predicate A *functor* that specifies some relationship existing in the problem domain. For example, `</2` is a built-in *predicate* specifying the relationship of one number being less than another. In contrast, the functor `+/2` is not (normally used as) a predicate.

A predicate is either *built-in* or is implemented by a *procedure*.

procedure A set of *clauses* in which the *head* of each clause has the same *predicate*. For instance, a group of clauses of the following form:

```
connects(san_francisco, oakland, bart_train).
connects(san_francisco, fremont, bart_train).
connects(concord, daly_city, bart_train).
```

is identified as belonging to the procedure `connects/3`.

program A set of *procedures* designed to perform a given task.

public A *procedure* in a *module* is public if it can be *imported* by other modules. The public predicates of a module are listed in the module declaration (see *module-file*).

QOF file a fully general way of storing arbitrary Prolog facts and rules in a form that can be quickly and easily used. QOF files contain a machine independent representation of both compiled and dynamic Prolog predicates. This means they are completely portable between different platforms running Quintus Prolog.

query

– *simple query*: A query is a question put by the user to the Prolog system. A simple query is written as a *goal* followed by a *full-stop* in response to the Prolog system prompt. For example,

```
| ?- father(edward, ralph).
```

refers to the *predicate* `father/2`. If a query has no *variable*s in it, the system will respond either '`yes`' or '`no.`' If a query contains variables, the system will try to find values of those variables for which the query is true. For example,

```
| ?- father(edward, X).
X = ralph
```

After the system has found one answer, the user can direct the system to look for additional answers to the query by typing `;` ⟨RET⟩.

– *compound query*: A compound query consists of two or more simple queries connected by commas. For a compound query to be true, all of its *goal*s

must be true simultaneously. For example, the following compound query will find Ralph's grandfather (G):

```
| ?- father(G, F), father(F, ralph).
```

`F` is a shared *variable* that is constrained by *unification* to have the same value in each of the two subgoals.

quintus-directory
: The root directory of the entire Quintus Prolog file hierarchy. Used by Prolog executables to relocate certain relative file names. Is the value of the `quintus_directory` Prolog flag. See Section 1.3 [int-dir], page 11.

recursion
: The process in which a running *procedure* calls itself, presumably with different *arguments* and for the purpose of solving some subset of the original problem.

region
: The text between the *cursor* and a previously-set mark in an Emacs buffer.

relative filename
: A name of a file giving the location of that file relative to the working directory. See *absolute filename* about differentiating absolute filenames from relative ones.

rule
: A *clause* with one or more *conditions*. For a rule to be true, all of its conditions must also be true. For example,

```
has_stiff_neck(ralph) :-
   hacker(ralph).
```

This rule states that if the individual `ralph` is a hacker, then he must also have a stiff neck. The *constant* `ralph` is replaced in

```
has_stiff_neck(X) :-
   hacker(X).
```

by the *variable* X. X *unifies* with anything, so this rule can be used to prove that any hacker has a stiff neck.

runtime-directory
: A platform-specific directory containing Quintus Prolog executables, object files and the like. Is the value of the `runtime_directory` Prolog flag. See Section 1.3 [int-dir], page 11.

saved-state
: A snapshot of the state of Prolog saved in a file by `save_program/1`, `save_modules/2`, or `save_predicates/2`. `save_program/1` saves the whole Prolog data base, `save_modules/2` and `save_predicates/2` save a list of modules and predicates respectively.

semantics
: The relation between the set of Prolog symbols and their combinations (as Prolog *terms* and *clauses*), and their meanings. Compare *syntax*.

static library
: See *shared object file*.

shared object file
: A file containing *object code* that can be dynamically loaded into programs. Sometimes called *shared library*. Shared object files have an operating system dependent extension, which is:

Windows: '`.dll`'

HPUX: samp{.sl}

other UNIX:
samp{.so}

side-effect A *predicate* that produces a side-effect is one that has any effect on the "outside world" (the user's terminal, a file, etc.), or that changes the Prolog *database*.

simple term
see *constant*.

source code
The human-readable, as opposed to the machine-executable, representation of a *program*.

spypoint A flag placed on a *predicate* by the command `spy/1` and removed by `nospy/1` that tells the debugger to stop execution and allow user interaction at *goals* for that predicate. Any number of predicates can have spypoints set on them.

static library
See *archive file*.

static predicate
A *predicate* that can be modified only by being reloaded via the *consult* or *compile* facility or by being *abolish*ed. (See *dynamic predicate*.)

stream An input/output channel.

structure (Also called a *compound term*.) A structure is a *functor* together with zero or more *argument*s. For example, in the structure

```
father(X)
```

`father/1` is the functor, and X is the first and only argument. The argument to a structure can be another structure, as in

```
father(father(X))
```

syntax The part of Prolog grammar dealing with the way in which symbols are put together to form legal Prolog *terms*. Compare *semantics*.

term A basic data object in Prolog. A term can be a *constant*, a *variable*, or a *structure*.

trace A mode of *program* execution in which the debugger single-steps to the next *port* and prints the *goal*.

unbound A *variable* is unbound if it has not yet been *instantiated*.

unification The process of matching a *goal* with the *head* of a *clause* during the evaluation of a *query*, or of matching arbitrary terms with one another during *program* execution. A goal unifies with the head of a clause if 1) they have the same *functor*, and 2) all of the *argument term*s can be unified. The rules governing the unification of terms are:

- Two *constant*s unify with one another if they are identical.

- A *variable* unifies with a *constant* or a *structure*. As a result of the unification, the variable is *instantiated* to the constant or structure.
- A variable unifies with another variable. As a result of the unification, they become the same variable.
- A structure unifies with another structure if they have the same functor and if all of the arguments can be unified.

unit clause

See *fact*.

variable Logical variable. A logical variable is a name that stands for objects that may or may not be determined at a specific point in a Prolog *program*. When the object for which the variable stands is determined in the Prolog program, the variable becomes instantiated (see *instantiation*). A logical variable may be unified (see *unification*) with a *constant*, a *structure*, or another variable. Variables become uninstantiated when the procedure they occur in backtracks (see *backtracking*) past the point at which they were instantiated.

A variable is written as a single word (with no intervening spaces) beginning either with a capital letter without quotes, or with the character '_'. Examples:

```
X   Y   Z   Name   Position   _c   _305  One_stop
```

volatile Predicate property. The clauses of a volatile predicate are not saved by in QOF files by the Prolog `save` predicates. However, they are saved by `qpc`.

window Under the Emacs interface, a region of the terminal screen. There are two types of window: the Prolog window, of which there is exactly one, and the the text window, of which there are one or more. Each window has a *mode line* at the bottom, and each text window displays the contents of one file.

8 The Prolog Language

8.1 Syntax

8.1.1 Overview

This section describes the syntax of Quintus Prolog.

8.1.2 Terms

8.1.2.1 Overview

The data objects of the language are called *term*s. A term is either a *constant*, a *variable*, or a *compound term*.

A constant is either a *number* (integer or floating-point) or an *atom*. Constants are definite elementary objects, and correspond to proper nouns in natural language.

Variables and compound terms are described in Section 8.1.2.5 [ref-syn-trm-var], page 161, and Section 8.1.3 [ref-syn-cpt], page 161, respectively.

Foreign data types are discussed in Chapter 13 [str], page 655, on the Structs library package.

8.1.2.2 Integers

The printed form of an integer consists of a sequence of digits optionally preceded by a minus sign (‘-’). These are normally interpreted as base 10 integers. It is also possible to enter integers in other bases (1 through 36); this is done by preceding the digit string by the base (in decimal) followed by an apostrophe. If a base greater than 10 is used, the characters A-Z or a-z are used to stand for digits greater than 9.

Examples of valid integer representations are:

```
1   -2345   85923   8'777   16'3F4A   -12'2A9
```

Note that

```
+525
```

is not a valid integer.

A base of zero will return the ASCII code of the (single) character after the quote; for example,

```
0'a = 97
```

8.1.2.3 Floating-point Numbers

A floating-point number (float) consists of a sequence of digits with an embedded decimal point, optionally preceded by a minus sign (-), and optionally followed by an exponent consisting of upper- or lowercase 'E' and a signed base 10 integer. Examples of floats are:

```
1.0   -23.45   187.6E12   -0.0234e15   12.0E-2
```

Note that there must be at least one digit before, and one digit after, the decimal point.

8.1.2.4 Atoms

An atom is identified by its name, which is a sequence of up to 65532 characters (other than the null character). An atom can be written in any of the following forms:

- Any sequence of alphanumeric characters (including '_'), starting with a lowercase letter. Note that an atom may not begin with an underscore.
- Any sequence from the following set of characters (except '/*', which begins a comment):

  ```
  + - * / \ ^ < > = ` ~ : . ? @ # $ &
  ```

- Any sequence of characters delimited by single quotes, such as:

  ```
  'yes'  '%'
  ```

  If the single quote character is included in the sequence it must be written twice, for example:

  ```
  'can''t'  ''''
  ```

  For further details see Section 8.1.8.6 [ref-syn-syn-nte], page 178.
- Any of:

  ```
  ! ; [] {}
  ```

  Note that the bracket pairs are special: '[]' and '{}' are atoms but '[', ']', '{', and '}' are not. The form [*X*] is a special notation for lists (see Section 8.1.3.1 [ref-syn-cpt-lis], page 162), and the form {*X*} is allowed as an alternative to {}(*X*).

Examples of atoms are:

```
a   void   =   :=   'Anything in quotes'   []
```

> WARNING: It is recommended that you do not invent atoms beginning with the character '$', since it is possible that such names may conflict with the names of atoms having special significance for certain built-in predicates.

8.1.2.5 Variables

Variables may be written as any sequence of alphanumeric characters (including '_') beginning with either a capital letter or '_'. For example:

```
X   Value   A   A1   _3   _RESULT
```

If a variable is referred to only once in a clause, it does not need to be named and may be written as an *anonymous* variable, represented by the underline character '_' by itself. Any number of anonymous variables may appear in a clause; they are read as distinct variables. Anonymous variables are not special at runtime.

8.1.2.6 Foreign Terms

Pointers to C data structures can be handled using the Structs package, described in Chapter 13 [str], page 655.

8.1.3 Compound Terms

The structured data objects of Prolog are compound terms. A compound term comprises a *name* (called the *principal functor* of the term) and a sequence of one or more terms called *arguments*. A functor is characterized by its *name*, which is an atom, and its *arity* or number of arguments. For example, the compound term whose principal functor is '`point`' of arity 3, and which has arguments X, Y, and Z, is written

```
point(X, Y, Z)
```

When we need to refer explicitly to a functor we will normally denote it by the form *Name*/*Arity*. Thus, the functor '`point`' of arity 3 is denoted

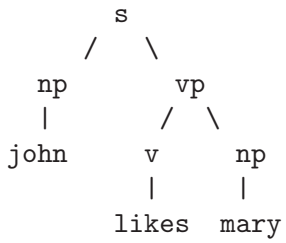
```
point/3
```

Note that a functor of arity 0 is represented as an atom.

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the (compound) term

```
s(np(john), vp(v(likes), np(mary)))
```

would be pictured as the following tree:



The principal functor of this term is `s/2`. Its arguments are also compound terms. In illustration, the principal functor of the first argument is `np/1`.

Sometimes it is convenient to write certain functors as *operators*; binary functors (that is, functors of two arguments) may be declared as *infix* operators, and unary functors (that is, functors of one argument) may be declared as either *prefix* or *postfix* operators. Thus it is possible to write

```
X+Y     P;Q     X<Y      +X     P;
```

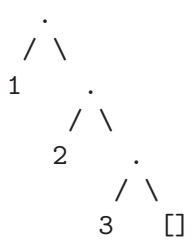
as optional alternatives to

```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

The use of operators is described fully in Section 8.1.5 [ref-syn-ops], page 165.

8.1.3.1 Lists

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of Lisp: a list is either the atom `[]`, representing the empty list, or else a compound term with functor `.` and two arguments, which are the head and tail of the list respectively, where the tail of a list is another list. Thus a list of the first three natural numbers is the structure



which could be written using the standard syntax, as (A) but which is normally written in a special list notation, as (B). Two examples of this list notation, as used when the tail of a list is a variable, are (C), which represent the structure in (D).

```
.(1,.(2,.(3,[])))                                    (A)
```

```
[1,2,3]                                              (B)
```

```
[X|L]      [a,b|L]                                              (C)

            .                  .
           / \                / \
          X     L            a     .
                                    / \
                                   b     L                      (D)
```

Note that the notation [X|L] does not add any new power to the language; it simply improves readability. These examples could be written equally well as (E).

```
.(X,L) .(a,.(b,L))                                              (E)
```

8.1.3.2 Strings As Lists

For convenience, a further notational variant is allowed for lists of integers that correspond to ASCII character codes. Lists written in this notation are called *strings*. For example,

```
"Humpty-Dumpty"
```

represents exactly the same list as

```
[72,117,109,112,116,121,45,68,117,109,112,116,121]
```

8.1.4 Character Escaping

The character escaping facility allows escape sequences to occur within strings and quoted atoms, so that programmers can put non-printable characters in atoms and strings and still be able to see what they are doing. This facility can be switched on and off by the commands:

```
| ?- prolog_flag(character_escapes, _, on).

| ?- prolog_flag(character_escapes, _, off).
```

See Section 8.10.1 [ref-lps-ove], page 245, for a description of `prolog_flag/3`. Character escaping is off by default.

Strings or quoted atoms containing the following escape sequences can occur in terms obtained by `read/[1,2]`, `compile/1`, and so on.

The 0' notation for the integer code of a character is also affected by character escaping.

With character escaping turned on, the only things that can occur in a string or quoted atom are the characters with ASCII codes 9 (horizontal tab), 32 (space), 33 through 126 (non-layout characters), or one of the following escape sequences:

The escape sequence: **Is converted to:**

‘`\b`’ backspace (ASCII 8)

‘`\t`’ horizontal tab (ASCII 9)

‘`\n`’ new line (ASCII 10)

‘`\v`’ vertical tab (ASCII 11)

‘`\f`’ form feed (ASCII 12)

‘`\r`’ carriage return (ASCII 13)

‘`\e`’ escape (ASCII 27)

‘`\d`’ delete (ASCII 127)

‘`\a`’ alarm = BEL (ASCII 7)

‘`\xCD`’ the ASCII character with code ‘`CD`’ (hexadecimal number)

‘`\`*octal string*’
the ASCII character with code *octal string* base 8, where <octal string> is either 1, 2, or 3 octal digits

‘`\^<control char>`’
the ASCII character whose code is the ASCII code of *control char* mod 32. ‘`\^?`’ is another name for ‘`\d`’.

‘`\`*layout char*’
no character, where *layout char* is a character with ASCII code =< 32 or ASCII code >= 127 (thus a newline or form-feed in a string or quoted atom can be ignored by immediately preceding it with a backslash)

‘`\c`’ no character; also, all characters up to, but not including, the next non-layout character are ignored

‘`\`*other*’ the character *other*, where *other* is any character not predefined here; thus ‘`\\`’ should be used to insert one backslash

It is an error if an escape sequence or ASCII character that is not defined above occurs in a string or quoted atom. For instance, an ordinary newline in an atom or string is regarded as an error when character escapes are on. This allows the syntax error of a missing closing quote to be caught much earlier, but it has the problem that some old programs will break (which is why character_escapses are off by default).

With character escaping turned on, the escape sequence ‘`\'`’ represents the same character as the sequence ‘`''`’ within a quoted atom, namely one single quote. Similarly, with character escaping turned on, the escape sequence ‘`\"`’ represents the same character as the sequence ‘`""`’ within a string, namely one double quote.

The escape sequence ‘`\c`’ (c for continue) is useful when formatting a string for readability. For example, the atom (A), is equivalent to (B):

```
          '!Ruth \c                                                    (A)
            Gehrig \c
            Cobb \c
            Williams!'

    '!Ruth Gehrig Cobb Williams!'                                      (B)
```

The following sequence denotes the integer 9:

```
0'\t
```

8.1.5 Operators and their Built-in Predicates

8.1.5.1 Overview

Operators in Prolog are simply a notational convenience. For example, '`+`' is an infix operator, so

```
2 + 1
```

is an alternative way of writing the term `+(2, 1)`. That is, `2 + 1` represents the data structure

```
    +
   / \
  2   1
```

and *not* the number 3. (The addition would only be performed if the structure were passed as an argument to an appropriate procedure, such as `is/2`; see Section 8.8.2 [ref-ari-eae], page 234.)

Prolog syntax allows operators of three kinds: *infix*, *prefix*, and *postfix*. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator follows its single argument.

Each operator has a *precedence*, which is a number from 1 to 1200. The precedence is used to disambiguate expressions in which the structure of the term denoted is not made explicit through the use of parentheses. The general rule is that the operator with the *highest* precedence is the principal functor. Thus if '`+`' has a higher precedence than '`/`', then

```
a+b/c      a+(b/c)
```

are equivalent, and denote the term `+(a,/(b,c))`. Note that the infix form of the term `/(+(a,b),c)` must be written with explicit parentheses:

```
(a+b)/c
```

If there are two operators in the expression having the same highest precedence, the ambiguity must be resolved from the *types* of the operators. The possible types for an infix operator are

- `xfx`
- `xfy`
- `yfx`

Operators of type '`xfx`' are not associative: it is required that both of the arguments of the operator be subexpressions of *lower* precedence than the operator itself; that is, the principal functor of each subexpression must be of lower precedence, unless the subexpression is written in parentheses (which gives it zero precedence).

Operators of type '`xfy`' are right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator. Left-associative operators (type '`yfx`') are the other way around.

An atom named *Name* is declared as an operator of type *Type* and precedence *Precedence* by the command

```
:-op(Precedence, Type, Name).
```

An operator declaration can be cancelled by redeclaring the *Name* with the same *Type*, but *Precedence* 0.

The argument *Name* can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds: infix, prefix, or postfix. An operator of any kind may be redefined by a new declaration of the same kind. Declarations for all these *built-in operators* can be found in Section 8.1.5.4 [ref-syn-ops-bop], page 169.

For example, the built-in operators '`+`' and '`-`' are as if they had been declared by (A) so that (B) is valid syntax, and means (C) or pictorially (D).

```
:-op(500, yfx, [+,-]).                                    (A)

a-b+c                                                      (B)

(a-b)+c                                                    (C)

            +
          /   \
         -     c
        / \
       a   b                                               (D)
```

The list functor `./2` is not a standard operator, but we could declare it to be (E) then (F) would represent the structure (G).

```
:-op(600, xfy, .).                                    (E)

a.b.c                                                 (F)

          .
         / \
        a   .
           / \
          b   c                                       (G)
```

Contrasting this with the diagram above for a-b+c shows the difference between '`yfx`' operators where the tree grows to the left, and '`xfy`' operators where it grows to the right. The tree cannot grow at all for '`xfx`' operators; it is simply illegal to combine '`xfx`' operators having equal precedences in this way.

The possible types for a prefix operator are:

- `fx`
- `fy`

and for a postfix operator they are:

`xf`

`yf`

If these precedence and associativity rules seem rather complex, remember that you can always use parentheses when in any doubt.

8.1.5.2 Manipulating and Inspecting Operators

To add or remove an operator, use `op(+Precedence, +Type, +Name)`. `op/3` declares the atom *Name* to be an operator of the stated *Type* and *Precedence*. If *Precedence* is 0, then the operator properties of *Name* (if any) are cancelled.

To examine the set of operators currently in force, use `current_op(*Precedence, *Type, *Name)`.

8.1.5.3 Syntax Restrictions

Note carefully the following syntax restrictions, which serve to remove potential ambiguities associated with prefix operators.

1. The arguments of a compound term written in standard syntax must be expressions of precedence *less than* 1000. Thus it is necessary to write the expression `P:-Q` in parentheses

```
assert((P:-Q))
```

because the precedence of the infix operator '`:-`', and hence of the expression `P:-Q`, is 1200. Enclosing the expression in parentheses reduces its precedence to 0.

2. Similarly, the elements of a list written in standard syntax must be expressions of precedence *less than* 1000. Thus it is necessary to write the expression `P->Q` in parentheses

```
[(P->Q)]
```

because the precedence of the infix operator '`->`', and hence of the expression `P->Q`, is 1050. Enclosing the expression in parentheses reduces its precedence to 0.

3. In a term written in standard syntax, the principal functor and its following '`(`' must *not* be separated by any intervening spaces, newlines, or other characters. Thus

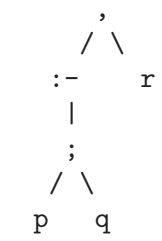
```
point (X,Y,Z)
```

is invalid syntax.

4. If the argument of a prefix operator starts with a '`(`', this '`(`' must be separated from the operator by at least one space or other layout character. Thus

```
:-(p;q),r.
```

(where '`:-`' is the prefix operator) is invalid syntax. The system would try to interpret it as the structure:

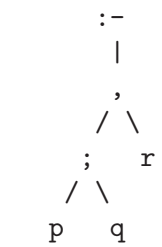


That is, it would take '`:-`' to be a functor of arity 1. However, since the arguments of a functor are required to be expressions of precedence less than 1000, this interpretation would fail as soon as the '`;`' (precedence 1100) were encountered.

In contrast, the term:

```
:- (p;q),r.
```

is valid syntax and represents the following structure:



5. If a prefix operator is written without an argument (as an ordinary atom), the atom is treated as an expression of the same precedence as the prefix operator, and must therefore be written in parentheses where necessary. Thus the parentheses are necessary in

```
X = (?-)
```

since the precedence of '`?-`' is 1200.

8.1.5.4 Built-in Operators

```
:-op( 1200, xfx, [ :-, --> ])
:-op( 1200,  fx, [ :-, ?- ])
:-op( 1150,  fx, [ dynamic, multifile,
                   meta_predicate, initialization, volatile ])
:-op( 1100, xfy, [ ; ])
:-op( 1050, xfy, [ -> ])
:-op( 1000, xfy, [ ',' ])
:-op(  900,  fy, [ \+, spy, nospy ])
:-op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                    =:=, =\=, <, >, =<, >= ])
:-op(  600, xfy, [:])
:-op(  500, yfx, [ +, -, \/, /\ ])
:-op(  500,  fx, [ +, - ])
:-op(  400, yfx, [ /, //, *, <<, >> div  ])
:-op(  300, xfx, [ mod ])
:-op(  200, xfy, [ ^ ])
```

Two additional operators are provided solely for compatibility with other Prologs:

```
:- op(1150, fx, [mode, public])
```

8.1.6 Commenting

Comments have no effect on the execution of a program, but they are very useful for making programs more comprehensible. Two forms of comments are allowed:

1. The character '`%`' followed by any sequence of characters up to the end of the line.
2. The symbol '`/*`' followed by any sequence of characters (including newlines) up to the symbol '`*/`'.

8.1.7 Predicate Specifications

A predicate is uniquely identified by its module (not always specified), name and arity (number of arguments). In Quintus Prolog these are the ways of specifying a predicate as an argument of a predicate:

1. The form *Module*:*Name*(*Term1*,*Term2*, . . . ,*TermN*) is called the *skeletal* predicate specification. It identifies the predicate *Name* of arity *N* in module *Module*. It is required by predicates when the specification was likely to have been obtained from predicates such as `clause/[2,3]`. This is the case when one is manipulating Prolog programs themselves.

   Module is optional; if omitted, the predicate is assumed to be in the source module.

When the skeletal specification is used as an *input* argument, the values of *Term1*,*Term2*, ...,*TermN* are not significant; they only serve as place-holders for determining the arity of *Name*. For example,

```
| ?- predicate_property(put(97), P1),
     predicate_property(put(98), P2).

P1 = P2 = built_in ;

no
```

When the skeletal specification is used as an *output* argument, *Module*:*Name* (*Term1*,*Term2*, ...,*TermN*) is made to be the most general term with name *Name* and arity *N* (that is, *Term1*, *Term2*, ..., *TermN* are each made to be variables, distinct from each other and any others in the system). For example,

```
| ?- compile(user).
| foo(1, 2, 3).
| ^D
% user compiled, 0.100 sec 196 bytes

yes
| ?- source_file(X, user).

X = foo(_224,_225,_226) ;

no
| ?- source_file(foo(7,8,9), user).

yes
```

2.

The *Module*:*Name*/*Arity* form is an alternative representation to the skeletal form. *Arity* can be a single arity, or a list of arities and/or arity ranges.

Module is optional; if omitted, the predicate is assumed to be in the current module. For example,

> `prog1:foo/1` specifies predicate `foo`, arity 1 in module `prog1`. `foo/1` specifies predicate `foo` of arity 1 in the current module.

Notes:

a. The form *Name*/[*Arity*] is used only by some library packages and for demonstrative purposes in this manual. Currently, it is not used by any supported built-in predicates.

b. The *Module*:*Name*/[*Arities*] form is required by declarations that take a predicate specification (or a list of predicate specifications) as an argument. For most predicates, this form requires fewer characters, which is desirable because these specifications will likely be typed by the user.

`abolish/2` is the only predicate that does not use either of the above specifications. Its first argument is the *Name* of the predicate and the second argument is the *Arity*. For consistency, it is recommended that `abolish/1` be used instead.

The following predicate can be used to convert between the *Name/Arity* specification and the skeletal specification, or to verify that two specifications identify the same predicate.

```
predicate_specification(NameAritySpec, SkeletalSpec) :-
       (nonvar(NameAritySpec) ; nonvar(SkeletalSpec)),
        !,
        NameAritySpec = Name/Arity,
        functor(SkeletalSpec, Name, Arity),
        atom(Name).
```

8.1.8 Formal Syntax

8.1.8.1 Overview

A Prolog program consists of a sequence of *sentences*. Each sentence is a Prolog *term*. How sentences are interpreted as terms is defined in Section 8.1.8.3 [ref-syn-syn-sen], page 172, below. Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the functor `:-/2` could be written in standard prefix notation instead of as the usual infix operator.

Terms are written as sequences of *tokens*. Tokens are sequences of characters, which are treated as separate symbols. Tokens include the symbols for variables, constants, and functors, as well as punctuation characters such as parentheses and commas.

The interpretation of sequences of tokens as terms is defined in Section 8.1.8.4 [ref-syn-syn-trm], page 173. Each list of tokens that is read in (for interpretation as a term or sentence) must be terminated by a

full-stop (a period followed by a layout character such as newline or space) token. Two tokens must be separated by a *space* if they could otherwise be interpreted as a single token. Both spaces and *comment*s are ignored when interpreting the token list as a term. A comment may appear at any point in a token list (separated from other tokens by spaces where necessary).

The interpretation of tokens as sequences of characters is defined on Section 8.1.8.5 [ref-syn-syn-tok], page 175. The next section describes the notation used in the formal definition of Prolog syntax.

8.1.8.2 Notation

- Syntactic categories (or *nonterminals*) are printed in italics, for example *query*. De-

pending on the section, a category may represent a class of either terms, token lists, or character strings.

- A syntactic rule takes the general form

```
C --> F1
    | F2
    | F3
    .
    .
    .
```

  which states that an entity of category *C* may take any of the alternative forms *F1*, *F2*, or *F3*.
- Certain definitions and restrictions are given in ordinary English, enclosed in braces (‘`{}`’).
- A category written as ‘`C...`’ denotes a sequence of one or more *C*s.
- A category written as ‘`?C`’ denotes an optional *C*. Therefore ‘`?C...`’ denotes a sequence of zero or more *C*s.
- A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables in the form of italicized capital letters. The meaning of such rules should be clear from analogy with the definite clause grammars described in Section 8.16 [ref-gru], page 298.
- In Section 8.1.8.4 [ref-syn-syn-trm], page 173, particular tokens of the category *Name* (a name beginning with a capital letter) are written as quoted atoms, while tokens that are individual punctuation characters are written literally.

8.1.8.3 Syntax of Sentences as Terms

sentence	`-->` *clause* `|` *directive* `|` *grammar-rule*	
clause	`-->` *non-unit-clause* `|` *unit-clause*	
directive	`-->` *command* `|` *query*	
non-unit-clause	`-->` *head* `:-` *goals*	
unit-clause	`-->` *head*	{where *head* is not otherwise a *sentence*}
command	`-->` `:-` *goals*	

query	--> ?- *goals*	
head	--> *term*	{where *term* is not a *number* or a *variable*}
goals	--> *goals* , *goals* | *goals* -> *goals* ; *goals* | *goals* -> *goals* | *goals* ; *goals* | *goal*	
goal	--> *term*	{where *term* is not a *number* and is not otherwise a *goals*}
grammar-rule	--> *gr-head* --> *gr-body*	
gr-head	--> *nonterminal* | *nonterminal* , *terminals*	
gr-body	--> *gr-body* , *gr-body* | *gr-body* ; *gr-body* | *gr-body* -> *gr-body* ; *gr-body* | *gr-body* -> *gr-body* | *nonterminal* | *terminals* | *gr-condition*	
nonterminal	--> *term*	{where *term* is not a *number* or *variable* and is not otherwise a *gr-body*}
terminals	--> *list* | *string*	
gr-condition	--> { *goals* }	

8.1.8.4 Syntax of Terms as Tokens

term-read-in	--> *subterm(1200) full-stop*	
subterm(N)	--> *term(M)*	{where *M* is less than or equal to *N*}

term(N)	--> *op(N,fx)*	
	| *op(N,fy)*	
	| *op(N,fx) subterm(N-1)*	{except the case '-' *number*}
		{if *subterm* starts with a '(', *op* must be followed by a *space*}
	| *op(N,fy) subterm(N)*	{if *subterm* starts with a '(', *op* must be followed by a *space*}
	| *subterm(N-1) op(N,xfx) subterm(N-1)*	
	| *subterm(N-1) op(N,xfy) subterm(N)*	
	| *subterm(N) op(N,yfx) subterm(N-1)*	
	| *subterm(N-1) op(N,xf)*	
	| *subterm(N) op(N,yf)*	
term(1000)	--> *subterm(999)* , *subterm(1000)*	
term(0)	--> *functor* (*arguments*)	
		{provided there is no *space* between *functor* and the '('}
	| (*subterm(1200)*)	
	| { *subterm(1200)* }	
	| *list*	
	| *string*	
	| *constant*	
	| *variable*	
op(N,T)	--> *name*	{where *name* has been declared as an operator of type *T* and precedence *N*}
arguments	--> *subterm(999)*	
	| *subterm(999)* , *arguments*	
list	--> []	
	| [*listexpr*]	
listexpr	--> *subterm(999)*	
	| *subterm(999)* , *listexpr*	
	| *subterm(999)* | *subterm(999)*	
constant	--> *atom* | *number*	
number	--> *integer* | *float*	

atom	--> *name*	{where *name* is not a prefix operator}
integer	--> *natural-number* | - *natural-number*	
float	--> *unsigned-float* | - *unsigned-float*	
functor	--> *name*	

8.1.8.5 Syntax of Tokens as Character Strings

token	--> *name* | *natural-number* | *unsigned-float* | *variable* | *string* | *punctuation-char* | *space* | *comment* | *full-stop*	
name	--> *quoted-name* | *word* | *symbol* | *solo-char* | [*?layout-char...*] | { *?layout-char...* }	
quoted-name	--> ' *?quoted-item...* '	
quoted-item	--> *char* | '' | \ *escape-sequence*	{other than '' or '\'} {unless character escapes have been switched off}
word	--> *small-letter ?alpha...*	
symbol	--> *symbol-char...*	{except in the case of a *full-stop* or where the first 2 chars are '/*' }
natural-number	--> *digit...*	

	| *base* ' *alphanumeric*. . .	{where each *alphanumeric* must be less than *base*; count 'a' as 10, 'b' as 11, etc.}
	| *zero* ' *quoted-item*	{This yields the ASCII equivalent of *quoted-item*}
base	--> *digit*. . .	{must be in the range 0..36}
zero	--> 0	
unsigned-float	--> *simple-float* | *simple-float E exponent*	
simple-float	--> *digit*. . . *decimal-point digit*. . .	
decimal-point	--> .	
E	--> E | e	
exponent	--> *digit*. . . | - *digit*. . . | + *digit*. . .	
variable	--> *underline ?alpha*. . .	
variable	--> *capital-letter ?alpha*..	
string	--> " *?string-item*. . . "	
string-item	--> *char*	{other than '"' or '\'}
	| ""	
	| \ *escape-sequence*	{unless character escapes have been switched off}
escape-sequence	-->b	{backspace, character code 8}
	| t	{horizontal tab, character code 9}
	| n	{newline, character code 10}
	| v	{vertical tab, character code 11}
	| f	{form feed, character code 12}
	| r	{carriage return, character code 13}
	| e	{escape, character code 27}
	| d	{delete, character code 127}
	| ^?	{delete, character code 127}
	| a	{alarm, character code 7}
	| x*CD*	{character code hex *CD*, 2 digits}
	| *Oct*	{character code octal *Oct*, up to 3 digits}

	`|` `^`*letter*	{the control character *letter* mod 32}
	`|` `c`*?layout-char. . .*	{ignored}
	`|` *layout-char*	{ignored}
	`|` *char*	{other than the above, represents itself}
space	`-->` *layout-char. . .*	
comment	`-->` `/*` *?char. . .* `*/`	{where *?char. . .* must not contain '`*/`' }
	`|` `%` *rest-of-line*	
rest-of-line	`-->` *newline*	
	`|` *?not-end-of-line. . .* *newline*	
not-end-of-line	`-->` {any character except *newline* }	
newline	`-->` {ASCII code 10}	
full-stop	`-->` `.` *layout-char*	
char	`-->` *layout-char*	
	`|` *alpha*	
	`|` *symbol-char*	
	`|` *solo-char*	
	`|` *punctuation-char*	
	`|` *quote-char*	
layout-char	`-->` {ASCII codes 1..32 and 127 — includes space, tab, newline, and del}	
alpha	`-->` *alphanumeric* `|` *underline*	
alphanumeric	`-->` *letter* `|` *digit*	
letter	`-->` *capital-letter* `|` *small-letter*	
capital-letter	`-->` `A | B | C | D | E | F | G | H`	
	`| I | J | K | L | M | N | O | P | Q`	
	`| R | S | T | U | V | W | X | Y | Z`	
small-letter	`-->` `a | b | c | d | e | f | g | h`	
	`| i | j | k | l | m | n | o | p | q`	
	`| r | s | t | u | v | w | x | y | z`	

digit `--> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

symbol-char `--> + | - | * | / | \ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | $ | &`

solo-char `--> ; | !`

punctuation-char `--> ( | ) | [ | ] | { | } | , | | | %`

quote-char `--> ' | "`

underline `--> _`

8.1.8.6 Notes

1. The expression of precedence 1000 (that is, belonging to syntactic category *term(1000)*) that is written

   ```
   X, Y
   ```

   denotes the term

   ```
   ','(X, Y)
   ```

   in standard syntax.
2. The parenthesized expression (belonging to syntactic category *term(0)*)

   ```
   (X)
   ```

   denotes simply the term *X*.
3. The curly-bracketed expression (belonging to syntactic category *term(0)*)

   ```
   {X}
   ```

   denotes the term

   ```
   '{}'(X)
   ```

   in standard syntax.
4. Note that, for example, `-3` denotes an integer, whereas `-(3)` denotes a compound term of which the principal functor is `-/1`.
5. The double quote character '`"`' within a string must be written twice for every time it is to appear in the string. That is,

   ```
   """"
   ```

   represents a string of one double quote character only. Similarly, for the single quote character within a quoted atom,

   ```
   ''''
   ```

   represents an atom whose printed representation is one single quote character.

8.1.9 Summary of Predicates

Detailed information is found in the reference pages for the following:

- `current_op/3`
- `op/3`

8.2 Semantics

This section gives an informal description of the semantics of Quintus Prolog.

8.2.1 Programs

A fundamental unit of a logic program is the *goal* or *procedure call* for example:

```
gives(tom, apple, teacher)

reverse([1,2,3], L)

 X < Y
```

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The principal functor of a goal is called a *predicate*. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic *program* consists simply of a sequence of statements called *sentences*, which are analogous to sentences in natural language.

A sentence comprises a *head* and a *body*. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (it may be empty). If the head is not empty, the sentence is called a *clause*.

If the body of a clause is empty, the clause is called a *unit clause*, and is written in the form (A) where P is the head goal. We interpret this *declaratively* as (B) and *procedurally* as (C).

P. (A)

"P is true." (B)

"Goal P is satisfied." (C)

If the body of a clause is non-empty, the clause is called a *non-unit clause*, and is written in the form (D) where P is the head goal and Q, R, and S are the goals that make up the body. We can read such a clause either declaratively as (E) or procedurally as (F).

P :- Q, R, S. (D)

"P is true if Q and R and S are true." (E)

"To satisfy goal P, satisfy goals Q, R, and S." (F)

A sentence with an empty head is called a *directive*, of which the most important kind is called a *query* and is written in the form (G) Such a query is read declaratively as (H), and procedurally as (I).

?- P, Q. (G)

"Are P and Q true?" (H)

"Satisfy goals P and Q." (I)

Sentences generally contain variables. A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writable storage location as in most programming languages; rather it is a local name for some data object, like the variable of pure Lisp. Note that variables in different sentences are completely independent, even if they have the same name — the *lexical scope* of a variable is limited to a single sentence. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

```
employed(X) :- employs(Y, X).
```
"Any X is employed if any Y employs X."

"To find whether a person X is employed, find whether any Y employs X."

```
derivative(X, X, 1).
```
"For any X, the derivative of X with respect to X is 1."

"The goal of finding a derivative for the expression X with respect to X itself is satisfied by the result 1."

```
?- ungulate(X), aquatic(X).
```
"Is it true, for any X, that X is an ungulate and X is aquatic?"

"Find an X that is both an ungulate and aquatic."

In any program, the *procedure* for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the

procedure for a predicate `concatenate` of three arguments might well consist of the two clauses shown in (J) where `concatenate(L1, L2, L3)` means "the list L1 concatenated with the list L2 is the list L3".

```
concatenate([], L, L).                                    (J)
concatenate([X|L1], L2, [X|L3]) :-
            concatenate(L1, L2, L3).                      (K)
```

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form *Name/Arity* is used, for example `concatenate/3`.

8.2.2 Types of Predicates Supplied with Quintus Prolog

Certain predicates are predefined by the Prolog system. Most of these cannot be changed or retracted. Such predicates are called *built-in predicates*.

Certain ones, however, can be modified or totally redefined. These are the hook predicates and the redefined predicates used in embedding.

8.2.2.1 Hook Predicates

Hook predicates are called by the system. They enable you to modify Quintus Prolog's behavior. They are either undefined by default (like `portray/1` and `message_hook/3`) or else they have a simple default definition that is dynamic and/or multifile (like `file_search_path/1` and `library_directory/1`, which are multifile by default).

If they do have a default definition, a definition provided by the user overrides it within the module where it is redefined. The idea of a hook predicate is that its clauses are independent of each other, and it makes sense to spread their definitions over several files (which may be written by different people).

8.2.2.2 Redefinable Predicates

Redefinable predicates exist to enable you to embed Prolog code within a program in another language. They have default definitions that are fairly complex. Source is provided for them. They are all in modules beginning with '`QU`'. Sophisticated users may wish to provide alternative definitions of these modules. You can redefine embeddable predicates at run-time too, by simply compiling new versions of the QU_ module-file (see Section 10.2 [fli-emb], page 365).

The key distinction is that it only makes sense to redefine embeddable predicates totally and globally. Hook predicates, on the other hand, can be extended piecemeal, and need not have any definition at all.

8.2.3 Disjunction

As we have seen, the goals in the body of a sentence are linked by the operator '`,`', which can be interpreted as conjunction (`and`). It is sometimes convenient to use an additional operator '`|`', standing for disjunction (`or`). (The precedence of '`|`' is such that it dominates '`,`' but is dominated by '`:-`'.) An example is the clause (A), which can be read as (B).

```
grandfather(X, Z) :-
     (   mother(X, Y)
     |   father(X, Y)
     ),
     father(Y, Z).                                                    (A)
```

"For any X, Y, and Z,
X has Z as a grandfather if
either the mother of X is Y
or the father of X is Y,
and the father of Y is Z." (B)

Such uses of disjunction can usually be eliminated by defining an extra predicate. For instance, (A) is equivalent to (C)

```
grandfather(X, Z) :- parent(X, Y), father(Y, Z).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).                                         (C)
```

Therefore, disjunction will not be mentioned further in the following more formal description of the semantics of clauses.

For historical reasons, the token '`|`', when used outside a list, is actually an alias for '`;`'. The aliasing is performed when terms are read in, so that (D) is read as if it were (E) thus you can use '`;`' instead of '`|`' for disjunction if you like.

```
a :- b | c.                                                           (D)
```

```
a :- b ; c.                                                           (E)
```

Note the double use of the '`.`' character. Here it is used as a sentence terminator, while in other instances it may be used in a string of symbols that make up an atom (for example, the list functor '`.`'). The rule used to disambiguate terms is that a '`.`' followed by a *layout-character* is regarded as the sentence terminator `full-stop`, where a layout-character is defined to be any character less than or equal to ASCII 32 (this includes space, tab, newline, and all control characters).

8.2.4 Declarative and Procedural Semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However, it is useful to have a precise definition. The *declarative semantics* of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows:

> A goal is *true* if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an *instance* of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the procedure for `concatenate/3`, declared in Section 8.2.1 [ref-sem-pro], page 179, then the declarative semantics tells us that (A) is true, because this goal is the head of a certain instance of the second clause (K) for `concatenate/3`, namely (B), and we know that the only goal in the body of this clause instance is true, because it is an instance of the unit clause that is the first clause for `concatenate/3`.

```
concatenate([a], [b], [a,b])

concatenate([a], [b], [a,b]):-
   concatenate([], [b], [b]).
```

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the *procedural semantics* that Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute his program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here is an informal definition of the procedural semantics.

> To *execute* a goal, the system searches forwards from the beginning of the program for the first clause whose head *matches* or *unifies* with the goal. The *unification* process (see "A Machine-Oriented Logic Based on the Resolution Principle" by J.A. Robinson, *Journal of the ACM* 12:23-44, January 1965) finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it *backtracks*; that is, it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal that activated the rejected clause, and tries to find a subsequent clause that also matches the goal.

For example, if we execute the goal expressed by the query (A) we find that it matches the head of the second clause for `concatenate/3`, with `X` instantiated to `[a|X1]`. The new

variable `X1` is constrained by the new goal produced, which is the recursive procedure call (B) and this goal matches the second clause, instantiating `X1` to `[b|X2]`, and yielding the new goal (C).

```
| ?- concatenate(X, Y, [a,b]).                                    (A)

concatenate(X1, Y, [b])                                           (B)

concatenate(X2, Y, [])                                            (C)
```

Now this goal will only match the first clause, instantiating both `X2` and `Y` to `[]`. Since there are no further goals to be executed, we have a solution

```
X = [a,b]
Y = []
```

That is, the following is a true instance of the original goal:

```
concatenate([a,b], [], [a,b])
```

If this solution is rejected, backtracking will generate the further solutions

```
X = [a]
Y = [b]

X = []
Y = [a,b]
```

in that order, by re-matching goals already solved once using the first clause of `concatenate/3`, against the second clause.

8.2.5 The Cut

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the *cut*, written '`!`'. It is inserted in the program just like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the cut is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the *parent goal*, that goal that matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation *commits* the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded. The goals thus rendered *determinate* are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals that were executed during the execution of those preceding goals.

For example, the procedure

```
member(X, [X|L]).
member(X, [Y|L]) :-
   member(X, L).
```

can be used to test whether a given term is in a list:

```
| ?- member(b, [a,b,c]).
```

returns the answer 'yes'. The procedure can also be used to extract elements from a list, as in

```
| ?- member(X, [d,e,f]).
```

With backtracking this will successively return each element of the list. Now suppose that the first clause had been written instead:

```
member(X, [X|L]) :- !.
```

In this case, the second call above would extract only the first element of the list ('d'). On backtracking, the cut would immediately fail the entire procedure.

Another example:

```
x :- p, !, q.
x :- r.
```

This is analogous to "if p then q else r" in an Algol-like language.

Note that a cut discards all the alternatives subsequent to the parent goal, even when the cut appears within a disjunction. This means that the normal method for eliminating a disjunction — by defining an extra predicate — cannot be applied to a disjunction containing a cut.

8.2.6 Occur Check

Prolog's unification does not have an *occur check*; that is, when unifying a variable against a term, the system does not check to see if the variable occurs in the term. When the variable occurs in the term, unification should fail, but the absence of the check means that the unification succeeds, producing a `circular term`. Trying to print a circular term, or trying to unify circular terms, will cause a loop. (You can always get out of a loop by typing `^c` followed by an `a` for abort.)

The absence of the occur check is not a bug or a design oversight, but a conscious design decision. The reason for this decision is that unification with the occur check is at best linear on the sum of the sizes of the terms being unified, whereas unification without the occur check is linear on the size of the smallest of the terms being unified. For any programming language to be practical, basic operations should take constant time. Unification against a variable may be thought of as the basic operation of Prolog, and this can take constant time

only if the occur check is omitted. Thus the absence of an occur check is essential to Prolog's practicality as a programming language. The inconvenience caused by this restriction is, in practice, very slight. Furthermore, the functionality is available as `library(occurs)`.

8.2.7 Control

P,Q	prove P and Q
P;Q	prove P or Q
`!`	cut any choices taken in the current procedure
`call(P)`	prove (execute) P
`\+` P	goal P is not provable
P`->`Q;R	if P succeeds, prove Q; if not, prove R
P`->`Q	if P succeeds, prove Q; if not, fail
`true`	succeed
`otherwise`	same as `true`
`fail`	fail (start backtracking)
`false`	same as `fail`
`repeat`	succeed repeatedly on backtracking

8.3 Invoking Prolog

8.3.1 Prolog Command Line Argument Handling

There are three ways a Prolog system can be invoked:

```
% program Prolog's arguments

% program Prolog's arguments + emacs' arguments

% program Prolog's arguments +z user's arguments
```

where *program*, generally `prolog`, but can be an executable QOF file (see Section 8.5.4 [ref-sls-sst], page 196) or a stand-alone program (see Section 9.1 [sap-srs], page 337).

Prolog's arguments consists of:

user's arguments
: these arguments can be retrieved in a program by calling `unix(argv(ArgList))`.

system arguments
: The current valid system arguments are:

  '`+`' invoke Emacs; subsequent arguments passed to Emacs;

  '`+f`' fast startup; do not load user's '`prolog.ini`' file;

  '`+l` *file*' load the specified file on startup; *file* may be a Prolog file or a QOF file, and it may be specified either as a string (e.g. '`file`', '`~/prolog/file.pl`') or as a file search path specification (e.g. '`library(file)`', '`home(language(file))`'); note, however, that the latter needs to be quoted to escape the shell interpretation of the parentheses; giving the extension is not necessary; if both source ('`.pl`') and QOF ('`.qof`') files exist, the more recent of the two will be loaded;

  '`+L` *file*' like '`+l`', but search for *file* in the directories given by the shell environment variable `PATH`; and

  '`+p [`*path-name*`]`'
  : prints the Prolog file search path definitions that begin with the string *path-name* (e.g. `library` if '`+p lib`' is specified); *path-name* is optional, and if not given, causes prolog to print all file search path definitions; prolog exits after producing the required output to `stdout`;

  '`+P [`*path-name*`]`'
  : same as '`+p`', but the absolutized versions of the file search path definitions are printed;

  '`+tty`' force the three standard streams associated with a Prolog process to act as tty streams; a tty stream is usually line buffered and handles the prompt automatically;

  '`+z`' all subsequent arguments are user's arguments.

  Only one of '`+`' or '`+z`' is possible on one command line.

emacs' arguments
: arguments to the Emacs interface. This may include file names to edit and may also include GNU Emacs arguments (see Section 4.2 [ema-emi], page 88).

All command line arguments beginning with a '`+`' are reserved for system arguments. If user arguments need to begin with a '`+`', they should be given as '`++`' instead. The '`++`' is converted into a single '`+`' by the argument handling routines, and thus, to the user's code, only the single '`+`' argument is visible. An exception to this is when an argument is given following a '`+z`' option in which case no conversion is done.

Runtime systems do not interpret system arguments; they treat all arguments as user's arguments.

There can be any number of '`+l`' and '`+L`' arguments. In Release 3, invoking a saved-state, an executable QOF-file, as a command causes the corresponding Prolog executable, the one from which the saved-state was created, to be invoked with the arguments '`+L` *saved-state* '.

The user's arguments are accessible in Prolog via `unix(argv(ArgList))`, which returns a list of all the user's arguments. For example, if Prolog is invoked by the command (A), then the Prolog goal (B) returns (C):

```
% prolog ++file1 -file2                                    (A)

| ?- unix(argv(ArgList)).                                  (B)

ArgList = ['+file1','-file2']                              (C)
```

8.3.1.1 The Initialization File

Once invoked, the default `prolog` looks in your home directory for a file named '`prolog.ini`', and if it finds one, loads it. Stand-alone programs also look for and load '`prolog.ini`'. Runtime systems do not.

Typically, '`prolog.ini`' files are used to define file search paths, library directories, and term expansions.

If the '`+f`' option is specified, the initialization file '`prolog.ini`' is not loaded.

8.3.2 Exiting Prolog

To exit from the Prolog system, either type

```
| ?- halt.
```

or your end-of-file character. The end-of-file character is `^d` by default. You will use a different command set to exit Prolog running under Emacs; see Section 4.2.2 [ema-emi-key], page 89. The commands for exiting are as follows:

without Emacs:
: type `^d` (or `^C e`). You may also execute the goal `halt/0`.

with GNU Emacs:
: type `^c ^d` (or ⟨ESC⟩ *X exit-emacs*).

with QUI: Use **Interrupt** button in main window. Select **Exit**.

If you would like to pause Prolog while keeping the job in the background:

without Emacs:
: type `^Z`.

with GNU Emacs:
type *^X^Z*.

with QUI: iconify the main window

8.4 Loading Programs

8.4.1 Overview

There are two ways of loading programs into Prolog — compiling source files and loading pre-compiled QOF files. Earlier releases of Prolog distinguished between compiling and consulting source files. Consulting a file caused the code to be loaded in an interpreted mode so that it could be debugged. Now compiled code is fully debuggable, so there is no longer any need to distinguish between compiling and consulting, and the built-in predicate `consult/1` is now just a synonym for `compile/1`. Interpretation is now only used for the execution of dynamic code.

This section contains references to the use of the module system. These can be ignored if the module system is not being used (see Section 8.13 [ref-mod], page 271 for information on the module system).

8.4.2 The Load Predicates

Loading a program is accomplished by one of these predicates

`load_files(`*File*`)`
compiles source file or loads QOF file, whichever is the more recent. `load_files(`*File*`)` can also be written as `[`*File*`]`.

`compile(`*File*`)`
compiles source file.

`consult(`*File*`)`
Same as `compile`

`ensure_loaded(`*File*`)`
loads more recent of source and QOF files, unless the file has already been loaded and it has not been modified since it was loaded.

`load_files(`*File*`, `*Options*`)`
loads file according to the specified options. All the above predicates can be regarded as special cases of this one.

`reconsult(`*Files*`)`
same as `consult`

The following notes apply to all the Load Predicates:

1. The *File* argument must be one of:
   - an atom that is the name of a file containing Prolog code; a '`.pl`' or a '`.qof`' suffix to a filename may be omitted (see Section 8.6.1.3 [ref-fdi-fsp-fde], page 209;)
   - a list of any atom listed above;
   - the atom `user`

     Please note: If the filename is not a valid atom, it must be enclosed in single quotes. For example,

     ```
     load_files(expert)
     load_files('Expert')
     compile('/usr/joe/expert')
     ensure_loaded('expert.pl')
     ```
2. These predicates resolve relative file names in the same way as `absolute_file_name/2`. For information on file names refer to Section 8.6 [ref-fdi], page 205.
3. The above predicates raise an exception if any of the files named in *File* does not exist, unless the `fileerrors` flag is set to `off` using `nofileerrors/0`.

   Errors detected during compilation, such as an attempt to redefine a built-in predicate, also cause exceptions to be raised. However, these exceptions are caught by the compiler, and an appropriate error message is printed.
4. There are a number of *style warnings* that may appear when a file is compiled. These are designed to aid in catching simple errors in your programs, but some or all of them can be turned off if desired using `no_style_check/1`. The possible style warnings are:
   a. A named variable occurs only once in a clause. Variables beginning with a '`_`' are considered not named.
   b. All the clauses for a predicate are not adjacent to one another in the file.
   c. A predicate is being redefined in a file different from the one in which it was previously defined.
5. By default, all clauses for a predicate are required to come from just one file. A predicate must be declared `multifile` if its clauses are to be spread across several different files. See the reference page for `multifile/1`.
6. If a file being loaded is not a module-file, all the predicates defined in the file are loaded into the source module. The form `load_files(Module:File)` can be used to load the file into the specified module. See Section 8.13.3 [ref-mod-def], page 272, for information about module-files. If a file being loaded *is* a module-file, it is first loaded in the normal way, then the source module imports all the public predicates of the module-file except for `use_module` and `load_file` if you specify an import list.
7. If there are any directives in the file being loaded, that is, any terms with principal functor `:-/1` or `?-/1`, then these are executed as they are encountered. A common type of directive to have in a file is one that loads another file, such as

   ```
   :- [otherfile].
   ```

   In this case, if otherfile is a relative filename it is resolved with respect to the directory containing the file that is being loaded, not the current working directory of the Prolog system.

Any legal Prolog goal may be included as a directive. Note, however, that if the file is compiled by `qpc`, the goal will be executed by `qpc`, not when the '`.qof`' file is loaded or when application begins execution. The `initialization/1` declaration provides this functionality. There is no difference between a '`:-/1`' and a '`?-/1`' goal in a file being compiled.

8. If *File* is the atom `user`, or *File* is a list, and during loading of the list `user` is encountered, procedures are to be typed directly into Prolog from the terminal. A special prompt, '`| `', is displayed at the beginning of every new clause entered from the terminal. Continuation lines of clauses typed at the terminal are preceded by a prompt of five spaces. When all clauses have been typed in, the last should be followed by an end-of-file character.
9. Terms that are notational variants of Prolog terms, notably grammar terms, are expanded into Prolog code during compilation. By defining the hook predicate `term_expansion/2` (in module user), you can specify any desired transformation to be done as clauses are loaded.
10. Any predicates that need to be called during the compilation of a file, including `term_expansion/2` and all the predicates it calls, must be treated specially if you wish to be able to compile that file with `qpc`. See Section 9.1.6.1 [sap-srs-eci-crt], page 349 for information on this.
11. The current load context (module, file, stream, directory) can be queried using `prolog_load_context/2`.

8.4.3 Redefining Procedures during Program Execution

You can redefine procedures during the execution of the program, which can be very useful while debugging. The normal way to do this is to use the '`break`' option of the debugger to enter a break state (see `break/0`, Section 8.11.1 [ref-iex-int], page 250), and then load an altered version of some procedures. If you do this, it is advisable, after redefining the procedures and exiting from the break state, to wind the computation back to the first call to any of the procedures you are changing: you can do this by using the '`retry`' option with an argument that is the invocation number of that call. If you do not wind the computation back like this, then:

- if you are in the middle of executing a procedure that you redefine, you will find that the old definition of the procedure continues to be used until it exits or fails;
- if you should fail back into a procedure you have just redefined, then alternative clauses in the old definition will still be used.

8.4.4 Predicate List

Detailed information is found in the reference pages for the following:

- `compile/1`
- `consult/1`

- `ensure_loaded/1`
- `load_files/[1,2]`
- `multifile/1`
- `no_style_check/1`
- `style_check/1`
- `prolog_load_context/2`
- `term_expansion/2`
- `use_module/[1,2,3]`

8.5 Saving and Loading the Prolog Database

8.5.1 Overview of QOF Files

Quintus Prolog release 3 provides a much more powerful alternative to the traditional save/restore facilities of Prolog. It is now possible to save, and subsequently load, individual predicates, or sets of predicates, or entire modules of predicates, or indeed the complete Prolog database. Such sets of predicates are saved into Quintus' standard *Quintus Object Format* ("QOF" files). This significantly extends the utility of QOF files, which were previously only used to produce runtime systems and stand-alone programs. *QOF files* are now a fully general way of storing arbitrary Prolog facts and rules in a form that can be quickly and easily used. QOF files contain a machine independent representation of both compiled and dynamic Prolog predicates. This means they are completely portable between different platforms running Quintus Prolog.

QOF files can be generated by:

- saving a selected set of predicates from a running Prolog application (see Section 8.5 [ref-sls], page 192);
- using the stand-alone `qpc` Prolog compiler to compile individual Prolog source files (see Section 9.1.2 [sap-srs-qpc], page 341).

QOF files can be used by:

- loading them into a running Prolog application (see Section 8.5 [ref-sls], page 192);
- linking them together, using the stand-alone `qld` linker, into an object file for linking into an executable with static Prolog code made shareable (see Section 9.1.3 [sap-srs-qld], page 343). This executable can either:
  - be a full Quintus Prolog Development System allowing continued development;
  - or can be a runtime system for deployment of the application.

QOF saving and loading is available in both Quintus Prolog development systems and runtime systems built for distribution. In a development system, code can be incrementally

compiled using the built-in compiler and then saved into a QOF file. In a runtime system, which does not include the built-in compiler, dynamic code can be asserted and then saved. Runtime systems can load QOF files containing previously compiled code.

This combination of the capabilities of the Quintus Prolog Runtime Generator, with the saving and loading facilities of Quintus Prolog release 3, and the cross-platform portability of QOF files, provides tremendous flexibility that can be used for many purposes. For example:

- precompiling Prolog libraries for fast loading;
- packaging Prolog code for distribution;
- generating precompiled databases of application data;
- selectively loading particular application databases (and rule bases);
- saving Prolog data across application runs;
- building and saving new application databases from within applications;
- linking selected application databases into the application executable for ease of distribution, or to get code sharing and better memory and paging performance.

The facilities for saving and loading QOF files are more than just a convenience when developing programs; they are also a powerful tool that can be used as part of the application itself.

8.5.2 Compatibility with save/restore in previous releases

Unfortunately, it has not been possible to retain the semantics of `save/[1,2]` available in previous releases of Quintus Prolog. This is regrettable because it means that programs that incorporate code for building saved-states will need to be changed. This section explains why it was necessary to remove these predicates. Note, however, that `save_program/1` *is* available and has the same semantics as previous releases (except for foreign code), although it is based on a new implementation using QOF files. A new predicate `save_program/2`, described in Section 8.5.4 [ref-sls-sst], page 196, has been provided, which supports the most common usage of `save/[1,2]`, which was to specify an initial goal for a saved-state to call when run.

The difference between `save_program/1` and `save/[1,2]` in previous releases of Quintus Prolog was that `save_program/1` saved only the Prolog database, whereas `save/[1,2]` saved both the Prolog database and the Prolog execution stacks. It has not been possible to retain the saving of the Prolog execution stacks in a way consistent with the release 3 support of embeddability and the general portability of QOF files. This is why `save/[1,2]` have been removed. The reasoning goes as follows:

1. QOF files are a completely portable machine-independent representation of Prolog data.
2. It is difficult, if not impossible, to make the Prolog execution state portable in the same way as facts and rules in QOF files (see further points).

3. QOF files can also be combined and loaded in flexible ways, and it is unclear what this would mean for execution states.
4. The QOF file saved-states do *not* save any C (or other foreign language) state. This is a change from the previous Quintus Prolog saved-states, and is further discussed below.
5. In the general case, Prolog execution can now be arbitrarily interleaved with C (or other) function calls since Prolog and C are completely intercallable and can call each other recursively.
6. Since the C state is not saved, it is not possible to meaningfully save the Prolog execution state in the general case where it depends on interleaved C execution state.
7. In addition, Prolog code embedded in a C (or other) application is highly likely to be manipulating C data, such as pointers and other process-specific information. This data would be meaningless if restored into another process, and indeed would be likely to cause faults.

The model that an arbitrary Prolog execution state can be saved thus only works well within a Prolog-only situation. In the complex embedded environments supported by Quintus Prolog release 3 this model cannot work properly. Hence the removal of the facility.

As mentioned in points 4-7 above, an additional important aspect here is that Prolog no longer makes any attempt to save the state of C (or other foreign language) code. This was a feature of saved-states in previous releases where both the C code and its data structures were saved (as a memory image) into saved-states. This was a feature that caused many problems. A primary problem was that the saved C state was initialized (variables retained their values when restored) and yet the initialized C state could contain many items that were no longer valid in the new process, such as addresses and file descriptors. Such code would often fail when restored. In addition, Prolog was unable to guarantee that it had saved all the necessary foreign code state. With the advent of shared libraries and other complex memory management facilities in the operating system, it became impossible for Prolog to control and manage the states of other tools in the address space.

When one takes a step back and looks at Prolog in the light of the goals of release 3 (see Section 1.2 [int-hig], page 4) — where Prolog code is a component that can be embedded in complex applications written in many languages — it is clearly unreasonable for Prolog to try and control, let alone save, arbitrary non-Prolog state. The Prolog operations for saving and loading QOF files now operate solely on the Prolog database and these operations do not involve making any assumptions about non-Prolog state. This is a much cleaner and more robust approach, and is the most appropriate when Prolog applications become embedded software components.

8.5.3 Foreign Code

Prolog QOF files do *not* contain any foreign code or foreign data structures. As discussed in the previous section, this is different from saved-states in previous releases of Quintus Prolog.

However, QOF files can have dependencies on object files that will be automatically loaded when the QOF file is loaded. These dependencies can arise because:

1. The QOF file was compiled with the stand-alone compiler `qpc` (see Section 9.1.2 [sap-srs-qpc], page 341) and its source file contained calls to `load_foreign_files/2`, which will have been turned into object file dependencies in the QOF file.
2. The QOF file was saved from a Prolog session, using `save_program/[1,2]` or `save_modules/2`, in which foreign code had been previously loaded. The object files loaded will be recorded as object file dependencies in the QOF file.

The `qnm` utility can be used (from the command prompt) to see the list of dependencies in a QOF file:

```
% qnm -D file.qof
```

It is slower to start up a QOF file with object file dependencies because the object files must be re-linked and re-loaded. This re-linking and re-loading will occur every time the QOF file is loaded and the necessary object files are not yet loaded into the system. If this start-up time becomes a problem then this can be tackled by statically linking your foreign code into the Prolog system. This is described in Section 9.1 [sap-srs], page 337. This will make the foreign code become part of the Prolog executable so there is no run-time linking required at all. In addition, on most systems, this Prolog code will now be demand-paged from the executable, which will improve start-up time and reduce paging (as compared with dynamic linking). We recommend switching over to using static linking for programs with a lot of foreign code. The dynamic foreign language interface can be used for loading code while testing, but once your foreign code is stable it is better to have it statically linked. Furthermore, dynamically loaded foreign code cannot be debugged with a debugger such as `gdb(1)`, so you will also need to statically link the foreign code so that the debugger can be used on the resulting executable. This contradicts the fact that foreign code should be dynamically loaded while testing.

All foreign code is either linked into the Prolog executable, or is re-loaded when a QOF file is loaded. This means that when a program is started the foreign code will always be in an uninitialized state. This is exactly the same as any other program. However, this is a change from saved-states in previous releases of Quintus Prolog that saved the initialized foreign state. The new semantics is much cleaner, is consistent with standard practice, and avoids previous problems with invalid initializations that were not valid in the new process. This latter problem was particularly problematic for libraries (such as Curses, X Windows, and database interfaces) since the user did not usually have source code for the libraries and how they initialized and what they depended upon could not be easily understood. In Quintus Prolog Release 3, foreign code linked with Prolog, or loaded into Prolog, will work just the same as if it were a separate program independent of Prolog.

It is possible that some of your previous programs relied on the saving of foreign state into saved-states. If you have such programs then they will need to be changed. Usually the change will involve making sure that the foreign code is explicitly initialized each time the application is run. The initialization facilities described below, see Section 8.5.6 [ref-sls-igs], page 199, may be useful for this.

8.5.4 Saved-States

Saved-states are just a special case of QOF files. The `save_program/[1,2]` predicate will save the entire Prolog database into a QOF file, and then in addition will make that QOF file executable. Under UNIX, the QOF file is made executable by making the first line of the file be a `sh(1)` script. This script runs the executable that the QOF file was saved from, telling it to load the QOF file.

So, if a saved-state is created as follows:

```
| ?- save_program(savedstate).
```

then if we look at the first line of the file we will see something like the following. Note that '`+L`' is a Prolog command line option to load a QOF file ('`$0`' will be the name of the QOF file and '`$*`' will be any other arguments given when it is run).

```
% head -1 savedstate
exec /usr/local/quintus/bin3.5/sun4-5/prolog +L $0 $*
```

This QOF file can then be run from the command line as follows:

```
% savedstate
```

In addition to the user's code from the Prolog database, a saved-state saved by `save_program/[1,2]` also contains Prolog system information such as flag settings, debugging information and so forth. When the saved-state is loaded this system state is also restored, which is convenient for continued development.

Apart from being made executable, and containing additional Prolog system information, a saved-state saved through `save_program/[1,2]` is just a standard QOF file. This means that it can be used anywhere you would otherwise use a QOF file, for such things as loading into Prolog, linking together with other QOF files, and linking into executables (see Section 9.1 [sap-srs], page 337 for information on these linking capabilities).

A saved-state, or any QOF file, can be restored using the `restore/1` predicate from within Prolog:

```
| ?- restore(savedstate).
```

The `restore/1` predicate will re-execute the running executable (using the `execv(3)` system call) in order to obtain a completely new environment, and will then load the QOF file. If the QOF file was saved with `save_program/[1,2]` then this will restore exactly the same Prolog database state as when the saved-state was saved. In runtime systems, however, it is the application program's responsibility to load the file into the restarted executable, see Section 18.3.153 [mpg-ref-restore], page 1266 and Section 9.1.8 [sap-srs-sqf], page 354.

Note that the executable that will be re-executed by `restore/1` is the one currently running. This may be different from the one named in the first line in the QOF file, if that QOF file was saved from some different executable. To use the executable that originally saved the

QOF file you should return to the command interpreter and run the QOF file directly. To use the executable you are currently running, you should use `restore/1`.

If the loaded QOF file has object file dependencies then those object files will be re-linked and re-loaded as part of loading the QOF file. If the object file cannot be found or linked, then an exception will be raised. Similarly, QOF dependencies are also reloaded at this point.

> **Windows caveat:**
>
> Under Windows, it is not possible to replace a running executable with another. Under Windows, `restore/1` will instead start a new sub-process and then terminate the running process. For more details see the Microsoft documentation for `execv()`.
>
> In a Windows command prompt window, the command interpreter does not wait when a process executes an `execv()` library call. Thus after `restore/1`, the program gives the appearance of running in the background.
>
> **Please note:** The QOF file saved by `save_program/2` does *not* contain any of the Prolog code that is statically linked into the executable. Only the Prolog database (both compiled and dynamic) that has been built since the executable started running is saved. This is done to avoid code duplication in the saved-state. However, this does mean that if the QOF-file is loaded into a different executable, then the program may be missing some code that it assumes should be there, because it was present in the original executable. An example would be a saved-state that was saved from an executable containing Quintus' ProWINDOWS add-on product. If that saved-state is loaded into a normal Prolog executable without ProWINDOWS then any calls to ProWINDOWS will not work (they will generate undefined predicate exceptions). The correct thing to do is clearly to make sure that you use either the original executable, or an executable that contains the necessary programs, or you load the necessary programs in addition to loading the saved-state QOF file.

The `save_program/2` predicate can be used to specify an initial goal that will be run when the saved-state is re-loaded. This usage of `save_program/2` replaces the most common uses of the old `save/[1,2]` predicates that are no longer available. For example:

```
| ?- save_program(saved_state,initial_goal([a,b,c])).
```

When '`saved_state`' is loaded `initial_goal/1` will be called. This allows saved-states to be generated that will immediately start running the user's program when they are executed. In addition to this `save_program/2` facility there is also a comprehensive facility for programs to set up initializations to be run when they are loaded or otherwise started. This is described below in Section 8.5.6 [ref-sls-igs], page 199.

8.5.5 Selective saving and loading of QOF files

The `save_program/[1,2]` and `restore/1` predicates discussed in the previous section are used for saving and restoring the entire Prolog database. To save selected parts of a Prolog database, the predicates `save_modules/2` and `save_predicates/2` are used.

For example, to save the modules `user` and `special` you would use:

```
| ?- save_modules([user,special],'file1.qof').
```

All predicates in those modules will be saved, and in addition any foreign code files previously loaded into these modules will generate an object file dependency in the QOF file. All information in these modules about predicates attached to foreign functions, and also predicates that have been made externally callable from foreign code, is saved as a normal part of the module.

For each module imported by one of the specified modules, a QOF file dependency is included in the QOF file. This means that when you load '`file1.qof`' into another Prolog session, it will automatically load any additional QOF files that it needs.

To just save certain predicates you would use:

```
| ?- save_predicates([person/2,dept/4],'file2.qof').
```

This will only save the predicates specified. In this case no additional dependency information is saved into the QOF file. Note that the module information for these predicates is included. When the QOF file is loaded the predicates will be loaded into the same module they were in originally.

Any QOF file, however generated, can be loaded into Prolog with `load_files/[1,2]`:

```
| ?- load_files('file1.qof')
```

or, equivalently:

```
| ?- ['file1.qof'].
```

The information from each QOF file loaded is incrementally added to the database. This means that definitions from later loads may replace definitions from previous loads. A saved-state QOF file saved with `save_program/[1,2]` can also be loaded with `load_files/[1,2]` in which case the contents of the saved-state are just incrementally added to the database as for any other QOF file. The use of `load_files/[1,2]` for this is different from the use of `restore/1` in that `restore/1` will re-execute the executable thus reinitializing the database. Using `load_files/[1,2]` allows the database to be incrementally changed within the same process.

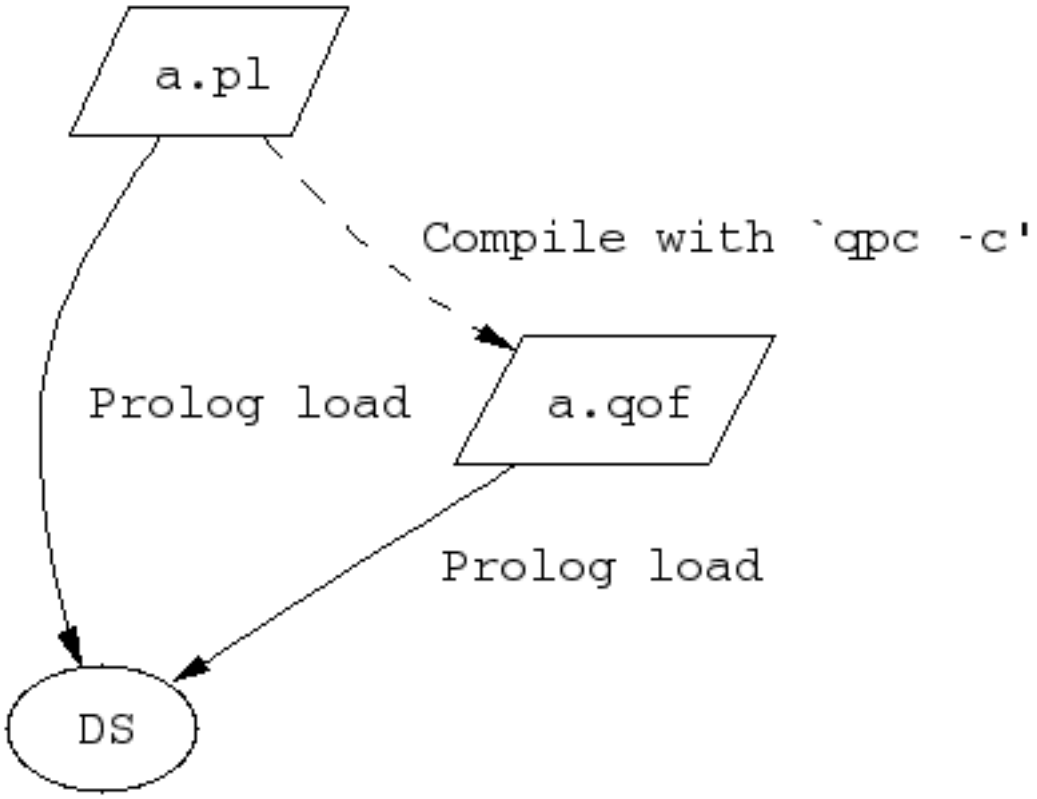
If the loaded QOF file has object file dependencies then those object files will be linked and loaded as part of loading the QOF file unless they have already been loaded. If the object file cannot be found or linked, then an exception will be raised.

The predicates `load_files/[1,2]` are used for compiling and loading source files as well as QOF files. If '`file1.qof`' and '`file1.pl`' both exist (and '`file1`' does not), then `load_files` (file 1) will load the source ('`.pl`') or the QOF, whichever is the most recent. Refer to Section 8.4 [ref-lod], page 189 for more information on loading programs, and also to the reference page for `load_files/[1,2]`.

> **Advanced note:** Both `save_modules/2` and `save_predicates/2` will save Prolog code that is statically linked if such modules or predicates are specified. This is different from `save_program/[1,2]`, which will not save statically linked Prolog code. Note that if such a QOF file is loaded back into the same executable that saved it, then the new definitions from the QOF file will replace the statically linked code. There is no problem with this, except that some space will be wasted. The original statically linked code will not be used, but since it is linked into the executable its space cannot be reclaimed. Since static linking is normally used to optimize start-up time and the space usage for code, it is somewhat of a waste to circumvent this by saving and loading a lot of Prolog code that is already in the executable. If the QOF file is to be used for other purposes, such as re-linking the executable, or as a part to be loaded into another program, then, of course, the saving of statically linked code is probably exactly what is required.

8.5.6 Initializing Goals in Saved States

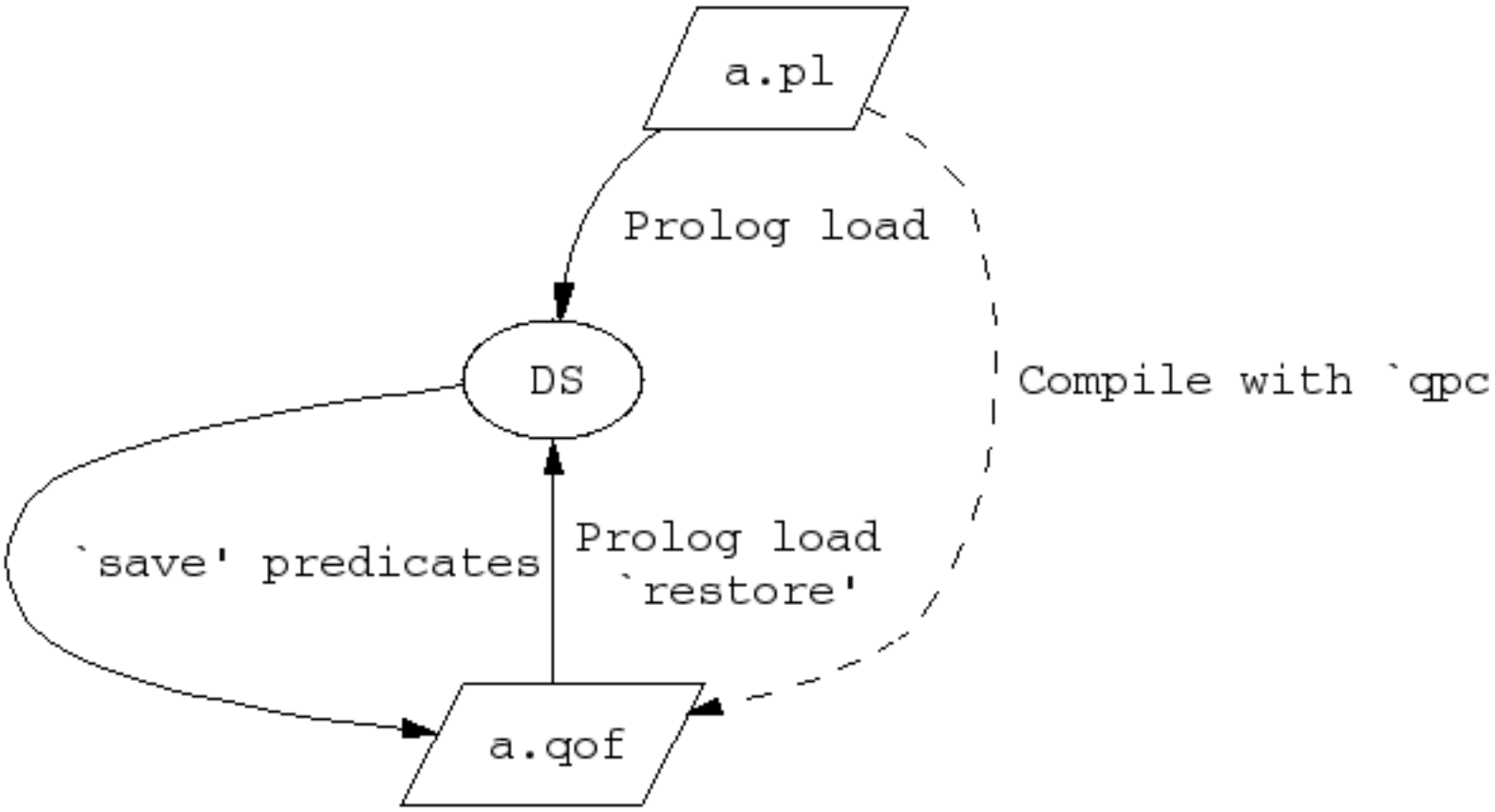
Under the earlier model, a Prolog file could either be compiled into the development system, or compiled to Quintus Object Format by `qpc`, as shown in the following figure.



Compilation options: Quintus Prolog 2.5

The ability to save and load QOF files in a development system makes the picture more complicated. The following figure shows the ways a Prolog source file can be compiled or saved.

It would be natural to expect '`a.qof`' to be the same, however generated. But both the '`save`' predicates and `qpc` offer a rich variety of options, and the reality is less simple (see the following figure).



Saving and loading options: Quintus Prolog 3.5

8.5.6.1 The Initialization Declaration

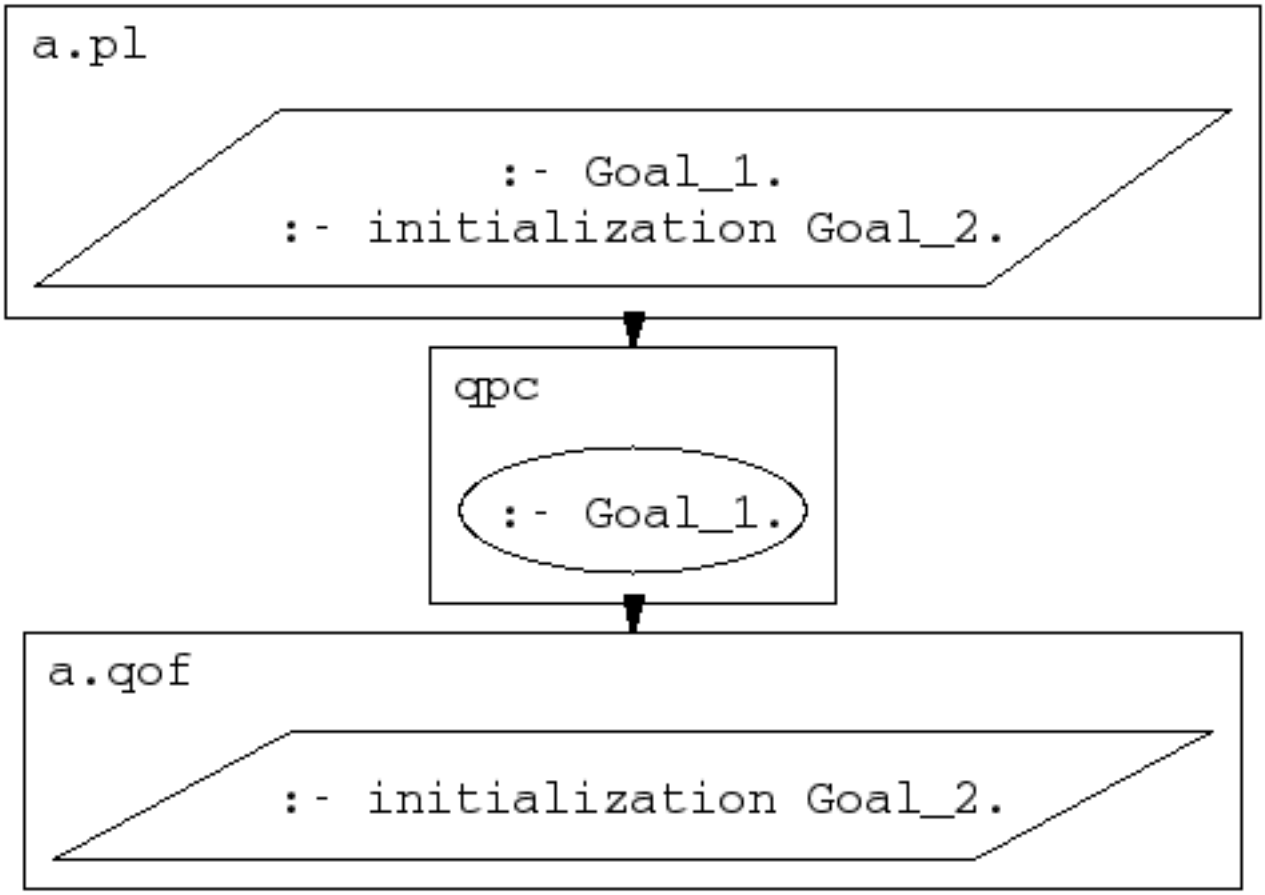
The `initialization/1` predicate is an important complement to the embedded directive construct '`:- Goal`' appearing in a file being consulted or compiled, and can in many cases not only replace the directive, but also make the code work better when used in stand-alone programs and runtime systems.

The main reason for this is that '`:- Goal`' directive is executed at compile-time, not when the file in which the construct occurs is actually loaded into a running system. This causes no problems within development systems, but if we want to save states and compile programs into qof-files, link them together, and later start them up again, problems arise because:

- The '`:- Goal`' construct calls *Goal* only once, when the file is compiled, not when a saved state containing the file is restored.
- *Goal* is called at compile-time, which means that if you use `qpc` to compile source code into a qof-file, your directives will be run during this compilation, not when you load the qof-file or start up a stand-alone system to which the qof-file has been linked.

The `initialization/1` predicate, on the other hand, provides a way of letting initialization routines be called when a file is actually loaded or a system containing the file is started up. This allows for correct initialization in stand-alone programs and runtime systems; therefore a recommended programming style is to use `initialization/1` instead of a bare '`:- Goal`' construct whenever appropriate.

In the following figure, *Goal_1* might typically be an operator declaration and *Goal_2*, an initialization predicate that modifies the database.



Embedded directives (*goal_1*) vs. initialized goals (*goal_2*)

The initialization goal, *Goal_2*, is defined to be run when:

- a source file with a `:- initialization` *Goal* directive is loaded into a running system (using compile, consult etc.)
- a stand-alone program or runtime system is started up, and some file linked to the system had a `:- initialization` *Goal* directive. If several files had such directives, the order in which the goals are run is not defined.
- a saved state is restored, and some file loaded to the saved system had a `:-`

`initialization` *`Goal`* directive, or `initialization(`*`Goal`*`)` was called before the state was saved. If several initialization goals were defined in the system, the order in which they are run upon a restore is not defined.

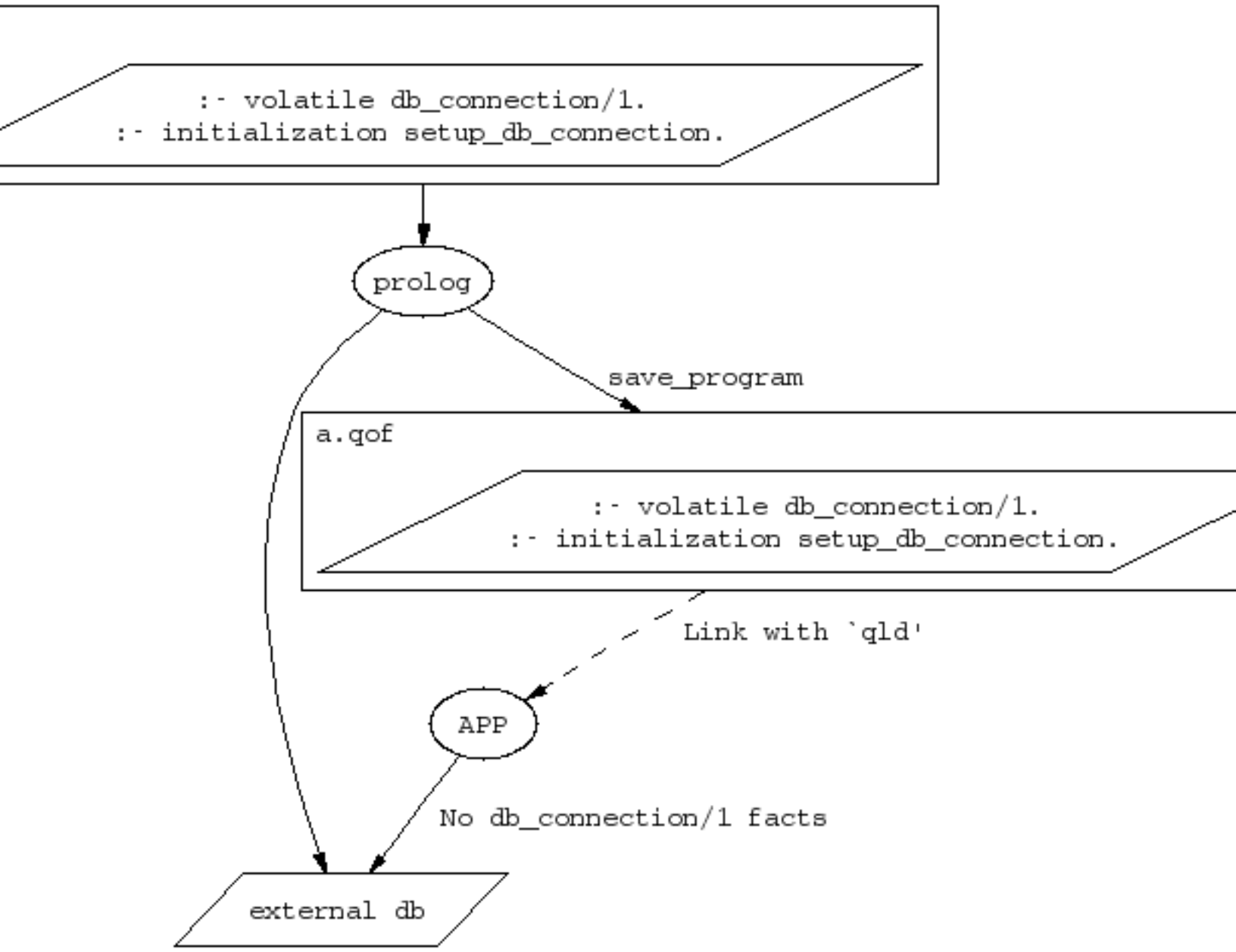
- a QOF file is loaded into a running system, and the source file that was compiled into the qof file had a '`(:- initialization` *`Goal`*`)`' directive.

8.5.6.2 Volatile Predicates

A predicate should be declared as volatile if it refers to data that cannot or should not be saved in a QOF file. In most cases a volatile predicate will be dynamic, and it will be used to keep facts about streams or references to C-structures. When a state or a module is saved at run-time, the clauses of all volatile predicates defined in the context will be left unsaved. The predicate definitions will be saved though, which means that the predicates will keep all properties, that is volatile and maybe dynamic or multifile, when the saved state is restored.

For example, if a Prolog application connects to an external database at start up, establishing a connection by an assertion like (A), a volatile declaration would prevent each particular connection from getting saved in the QOF file, as illustrated in the following figure. A code example is found in the reference page for `initialization/1`.

```
assert(db_connection(Connection))                                    (A)
```



Using the Volatile Property

When used as a compile-time directive, the volatile declaration of a predicate must appear before all clauses of that predicate. The predicate is reinitialized. When used as a callable goal, the only effect on the predicate is that it is set to be volatile.

8.5.6.3 Fine Tuning

To tune the initialization of a file or system to be run only when it should be run, `volatile/1`, in combination with other declarations, give `initialization/1` the infor-

mation necessary to distinguish different loading situations. In the reference pages, we show how some common situations can be programmed using these predicates.

If a source file contains data that is supposed to be transformed according to some complicated rules (which cannot be done with `term_expansion/2`), and the data after the transformation can be saved into a saved state, we might want the transformation to be done when the file is loaded, but not when a saved state is restored. The following program defines the initialization to be run only when the file is loaded:

```
:- dynamic do_not_transform/0. % reset fact
:- initialization my_init.

my_init :-
  ( do_not_transform ->
    true
  ; undo_transform, % remove old data
    do_transform,
    assert(do_not_transform)
  ).
```

In the above example, `do_transform/0` and `undo_transform/0` are user defined.

8.5.7 Predicate List

Detailed information is found in the reference pages for the following:

- `initialization/1`
- `load_files/[1,2]`
- `prolog_load_context/2`
- `restore/1`
- `save_modules/2`
- `save_predicates/2`
- `save_program/[1,2]`
- `volatile/1`

8.6 Files and Directories

8.6.1 The File Search Path Mechanism

As a convenience for the developer and as a means for extended portability of the final application, Quintus Prolog provides a flexible mechanism to localize the definitions of the system dependent parts of the file and directory structure a program relies on, in such a

way that the application can be moved to a different directory hierarchy or to a completely new file system, with a minimum of effort.

This mechanism, which can be seen as a generalization of the `library_directory/1` scheme available in previous releases, presents two main features:

1. An easy way to create aliases for frequently used directories, thus localizing the external, file system and directory structure dependent directory name, to one single place in the program.
2. A possibility to associate more than one directory specification with each alias, thus giving the developer full freedom in sub-dividing libraries, and other collections of programs, as it best suits the structure of the external file system, without making the process of accessing files in the libraries any more complicated. In this case, the alias can be said to represent a file search path, not only a single directory.

The directory aliasing mechanism, together with the additional file search capabilities of `absolute_file_name/3`, can effectively serve as an intermediate layer between the external world and a portable program. For instance, the developer can hide the directory representation by defining directory aliases, and he can automatically get a proper file extension added, dependent on the type of file he wants to access, by using the appropriate options to `absolute_file_name/3`.

A number of directory aliases and file search paths, are predefined in the Quintus Prolog system (though they can be redefined by the user). The most important of those is the `library` file search path, giving the user instant access to the Quintus library, consisting of several sub-directories and extensive supported programs and tools.

Specifying a library file, using the alias, is possible simply by replacing the explicit file (and directory) specification with the following term:

`library(`*file*`)`

The name of the file search path, in this case `library`, is the main functor of the term, and indicates that *file* is to be found in one of the library directories.

The association between the alias `library` (the name of the search path) and the library directories (the definitions of the search path), is defined by Prolog facts, `library_directory/1`, which are searched in sequence to locate the file. Each of these facts specifies a directory where to search for *file*, whenever a file specification of the form `library(`*file*`)` is encountered.

The library mechanism discussed above, which can be extended with new directories associated with the alias `library`, has become subsumed by the more general aliasing mechanism, in which arbitrary names can be used as aliases for directories. The general mechanism also gives the possibility of defining path aliases in terms of already defined aliases.

In addition to `library`, the following aliases are predefined in Quintus Prolog: `quintus`, `runtime`, `system`, `helpsys language` and `tutorial`. The interpretation of the predefined aliases are explained below.

8.6.1.1 Defining File Search Paths

The information about which directories to search when an alias is encountered, is defined by the dynamic, multifile predicate `file_search_path/2`. The clauses for this predicate are located in module *user*, and have the following form:

```
file_search_path(PathAlias, DirectorySpec).
```

PathAlias must be an atom. It can be used as an alias for *DirectorySpec*

DirectorySpec
: Can either be an atom, spelling out the name of a directory, or a compound term using other path aliases to define the location of the directory.

The directory path may be absolute, as in (A) or relative as in (B), which defines a path relative to the current working directory.

Then, files may be referred to by using file specifications of the form similar to `library(file)`. For example, (C), names the file '`/usr/jackson/.login`', while (D) specifies the path '`etc/demo/my_demo`' relative to the current working directory.

```
file_search_path(home, '/usr/jackson').                    (A)

file_search_path(demo, 'etc/demo').                        (B)

home('.login')                                             (C)

demo(my_demo)                                              (D)
```

As mentioned above, it is also possible to have multiple definitions for the same alias. If clauses (E) and (F) define the `home` alias, then to locate the file specified by (G) each `home` directory is searched in sequence for the file '`.login`'. If '`/usr/jackson/.login`' exists, it is used. Otherwise, '`/u/jackson/.login`' is used if it exists.

```
file_search_path(home, '/usr/jackson').                    (E)
file_search_path(home, '/u/jackson').                      (F)

home('.login')                                             (G)
```

The directory specification may also be a term of arity 1, in which case it specifies that the argument of the term is relative to the `file_search_path/2` defined by its functor. For example, (H) defines a directory relative to the directory given by the `home` alias. Therefore, the alias `qp_directory` represents the search path '`/usr/jackson/prolog/qp`' followed by '`/u/jackson/prolog/qp`'. Then, the file specification (I) refers to the file (J), if it exists. Otherwise, it refers to the file (K), if it exists.

```
file_search_path(qp_directory, home('prolog/qp')).          (H)

qp_directory(test)                                          (I)

/usr/jackson/prolog/qp/test                                 (J)

/u/jackson/prolog/qp/test                                   (K)
```

Aliases such as `home` or `qp_directory` are useful because even if the `home` directory changes, or the `qp_directory` is moved to a different location, only the appropriate `file_search_path/2` facts need to be changed. Programs relying on these paths are not affected by the change of directories because they make use of file specifications of the form `home(`*`file`*`)` and `qp_directory(`*`file`*`)`.

All built-in predicates that take file specification arguments allow these specifications to include path aliases defined by `file_search_path/2` facts. These predicates are:

- `absolute_file_name/[2,3]`
- `compile/1`
- `consult/1`
- `ensure_loaded/1`
- `load_files/[1,2]`
- `open/[2,3]`
- `restore/1`
- `save_module/2`
- `save_predicates/2`
- `save_program/[1,2]`
- `see/1`
- `tell/1`
- `use_module/[1,2,3]`

Notes:

1. The `file_search_path/2` database may contain directories that do not exist or are syntactically invalid (as far as the operating system is concerned). If an invalid directory is part of the database, the system will fail to find any files in it, and the directory will effectively be ignored.
2. This facility is provided so that one can load library or other files without knowing their absolute file names, but this does not restrict the way a file can be accessed. It is *strongly* suggested that writing to a file not be done using the *PathAlias*(*FileSpec*) facility. (One could write to *PathAlias*(*FileSpec*) but this may not have the desired effect, since the system will write to one of possibly many files depending upon the current order of the clauses in the `file_search_path/2` predicate.) The absolute name of the file to which one is writing should be known. To find the absolute name of a library file, for example, one can type

```
| ?- absolute_file_name(library(FileSpec), AbsFileName).
```

3. `file_search_path/2` must be defined in the default module `user` — definitions in any other module will not be found.

8.6.1.2 Frequently Used File Specifications

Frequently used `file_search_path/2` facts are best defined using the initialization file '`prolog.ini`' , which is consulted at startup time by the Development System. Therefore, with reference to the examples from Section 8.6.1.1 [ref-fdi-fsp-def], page 207, clauses like one following should be placed in the '`prolog.ini`' file so that they are automatically available to user programs after startup:

```
:- multifile file_search_path/2.
:- dynamic file_search_path/2.
file_search_path(home, '/usr/jackson').
file_search_path(qp_directory, home('prolog/qp')).
file_search_path(demo, 'etc/demo').
```

If it is necessary to avoid multiple definitions of the same fact, this would be useful, for example, when restoring a saved state saved by `save_program/1` at which time the '`prolog.ini`' file is consulted again, a predicate such as `add_my_search_path/2` can be defined in the '`prolog.ini`' file.

```
add_my_search_path(Name, FileSpec) :-
    file_search_path(Name, FileSpec),
    !.
add_my_search_path(Name, FileSpec) :-
    assert(file_search_path(Name, FileSpec)).
```

This predicate only asserts a clause into the database if it is not already defined. Then, using goals of the following form avoids multiple definitions:

```
:- add_my_search_path(home, '/usr/jackson').
:- add_my_search_path(demo, 'etc/demo').
:- add_my_search_path(qp_directory, home('prolog/qp')).
```

8.6.1.3 Filename Defaults

Some of the predicates that take file specification arguments not only can search for a file among the directories defined by `file_search_path/2` facts (if a path alias is used), but also can help the user in finding the correct file by adding appropriate extensions and/or looking for the most recent file by comparing modification times.

`load_files/[1,2]` (and the predicates defined in terms of `load_files/2`), uses the following algorithm to find the most appropriate file to load:

1. if the file specification is of the form *PathAlias*(*FileName*), retrieve the first directory in the search path associated with *PathAlias* and apply the algorithm below in that directory (for instance, if `library(strings)` are given, look in the first library directory, with *FileName* set to `strings`):
2. if '*FileName*' exists, load it.
3. if '*FileName*`.pl`' exists, but not '*FileName*`.qof`', load '*FileName*`.pl`'
4. if '*FileName*`.qof`' exists, but not '*FileName*`.pl`', load '*FileName*`.qof`'
5. if both '*FileName*`.pl`' and '*FileName*`.qof`' exist, load the one that was most recently modified.
6. if the file specification contained a path alias, retrieve the next directory in the path and retry from (2).

For example,

```
| ?- [user].
| :- multifile file_search_path/2.
| :- dynamic file_search_path/2.
| file_search_path(home, '/usr').
| file_search_path(home, '/usr/prolog').
| end_of_file.        % (or <^D>)
% user compiled in module user, 0.034 sec 284 bytes

yes
```

In this case the directory '`/usr`' is searched first and '`/usr/prolog`' second. Therefore, if the file '`foo.pl`' exists in both of these directories, the following query will compile '`foo.pl`' in the directory '`/usr`' (on the condition that '`foo.qof`' does not exist).

```
| ?- compile(library(foo)).
```

8.6.1.4 Predefined file_search_path Facts

An example of a directory hierarchy that has a constant structure, but that may be installed at different parts of the file system, is the Quintus installation hierarchy. Several `file_search_path/2` facts are defined in the system to support the flexibility of this installation.

The predefined `file_search_path/2` facts are dynamic and multifile, so they can be redefined or expanded by users. In the Quintus Prolog Development System installed for a Sparc running Solaris, the following predefined `file_search_path/2` facts exist to specify the location of certain Development System related directories:

```
file_search_path(quintus,quintus-directory).
file_search_path(runtime,runtime-directory).
file_search_path(runtime,'').
file_search_path(system,'sun4-5').
file_search_path(system,sun4).
file_search_path(system,'').
file_search_path(helpsys,quintus('generic/q3.5/helpsys')).
file_search_path(helpsys,package(helpsys)).
file_search_path(qplib,quintus('generic/qplib3.5')).
file_search_path(library,A) :-
        library_directory(A).
file_search_path(messages,qplib(embed)).
file_search_path(language,english).
file_search_path(demo,quintus('generic/q3.5/demo/bench')).
file_search_path(demo,quintus('generic/q3.5/demo/chat')).
file_search_path(demo,quintus('generic/q3.5/demo/curses')).
file_search_path(demo,quintus('generic/q3.5/demo/math')).
file_search_path(demo,quintus('generic/q3.5/demo/menu')).
file_search_path(demo,quintus('generic/q3.5/demo/search')).
file_search_path(demo,quintus('generic/q3.5/demo/wafer')).
file_search_path(demo,qplib('IPC/TCP/demo')).
file_search_path(demo,qplib('IPC/RPC/demo')).
file_search_path(demo,package(demo)).
file_search_path(tutorial,quintus('generic/q3.5/tutorial')).
file_search_path(tutorial,package(tutorial)).
file_search_path(package,qplib(structs)).
file_search_path(package,qplib(objects)).
file_search_path(package,qplib(prologbeans)).
file_search_path(package,quintus('qui3.5')).
file_search_path(package,quintus('proxt3.5')).
file_search_path(package,quintus('proxl3.5')).
```

quintus-directory is the root of the Quintus installation hierarchy. It is the directory where Quintus Prolog is installed, and is also returned by

```
| ?- prolog_flag(quintus_directory, QuintusDir).
```

(see Section 8.10.4.1 [ref-lps-flg-cha], page 246, for discussion of `prolog_flag`s). The Prolog flag `host_type` creates the *system* facts.

The path aliases predefined by the `file_search_path/2` facts above have the following interpretation:

`quintus` gives the absolute name of the *quintus-directory*; *quintus-directory* is the root of the Quintus installation hierarchy;

`runtime` set to the value of the `runtime` `prolog_flag/2`; in the Development System, the current working directory is also added as a runtime path;

`system` gives the name of the system specific directories; see Section 8.6.1.5 [ref-fdi-fsp-sys], page 213 below for more discussion of the `system` specific directories;

`helpsys` gives the location of the help-system files; only defined for the Development System;

`qplib` gives the root directory of the Quintus libraries; see the `library_directory/1` facts below;

`library` defined in terms of the `library_directory/1` facts for compatibility with previous releases;

`package` lists Quintus Prolog packages, such as add-ons, for which general file search path facts are defined (e.g. `library`, `helpsys`, `demo`, and `tutorial`);

`messages` gives the location of message files (e.g. '`QU_messages`');

`language` gives the name of the current language specific directory. One language specific directory exists under the embeddability directory in the library. This directory contains, for example, the file '`QU_messages.pl`', which thus can be retrieved using the file specification `messages(language('QU_messages'))`;

`demo` gives the location of the Quintus Prolog demos;

`tutorial` gives the location of the Quintus Prolog tutorials.

Windows note: The `syslib` file search path is provided to allow standard convention for Windows to be followed when searching for DLLs and libraries specified in `load_foreign_executable/1`. At startup time, Prolog asserts `syslib` file search path facts based upon the path specified in the environment variable `PATH` (as well as a couple of standard locations).

When running `qld`, the `syslib` file search path will be initialized to the path specified in the environment variable `LIB` in order to follow the Microsoft linker convention.

Therefore, the directive

```
:- load_foreign_executable(syslib(kernel32)).
```

executed in the Development System will load `kernel32.dll` from a directory in the `PATH` environment variable, whereas if it is encountered by `qld`, the environment variable `LIB` will be used to locate the import library '`kernel32.lib`'.

The syslib file search paths can be modified by user code in the Development System or with the '`-f`' and '`-F`' options to `qld` if necessary.

The library directories defined by the system are:

```
library_directory(qplib(library)).
library_directory(qplib(tools)).
library_directory(qplib('IPC/TCP')).
library_directory(qplib('IPC/RPC')).
library_directory(qplib(embed)).
library_directory(package(library)).
```

Note that these `file_search_path/2` and `library_directory/1` tables, except for `helpsys`, are also defined in `qpc` and `qld` (see Section 20.1.6 [too-too-qpc], page 1489 and Section 20.1.4 [too-too-qld], page 1481).

8.6.1.5 The system file_search_path

The `system` directory is used to store system dependent files. Its main purpose is to allow different platforms sharing the same file system to share system independent files, such as '`.pl`' and '`.qof`' files, but still be able to locate the necessary system dependent files.

Generally, several `system` directories are defined by `file_search_path/2` facts to include different host and operating system combinations. The reason for this is that certain system files are only machine dependent and can be shared by applications running on the same type of machine, but some system files are operating system dependent also.

The most commonly used system files are object files that are loaded by `load_foreign_files/2` and `load_foreign_executable/1` predicates. Quintus Prolog libraries, for example, make frequent use of system dependent foreign code, but also contain system independent files that are shared by all platforms. Therefore, the system dependent object files are stored in a system directory for each platform. The system directories are located under the library directory, and the library files use calls like these:

```
:- load_foreign_files([system('file')], []).

:- load_foreign_executable(system('file')).
```

to specify files located in system directories. The library object files are compiled for each platform at installation time, and placed in the system directory specific to the platform.

The `file_search_path/2` list in Section 8.6.1.4 [ref-fdi-fsp-pre], page 210, shows the different combinations of `system file_search_path` facts for a Sun 4 running SunOS 5.x. The order of search starts at the most specific system definition, which includes the host and full operating system, and proceeds to less specific forms of the system definitions. Finally, if none of the system directories exist, or none contain the specified file, the current working directory is tried. This way, if Quintus Prolog is to run on a single platform, the system dependent files may reside in the same directory as the system independent files.

8.6.1.6 The Library Paths

The library path is special in that there are two methods of establishing and finding library search paths. Although `file_search_path/2` is more general and more powerful, you may choose to define `library_directory/1` if it is adequate for your needs.

The default clauses for `library_directory/1` in the Prolog system can be seen by calling `listing/[0,1]`, or by usint the '`+p library`' option to prolog (see Section 20.1.1 [too-too-prolog], page 1476). They are also shown in Section 8.6.1.4 [ref-fdi-fsp-pre], page 210. To specify an additional directory to be searched before the default ones, add a goal of the form

```
:- asserta(library_directory(Directory)).
```

to your '`prolog.ini`' file. See Section 8.3 [ref-pro], page 186 for a description of '`prolog.ini`' files.

8.6.1.7 Editor Command for Library Search

Under the Emacs interface there is a command ⟨ESC⟩ *x library*, which, when given *File-Name* as an argument, visits the file specified by `library(FileName)`. This facility is provided so one can visit/edit the copy of the file that will be accessed as a result of a library search.

The notes at the end of Section 8.6.1.1 [ref-fdi-fsp-def], page 207 apply here as well.

8.6.2 List of Predicates

- `absolute_file_name/[2,3]`
- `file_search_path/2`
- `library_directory/1`
- `source_file/[1,2,3]`

8.7 Input and Output

8.7.1 Introduction

Prolog provides two classes of predicates for input and output: those that handle individual characters, and those that handle complete Prolog terms.

The following predicates have been added for I/O at the Prolog level:

- `at_end_of_file/[0,1]`
- `at_end_of_line/[0,1]`
- `open/4`
- `peek_char/[1,2]`
- `prompt/3`
- `read_term/[2,3]`
- `seek/4`
- `skip_line/[0,1]`
- `write_term/[1,2]`

Input and output happen with respect to *stream*s. Therefore, this section discusses predicates that handle files and streams in addition to those that handle input and output of characters and terms.

In Quintus Prolog Release 3, the I/O system has been redesigned. Streams are now record based by default, for an increase in efficiency and portability. This leads to increased efficiency in opening streams, putting characters, flushing output, and in character I/O operations involving lines (end of line, new line, skipping lines). For a description of the new model, see Section 10.5.2 [fli-ios-iom], page 434. Combining Prolog and C I/O operations on the same stream is facilitated by a more complete set of functions.

8.7.2 About Streams

A Prolog stream can refer to a file or to the user's terminal[1]. Each stream is used either for input or for output, but not for both. At any one time there is a *current input* stream and a *current output* stream.

Input and output predicates fall into three categories:

1. those that use the current input or output stream;
2. those that take an explicit stream argument;
3. those that use the *standard* input or output stream — these generally refer to the user's terminal. Their names begin with '`tty`'.

Initially, the current input and output streams both refer to the user's terminal. Each input and output built-in predicate refers implicitly or explicitly to a stream. The predicates that perform character and term I/O operations come in pairs such that (A) refers to the current stream, and (B) specifies a stream.

```
predicate_name/n                                    (A)
predicate_name/n+1                                  (B)
```

[1] At the C level, you can define more general streams, e.g. referring to pipes or to encrypted files (see Section 10.5.7.2 [fli-ios-uds-est], page 465).

8.7.2.1 Programming Note

Deciding which version to use involves a trade-off between speed and readability of code: In general, version (B), which specifies a stream, runs slower than (A). So it may be desirable to write code that changes the current stream and uses version (A). However, the use of (B) avoids the use of global variables and results in more readable programs.

8.7.2.2 Stream Categories

Quintus Prolog streams are divided into two categories, those opened by `see/1` or `tell/1` and those opened by `open/[3,4]`. A stream in the former group is referred to by its *file specification*, while a stream in the latter group is referred to by its *stream object* (see the figure "Categorization of Stream Handling Predicates"). For further information about file specifications, see Section 8.6 [ref-fdi], page 205. Stream objects are discussed in Section 8.7.7.1 [ref-iou-sfh-sob], page 226. Reading the state of open streams is discussed in Section 8.7.8 [ref-iou-sos], page 230.

Each operating system permits a different number of streams to be open. For more information on this, refer to Section 2.4 [bas-lim], page 32.

8.7.3 Term Input

Term input operations include:

- reading a term and
- changing the prompt that appears while reading.

8.7.3.1 Reading Terms: The "Read" Predicates

The "Read" predicates are

- `read(-`*`Term`*`)`
- `read(+`*`Stream`*`, -`*`Term`*`)`
- `read_term(+`*`Options`*`, -`*`Term`*`)`
- `read_term(+`*`Stream`*`, +`*`Options`*`, -`*`Term`*`)`

`read_term/[2,3]` offers many options to return extra information about the term.

When Prolog reads a term from the current input stream the following conditions hold:

- The term must be followed by a full-stop. See Section 8.1.8.1 [ref-syn-syn-ove], page 171. The full-stop is removed from the input stream but is not a part of the term that is read.

`read/[1,2]` does not terminate until the full-stop is encountered. Thus, if you type at top level

```
| ?- read(X)
```

you will keep getting prompts (first '`|: `', and five spaces thereafter) every time you type ⟨RET⟩, but nothing else will happen, whatever you type, until you type a full-stop.

- The term is read with respect to current operator declarations. See Section 8.1.5 [ref-syn-ops], page 165, for a discussion of operators.
- When a syntax error is encountered, an error message is printed and then the "read" predicate tries again, starting immediately after the full-stop that terminated the erroneous `term`. That is, it does not fail on a syntax error, but perseveres until it eventually manages to read a term. This behavior can be changed with `prolog_flag/3` or using `read_term/[2,3]`.
- If the end of the current input stream has been reached, then `read(X)` will cause X to be unified with the atom `end_of_file`.

8.7.3.2 Changing the Prompt

To query or change the sequence of characters (prompt) that indicates that the system is waiting for user input, call `prompt/[2,3]`.

This predicate affects only the prompt given when a user's program is trying to read from the terminal (for example, by calling `read/1` or `get0/1`). Note also that the prompt is reset to the default '`|: `' on return to the top level.

8.7.4 Term Output

Term output operations include:

- writing to a stream (various "write" Predicates)
- displaying, usually on the user's terminal (`display/1`)
- changing the effects of `print/1` `portray/1`
- writing a clause as `listing/[0,1]` does. (`portray_clause`)

8.7.4.1 Writing Terms: the "Write" Predicates

- `write(+`*`Stream`*`, +`*`Term`*`)`
- `write(+`*`Term`*`)`
- `writeq(+`*`Stream`*`, +`*`Term`*`)`
- `writeq(+`*`Term`*`)`
- `write_canonical(+`*`Term`*`)`
- `write_canonical(+`*`Stream`*`, +`*`Term`*`)`

- `write_term(+Stream, +Term, +Options)`
- `write_term(+Term, +Options)`

`write_term/[2,3]` is a generalization of the others and provides a number of options.

8.7.4.2 Common Characteristics

The output of the "Write" predicates is not terminated by a full-stop; therefore, if you want the term to be acceptable as input to `read/[1,2]`, you must send the terminating full-stop to the output stream yourself. For example,

```
| ?- write(a), put(0'.), nl.
```

If *Term* is uninstantiated, it is written as an anonymous variable (an underscore followed by a non-negative integer).

`write_canonical/[1,2]` is provided so that *Term*, if written to a file, can be read back by `read/[1,2]` regardless of special characters in *Term* or prevailing operator declarations.

8.7.4.3 Distinctions Among the "write" Predicates

- With `write` and `writeq` the term is written with respect to current operator declarations (See Section 8.1.5 [ref-syn-ops], page 165, for a discussion of operators).

  `write_canonical(Term)` writes *Term* to the current or specified output stream in standard syntax (see Section 8.1 [ref-syn], page 159 on Prolog syntax), and quotes atoms and functors to make them acceptable as input to `read/[1,2]`. That is, operator declarations are not used and compound terms are therefore always written in the form:

  ```
  predicate_name(arg1, ..., argn)
  ```

- Atoms output by `write/[1,2]` cannot in general be read back using `read/[1,2]`. For example,

  ```
  | ?- write('a b').
  a b
  ```

  If you want to be sure that the atom can be read back by `read/[1,2]`, you should use `writeq/[1,2]`, or `write_canonical/[1,2]`, which put quotes around atoms when necessary, or use `write_term/[2,3]` with the `quoted` option set to `yes`.

- `write/[1,2]` and `writeq/[1,2]` treat terms of the form `'$VAR'(N)` specially: they write '`A`' if N=0, '`B`' if N=1, ...'`Z`' if N=25, '`A1`' if N=26, etc. Terms of this form are generated by `numbervars/3` (see Section 8.9.5 [ref-lte-anv], page 241). Terms of the form `'$VAR'(X)`, where X is not a number are written as unquoted terms. For example,

  ```
  | ?- writeq(a('$VAR'(0),'$VAR'('Test'))).
  a(A,Test)
  ```

  `write_canonical/1` does *not* treat terms of the form `'$VAR'(N)` specially. It writes square bracket lists using `./2` and `[]` (that is, `[a,b]` is written as '`.(a,.(b,[]))`'). If

the `character_escapes` flag is `on` then `write_canonical/1` tries to write layout characters (except ASCII 9 and ASCII 32) in the form '`\`*`lower-case-letter`*', if possible; otherwise, `write_canonical/1` writes the '`\^`*`control-char`*' form. If the `character_escapes` flag is `off` then it writes the actual character, without using an escape sequence (see Section 8.1.4 [ref-syn-ces], page 163).

- Depending upon whether character escaping is on or off, `writeq/[1,2]` and `write_canonical/[1,2]` behave differently when writing quoted atoms. If character escaping is on:
  1. The characters with ASCII codes 9 (horizontal tab), 32 (space), and 33 through 126 (non-layout characters) are written as themselves.
  2. The characters with ASCII codes 8, 10, 11, 12, 13, 27, and 127 are written in their '`\`*`lowercase letter`*' form.
  3. The character with ASCII code 39 (single quote) is written as two consecutive single quotes.
  4. The character with ASCII code 92 (back slash) is written as two consecutive back slashes.
  5. All other characters are written in their '`\^`*`control char`*' form.

  If character escaping is off:
  1. The character with ASCII code 39 (single quote) is written as two consecutive single quotes.
  2. All other characters are written as themselves.
  3. In general, one can only read (using `read/[1,2]`) a term written by `write_canonical/[1,2]` if the value of the `character_escapes` flag is the same when the term is read as when it was written.

8.7.4.4 Displaying Terms

Like `write_canonical`, `display/1` ignores operator declarations and shows all compound terms in standard prefix form. For example, the command

```
| ?- display(a+b).
```

produces the following:

```
+(a,b)
```

Calling `display/1` is a good way of finding out how Prolog parses a term with several operators. Unlike `write_canonical/[1,2]`, `display/1` does not put quotes around atoms and functors.

8.7.4.5 Using the 'portray' hook

`print/1` is called from within the system in two places:

1. to print the bindings of variables after a question has succeeded
2. to print a goal during debugging

By default, the effect of `print/1` is the same as that of `write/1`, but you can change its effect by providing clauses for the hook predicate `portray/1`.

If *X* is a variable, then it is printed using `write(X)`. Otherwise the user-definable procedure `portray(X)` is called. If this succeeds, then it is assumed that *X* has been printed and `print/1` exits (succeeds). Note that `print/1` always calls `portray/1` in module `user`. Therefore, to be visible to `print/1`, `portray/1` must either be defined in or imported into module `user`.

If the call to `portray/1` fails, and if *X* is a compound term, then `write/1` is used to write the principal functor of *X* and `print/1` is called recursively on its arguments. If *X* is atomic, it is written using `write/1`.

When `print/1` has to print a list, say `[X1,X2,...,Xn]`, it passes the whole list to `portray/1`. As usual, if `portray/1` succeeds, it is assumed to have printed the entire list, and `print/1` does nothing further with this term. Otherwise `print/1` writes the list using bracket notation, calling `print/1` on each element of the list in turn.

Since `[X1,X2,...,Xn]` is simply a different way of writing `.(X1,[X2,...,Xn])`, one might expect `print/1` to be called recursively on the two arguments `X1` and `[X2,...,Xn]`, giving `portray/1` a second chance at `[X2,...,Xn]`. This does *not* happen; lists are a special case in which `print/1` is called separately for each of `X1,X2,...Xn`.

If you would like lists of character codes printed by `print/1` using double-quote notation, you should include `library(printchars)` (described in Chapter 12 [lib], page 521) as part of your version of `portray/1`.

Often it is desirable to define clauses for `portray/1` in different files. This can be achieved either by declaring it multifile in each of the files, or by using `library(addportray)`.

8.7.4.6 Portraying a Clause

If you want to print a clause, `portray_clause/1` is almost certainly the command you want. None of the other term output commands puts a full-stop after the written term. If you are writing a file of facts to be loaded by `compile/1`, use `portray_clause/1`, which attempts to ensure that the clauses it writes out can be read in again as clauses.

The output format used by `portray_clause/1` and `listing/1` has been carefully designed to be clear. We recommend that you use a similar style. In particular, never put a semicolon (disjunction symbol) at the end of a line in Prolog.

8.7.5 Character Input

8.7.5.1 Overview

> **Please note:** For compatibility with DEC-10 Character I/O a set of predicates are provided, which are similar to the primary ones except that they always use the standard input and output streams, which normally refer to the user's terminal rather than to the current input stream or current output stream. They are easily recognizable as they all begin with '`tty`'.
>
> Given stream-based input/output, these predicates are actually redundant. For example, you could write `get0(user, C)` instead of `ttyget0(C)`.
>
> This note applies to the character output as well (see Section 8.7.6 [ref-iou-cou], page 222).

The operations in this category are:

- reading characters ("get" predicates),
- peeking
- skipping
- checking for end of line or end of file

8.7.5.2 Reading Characters

`get0(`*N*`)` unifies *N* with the ASCII code of the next character from the current input stream. Cf. `ttyget0/1`.

`get(`*N*`)` unifies *N* with the ASCII code of the next non-layout character from the current input stream. Layout characters are all outside the inclusive range 33..126; this includes space, tab, linefeed, delete, and all control characters. Cf. `ttyget/1`.

8.7.5.3 Peeking

Peeking at the next character without consuming it is useful when the interpretation of "this character" depends on what the next one is. Use `peek_char/[1,2]`.

8.7.5.4 Skipping

There are two ways of skipping over characters in the current input stream: skip to a given character, or skip to the end of a line (or record).

- To skip over characters from the current input or specified stream through the first occurrence of the character with ASCII code *N*, (see Section 18.3.169 [mpg-ref-skip], page 1293) Cf. `ttyskip/1`.

- Generally it is more useful to skip to the end of a line using `skip_line/[0,1]`. Use of this predicate helps portability of code since it avoids dependence on any particular character code(s) being returned at the end of a line.

8.7.5.5 Finding the End of Line and End of File

To test whether the end of a line on the end of the file has been reached on the current or specified input stream, use `at_end_of_line/[0,1]` or `at_end_of_file/[0,1]`.

8.7.6 Character Output

The character output operations are:

- writing (putting) characters
- creating newlines and tabs
- flushing buffers
- formatting output.

> **Please note:** The note about "tty-" predicates at the beginning of Section 8.7.5 [ref-iou-cin], page 220 applies here as well.

8.7.6.1 Writing Characters

`put(`*N*`)` writes *N* to the current output stream or `put(`*Stream, N*`)` writes *N* to a specified one, *Stream*. *N* should be a legal ASCII character code or an integer expression.

If *N* evaluates to an integer, the least significant 8 bits are written.

The character is not necessarily printed immediately; they may be flushed if the buffer is full. See Section 8.7.7.9 [ref-iou-sfh-flu], page 230.

Cf. `ttyput/1`.

8.7.6.2 New Line

`nl` and `nl(`*Stream*`)` terminates the current output record on the current output stream or on a specified one, *Stream*. If the stream format is `delimited(lf)` or `delimited(tty)` (the default), a linefeed character (ASCII) is printed.

Cf. `ttynl/0`.

8.7.6.3 Tabs

`tab(N)` writes N spaces to the current output stream. N may be an integer expression.

The spaces are not necessarily printed immediately; see Section 8.7.7.9 [ref-iou-sfh-flu], page 230.

Cf. `ttytab/1`

8.7.6.4 Formatted Output

`format(Control, Arguments)` interprets the *Arguments* according to the *Control* string and prints the result on the current output stream. A stream can be specified using `format/3`.

This is used to produce output like this, either on the current output or on a specified stream:

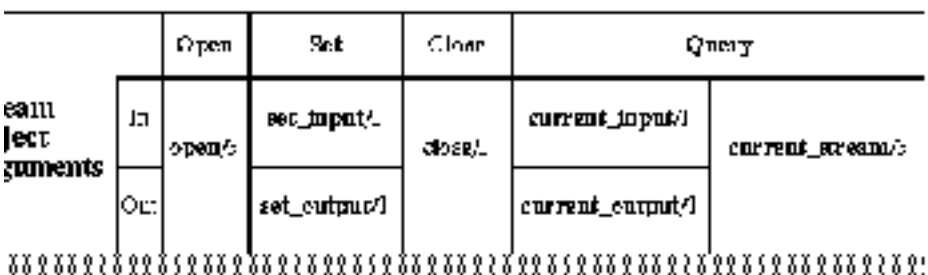
```
| ?- toc(1.5).
Table of Contents                                                   i

                            Table of Contents

1. Documentation supplement for Quintus Prolog Re-
lease 1.5 ........... 2

   1-1 Defini-
tion of the term "loaded" ............................... 2
   1-2 Finding all solu-
tions ..................................... 3
   1-3 Searching for a file in a li-
brary ............................ 4
   1-4 New Built-in Predi-
cates ..................................... 5
       1-4-
1 write_canonical (?Term) ................................ 5
                     .
                     .
                     .
   1-7 File Specifica-
tions ..................................... 17
       1-7-1 multi-
file(+PredSpec) ................................ 18

yes
```

For details, including the code to produce this example, see the example program in the reference page for `format/[2,3]`. The character escaping facility is also used.

8.7.7 Stream and File Handling

The operations implemented are opening, closing, querying status, flushing, error handling, setting.

The predicates in the "see" and "tell" families are supplied for DEC-10 Prolog compatibility. They take either file specifications or stream objects as arguments (see Section 18.1 [mpg-ref], page 985) and they specify an alternative, less powerful, mechanism for dealing with files and streams than the similar predicates (`open/[3,4]`, etc.), which take stream objects (see the figure "Categorization of Stream Handling Predicates").


Open
Set
Close
Query
ea111
lect
guments
In
Out
open/3
set_input/1
set_output/1
close/1
current_input/1
current_output/1
current_stream/3

Categorization of Stream Handling Predicates

8.7.7.1 Stream Objects

Each input and output stream is represented by a unique Prolog term, a *stream_object*. In general, this term is of the form

```
'$stream'(X).
```

where X is an integer. In addition, the following terms are used to identify the standard I/O streams:

- `user_input`
- `user_output`
- `user_error`

Stream objects are created by the predicate `open/[3,4]` Section 8.7.7.4 [ref-iou-sfh-opn], page 227and passed as arguments to those predicates that need them. Representation for stream objects to be used in C code is different. Use `stream_code/2` to convert from one to the other when appropriate.

8.7.7.2 Exceptions related to Streams

All predicates that take a stream argument will raise the following exceptions:

`instantiation_error`
: *Stream* argument is not ground

`type_error`
: *Stream* is not an input (or output) stream type.

`existence_error`
: *Stream* is syntactically valid but does not name an open stream.

`permission_error`
: *Stream* names an open stream but the stream is not open for input (or output).

The reference page for each stream predicate will simply refer to these as "Stream errors" and will go on to detail other exceptions that may be raised for a particular predicate.

8.7.7.3 Suppressing Error Messages

`nofileerrors/0` resets the `fileerrors` flag, so that the built-in predicates that open files simply fail, instead of raising an exception if the specified file cannot be opened.

To cancel the effect of `fileerrors/0`, call `nofileerrors/0`. It sets the `fileerrors` flag to its default state, `on`, in which an error message is produced by `see/1`, `tell/1`, and `open/3` if the specified file cannot be opened. The error message is followed by an `abort/0`; that is, execution of the program is abandoned and the system returns to top level.

The `fileerrors` flag is only enabled or disabled by an explicit call to `fileerrors/0` or `nofileerrors/0`, or via `prolog_flag/[2,3]`, which can also be used to obtain the current value of the `fileerrors` flag. See Section 8.10.1 [ref-lps-ove], page 245, for more information on the `fileerrors` flag.

8.7.7.4 Opening a Stream

Before I/O operations can take place on a stream, the stream must be opened, and it must be set to be current input or current output. As illustrated in the figure "Categorization of Stream Handling Predicates", the operations of opening and setting are separate with respect to the stream predicates, and combined in the File Specification Predicates.

- `open(+`*`File`*`, +`*`Mode`*`, -`*`Stream`*`)` attempts to open the file *File* in the mode specified (read,write or append). If the `open/3` request is successful, a stream object, which can be subsequently used for input or output to the given file, is unified with *Stream*.

  The `read` mode is used for input. The `write` and `append` modes are used for output. The `write` option causes a new file to be created for output. If the file already exists, then it is set to empty and its previous contents are lost. The `append` option opens an already-existing file and adds output to the end of it. The `append` option will create the file if it does not already exist.

  Options can be specified by calling `open/4`.
- `set_input(`*`Stream`*`)` makes *Stream* the current input stream. Subsequent input predicates such as `read/1` and `get0/1` will henceforth use this stream.
- `set_output(`*`Stream`*`)` makes *Stream* the current output stream. Subsequent output predicates such as `write/1` and `put/1` will henceforth use this stream.

Opening a stream and making it current are combined in `see` and `tell`:

- `see(`*`S`*`)` makes file *S* the current input stream. If *S* is an atom, it is taken to be a file specification, and
  - if there is an open input stream associated with the filename, and that stream was opened by `see/1`, then it is made the current input stream;
  - Otherwise, the specified file is opened for input and made the current input stream. If it is not possible to open the file, `see/1` fails. In addition, if the `fileerrors` flag is set (as it is by default), `see/1` sends an error message to the standard error stream and calls `abort/0`, returning to the top level.
- `tell(`*`S`*`)` makes *S* the current output stream.
  - if there is an open output stream currently associated with the filename, and that stream was opened by `tell/1`, then it is made the current output stream;

– Otherwise, the specified file is opened for output and made the current output stream. If the file does not exist, it is created. If it is not possible to open the file (because of protections, for example), `tell/1` fails. In addition, if the `fileerrors` flag is set (which it is by default), `tell/1` sends an error message to the standard error stream and calls `abort/0`, returning to the top level.

It is important to remember to close streams when you have finished with them. Use `seen/0` or `close/1` for input files, and `told/0` or `close/1` for output files.

- `open_null_stream(`*`Stream`*`)` opens an output stream that is not connected to any file and unifies its stream object with *Stream*. Characters or terms that are sent to this stream are thrown away. This predicate is useful because various pieces of local state are kept for null streams: the predicates `character_count/2`, `line_count/2`, and `line_position/2` can be used on these streams (see Section 8.7.8 [ref-iou-sos], page 230).

8.7.7.5 Finding the Current Input Stream

- `current_input(`*`Stream`*`)` unifies *Stream* with the current input stream.
- `seeing(`*`S`*`)` unifies *S* with the current input stream. This is exactly the same as `current_input`(*S*), except that *S* will be unified with a filename if the current input stream was opened by `see/1` (see Section 8.7.7.4 [ref-iou-sfh-opn], page 227).

  `seeing/1` can be used to verify that FileNameOrStream is still the current input stream as follows:

  ```
  /* nonvar(FileNameOrStream), */
  see(FileNameOrStream),
  ...
  seeing(FileNameOrStream)
  ```

  If the current input stream has not been changed (or if changed, then restored), the above sequence will succeed for all file names and all stream objects opened by `open/[3,4]`. However, it will fail for all stream objects opened by `see/1` (since only filename access to streams opened by `see/1` is supported). This includes the stream object `user_input` (since the standard input stream is assumed to be opened by `see/1`, and so `seeing/1` would return `user` in this case).

  `seeing/1` can be followed by `see/1` to ensure that a section of code leaves the current input unchanged

8.7.7.6 Finding the current output stream

- `current_output(`*`Stream`*`)` unifies *Stream* with the current output stream.
- `telling(`*`S`*`)` unifies *S* with the current output stream. This is exactly the same as `current_output`(*S*), except that *S* will be unified with a filename if the current output stream was opened by `tell/1`.

  A common usage of `telling/1` is

```
tell('Some File Name')
...
telling('Some File Name')
```

`telling/1` should succeed if the current input stream was not changed (or if changed, restored). It succeeds for any filename (including `user`) and any stream object opened by `open/3` (see Section 8.7.7.4 [ref-iou-sfh-opn], page 227), but fails for `user_output` and any stream object opened by `tell/1` (see Section 8.7.7.4 [ref-iou-sfh-opn], page 227). Passing file names to `tell/1` is the only DEC-10 Prolog usage of `telling/1`, so Quintus Prolog is compatible with this usage.

WARNING: The sequence

```
telling(File),
...
set_output(File),
```

will signal an error if the current output stream was opened by `tell/1`. The only sequences that are guaranteed to succeed are

```
telling(FileOrStream),
...
tell(FileOrStream)
```

and

```
current_output(Stream),
...
set_output(Stream)
```

8.7.7.7 Backtracking through Open Streams

- `current_stream(*File, *Mode, *Stream)` succeeds if *Stream* is a stream that is currently open on file *File* in mode *Mode*, where *Mode* is either `read`, `write`, or `append`. None of the arguments need be initially instantiated. This predicate is nondeterminate and can be used to backtrack through all open streams. It fails when there are no (further) matching open streams.

  `current_stream/3` ignores the three special streams for the standard input, output, and error channels.

8.7.7.8 Closing a Stream

- `close(X)` closes the stream corresponding to *X*.

  If *X* is a stream object, then if the corresponding stream is open, it will be closed; otherwise, `close/1` succeeds immediately, taking no action.

  If *X* is a file specification, the corresponding stream will be closed. It is only closed if the file was opened by `see/1` or `tell/1`. In the example

  ```
  see(foo),
  ...
  close(foo)
  ```

  'foo' will be closed. However, in the example

```
open(foo, read, S),
...
close(foo)
```

an exception will be raised and '`foo`' will not be closed.

- `told/0` closes the current output stream. The current output stream is then set to be `user_output`; that is, the user's terminal.
- `seen/0` closes the current input stream. The current input stream is then set to be `user_input`; that is, the user's terminal.

8.7.7.9 Flushing Output

Output to a stream is not necessarily sent immediately; it is buffered. The predicate `flush_output/1` flushes the output buffer for the specified stream and thus ensures that everything that has been written to the stream is actually sent at that point.

- `flush_output(`*`Stream`*`)` sends all data in the output buffer to stream *Stream*.
- `ttyflush/0` is equivalent to `flush_output(user)`.

8.7.8 Reading the State of Opened Streams

Character count, line count and line position for a specified stream are obtained as follows:

- `character_count(`*`Stream, N`*`)` unifies *N* with the total number of characters either read or written on the open stream *Stream*.
- `line_count(`*`Stream, N`*`)` unifies *N* with the total number of lines either read or written on the open stream *Stream*. A freshly opened stream has a line count of 1.
- `line_position(`*`Stream, N`*`)` unifies *N* with the total number of characters either read or written on the current line of the open stream *Stream*. A fresh line has a line position of 0.

8.7.8.1 Stream Position Information for Terminal I/O

Input from Prolog streams that have opened the user's terminal for reading is echoed back as output to the same terminal. This is interleaved with output from other Prolog streams that have opened the user's terminal for writing. Therefore, all streams connected to the user's terminal share the same set of position counts and thus return the same values for each of the predicates `character_count/2`, `line_count/2`, and `line_position/2`. The following example assumes that `user_input`, `user_output`, and `user_error` are all connected to the user's terminal (which may not always be true if I/O is being redirected),

```
| ?- line_count(user, X1),
     line_count(user_input, X2),
     line_count(user_output, X3),
     line_count(user_error, X4).

X1 = X2 = X3 = X4 = 36 ;

no
| ?- line_position(user, X1),
     line_position(user_input, X2),
     line_position(user_output, X3),
     line_position(user_error, X4).

X1 = X2 = X3 = X4 = 0 ;

no
| ?- character_count(user, X1),
     character_count(user_input, X2),
     character_count(user_output, X3),
     character_count(user_error, X4).

X1 = X2 = X3 = X4 = 1304 ;

no
```

8.7.9 Random Access to Files

There are two methods of finding and setting the stream position, *stream positioning* and *seeking*. The current position of the read/write pointer in a specified stream can be obtained by using `stream_position/2`. It may be changed by using `stream_position/3`. Alternatively, `seek/4` may be used.

Seeking is more general, and stream positioning is more portable. The differences between them are:

- `stream_position/2` is equivalent to `seek/4` with *Offset* = 0, and *Method* = `current`.
- Where `stream_position/3` asks for stream position objects, `seek/4` uses integer expressions to represent the position or offset. Stream position objects are obtained by calling `stream_position/[2,3]`, and are discussed in the reference page.
- `seek/4` is supported only on certain operating systems. `stream_position/3` is portable.

8.7.10 Summary of Predicates and Functions

Reference pages for the following provide further detail on the material in this section.

- at_end_of_file/[0,1]
- at_end_of_line/[0,1]
- character_count/2
- close/1
- current_input/1
- current_output/1
- current_stream/3
- display/1
- fileerrors/0
- flush_output/1
- format/[2,3]
- get0/[1,2]
- get/[1,2]
- line_count/2
- line_position/2
- nl/[0,1]
- nofileerrors/0
- open/[3,4]
- open_null_stream/1
- peek_char/[1,2]
- portray/1
- portray_clause/1
- print/[1,2]
- prompt/[2,3]
- put/[1,2]
- read/[1,2]
- read_term/[2,3]
- see/1
- seeing/1
- seek/4
- seen/0
- set_input/1
- set_output/1
- skip/[1,2]
- skip_line/[0,1]

- `stream_position/[2,3]`
- `tab/[1,2]`
- `tell/1`
- `telling/1`
- `told/0`
- `ttyflush/0`
- `ttyget0/1`
- `ttyget/1`
- `ttynl/0`
- `ttyput/1`
- `ttyskip/1`
- `ttytab/1`
- `write/[1,2]`
- `write_canonical/[1,2]`
- `writeq/[1,2]`
- `write_term/[2,3]`

Also, see Section 10.5 [fli-ios], page 433.

8.7.11 Library Support

- `library(addportray)`
-
- `library(printchars)`

8.8 Arithmetic

8.8.1 Overview

In Prolog, arithmetic is performed by certain built-in predicates, which take arithmetic expressions as their arguments and evaluate them. Arithmetic expressions can evaluate to integers or floating-point numbers (floats).

With release 3 Quintus Prolog has full 32 bit integer arithmetic and full 64 bit double precision floating point arithmetic. The range of integers is -2147483648 (-2^31) to 2147483647 (2^31-1) both inclusive. Arithmetic operations like integer addition and multiplication raise a representation error if there is an overflow.

The range of floating-point numbers is approximately 2.3E-308 to 1.7E+308. Floats are represented by 64 bits and they conform to the IEEE 754 standard. The behavior on floating-point overflow or underflow is machine-dependent.

Chapter summary: The arithmetic operations of evaluation and comparison are implemented in the predicates described in Section 8.8.2 [ref-ari-eae], page 234 and Section 8.8.3 [ref-ari-acm], page 234. All of them take arguments of the type *Expr*, which is described in detail in Section 8.8.4 [ref-ari-aex], page 235.

8.8.2 Evaluating Arithmetic Expressions

The most common way to do arithmetic calculations in Prolog is to use the built-in predicate `is/2`.

`-`*Term* `is` `+`*Expr*
: *Term* is the value of arithmetic expression *Expr*.

Term must not contain any uninstantiated variables. Do not confuse `is/2` with `=/2`.

8.8.3 Arithmetic Comparison

Each of the following predicates evaluates each of its arguments as an arithmetic expression, then compares the results. If one argument evaluates to an integer and the other to a float, the integer is coerced to a float before the comparison is made.

Note that two floating-point numbers are equal if and only if they have the same bit pattern. Because of rounding error, it is not normally useful to compare two floats for equality.

Expr1 `=:=` *Expr2*
: succeeds if the results of evaluating terms *Expr1* and *Expr2* as arithmetic expressions are equal

Expr1 `=\=` *Expr2*
: succeeds if the results of evaluating terms *Expr1* and *Expr2* as arithmetic expressions are not equal

Expr1 `<` *Expr2*
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is less than the result of evaluating *Expr2* as an arithmetic expression.

Expr1 `>` *Expr2*
: succeeds if the result of evaluating *Expr1* as an arithmetic expression *Expr1* is greater than the result of evaluating *Expr2* as an arithmetic expression.

Expr1 `=<` *Expr2*
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is not greater than the result of evaluating *Expr2* as an arithmetic expression.

`Expr1 >= Expr2`
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is not less than the result of evaluating *Expr2* as an arithmetic expression.

8.8.4 Arithmetic Expressions

Arithmetic evaluation and testing is performed by predicates that take arithmetic expressions as arguments. An *arithmetic expression* is a term built from numbers, variables, and functors that represent arithmetic functions. These expressions are evaluated to yield an arithmetic result, which may be either an integer or a float; the type is determined by the rules described below.

At the time of evaluation, each variable in an arithmetic expression must be bound to a number or another arithmetic expression. If the expression is not sufficiently bound or if it is bound to terms of the wrong type then Prolog raises exceptions of the appropriate type (see Section 8.19.3 [ref-ere-hex], page 312). Some arithmetic operations can also detect overflows. They also raise exceptions. e.g. Division by zero results in a domain error being raised.

Only certain functors are permitted in arithmetic expressions. These are listed below, together with a description of their arithmetic meanings. For the rest of the section, *X* and *Y* are considered to be arithmetic expressions.

8.8.4.1 Arithmetic calculations

`X + Y`
: Evaluates to the sum of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, the result is a float. If integer addition results in an overflow, a representation error is raised.

`X - Y`
: Evaluates to the difference of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, the result is a float. If integer subtraction results in an underflow, a representation error is raised.

`X * Y`
: Evaluates to the product of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, the result is a float. If integer multiplication results in an overflow, a representation error is raised.

`- X`
: Evaluates to the negative of *X*. The type of the result, integer or float, is the same as the type of the operand.

`abs(X)`
: Evaluates to *X* if *X* is a positive number, -*X* if it is a negative number.

`X / Y`
: Evaluates to the quotient of *X* and *Y*. The result is always a float, regardless of the types of the operands *X* and *Y*. Attempt to divide by zero results in a domain error being raised.

`X // Y`
: Evaluates to the integer quotient of *X* and *Y*. *X* and *Y* must both be integers. The result is truncated to the nearest integer that is between it and 0. Attempt to divide by zero results in a domain error being raised.

`X div Y` Equivalent to '`//`'.

`X mod Y` Evaluates to the remainder after the integer division of X by Y. X and Y must both be integers. The result, if non-zero, has the same sign as X. If Y evaluates to 0, a domain error is raised.

`integer(X)`
Evaluates to X if X is an integer. Otherwise (if X is a float) the result is the nearest integer that is between it and 0.

`float(X)` Evaluates to X if X is a float. Otherwise (if X is an integer) the result is the floating-point equivalent of X.

`min(X,Y)` Evaluates to the minimum of X and Y.

`max(X,Y)` Evaluates to the maximum of X and Y.

8.8.4.2 Peeking into Memory

The following operations can be used to peek into memory. They can be used in conjunction with the foreign interface to peek into data structures within foreign code from Prolog. These operations take an integer argument and access the data stored at the address represented by the argument. Note that these operations can result in segmentation faults and bus errors if the argument you are trying to access is a bad address or if the address is not aligned properly for the data you are going to access from it. The only sure way of getting an integer in Prolog that represents an address that makes sense is by returning an address from a foreign function through the foreign language interface (see Section 10.3.9 [fli-p2f-poi], page 397). For built-ins that poke ("store") values into memory, see the reference page for `assign/2` in the reference section. For more structured ways of doing this, see the Structs and Objects packages.

`integer_8_at(X)`
Evaluates to the signed byte stored at address X.

`unsigned_8_at(X)`
Evaluates to the unsigned byte stored at address X.

`integer_16_at(X)`
Evaluates to the signed short stored at address X.

`unsigned_16_at(X)`
Evaluates to the unsigned short stored at address X.

`integer_at(X)`
Evaluates to the signed integer stored at address X.

`address_at(X)`
Evaluates to the address stored at address X.

`single_at(X)`
Evaluates to the single precision floating point number stored at address X.

`double_at(X)`
: Evaluates to the double precision floating point number stored at address X.

8.8.4.3 Bit-vector Operations

The following bit-vector operations apply to integer arguments only. Supplying non-integer arguments results in an exception being raised. Note that the slant lines used in these operator names are produced with the forward and backward slash keys.

`X /\ Y` Evaluates to the bitwise conjunction of X and Y.

`X \/ Y` Evaluates to the bitwise disjunction of X and Y.

`\(X,Y)` Evaluates to the bitwise xor of X and Y. Note that this is not an operator.

`\ (X)` Evaluates to the complement of the bits in X.

`X << Y` X is shifted left Y places. Equivalent to `X << (Y /\ 2'11111)`.

`X >> Y` X is shifted right Y places with sign extension.

8.8.4.4 Character Codes

The following operation is included in order to allow integer arithmetic on character codes.

`[X]`
: Evaluates to X for numeric X. This is relevant because character strings in Prolog are lists of character codes, that is, integers. Thus, for those integers that correspond to character codes, the user can write a string of one character in place of that integer in an arithmetic expression. For example, the expression (A) is equivalent to (B), which in turn becomes (C) in which case X is unified with 2:

```
X is "c" - "a"          (A)
X is [99] - [97]        (B)
X is 99 - 97            (C)
```

A cleaner way to do the same thing is

```
X is 0'c - 0'a
```

8.8.5 Predicate Summary

- `is/2`
- `=:=/2`
- `=\=/2`
- `</2`

- `>//2`
- `=</2`
- `>=/2`

8.8.6 Library Support

Additional arithmetic predicates can be found in `library(math)`.

8.9 Looking at Terms

8.9.1 Meta-logical Predicates

Meta-logical predicates are those predicates that allow you to examine the current instantiation state of a simple or compound term, or the components of a compound term. This section describes the meta-logical predicates as well as others that deal with terms as such.

8.9.1.1 Type Checking

The following predicates take a term as their argument. They are provided to check the type of that term.

Predicate	**Succeeds if term is:**
`var/1`	a variable; the term is currently uninstantiated.
`nonvar/1`	a non-variable; the term is currently instantiated.
`integer/1`	an integer
`atom/1`	an atom
`float/1`	a float
`number/1`	an integer or float.
`atomic/1`	an atom, number or database reference.
`simple/1`	an atom, number, variable or database reference.
`compound/1`	a compound term (arity > 0).
`callable/1`	a term that `call/1` would take as an argument; atom or compound term.
`ground/1`	ground; the term contains no uninstantiated variables.

`db_reference/1`
: a Prolog database reference

  Please note: Although database references are read and written as compound terms, and formerly were, they now are a distinct atomic term type (see Section 8.14.3 [ref-mdb-dre], page 288).

The reference pages for these predicates include examples of their use.

8.9.1.2 Unification and Subsumption

To unify two items simply use `=/2`, which is defined as if by the clause

```
=(X, X).
```

Please note: Do not confuse this predicate with `=:=/2` (arithmetic comparison) or `==/2` (term identity).

Term subsumption is a sort of one-way unification. Term S and T unify if they have a common instance, and unification in Prolog instantiates both terms to that common instance. S *subsumes* T if T is already an instance of S. For our purposes, T is an *instance* of S if there is a substitution that leaves T unchanged and makes S identical to T.

Subsumption is checked by `subsumes_chk/2`. It is especially useful in applications such as theorem provers. The built-in predicate behaves identically to the original library version but is much more efficient.

Related predicates are defined in `library(subsumes)` and `library(occurs)`. (For information on these packages see Chapter 12 [lib], page 521).

8.9.2 Analyzing and Constructing Terms

The built-in predicate `functor/3` performs these functions

- Decomposes a given term into its name and arity or
- given a name and arity, it constructs the corresponding compound term creating new uninstantiated variables for its arguments.

The built-in predicate `arg/3` performs these functions:

- Unifies a term with a specified argument of another term.

The built-in predicate *Term* `=..` *List* performs these functions:

- The built-in predicate `=../2` (otherwise known as "univ") unifies *List* with a list whose head is the atom corresponding to the principal functor of *Term* and whose tail is a list of the arguments of *Term*.

8.9.3 Analyzing and Constructing Lists

To combine two lists to form a third list, use `append(+Head, +Tail, -List)`.

To analyze a list into its component lists in various ways, use `append/3` with *List* instantiated to a proper list. The reference page for `append/3` includes examples of its usage, including backtracking.

To check the length of a list call `length(+List, -Integer)`.

To produce a list of a certain length, use `length/2` with *Integer* instantiated and *List* uninstantiated or instantiated to a list whose tail is a variable.

8.9.4 Converting between Constants and Text

Three predicates convert between constants and lists of ASCII character codes: `atom_chars/2`, `number_chars/2`, and `name/2`.

There is a general convention that a predicate that converts objects of type *foo* to objects of type *baz* should have one of these forms:

```
foo_to_baz(+Foo, -Baz)                                    (1)

foo_baz(?Foo, ?Baz)                                       (2)
```

Use (1) if the conversion works only one way, or (2) if for any *Foo* there is exactly one related *Baz* and for any *Baz* at most one *Foo*.

The type name used for lists of ASCII character codes is *chars* thus, the predicate that relates an atom to its name is `atom_chars(?Atom, ?Chars)`, and the predicate that relates a number to its textual representation is `number_chars(?Number, ?Chars)`.

`atom_chars`(*Atom*, *Chars*) is a relation between an atom *Atom* and a list *Chars* consisting of the ASCII character codes comprising the printed representation of *Atom*. Initially, either *Atom* must be instantiated to an atom, or *Chars* must be instantiated to a proper list of character codes.

`number_chars`(*Number*, *Chars*) is a relation between a number *Number* and a list *Chars* consisting of the ASCII character codes comprising the printed representation of *Number*. Initially, either *Number* must be instantiated to a number, or *Chars* must be instantiated to a proper list of character codes.

`name/2` converts from any sort of constant to a chars representation. Given a chars value, `name/2` will convert it to a number if it can, otherwise to an atom. This means that there are atoms that can be constructed by `atom_chars/2` but not by `name/2`. `name/2` is retained for backwards compatibility with DEC-10 Prolog and C-Prolog. New programs should use `atom_chars/2` or `number_chars/2` as appropriate.

8.9.5 Assigning Names to Variables

Each variable in a term is instantiated to a term of the form `'$VAR'(`*N*`)`, where *N* is an integer, by the predicate `numbervars/2`. The "write" predicates (`write/1`, `writeq/1`, and `write_term/2` with the numbervars option set to `true`) transform these terms into upper case letters.

8.9.6 Copying Terms

The meta-logical predicate `copy_term/2` makes a copy of a term in which all variables have been replaced by new variables that occur nowhere else. This is precisely the effect that would have been obtained from the definition

```
copy_term(Term, Copy) :-
   recorda(copy, copy(Term), DBref),
   instance(DBref, copy(Temp)),
   erase(DBref),
   Copy = Temp.
```

although the built-in predicate `copy_term/2` is more efficient.

When you call `clause/[2,3]` or `instance/2`, you get a new copy of the term stored in the database, in precisely the same sense that `copy_term/2` gives you a new copy. One of the uses of `copy_term/2` is in writing interpreters for logic-based languages; with `copy_term/2` available you can keep "clauses" in a Prolog data structure and pass this structure as an argument without having to store the "clauses" in the Prolog database. This is useful if the set of "clauses" in your interpreted language is changing with time, or if you want to use clever indexing methods.

A naive way to attempt to find out whether one term is a copy of another is shown in this example:

```
identical_but_for_variables(X, Y) :-
   \+ \+ (
      numbervars(X, 0, N),
      numbervars(Y, 0, N),
      X = Y
   ).
```

This solution is sometimes sufficient, but will not work if the two terms have any variables in common. If you want the test to succeed even when the two terms do have some variables in common, you need to copy one of them; for example,

```
identical_but_for_variables(X, Y) :-
   \+ \+ (
      copy_term(X, Z),
      numbervars(Z, 0, N),
      numbervars(Y, 0, N),
      Z = Y
   ).
```

`copy_term/2` is efficient enough to use without hesitation if there is no solution that does not require the use of meta-logical predicates. However, for the sake of both clarity and efficiency, such a solution should be sought before using `copy_term/2`.

8.9.7 Comparing Terms

8.9.7.1 Introduction

The predicates are described in this section used to compare and order terms, rather than to evaluate or process them. For example, these predicates can be used to compare variables; however, they never instantiate those variables. These predicates should not be confused with the arithmetic comparison predicates (see Section 8.8.3 [ref-ari-acm], page 234) or with unification.

8.9.7.2 Standard Order of Terms

These predicates use a standard total order when comparing terms. The standard total order is:

variables `@<` database references `@<` numbers `@<` atoms `@<` compound terms

(Interpret '`@<`' as "comes before".)

Within these categories, ordering is as follows.

- Variables are put in a standard order. (Roughly, the oldest variable is put first; the order is not related to the names of variables. Users should not rely on the order of variables. They should be considered implementation dependent. The ordering of variables within a sorted list, as produced by `setof/3` or `sort/2`, shall remain constant.)
- Database references are put in a standard order (the order is based roughly on the time of creation of the reference).
- Numbers are put in numeric order. Where a number may be represented by an integer or a floating-point number, as in 2 and 2.0, the integer is considered to be infinitesimally smaller than its floating-point counterpart.

- Atoms are put in alphabetical order according to the character set in use.
- Compound terms are ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order).
- Lists are compared as ordinary compound terms with functor `./2`

For example, here is a list of terms in the standard order:

```
[ X, '$ref'(123456,12), -9, 1, 1.0, fie, foe, fum, [1],
     X = Y, fie(0,2), fie(1,1) ]
```

The predicates for comparison of terms are described below.

T1 `==` *T2* *T1* and *T2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical).

T1 `\==` *T2*
T1 and *T2* are *not* literally identical.

T1 `@<` *T2* *T1* is before term *T2* in the standard order.

T1 `@>` *T2* *T1* is after term *T2*

T1 `@=<` *T2*
T1 is not after term *T2*

T1 `@>=` *T2*
T1 is not before term *T2*

`compare(`*Op, T1, T2*`)`
the result of comparing terms *T1* and *T2* is *Op*, where the possible values for *Op* are:

`=`	if *T1* is identical to *T2*,
`<`	if *T1* is before *T2* in the standard order,
`>`	if *T1* is after *T2* in the standard order.

8.9.7.3 Sorting Terms

Two predicates, `sort/2` and `keysort/2` sort lists into the standard order. `keysort/2` takes a list consisting of key-value pairs and sorts according to the key.

Further sorting routines are available in `library(samsort)`.

8.9.8 Library Support

- `library(occurs)`
- `library(subsumes)`
- `library(samsort)`

Regarding list-processing: Section 12.2 [lib-lis], page 528

8.9.9 Summary of Predicates

- '='/2
- '=..'/2
- ==/2
- \==/2
- @</2
- @=</2
- @>=/2
- @>/2
- append/3
- arg/3
- atom/1
- atom_chars/2
- atomic/1
- compare/3
- copy_term/2
- float/1
- functor/3
- integer/1
- keysort/2
- length/2
- name/2
- nonvar/1
- number/1
- number_chars/2
- numbervars/3
- sort/2
- subsumes_chk/2
- var/1

8.10 Looking at the Program State

8.10.1 Overview

Various aspects of the program state can be inspected: The clauses of all or selected dynamic procedures, currently available atoms, user defined predicates, source files of predicates and clauses, predicate properties and the current load context can all be accessed by calling the predicates listed in Section 8.10.1 [ref-lps-ove], page 245. Furthermore, the values of prolog flags can be inspected and, where it makes sense, changed. The following predicates accomplish these tasks:

`listing` list all dynamic procedures in the type-in module

`listing(`*P*`)`
: list the dynamic procedure(s) specified by *P*

`current_atom(`*A*`)`
: *A* is a currently available atom (nondeterminate)

`current_predicate(`*A*`,`*P*`)`
: *A* is the name of a predicate with most general goal *P* (nondeterminate)

`predicate_property(`*P*`,`*Prop*`)`
: *Prop* is a property of the loaded predicate *P* (nondeterminate)

`prolog_flag(`*F*`,`*V*`)`
: *V* is the current value of Prolog flag *F* (nondeterminate)

`prolog_flag(`*F*`,`*O*`,`*N*`)`
: *O* is the old value of Prolog flag *F*; *N* is the new value

`prolog_load_context(K,`*V*`)`
: find out the context of the current load

`source_file(`*F*`)`
: *F* is a source file that has been loaded into the database

`source_file(`*P*`,`*F*`)`
: *P* is a predicate defined in the loaded file *F*

`source_file(`*P*`,`*F*`,`*N*`)`
: Clause number *N* of predicate *P* came from file *F*

8.10.2 Associating Predicates with their Properties

The following properties are associated with predicates either implicitly or by declaration:

- `built_in`
- `checking_advice`
- `compiled`
- `dynamic`
- `exported`

- `extern_link`
- `foreign`
- `has_advice`
- `imported_from`
- `interpreted`
- `locked`
- `meta_predicate`
- `multifile`
- `spied`
- `volatile`

These are described elsewhere in the manual (see Index). To query these associations, use `predicate_property/2`. The reference page contains several examples.

8.10.3 Associating Predicates with Files

Information about loaded files and the predicates and clauses in them is returned by `source_file/[1,2,3]`. `source_file/1` can be used to identify an absolute filename as loaded, or to backtrack through all loaded files. To find out the correlation between loaded files and predicates, call `source_file/2`. `source_file/3` allows for querying about which clause for a predicate is in which loaded file. `source_file/3` is useful for handling multifile predicates (see Section 18.3.105 [mpg-ref-multifile], page 1184), but it works for predicates defined completely in one file, as well.

Any combination of bound and unbound arguments is possible, and `source_file/3` will generate the others.

8.10.4 Prolog Flags

8.10.4.1 Changing or Querying System Parameters

Prolog flags enable you to modify certain aspects of Prolog's behavior, as outlined below. This is accomplished by using `prolog_flag/3`. If you simply want to query the value of a flag, use `prolog_flag/2`.

By using the prolog flags listed below, it is possible to:

Flag	Purpose
`character_escapes`	Enable or disable escaping of special characters in I/O operations. (See Section 8.1.4 [ref-syn-ces], page 163)

`debugging`
: Turn on/off trace and debug mode by using `prolog_flag/3` or by using the predicates `trace/0`, `debug/0`, `notrace/0`, and `nodebug/0`.

`fileerrors`
: Set or reset the `fileerrors` flag by using `prolog_flag/3` or by using the pair of predicates `fileerrors/0` and `nofileerrors/0`.

`gc`
: Turn on/off garbage collection by using `prolog_flag/3` or by using the predicates `gc/0` and `nogc/0`.

`gc_margin`
: Set the number of bytes that must be reclaimed by a garbage collection in order to avoid heap expansion (not available on some systems; see Section 8.12 [ref-mgc], page 256)

`gc_trace`
: Enable or disable diagnostic tracing of garbage collections.

`multiple`
: on or off.

`single_var`
: on or off.

`syntax_error`
: Control Prolog's response to syntax errors. See Section 8.19.4.10 [ref-ere-err-syn], page 322.

`unknown`
: Set the action to be taken on unknown procedures by using `prolog_flag/3` or `unknown/2` (see Section 6.1.5.4 [dbg-bas-con-unk], page 120). `unknown/2` writes a message to `user_output` saying what the new state is. It is intended for use at the top level. `prolog_flag/3` does not write a message. It is intended for use in code.

For further details, see the reference page. Also see Section 8.12 [ref-mgc], page 256 for more detailed descriptions of the garbage collection flags.

To inspect the value of a flag without changing it, one can say

```
| ?- prolog_flag(FlagName, Value).
```

You can use `prolog_flag/2` to enumerate all the *FlagName*s that the system currently understands, together with their current values.

8.10.4.2 Parameters that can be Queried Only

Prolog flags can be used to effect the changes listed above, or to ask about the current values of those parameters. In addition, you can use `prolog_flag/2` (not `prolog_flag/3`) to make the following queries using the flag names listed below:

Flag	Purpose
`add_ons`	What add-on products are statically linked into to Prolog system?

`host_type`
: What is the host-type?

`quintus_directory`
: What is the absolute name of the Quintus directory, where is the root of the entire Quintus Installation hierarchy?

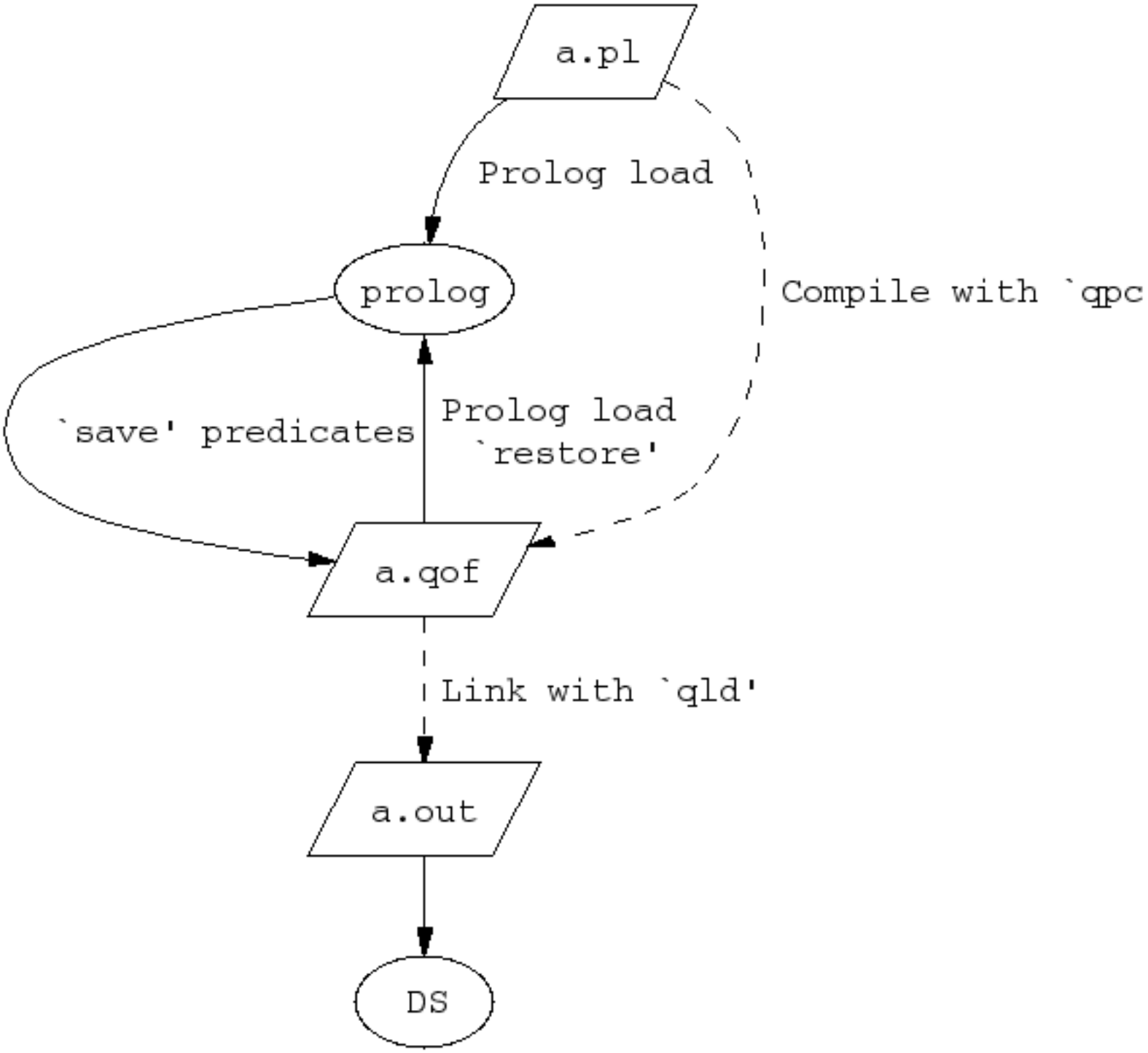
`runtime_directory`
: What is the absolute name of the directory where all Prolog executables reside? In the Runtime System, it is expected that this value will be overwritten, using `qsetpath` when the runtime system is installed (see Section 20.1.3 [too-too-qgetpath], page 1480 and Section 20.1.8 [too-too-qsetpath], page 1495). This flag is used to define `file_search_path(`*`runtime`*`,`*`RuntimeDir`*`)`.

`version`
: What version of Prolog is being run?

`system_type`
: `development`

Use `prolog_flag/2` to make queries, `prolog_flag/3` to make changes.

8.10.5 Load Context



Load Context

By calling `prolog_load_context/[2,3]` you can determine:

- whether the current context is in a loading/compilation or a start-up of an application (see the above figure).
- the current Prolog load/compilation context: module, file, directory or stream.

8.10.5.1 Predicate Summary

- `current_atom/1`
- `current_predicate/2`
- `listing/[0,1]`
- `predicate_property/2`
- `prolog_flag/[2,3]`
- `prolog_load_conitemize/2`
- `source_file/[1,2,3]`

8.11 Interrupting Execution

8.11.1 Control-c Interrupts

At any time, Prolog's execution can be interrupted by typing `^c`. Under Windows, interruption may be delayed if Prolog is in certain OS routines, especially when waiting for input; the interrupt will then happen when returning to the OS routine. The following prompt is then displayed:

```
Prolog interruption (h for help)?
```

If you then type *h*, followed by ⟨RET⟩, you will get a list of the possible responses to this prompt:

```
Prolog interrupt options:
  h  help         - this list
  c  continue     - do nothing
  d  debug        - debugger will start leaping
  t  trace        - debugger will start creeping
  a  abort        - abort to the current break level
  q  really abort - abort to the top level
  e  exit         - exit from Prolog
```

The *d* option will cause you to enter the debugger the next time control passes to a spypoint. You can then use the *g* (ancestors) option of the debugger to find out at what level of execution you interrupted the program.

The `t` option will also cause you to enter the debugger at the next call.

The `a` option causes an abort to the current break level. In previous releases of Quintus Prolog, the `a` option ignored break levels, always aborting to the top level (break level 0).

The `q` option causes an abort to the top level (break level 0). This behavior is identical to the `a` option in previous Quintus Prolog systems.

In runtime systems, the default behavior on a `^c` interrupt is to abort immediately, rather than display the above menu.

For more information on `^c` interrupts and signal handlers, see Section 8.11.2 [ref-iex-iha], page 251.

The predicates that control this are:

`halt/0` exit from Prolog

`break/0` start a new break-level to interpret commands from the user

`abort/0` abort execution of the program; return to current break level

8.11.2 Interrupt Handling

8.11.2.1 Changing Prolog's Control Flow from C

If the application has a toplevel, the function `QP_action()` can be called from C to alter Prolog's flow of control. This function allows the user to make Prolog abort, exit, suspend execution, turn on debugging, or prompt for the desired action. To use it, use `#include <quintus/quintus.h>` in your C source code. This file should be installed in a central place; if not, there should be a copy of it in the '`embed`' directory (refer to Section 1.3 [int-dir], page 11 for location). '`quintus.h`' defines the following constants:

`QP_ABORT` *Abort to the current break level

`QP_REALLY_ABORT`
*Abort to top level

`QP_STOP` Stop (suspend) process

`QP_IGNORE`
Do nothing

`QP_EXIT` Exit Prolog immediately

`QP_MENU` Present action menu

`QP_TRACE` Turn on trace mode

`QP_DEBUG` Turn on debugging

To change Prolog's control flow in a given instance, call `QP_action()` with one of these constants; for example,

```
#include <quintus/quintus.h>
void abort_execution(){
    QP_action(QP_ABORT);
}
```

Some calls to `QP_action()` do not normally return, for example when the `QP_ABORT` constant is specified. However, calls to `QP_action()` from an interrupt handler must be viewed as *requests*. They are requests that will definitely be honored, but not always at the time of the call to `QP_action()`. Therefore calls to `QP_action()` should be prepared for the function to return.

It is currently not possible to call Prolog from an interrupt handler.

For systems that do not have a toplevel, the actions marked with an asterisk will have no effect other than to make `QP_action()` return `QP_ERROR`.

What does it mean to have a toplevel? If the application is calling the function `QP_toplevel()`, then the application has a toplevel. The development system and runtime systems both call `QP_toplevel()`. An exit from either of these environments is effectively a return from `QP_toplevel()`. An embedded application may or may not call `QP_toplevel()`. One of the things `QP_toplevel()` does is establish signal handlers. Another thing it does is establish a place for `QP_action()` to *jump to*, the actions marked with an asterik are essentially a *jump* to the toplevel, and will not work in systems without a toplevel. For more about `QP_toplevel()`, see Section 19.3.66 [cfu-ref-toplevel], page 1446.

8.11.2.2 User-specified signal handlers

This section only applies to UNIX as signals are not used under Windows.

Prolog sets up signal handlers when either `QP_initialize()` or `QP_toplevel()` is called. These handlers provide the default interrupt handling for `^c` described in the previous sections. `QP_initialize()` and `QP_toplevel()` sets handlers for all signals that have the default handler and the default behavior is not what Prolog wants. If users have set their own signal handlers (which are different from the default signal handlers) then Prolog will not change these handlers. Once Prolog has started up and is running the toplevel read-prove loop, Prolog will not change any signal handlers unless the user calls `QP_toplevel()`.

Users can set and remove signal handlers using the system function `signal(2)`.

To set up a signal handler, call the routine

```
signal(signal_name, function_name)
```

from within a C foreign function, where *signal_name* is a constant identifying the signal being trapped and *function_name* is the name of the function to be called in the

event of the signal. The constants identifying the various signals are defined in the file '`/usr/include/signal.h`'.

The example below shows how one would define an interrupt handler using `signal` and `QP_action()`. For most users `^C` is the interrupt character. The files '`interrupt.c`' and '`interrupt.pl`' make up this example; the interrupt handler is set up by calling `establish_handler/0` after compiling '`interrupt.pl`'.

interrupt.pl

```
/*
    This is the foreign interface file for a sample interrupt handler.
*/

foreign_file('interrupt',[establish_handler]).

foreign(establish_handler, establish_handler).

:- load_foreign_files(['interrupt'], []).
```

interrupt.c

```
/*
    The function my_handler is called when the user types the interrupt
    character (normally ^c).  This function prompts for a response
    and executes the user's choice.
*/

#include <signal.h>
#include <quintus/quintus.h>

int my_handler()
{
    char c;

    for(;;) {
        printf("\nWell? ");
        c = getchar();
        if (c != '\n')
            while (getchar() != '\n') {};
        switch(c) {
        case 'a': QP_action(QP_ABORT);
        case 'e': QP_action(QP_EXIT);
        case 'c': return;
        default: printf("a, c or e, please");
        }
    }
}

void establish_handler()
{
    signal(SIGINT,  my_handler);
}
```

The following trace illustrates the use of these files:

```
% cc -c interrupt.c
% prolog
Quintus Prolog Release 3.5 (Sun 4, SunOS 5.5)

| ?- compile(interrupt).
% compiling file /goedel/tim/interrupt.pl
%  foreign file /goedel/tim/interrupt.o loaded
% interrupt.pl compiled in module user, 0.150 sec 1,508 bytes

yes
| ?- establish_handler.

yes
| ?- write(hi).
hi
yes
| ?- ^C
Well? g
a, c or e, please
Well? a

! Execution aborted

| ?- ^C
Well? e
%
```

8.11.2.3 Critical Regions

A critical region is a section of code during whose execution interrupts are to be ignored. To create a critical region, one must block signals for the duration of the critical region and unblock the signals when leaving the critical region. Examples of how to do this in both Prolog and C are in the library files '`critical.pl`' and '`critical.c`', which is discussed further in Section 8.19.6 [ref-ere-ecr], page 324.

8.11.3 Predicate/Function Summary

- `abort/0`
- `break/0`
- `halt/0`
- `QP_action()`

8.11.4 Library Support

`library(critical)`

8.12 Memory Use and Garbage Collection

8.12.1 Overview

Quintus Prolog uses three data areas: program space, local stack space, and global stack space. Each of these areas is automatically expanded if it overflows; if necessary, the other areas are shifted to allow this.

The local stack contains all the control information and variable bindings needed in a Prolog execution. Space on the local stack is reclaimed on determinate success of predicates and by tail recursion optimization, as well as on backtracking.

The global stack space contains the heap (also known as the global stack) and the trail. The heap contains all the data structures constructed in an execution of the program, and the trail contains references to all the variables that need to be reset when backtracking occurs. Both of these areas grow with forward execution and shrink on backtracking. These fluctuations can be monitored by `statistics/[0,2]`.

The program space contains compiled and interpreted code, recorded terms, and atoms. The space occupied by compiled code, interpreted code, and recorded terms is recovered when it is no longer needed; the space occupied by atoms that are no longer in use can be recovered by atom garbage collection described in Section 8.12.8 [ref-mgc-ago], page 266.

Quintus Prolog uses the heap to construct compound terms, including lists. Heap space is used as Prolog execution moves forward. When Prolog backtracks, it automatically reclaims space on the heap. However, if a program uses a large amount of space before failure and backtracking occur, this type of reclamation may be inadequate.

Without garbage collection, the Prolog system must attempt to expand the heap whenever a heap overflow occurs. To do this, it first requests additional space from the operating system. If no more space is available, the Prolog system attempts to allocate unused space from the other Prolog data areas. If additional space cannot be found, a resource error is raised.

Heap expansion and abnormal termination of execution due to lack of heap space can occur even if there are structures in the heap that are no longer accessible to the computation (these structures are what is meant by “garbage”). The proportion of garbage to non-garbage terms varies during execution and with the Prolog code being executed. The heap may contain no garbage at all, or may be nearly all garbage.

The garbage collector periodically reclaims inaccessible heap space, reducing the need for heap expansion and lessening the likelihood of running out of heap. When the garbage collector is enabled (as it is by default), the system makes fewer requests to the operating system for additional space. The fact that less space is required from the operating system can produce a substantial savings in the time taken to run a program, because paging overhead can be much less.

For example, without garbage collection, compiling a file containing the sequence

```
p(_) :- p([a]).
:- p(_).
```

causes the heap to expand until the Prolog process eventually runs out of space. With garbage collection enabled, the above sequence continues indefinitely. The list built on the heap by each recursive call is inaccessible to future calls (since `p/1` ignores its argument) and can be reclaimed by the garbage collector.

Garbage collection does not guarantee freedom from out-of-space errors, however. Compiling a file containing the sequence

```
p(X) :- p([X]).
:- p(a).
```

expands the heap until the Prolog process eventually runs out of space. This happens in spite of the garbage collector, because all the terms built on the heap are accessible to future computation and cannot be reclaimed.

8.12.1.1 Reclaiming Space

`trimcore/0` reclaims space in all of Prolog's data areas. At any given time, each data area contains some free space. For example, the local stack space contains the local stack and some free space for that stack to grow into. The data area is automatically expanded when it runs out of free space, and it remains expanded until `trimcore/0` is called, even though the stack may have shrunk considerably in the meantime. The effect of `trimcore/0` is to reduce the free space in all the data areas as much as possible, and to give the space no longer needed back to the operating system. `trimcore/0` is called each time Prolog returns to the top level or the top of a break level.

8.12.1.2 Displaying Statistics

Statistics relating to memory usage, run time, and garbage collection, including information about which areas of memory have overflowed and how much time has been spent expanding them, can be displayed by calling `statistics/0`.

The output from `statistics/0` looks like this:

```
memory (total)         377000 bytes:     350636 in use,      26364 free
   program space       219572 bytes
      atom space        (2804 atoms)      61024 in use,  43104 free

   global space         65532 bytes:       9088 in use,      56444 free
      global stack                         6984 bytes
      trail                                  16 bytes
      system                               2088 bytes
   local stack          65532 bytes:        356 in use,      65176 free
      local stack                          332 bytes
      system                                24 bytes

 0.000 sec. for 0 global and 0 local space shifts
 0.000 sec. for 0 garbage collections which collected 0 bytes
 0.000 sec. for 0 atom garbage collections which collected 0 bytes
 0.233 sec. runtime
```

Note the use of indentation to indicate sub-areas. That is, memory contains the program space, global space, and local stack, and the global space contains the global stack and trail.

The memory (total) figure shown as "in use" is the sum of the spaces for the program, global and local areas. The "free" figures for the global and local areas are for free space within those areas. However, this free space is considered used as far as the memory (total) area is concerned, because it has been allocated to the global and local areas. The program space is not considered to have its own free space. It always allocates new space from the general memory (total) free area.

Individual statistics can be obtained by `statistics/2`, which accepts a keyword and returns a list of statistics related to that keyword.

The keys and values for `statistics(`*`Keyword`*`, `*`List`*`)` are summarized below. The keywords `core` and `heap` are included to retain compatibility with DEC-10 Prolog. Times are given in milliseconds and sizes are given in bytes.

	Keyword List
runtime	[cpu time used by Prolog, cpu time since last call to `statistics/[0,2]`]
system_time	[cpu time used by the operating system, cpu time used by the system since the last call to `statistics/[0,2]`]
real_time	[wall clock time since 00:00 GMT 1/1/1970, wall clock time since the last call to `statistics/[0,2]`]
memory	[total virtual memory in use, total virtual memory free]
stacks	[total global stack memory, total local stack memory]
program	[program space, 0]

global_stack
: [global stack in use, global stack free]

local_stack
: [local stack in use, local stack free]

trail
: [size of trail, 0]

garbage_collection
: [number of GCs, freed bytes, time spent]

stack_shifts
: [number of global stack area shifts, number of local stack area shifts, time spent shifting]

atoms
: [number of atoms, atom space in use, atom space free]

atom_garbage_collection
: [number of AGCs, freed bytes, time spent]

core
: (same as memory)

heap
: (same as program)

For the keywords `program` and `trail`, the second element of the returned list is always 0. This is for backward compatibility only, 0 being the most appropriate value in the Quintus Prolog system for the quantities that would be returned here in DEC-10 Prolog and previous releases of Quintus Prolog .

To see an example of the use of each of these keywords, type

```
| ?- statistics(K, L).
```

and then repeatedly type ';' to backtrack through all the possible keywords. As an additional example, to report information on the runtime of a predicate p/0, add the following to your program:

```
:- statistics(runtime, [T0| _]),
   p,
   statistics(runtime, [T1|_]),
   T is T1 - T0,
   format('p/0 took ~3d sec.~n', [T]).
```

8.12.2 Garbage Collection and Programming Style

The availability of garbage collection can lead to a more natural programming style. Without garbage collection, a procedure that generates heap garbage may have to be executed in a failure-driven loop. Failure-driven loops minimize heap usage from iteration to iteration of a loop via Quintus Prolog's automatic recovery of heap space on failure. For instance, in the following procedure `echo/0` echoes Prolog terms until it reads an end-of-file character. It uses a failure-driven loop to recover inaccessible heap space.

```
echo :- repeat,
        read(Term),
        echo_term(Term),
        !.

echo_term(Term) :-
        Term == end_of_file.
echo_term(Term) :-
        write(Term), nl,
        fail.
```

Any heap garbage generated by `read/1` or `write/1` is automatically reclaimed by the failure of each iteration.

Although failure-driven loops are an accepted Prolog idiom, they are not particularly easy to read or understand. So we might choose to write a clearer version of `echo/0` using recursion instead, as in

```
echo :- read(Term),
        echo_term(Term).

echo_term(Term) :-
        Term == end_of_file,
        !.
echo_term(Term) :-
        write(Term), nl,
        echo.
```

Without garbage collection the more natural recursive loop accumulates heap garbage that cannot be reclaimed automatically. While it is unlikely that this trivial example will run out of heap space, larger and more practical applications may be unable to use the clearer recursive style without garbage collection. With garbage collection, all inaccessible heap space will be reclaimed by the garbage collector.

Using recursion rather than failure-driven loops can improve programming style further. We might want to write a predicate that reads terms and collects them in a list. This is naturally done in a recursive loop by accumulating results in a list that is passed from iteration to iteration. For instance,

```
collect(List) :-
        read(Term),
        collect_term(Term, List).

collect_term(Term, []) :-
        Term == end_of_file,
        !.
collect_term(Term, [Term|List0]) :-
        collect(List0).
```

For more complex applications this sort of construction might prove unusable without garbage collection. Instead, we may be forced to use a failure-driven loop with side-effects to store partial results, as in the following much less readable version of `collect/1`:

```
collect(List) :-
        repeat,
        read(Term),
        store_term(Term),
        !,
        collect_terms(List).

store_term(Term) :-
        Term == end_of_file.

store_term(Term) :-
        assertz(term(Term)),
        fail.

collect_terms([M|List]) :-
        retract(term(M)),
        !,
        collect_terms(List).
collect_terms([]).
```

The variable bindings made in one iteration of a failure-driven loop are unbound on failure of the iteration. Thus partial results cannot simply be stored in a data structure that is passed along to the next iteration. We must instead resort to storing partial results via side-effects (here, `assertz/1`) and collect (and clean up) partial results in a separate pass. The second example is much less clear to most people than the first. It is also much less efficient than the first. However, if there were no garbage collector, larger examples of the second type might be able to run where those of the first type would run out of memory.

8.12.3 Enabling and Disabling the Garbage Collector

The user has the option of executing programs with or without garbage collection. Procedures that do not use a large amount of heap space before backtracking may not be affected when garbage collection is enabled. Procedures that do use a large amount of heap space may execute more slowly due to the time spent garbage collecting, but will be more likely to run to completion. On the other hand, such programs may run faster when the garbage collector is enabled because the virtual memory is not expanded to the extent that "thrashing" occurs. `gc/0` and `nogc/0` are the built-in predicates that are used to enable and disable the garbage collector. Alternatively, the `gc` Prolog flag can be set to `on` or `off`. To run the `gc` in a verbose mode, set the `gc_trace` flag to `on`. By default, garbage collection is enabled.

8.12.4 Monitoring Garbage Collections

By default, the user is given no indication that the garbage collector is operating. If no program ever runs out of space and no program using a lot of heap space requires an inordinate amount of processing time, then such information is unlikely to be needed.

However, if a program thought to be using much heap space runs out of space or runs inordinately slowly, the user may want to determine whether more or less frequent garbage collections are necessary. Information obtained from the garbage collector by turning on the `gc_trace` option of `prolog_flag/3` can be helpful in this determination.

8.12.5 Interaction of Garbage Collection and Heap Expansion

For most programs, the default settings for the garbage collection parameters should suffice. For programs that have high heap requirements, the default parameters may result in a higher ratio of garbage collection time to run time. These programs should be given more space in which to run.

The `gc_margin` is a non-negative integer specifying the desired margin in kilobytes. For example, the default value of 1000 means that the heap will not be expanded if garbage collection can reclaim at least one megabyte. The advantage of this criterion is that it takes into account both the user's estimate of the heap usage and the effectiveness of garbage collecting.

1. Setting the `gc_margin` higher than the default will cause fewer heap expansions and garbage collections. However, it will use more space, and garbage collections will be more time-consuming when they do occur.

   Setting the margin too large will cause the heap to expand so that if it does overflow, the resulting garbage collection will significantly disrupt normal processing. This will be especially so if much of the heap is accessible to future computation.
2. Setting the `gc_margin` lower than the default will use less space, and garbage collections will be less time-consuming. However, it will cause more heap expansions and garbage collections.

   Setting the margin too small will cause many garbage collections in a small amount of time, so that the ratio of garbage-collecting time to computation time will be abnormally high.
3. Setting the margin correctly will cause the heap to expand to a size where expansions and garbage collections are infrequent and garbage collections are not too time-consuming, if they occur at all.

The correct value for the `gc_margin` is dependent upon many factors. Here is a non-prioritized list of some of them:

- The amount of memory available to the Prolog process

- The maximum memory limit imposed on the Prolog process (see Section 8.12.7 [ref-mgc-osi], page 264, Section 8.12.7 [ref-mgc-osi], page 264)
- The program's rate of heap garbage generation
- The program's rate of heap non-garbage generation
- The program's backtracking behavior
- The amount of time needed to collect the generated garbage
- The growth rate of the other Prolog stacks

The algorithm used when the heap overflows is as follows:

```
if gc is on
and the heap is larger than gc_margin kilobytes then
   garbage collect the heap
   if less than gc_margin bytes are reclaimed then
      try to expand the heap
   endif
else
   try to expand the heap
endif
```

The user can use the `gc_margin` option of `prolog_flag/3` to reset the `gc_margin` (see Section 8.10.1 [ref-lps-ove], page 245). If a garbage collection reclaims at least the `gc_margin` kilobytes of heap space the heap is not expanded after garbage collection completes. Otherwise, the heap is expanded after garbage collection. This expansion provides space for the future heap usage that will presumably occur. In addition, no garbage collection occurs if the heap is smaller than `gc_margin` kilobytes.

> **Please note:** `prolog_flag(gc_margin, Old, New)` has nothing to do with the `gcguide(margin, Old, New)` of older Prolog systems. The "margin" of those other systems was used for entirely different purposes.

8.12.6 Invoking the Garbage Collector Directly

Normally, the garbage collector is invoked only when some Prolog data area overflows, so the time of its invocation is not predictable. In some applications it may be desirable to invoke the garbage collector at regular intervals (when there is known to be a significant amount of garbage on the heap) so that the time spent garbage collecting is more evenly distributed in the processing time. For instance, it may prove desirable to invoke the garbage collector after each iteration of a question-and-answer loop that is not failure-driven.

In rare cases the default garbage collection parameters result in excessive garbage collecting costs or heap expansion, and the user cannot tune the `gc_margin` parameter adequately. Explicitly invoking the garbage collector using the built-in predicate `garbage_collect/0` can be useful in these circumstances.

8.12.7 Operating System Interaction

This section describes the various system parameters required to run Prolog.

There is normally no need for you to seek any special privileges or quotas in order to run Prolog. Prolog will automatically expand its space up to the total amount of virtual space you are allowed. If it should run out of space, Prolog will raise a resource error.

This may happen because of an infinite recursion in your program, or it may be that your program really needs more space than is available. Under UNIX, if you are using the C shell (`csh`), you can find out how much space is available by means of the `csh` command `limit`. The command

```
% limit
```

will list a number of limits of which the relevant one is `datasize`. This number is the number of kilobytes available to Prolog for its data areas. You can reduce this limit by typing, for example,

```
% limit datasize 2000
```

The main reason that you might want to reduce the limit is that some systems allow the allocation of more virtual memory than there is swap space available, and then to crash. You can run quite large programs with a datasize of 2000 kilobytes.

UNIX Caveat:

On some UNIX systems, the specified `datasize` (program) limit (see `limit(csh)` and `getrlimit(3)`) can be grossly higher then the maximum break that a process can set. This is because the setting of the break is dependent upon the amount of swap space available. Since all processes share the same swap space, the space available to any one process is based on the space usage of all other processes running on the machine. Therefore, one process that has set a large program break may prevent another process from doing the same, if both are running simultaneously.

The Quintus Prolog memory manager makes calculations based upon the specified `datasize` limit, since the actual limit cannot be determined except by experimentation, and even then the limit changes over time. Better memory management will result when the specified `datasize` limit is close to the actual limit.

The default behavior of Prolog is tuned to be optimal for a large class of programs. If the programmers need greater control of the way Prolog grows and frees memory, they can set environment variables, the documentation for which follow. Note that the default values for these variables should satisfy almost all programs and you really do not need to set these variables at all. The values for these variables are entered in bytes, but may be followed by '`K`' or '`M`' meaning kilobytes or megabytes respectively.

`PROLOGINITSIZE`

Controls the size of Prolog's initial memory allocation. Can be set to a sufficiently large size to allow the Prolog application to execute without needing to expand. This must be done before Prolog is invoked.

By default, the value is the minimum memory required for Prolog to start up. In addition, the value is constrained to be at least that amount, regardless of the user setting.

`PROLOGMAXSIZE`

Can be used to place a limit on the amount of data space that a given Prolog process will use.

The `csh` command `limit` can also be used to set the amount of data space that can be used by the the current shell and all processes within it.

By default, the value is effectively infinity, which is to say that Prolog's expansion will only be limited by the space that the shell is able to provide it.

`PROLOGINCSIZE`

Can be used to control the amount of space Prolog asks the operating system for in any given memory expansion.

By default, the value is the minimum amount of memory that will allow Prolog to expand one of its data areas, by kilobytes. In addition, the value is constrained to be at least that amount, regardless of the user setting.

`PROLOGKEEPSIZE`

Can be used to control the amount of space Prolog retains after performing some computation. By default, Prolog gets memory from the operating system as the user program executes and returns all the free memory back to the operating system when the user program does not need any more. If the programmer knows that her program once it has grown to a certain size is likely to need as much memory for future computations, then she can advise Prolog not to return all the free memory back to the operating system by setting the value to K. Once Prolog grows to K bytes, it will always keep at least K bytes around. Only memory that was allocated above and beyond K bytes is returned to the OS.

`PROLOGLOCALMIN`

Can be used to control the amount of space Prolog reserves for the local stack. The purpose of the local stack is described in detail in Section 8.12 [ref-mgc], page 256. The default value is 64Kb.

`PROLOGGLOBALMIN`

Can be used to control the amount of space Prolog reserves for the global stack. The purpose of the global stack is described in detail in Section 8.12 [ref-mgc], page 256. The default value is 64Kb.

8.12.8 Atom Garbage Collection

By default, atoms created during the execution of a program remain permanently in the system until Prolog exits. For the majority of applications this behavior is not a problem and can be ignored. However, for two classes of application this can present problems. Firstly the internal architecture of Quintus Prolog limits the number of atoms that be can created to 2,031,616 and this can be a problem for database applications that read large numbers of atoms from a database. Secondly, the space occupied by atoms can become significant and dominant memory usage, which can be a problem for processes designed to run perpetually.

These problems can be overcome by using atom garbage collection to reclaim atoms that are no longer accessible to the executing program.

Atoms can be created in many ways: when an appropriate token is read with `read_term/3`, when source or QOF files are loaded, when `atom_chars/2` is called with a character list, or when `QP_atom_from_string()` is called in C code. In any of these contexts an atom is only created if it does not already exist; all atoms for a given string are given the same identification number, which is different from the atom of any other string. Thus, atom recognition and comparison can be done quickly, without having to look at strings. An occurrence of an atom is always of a fixed, small size, so where a given atom is likely to be used in several places simultaneously the use of atoms can also be more compact than the use of strings.

A Prolog functor is implemented like an atom, but also has an associated arity. For the purposes of atom garbage collection, a functor is considered to be an occurrence of the atom of that same name.

Atom garbage collection is similar to heap garbage collection except that it is not invoked automatically, but rather through a call to the built-in predicate `garbage_collect_atoms/0`. The atom garbage collector scans Prolog's data areas looking for atoms that are currently in use and then throws away all unused atoms, reclaiming their space.

Atom garbage collection can turn an application that continually grows and eventually either runs into the atom number limit or runs out of space into one that can run perpetually. It can also make feasible applications that load and manipulate huge quantities of atom-rich data that would otherwise become full of useless atoms.

8.12.8.1 The Atom Garbage Collector User Interface

Because the creation of atoms does not follow any other system behaviors like memory growth or heap garbage collection, Quintus has chosen to keep the invocation of atom garbage collection independent of any other operation and to keep the invocation of atom garbage collection explicit rather than making it automatic. It is often preferable for the programmer to control when it will occur in case preparations need to be made for it.

Atom garbage collection is invoked by calling the new built-in predicate `garbage_collect_atoms/0`. The predicate normally succeeds silently. The user may determine whether to invoke atom garbage collection at a given point based on information returned from a call to `statistics/2` with the keyword `atoms`. That call returns a list of the form

```
[number of atoms, atom space in use, atom space free]
```

For example,

```
| ?- statistics(atoms, Stats).

Stats = [4313,121062,31032]
```

One would typically choose to call `garbage_collect_atoms/0` prior to each iteration of an iterative application, when either the number of atoms or the atom space in use passes some threshold, e.g.

```
<driver loop> :-
        ...
        repeat,
           maybe_atom_gc,
           <do next iteration>
           ...
        fail.
<driver loop>.
```

where

```
maybe_atom_gc :-
        statistics(atoms, [_,Inuse,_]),
        atom_gc_space_threshold(Space),
        ( Inuse > Space -> garbage_collect_atoms ; true ).

% Atom GC if there are more than 100000 bytes of atoms:
atom_gc_space_threshold(100000).
```

More sophisticated approaches might use both atom number and atom space thresholds, or could adjust a threshold if atom garbage collection didn't free an adequate number of atoms.

To be most effective, atom garbage collection should be called when as few as possible atoms are actually in use. In the above example, for instance, it makes the most sense to do atom garbage collection at the beginning of each iteration rather than at the end, as at the beginning of the iteration the previous failure may just have freed large amounts of atom-rich global and local stack. Similarly, it's better to invoke atom garbage collection after abolishing or retracting a large database than to do so before.

8.12.8.2 Protecting Atoms in Foreign Memory

Quintus Prolog's foreign language interface allows atoms to be passed to foreign functions. When calling foreign functions from Prolog, atoms are passed via the `+atom` argument type in the predicate specifications of `foreign/[2,3]` facts. The strings of atoms can be passed to foreign functions via the `+string` argument type. In the latter case a pointer to the Prolog symbol table's copy of the string for an atom is what is passed. When calling Prolog from C, atoms are passed back from C to Prolog using the `-atom` and `-string` argument types in `extern/1` declarations. Atoms can also be created in foreign code via functions like `QP_atom_from_string()`.

Prolog does not keep track of atoms (or strings of atoms) stored in foreign memory. As such, it cannot guarantee that those atoms will be retained by atom garbage collection. Therefore Quintus Prolog provides functions to *register* atoms (or their strings) with the atom garbage collector. Registered atoms will not be reclaimed by the atom garbage collector. Atoms can be registered while it is undesirable for them to be reclaimed, and then unregistered when they are no longer needed.

Of course, the majority of atoms passed as atoms or strings to foreign functions do not need to be registered. Only those that will be stored across foreign function calls (in global variables) or across nested calls to Prolog are at risk. An extra margin of control is given by the fact the programmer always invokes atom garbage collection explicitly, and can ensure that this is only done in contexts that are "safe" for the individual application.

To register or unregister an atom, one of the following functions is used:

```
int QP_register_atom(atom)
QP_atom atom;

int QP_unregister_atom(atom)
QP_atom atom;
```

These functions return either `QP_ERROR` or a non-negative integer. The return values are discussed further in Section 8.12.8.4 [ref-mgc-ago-are], page 270.

As noted above, when an atom is passed as a string (`+string`) to a foreign function, the string the foreign function receives is the one in Prolog's symbol table. When atom garbage collection reclaims the atom for that string, the space for the string will also be reclaimed.

Thus, if the string is to be stored across foreign calls then either a copy of the string or else the atom (`+atom`) should be passed into the foreign function so that it can be registered and `QP_string_from_atom()` can be used to access the string from the atom.

Keep in mind that the registration of atoms only pertains to those passed to foreign functions or created in foreign code. Atoms in Prolog's data areas are maintained automatically. Note also that even though an atom may be unregistered in foreign code, atom garbage collection still may not reclaim it as it may be referenced from Prolog's data areas. But if an atom

is registered in foreign code, it will be preserved regardless of its presence in Prolog's data areas.

The following example illustrates the use of these functions. In this example the current value of an object (which an atom) is being stored in a C global variable. There are two C functions that can be called from Prolog, one to update the current value and one to access the value.

```
#include <quintus/quintus.h>

QP_atom current_object = NULL;

update_object(newvalue)
QP_atom newvalue;
{
        /* if current_object contains an atom, unregister it */
        if (current_object)
                (void) QP_unregister_atom(current_object);

        /* register new value */
        (void) QP_register_atom(newvalue);
        current_object = newvalue;
}

QP_atom get_object()
{
        return current_object;
}
```

8.12.8.3 Permanent Atoms

Atom garbage collection scans all Prolog's dynamic data areas when looking for atoms that are in use. Scanning finds atoms in the Prolog stacks and in all compiled and interpreted code that has been dynamically loaded into Prolog via `consult/1`, `use_module/1`, `assert/2`, etc. However, there are certain potential sources of atoms in the Prolog image from which atoms cannot be reclaimed. Atoms for Prolog code that has been statically linked with either the Prolog Development Environment or the Runtime Environment have been placed in the text space, making them (and the code that contains them) effectively permanent. Although such code can be abolished, its space can never be reclaimed.

These atoms are internally flagged as permanent by the system and are always retained by atom garbage collection. An atom that has become permanent cannot be made non-permanent, so can never be reclaimed.

8.12.8.4 Details of Atom Registration

The functions that register and unregister atoms are in fact using reference counting to keep track of atoms that have been registered. As a result, it is safe to combine your code with libraries and code others have written. If the other code has been careful to register and unregister its atoms as appropriate, atoms will not be reclaimed until everyone has unregistered them.

Of course, it is possible when writing code that needs to register atoms that errors could occur. Atoms that are registered too many times simply will not be garbage collected until they are fully unregistered. However, atoms that aren't registered when they should be may be reclaimed on atom garbage collection. One normally doesn't need to think about the reference counting going on in `QP_register_atom()` and `QP_unregister_atom()`, but some understanding of its details could prove helpful when debugging.

To help you diagnose problems with registering and unregistering atoms, `QP_register_atom()` and `QP_unregister_atom()` both normally return the current reference count for the atom. If an error occurs, e.g. a nonexistent atom is registered or unregistered, `QP_ERROR` is returned.

An unregistered atom has a reference count of 0. Unregistering an atom that is unregistered is a no-op; in this case, `QP_unregister_atom()` returns 0. A permanent atom has a reference count of 128. In addition, if an atom is simultaneously registered 128 times, it becomes permanent. (An atom with 128 distinct references is an unlikely candidate for reclamation!) Registering or unregistering an atom that is permanent is also a no-op; `QP_register_atom()` and `QP_unregister_atom()` return 128.

Various safeguards enable you to detect when an atom may have been reclaimed prematurely. An atom that has been reclaimed and has not yet been reused appears as the special system atom `'$anon'`, which cannot match any user atom (even a user-supplied `'$anon'`, which will be a distinct atom). However, once an atom's space is reused, any references to the old atom will now see only the new atom. It is not possible to detect that an atom has been reused once the reuse occurs.

8.12.9 Summary of Predicates

- `garbage_collect/0`
- `gc/0`
- `nogc/0`
- `prolog_flag/3`
- `statistics/[0,2]`
- `trimcore/0`
- `garbage_collect_atoms/0`

8.13 Modules

8.13.1 Overview

The module system lets the user divide large Prolog programs into *modules*, or rather smaller sub-programs, and define the interfaces between those modules. Each module has its own name space; that is, a predicate defined in one module is distinct from any predicates with the same name and arity that may be defined in other modules. The module system encourages a group of programmers to define the dependence each has on others' work before any code is written, and subsequently allows all to work on their own parts independently. It also helps to make library predicates behave as extensions of the existing set of built-in predicates.

The Quintus Prolog library uses the module system and can therefore serve as an extended example of the concepts presented in the following text. The design of the module system is such that loading library files and calling library predicates can be performed without knowledge of the module system.

Some points to note about the module system are that:

- It is based on predicate modularity rather than on data modularity; that is, atoms and functors are global.
- It is flat rather than hierarchical; any module may refer to any other module by its name — there is no need to specify a path of modules.
- It is not strict; modularity rules can be explicitly overridden. This is primarily for flexibility during debugging.
- It is efficient; calls to predicates across module boundaries incur little or no overhead.
- It is compatible with previous releases of Quintus Prolog; existing Prolog code should run unchanged.

8.13.2 Basic Concepts

Each predicate in a program is identified by its *module*, as well as by its name and arity.

A module defines a set of predicates, some of which have the property of being *public*. Public predicates are predicates that can be *imported* by other modules, which means that they can then be called from within those modules. Predicates that are not public are *private* to the module in which they are defined; that is, they cannot be called from outside that module (except by explicitly overriding the modularity rules as described in Section 8.13.6 [ref-mod-vis], page 274).

There are two kinds of importation:

1. A module *M1* may import a specified set of predicates from another module *M2*. All

the specified predicates should be public in *M2*.

2. A module *M1* may import all the public predicates of another module *M2*.

Built-in predicates do not need to be imported; they are automatically available from within any module.

There is a special module called `user`, which is used by default when predicates are being defined and no other module has been specified.

If you are using a program written by someone else, you need not be concerned as to whether or not that program has been made into a module. The act of loading a module from a file using `compile/1`, or `ensure_loaded/1` (see Section 8.4 [ref-lod], page 189) will automatically import all the public predicates in that module. Thus the command

```
:- ensure_loaded(library(basics)).
```

will load the basic list-processing predicates from the library and make them available.

8.13.3 Defining a Module

The normal way to define a module is by creating a *module-file* for it and loading it into the Prolog system. A module-file is a Prolog file that begins with a *module declaration*.

A module declaration has the form

```
:- module(+ModuleName, +PublicPredList).
```

Such a declaration must appear as the first term in a file, and declares that file to be a module-file. The predicates in the file will become part of the module *ModuleName*, and the predicates specified in *PublicPredList* are those that can be imported by other modules; that is, the public predicates of this module.

Instead of creating and loading a module-file, it is also possible to define a module dynamically by, for example, asserting clauses into a specified module. A module created in this way has no public predicates; all its predicates are private. This means that they cannot be called from outside that module except by explicitly overriding the modularity rules as described in Section 8.13.6 [ref-mod-vis], page 274. Dynamic creation of modules is described in more detail in Section 8.13.9 [ref-mod-dmo], page 276.

8.13.4 Converting Non-module-files into Module-files

The Prolog cross-referencer located in `qplib(tools)` can automatically generate `module/2` declarations from its cross-reference information. This is useful if you want to take a set of files making up a program and make each of those files into a module-file. See the file `library('xref.doc')` for more information.

Alternatively, if you have a complete Prolog program consisting of a set of source files `{file1, file2, ...}`, and you wish to encapsulate it in a single module *mod*, then this can be done by creating a "driver" file of the following form:

```
:- module(mod, [ ... ]).

:- ensure_loaded(file1).
:- ensure_loaded(file2).
   .
   .
   .
```

When a module is created in this way, none of the files in the program `{file1, file2, ...}` have to be changed.

8.13.5 Loading a Module

To gain access to the public predicates of a module-file, load it as you would any other file—using `compile/1`, or `ensure_loaded/1` as appropriate. For example, if your code contains a directive such as

```
:- ensure_loaded(File).
```

this directive will load the appropriate file *File* whether or not *File* is a module-file. The only difference is that if *File* is a module-file any private predicates that it defines will not be visible to your program.

The load predicates are adequate for use at Prolog's top level, or when the file being loaded is a utility such as a library file. When you are writing modules of your own; `use_module/[1,2,3]` is the most useful.

The following predicates are used to load modules:

`use_module(F)`
: import the module-file(s) F, loading them if necessary; same as `ensure_loaded(F)` if all files in F are module-files

`use_module(F,I)`
: import the procedure(s) I from the module-file F, loading module-file F if necessary

`use_module(M,F,I)`
: import I from module M, loading module-file F if necessary

Before a module-file is loaded, the associated module is *reinitialized*: any predicates previously imported into that module are forgotten by the module.

If a module of the same name with a different *PublicPredList* or different meta-predicate list has previously been loaded from a different module-file, a warning is printed and you are given the option of abandoning the load. Only one of these two modules can exist in the system at one time.

Normally, a module-file can be reloaded after editing with no need to reload any other modules. However, when a module-file is reloaded after its *PublicPredList* or its meta-predicate declaration (see Section 8.13.17 [ref-mod-met], page 284) has been changed, any modules that import predicates from it may have become inconsistent. This is because a module is associated with a predicate at compile time, rather than run time. Thus, other modules may refer to predicates in a module-file that are no longer public or whose module name expansion requirements have changed. In the case of module-importation (where all, rather than specific, public predicates of a module are imported), it is possible that some predicates in the importing module should now refer to a newly-public predicate but do not. Whenever the possibility of such inconsistency arises, you will be warned at the end of the load that certain modules need to be reloaded. This warning will be repeated at the end of each subsequent load until those modules have been reloaded.

Modules may be saved to a QOF file by calling `save_modules(`*`Modules`*`,`*`File`*`)` (see Section 8.5 [ref-sls], page 192).

8.13.6 Visibility Rules

By default, predicates defined in one module cannot be called from another module. This section enumerates the exceptions to this—the ways in which a predicate can be *visible* to modules other than the one in which it is defined.

1. The built-in predicates can be called from any module.
2. Any predicate that is named in the *PublicPredList* of a module, and that is imported by some other module *M*, can be called from within *M*.
3. Module Prefixing: Any predicate, whether public or not, can be called from any other module if its module is explicitly given as a prefix to the goal, attached with the `:/2` operator. The module prefix overrides the default module. For example,

   ```
   :- mod:foo(X,Y).
   ```

   always calls `foo/2` in module *mod*. This is effectively a loophole in the module system, which allows you to override the normal module visibility rules. It is intended primarily to facilitate program development and debugging, and it should not be used extensively since it subverts the original purposes of using the module system.

   Note that a predicate called in this way does not necessarily have to be defined in the specified module. It may be imported into it. It can even be a built-in predicate, and this is sometimes useful — see Section 8.13.7 [ref-mod-som], page 275, for an example.

8.13.7 The Source Module

For any given procedure call, or goal, the *source module* is the module in which the corresponding predicate must be visible. That is, unless the predicate is built-in it must be defined in, or imported into, the source module.

For goals typed at the top level, the source module is the *type-in module*, which is `user` by default — see Section 8.13.8 [ref-mod-tyi], page 276. For goals appearing in a file (either as goal clauses or as normal clauses), the source module is the one into which that file has been loaded.

There are a number of built-in predicates that take predicate specifications, clauses, or goals as arguments. Each of these types of argument must be understood with reference to some module. For example, `assert/1` takes a clause as its argument, and it must decide into which module that clause should be asserted. The default assumption is that it asserts the clause into the source module. Another example is `call/1`. The goal (A) calls the predicate `foo/1` in the source module; this ensures that in the compound goal (B) both occurrences of `foo/1` refer to the same predicate.

```
call(foo(X))                                                    (A)

call(foo(X)), foo(Y)                                            (B)
```

All predicates that refer to the source module allow you to override it by explicitly naming some other module to be used instead. This is done by prefixing the relevant argument of the predicate with the module to be used followed by a ':' operator. For example (C), asserts `f(x)` in module `m`.

```
| ?- assert(m:f(x)).                                            (C)
```

Note that if you call a goal in a specified module, overriding the normal visibility rules (see Section 8.13.6 [ref-mod-vis], page 274), then the source module for that goal is the one you specify, not the module in which this call occurs. For example (D), has exactly the same effect as (C)—`f(x)` is asserted in module `m`. In other words, prefixing a goal with a module duplicates the effect of calling that goal from that module.

```
| ?- m:assert(f(x)).                                            (D)
```

Another built-in predicate that refers to the source module is `compile/1`. In this case, the argument is a file, or list of files, rather than a predicate specification, clause, or goal. However, in the case where a file is not a module-file, `compile/1` must decide into which module to compile its clauses, and it chooses the source module by default. This means that you can compile a file *File* into a specific module *M* using

```
| ?- compile(M:File).
```

Thus if *File* is a module-file, this command would cause its public predicates to be imported into module *M*. If *File* is a non-module-file, it is loaded into module *M*.

For a list of the built-in predicates that depend on the source module, see Section 8.13.16 [ref-mod-mne], page 282. In some cases, user-defined predicates may also require the concept of a source module. This is discussed in Section 8.13.17 [ref-mod-met], page 284.

8.13.8 The Type-in Module

The *type-in* module is the module that is taken as the source module for goals typed in by the user. The name of the default type-in module is `user`. That is, the predicates that are available to be called directly by the user are those that are visible in the module `user`.

When debugging, it is often useful to call, directly from the top level, predicates that are private to a module, or predicates that are public but that are not imported into `user`. This can be done by prefixing each goal with the module name, as described in Section 8.13.6 [ref-mod-vis], page 274; but rather than doing this extensively, it may be more convenient to make this module the type-in module.

The type-in module can be changed using the built-in predicate `module/1` (see Section 18.3.103 [mpg-ref-module1], page 1182); for example,

```
| ?- module(mod).
```

This command will cause subsequent goals typed at the top level to be executed with `mod` as their source module.

The name of the type-in module is always displayed, except when it is `user`. If you are running Prolog under the editor interface, the type-in module is displayed in the status line of the Prolog window. If you are running Prolog without the editor interface, the type-in module is displayed before each top-level prompt.

For example, if you are running Prolog without the editor:

```
| ?- module(foo).

yes
[foo]
| ?-
```

It should be noted that it is unlikely to be useful to change the type-in module via a directive embedded in a file to be loaded, because this will have no effect on the load — it will only change the type-in module for commands subsequently entered by the user.

8.13.9 Creating a Module Dynamically

There are several ways in which you can create a module without loading a module-file for it. One way to do this is by asserting clauses into a specified module. For example,

the command (A) will create the dynamic predicate `f/1` and the module `m` if they did not previously exist.

```
| ?- assert(m:f(x)).                                    (A)
```

Another way to create a module dynamically is to compile a non-module-file into a specified module. For example (B), will compile the clauses in *File* into the module *M*.

```
| ?- compile(M:File).                                   (B)
```

The same effect can be achieved by (temporarily) changing the type-in module to *M* (see Section 8.13.8 [ref-mod-tyi], page 276) and then calling `compile(File)`, or executing the command in module *M* as in (C).

```
| ?- M:compile(File).                                   (C)
```

8.13.10 Module Prefixes on Clauses

Every clause in a Prolog file has a source module implicitly associated with it. If the file is a module-file, then the module named in the module declaration at the top of the file is the source module for all the clauses. If the file is not a module-file, the relevant module is the source module for the command that caused this file to be loaded.

The source module of a predicate decides in which module it is defined (the module of the head), and in which module the goals in the body are going to be called (the module of the body). It is possible to override the implicit source module, both for head and body, of clauses and directives, by using prefixes. For example, consider the module-file:

```
:- module(a, []).

:- dynamic m:a/1.
b(1).
m:c([]).
m:d([H|T]) :- q(H), r(T).
m:(e(X) :- s(X), t(X)).
f(X) :- m:(u(X), v(X)).
```

In the previous example, the following modules apply:

1. `a/1` is declared dynamic in the module `m`.
2. `b/1` is defined in module `a` (the module of the file).
3. `c/1` is defined in module `m`.
4. `d/1` is defined in module `m`, but `q/1` and `r/1` are called in module `a` (and must therefore be defined in module `a`).
5. `e/1` is defined in module `m`, and `s/1` and `t/1` are called in module `m`.
6. `f/1` is defined in module `a`, but `u/1` and `v/1` are called in module `m`.

Module prefixing is especially useful when the module prefix is `user`. There are several predicates that have to be defined in module `user` but that you may want to define (or extend) in a program that is otherwise entirely defined in some other module or modules:

- `runtime_entry/1`
- `term_expansion/2`
- `portray/1`
- `file_search_path/2`
- `library_directory/1`

Note that if clauses for one of these predicates are to be spread across multiple files, it will be necessary to declare that predicate to be multifile by putting a multifile declaration in each of the files.

8.13.10.1 Current Modules

A loaded module becomes current as soon as it is encountered, and a module can never lose the property of being current.

8.13.11 Debugging Code in a Module

Having loaded a module to be debugged, you can trace through its execution in the normal way. When the debugger stops at a port, the procedure being debugged is displayed with its module name as a prefix unless the module is `user`.

The predicate `spy/1` depends on the source module. It can be useful to override this during debugging. For example,

```
| ?- spy mod1:f/3.
```

puts a spypoint on `f/3` in module `mod1`.

It can also be useful to call directly a predicate that is private to its module in order to test that it is doing the right thing. This can be done by prefixing the goal with its module; for example,

```
| ?- mod1:f(a,b,X).
```

8.13.12 Modules and Loading through the Editor Interface

When you (re)load some Prolog code through the editor interface, the module into which the code is to be loaded is selected as follows.

- if the code begins with a module declaration, this is exactly the same as loading that text from a file using the Load Predicates;
- otherwise, if the file containing the code has previously been associated with some module other than `user`, the code is reloaded into that module;
- otherwise, if the type-in module is `user`, the code is loaded into `user`;
- otherwise, you are prompted to confirm that you wish to load the code into the type-in module — if not, the load is abandoned.

Note that when a fragment of code has been loaded into a particular module other than `user`, the editor will subsequently insist that that code belongs to that module. In order to change this, the entire module must be reloaded.

When a module declaration is processed, the module is reinitialized; all predicates previously imported into that module are forgotten. Therefore, when only *part* of a module-file is reloaded through the editor interface, that part should generally *not* include the module declaration.

Loading an entire module through the editor interface is like loading the module via the Load Predicates in that all the public predicates in the module are imported into the type-in module. The only difference is that in the case in which you load the module through the editor interface you will be prompted for confirmation before the importation takes place. This is because there are situations in which you might want to reload a module via the editor interface without importing it into the type-in module; that is, situations in which you would not want to allow the importation to happen. For example, suppose that the type-in module is the default `user`, and that you have been modifying a module `m1` from which another module `m2` imports predicates, but from which `user` does not import anything. In this case, you may want to reload `m1`, using the editor interface, without importing it into `user`.

When a file that is not a module-file is loaded into several different modules, reloading all or part of it through the editor interface affects only the module into which it was most recently loaded.

8.13.13 Name Clashes

A *name clash* can arise if:

1. a module tries to import a predicate from some other module `m1` and it has already imported a predicate with the same name and arity from a module `m2`;
2. a module tries to import a predicate from some other module `m1` and it already contains a definition of a predicate with the same name and arity; or
3. a module tries to define a predicate with the same name and arity as one that it has imported.

Whenever a name clash arises, a message is displayed beginning with the words '`NAME CLASH`'. If the module that is importing or defining the clashing predicate is not `user`, then

this message is just a warning, and the attempt to import or define the predicate simply fails. Otherwise, if the module *is* `user`, the user is asked to choose from one of several options; for example,

```
NAME CLASH: f/3 is already imported into module user
            from module m1;
            do you want to override this definition with
            the one in m2? (y,n,p,s,a or ?)
```

The meanings of the four recognized replies are as follows:

- *y* means forget the previous definition of `f/3` from `m1` and use the new definition of `f/3` from `m2` instead.
- *n* means retain the previous definition of `f/3` from `m1` and ignore the new definition of `f/3` from `m2`.
- *p* (for proceed) means forget the previous definition of `f/3` and of all subsequent predicate definitions in `m1` that clash during the current load of `m2`. Instead, use the new definitions in `m2`. When the *p* option is chosen, predicates being loaded from `m1` into `m2` will cause no '`NAME CLASH`' messages for the remainder of the load, though clashes with predicates from other modules will still generate such messages.
- *s* (for suppress) means forget the new definition of `f/3` and of all subsequent predicate definitions in `m1` that clash during the current load of `m2`. Instead, use the old definitions in `m2`. When the *s* option is chosen, predicates being loaded from `m1` into `m2` will cause no '`NAME CLASH`' messages for the remainder of the load, though clashes with predicates from other modules will still generate such messages.
- *?* gives brief help information.

8.13.14 Obtaining Information about Loaded Modules

`current_module(`*M*`)`
: *M* is the name of a current module

`current_module(`*M*`,`*F*`)`
: *F* is the name of the file in which *M's* module declaration appears

8.13.14.1 Predicates Defined in a Module

The built-in predicate `current_predicate/2` can be used to find the predicates that are defined in a particular module.

To backtrack through all of the predicates defined in module `m`, use

```
| ?- current_predicate(_, m:Goal).
```

To backtrack through *all* predicates defined in *any* module, use

```
| ?- current_predicate(_, M:Goal).
```

This succeeds once for every predicate in your program.

8.13.14.2 Predicates Visible in a Module

The built-in predicate `predicate_property/2` can be used to find the properties of any predicate that is visible to a particular module.

To backtrack through all of the predicates imported by module `m`, use

```
| ?- predicate_property(m:Goal, imported_from(_)).
```

To backtrack through all of the predicates imported by module `m1` from module `m2`, use

```
| ?- predicate_property(m1:Goal, imported_from(m2)).
```

For example, you can load the `basics` module from the library and then remind yourself of what predicates it defines like this:

```
| ?- compile(library(basics)).
% ... loading messages ...

yes
| ?- predicate_property(P, imported_from(basics)).

P = member(_2497,_2498) ;
P = memberchk(_2497,_2498) ;
  .
  .
  .
```

This tells you what predicates are imported into the type-in module from `basics`.

You can also find *all* imports into *all* modules using

```
| ?- predicate_property(M1:G, imported_from(M2)).
```

To backtrack through all of the predicates exported by module `m`, use

```
| ?- predicate_property(m:Goal, exported).
```

There is a library package, `library(showmodule)`, which prints out information about current modules. For more information see Chapter 12 [lib], page 521.

8.13.15 Importing Dynamic Predicates

Imported dynamic predicates may be asserted and retracted. For example, suppose the following file is loaded via `use_module/1`:

```
:- module(m1, [f/1]).
:- dynamic f/1.
f(0).
```

Then `f/1` can be manipulated as if it were defined in the current module. For example,

```
| ?- clause(f(X), true).

X = 0
```

The built-in predicate `listing/1` distinguishes predicates that are imported into the current source module by prefixing each clause with the module name. Thus,

```
| ?- listing(f).

m1:f(0).
```

However, `listing/1` does not prefix clauses with their module if they are defined in the source module itself. Note that

```
| ?- listing.
```

can be used to see all the dynamic predicates defined in or imported into the current type-in module. And

```
| ?- listing(m1:_).
```

can be used to see all such predicates that are defined in or imported into module `m1`.

8.13.16 Module Name Expansion

The concept of a source module is explained in Section 8.13.7 [ref-mod-som], page 275. For any goal, the applicable source module is determined when the goal is compiled rather than when it is executed.

A procedure that needs to refer to the source module has arguments designated for module name expansion. These arguments are expanded at compile time by the transformation

```
X -> M:X
```

where *M* is the name of the source module. For example, the goal `call(X)` is expanded into `call(M:X)` and the goal `clause(Head, Body)` is expanded into `clause(M:Head, Body)`.

Module name expansion is avoided if the argument to be expanded is already a `:/2` term. In this case it is unnecessary since the module to be used has already been supplied by the programmer.

The built-in predicates that use module name expansion, and the arguments requiring module name expansion are shown below. These arguments are labeled '`[MOD]`' in the Arguments field of the reference page for each.

- `abolish(M:Pred)`
- `abolish(M:Name, Arity)`
- `assert(M:Term)`
- `assert(M:Term, Ref)`
- `asserta(M:Term)`
- `asserta(M:Term, Ref)`
- `assertz(M:Term)`
- `assertz(M:Term, Ref)`
- `bagof(T, M:P, S)`
- `call(M:Goal)`
- `check_advice(M:ListOfPredSpecs)`
- `clause(M:Head, Body)`
- `clause(M:Head, Body, Ref)`
- `compile(M:Files)`
- `consult(M:Files)`
- `current_advice(M1:Goal, Port, M2:Action)`
- `current_predicate(Name, M:Term)`
- `debugger(Current, M:New)`
- `ensure_loaded(M:Files)`
- `findall(T, M:Pred, List)`
- `initialization(M:Goal)`
- `listing(M:List)`
- `load_files(M:Files)`
- `load_files(M:Files, Options)`
- `load_foreign_files(M:Files, Libs)`
- `multifile_assertz(M:Term)`
- `nocheck_advice(M:ListOfPredSpecs)`
- `nospy(M:List)`
- `phrase(M:Phrase, S0)`
- `phrase(M:Phrase, S0, S)`
- `predicate_property(M:Goal, Property)`
- `remove_advice(M:Goal, Port, Action)`
- `retract(M:Term)`
- `retractall(M:Term)`
- `save_predicates(M:PredSpecs,File)`

- `save_program(File,M:Goal)`
- `setof(T, M:P, S)`
- `source_file(M:Term,File)`
- `source_file(M:PredSpec,ClauseNumber,File)`
- `spy(M:List)`
- `use_module(M:Files)`
- `use_module(M:File, IL)`
- `use_module(ExportModule, M:File, IL)`
- `volatile(M:PredSpec)`
- `X^(M:Goal)`
- `[M:File|Rest]`

In all of these predicates, `M:` can stand for multiple modules. It is the innermost module that is used in this case. For example, `call(m1:m2:m3:p)` calls `m3:p/0`.

8.13.17 The `meta_predicate` Declaration

Sometimes a user-defined predicate will require module name expansion (see Section 8.13.16 [ref-mod-mne], page 282). This can be specified by providing a `meta_predicate` declaration for that procedure.

Module name expansion is needed whenever the argument of a predicate has some module-dependent meaning. For example, if this argument is a goal that is to be called, it will be necessary to know in which module to call it — or, if the argument is a clause to be asserted, in which module it should go.

Consider, for example, a sort routine to which the name of the comparison predicate is passed as an argument. In this example, the comparison predicate should be called with respect to the module containing the call to the sort routine. Suppose that the sort routine is

```
mysort(+CompareProc, +InputList, -OutputList)
```

An appropriate `meta_predicate` declaration for this is

```
:- meta_predicate mysort(:, +, -).
```

The significant argument in the `mysort/3` term is the ':', which indicates that module name expansion is required for this argument. This means that whenever a goal `mysort(A, B, C)` appears in a clause, it will be transformed at load time into `mysort(M:A, B, C)`, where *M* is the source module. There are some exceptions to this compile-time transformation rule; the goal is not transformed if either of the following applies:

1. *A* is of the form *Module:Goal*.

2. *A* is a variable and the same variable appears in the head of the clause in a module-name-expansion position.

The reason for (2) is that otherwise module name expansion could build larger and larger structures of the form *Mn: . . . :M2:M1:Goal*. For example, consider the following program fragment adapted from the library (see `library(samsort)` for the full program):

```
:- module(samsort, [samsort/3]).

:- meta_predicate
        samsort(:, +, ?),
        sam_sort(+, :, +, +, ?).

samsort(_, [], []) :- !.
samsort(Order, List, Sorted) :-
        sam_sort(List, Order, [], 0, Sorted).
  .
  .
  .
```

Normally, the `sam_sort/5` goal in this example would have the module name of its second argument expanded thus:

```
sam_sort(List, samsort:Order, [], 0, Sorted)
```

because of the `meta_predicate` declaration. However, in this situation the appropriate source module will have already been attached to *Order* because it is the first argument of `samsort/3`, which also has a `meta_predicate` declaration. Therefore it is not useful to attach the module name (`samsort`) to *Order* in the call of `sam_sort/5`.

The argument of a `meta_predicate` declaration can be a term, or a sequence of terms separated by commas. Each argument of each of these terms must be one of the following:

‘`:`’ requires module name expansion

non-negative integer
same as ‘`:`’

‘`+`’, ‘`-`’, ‘`*`’ ignored

The reason for allowing a non-negative integer as an alternative to ‘`:`’ is that this may be used in the future to supply additional information to the cross-referencer (`library(xref)`) and to the Prolog compiler. An integer *n* is intended to mean that that argument is a term that will be supplied *n* additional arguments. Thus, in the example above where the meta-argument is the name of a comparison routine that would be called with two arguments, it would be appropriate to write the integer 2 instead of a ‘`:`’.

The reason for '+', '-' and '*' is simply so that the information contained in a DEC-10 Prolog-style "mode" declaration may be represented in the `meta_predicate` declaration if you wish. There are many examples of `meta_predicate` declarations in the library.

8.13.18 Predicate Summary

- `current_module/[1,2]`
- `meta_predicate/1`
- `module/1`
- `module/2`
- `save_modules/2`
- `use_module/[1,2,3]`

8.14 Modification of the Database

8.14.1 Introduction

The family of assertion and retraction predicates described below enables you to modify a Prolog program by adding or deleting clauses while it is running. These predicates should not be overused. Often people who are experienced with other programming languages have a tendency to think in terms of global data structures, as opposed to data structures that are passed as procedure arguments, and hence they make too much use of assertion and retraction. This leads to less readable and less efficient programs.

An interesting question in Prolog is what happens if a procedure modifies itself, by asserting or retracting a clause, and then fails. On backtracking, does the current execution of the procedure use new clauses that are added to the bottom of the procedure?

> **Historical note:** In early releases of Quintus Prolog, changes to the Prolog database became globally visible upon the success of the built-in predicate modifying the database. An unsettling consequence was that the definition of a procedure could change while it was being run. This could lead to code that was difficult to understand. Furthermore, the memory performance of the interpreter implementing these semantics was poor. Worse yet, the semantics rendered ineffective the added determinacy detection available through indexing.

Quintus Prolog implements the "logical" view in updating dynamic predicates. This means that the definition of a dynamic procedure that is visible to a call is effectively frozen when the call is made. A procedure always contains, as far as a call to it is concerned, exactly the clauses it contained when the call was made.

A useful way to think of this is to consider that a call to a dynamic procedure makes a *virtual copy* of the procedure and then runs the copy rather than the original procedure. Any changes to the procedure made by the call are immediately reflected in the Prolog database, but not in the copy of the procedure being run. Thus, changes to a running procedure will not be visible on backtracking. A subsequent call, however, makes and runs a copy of the modified Prolog database. Any changes to the procedure that were made by an earlier call will now be visible to the new call.

In addition to being more intuitive and easy to understand, the new semantics allow interpreted code to execute with the same determinacy detection (and excellent memory performance) as static compiled code (see Section 2.5.3 [bas-eff-ind], page 36, for more information on determinacy detection).

8.14.2 Dynamic and Static Procedures

All Prolog procedures are classified as being either *static* or *dynamic procedures*. Static procedures can be changed only by completely redefining them using the Load Predicates (see Section 8.4 [ref-lod], page 189). Dynamic procedures can be modified by adding or deleting individual clauses using the assert and retract procedures.

If a procedure is defined by being compiled, it is static by default. If you need to be able to add, delete, or inspect the individual clauses of such a procedure, you must make the procedure dynamic.

There are two ways to make a procedure dynamic:

- If the procedure is to be compiled, then it must be declared to be dynamic before it is defined.
- If the procedure is to be created by assertions only, then the first `assert` operation on the procedure automatically makes it dynamic.

A procedure is declared dynamic by preceding its definition with a declaration of the form:

```
:- dynamic Pred
```

where *Pred* must be a procedure specification of the form *Name*/*Arity*, or a sequence of such specifications, separated by commas. For example,

```
:- dynamic exchange_rate/3, spouse_of/2,
           gravitational_constant/1.
```

where '`dynamic`' is a built-in prefix operator. If *Pred* is not of the specified form an exception is raised, and the declaration is ignored.

Note that the symbol '`:-` ' preceding the word '`dynamic`' is essential. If this symbol is omitted, a permission error is raised because it appears that you are trying to define a

clause for the built-in predicate `dynamic/1`. Although `dynamic/1` is a built-in predicate, it may only be used in declarations.

When a dynamic declaration is encountered in a file being compiled, it is considered to be a part of the redefinition of the procedures specified in its argument. Thus, if you compile a file containing only

```
:- dynamic hello/0
```

the effect will be to remove any previous definition of `hello/0` from the database, and to make the procedure dynamic. You cannot make a procedure dynamic retroactively. If you wish to make an already-existing procedure dynamic it must be redefined.

It is often useful to have a dynamic declaration for a procedure even if it is to be created only by assertions. This helps another person to understand your program, since it emphasizes the fact that there are no pre-existing clauses for this procedure, and it also avoids the possibility of Prolog stopping to tell you there are no clauses for this procedure if you should happen to call it before any clauses have been asserted. This is because unknown procedure catching (see Section 6.1.5.4 [dbg-bas-con-unk], page 120) does not apply to dynamic procedures; it is presumed that a call to a dynamic procedure should simply fail if there are no clauses for it.

If a program needs to make an undefined procedure dynamic, this can be achieved by calling `clause/2` on that procedure. The call will fail because the procedure has no clauses, but as a side-effect it will make the procedure dynamic and thus prevent unknown procedure catching on that procedure. See the Reference page for details of `clause/2`.

Although you can simultaneously declare several procedures to be dynamic, as shown above, it is recommended that you use a separate dynamic declaration for each procedure placed immediately before the clauses for that procedure. In this way when you reconsult or recompile the procedure using the editor interface, you will be reminded to include its dynamic declaration.

Dynamic procedures are implemented by interpretation, even if they are included in a file that is compiled. This means that they are executed more slowly than if they were static, and also that can be printed out using `listing/0`. Dynamic procedures, as well as static procedures, are indexed on their first argument; see Section 2.5.3 [bas-eff-ind], page 36.

8.14.3 Database References

A *database reference* is a term that uniquely identifies a clause or recorded term (see Section 8.14.8 [ref-mdb-idb], page 294) in the database. Database references are provided only to increase efficiency in programs that access the database in complex ways. Use of a database reference to a clause can save repeated searches using `clause/2`. However, it does *not* normally pay to access a clause via a database reference when access via first argument indexing is possible.

Consistency checking is done whenever a reference is used; any attempt to use a reference to a clause that has been retracted will cause an existence error to be raised.

There is no restriction on the use of references. References may be included in asserted clauses. Database references to clauses and in clauses are preserved across saving and restoring via QOF files (see also Section 9.1.1.5 [sap-srs-bas-cld], page 339).

In release 3, a database reference reads and writes like a Prolog term of the form `'$ref'(integer,integer)`; however, it is actually represented as a distinguished atomic data type by the Prolog system. As a result, Prolog operations like `functor/3` and `arg/3` treat database references as they would numbers or atoms:

```
| ?- assert(foo,M).

M = '$ref'(1296804,1)

| ?- functor('$ref'(1296804,1), N, A).

N = '$ref'(1296804,1),
A = 0

| ?- arg(1, '$ref'(1296804,1), A).

no
| ?-
```

Database references can be identified using the type test `db_reference/1`.

In previous releases of Quintus Prolog, operations such as the above were occasionally used on database references so that their components could be indexed on in asserted clauses. Such operations have always been discouraged. In release 3, full indexing is automatically available on the entire database reference, so it is unnecessary to have access to its components.

As in the past, the representation of database references may change in future releases, so programs should not rely on it.

8.14.4 Adding Clauses to the Database

The assertion predicates are used to add clauses to the database in various ways. The relative position of the asserted clause with respect to other clauses for the same predicate is determined by the choice among `assert/1`, `asserta/1`, and `assertz/1`. A database reference that uniquely identify the clause being asserted is established by providing an optional second argument to any of the assertion predicates.

`assert(`*C*`)`
: clause *C* is asserted in an arbitrary position in its predicate

`assert(C,R)`
: as `assert/1`; reference R is returned

`asserta(C)`
: clause C is asserted before existing clauses

`asserta(C,R)`
: as `asserta/1`; reference R is returned

`assertz(C)`
: clause C is asserted after existing clauses

`assertz(C,R)`
: as `assertz/1`; reference R is returned

`multifile_assertz(C)`
: add clause C to the end of a (possibly compiled) multifile procedure

8.14.5 Removing Clauses from the Database

This section briefly describes the predicates used to remove the clauses and/or properties of a predicate from the system.

> **Please note:** Removing all of a predicate's clauses by `retract/1` and/or `erase/1` (see Section 8.14.5.1 [ref-mdb-rcd-efu], page 290) does not remove the predicate's properties (and hence its definition) from the system. The only way to completely remove a predicates clauses *and* properties is to use `abolish/[1,2]`.

`retract(C)`
: erase the first dynamic clause that matches C

`retractall(H)`
: erase every clause whose head matches H

`abolish(F)`
: abolish the predicate(s) specified by F

`abolish(F,N)`
: abolish the predicate named F of arity N

`erase(R)` erase the clause or recorded term (see Section 8.14.8 [ref-mdb-idb], page 294) with reference R

8.14.5.1 A Note on Efficient Use of `retract/1`

WARNING: `retract/1` is a nondeterminate procedure. Thus, we can use

```
| ?- retract((foo(X) :- Body)), fail.
```

to retract all clauses for `foo/1`. A nondeterminate procedure in Quintus Prolog uses a *choice point*, a data structure kept on an internal stack, to implement backtracking. This applies to user-defined procedures as well as to built-in and library procedures. In a simple model, a choice point is created for each call to a nondeterminate procedure, and is deleted on determinate success or failure of that call, when backtracking is no longer possible. In fact, Quintus Prolog improves upon this simple model by recognizing certain contexts in which choice points can be avoided, or are no longer needed.

The Prolog *cut* (‘`!`’) works by removing choice points, disabling the potential backtracking they represented. A choice point can thus be viewed as an “outstanding call”, and a *cut* as deleting outstanding calls.

To avoid leaving inconsistencies between the Prolog database and outstanding calls, a retracted clause is reclaimed only when the system determines that there are no choice points on the stack that could allow backtracking to the clause. Thus, the existence of a single choice point on the stack can disable reclamation of retracted clauses for the procedure whose call created the choice point. Space is recovered only when the choice point is deleted.

Often `retract/1` is used determinately; for example, to retract a single clause, as in

```
| ?- <do some stuff>
      retract(Clause),
     <do more stuff without backtracking>.
```

No backtracking by `retract/1` is intended. Nonetheless, if Clause may match more than one clause in its procedure, a choice point will be created by `retract/1`. While executing “<do more stuff without backtracking>”, that choice point will remain on the stack, making it impossible to reclaim the retracted Clause. Such choice points can also disable tail recursion optimization. If not cut away, the choice point can also lead to runaway retraction on the unexpected failure of a subsequent goal. This can be avoided by simply cutting away the choice point with an explicit `cut` or a local cut (‘`->`’). Thus, in the previous example, it is preferable to write either

```
| ?- <do some stuff>
      retract(Clause),
      !,
     <do more stuff without backtracking>.
```

or

```
| ?- <do some stuff>
     ( retract(Clause) -> true ),
     <do more stuff without backtracking>.
```

This will reduce stack size and allow the earliest possible reclamation of retracted clauses. Alternatively, you could use `retract_first/1`, defined in `library(retract)`.

8.14.6 Accessing Clauses

Goal **Succeeds If:**

`clause(P,Q)`
: there is a clause for a dynamic predicate with head P and body Q

`clause(P,Q,R)`
: there is a clause for a dynamic predicate with head P, body Q, and reference R

`instance(R,T)`
: T is an instance of the clause or term referenced by R

8.14.7 Modification of Running Code: Examples

The following examples show what happens when a procedure is modified while it is running. This can happen in two ways:

1. The procedure calls some other procedure that modifies it.
2. The procedure succeeds nondeterminately, and a subsequent goal makes the modification.

In either case, the question arises as to whether the modifications take effect upon backtracking into the modified procedure. In Quintus Prolog the answer is that they do not. As explained in the overview to this section (see Section 8.14.1 [ref-mdb-bas], page 286), modifications to a procedure affect only calls to that procedure that occur after the modification.

8.14.7.1 Example: assertz

Consider the procedure `foo/0` defined by

```
:- dynamic foo/0.
foo :- assertz(foo), fail.
```

Each call to `foo/0` asserts a new last clause for `foo/0`. After the Nth call to `foo/0` there will be N+1 clauses for `foo/0`. When `foo/0` is first called, a virtual copy of the procedure is made, effectively freezing the definition of `foo/0` for that call. At the time of the call, `foo/0` has exactly one clause. Thus, when `fail/0` forces backtracking, the call to `foo/0` simply fails: it finds no alternatives. For example,

```
| ?- compile(user).
| :- dynamic foo/0.
| foo :- assertz(foo), fail.
| ^D
% user compiled in module user, 0.100 sec 2.56 bytes

yes
| ?- foo.  % The asserted clause is not found

no
| ?- foo.  % A later call does find it, however

yes
| ?-
```

Even though the virtual copy of `foo/0` being run by the first call is not changed by the assertion, the Prolog database is. Thus, when a second call to `foo/0` is made, the virtual copy for that call contains two clauses. The first clause fails, but on backtracking the second clause is found and the call succeeds.

8.14.7.2 Example: retract

```
| ?- assert(p(1)), assert(p(2)), assert(p(3)).

yes
| ?- p(N), write(N), nl, retract(p(2)),
     retract(p(3)), fail.
1
2
3

no
| ?- p(N), write(N), fail.
1
no
| ?-
```

At the first call to `p/1`, the procedure has three clauses. These remain visible throughout execution of the call to `p/1`. Thus, when backtracking is forced by `fail/0`, N is bound to 2 and written. The retraction is again attempted, causing backtracking into `p/1`. N is bound to 3 and written out. The call to `retract/1` fails. There are no more clauses in `p/1`, so the query finally fails. A subsequent call to `p/1`, made after the retractions, sees only one clause.

8.14.7.3 Example: abolish

```
| ?- compile(user).
| :- dynamic q/1.
| q(1).
| q(2).
| q(3).
| ^D
% user compiled in modules user, 0.117 sec 260 bytes

yes
| ?- q(N), write(N), nl, abolish(q/1), fail.
1
2
3

no
| ?-
```

Procedures that are abolished while they have outstanding calls do not become invisible to those calls. Subsequent calls however, will find the procedure undefined.

8.14.8 The Internal Database

The following predicates are provided solely for compatibility with other Prolog systems. Their semantics can be understood by imagining that they are defined by the following clauses:

```
recorda(Key, Term, Ref) :-
     functor(Key, Name, Arity),
     functor(F, Name, Arity),
     asserta('$recorded'(F,Term), Ref).
recordz(Key, Term, Ref) :-
     functor(Key, Name, Arity),
     functor(F, Name, Arity),
     assertz('$recorded'(F,Term), Ref).
recorded(Key, Term, Ref) :-
     functor(Key, Name, Arity),
     functor(F, Name, Arity),
     clause('$recorded'(F,Term), _, Ref).
```

The reason for the calls to `functor/3` in the above definition is that only the principal functor of the key is significant. If *Key* is a compound term, its arguments are ignored.

> **Please note:** Equivalent functionality and performance, with reduced memory costs, can usually be had through normal dynamic procedures and indexing

(see Section 8.14.1 [ref-mdb-bas], page 286, and indexing tutorial in "Writing Efficient Programs" section).

In some implementations, database references are also represented by compound terms, and thus subject to the limitation described above.

`recorda(+Key, +Term, -Ref)` records the *Term* in the internal database as the first item for the key *Key*; a database reference to the newly-recorded term is returned in *Ref*.

`recordz(+Key, +Term, -Ref)` is like `recorda/3` except that it records the term as the last item in the internal database.

`recorded(*Key, *Term, *Ref)` searches the internal database for a term recorded under the key *Key* that unifies with *Term*, and whose database reference unifies with *Ref*.

`current_key(*KeyName, *KeyTerm)` succeeds when *KeyName* is the atom or integer that is the name of *KeyTerm*. *KeyTerm* is an integer, atom, or compound term that is the key for a currently recorded term.

8.14.9 Summary of Predicates

- `abolish/[1,2]`
- `assert/[1,2]`
- `asserta/[1,2]`
- `assertz/[1,2]`
- `clause/[2,3]`
- `erase/1`
- `current_key/3`
- `instance/2`
- `recorda/3`
- `recorded/3`
- `recordz/3`
- `retract/1`
- `retractall/1`

8.15 Sets and Bags: Collecting Solutions to a Goal

8.15.1 Introduction

When there are many solutions to a goal, and a list of all those solutions is desired, one means of collecting them is to write a procedure that repeatedly backtracks into that goal to get another solution. In order to collect all the solutions together, it is necessary to use the

database (via assertion) to hold the solutions as they are generated, because backtracking to redo the goal would undo any list construction that had been done after satisfying the goal.

The writing of such a backtracking loop can be avoided by the use of one of the built-in predicates `setof/3`, `bagof/3` and `findall/3`, which are described below. These provide a nice logical abstraction, whereas with a user-written backtracking loop the need for explicit side-effects (assertions) destroys the declarative interpretation of the code. The built-in predicates are also more efficient than those a user could write.

8.15.2 Collecting a Sorted List

`setof(`*`Template, Generator, Set`*`)` returns the set *Set* of all instances of *Template* such that *Generator* is provable, where that set is non-empty. The term *Generator* specifies a goal to be called as if by `call/1`. *Set* is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 8.9.7 [ref-lte-cte], page 242).

Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case *Set* will only provide an imperfect representation of what is in reality an infinite set.

If *Generator* is instantiated, but contains uninstantiated variables that do not also appear in *Template*, then `setof/3` can succeed nondeterminately, generating alternative values for *Set* corresponding to different instantiations of the free variables of *Generator*. (It is to allow for such usage that *Set* is constrained to be non-empty.) For example, if your program contained the clauses

```
likes(tom, beer).
likes(dick, beer).
likes(harry, beer).
likes(bill, cider).
likes(jan, cider).
likes(tom, cider).
```

then the call

```
| ?- setof(X, likes(X,Y), S).
```

might produce two alternative solutions via backtracking:

```
X = _872,
Y = beer,
S = [dick,harry,tom] ;

X = _872,
Y = cider,
S = [bill,jan,tom] ;

no
```

The call

```
| ?- setof((Y,S), setof(X,likes(X,Y),S), SS).
```

would then produce

```
Y = _402,
S = _417,
X = _440,
SS = [(beer,[dick,harry,tom]),(cider,[bill,jan,tom])] ;

no
```

8.15.2.1 Existential Quantifier

X ^ P is recognized as meaning "there exists an X such that P is true", and is treated as equivalent to simply calling P. The use of the explicit existential quantifier outside `setof/3` and `bagof/3` is superfluous.

Variables occurring in *Generator* will not be treated as free if they are explicitly bound within *Generator* by an existential quantifier. An existential quantification is written:

```
Y^Q
```

meaning "there exists a Y such that Q is true", where Y is some Prolog variable. For example:

```
| ?- setof(X, Y^likes(X,Y), S).
```

would produce the single result

```
X = _400,
Y = _415,
S = [bill,dick,harry,jan,tom] ;

no
```

in contrast to the earlier example.

Furthermore, it is possible to existentially quantify a term, where all the variables in that term are taken to be existentially quantified in the goal. e.g.

```
A=term(X,Y), setof(Z, A^foo(X,Y,Z), L).
```

will treat X and Y as if they are existentially quantified.

8.15.3 Collecting a Bag of Solutions

`bagof/3` is is exactly the same as `setof/3` except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. This relaxation saves time and space in execution.

8.15.3.1 Collecting All Instances

`findall/3` is a special case of `bagof/3`, where all free variables in the generator are taken to be existentially quantified. Thus the use of the operator `^` is avoided. Because `findall/3` avoids the relatively expensive variable analysis done by `bagof/3`, using `findall/3` where appropriate rather than `bagof/3` can be considerably more efficient.

Previously, `findall/3` was available in `library(findall)`.

8.15.4 Library Support

- `library(basics)`
- `library(lists)`
- `library(ordsets)`
- `library(sets)`

8.15.5 Predicate Summary

- `setof/3`
- `bagof/3`
- `findall/3`
- `^/2`

8.16 Grammar Rules

This section describes Quintus Prolog's grammar rules, and the translation of these rules into Prolog clauses. At the end of the section is a list of grammar-related built-in predicates.

8.16.1 Definite Clause Grammars

Prolog's grammar rules provide a convenient notation for expressing definite clause grammars, which are useful for the analysis of both artificial and natural languages.

The usual way one attempts to make precise the definition of a language, whether it is a natural language or a programming lanaguage, is through a collection of rules called a "grammar". The rules of a grammar define which strings of words or symbols are valid sentences of the language. In addition, the grammar generally analyzes the sentence into a structure that makes its meaning more explicit.

A fundamental class of grammar is the context-free grammar (CFG), familiar to the computing community in the notation of "BNF" (Backus-Naur form). In CFGs, the words, or basic symbols, of the language are identified by "terminal symbols", while categories of phrases of the language are identified by non-terminal symbols. Each rule of a CFG expresses a possible form for a non-terminal, as a sequence of terminals and non-terminals. The analysis of a string according to a CFG is a parse tree, showing the constitutent phrases of the string and their hierarchical relationships.

Context-free grammars (CFGs) consist of a series of rules of the form:

```
nt --> body.
```

where *nt* is a non-terminal symbol and body is a sequence of one or more items separated by commas. Each item is either a non-terminal symbol or a sequence of terminal symbols. The meaning of the rule is that *body* is a possible form for a phrase of type *nt*. A non-terminal symbol is written as a Prolog atom, while a sequence of terminals is written as a Prolog list, whereas a terminal may be any Prolog term.

Definite clause grammars (DCGs) are a generalization of context-free grammars and rules corresponding to DCGs are referred to as "Grammar Rules". A grammar rule in Prolog takes the general form

```
head --> body.
```

meaning "a possible form for *head* is *body*". Both *body* and *head* are sequences of one or more items linked by the standard Prolog conjunction operator '`,`' (comma).

Definite clause grammars extend context-free grammars in the following ways:

- A non-terminal symbol may be any Prolog term (other than a variable or integer).
- A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list '`[]`'. If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list ('`[]`' or '`""`').
- Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. These extra conditions allow the explicit use of procedure

calls in the body of a rule to restrict the constitutents accepted. Such procedure calls are written enclosed in curly brackets (‘`{`’ and ‘`}`’).

- The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
- Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ‘`;`’ (semicolon) as in Prolog. (The disjunction operator can also be written as ‘`|`’ (vertical-bar).)
- The cut symbol ‘`!`’ (exclamation point) may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in curly brackets. The conditional arrow ‘`->`’ can also be used in grammar rules, without the curly brackets. However, all other control predicates, `repeat/0` for example, can only be used within curly brackets. If you use the goal `repeat/0` without the brackets it will be taken to be a non-terminal symbol.
- The extra arguments of non-terminals provide the means of building structure (such as parse trees) in grammar rules. As non-terminals are “expanded” by matching against grammar rules, structures are progressively built up in the course of the unification process.
- The extra arguments of non-terminals can also provide a general treatment of context dependency by carrying test and contextual information.

8.16.2 How to Use the Grammar Rule Facility

Following is a summary of the steps that enable you to construct and utilitze definte clause grammars:

STEPS:

1. Write a grammar, using `-->/2` to formulate rules.
2. Compile the file containing the grammar rules. The Load Predicates call `expand_term/2`, which translates the grammar rules into Prolog clauses.
3. Use `phrase/[2,3]` to parse or generate strings.

OPTIONAL STEPS:

1. Modify the way in which Prolog translates your grammar rules by defining clauses for `term_expansion/2`.
2. In debugging or in using the grammar facility for more obscure purposes it may be useful to understand more about `expand_term/2` and `'C'/3`.

8.16.3 An Example

As an example, here is a simple grammar that parses an arithmetic expression (made up of digits and operators) and computes its value. Create a file containing the following rules:

grammar.pl

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

In the last rule, C is the ASCII code of a decimal digit.

This grammar can now be used to parse and evaluate an expression by means of the built-in predicates `phrase/2` and `phrase/3`. For example,

```
| ?- [grammar].
| ?-  phrase(expr(Z), "-2+3*5+1").

Z = 14

| ?-  phrase(expr(Z), "-2+3*5", Rest).

Z = 13,
Rest = [] ;

Z = 1,
Rest = "*5" ;

Z = -2,
Rest = "+3*5" ;

no
```

8.16.4 Translation of Grammar Rules into Prolog Clauses

Grammar rules are merely a convenient abbreviation for ordinary Prolog clauses. Each grammar rule is translated into a Prolog clause as it is compiled. This translation is described below.

The procedural interpretation of a grammar rule is that it takes an input list of symbols or character codes, analyzes some initial portion of that list, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the

input and output lists are not written explicitly in a grammar rule, but are added when the rule is translated into an ordinary Prolog clause. The translations shown differ from the output of `listing/[0,1]` in that internal translations such as variable renaming are not represented. This is done in the interests of clarity. For example, a rule such as (A) will be depicted as translating into (B) rather than (C).

```
p(X) --> q(X).                                                    (A)

p(X, S0, S) :-
        q(X, S0, S).                                              (B)

p(A,B,C) :-
        q(A,B,C).                                                 (C)
```

If there is more than one non-terminal on the right-hand side, as in (D) the corresponding input and output arguments are identified, translating into (E):

```
p(X, Y) --> q(X), r(X, Y), s(Y).                                  (D)


p(X, Y, S0, S) :-                                                 (E)
    q(X, S0, S1),
    r(X, Y, S1, S2),
    s(Y, S2, S).
```

Terminals are translated using the built-in predicate `'C'(S1, X, S2)`, read as "point S1 is connected by terminal X to point S2", and defined by the single clause

```
'C'([X|S], X, S).
```

(This predicate is not normally useful in itself; it has been given the name uppercase '`C`' simply to avoid pre-empting a more useful name.) Then, for instance (F) is translated into (G):

```
p(X) --> [go, to], q(X), [stop].                                  (F)

p(X, S0, S) :-                                                    (G)
    'C'(S0, go, S1),
    'C'(S1, to, S2),
    q(X, S2, S3),
    'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls, enclosed in curly braces, naturally translate into themselves. For example (H) translates to (I):

```
p(X) --> [X], {integer(X), X > 0}, q(X).                          (H)
```

```
p(X, S0, S) :-                                                    (I)
    'C'(S0, X, S1),
    integer(X),
    X > 0,
    q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule, enclosed in square brackets, translate into `'C'/3` goals with the first and third arguments reversed. For example, (J) becomes (K):

```
is(N), [not] --> [aint].                                          (J)
```

```
is(N, S0, S) :-                                                   (K)
    'C'(S0, aint, S1),
    'C'(S, not, S1).
```

Disjunction has a fairly obvious translation. For example, (L), a rule that equates phrases like "(sent) a letter to him" and "(sent) him a letter", translates to (M):

```
args(X, Y) -->                                                    (L)
        dir(X), [to], indir(Y) |
        indir(Y), dir(X).
```

```
args(X, Y, S0, S) :-                                              (M)
    (   dir(X, S0, S1),
        'C'(S1, to, S2),
        indir(Y, S2, S)
    |   indir(Y, S0, S1),
        dir(X, S1, S)
    ).
```

8.16.4.1 Listing Grammar Rules

In order to look at these translations, declare the grammar rules dynamic and use `listing/[0,1]`. However, in a grammar rule like '*head* `-->` *body*', if the functor of *head* is `foo/n`, the dynamic declaration is for `foo/n+2`. For example, the following declaration for grammar rule (L) would enable you to list its translation, (M):

```
:- dynamic(args/4).
```

8.16.5 Summary of Predicates

- `-->/2`
- `'C'/3`
- `expand_term/2`

- `phrase/[2,3]`
- `term_expansion/2`

8.17 On-line Help

8.17.1 Introduction

The documentation for Quintus Prolog is contained in the Quintus Prolog Manual. This documentation can be accessed on-line by using special built-in predicates, which are described below. These predicates view the manual as a single tree structure with nodes threaded together by cross-references, which you can traverse in order to find the information you need. It is easier to use this help system through the Emacs interface or the Quintus User Interface, but it is possible to do everything without them. For documentation on using the help system for Quintus User Interface see Section 3.6.1 [qui-hlp-hlp], page 69.

There are two basic ways of getting to information in the help system. The first is via a series of menus, which corresponds to the hierarchy of parts, chapters, sections and subsections in the manual. This method of access is analogous to looking through the Table of Contents sections of the printed document. The second method is by keyword search. This method is analogous to looking in the Index sections of the printed manual. These access methods are described in Section 8.17.3.2 [ref-olh-hlp-sum], page 306.

8.17.2 Help Files

8.17.2.1 Overview

The help system is largely based on the Info file format, which GNU Emacs uses for online documentation. When accessed by following a menu entry or a cross reference, the help system will locate the corresponding Info node. When accessed by keyword search, an ad-hoc menu will be created instead.

If you are running outside Emacs, the information will be displayed on the terminal, controlled by the environment variable `PAGER`.

If you are running under Emacs, Emacs will try to use the Info subsystem as far as possible. If Emacs is unable to find the Info nodes requested, or when displaying an ad-hoc menu, it displays the information in a buffer where specialized key bindings apply as summarized in Section 8.17.3 [ref-olh-hlp], page 306. The most important key binding is '`?`', which will display the appropriate key-binding summary. After typing a question mark, typing a '`b`' will get you back to where you were.

The organization of the help files corresponds directly to that of the printed manual. Files that do not begin with a menu correspond to *leaf* nodes of the manual tree; that is, to sections of the manual that have no subdivisions. Files that do begin with a menu correspond to the non-leaf nodes of the tree.

8.17.2.2 Menus

A menu consists of a numbered sequence of choices. Each choice has the form

```
* {manual(Tag)}                    Subject
```

If you are running Prolog without Emacs, you can select a menu choice by typing

```
| ?- manual(Tag).
```

where *Tag* is exactly as shown in the menu. Tags are in general of the form *Chapter-Section-Subsection-...* where *Chapter*, *Section*, and *Subsection* are abbreviations of a section in the printed manual.

Under Emacs: you will find that a special mode is entered when you are looking at a menu. See Section 8.17.3 [ref-olh-hlp], page 306, for more information.

8.17.2.3 Cross-References

Occasionally you will see a cross-reference in the text. Cross-references look like (A) in the printed manual, and like (B) in the on-line manual:

```
...see also Section 1.3 [int-dir], page 11                    (A)
```

and

```
...see also {manual(int-dir)}                                 (B)
```

If you are not using Emacs, then you should type the following goal in order to follow this cross-reference.

```
| ?- manual(int-dir).
```

Under Emacs: There is a more convenient way to do this: type 'x' to move the cursor to the front of the cross-reference, then type ⟨RET⟩ to display the cross-referenced text.

8.17.2.4 Displaying help files

The help-system always writes its output to `current_output` instead of `user_output`. This makes it possible to redirect information produced by `help/[0,1]` or `manual/[0,1]` to a

file. For example, to save the documentation on `assert/1` in a file called '`assert.doc`', type:

```
?- tell('assert.doc'), help(assert), told.
```

Furthermore, if the `current_output` is the same as `user_output` (i.e. the terminal), then the environment variable `PAGER` is used, if set, to display the information. If `PAGER` is not set, the default pager, `more`, is used.

Under Emacs: This strategy is not applicable in the Emacs interface, which provides its own way of saving displayed text, nor in QUI.

8.17.3 Emacs Commands for Using the Help System

8.17.3.1 Emacs Commands

The keys available when viewing a menu file of the help system under Emacs are:

q	Quit the help system.
b	
l	Move Back to the previous help file viewed.
u	Move Up to the parent menu in the manual hierarchy.
?	Display this manual page.
x	Move to the next menu entry.
X	Move to the previous menu entry.
⟨RET⟩	Select the current menu entry.
⟨ESC⟩ *z*	Scroll the menu one line up.
⟨ESC⟩ *^z*	Scroll the menu one line down.
⟨SPC⟩	Scroll the menu one page up.
⟨BSP⟩	Scroll the menu one page down.
<	Go to the beginning of the buffer.
>	Go to the end of the buffer.

> **Please note:** If you are viewing this under Emacs, type *b* to return to where you just were.

8.17.3.2 Predicate Summary

`help`	prints the top level menu of the manual set

`help(Topic)`
: gives index access to the on-line manual

`manual`
: accesses the top level of the on-line manual

`manual(X)`
: accesses the manual section specified by X

`user_help`
: hook; called by `help/0`

8.18 Access to the Operating System

8.18.1 Overview

The predicate described here, `unix/1`, provides the most commonly needed access to the operating system. This minor extension to the Prolog system should be sufficient for most of your needs. However, you can also extend the Prolog system with additional C code, including system calls, using the foreign function interface (see Section 10.3 [fli-p2f], page 375).

The reason for channeling all the interaction with the operating system through a single built-in predicate, rather than having separate predicates for each function, is simply to localize the system dependencies. Admittedly, this makes for more cumbersome commands, so you may wish to put some clauses such as these in your '`prolog.ini`' file:

```
cd :- unix(cd).
cd(X) :- unix(cd(X)).
```

Initialization files are discussed in Section 8.3 [ref-pro], page 186.

8.18.2 Executing Commands from Prolog

8.18.2.1 Changing the Working Directory

`unix(cd(+Path))` changes the working directory of Prolog (and of Emacs, if running under the editor interface) to that specified by *Path*, which should be an atom corresponding to a legal directory. If *Path* is not specified, `unix(cd)` changes the working directory of Prolog (and of Emacs if running under the editor interface) to your home directory.

Note that the ⟨ESC⟩ *x cd Path* and ⟨ESC⟩ *x cd* commands under Emacs have the same effect as this, except that Emacs also provides filename completion.

8.18.2.2 Other Commands

To spawn a command interpreter and execute a command use `unix(shell(+Command))`. The reference page for `unix/1` contains examples. If *Command* is an atom, a new shell process is invoked, and *Command* is passed to it for execution as a shell command. Under UNIX, the shell invoked depends on your `SHELL` environment variable. Under Windows, the default shell is used, as determined by the `system()` C library function. If the shell returns with a non-zero result (for example, because the command was not found), `unix(shell(Command))` simply fails.

Similarly, a new standard shell process can be invoked by calling `unix(system(+Command))`. The standard shell is `sh(1)` under UNIX and as determined by the `system()` C library function under Windows.

8.18.2.3 Spawning an Interactive Shell

`unix(shell)` starts up an interactive shell. Under UNIX, the shell run depends on your `SHELL` environment variable. You can exit from the shell by typing `^d` (or your end-of-file character) unless under Emacs, in which case you should type `^x ^d`. The Prolog idiom `end_of_file.` will not work in this context. If `ignoreeof` is set (for example, in your '`.cshrc`' file), `^d` may not work (setting `ignoreeof` turns off `^d`). In this case, you may type `exit` to the shell to kill it. The call to `unix(shell)` fails if a non-zero result is returned by the shell.

> **Please note:** Under UNIX, invoking the predicate `unix(shell)` when your `SHELL` environment variable is set to a non-standard shell (not `csh(1)` or `sh(1)`) may cause echoing problems under the Emacs interface due to the `stty` settings of the non-standard shell. If a non-standard shell proves to be a problem, an alternative is to use either `unix(shell(sh))` or `unix(shell(csh))` to invoke the standard shell, respectively.

8.18.3 Accessing Command Line Arguments

To return the arguments that were typed on the command line following the command that invoked the current Prolog saved state (see Section 8.3 [ref-pro], page 186) use either of the following:

```
| ?- unix(argv(ArgList)).  % mnemonic; 'vector', 'value'

| ?- unix(args(ArgList)).  % mnemonic; 'string'
```

8.18.3.1 Arguments as Numbers or as Strings

The difference between using '`argv`' and '`args`' is evident when Prolog is invoked with numbers as arguments.

The objects returned by `unix(argv(_))` are Prolog objects; that is, if the command line argument is a number, then it will be returned as a number. Thus:

```
% prolog 1
    .
    .
    .
| ?- unix(argv([1])).

yes
| ?- unix(argv(['1'])).

no
| ?- unix(args(['1'])).

yes

% prolog 1 6.999999
    .
    .
    .

| ?- unix(argv(X)).

X = [1, 6.9999999E+00]

| ?- unix(args(X)).

X = ['1', '6.999999']
```

So if your program treats the command line argument as a number, use the form with '`argv`', but if it is to be treated as a string, use '`args`'. For example if the program is called with a number and performs some arithmetic operation on the argument then displays the result, use '`argv`'.

```
| ?- [user].
| runtime_entry(start) :-
      unix(argv([A])),
      Y is A+1,
      display(Y).
| ^D
% user compiled in module user, 0.083 sec 8 bytes

yes
| ?- runtime_entry(start).
46
yes
```

8.18.3.2 Accessing Prolog's Arguments from C

The command line arguments passed to Prolog can be accessed from C through `QP_argv` and `QP_argc`. These are similar to `argc` and `argv` of the `main()` API, except that they only store Prolog's arguments. In '`<quintus/quintus.h>`' they are classed as

```
extern  int QP_argc
extern  char **QP_argv
```

8.18.4 Predicate Summary

- `unix/1`
- `QP_initialize()`
- `toplevel`

8.18.5 Library Support

`system/1` — from `library(strings)`

8.19 Errors and Exceptions

8.19.1 Overview

Whenever the Prolog system encounters a situation where it cannot continue execution, it raises an exception. For example, if a built-in predicate detects an argument of the wrong type, it raises a `type_error` exception. The manual page description of each built-in predicate lists the kinds of exceptions that can be raised by that built-in predicate.

The default effect of raising an exception is to terminate the current computation and then print an error message. After the error message, you are back at Prolog's top level. For example, if the goal

```
X is a/2
```

is executed somewhere in a program you get

```
! Type error in argument 2 of is/2
! number expected, but a found
! goal:  A is a/2

| ?-
```

Particular things to notice in this message are:

'`! `' This character indicates that this is an error message rather than a warning[2] or informational message.

'`Type Error`'
This is the *exception class*. Every exception raised by the system is categorized into one of a small number of classes. The classes are listed in Section 8.19.4 [ref-ere-err], page 313.

'`goal:`' The goal that caused the exception to be raised.

8.19.2 Raising Exceptions

You can raise exceptions from your own code using the built-in predicate

```
raise_exception(+ExceptionCode)
```

The argument to this predicate is the *exception code*; it is an arbitrary non-variable term of which the principal functor indicates the exception class. You can use the same exception classes as the system (see Section 8.19.4 [ref-ere-err], page 313), or you can use your own exception classes.

Error messages like the one shown above are printed using the built-in predicate `print_message/2`. One of the arguments to `print_message` is the exception code. `print_message` can be extended, as described in Section 8.20 [ref-msg], page 325, so that you can have appropriate error messages printed corresponding to your own exception classes.

[2] The difference between an error (including exceptions) and a warning: A *warning* is issued if Prolog detects a situation that is likely to cause problems, though it is possible that you intended it. An *error*, however, indicates that Prolog recognizes a situation where it cannot continue.

8.19.3 Handling Exceptions

It is possible to protect a part of a program against abrupt termination in the event of an exception. There are two ways to do this:

- Trap exceptions to a particular goal by calling `on_exception/3` as described in Section 8.19.3.1 [ref-ere-hex-pgo], page 312.
- Handle undefined predicates or subsets of them through the hook predicate `unknown_predicate_handler/3`; see Section 8.19.3.2 [ref-ere-hex-hup], page 313.

8.19.3.1 Protecting a Particular Goal

The built-in predicate `on_exception/3` enables you to handle exceptions to a specific goal:

`on_exception(`*`?ExceptionCode, +ProtectedGoal, +Handler`*`)`

ProtectedGoal is executed. If all goes well, it will behave just as if you had written *ProtectedGoal* without the `on_exception/3` wrapper. If an exception is raised while *ProtectedGoal* is running, Prolog will abandon *ProtectedGoal* entirely. Any bindings made by *ProtectedGoal* will be undone, just as if it had failed. Side-effects, such as asserts and retracts, are not undone, just as they are not undone when a goal fails. After undoing the bindings, Prolog tries to unify the exception code raised with the *ExceptionCode* argument. If this unification succeeds, *Handler* will be executed as if you had written

> *ExceptionCode*=<the actual exception code>,
> *Handler*

If this unification fails, Prolog will keep searching up the ancestor list looking for another exception handler. If it reaches Prolog's top level (or a break level) without having found a call to `on_exception/3` with a matching *ExceptionCode*, an appropriate error message is printed (using `print_message/2`).

ProtectedGoal need not be determinate. That is, backtracking into *ProtectedGoal* is possible, and the exception handler becomes reactivated in this case. However, if *ProtectedGoal* is determinate, then the call to `on_exception/3` is also determinate.

Setting up an exception handler with `on_exception/3` is cheap provided that *ProtectedGoal* is an ordinary goal. Some efficiency is lost, in the current implementation, if *ProtectedGoal* is of the form *(Goal1,Goal2)* or *(Goal1;Goal2)*.

The *ProtectedGoal* is logically *inside* the `on_exception/3` form, but the *Handler* is *not*. If an exception is raised inside the *Handler*, this `on_exception/3` form will *not* be reactivated. If you want an exception handler that protects itself, you have to program it, perhaps like this:

```
recursive_on_exception_handler(Err, Goal, Handler) :-
    on_exception(Err, Goal,
        recursive_on_exception_handler(Err, Handler, Handler)).
```

8.19.3.2 Handling Unknown Predicates

Users can write a handler for the specific exception occurring when an undefined predicate is called by defining clauses for the hook predicate `unknown_predicate_handler/3`. This can be thought of as a "global" exception handler for this particular exception, because unlike `on_exception/3`, its effect is not limited to a particular goal. Furthermore, the exception is handled at the point where the undefined predicate is called.

The handler can be written to apply to all unknown predicates, or to a class of them. The reference page contains an example of constraining the handler to certain predicates.

8.19.4 Error Classes

Exceptions raised by the Prolog system are called errors. The set of exception classes used by the system has been kept small. Here is a complete list:

Instantiation Error
: An input argument is insufficiently instantiated.

Type Error
: An input argument is of the type.

Domain Error
: An input argument is illegal but of the right type.

Range Error
: A value specified for an output argument is illegal.

Representation Error
: A computed value cannot be represented.

Existence Error
: Something does not exist.

Permission Error
: Specified operation is not permitted.

Context Error
: Specified operation is not permitted in this context.

Consistency Error
: Two otherwise correct values are inconsistent with each other.

Syntax Error
: Error in reading a term.

Resource Error
: Some resource limit has been exceeded.

System Error
: An error detected by the operating system.

The exception codes corresponding to these classes are:

- `instantiation_error(`*`Goal, ArgNo`*`)`
- `type_error(`*`Goal, ArgNo, TypeName, Culprit`*`)`
- `domain_error(`*`Goal, ArgNo, DomainName, Culprit, Message`*`)`
- `range_error(`*`Goal, ArgNo, TypeName, Culprit`*`)`
- `representation_error(`*`Goal, ArgNo, Message`*`)`
- `existence_error(`*`Goal, ArgNo, ObjectType, Culprit, Message`*`)`
- `permission_error(`*`Goal, Operation, ObjectType, Culprit, Message`*`)`
- `context_error(`*`Goal, ContextType, CommandType`*`)`
- `consistency_error(`*`Goal, Culprit1, Culprit2, Message`*`)`
- `syntax_error(`*`Goal, Position, Message, Left, Right`*`)`
- `resource_error(`*`Goal, Resource, Message`*`)`
- `system_error(`*`Message`*`)`

Most exception codes include a copy of the *Goal* that raised the exception.

In general, built-in predicates that cause side-effects, such as the opening of a stream or asserting a clause into the Prolog database, attempt to do all error checking before the side-effect is performed. Unless otherwise indicated in the documentation for a particular predicate or error class, it should be assumed that goals that raise exceptions have not performed any side-effect.

8.19.4.1 Instantiation Errors

An instantiation error occurs when a predicate or command is called with one of its input arguments insufficiently instantiated.

The exception code associated with an instantiation error is

```
instantiation_error(Goal, ArgNo)
```

ArgNo is a non-negative integer indicating which argument caused the problem. ArgNo=0 means that the problem could not be localized to a single argument. , `atom_chars/2`, `functor/3` etc., which allow alternative instantiation patterns. Maybe this will be fixed but lets leave it vague for now. -DLB}

Note that the *ArgNo*th argument of *Goal* might well be a non-variable: the error is *in* that argument. For example, the goal

```
X is Y+1
```

where *Y* is uninstantiated raises the exception

```
instantiation_error(_2298 is _2301+1,2)
```

because the second argument to `is/2` contains a variable.

8.19.4.2 Type Errors

A type error occurs when an input argument is of the wrong *type*. In general, a *type* is taken to be a class of terms for which there exists a unary *type test predicate*. Some types are built-in, such as `atom/1` and `integer/1`. Some are defined in `library(types)`, such as `chars/1`.

The type of a term is the sort of thing you can tell just by looking at it, without checking to see how *big* it is. So "integer" is a type, but "non-negative integer" is not, and "atom" is a type, but "atom with 5 letters in its name" and "atom starting with '`x`'" are not.

The point of a type error is that you have *obviously* passed the wrong sort of argument to a command; perhaps you have switched two arguments, or perhaps you have called the wrong predicate, but it isn't a subtle matter of being off by one.

Most built-in predicates check all their input arguments for type errors.

The exception code associated with a type error is

```
type_error(Goal, ArgNo, TypeName, Culprit)
```

ArgNo *Culprit* occurs somewhere in the *ArgNo*th argument of *Goal*.

TypeName
says what sort of term was expected; it should be the name of a unary predicate that is true of whatever terms would not provoke a type error.

Some *TypeName*s recognized by the system include:

- `0` — No type name specified
- `atom`
- `atomic`
- `callable`
- `db_reference`
- `integer`
- `number`

Culprit is the actual term being complained about: *TypeName*(*Culprit*) should be false.

For example, suppose we had a predicate

```
date_plus(NumberOfDays, Date0, Date)
```

true when *Date0* and *Date* were `date(Y,M,D)` records and *NumberOfDays* was the number of days between those two dates. You might see an error term such as

```
type_error(/* Goal     */ date_plus(27, date(18,mar,11), _235),
           /* Argno    */ 2,
           /* TypeName */ integer,
           /* Culprit  */ mar
```

8.19.4.3 Domain Errors

A domain error occurs when an input argument is of the right type but there is something wrong with its value. For example, the second argument to `open/3` is supposed to be an atom that represents a valid mode for opening a file, such as `read` or `write`. If a number or a compound term is given instead, that is a type error. If an atom is given that is not a valid mode, that is a domain error.

The main reason that we distinguish between type errors and domain errors is that they usually represent different sorts of mistake in your program. A type error usually indicates that you have passed the wrong argument to a command, whereas a domain error usually indicates that you passed the argument you meant to check, but you hadn't checked it enough.

The exception code associated with a domain error is

```
domain_error(Goal, ArgNo, DomainName, Culprit, Message)
```

The arguments correspond to those of the exception code for a type error, except that *DomainName* is not in general the name of a unary predicate: it needn't even be an atom. For example, if some command requires an argument to be an integer in the range 1..99, it might use `between(1,99)` as the *DomainName*. With respect to the `date_plus` example under **Type Errors**, if the month had been given as 13 it would have passed the type test but would raise a domain error.

For example, the goal

```
open(somefile,rread,S)
```

raises the exception

```
domain_error(open(somefile,rread,_2490),2,'i/o mode',rread,'')
```

The Message argument is used to provide extra information about the problem.

Some *DomainName*s recognized by the system include:

- `0` — no domain name specified

- `atom`
- `'atom or list'`
- `atomic`
- `between(L,H)` — 'something between L and H
- `built_in`
- `callable`
- `char`
- `character`
- `compiled`
- `directory`
- `declaration`
- `db_reference`
- `foreign`
- `file`
- `flag`
- `imported_predicate`
- `integer`
- `interpreted`
- `list`
- `module`
- `number`
- `one_of(List)` — a member of the set *List*
- `pair`
- `predicate_specification`
- `procedure`
- `stream`
- `term`
- `value(X)` — the value X
- `clause`

8.19.4.4 Range Errors

A range error occurs when an *output* argument was supplied with an illegal value. This is similar to a type error or a domain error, except that it is a hint that a variable would be a good thing to supply instead; type and domain errors are associated with input arguments, where a variable would usually not be a good idea.

The exception code associated with a range error is

```
range_error(Goal, ArgNo, TypeName, Culprit)
```

This has the same arguments as a type error.

Most built-in predicates do not raise any range errors. Instead they fail quietly when an output argument fails to unify.

8.19.4.5 Representation Errors

A representation error occurs when your program calls for the computation of some well-defined value that cannot be represented.

Most representation errors are some sort of overflow:

```
functor(T, f, 1000)              % maximum arity is 255
X is 16'7fffffff, Y is X+1       % 32-bit signed integers
atom_chars(X, L)                 % if length of L > 1024
```

are all representation errors. Floating-point overflow is a representation error.

The exception code for a representation error is

```
representation_error(Goal, ArgNo, Message)
```

ArgNo identifies the argument of the goal that cannot be constructed.

Message further classifies the problem. A message of `0` or `''` provides no further information.

8.19.4.6 Existence Errors

An existence error occurs when a predicate attempts to access something that does not exist. For example, trying to compile a file that does not exist, erasing a database reference that has already been erased. A less obvious example: reading past the end of file marker in a stream is regarded as asking for an object (the next character) that does not exist.

The exception code associated with an existence error is

```
existence_error(Goal, ArgNo, ObjectType, Culprit, Message)
```

ArgNo index of argument of *Goal* where *Culprit* appears

ObjectType expected type of non-existent object

Culprit name for the non-existent object

Message the constant `0` or `''`, or

some additional information provided by the operating system or other support system indicating why *Culprit* is thought not to exist.

For example, '`see('../brother/niece')`' might raise the exception

```
existence_error(see('../brother/niece'),
        1, file, '/usr/stella/parent/brother/niece',
        errno(20))
```

where the *Message* encodes the system error `ENOTDIR` (some component of the path is not a directory).

As a general rule, if *Culprit* was provided in the goal as some sort of context-sensitive name, the Prolog system will try to resolve it to an absolute name, as shown here, so that you can see whether the problem is just that the name was resolved in the wrong context.

8.19.4.7 Permission Errors

A permission error occurs when an operation is attempted that is among the kinds of operation that the system is in general capable of performing, and among the kinds that you are in general allowed to request, but this particular time it isn't permitted. Usually, the reason for a permission error is that the *owner* of one of the objects has requested that the object be protected.

An example of this inside Prolog is attempting to change a predicate that has not been declared `:-dynamic`.

File system protection is the main cause of such errors.

The exception code associated with a permission error is

```
permission_error(Goal, Operation, ObjectType, Culprit, Message)
```

Operation operation attempted; *Operation* exists but is not permitted with *Culprit*.

Some *Operation*s recognized by the system include:

- `0` — no operation specified
- `'find absolute path of'`
- `'get the time stamp of'`
- `'set prompt on'`
- `'use close(filename) on'`
- `abolish`
- `change`
- `check_advice`
- `clauses`access clauses of

- `close`
- `create`
- `export`
- `flush`
- `load`
- `nospy`
- `nocheck_advice`
- `open`
- `position`
- `read`
- `redefine`
- `save`
- `spy`
- `write`

ObjectType
: *Culprit*'s type.

Culprit
: name of protected object.

Message
: provides such operating-system-specific additional information as may be available. A message of `0` or `''` provides no further information.

8.19.4.8 Context Errors

A context error occurs when a goal or declaration appears in the wrong place. There may or may not be anything wrong with the goal or declaration as such; the point is that it is out of place. Calling `multifile/1` as a goal is a context error, as is having `:-module/2` anywhere but as the first term in a source file.

The exception code associated with a context error is

```
context_error(Goal, ContextType, CommandType)
```

ContextType
: the context in which the command was attempted.

  Some *ContextType*s recognized by the system include:

  - `'pseudo-file ''user'''` — for pseudo-file `'user'`
  - `if` — inside an if
  - `bof` — at beginning of file
  - `bom` — at beginning of module
  - `query` — in query
  - `before` — before

- `'after clauses'` — after clauses
- `'not multifile and defined'` — for defined, non-multifile procedure
- `'static multifile'` — for static multifile procedure.
- `language(`*L*`)` — for language *L*.
- `file_load` — during load of file(s).
- `started` — started up
- `notoplevel` — when no top level

CommandType
: the type of command that was attempted.

  Some *CommandType*s recognized by the system include:

  - `0` — no commandtype specified
  - `cut`
  - `clause`
  - `declaration`
  - `'meta_predicate declaration'`
  - `use_module`
  - `'multifile assert'`
  - `'module declaration'`
  - `'dynamic declaration'`
  - `meta_predicate(`*M*`)` — meta_predicate declaration for *M*
  - `argspec(`*A*`)` — Invalid argument specification *A*
  - `foreign_file(`*File*`)` — `foreign_file/2` declaration for *File*
  - `foreign(`*F*`)` — `foreign/3` declaration for *F*
  - `(initialization)` — initialization hook
  - `abort` — call to `abort/0`

8.19.4.9 Consistency Errors

A consistency error occurs when two otherwise valid values or operations have been specified that are inconsistent with each other. For example, if two modules each import the same predicate from the other, that is a consistency error.

The exception code associated with a consistency error is

```
consistency_error(Goal, Culprit1, Culprit2, Message)
```

Culprit1 One of the conflicting values/operations.

Culprit2 The other conflicting value/operation..

Message Additional information, or `0`, or `''`.

8.19.4.10 Syntax Errors

A syntax error occurs when data are read from some external source but have an improper format or cannot be processed for some other reason. This category mainly applies to `read/1` and its variants.

The exception code associated with a syntax error is

```
syntax_error(Goal, Position, Message, Left, Right)
```

where *Goal* is the goal in question, *Position* identifies the position in the stream where reading started, and *Message* describes the error. Left and right are lists of tokens before and after the error, respectively.

Note that the *Position* is where reading started, not where the error *is*.

`read/1` does two things. First, it reads a sequence of characters from the current input stream up to and including a clause terminator, or the end of file marker, whichever comes first. Then it attempts to parse the sequence of characters as a Prolog term. If the parse is unsuccessful, a syntax error occurs. Thus, in the case of syntax errors, `read/1` disobeys the normal rule that predicates should detect and report errors before they perform any side-effects, because the side-effect of reading the characters has been done.

A syntax error does not necessarily cause an exception to be raised. The behavior can be controlled via a Prolog flag as follows:

`prolog_flag(syntax_errors, quiet)`
: When a syntax error is detected, nothing is printed, and `read/1` just quietly fails.

`prolog_flag(syntax_errors, dec10)`
: This provides compatibility with DEC-10 Prolog and earlier versions of Quintus Prolog: when a syntax error is detected, a syntax error message is printed on `user_error`, and the `read` is repeated. This is the default for the sake of compatibility with earlier releases.

`prolog_flag(syntax_errors, fail)`
: This provides compatibility with C Prolog. When a syntax error is detected, a syntax error message is printed on `user_error`, and the `read` then fails.

`prolog_flag(syntax_errors, error)`
: When a syntax error is detected, an exception is raised.

8.19.4.11 Resource Errors

A resource error occurs when some resource runs out. For example, you can run out of virtual memory, or you can exceed the operating system limit on the number of simultaneously open files.

Often a resource error arises because of a programming mistake: for example, you may exceed the maximum number of open files because your program doesn't close files when it has finished with them. Or, you may run out of virtual memory because you have a non-terminating recursion in your program.

The exception code for a resource error is

```
resource_error(Goal, Resource, Message)
```

Goal A copy of the goal, or `0` if no goal was responsible; for example there is no particular goal to blame if you run out of virtual memory.

Resource identifies the resource that was exhausted.

Some *Resource*s recognized by the system include:

- `0` — No resource specified
- `memory` — out of memory
- `'too many open files'`

Message an operating-system-specific message. Usually it will be `errno(ErrNo)`.

8.19.4.12 System Errors

System errors are problems that the operating system notices (or causes). Note that many of the exception indications returned by the operating system (such as "file does not exist") are mapped to Prolog exceptions; it is only really unexpected things that show up as system errors.

The exception code for a system error is

```
system_error(Message)
```

where *Message* is not further specified.

8.19.5 An Example

Suppose you want a routine that is given a filename and a prompt string. This routine is to open the file if it can; otherwise it is to prompt the user for a replacement name. If the user enters an empty name, it is to fail. Otherwise, it is to keep asking the user for a name until something works, and then it is to return the stream that was opened. (There is no need to return the file name that was finally used. We can get it from the stream.)

```
:- use_module(library(prompt), [
        prompted_line/2
   ]).

open_output(FileName, Prompt, Stream) :-
        on_exception(Error,
            open(FileName, write, Stream),
            (   file_error(Error) ->
                print_message(warning, Error),
                retry_open_output(Prompt, Stream)
            ;   raise_exception(Error)
            )).

file_error(domain_error(open(_,_,_), 1, _, _, _)).
file_error(existence_error(open(_,_,_), 1, _, _, _)).
file_error(permission_error(open(_,_,_), _, _, _, _)).

retry_open_output(Prompt, Stream) :-
        prompted_line(Prompt, Chars),
        atom_chars(FileName, Chars),
        FileName \== '',
        open_output(FileName, Prompt, Stream).
```

What this example does *not* catch is as interesting as what it does. All instantiation errors, type errors, context errors, and range errors are re-raised, as they represent errors in the program.

As the previous example shows, you generally do not want to catch *all* exceptions that a particular goal might raise.

8.19.6 Exceptions and Critical Regions

The point of critical regions in your code is that sometimes you have data-base updates or other operations that, once started, must be completed in order to avoid an inconsistent state. In particular, such operations should not be interrupted by a `^C` from the keyboard.

In releases of Quintus Prolog prior to release 3.0, `library(critical)` was provided to allow critical regions to be specified using the predicates `begin_critical/0` and `end_critical/0`. These predicates are still provided, but they should be regarded as obsolete since they do not interact well with exception handling. An exception occurring in between the `begin_critical` and the `end_critical` could cause two problems:

1. the Prolog database could be left in an inconsistent state, and
2. the critical region would never be exited, so interrupts would be left disabled indefinitely.

To avoid these problems, you should use the new predicates

```
critical(Goal)
critical_on_exception(ErrorCode, Goal, Handler)
```

which `library(critical)` now provides.

`critical/1` runs the specified goal inside a critical region. It solves (2) by catching any exceptions that are raised and taking care to close the critical region before re-raising the exception.

`critical_on_exception/3` allows you to solve (1) by specifying appropriate clean-up actions in *Handler*. If an exception occurs during *Goal*, and the exception code unifies with *ErrorCode*, `critical_on_exception/3` acts as if you had written

```
critical(Handler)
```

instead. That is, the *Handler* will still be inside the critical region, and only the first solution it returns will be taken.

These forms also have the effect of committing to the first solution of *Goal*. Since the point of a critical region is to ensure that some operation with side-effects is completed, *Goal* should be determinate anyway, so this should be no problem.

8.19.7 Summary of Predicates and Functions

- `critical/1`
- `critical_on_exception/3`
- `on_exception/3`
- `raise_exception/3`
- `QP_error_message()`
- `QP_exception_term()`
- `QP_perror()`
- `print_message/2`

8.19.8 Summary of Relevant Libraries

- `critical`

8.20 Messages

8.20.1 Overview

Prolog responds to many situations by displaying messages. These messages fall into five categories: error, warning, informational, help and silent. Here are examples of a messages of each category printed without customization:

Error message: (1)

```
| ?- X is apples + oranges.
! Type error in argument 2 of is/2
! number expected, but apples found
! goal:  _2277 is apples+oranges
```

Informational message: (2)

```
| ?- debug.
% The debugger will first leap -- showing spypoints (debug)
```

Warning message: (3)

```
| ?- spy g.
* There are no predicates with the name g in module user
```

Help messages (response to a user's request for information)**:** (4)

```
| ?- version.
Quintus Prolog Release 3.5 (Sun 4, SunOS 5.5)
```

Silent messages (nothing is printed for these message, but the users can define a message hook to catch these messages)**:** (5)

There is a "silent" message every time a top level goal is executed.

Note that each type of message has a characteristic prefix:

'!'	`error`
'%'	`informational`
'*'	`warning`
(none)	`help`
(none)	`silent`

The display of messages such as these can be customized in various ways, including translation into other languages to facilitate internationalization of applications, adding new types of messages, most notably exception messages, changing the text or appearance of the messages as displayed.

Furthermore it is possible to extend the programming environment by specifying goals to be called when specified events occur. These goals would override the straightforward display of message text. A major type of application here is graphical user interfaces.

Finally, you can control how the system asks for and interprets responses from users.

8.20.2 Implementation: Term-Based Messages

The message facility design is based on transforming *message terms* to lists of *format commands* according to *definite clause grammars*. The format commands are processed by `print_message_lines/3` to become the message text. For information about `format/2`, see Section 8.7.6.4 [ref-iou-cou-fou], page 223 in the input/output section of this manual. Definite clause grammars are described in Section 8.16 [ref-gru], page 298.

This design has the advantage of being able to utilize a relatively small list of message terms, each of which generates a set of related messages, and it gives the translator or other user explicit control over the order in which variables are referred to in the translated/customized message.

In release 3, all output from Prolog is represented internally as terms, called message terms. These terms consist of a *functor* that names the message, and *arguments* that give information about the particulars. For example, these *message terms* underlie the *messages* given in the overview:

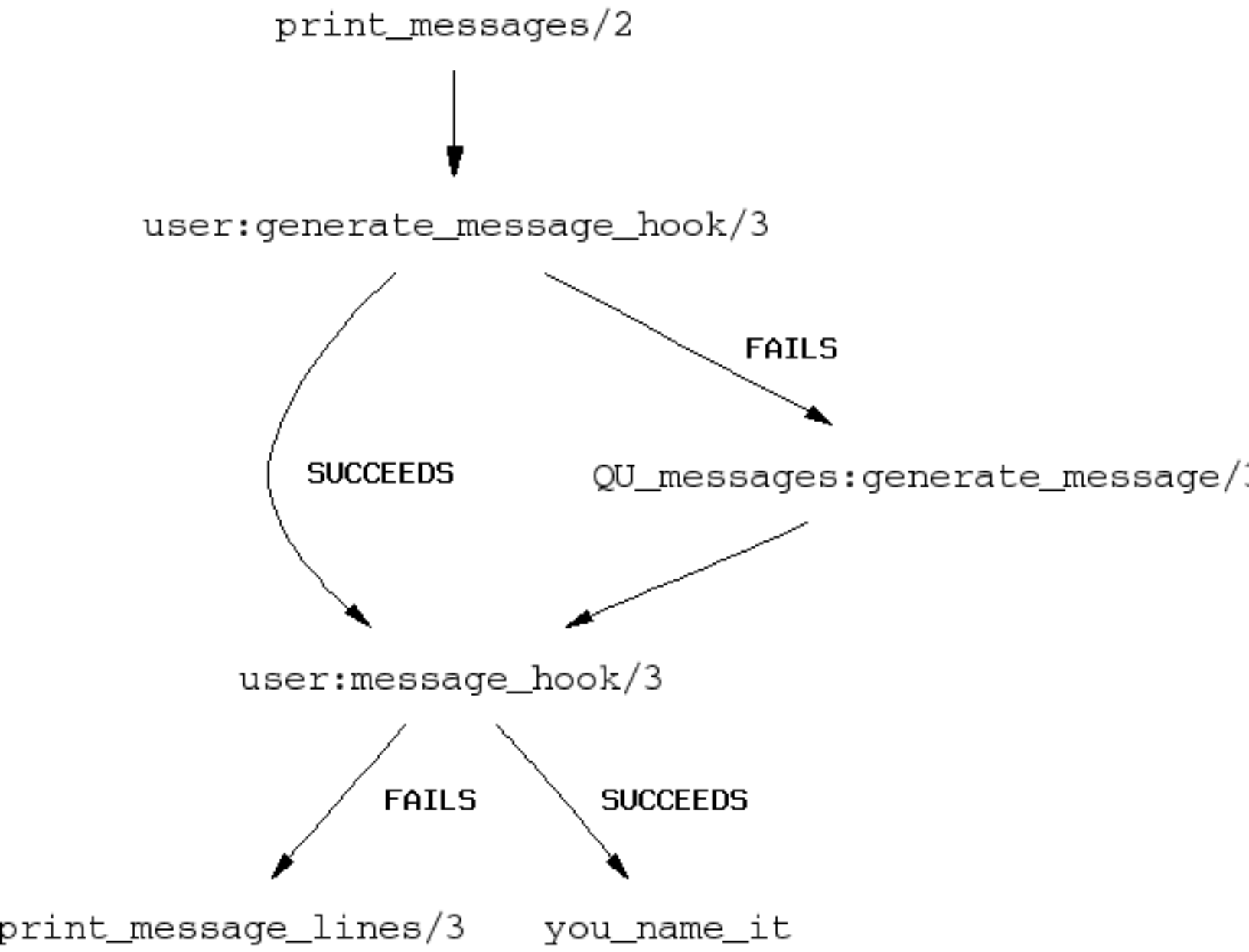
```
type_error(is(_, number, apples + oranges), 2, number, apples)      (1)

debug_message(leap)                                                 (2)

advice(no_preds, '', predicate_family(user:f))                      (3)

version(3.5, 'Sun-4, SunOS 5.5')                                    (4)
```

The *message generator* is defined in the file:

```
messages(language('QU_messages.pl'))
```

It associates such message terms with the corresponding messages. This is accomplished by a series of default clauses for `generate_message/3` within the module. This section will explain and exemplify various ways available to customize these messages.

In addition to the message generator, the message facility consists of a set of built-in predicates & hooks. Once an event is correlated with a call to `print_message/2`, either by Quintus Prolog or by user code, these procedures can be used as shown in the following figure to customize system messages, or use the underlying message terms as triggers for other events.



Message Facility Control Flow

The heavy arrows in the diagram trace the default path of a message term through the message generator.

To print a message, call `print_message/2`. This may be done explicitly in your code. Often, however, it is called by the system. For example, an unhandled exception message *E* will always be captured at the Prolog top level, where it calls `print_message(error, E)`.

Prolog first tries calling user:`generate_message_hook/3` to transform a message term into textual form. If that fails, it then tries `'QU_messages':generate_message/3` instead. (If that also fails the message term will simply be written out as if by `write/1`.) There is one exception to this rule. Messages that fall into the `silent` category do not go through `generate_message_hook/3`.

Once the text of the message, represented as a list of format commands, is decided, Prolog calls user:`message_hook/3`. If this succeeds, nothing else is done; otherwise, `print_message_lines/3` is called to print the message. `silent` messages just go through `message_hook/3` but even if it does not succeed, it is not passed to `print_message_lines/3`.

8.20.3 Examples of Using the Message Facility

In the Overview we discussed several likely ways this facility might be used. The ways to accomplish such goals are discussed in the following sections. To sum up, here are some motivations for customizing messages, with an indication of which procedures should be used:

1. Add a few new types of messages to the ones supplied; for example, a new exception type. (Define user:`generate_message_hook/3`)
2. Change the text of a few existing messages. (Define user:`generate_message_hook/3`)
3. Customize default ways of printing existing message text, e.g. changing the default prefixes; (use `message_hook/3` to override the defaults given by `print_message_lines/3`).
4. "you_name_it" as in Figure Section 8.2.2.2 [ref-sem-typ-rpr], page 181.

   You can specify an action to replace or augment the default action by defining clauses for user:`message_hook/3`. Examples:

   - For `warning` messages, blink a light.
   - In a graphical user interface, let `warning` messages cause a window of a certain design to pop up and display the message.
   - Turn off all `informational` messages.
   - Send all error messages to a certain file.
5. Control how the system asks for and interprets responses from users. (`query_hook/6`)
6. Translate all the existing messages, with or without customizations, to another language. This includes queries and user responses. (redefine `generate_message/3`; see Section 8.20.4 [ref-msg-ime], page 332)

8.20.3.1 Adding messages

To add new messages, define clauses for `generate_message_hook/3`. For instance, to create a new message term indicating that the world is round, one could choose the message term `round_world` and provide the definition:

```
generate_message_hook(round_world) --> ['The world is round.'-[],nl]
```

Please note: The atom `nl` is used for breaking the message into lines. Using the format specification '`~n`' (new-line) is strongly discouraged, since applications typically require explicit control over new-lines.

Though this was not obvious from the `round_world` example, message terms parse to a list of *Control-Args* pairs, where *Control* is a format specification string, and *Args* is the list of arguments for the given Control string. To illustrate, if we wanted to say that the world was flat and wet, we could write

```
user:generate_message_hook(world(X,Y))-->
                        ['The world is ~w and ~w.'-[X,Y],nl].

?- print_message(help,world(flat,wet)).
```

Here is an example of how one might implement a new exception, `bad_font`, when the File associated with that font doesn't exist.

```
check_font(Font,File):-
    (   file_exists(File)->true
    ;   raise_exception(bad_font(Font, File))
    ).

generate_message_hook(bad_font(Font,File)) -->
    ['Can''t find the file ~w corresponding to font ~w'-
        [File, Font],nl]).
```

Instead of using the message facility, you could use an existing error message in this way:

```
check_font1(Font,File):-
    (   file_exists(File)->true
    ;   raise_exception(existence_error(check_font1(Font,File),2,
                                                    file, File,0))
    ).
```

8.20.3.2 Changing message text

Similarly, the text of an existing message may be changed by defining `generate_message_hook/3`. For example, the following definition will change the text of a default message (see `messages(language('QU_messages.pl'))` for original text).

```
user:generate_message_hook(no_source(Name, Arity))-->
              ['~q/~d has no source file' - [Name, Arity], nl].
```

8.20.3.3 Intercepting the printing of a message

After a message is parsed, but before the message is printed out, `print_message/2` calls

```
user:message_hook(+Message,+Severity,+Lines)
```

where Lines is of the form [*Line1*, *Line2*, . . .], where each *Linei* is of the form [*Control_1-Args_1*,*Control_2-Args_2*, . . .].

In our example,

```
message_hook(world(flat,wet),error,
    [['The world is ~w and ~w.'-[flat,wet]]])
```

would be called.

If the call to user:`message_hook/3` succeeds, `print_message/2` succeeds without further processing. Otherwise, the built-in message display is used.

An example of using a hook to redirect output can be seen in the reference page for `print_message_lines/3`.

It is often useful for a message hook to execute some code for its effects, and then fail. This allows other message hooks a chance to run, and allows the printing of the message to proceed normally.

For example, we might want to ring a bell when printing an error message, and print a count of error messages seen so far. This could be done as follows:

```
:- dynamic error_count/1.

user:message_hook(_, error, _) :-
        put(7),                          % ASCII 7 rings bell
        (   error_count(N) -> true
        ;   N = 1
        ),
        format('! Error ~w:~n', [N]),
        N1 is N+1,
        retractall(error_count(_)),
        assert(error_count(N1)),
        fail.
```

8.20.3.4 Interaction

Prolog's default ways of eliciting keyboard input are enumerated in the clauses for `query_abbreviation/3` in `messages(language('QU_messages.pl'))`. These clauses specify valid abbreviations for a given key word. For example,

```
query_abbreviation(yes_or_no,'(y/n)',[yes-"yY",no-"nN"]).
```

A French translator might decide that the letters '`O`' and '`o`' are reasonable abbreviations for '`oui`' (yes), and therefore write '`yes-"oO"`'. See the reference page for more information on `query_abbreviation/3`.

The query hook provides a means of overriding the default interaction. Whenever Prolog attempts to solicit input from the user, it first looks to see if the application wants to take control of the query by calling user:`query_hook/6` (see the above figure). The various queries are listed in the manual page for `query_hook/6`. For example, if Prolog is looking for a yes-no response, as in the toplevel, this request for input can be captured as follows, where `my_yes_no/2` binds Answer to `yes` or `no`:

```
query_hook(toplevel,_,_,_,_,Answer):-
    my_yes_no('Done?',Answer).
```

8.20.4 Internationalization of Quintus Prolog messages

By default, `generate_message/3` sends the message term through the English message generator, `messages('english/QU_messages')`. The definite clause grammar (DCG) rules in this file transform Prolog message terms into English messages. In addition, the message generator is designed to allow internationalization of the messages printed by the Quintus Prolog development system. You can translate the output of the rules in the message generator into any natural language. To do this, you would translate part of the right hand side of each rule into the target language, e.g. French, to produce `messages('french/QU_messages')`, which generates French messages from the same message terms. This section explains how to go about providing a translation.

The default '`QU_messages.pl`' is installed as:

quintus-directory`/generic/qplib3.5/embed/english/QU_messages.pl`

To have all messages printed in another language, the basic steps are as follows

1. Take a copy of '`QU_messages.pl`' and translate all the messages.
2. Test the translated '`QU_messages.pl`' and then install it in the Quintus Prolog directory hierarchy.
3. Install or re-install Quintus Prolog to get a version that uses the translated messages.

8.20.4.1 Translating the Messages

Each grammar rule in '`QU_messages.pl`' defines the generation of a message term from an internal form. For example, in (A) the text that needs to be translated is the quoted text inside the list '`[]`' brackets: '`Type error`'. The rest of the rule does not need to be changed.

```
generate_message(type_error(Goal,ArgNo,TypeName,Culprit)) -->        (A)
    ['Type error'-[]],
    head(Goal,ArgNo),
    type(TypeName,Culprit),
    goal(Goal).
```

The general form of text components in a list is (B), where *control-string* and *arg-list* are valid for a call such as (C) to the built-in predicate `format/2` as exemplified in (D).

```
control-string-arg list                                              (B)
```

```
| ?- format(control-string, arg-list).                               (C)
```

```
| ?- format('Type error', []).                                       (D)
Type error
yes
```

In example (E), when the message is printed, the List will be inserted in place of the ~q. The ~q means that the List will be printed as if by the built-in predicate `writeq/1`. See the documentation of `format/[2,3]` for full details of what a control-string can do.

```
typename(one_of(List)) --> ['a member of the set ~q'-[List]].        (E)
```

In addition to text components of the form (F), it is also possible to have text components of the form (G), which cause a newline to be output.

```
control-string-arg list                                              (F)
```

```
nl                                                                   (G)
```

The format option '`~n`' should not be used in control strings: any required newlines should be specified with the `nl` text components. For example, the list in (H) contains two text components; in general, a number of text components can be collected into a single list like this, or they can appear in separate lists, as in (I):

```
advice(invalid_argument,Type,X) -->                                  (H)
        ['Invalid argument to ~w, ~q was ignored.'-[Type,X],nl].

advice(invalid_argument,Type,X) -->                                  (I)
    ['Invalid argument to ~w, ~q was ignored.'-[Type,X]], [nl].
```

All (complete) messages must end with an `nl`.

In addition to translating messages, it is also possible to change the characters that Prolog will accept when it requires input from the user. This is done by modifying the definition of `query_abbreviation/2` at the end of '`QU_messages.pl`'. For example, (J) means that when the Prolog system wants a yes-or-no answer it expects the first character typed to be a '`y`' or a '`n`' and the case does not matter. Since "yes" in French is "oui" a French translator might change this to (K).

```
query_abbreviation(yes_or_no,[yes-"yY",no-"nN"]).          (J)

query_abbreviation(yes_or_no,[yes-"oO",no-"nN"]).          (K)
```

8.20.4.2 Testing and Installing the Translated Messages

A translated version of '`QU_messages.pl`' can be tested by compiling it into an ordinary Prolog and seeing how the messages printed by the system look. That is:

```
% prolog
| ?- compile('QU_messages').
```

Once the new '`QU_messages.pl`' has been tested, it can be installed in

quintus-directory`/generic/qplib3.5/embed/`*language*`/QU_messages.pl`

The file should be compiled to QOF in that directory:

```
% cd quintus-directory/generic/qplib3.5/embed/language
% qpc -c QU_messages.pl
```

8.20.4.3 Building a Version of Prolog using the Translated Messages

First set the `QUINTUS_LANGUAGE` environment variable to *language*. Then re-install Prolog, following the normal installation instructions.

When using `qld` to build stand-alone programs users will also need to have the `QUINTUS_LANGUAGE` environment variable set. Therefore, all users who wish to use the translated messages should arrange to set this variable in their '`.login`' or '`.cshrc`' files.

`qpc` is itself a Prolog runtime system. Therefore, if it has been installed with English messages it will need to be re-built with messages in the preferred language. To do this, the `qpc` in the directory (B) should be deleted before re-installation.

quintus-directory`/bin`*version/platform* (B)

In multilingual environments it may not always be convenient to have different versions of executable files for the language spoken by each user. An alternative is for users to put (C) in their '`prolog.ini`' files. Provided that they have set the `QUINTUS_LANGUAGE` environment variable, this will ensure that the messages in their chosen language are loaded whenever they start up an executable that was created with messages in some other language.

```
:- use_module(messages(language('QU_messages'))).          (C)
```

8.20.4.4 Using Kanji characters

To permit the use of Kanji characters in atoms, variable names, predicate names, messages and comments you need to set the environment variable `QUINTUS_KANJI_FLAG` to `true` before starting Prolog.

When the Kanji flag is set, an 8-bit character codes (that is, a character code between 128 and 255) is assumed to be part of a multi-byte sequence representing a Kanji character. Such characters are treated, for the purposes of Prolog's syntax, as if they were lowercase alphabetic characters. The Kanji flag does not affect character I/O, or the conversion between atoms and lists of character codes using `name/2` or `atom_chars/2`. In the case of I/O, two (or more) character codes will need to be read or written for every Kanji character. Similarly, an atom made up of Kanji characters will be transformed by `name/2` or `atom_chars/2` into a list of twice (or more than twice) as many 8-bit character codes.

JIS (Japanese Industry Standard) is the Japanese analogue to ASCII. This includes two standards: JIS-X0201, which defines half-size Katakana characters, represented in one byte and JIS-X0208 (called "JIS" code) that defines 6349 Kanji and 453 non-Kanji characters stored in two bytes, using the lower seven bits of each byte.

SHIFT-JIS
Is JIS code shifted so that the high bit is used (so it doesn't interfere with some MSDOS control codes and special characters).

JLE 1.0 EUC
Is Sun's Japanese Language Environment. For each codeset this defines exactly one possible character set:

codeset		**byte(s) representation**
0:	ASCII	0xxxxxxx
1:	JIS-X0208	1xxxxxxx 1xxxxxxx
2:	JIS-X0201	10001110 1xxxxxxx
3:	Gaiji	10001111 1xxxxxxx 1xxxxxxx

Codeset 1 is JIS code with the hit bits set; Gaiji is a user-defined character set.

DEC This is a standard used by Digital. All I know about this is that character codes 0-127 indicate a one byte ASCII character, and the codes 128-255 indicates a two byte Kanji character.

8.20.5 Summary of Predicates

- `generate_message/3`
- `query_hook/6`
- `user:query_abbreviation/3`
- `print_message/2`
- `user:message_hook/3`

- `print_message_lines/3`
- `user:generate_message_hook/3`

9 Creating Executables

9.1 Stand-Alone Programs & Runtime Systems

9.1.1 Basic Concepts

Traditionally, the way to develop Prolog programs has been to compile all the sources into memory and then to create saved-states for stable portions of the program. The use of saved-states avoids the need to recompile unchanged code each time that a testing/debugging session is started. It can also be used as a way of packaging a completed application for later re-use or for use by others.

This section explains the use of tools that provide an alternative approach to program development: compiling and linking your sources to build a *stand-alone program*. This is very much like the normal way of developing programs written in languages such as C.

9.1.1.1 Terminology

The following terminology is used both here and in Section 9.2 [sap-rge], page 355.

Kernel
: The code implementing Prolog's built-in predicates, plus support code such as memory management needed for running Prolog.

Runtime Kernel
: Provides only the core set of built-in predicates.

Extended Runtime Kernel
: An add-on product. It provides the core set of built-in predicates, and in addition allows the compiler and dynamic foreign language interface to be used in runtime systems.

Development Kernel
: Provides the facilities of the Runtime Kernel, plus additional built-in predicates, the compiler, the debugger, and so on.

runtime system
: A Prolog application that has been linked with the Runtime Kernel; intended for delivery to end-users.

stand-alone program
: A Prolog program that has been linked with the Development Kernel; an extended version of the Development System.

static linking
: Linking a Prolog program to either the Development Kernel or the Runtime Kernel.

9.1.1.2 Shared Libraries and Delivering Executables

WARNING: An application may depend upon *shared libraries*, but they are separate from the executable. So it is necessary to take explicit steps to ensure that they will in fact be present when the application is run on the target machine. They must be explicitly linked in to the deliverable as described in Section 9.2.5 [sap-rge-sos], page 359.

9.1.1.3 Stand-Alone Programs

A stand-alone Prolog program is a single, standard, executable file that can be considered to be an extended version of the Development System, with your Prolog and foreign code being pre-loaded into it. This approach has several advantages as compared with the saved-state approach:

1. Since a stand-alone program is a standard executable file, you can use standard source-level debugging tools such as `gdb(1)` to debug your foreign code at the source level.
2. Standard tools such as `make(1)` can be easily used to construct your program, ensuring that everything is up-to-date. This is particularly useful if you are using a lot of foreign code.
3. Startup is faster.
4. Your code (except for dynamic predicates) will be loaded into the text segment, so that it will be shared if the program is to be run by more than one process at a time on a single machine.
5. Different application programs can share compiled files. The format of compiled files, "Quintus Object Format" or QOF, is portable across a number of different hardware types and operating systems.

The disadvantage of linking, in comparison with using a saved state, is that the stand-alone program will require more disk space than a saved state, since it contains the Development Kernel and the user's foreign code.

9.1.1.4 Runtime Systems

Generating a stand-alone program is very much like generating a runtime system with the Quintus Prolog Runtime Generator. Exactly the same tools are used in each case, and the use of these tools is described in this section.

The purpose of the Runtime Generator is to provide a convenient and cost-effective way to distribute application programs to end-users. A runtime system differs from a stand-alone program in the following ways:

- A runtime system can be moved to another machine without requiring an authorization code for that machine.

- A runtime system requires that its entry point be specified by providing a definition for `runtime_entry/1`. In contrast, a stand-alone program starts up at the normal Prolog top level (unless initializations are used; see the reference page for `initialization/1`).
- Runtime systems do not contain development oriented components of the Development System such as the compiler, the debugger, the editor interface, the on-line help system, the top level, and the foreign language interface. (Runtime systems can, of course, contain foreign code, but they cannot call `load_foreign_executable/1` or `load_foreign_files/2` to load additional foreign code when they are run. However, they can load QOF files that load foreign code. See Section 9.1.6.6 [sap-srs-eci-ctc], page 353)

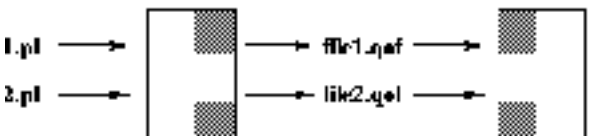
For information on the Runtime Generator see Section 9.2 [sap-rge], page 355.

9.1.1.5 Compiling and Linking

There are two major tools needed to convert a set of source files into a stand-alone program or a runtime system:

1. the compiler, `qpc`, which translates Prolog files into QOF files. This is described in Section 9.1.2 [sap-srs-qpc], page 341.
2. the link editor, `qld`, which links one or more QOF files together and builds an executable file. Often the user will not need to call `qld` directly because `qpc` will do this automatically. The linker is described in Section 9.1.3 [sap-srs-qld], page 343.

It is also possible to build QOF files by saving them directly from a Prolog development system session, as described in Section 8.5 [ref-sls], page 192. QOF files saved in this way can then be used directly to build a stand-alone program with `qld`.


1.pl
2.pl
file1.qof
file2.qof

Major steps in creating a stand-alone program

`qpc` allows *independent compilation* of the files that make up an application program. The dotted box in the above figure illustrates this functionality.

The use of `qpc` and `qld` correspond to the use of the C compiler and linker as follows:

	Quintus Prolog	**C**
sources:	'`a.pl`'	'`a.c`'
compiler:	`qpc`	compiler
object files:	'`a.qof`'	e.g. '`a.o`'
linker:	`qld`	linker
executable:	'`a.out`'	e.g. '`a.out`'

9.1.1.6 The Runtime Kernel vs. Development Kernel

There is a command-line option, '`-D`', that can be given to either `qpc` or `qld` to indicate that a stand-alone program rather than a runtime system is desired. This flag determines whether your program is to be linked with the Development Kernel or the Runtime Kernel. Each of these kernels consists of a QOF file and an object file, which together supply all the necessary support code for running Prolog. This support code includes memory management and the built-in predicates. The Development Kernel additionally contains support for Prolog development, such as the compiler and the debugger.

9.1.2 Invoking `qpc`, the Prolog-to-QOF Compiler

There are two main ways to invoke `qpc`:

```
% qpc -c P1.pl,...,Pn.pl                                    (A)

% qpc [-D] [-o output-file] P1.pl,...,Pn.pl                 (B)
```

(A). Invoking `qpc` with the '`-c`' option, means "compile to QOF and stop"; it simply produces a QOF file for each source file, as shown in the above figure.

(B). Invoking `qpc` without specifying '`-c`' compiles all the sources to QOF and then calls `qld` to build an executable file corresponding to those sources. '`-D`' tells `qld` that the program is to be linked with the Development Kernel rather than the Runtime Kernel. '`-o`' specifies a name for the executable file to be built by `qld`. Defaults to '`a.out`'. The '`-D`' and '`-o` *`output-file`*' options are passed on to `qld` if they were specified in the `qpc` command line.

> **Please note:** '`.pl`' extensions may be omitted in the `qpc` command line (provided that there is not another file with the same name and no extension.) Also, the Prolog files need not have '`.pl`' extensions. If a Prolog file does not

have one, the name of the corresponding object format file is simply the name of the Prolog source file extended with '`.qof`'. Otherwise, the name of the corresponding object format file is the name of the Prolog source file with the '`.pl`' extension replaced by '`.qof`'. Source files may be specified as absolute or relative filenames; each QOF file goes in the same directory as its source.

Further options allow you to run `qpc` in a verbose mode, to specify initialization files or add-ons, to customize the library search path, or make certain predicates invisible to the debugger.

For a summary of all the options to `qpc`, see Section 20.1.6 [too-too-qpc], page 1489.

9.1.3 Invoking `qld`, the QOF Link eDitor

Components of the qld Program

This diagram expands the `qld` "black box" in the earlier figure. '`a.qof`' is a temporary QOF file, and '`a.o`' is a temporary object file.

9.1.3.1 Implicit invocation via `qpc`

When `qpc` is called without the '`-c`' command-line option, as in invocation (B), above, it first compiles all the specified Prolog files into QOF and then invokes `qld` as follows:

```
% qld [-W] [-D] [-v] [-o output-file] -d qof-files object-files
```

When `qld` is called by `qpc`, it is called in the verbose mode ('`-v`'), with the following additional options:

'`-W`' (Windows only) determines that `qld` should build a "windowed" executable, which runs in its own window, as opposed to a console-based one, which runs in a command prompt window. The properties of the Windows component of an executable built with '`-W`' can be controlled with resource files and the environment variable `CONSOLE`; see Section 20.1.4 [too-too-qld], page 1481.

'`-D`' determines that `qld` is to be linked with the Development Kernel rather than the Runtime Kernel. In either case two kernel files, one QOF and one object file, must be linked in addition to the application files. (See the above figure.)

'`-o`' specifies the name of the executable file that is to be the final product; defaults to '`a.out`'.

'`-d`' tells `qld` to link in any additional files on which any of the specified QOF files depends. See Section 9.1.4 [sap-srs-dep], page 345 for more details on file dependencies.

qof-files is a list of QOF files, '`P1.qof`' ,. . . , '`Pn.qof`', the output of the `qpc` call.

object-files is a list of object files built by compiling your foreign-language files with the appropriate compiler(s). These may include foreign libraries (e.g. `qpc -lX11`).

In addition to the above arguments, `qpc` also passes to `qld` appropriate '`-f`', '`-F`' and '`-L`' options if any non-default file search paths or library directories have been specified. The '`-f`', '`-F`' and '`-L`' options have the same meaning for `qld` as they do for `qpc`; they are only meaningful when the '`-d`' option is specified, and they tell `qld` where to look for file specifications. See Section 9.1.5 [sap-srs-fsp], page 348 for information on how `qld` makes use of file search paths and library directories.

9.1.3.2 Explicit Invocation

If you wish to call `qld` with other options than those described in Section 9.1.3.1 [sap-srs-qld-iin], page 344, use `qpc -c` and then call `qld` explicitly. The '`-c`' option to `qpc` causes

`qld` to stop after generating an object file, rather than continuing and calling the linker. The object file will be called '`a.o`' (UNIX) or '`a.obj`' (Windows) by default; this can be overridden with the '`-o`' option. Then you can make your own call to the linker; this call must include any needed object-files and libraries.

One reason you might wish to do this is to avoid the use of shared object files and shared libraries in a runtime system that is to be delivered on a different machine. See Section 9.2.5 [sap-rge-sos], page 359 for more information on this and an example.

The steps taken by `qld` — as illustrated in the above figure — are as follows:

1. Link all the specified QOF files together with either the Runtime Kernel or Development Kernel. The result of this phase is a temporary QOF file.
2. "Consolidate" the temporary QOF file into a temporary object file using the subsidiary program `qcon`.
3. UNIX: Call the C compiler to build an executable file. The command, which is echoed to standard output if the '`-v`' option is specified, resembles:

   ```
   % cc [-v] [-o output-file] runtime-directory/qprel.o temp.o
                    object-files runtime-directory/libqp.a
   ```

   The file '*runtime-directory*`/qprel.o`' is the Development Kernel object file. If you are linking to a Runtime Kernel '`qprel.o`' will be replaced by '`qprte.o`' in the above command. '`temp.o`' is the output of Step 2. '`libqp.a`' is the Quintus C library.
4. Windows: Call the C compiler, `cl` to build an executable file. The form of the link command, which is echoed to standard output if the '`-v`' option is specified, resembles:

   ```
   % link [-v] [-o output-file] temp.obj
                runtime-directory/qpeng.lib runtime-directory
   /qprel.lib
                [runtime-directory/libpl.lib] object-files
                runtime-directory/libqp.lib
   ```

   The file '*runtime-directory*`/qprel.lib`' is the Development Kernel object file. If you are linking to a Runtime Kernel '`qprel.lib`' will be replaced by '`qprte.lib`' in the above command. '*temp*`.obj`' is the output of Step 2. '`libqp.lib`' is the Quintus C library.

For a complete summary of all the possible options to `qld`, see Section 20.1 [too-too], page 1475.

9.1.4 Dependencies of QOF files

Each QOF file contains a record of all the files, including library files, upon which it depends, that is, for which its source file contains any of the following load commands:

```
:- compile(Files).
:- ensure_loaded(Files).
:- use_module(Files).
:- use_module(File, ImportList).
:- use_module(Module, File, ImportList).
:- [Files].
:- reconsult(Files).
:- load_foreign_executable(File).
:- load_foreign_files(Files, Libraries).
:- load_files (Files).
:- load_files (Files, Options).
```

Each record is in the form that was used to specify the file in the load command: it may be a relative or absolute filename, or else it may be a file search path specification, such as:

- `library(`*Name*`)`
- `system(`*Name*`)`
- `quintus(`*Name*`)`
- `language(`*Name*`)`
- `mypath(`*Name*`)`

When the QOF file is passed to `qld` and the '`-d`' option is specified, this information will be used to find all the QOF and object files on which this QOF file depends.

Under Windows, a specification of the form:

- `syslib(`*Name*`)`

has a special meaning: `qld` looks up the import library '*Name*`.lib`' in a directory in the `LIB` environment variable.

9.1.4.1 Generating QOF Files and Dependencies

When Prolog files contain embedded commands to compile other files, each Prolog source file is compiled into a separate QOF file with one exception: if a module-file contains a command to load a non-module file, then the non-module-file is compiled directly into the QOF file corresponding to the module-file. That is, there is no separate QOF file for a non-module-file that is loaded into a module unless it is loaded into the default module `user`. Each QOF file is written into the same directory as the corresponding Prolog source file.

Embedded `ensure_loaded/1` and `use_module/[1,2,3]` commands also cause the specified files to be compiled *unless* there is a corresponding QOF file more recent than the corresponding Prolog source file. For example,

```
:- ensure_loaded(file).
```

causes '`file.pl`' to be compiled unless there is a '`file.qof`' more recent than the source. In the case where the QOF file is more recent than the '`.pl`' file, then the file is not compiled again. However, the QOF file's dependencies are checked and recompiled if not up to date.

9.1.4.2 Example

file.pl

```
:- ensure_loaded(library(basics)).
:- ensure_loaded(file1).
:- ensure_loaded(file2).

runtime_entry(start) :- go.
```

file1.pl

```
< some foreign/[2,3] facts >
< some foreign_file/2 facts >

:- load_foreign_files([system(foreign)],[]).
```

```
% qpc file                                                    (A)
```

Given the above files, the command (A) will have these results:

- Compile '`file.pl`' into !sq'file.qof'
- Cause each of the QOF files in (B) to be produced unless it already exists and is more recent than its source:

  ```
  library-directory/basics.qof file1.qof file2.qof            (B)
  ```

- Records the dependency of '`file.qof`' on the three QOF files in (B) and records the dependency of '`file1.qof`' on '`foreign.o`' in '`file1.qof`', so that when `qpc` then calls (C) an executable file is built for the entire program.

  ```
  % qld -d file.qof                                           (C)
  ```

If for some reason you didn't want to use the '`-d`' option to `qld`, you could achieve the same effect as `qpc file` by the following sequence of commands:

```
% qpc -c file file1 file2
% qld file.qof library-directory
/basics.qof file1.qof file2.qof foreign.o
```

Alternatively, these commands would work:

```
% qpc -c file file1 file2
% qld file.qof "library(basics)" file1.qof file2.qof foreign.o
```

Note that moving a QOF file from one directory to another may render its dependencies incorrect, so that the '`-d`' option cannot be used when loading that file. If relative filenames

are used, a set of mutually dependent files *can* safely be moved around the directory hierarchy, or to another machine, provided that they retain the same positions relative to one another. In particular, a set of files that are all in the same directory can safely be moved. Using file search path specifications (see Section 9.1.5 [sap-srs-fsp], page 348 and Section 8.4 [ref-lod], page 189) enables you to create alterable paths.

9.1.4.3 Using the `make(1)` utility

The `make(1)` utility may also be used to ensure that QOF files are up to date. For example, the following lines can be added to a make file to tell `make(1)` how to build a QOF file from a '`.pl`' file.

Makefile

```
#       Quintus Prolog Compiler (qpc) section
.SUFFIXES: .qof .pl

QPC=qpc
QPCFLAGS=

.pl:
${QPC} $(QPCFLAGS) -o $ $<
.pl.qof:
${QPC} $(QPCFLAGS) -cN $<
```

9.1.5 File Search Paths and qld

When a directive such as (A), below, is encountered by `qpc`, it notes that the QOF file being produced has a dependency on '`basics.qof`' in the library. The dependency is not stored as an absolute filename, so that when (B) is called (recall that the '`-d`' option causes `qld` to pull in all the dependencies of the specified QOF files), '`basics.qof`' will be sought wherever the current libraries are located. These need not be in the same place as at compilation time; in particular the QOF file may have been moved to a different machine.

```
:- ensure_loaded(library(basics)).                    (A)
```

```
% qld -d ...                                          (B)
```

The '`-L`' option of `qpc` and `qld` allow prepending library directory definitions to the already existing ones. The '`-f`' and '`-F`' options perform similar functions but are more flexible than '`-L`': '`-f`' appends a file search path definition to the already existing set, while '`-F`' prepends a file search path. A '`-f`' option, as exemplified in (C), corresponds to a `file_search_path/2` call, (D). In such calls, `path` can itself be a file search path, as in (E).

```
-f pathname:path                                    (C)
file_search_path(pathname, path)                    (D)

-f "mypath:library(mypackage)"                      (E)
```

For more detail on these options, see Section 20.1 [too-too], page 1475.

9.1.6 Embedded Commands and Initialization Files

This section discusses some differences that exist between compiling a file into QOF with `qpc` and compiling that file into memory using `compile/1` under the Development System. In certain cases, if an application program was developed interactively using the built-in compiler, some changes may have to be made to the code before using `qpc` to compile it and link it with the Development Kernel or Runtime Kernel.

For example, if a file containing the following is compiled into memory, the embedded command will succeed after writing an '`x`' and a newline to the current output stream.

```
f(x).
f(z).

:- f(X), write(X), nl.
```

Whereas, if the same file is given to `qpc`, a warning will be printed indicating that the embedded command failed. The reason for this is that when `qpc` compiles a Prolog file, it reads clauses from the source file one after the other and compiles them into a QOF file. The clauses for `f/1` are not kept in memory, and the attempt to access them fails.

You do *not* need to read this section if both of the following are true.

1. The only embedded commands that you use in your Prolog files are commands to load other files, that is, commands from the list at the beginning of Section 9.1.4 [sap-srs-dep], page 345.
2. You do not use `term_expansion/2` to transform your source code at compile-time.

9.1.6.1 Compile-time code vs. Runtime code

A Prolog program has up to three types of code in it:

1. code that implements the application;
2. code that helps compile the application, and that is not used during the execution of the application;
3. code that is necessary to both the execution of the application and its compilation.

The first type of code is the normal case and may be referred to as *runtime code*, since it is intended to be executed when the application is run. The second type of code is

called *compile-time code*. Any predicates that are to be called from embedded commands are examples of compile-time code. The other main use of compile-time code is in the definition of `term_expansion/2`, which allows you to specify arbitrary transformations to be done by the compiler as it reads in clauses. See the reference page for more information on `term_expansion/2`.

When using the built-in compiler in the Development System, no distinction has to be made between the three types of code. They can coexist in one file. Before using `qpc` on a program, however, compile-time code must be separated out into its own file (or files). Then, to each file that needs this new file or files at compile time, add the goal (A) near the top of the file. This tells `qpc` to load *NewFile* directly into `qpc` before further compiling the current file. It does *not* include *NewFile* as a runtime dependency of the file. If you need *NewFile* to be loaded at compile time and also at runtime, use the goal (B) instead. This approach will work in the Development System as well as `qpc`.

```
:- load_files(NewFile, [when(compile_time)]).                    (A)
:- load_files(NewFile, [when(both)]).                            (B)
```

Alternatively, you may omit the use of `load_files/2`, instead specifying files to be loaded into `qpc` with the '`-i`' option. In this case, when you want to compile this file into the Development System, remember to first load the file(s) needed at compile time.

It is good programming style to use `initialization/1` for goals to be activated at runtime. Note that predicates called as '`:- Goal.`' need to be available at compile time, whereas predicates called as '`:- initialization(Goal).`' only need to be available at runtime.

9.1.6.2 Initialization Files

`qpc` is implemented as a normal runtime system. Hence it has its own internal Prolog database. All compile-time code must be loaded into this database so that `qpc` can run it.

The '`-i`' command-line option is used to specify an initialization file. See the reference page for `qpc(1)`.

9.1.6.3 Side-Effects in Compile-Time Code

One other way to add clauses to `qpc`'s internal database is to assert them in an embedded command. For example, the sequence:

```
:- asserta(f(x)).
:- f(X), write(X), nl.
```

in a file given to `qpc` will work just as it would if the file were compiled into the Development System.

There are some problems with asserting clauses like this. One problem is that the asserted clauses will not be available at run-time. If the file had been loaded into the Development System, they would be available when the program was run.

Another problem arises if the compilation of one file depends on facts that are expected to be asserted into the database during the compilation of some other file. An approach of this sort may be useful in the Development System, but it is contradictory to the notion of independent compilation (see the first figure), which is one of the important features of `qpc`. This problem is not specific to asserting clauses; it arises with any compile-time side-effects that are intended to affect future compilation.

It is possible to avoid using separate compilation, by always recompiling your entire program every time any part of it is modified. It is still not generally safe to use compile-time side-effects in one file that affect the compilation of other files. This is because the order in which files are compiled is different in `qpc`. When `qpc` finds a command to compile a file, it looks in that file immediately to find out whether it is a module-file and if so what are its exports. But it does not actually compile the file immediately: it puts it on a queue to be compiled when the current file has been finished with. This is in contrast to compilation in the Development System, where embedded `compile/1` commands are processed immediately as they are encountered.

Therefore, it is strongly recommended that side-effects in compile-time code be avoided, or at least restricted so that only the compilation of the current file is affected.

9.1.6.4 Modules and Embedded Commands

Embedded commands are called in the modules in which they are contained. Sometimes this seems strange, since the module is really a property of the program being compiled into QOF, and the embedded command is to be interpreted not with respect to that program, but rather with respect to the internal database of `qpc`.

9.1.6.5 Predicates Treated in a Special Way

While `qpc` is compiling Prolog source into QOF, certain built-in predicates are treated in a special way. Their behavior when used as embedded commands under `qpc` is different from their normal behavior. For example, (A) causes the file '`foo.pl`' to be compiled into '`.qof`' format, not, as you would expect from its normal meaning, into (`qpc`'s) memory.

```
:- compile(foo).                                                    (A)
```

Similarly, if you define (B) in an initialization file, then the command (C) will cause foo to be compiled into QOF format after whatever goals you specified have been called.

```
my_compile(File) :-                                          (B)
    ...{some goals}...,
    compile(File).

:- my_compile(foo).                                          (C)
```

The `load_files/2 when` option can be used to force a file to be loaded into memory at compile-time if so desired.

Note that the change of meaning of `compile/1` etc does not apply during the loading of an initialization file, only while compilation to '`.qof`' format is taking place. Thus, if you put

```
:- my_compile(foo).
```

in your initialization file (after the definition of `my_compile/1`), then this would mean compile '`foo.pl`' into memory.

The predicates following this behavior are:

`compile/1`
: compile files

`consult/1`
: compile files

`load_files/[1,2]`
: compile files

`ensure_loaded/1`
: compile files

`load_foreign_files/2`
: compile links to foreign code

`load_foreign_executable/1`
: compile links to foreign code"

`no_style_check/1`
: disable style checking

`op/3` : declare operator(s) (see Section 9.1.7 [sap-srs-ode], page 353)

`reconsult/1`
: compile files

`style_check/1`
: enable style checking

`use_module/[1,2,3]`
: compile module-files

`./2` : (usually written [*Files*]) compile files

Note that an embedded command of the form

```
:- compile(user).
```

will cause an error message from `qpc`. The same is true for specifying '`user`' in embedded calls to `consult/1` and similar commands, as well as in the command line of `qpc`. The reason for this restriction is to avoid possible confusion; under the Development System, giving '`user`' as the argument to one of these predicates allows you to enter clauses directly from the terminal.

Clauses for the predicates `foreign/[2,3]` and `foreign_file/2` are treated specially by `qpc`. They are always assumed to be compile-time predicates, to be used by a subsequent embedded `load_foreign_executable/1` or `load_foreign_files/2` goal. Therefore they are consulted into `qpc`'s internal database rather than being compiled into QOF.

9.1.6.6 Restriction on Compile-Time Code

Since `qpc` is itself a runtime system, any code to be run at compile-time must obey the same restrictions as for any other runtime system. In particular, foreign code cannot be loaded into `qpc` with `load_foreign_executable/1` or `load_foreign_files/2`. However, you can load QOF files into `qpc`, and if the QOF file has object file dependencies, they will be loaded also. For example, you might compile '`file.pl`' with `qpc` to get '`file.qof`':

file.pl

```
:- load_foreign_file([prog1], [])
:- load_foreign_executable(prog2)
```

file.qof

```
... <dependency on object file prog1> ...
... <dependency on shared object file prog2> ...
```

In this case, '`file.pl`' and '`file.qof`' both depend on the same object files. However, while a runtime system can load '`file.qof`', it cannot load '`file.pl`', because `load_foreign_executable/1` and `load_foreign_files/2` are not available in the runtime kernel.

9.1.7 Operator Declarations

An operator declaration is a call to the predicate `op/3` in an embedded command. See Section 8.1.5 [ref-syn-ops], page 165 for more information about operator declarations.

An operator declaration takes effect when it is encountered and remains in force during compilation of the file and during runtime as in the Development System.

9.1.8 Saved-States and QOF files

Saved-states may be created from within a stand-alone program or a runtime system in the normal way, using `save_program/[1,2]`. A saved-state may be restored, using the built-in predicate `restore/1`, or incrementally loaded using `load_files/[1,2]`. As discussed in Section 8.5 [ref-sls], page 192, saved-states are just QOF files, and there is complete flexibility in how they can be selectively saved and loaded. Saved states, and other selections of predicates and modules saved into QOF files, can also be directly used with `qld` to build a stand-alone program or runtime system.

Note that the `restore/1` command is not as useful to load a saved-state (or any QOF file) into a runtime system as the `load_files/[1,2]` command. While a `restore/1` in a stand-alone program, just as in the Development System, restarts the running executable and then loads the argument QOF file, a runtime system only restarts the executable, and the file, preceded by a '`+l`' flag, is passed to the application program, which may elect to parse the arguments and then load the file. Similarly, if a saved-state is created from a runtime system and then restarted from the command line, the executable will be started, but the options list need to be parsed and the file loaded by the application program (see Section 18.3.153 [mpg-ref-restore], page 1266, and Section 8.3.1 [ref-pro-arg], page 186 or Section 20.1.1 [too-too-prolog], page 1476 for a description of the '`+l`' option).

9.1.9 Dynamic Foreign Interface

Runtime systems cannot dynamically load (additional) foreign code. However, they can load QOF files and if those have object file dependencies then the object files will be dynamically loaded at that time.

It is possible to dynamically load foreign code into a stand-alone program using `load_foreign_executable/1` or `load_foreign_files/2`.

9.1.10 Linking with QUI

During development of applications involving both Prolog and foreign code, it can be very useful to build a stand-alone program that includes QUI. This way, you can use the QUI debugger for stepping through the Prolog code and a standard debugger such as `gdb(1)` for stepping through the foreign code.

A good way to do this is to create a file '`qui.pl`' containing just the one line:

```
:- ensure_loaded(library(qui)).
```

This should then be compiled to QOF in the normal way and linked into your application. For example:

```
% qpc -D <application files> qui.pl
```

Then start up the resulting executable file using a command such as

```
% gdb a.out
```

See Section 10.3.11 [fli-p2f-fcr], page 400 for information about using a standard debugger in conjunction with a Prolog executable file.

> **Please note:** The reason for putting the `ensure_loaded` command in a file by itself, rather than including it in your application code, is that it will not work if executed in an ordinary `prolog` system that does not include QUI; it is not possible to load QUI dynamically.

9.2 The Runtime Generator

9.2.1 Introduction

The purpose of the Runtime Generator is to provide a convenient and cost-effective way to distribute Prolog application programs to end-users. Initially a Prolog application should be developed using the Development System. The Development System allows the application developer to load, run, modify and debug programs interactively, without having to leave the environment. When an application program has been completed, the Runtime Generator product allows a developer to create a production version of the application, ready to be shipped to end-users. This version of the application is called a *runtime system*.

The important points to note about runtime systems are:

1. They can easily be deployed, because they do not require any authorization codes in order to run.
2. They are smaller than programs built on top of the Development System, since they do not include program development features such as the compiler, the debugger and the Emacs interface

The process of building a runtime system is almost identical to the process of building a *stand-alone program* with the Development System, and both of these processes are documented in Section 9.1 [sap-srs], page 337. The compiler `qpc` is used to compile your Prolog source files into Quintus Object Format (QOF) files, and these are then linked together using the link editor `qld`. Both `qpc` and `qld` require a '`-D`' command-line option when building a stand-alone program; the default is to build a runtime system.

It is recommended that before turning an application program into a runtime system you first build it as a stand-alone program. This way, if there are any problems, you will have the debugger available to help you eliminate them.

Once you have built a stand-alone program, it should be very easy to rebuild the application as a runtime system. This section describes the few points that may need to be considered in making this transition:

1. Some built-in predicates are not available in runtime systems, since they are intended for use in program development rather than for use in application programs. See Section 9.2.2 [sap-rge-dspn], page 356.
2. Runtime systems do not have the normal Prolog top level. Instead, you must specify a starting point for the program by defining the predicate `runtime_entry/1`. This predicate also allows you to specify what is to be done after an abort occurs. See Section 9.2.3 [sap-rge-pro], page 357.
3. Runtime systems have different default behavior on a `^C` interrupt. Instead of giving the user a set of choices, the system immediately aborts. See Section 9.2.4 [sap-rge-iha], page 358.
4. If your runtime system contains dependencies on "non-standard" shared libraries, or on one or more shared object files (specified in calls to `load_foreign_executable/1`), then in order to be able to deliver the runtime system to a different machine you will need either
   a. to make the system completely self-contained by replacing the shared object files and libraries with equivalent archive files when building it; or
   b. to ensure that all necessary shared libraries and shared object files are available on the target machine.

   See Section 9.2.5 [sap-rge-sos], page 359 for more information on this issue.

Another difference between runtime systems and programs running under the Development System is that runtime systems suppress informative error messages, on the basis that such messages are intended for the application developer, not the application user. For example, if a runtime system consults a file, no message is printed to the screen to indicate this. If you wanted some such message to appear at runtime, you would have to program it yourself.

Error messages are not suppressed, however. If you want to suppress the printing of some (or all) error messages you can do it by providing a definition for the predicate `message_hook/3`. See Section 8.20.3.3 [ref-msg-umf-ipm], page 331.

9.2.2 Predicates not supported by the Runtime Kernel

The predicates listed below are not supported by the Runtime Kernel, since their purpose is to aid the development process rather than to be used in application programs. `qld` or `qcon` will print '`Undefined`' warning messages if you try to build a runtime system with calls to any of these predicates. At run time, these calls will raise existence errors in a runtime system.

debugger: `debug/0`, `nodebug/0`, `trace/0`, `notrace/0`, `spy/1`, `nospy/1`, `nospyall/0`, `debugging/0`, `leash/1`, `add_spypoint/1`, `current_spypoint/1`, `remove_spypoint/1`,

help system:
`help/[0,1]`, `manual/[0,1]`

advice: `add_advice/3`,
`check_advice/[0,1]`, `current_advice/3`, `nocheck_advice/[0,1]`, `remove_advice/3`,

program development:
`break/0`

If you call any of these predicates from compile-time code, such as from an embedded command or in a definition of `term_expansion/2`, the call will raise existence errors when using `qpc`.

Any use of these predicates should be eliminated from both your run-time and your compile-time code, unless you can be sure that they won't be called or you don't mind them raising existence errors.

In addition, the predicates `load_foreign_files/2` and `load_foreign_executable/1` are not available in the Runtime Kernel but may be used in embedded commands. That is, you can use them at compile time, in order to link foreign code into your program, but you cannot use them when the runtime system is running.

Since the compiler is not included in a runtime system, the effect of the load predicates is altered in a runtime system. Whenever one of these predicates would normally compile a file into memory, it instead loads that file into memory as dynamic. This is equivalent to

```
load_files(file, all_dynamic(true)).
```

This should not normally make any significant difference, except that loading the file is faster and running it is slower.

The predicate `multifile_assertz/1` cannot be used on compiled (static) predicates in a runtime system. This restriction does not apply to predicates that are loaded by `compile(`*`File`*`)` at run time, since such predicates are really loaded as dynamic.

9.2.3 Providing a Starting Point: `runtime_entry/1`

The application developer *must* specify what is to happen when the program is started up. This is done by defining the predicate `runtime_entry/1`. When the runtime system is run, the goal (A) is invoked. When that goal terminates, either by succeeding or by failing, the runtime system terminates.

```
runtime_entry(start)                                    (A)
```

Similarly, it is possible to specify what is to be done on an abort. An abort happens when a call is made either to the built-in predicate `abort/0` or to the C routine `QP_action(QP_ABORT)`. (By default, a call of `QP_action(QP_ABORT)` happens when a user types `^`*`C`*— see

Section 9.2.4 [sap-rge-iha], page 358). At this point, the current computation is abandoned and the program is restarted with the goal (B).

```
runtime_entry(abort)                                                    (B)
```

Effectively this replaces the original call to `runtime_entry(start)`, so that when this call succeeds or fails, the runtime system terminates For example (C) will obviously loop indefinitely until you interrupt it with a `^C`. At that point it will abort, and since the goal `runtime_entry(abort)` will fail, the program will terminate.

If you were to add the clause (D) you would make the program impervious to `^C` interrupts and quite hard to terminate.

```
runtime_entry(start) :- go.                                             (C)
go :- go.

runtime_entry(abort) :- go.                                             (D)
```

For this reason, it is recommended that you *not* write your code as (E) as this will cause your program to restart on `^C` or errors.

Users of the module system should ensure that the predicate `runtime_entry/1` is defined in the module `user`, that is, not inside any user-defined module. You may use a clause of the form (F) in a module-file to do this. (see Section 8.13 [ref-mod], page 271).

```
runtime_entry(_):- ...                                                  (E)


user:(runtime_entry(...):-...)                                          (F)
```

9.2.4 Control-c Interrupt Handling

By default, `^c` causes a runtime system to abort its current computation and to restart with the goal `runtime_entry(abort)`, exactly as if the built-in predicate `abort/0` had been called. This behavior can be modified from C code by means of the system function `signal(2)`. Such modification is done in exactly the same way as under a Development System; see the Reference Pages for further information.

Windows Caveats:

- In Windows 95/98, `^C` interrupt handling only works in windowed executables, and only when waiting for input.
- In statically linked executables running under Windows NT, `^C` interrupt handling only works when waiting for input.

9.2.5 Shared vs. Static Object Files

A runtime system is a single executable program that should be easily transferable to a different machine. By default, the executable built by `qld` will use dynamic libraries where it can, such as the dynamic C library ('`-lc`' under UNIX, '`/MD`' under Windows). This requires that a corresponding library exist on the target machine on which the executable will be run.

Under UNIX, you may encounter problems if libraries included in your executable are not installed in "standard places" — e.g. in '`/usr/lib`' — on the target machine. For example, if you specify '`-lX11`', but the corresponding file '`libX11.so.4.2`' resides in '`/usr/local/lib/X11`'. Typically, a user has to set the `LD_LIBRARY_PATH` environment variable to find libraries in non-standard places, although another option at installation time might be to add a call to `ldconfig(8)` in '`/etc/rc.local`' to include the directory containing a shared library into the system-wide list of "standard places" to find shared libraries. The `ldd(1)` command lists the dynamic dependencies of an executable and whether or not these can be found.

The problem exists under Windows but the details are different. In particular Windows shared libraries are looked for in the folders specified by the `PATH` environment variable and in some further "standard places". Consult the Windows documentation for details.

In addition to shared libraries, your executable may contain shared object files, specified in calls to `load_foreign_executable/1`. `qld` passes the absolute file paths for these files to the linker, which results in the dependencies for these shared object files being stored in the runtime system executable as absolute filenames. Note that this problem can arise even if you have no foreign code of your own, if you are using the Quintus Prolog library or X interfaces.

A solution to the shared object files problem is to use the '`-S`' option with `qld`, which tries to substitute archive files for shared object files where they exist. If the shared object files have dependencies on other shared libraries then those libraries need to be explicitly listed in the `qld` command, as `qld` does not track these dependencies. This makes your executable file larger, as the library code is stored within the executable rather than linked in at start-up time. Each shared object file provided in the Quintus libraries has an equivalent archive file that can be substituted by the '`qld -S`' command.

For example, under UNIX, building a runtime system from the file in (A) with the command (B) produces an '`a.out`' file, which prints the date and time, as in (C).

test.pl

```
:- use_module(library(date)).                                    (A)

runtime_entry(start) :-
        datime(X),
        portray_date(X),
        nl.
```

```
% qpc test.pl                                                    (B)
```

```
% ./a.out                                                        (C)
2:10:09 PM 1-Feb-91
```

This '`a.out`' file has a dependency on the Prolog library as well as on the C library, as shown by the `ldd(1)` command:

```
% ldd a.out
    /usr/local/quintus/generic/qplib3.5/library/sun4-5/libpl.so
    libc.so.1 => /usr/lib/libc.so.1.8
```

Under Windows, the example would be almost the same, with the difference that the default name of the executable produced by `qpc` is '`a.exe`' rather than '`a.out`'. To view dependencies you can type *`dumbin /DEPENDENTS`* `a.exe`.

Hence this runtime system will not work on a machine where Quintus Prolog is not installed. To build a runtime system that does not have this dependency, it is necessary to call *`qld -S`* explicitly, rather than just calling *`qpc test.pl`* as shown above. Under UNIX, the necessary command sequence is:

```
% qpc -c test.pl
% qld -Sd test.qof
```

That is, `qpc` is called with the '`-c`' option so that it stops after producing a '`.qof`' file, rather than calling `qld`. Then `qld` is called with the '`-S`' option so that it substitutes '`libpl.a`' for '`libpl.so`'. The result is an executable that depends only on the C shared library:

```
% ldd a.out
    ilbc.so.1 => /usr/lib/libc.so.1.8
```

This executable should run without problem on a different machine.

Under Windows, the linker does not automatically add all needed OS libraries. These need to be added explicitly as follows:

```
C:\> qpc -c test.pl
C:\> qld -Sd test.qof -LD qpcon-
soles.lib user32.lib gdi32.lib comdlg32.lib
```

Windows notes:

1. The '`-S`' and '`-W`' flags can be combined.
2. If the '`-S`' option is used, the '`-LD`' option must also be used, together with the library references '`qpconsoles.lib`', '`user32.lib`', '`gdi32.lib`', and '`comdlg32.lib`'.
3. If the Prolog code has a foreign executable dependency on '`myforeignex`', a static library '`myforeignexs.lib`' needs to be created. The trailing '`s`' is significant (see below); `qld` assumes this naming convention. Here is an example of the necessary sequence of commands to create a static executable '`myforeignex.exe`':

```
C:\> cl /c /MD myforeignex.c
C:\> link /lib /OUT:myforeignexs.lib myforeignex.obj
C:\> qpc -c myprog.pl
C:\> qld -Sdvo myprog.exe myprog.qof -LD \
     user32.lib comdlg32.lib qpconsoles.lib gdi32.lib
```

   The above command produces an executable that uses the static version of the Runtime Kernel and has no DLL dependencies. Statically linked applications can still dynamically load foreign code DLLs, provided these DLLs do not call any of the Quintus Prolog C API functions.

   In order to distinguish static libraries from DLL import libraries in foreign dependencies, the following naming convention has been chosen. If the '`-S`' option is used, when processing a library dependency `qld` will first search for the library with an '`s`' appended to its name, for example '`libqps.lib`' for the Embedding Layer, and if not found it tries the original name.

9.2.6 Building DLLs containing Prolog code

This section is only relevant for Windows.

It is possible to build DLLs containing Prolog code to be linked dynamically into applications. Packaging your code as a DLL promotes sharing and is also a requirement for applications where an application needs to dynamically link your code. An example illustrating this and other techniques that are useful when embedding Prolog in C is the Visual Basic interface source code; see '*quintus-directory*`\src\vbqp`'.

Applications typically look for DLLs in the same directory as the application itself and also in any directories specified in the `PATH` environment variable. Applications also look in some system directories but that is less useful for our purposes.

The easiest way to ensure that the DLL (and the Quintus runtime DLLs) are found is to put the DLL together with the Quintus runtime DLLs in *runtime-directory* and then ensure that the `PATH` environment variable is set up to include that directory, as described below.

Another method, especially suitable for running on a machine where Quintus Prolog is not installed, is to put all the DLLs in the same directory as the application. This includes the DLL built as below and the appropriate Quintus runtime DLLs.

See the Microsoft documentation for more information about DLLs.

To build a DLL containing Prolog code, follow these steps:

9.2.6.1 Setting up the environment

Set up your environment variables so that the Quintus binaries can be found. The batch file '*runtime-directory*`\qpvars.bat`' can be run for this purpose.

9.2.6.2 Compiling the Prolog code

Compile your Prolog code, e.g.

```
C:\> qpc -c vbqp.pl
```

The option '`-c`' ensures that only a Quintus object file (QOF) is produced, in this case '`vbqp.qof`'.

9.2.6.3 Compiling the C code

Compile your C code, e.g.

```
C:\> cl /nologo /c /MD vbqp.c
```

The option '`/MD`' ensures that the DLL version of the C runtime libraries are used. The option '`/c`' ensures that no linking is performed, only an object file is produced, in this case '`vbqp.obj`'.

9.2.6.4 Linking the DLL

Finally, use `qld` to link the DLL with the code produced in the previous steps and any additional libraries, e.g.

```
C:\> qld -Ydo vbqp.dll vbqp.qof vbqp.obj -LD oleaut32.lib
```

Please note: `qcon` will complain that there is no `runtime_entry/1`. This can be ignored.

The option '`-o`' is used to name the resulting DLL, in this case '`vbqp.dll`'. The option '`-d`' is used to ensure that any dependencies are also linked. The option '`-LD`' passes the rest of the command line to the linker, in this case it causes the library '`oleaut32.lib`' to be linked with the resulting DLL.

The option '`-Y`' (new in this release) tells `qld` to produce a DLL as opposed to an ordinary EXE file. A similar effect could be obtained with:

```
C:\> qld -do vbqp.dll vbqp.qof vbqp.obj -LD /dll oleaut32.lib
```

9.2.7 Installing an Application: runtime(*File*)

A runtime system is a single, executable program that can easily be transferred to a different machine. However, in many cases, application programs require access to some auxiliary files in the course of their execution. These may be files of Prolog code that are to be consulted at run time, or they may be data files in any arbitrary format. If your application requires some such files, you may need to require users of the application to follow some installation procedure before they can use it.

One approach to this problem is to use the runtime file search path for accessing all your auxiliary files. The default runtime file search path is the runtime directory where Prolog executables and objects are located. You can modify this by putting a goal such as the following in your '`prolog.ini`' file.

```
:- retractall(file_search_path(runtime,_)),
   assert(file_search_path(runtime,
                           '/usr/fred/runtime_files')).
```

This allows you to use `runtime(`*`File`*`)` anywhere you need to specify a file, such as in calls to `open/3`, `see/1`, `consult/1` or `absolute_file_name/2`. When you do this, the file will be sought in the directory '`/usr/fred/runtime_files`'.

When you have built a runtime system, you can change the runtime directory and thus the runtime file_search_path by means of the `qsetpath` utility program (see Section 20.1.8 [too-too-qsetpath], page 1495). This can be changed, perhaps from an installation shell script to be run by the end user. The appropriate command is:

```
% qsetpath -rdirectory runtime-system
```

The program *runtime-system* will then look in *directory* for its runtime files. Note that write permission will be needed on *runtime-system* for the `qsetpath` command to work.

There is another utility, `qgetpath` (see Section 20.1.3 [too-too-qgetpath], page 1480), which can be used to print the runtime directory of a runtime system. For example,

```
% qgetpath -r runtime-system
```

writes the runtime directory to standard output. See the `qsetpath` and `qgetpath` Reference Pages for more information.

10 Foreign Language Interface

10.1 Overview

The *Foreign Language Interface* is the protocol by which you can call functions written in other programming languages from Prolog (see Section 10.3 [fli-p2f], page 375), and by which you can call Prolog predicates from C (see Section 10.4 [fli-ffp], page 413).

In both cases you have to supply declarations in Prolog specifying the argument types of the function/predicate being called. This is necessary so that the Prolog system can automatically make the necessary transformations of data as they are passed between the languages. The declarations are compiled into special abstract machine instructions in order to minimize the cost of inter-language calls.

It is possible to specify that an argument is a Prolog term. In this case a foreign function receiving such a term sees it as a special kind of object called a *QP_term_ref*. A set of functions is provided that allow foreign functions to manipulate terms via these QP_term_refs. Term passing is described in Section 10.3.8 [fli-p2f-trm], page 395 and Section 10.4.4.2 [fli-ffp-a2s-trm], page 420. The functions for manipulating terms are described in Chapter 19 [cfu], page 1345.

Some kinds of data are not best represented in Prolog. It is better to keep such data in foreign data structures and just pass pointers to these structures to Prolog. Given such pointers, Prolog arithmetic predicates such as `is/2` and `=:=/2` allow access to foreign data; see Section 8.8.4 [ref-ari-aex], page 235. Foreign data can also be modified directly from Prolog using `assign/2`, which is described in Chapter 19 [cfu], page 1345.

Input/Output operations on Prolog streams can be performed in C. Functions, macros and variables for this purpose are summarized in Section 10.5.8 [fli-ios-bio], page 479. The I/O system has been designed to make it maximally customizable at the C level. The kinds of customization that you might want to do are to create a stream to read from or write to a socket, or to create a stream to read from an encrypted file. The C level of the I/O system is described in Section 10.5 [fli-ios], page 433.

A number of functions are defined to aid in embedding Prolog programs into other software systems. These functions all begin with the letters '`QU`' and they are defined in Section 10.2 [fli-emb], page 365. The source of these functions is provided with the system, and you may define your own versions to replace them. To do this you will need to link your routines with the development system to make your own customized version of the development system. How to do this is described in Chapter 9 [sap], page 337.

10.2 Embedding Prolog Programs

10.2.1 Overview

As outlined in Section 1.2 [int-hig], page 4, a constellation of new features greatly extends the relationship between Quintus Prolog code and foreign language code. It is now possible to embed Prolog code in a program written in another language without restrictions. Clearly, the first requirement for embedding Prolog code freely in foreign code is to be able to call Prolog from foreign code and vice versa. C calling Prolog is a major new feature of release 3 and is discussed in Section 10.4 [fli-ffp], page 413.

A further requirement is that all types of data structures can be passed between Prolog and the foreign code. Previously it was not possible to pass compound Prolog terms between Prolog and foreign code. In addition there were serious limitations on passing mathematical data. release 3 adds

- the ability to access Prolog terms from the foreign language once they are passed to it and perform Prolog operations on them. (Discussed in detail in Section 10.3 [fli-p2f], page 375 and Section 10.4 [fli-ffp], page 413, and in the reference pages cited there for term passing predicates and functions).
- the ability to access data structures in a foreign language from Prolog and to perform destructive operations on them. (Discussed in Section 10.3.9 [fli-p2f-poi], page 397 and `assign/2` and `is/2`).
- 32 bit integers and 64 bit floats

With these new features, Quintus Prolog fulfills the full data passing requirement.

The foreign language interface is now fully bidirectional. This in itself is not sufficient for embeddability in a strong sense. The Prolog portions of the application must in addition be *well-behaved*. That is, they must not make any assumptions about how the operating system will handle such matters as memory and input/output operations. This is where the *embedding layer* of Quintus Prolog comes in.

10.2.2 The Embedding Layer

In many cases, the embedding layer will be transparent to the application developer. It provides a full set of default functions for interaction between Prolog and the host operating system. Frequently all you will need to do is use the extended FLI and let the defaults provided by Quintus Prolog take care of the operating system requirements concerning memory management and I/O. However, Prolog no longer insists on controlling memory management and input/output operations, should this be impossible or undesirable in your application. These default interfaces are fully user redefinable.

Memory management: Quintus Prolog release 3 does not have any restrictions on the underlying memory. This is a crucial aspect of embeddability. Any good Prolog implementation will start up with the minimum amount of memory necessary and expand and shrink depending on the memory needed to execute each goal. In previous releases of Quintus Prolog

(as well as most other Prolog implementations) all the memory that Prolog used had to be contiguous. So it was possible that if some foreign component of the application allocated memory from the top, it would disable Prolog from growing any further. With release 3, Quintus Prolog runs on discontiguous memory. Therefore Prolog can share the process's address space with memory allocated to Prolog interspersed with memory allocated by other components of the application.

The user can replace Quintus Prolog's low level memory management functions. This is essential if the user has an application that would like to take care of all memory management and does not want Prolog to directly make system calls to the OS to allocate memory. This makes it easy to link Prolog with other components that have more rigid restrictions about its memory allocation. (Discussed further in Section 10.2.3.2 [fli-emb-how-mem], page 373).

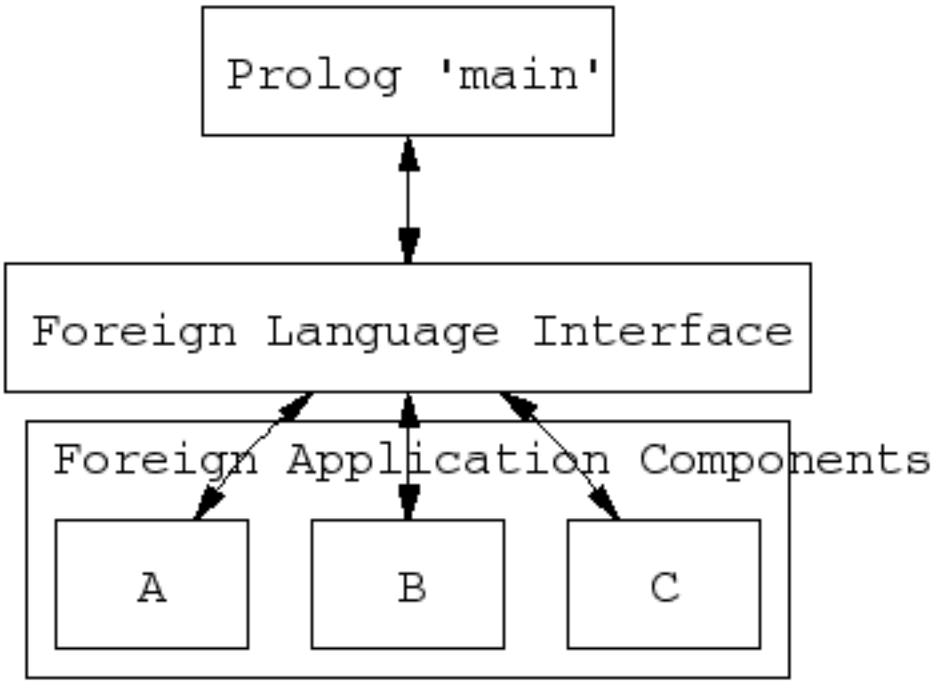
Input/Output: The user can create, access and manipulate Prolog I/O streams from foreign code. This provides a unified way of performing I/O from Prolog as well as foreign code. It also gives the user the ability to have I/O streams to sockets, pipes or even windows. Graphical user interfaces have become a natural Prolog component of an application.

The low level I/O functions can also be replaced. This is essential if the user has a large application and wants to take care of all I/O without any direct calls from Prolog to the OS to perform I/O. This is elaborated in Section 10.2.3.3 [fli-emb-how-iou], page 374.

> **Windows caveat:** Redefining functions in the Embedding Layer only works in executables built with '`-S`'; see Section 9.2.5 [sap-rge-sos], page 359.

10.2.2.1 Contrasting Old and New Models

To understand the motivation for the new "embeddability" layer, contrast the model of foreign language interface that previously held, as illustrated in the two following figures, with the new model illustrated in the figure "New Model".

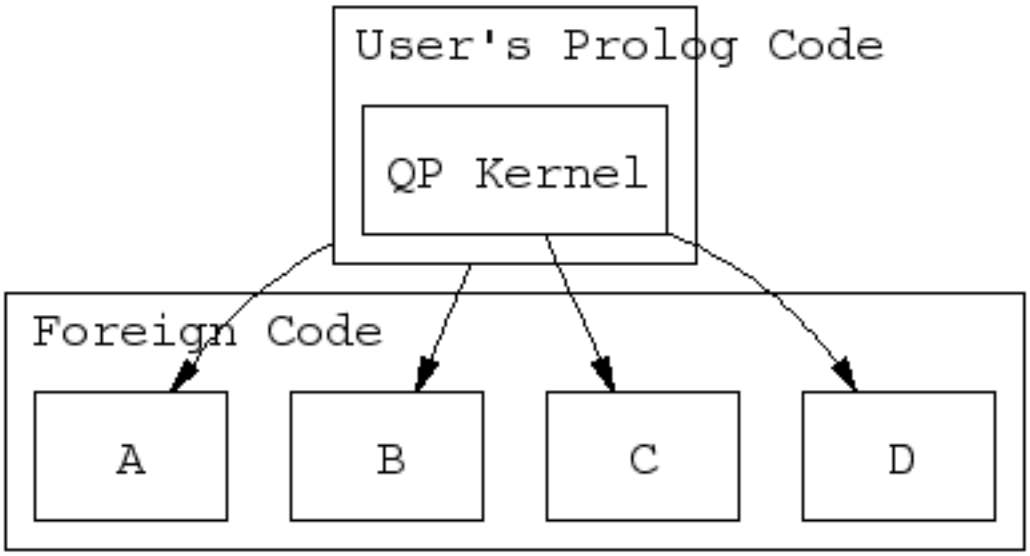


Former Prolog interface to foreign code

The Old Model:

Under the one-directional foreign language interface, it was necessary to write a `main` program in Prolog as illustrated in the above figure The foreign language interface was able to call foreign code from this `main`. There were basically two components, the Prolog Main, and the Foreign Program. The foreign program itself could have all sorts of components. However, from the point where the foreign code began, no more Prolog code could be inserted. For instance, if you wanted to add a Prolog component to Module C of the program, it would be necessary to restructure the program to enable control to return to the Prolog `main`, where the new Prolog code could be called, and then reinvoke the foreign code in Module C.

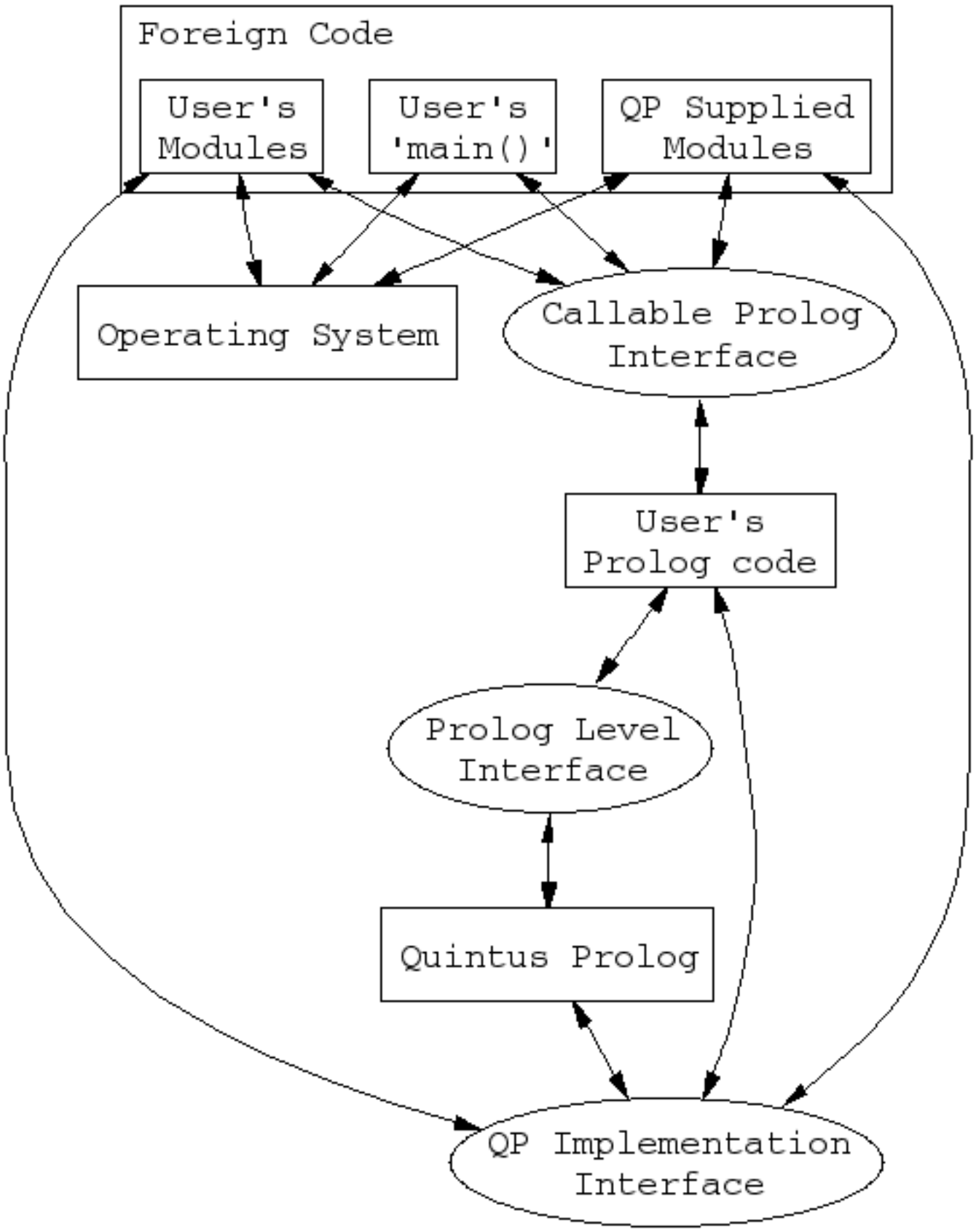
Another limitation of the old foreign interface was the possibility of conflicts between the foreign code called by the user's Prolog code and the foreign code used by the Quintus Prolog kernel. For example, the Quintus Prolog kernel required total control of all memory allocation to ensure that the Prolog memory areas were contiguous. Therefore the users code could not use the system call `sbrk(2)` to allocate memory, but had to use the `malloc(3)` function provided with Quintus Prolog (see the following figure). Now, however, the foreign functions used by the Quintus Prolog kernel forms the Embedding Layer and it is possible for the user to redefine these functions to conform to the requirements of his foreign code.



Prolog Kernel and Application calling Foreign Code

The New Model:

The new model can be represented as in the following figure. The Embedding Layer contains C functions that establish defaults for memory management and I/O. The user can redefine any of these modules so as to prevent conflicts between the application's C calls and the C calls made by the Quintus Prolog kernel.


Foreign Code
User's Modules
User's 'main()'
QP Supplied Modules
Operating System
Callable Prolog Interface
User's Prolog code
Prolog Level Interface
Quintus Prolog
QP Implementation Interface

New Model

10.2.3 How Embedding Works

The next three sections describe the major areas of Prolog that can be redefined to facilitate the embedding of Prolog code in foreign language applications:

- Initialization, i.e. `main()`
- Memory management
- Input/Output

Consider the details presented in these sections in the context of this overview of the process of creating a program with a Prolog component:

1. Write the foreign code (‘`prog.c`’) and the Prolog code (‘`component.pl`’), using the steps for the C calling Prolog interface described in Section 10.4.1.1 [fli-ffp-bas-sum], page 414 to call the Prolog predicates. Most likely you will be defining a `main()` routine; in this case be sure to call `QP_initialize()`. An example is found in the reference page.
2. Determine whether it is necessary to customize any of the API modules for initialization, memory management or input/output. This is discussed in Section 10.2.3.1 [fli-emb-how-mai], page 372, Section 10.2.3.2 [fli-emb-how-mem], page 373, and Section 10.2.3.3 [fli-emb-how-iou], page 374.
3. If customization is required, and it seldom will be, rewrite the relevant modules.
4. Produce the executable:
   a. Compile ‘`component.pl`’ using *`qpc -c component.pl`*
   b. Compile ‘`prog.c`’ using e.g. *`cc -c prog.c`*
   c. Link the two using `qld`. A typical call would be:

      ```
      % qld -Dd component.qof prog.o -o BigApplication
      ```

      You may also want to link in QUI in order to be able to use the debugger, as described in Section 6.2 [dbg-sld], page 121. The process of linking QUI into an application is discussed in Section 9.1.10 [sap-srs-qui], page 354.
   d. Run BigApplication. Debug, using standard debugging tools such as `gdb(1)` for C code, and Prolog debugging tools for Prolog code.

Quintus Prolog provides defaults for interfacing the operating system. If customization is necessary in this area, a user must completely redefine, not just extend, the supplied functions.

> **Please note:** The default OS interface functions that can be redefined all have names beginning with the prefix ‘`QU_`’.
>
> The QU_ functions are like hooks in the sense that they provide you with a place to insert code that changes Prolog’s behavior. However, we do not include

Prolog hooks such as `message_hook/3` in this discussion because the point of embedding is to call Prolog code from foreign programs. The Prolog hooks are used independently of embedding.

10.2.3.1 Defining your own main()

Normally, when building an executable with `qld` the Quintus `main()` routine is linked in to the executable, which initializes the Prolog environment and calls `QP_toplevel()`. `QP_toplevel()` will either:

1. Start an interactive top-level, which prompts for a command to be typed; or
2. Call runtime(start) in the case of a runtime system (i.e. if '`-D`' was omitted in the call to `qld`).

However, you are not limited to using this default `main()`. You can define your `main()` and have Prolog as a function call.

This should be done if the Prolog component(s) of your application are such that they may not be called in a given run of the program. In that case, you would not want to initialize Prolog unless it became necessary.

An example of this sort of case is a program written in C that utilizes menus. The end user can select a number of options. One of these options involves further decision making, and runs an expert system written in Prolog. If the user doesn't happen to select this menu option on a given occasion, there is no reason to use the resources involved in initializing Prolog. So you would write a `main()` that is dependent upon this menu selection. Once the user selects this option and thus starts up Prolog, however, subsequent invocations will recognize that Prolog is already initialized and will not do it again.

Another situation where it makes sense to "redefine" `main()` is where you already have a large application written in C or some other foreign language and you wish to extend it with a module written in Prolog without having to rewrite the top level of the existing program.

If you choose to use a different `main()`, you should be aware that the default Quintus `main()` provides certain functionality, which will have to be included in the user-supplied `main()`:

- Initialize memory, I/O.
- Set up command line arguments.
- Initialize file search paths, file tables and symbol tables.
- Do start up hooks associated with a statically linked component in QOF files.
- Do any necessary restores, and any start up hooks associated with the restored files.

The built-in function `QP_initialize()` takes care of these tasks. An example of a user-supplied `main()` can be found in the reference page for `QP_initialize()`.

The QP_* functions require that Prolog be initialized for memory management, etc. Thus, whenever `main()` is redefined, it will be necessary to call `QP_initialize()`. There is no harm in calling this routine more than once. So people writing portions of large projects can safely assume Prolog isn't initialized, and call `QP_initialize()`.

10.2.3.2 The Embedding Functions for Memory Management

Release 3 of Quintus Prolog makes it possible to run Prolog as an embedded system. In terms of memory management this means that Prolog does not assume full control of the address space or that all its memory is going to be contiguous. This makes it possible to share the same address space between Prolog and other applications. The memory used by Prolog can be interspersed with the memory used by the application into which Prolog is embedded.

With Release 3 all of Prolog's sophisticated memory management can be built on top of a primitive layer, which users can replace with their own functions. Such replacement is only required when the application in which the Prolog code is embedded demands full control of the address space and memory allocation. In general it is not necessary or even advantageous to do this.

The embedding layer of memory management comprises three primitive functions: `QU_alloc_init_mem()`, `QU_alloc_mem()` and `QU_free_mem()`. The system has a default implementation of these functions based on `sbrk(2)` for UNIX and `VirtualAlloc()` for Windows. If Prolog is to become part of an embedded package that would like to provide its own memory management routines then the user can redefine these functions and statically link it with the Prolog system. (Static linking is discussed in Section 9.1 [sap-srs], page 337.) If the user does not provide these functions, the API functions (in e.g. '`libqp.a`' or '`libqps.lib`') will be used by default.

This layer is responsible for allocating memory to Prolog and freeing memory back to the Operating System. Prolog calls the functions `QU_alloc_mem()` and `QU_free_mem()` for these purposes. `QU_init_mem()` is called the first time Prolog makes a call to allocate memory. If the user redefines these functions the redefinition should meet the specifications for these functions mentioned in the reference pages. An example of defining your own memory management routines is given in the reference page for `QU_alloc_mem()`.

This layer is also responsible for the environment variables `PROLOGINCSIZE` and `PROLOGMAXSIZE`, which are available for customizing the default memory management routines. The user can set `PROLOGINCSIZE` to set the least amount by which Prolog should expand each time. The user can set `PROLOGMAXSIZE` to limit the maximum memory used by Prolog. See Section 8.12.7 [ref-mgc-osi], page 264.

The Prolog system top level supplied by Quintus automatically cleans up Prolog memory each time it returns to top level. However, when Prolog is called directly from a foreign function the Quintus top level (or a user-defined equivalent) need not be used. If nothing else is done (such as calling `trimcore/0` in the Prolog code), when a Prolog query returns,

the memory allocated to Prolog will stay expanded to whatever was required to compute the previous solutions.

In the case where it is more convenient to call a C function than a Prolog built-in, `QP_trimcore()` is provided to explicitly clean up Prolog memory. It has the same effect as `trimcore/0`. Like `trimcore/0` it should be used judiciously, as overuse can result in unnecessary time being spent in memory expansion and contraction. However, when Prolog is to be dormant for a period, or as much free memory as possible is desired, `QP_trimcore()` can be quite useful.

10.2.3.3 The Embedding Functions For Input/Output

Prolog streams are designed by default to be channels for I/O operations to a file or a terminal. User defined streams enable these operations to be performed on other types of object: notably, windows or a network channel.

The embedding input/output functions create the default Prolog streams and provide the user with the default parameters for creating a user-defined stream. It is possible to change these defaults. However, in general it is not necessary or even advantageous to do this.

Such replacement is only required when the application in which the Prolog code is embedded demands full control of the I/O system and does not want Prolog to make direct calls to the operating systen to perform I/O. One instance of such usage is to embed a Prolog program within an application that uses a graphical window-oriented user interface.

The embedding layer for input/output contains four functions:

`QU_stream_param()`
: sets up default field values in a `QP_stream` structure.

`QU_initio()`
: creates three Prolog initial streams: *user_input* stream, *user_output* stream and *user_error* stream.

`QU_open()`
: creates streams opened by `open/[3,4]` and `QP_fopen()`.

`QU_fdopen()`
: creates streams that were already opened by the system function `open(2)`.

Details of each function can be found in the individual reference pages. Any of these functions can be supplied in linking a Prolog system through `qld`. If any function is not supplied, the default version of that function is linked in.

A number of C macros and functions are provided in '`<quintus/quintus.h>`' and e.g. '`libqp.a`' to access and manipulate Prolog streams where it is more convenient to access them from C rather than calling a Prolog builtin. For example, `QP_getc()` will get a character from a Prolog stream in the same way as `get0/2`. These macros and functions are listed in Section 10.5.8 [fli-ios-bio], page 479.

10.2.4 Summary of Functions

Detailed information is found in the reference pages for the following:

- `QP_initialize()`
- `QP_toplevel()`
- `QU_alloc_init_mem()`
- `QU_alloc_mem()`
- `QU_fdopen()`
- `QU_free_mem()`
- `QU_init_mem()`
- `QU_initio()`
- `QU_open()`
- `QU_stream_param()`

10.3 Prolog Calling Foreign Code

10.3.1 Introduction

This section describes how to load and call programs written in C, Pascal, FORTRAN, or Assembly language. This may be desirable in order to:

- combine Prolog with existing programs and libraries, thereby forming composite systems;
- interface with the operating system or other system level programs;
- speed up certain critical operations.

Examples showing the correct use of the foreign interface are found in the library directory. Examples of incrementally loading C, Pascal and FORTRAN code can be found in Section 10.3.15 [fli-p2f-fex], page 402.

Foreign functions are loaded directly into the Prolog system by using one of the built-in predicates `load_foreign_executable/1` or `load_foreign_files/2`. These predicates load executable images or object files into the address space of the running Prolog.

Before calling these predicates, you must prepare facts in the database that describe which functions may be called by Prolog, the native language of each function, and the argument types of each function. This information is used to link Prolog predicates and foreign functions when loading the foreign code.

The foreign language interface supports the direct exchange of Prolog's atomic data types (atoms, integers and floating-point numbers). The data is automatically converted between Prolog's internal representation and the internal representation of the foreign language.

The foreign language interface also supports passing any Prolog term to C and receiving any Prolog term from C. A set of C functions is provided to type test and access terms passed to C and to create new Prolog terms in C. For information on these functions see Chapter 19 [cfu], page 1345.

Prolog procedures that are attached to foreign functions are determinate, in that they succeed at most once for a given call and are not re-entered on backtracking. This imposes no serious limitation, since it is always possible to divide a foreign function into the part to be done on the first call and the part to be redone on backtracking. Backtracking can then take place at the Prolog level where it is naturally expressed.

10.3.1.1 Summary of steps

Following is a summary of the steps that enable you to call foreign code from a Prolog predicate:

In the Prolog code:

1. Declare the relevant object file(s), and the names of the functions defined in them, by defining clauses for `foreign_file/2` (see Section 10.3.3 [fli-p2f-lnk], page 380).
2. Specify the argument passing interface for each function by defining clauses for `foreign/3` (see Section 10.3.4 [fli-p2f-api], page 382).
3. Load the foreign files into Prolog by calling `load_foreign_executable/1` or `load_foreign_files/2` (see Section 10.3.2 [fli-p2f-uso], page 376).

10.3.2 Using Shared Object Files and Archive Files

By default, foreign code must be packaged as shared object files for use with `load_foreign_executable/1`. Archive files are used for statically linking foreign code to executables (see Section 9.1 [sap-srs], page 337).

A shared object file is constructed from a list of object files (and libraries) using the system linker. An archive file is constructed from a list of object files using special tools. In both cases, the object files are generated using the foreign language compiler.

Linkers require special options to construct a shared object file, and may require that the object files used to generate the shared object files be compiled with *position independent code* or other special compiler options. Under Windows, Quintus Prolog also requires a special compiler option for inclusion into archive files.

Let *CC* denote the compiler command, let *SFLAGS* denote the compiler options for shared object files, let *AFLAGS* denote the compiler options for archive files, and let *LFLAGS*

denote the linker options. The following table gives these options for the supported Quintus Prolog platforms.

Platform	*CC*	*AFLAGS*	*SFLAGS*	*LFLAGS*
`linux`	`gcc`	(none)	'`-fPIC`'	'`-shared`'
`alpha`	`cc`	(none)	(none)	'`-taso -shared -expect_unresolved 'Q?_*''`
`hppa`	`cc`	'`+DAportable`'	'`+Z +DAportable`'	'`-b`'
`hppa`	`gcc`	(none)	'`-fPIC`'	'`-shared`'
`sgi`	`cc`	'`-n32`'	'`-n32`'	'`-n32 -shared`'
`sgi`	`gcc`	'`-mabi=n32`'	'`-mabi=n32 -fPIC`'	'`-mabi=n32 -shared`'
`sun4-5`	`cc`	(none)	'`-K pic`'	'`-G`'
`sun4-5`	`gcc`	(none)	'`-fPIC`'	'`-shared`'
`rs6000`	`cc`	(none)	(none)	'`-bI:`*`runtime-directory`*`/prolog.exp -e QP_entry glue.o`'
Windows	`cl`	'`/MD`'	'`/MD`'	'`/dll`'

To build a shared object file, say '`mylib.so`', under UNIX, issue the following:

```
% CC SFLAGS -c SOURCE1
% ...
% CC SFLAGS -c SOURCEn
% ld -o mylib.so LFLAGS OBJECTS
```

When you build a shared object file, say '`mylib.dll`', under Windows, a corresponding import library will normally also be built. Consult the Windows documentation for details. Issue the following:

```
C:\> cl SFLAGS /c SOURCE1
C:\> ...
C:\> cl SFLAGS /c SOURCEn
C:\> link /dll /out:mylib.dll OBJECTS
```

To build a archive file, say '`mylib.a`', under UNIX, issue the following:

```
% CC AFLAGS -c SOURCE1
% ...
% CC AFLAGS -c SOURCEn
% ar r mylib.a OBJECTS
% ranlib mylib.a
```

For archive files under Windows, a special naming convention is used: an extra '`s`' is placed before the '`.lib`' extension, to distinguish archive files from import libraries (see Section 9.2.5 [sap-rge-sos], page 359). To build an archive file, say '`mylibs.lib`', issue the following:

```
C:\> cl AFLAGS /c SOURCE1
C:\> ...
C:\> cl AFLAGS /c SOURCEn
C:\> link /lib /out:mylibs.lib OBJECTS
```

Platform specific notes:

`alpha` Foreign code must be compiled in native mode, i.e. *not* using the '`-xtaso_short`' option.

`rs6000` You must supply the file '`glue.o`'. See '*`QuintusDir`*`/generic/qplib3.5/structs/library/rs6000/structs_lnk.c`' for an example.

Windows

If building a DLL that calls any of the Quintus Prolog C API functions exported from the Runtime Kernel DLL ('`qpeng.dll`') or the Embedding Layer DLL ('`libqp.dll`') then you must also link in the import libraries for these DLLs, which are named '`qpeng.lib`' and '`libqp.lib`' and reside in the directory '*`quintus-directory`*`\lib\ix86`'. This is needed because DLLs must have their external references resolved at link time rather than at load time, in contrast to typical UNIX shared library implementations. Run the '`qpvars.bat`' file to set-up the environment variables necessary for the C-compiler and linker to fine the needed Quintus files.

The '`makefile.win`' file in the Quintus Library directory '*`quintus-directory`*`\src\library`' contains an example of how to build a DLL for dynamic loading into Quintus Prolog.

The following two sections describe the use of shared object files in Quintus Prolog:

10.3.2.1 Loading Foreign Executables

The built-in Prolog predicate `load_foreign_executable/1` is used to load foreign functions directly into Prolog from a shared object file and to attach selected functions and routines in the loaded file to Prolog predicates.

The example below demonstrates the use of these predicates to load code compiled using the C compiler.

In the above example, '`foreign`' (or '`foreign.pl`') is a file containing facts that describe how Prolog is to call the foreign functions. If it is given a filename without an extension then it automatically appends the appropriate extension; thus in the example above, '`math`' is specified to load '`math.so`'.

The loading process may fail if:

- the facts in the database (see Section 10.3.3 [fli-p2f-lnk], page 380) that describe how to link foreign functions to Prolog procedures are incomplete;
- the foreign functions specified have already been loaded;
- the shared object file contains undefined symbols that could not be resolved when loaded.

If the load does not complete successfully then an exception is raised and the call to `load_foreign_executable/1` fails; no change is made to the Prolog state. The load can be retried once the problem has been corrected.

Once a foreign program is loaded, it cannot be unloaded or replaced, although you can abolish or redefine any procedure attached to it.

Notes:

1. Any foreign file loaded via a `load_foreign_executable/1` command that is embedded in a file being loaded into Prolog will be sought relative to the directory from which the file is being loaded. For example, if the file '`/usr/fred/test.pl`' contains the command

   ```
   :- load_foreign_executable(test).
   ```

   then the file to be loaded would be '`/usr/fred/test.so`'.
2. When the linker is given a library such as '`-lX11`', it will look for a "shared library" version and if one exists record this library as a dependency in the shared object file. `load_foreign_executable/1` will then automatically load this library (if not already loaded) when it loads the shared object file.

   If the linker is given a library for which no shared library exists, then object files from the static library are incorporated into the shared object file as needed. This means that any routine in a static library that is to be accessed from Prolog must have some reference to it in one of the object files being linked into the shared object file.
3. It is better to load one large shared object file than many small ones. You may have several Prolog files that require routines from one shared object file — in other words, a shared library. The shared library is only loaded once, but different functions could be attached to Prolog predicates in different calls to `load_foreign_executable/1`. For example, under

UNIX, most of the files in the Prolog Library that load foreign code use the shared library file '`libpl.so`'.

A description of the internal operation of the `load_foreign_executable/1` predicate is given in Section 10.3.12 [fli-p2f-lfe], page 401 to help solve more difficult foreign code loading problems.

10.3.2.2 Loading Foreign Files

`load_foreign_files/2` is an alternative interface to `load_foreign_executable/1`, which constructs a shared object file from the list of object files and libraries given as its arguments and then maps the resulting shared object file into the Prolog address space. In general it is recommended that you take the responsibility for the construction of a shared object file and then use `load_foreign_executable/1` directly. Using `load_foreign_files/2` is slower because it has to invoke the linker to construct the shared object file everytime the program is loaded. A further disadvantage is that the linker may not be available at runtime on all systems.

Example:

```
% cc -c CFLAGS math.c
% cc -c CFLAGS other.c
% prolog

| ?- compile(foreign).
| ?- load_foreign_files([math,other],['-lm']).
```

Again, the file '`foreign`' (or '`foreign.pl`') contains the facts that describe how Prolog is to call the foreign functions. If the extensions on filenames given in the first argument to `load_foreign_files/2` are omitted, the proper extension is automatically appended to them.

10.3.3 Linking Foreign Functions to Prolog Procedures

When `load_foreign_executable/1` or `load_foreign_files/2` is called, it calls the hook predicates `foreign_file/2` and `foreign/3` in the current source module. These should have been previously defined by clauses of the form:

```
foreign_file(FileName, [Function1,Function2,...,FunctionN]).

foreign(Function1, Language, PredicateSpecification1).
foreign(Function2, Language, PredicateSpecification2).
...
foreign(FunctionN, Language, PredicateSpecificationN).
```

Example:

```
foreign_file(math, [sin,cos,tan]).

foreign(sin, c, sin(+float,[-float])).
foreign(cos, c, cos(+float,[-float])).
foreign(tan, c, tan(+float,[-float])).
```

Please note: If a Prolog module includes foreign code, all relevant `foreign/[2,3]` and `foreign_file/2` facts should be loaded into that module and the `load_foreign_executable/1` or `load_foreign_files/2` command should be called from that module.

A `foreign_file/2` fact lists the functions that will be provided by the associated (shared) object file. When using `load_foreign_files/2`, a fact of this form must be provided for each file specified in the *ListOfFiles* argument. The functions specified should be only those that are to be attached to Prolog procedures. Supporting functions that will not be called directly from Prolog should not be listed.

Each `foreign/3` fact describes how a foreign function is to be attached to a Prolog procedure. *PredicateSpecification* specifies the Prolog procedure and also the argument passing interface (described below). A fact of this form must be provided for each function that is to be attached to a Prolog procedure.

When `load_foreign_executable/1` or `load_foreign_files/2` is called, the specified files are loaded into the running Prolog and then all the specified Prolog procedures are abolished and redefined to be links to the foreign functions. Calling one of the Prolog procedures now results in a call to a foreign function.

Prolog procedures can be directly linked to library functions. Note, however, that some functions shown in the library documentation are actually C macros (found in included '`.h`' files). In this case, the simplest approach is to write a C function that uses the macro and then link to that function.

You may abolish or redefine (using `compile/1`) any procedure that has been attached to a foreign function. This severs the link between the Prolog predicate and the foreign function. It is not possible to reestablish this link.

The `foreign_file/2` and `foreign/3` facts must be consistent whenever `load_foreign_executable/1` or `load_foreign_files/2` is called. They are, however, not used after this point and may be abolished, if desired.[1].

The `load_foreign_executable/1` and `load_foreign_files/2` commands can be used any number of times in a Prolog session to load different foreign programs. For example:

[1] See example in the reference page for `foreign/[2,3]`.

```
| ?- compile(f1),
     load_foreign_executable(f1),
     abolish([foreign/3, foreign_file/2]).
| ?- compile(f2),
     load_foreign_executable(f2),
     abolish([foreign/3, foreign_file/2]).
```

Each `compile/1` installs a new set of facts describing a set of functions to be loaded by `load_foreign_executable/1`. Unless you abolish all `foreign/3` and `foreign_file/2` facts before each compilation, Prolog will warn you that `foreign/3` and `foreign_file/2` have been previously defined in another file.

A better way to do this is to insert the call to `load_foreign_executable` into the file that defines `foreign_file/2` and `foreign/3` as an embedded command. For example, you could add the following command to the end of the file '`f1.pl`':

```
:- load_foreign_executable(f1),
   abolish([foreign/3, foreign_file/2]).
```

so that compiling '`f1.pl`' will automatically load '`f1.so`'. This embedded command will also work when building a stand-alone program, as described in Section 9.1 [sap-srs], page 337.

10.3.4 Specifying the Argument Passing Interface

The argument passing interface is specified by defining facts for `foreign/3` of the form:

```
foreign(+Routine, +Language, +PredicateSpecification)
```

Routine is an atom that names a foreign code routine and *Language* is an atom (either `c`, `pascal`, or `fortran`) that names the language in which the routine is written.

> **Please note:** Assembly code can be loaded if it emulates the exact calling conventions of one of C, FORTRAN, or Pascal. *Language* is then chosen, accordingly, to be one of `c`, `fortran`, or `pascal`.

PredicateSpecification specifies the Prolog name given to the foreign code routine and how its arguments will be passed to and from the foreign code routine.

PredicateSpecification is of the form:

```
PredicateName(ArgSpec1, ArgSpec2, ...ArgSpecN)
```

where *PredicateName* is the name of the Prolog predicate (an atom) and each *ArgSpec* is an argument specification for each argument of the predicate. An *ArgSpec* informs the Prolog system how to pass or receive a Prolog term in the corresponding argument position.

Prolog checks the types of the input arguments; a foreign function call will raise an exception if any input argument is not of the right type.

If the argument passed is an atomic object then the interface automatically converts between Prolog's representation of the data and the representation expected by the foreign function. Thus the external function does not need to know how Prolog represents atoms, integers, or floats in order to communicate with Prolog. This feature simplifies the integration of foreign code with Prolog; in particular, it makes it easier to interface directly with already-written functions in libraries and other programs. It also allows for compatibility with later versions of Quintus Prolog and with versions of Quintus Prolog running on other hardware.

> **Please note:** The only atomic object that cannot be passed directly through the foreign interface is a db_reference. db_references can be passed to foreign code using the general term passing mechanism using `+term` and `-term`. You can take apart and build db_references in foreign language using the `QP_get_db_reference()` and `QP_put_db_reference()` functions.

On the other hand, generic Prolog terms passed to a foreign function (using `+term`) are not converted to any representation in the foreign language. Instead the foreign function gets a reference to a Prolog term. A set of functions/macros is provided to type test, access and create Prolog terms through these references (see Chapter 19 [cfu], page 1345). Similarly when a generic term is returned (using `-term` or `[-term]`) from a foreign function there is no conversion of any data structures in the foreign language into an equivalent Prolog representation. The foreign function has to return a reference to a Prolog term, which it originally got from Prolog or from one of the functions/macros provided to manipulate Prolog terms (the `QP_put*` and `QP_cons*` families of functions) Further details in Section 10.3.8 [fli-p2f-trm], page 395.

Arguments are passed to foreign functions in the same order as they appear in the Prolog call, except for the return value. At most one "return value" argument can be specified; that is, there can be only one `[-integer]`, `[-float]`, `[-atom]`, `[-string]`, `[-string(`*N*`)]`, or `[-address(`*typename*`)]` specification. There need not be any "return value" argument, in which case the value returned by the function is ignored. Both input and output specifications cause data to be passed to the foreign function (except of course for the "return value" argument, if present). Each input argument is appropriately converted and passed, by reference or by value, depending on the language's calling convention, and each output argument is passed as a pointer through which the foreign function will send back the result. Note that for C, input arguments are always passed by value.

Prolog assumes that a foreign function will return output arguments of the specified types; if it does not, the result is unpredictable. Normally, unbound variables will be supplied in the Prolog goal for all the output argument positions. However, any value may be supplied for an output argument; when the foreign function has been completed, its outputs are unified with the values supplied and a failure to unify results in the failure of the Prolog goal.

Detailed information about passing particular data types through the foreign interface can be found in Section 10.3.5 [fli-p2f-int], page 384 through Section 10.3.9 [fli-p2f-poi], page 397. Examples showing the correct use of the foreign interface are presented in Section 10.3.15 [fli-p2f-fex], page 402.

10.3.5 Passing Integers

In previous releases of Quintus Prolog, integers were represented with 29 bits. Since most languages represent integers using 32 bits, errors could occur when passing very large positive or negative numbers between Prolog and foreign functions. In release 3 and later releases of Quintus Prolog the precision of integers has been raised to 32 bits; hence, these errors no longer occur.

> **Please note:** Quintus Prolog Release 3 uses 32 bit integers and pointers internally. On 64 bit platforms, long integers and pointers are truncated to 32 bits when they are passed to Prolog, and sign-extended when passed in the other direction.

10.3.5.1 Passing an Integer to a Foreign Function

```
Prolog:  +integer
C:       long int x
Pascal:  x: integer
FORTRAN: integer x
```

The argument must be instantiated to an integer, otherwise the call will raise an exception. The Prolog integer is converted to a long integer and passed to the foreign function.

10.3.5.2 Returning an Integer from a Foreign Function

```
Prolog:  -integer
C:       long int *x;
         *x = ...
Pascal:  var x: integer
         x := ...
FORTRAN: integer x
         x = ...
```

A pointer to a long integer is passed to the function. It is assumed that the function will overwrite this integer with its result. When the foreign function returns, the integer being pointed to is converted to a Prolog integer and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, the call fails. If the foreign function does not overwrite the integer, the result is undefined.

10.3.5.3 An Integer Function Return Value

```
Prolog:  [-integer]
C:       long int f(...)
           {
             long int x;
             return x;
           }
Pascal:  function f(...): integer;
           var x: integer;
           begin
             f := x;
           end
FORTRAN: integer function f(...)
           integer x
           f = x
         end
```

No argument is passed to the foreign function. The return value from the function is assumed to be a long integer. It is converted to a Prolog integer and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, the call fails.

10.3.6 Passing Floats

In previous releases of Quintus Prolog, floating-point numbers had less precision than single-precision floats in other languages. The result was a loss of precision when floats were passed between Prolog and foreign functions. In release 3 and later releases of Quintus Prolog the precision of floating-point numbers has been raised to 64 bits (double precision); hence, these errors no longer occur.

Quintus Prolog release 3 also supports the passing of floating point numbers explicitly as doubles or singles.

10.3.6.1 Passing a Float to a Foreign Function

```
Prolog:  +float
C:       double x;
Pascal:  x: real
FORTRAN: real x
```

The argument must be instantiated to an integer or a float; otherwise the call will raise an exception. The Prolog number is converted to a 32-bit single-precision (FORTRAN) or a 64-bit double-precision (C or Pascal), float and passed to the foreign function. Many C compilers will allow the parameter declaration to be `float` instead of `double` because they always convert single-precision floating-point arguments to double-precision. However, C

compilers conforming to the new ANSI standard will not do this, so it is recommended that `double` be used.

```
Prolog:  +single
ANSI C:  float x;
FORTRAN: real x
```

The argument must be instantiated to an integer or a float; otherwise the call will raise an exception. The Prolog number is converted to a 32-bit single-precision float and passed to the foreign function. `+single` can also be used to interface Prolog to any foreign function where you know that the value passed is going to be picked up as a 32-bit float.

```
Prolog:  +double
C:       double x;
Pascal:  real x
```

The argument must be instantiated to an integer or a float; otherwise the call will raise an exception. The Prolog number is converted to a 64-bit double-precision (C or Pascal) float and passed to the foreign function. `+double` can also be used to interface Prolog to any foreign function where you know that the value passed is going to be picked up as a 64-bit float.

10.3.6.2 Returning a Float from a Foreign Function

```
Prolog:  -float
C:       float *x;
         *x = ...
Pascal:  var x: real
         x := ...
FORTRAN: real x
         x = ...
```

A pointer to a single-precision float is passed to the function. It is assumed that the function will overwrite this float with its result. When the foreign function returns, the float is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails. If the foreign function does not overwrite the float, the result is undefined.

```
Prolog:  -single
C:       float *x;
         *x = ...
Pascal:  var x: real
         x := ...
FORTRAN: real x
         x = ...
```

A pointer to a single-precision float is passed to the function. It is assumed that the function will overwrite this float with its result. When the foreign function returns, the float is

converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails. If the foreign function does not overwrite the float, the result is undefined.

```
Prolog:  -double
C:       double *x;
         *x = ...
Pascal:  var x: real
         x := ...
```

A pointer to a double-precision float is passed to the function. It is assumed that the function will overwrite this float with its result. When the foreign function returns, the float is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails. If the foreign function does not overwrite the float, the result is undefined.

10.3.6.3 A Floating-point Function Return Value

```
Prolog:  [-float]
C:       double f(...)
           {
             double x;
             return x;
           }
Pascal:  function f(...): real;
           var x: real;
           begin
             f := x;
           end
FORTRAN: real function f(...)
           real x
           f = x
         end
```

No argument is passed to the foreign function. The return value from the function is assumed to be a single-precision (FORTRAN) or double-precision (C and Pascal) floating point number. It is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails.

Many C compilers will allow the function return value to be `float` instead of `double` because they always convert single-precision floating-point arguments to double-precision. However, C compilers conforming to the new ANSI standard will not do this, so it is recommended that `double` be used.

```
Prolog:  [-single]
ANSI C:  float f(...)
           {
             float x;
             return x;
           }
Pascal:  function f(...): real;
           var x: real;
           begin
             f := x;
           end
FORTRAN: real function f(...)
           real x
           f = x
         end
```

No argument is passed to the foreign function. The return value from the function is assumed to be a single-precision (FORTRAN) or double-precision (C and Pascal) floating point number. It is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails.

```
Prolog:  [-double]
C:       double f(...)
           {
             double x;
             return x;
           }
Pascal:  function f(...): real;
           var x: real;
           begin
             f := x;
           end
FORTRAN: real function f(...)
           real x
           f = x
         end
```

No argument is passed to the foreign function. The return value from the function is assumed to be a single-precision (FORTRAN) or double-precision (C and Pascal) floating point number. It is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float, the call fails.

10.3.7 Passing Atoms

The foreign function interface allows Prolog atoms to be passed to functions either in a canonical form as unsigned integers, or as pointers to character strings.

For each Prolog atom there is a single canonical representation. Programs can rely on the property that identical atoms have identical canonical representations. Note, however, that the canonical form of an atom is not necessarily identical across different invocations of the program. This means that canonical atom representations should not be used in files or interprogram communication. For these purposes strings should be used. Foreign functions can store canonical atoms in data structures and pass them around and then back to Prolog, but they should not attempt any other operations on them.

Strings passed from Prolog to foreign functions should not be overwritten. Strings passed back from foreign functions to Prolog are automatically copied by Prolog if necessary. Thus the foreign program does not have to retain them and can reuse their storage space as desired.

There are three ways of passing atoms through the foreign interface:

1. as canonical integers (to or from any language): see Section 10.3.7.1 [fli-p2f-atm-cat], page 389.
2. as null-terminated strings (to or from C): see Section 10.3.7.2 [fli-p2f-atm-spc], page 390.
3. as fixed-length, blank-padded strings (to or from FORTRAN or Pascal): see Section 10.3.7.2 [fli-p2f-atm-spc], page 390.

10.3.7.1 Passing Atoms in Canonical Form

This section deals with passing atoms in canonical form, that is, as unsigned integers.

```
Prolog:  +atom
C:       QP_atom x;
Pascal:  x: integer
FORTRAN: integer x
```

The argument must be instantiated to an atom, otherwise the call will signal an error. An unsigned integer representing the Prolog atom is passed to the foreign function. Atoms can be converted to strings through the functions `QP_string_from_atom()` or `QP_padded_string_from_atom()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393).

```
Prolog:  -atom
C:       QP_atom *x
         *x = ...
Pascal:  var x: integer
         x := ...
FORTRAN: integer x
         x = ...
```

A pointer to an unsigned integer is passed to the function. It is assumed that the function will overwrite this unsigned integer with its result. This result should be a canonical representation of an atom already obtained from Prolog, or one generated through the function `QP_atom_from_string()` or the function `QP_atom_from_padded_string()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393). Returning an arbitrary integer will have undefined results. When the foreign function returns, the atom represented by the unsigned integer being pointed to is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, the call fails. If the foreign function does not overwrite the unsigned integer, the result is undefined.

```
Prolog:  [-atom]
C:       QP_atom f(...)
           {
             QP_atom x;
             return x;
           }
Pascal:  function f(...): integer;
           var x: integer;
           begin
             f = x;
           end
FORTRAN: integer function f(...)
           integer x
           f = x
         end
```

No argument is passed to the foreign function. The return value from the function is assumed to be an unsigned integer, which should be a canonical representation of an atom already obtained from Prolog, or one generated by one of the functions `QP_atom_from_string()` or `QP_atom_from_padded_string()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393). Returning an arbitrary integer will have undefined results. The atom represented by the unsigned integer is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, the call fails.

10.3.7.2 Passing Atoms as Strings between Prolog and C

This section describes passing atoms as pointers to null-terminated character strings. This is the way to pass atoms as strings between Prolog and C. For FORTRAN and Pascal, the

specification of string arguments is different than for C; see Section 10.3.7.3 [fli-p2f-atm-spf], page 392.

```
Prolog:  +string
C:       char *x
Pascal:  Not supported
FORTRAN: Not supported
```

The argument must be instantiated to an atom, otherwise the call will signal an error. A pointer to a null-terminated string of characters is passed to the C function. This string must not be overwritten by the C function.

```
Prolog:  -string
C:       char **x;
         *x = ...
Pascal:  Not supported
FORTRAN: Not supported
```

A pointer to a character pointer is passed to the C function. It is assumed that C will overwrite this character pointer with the result it wishes to return. This result should be a pointer to a null-terminated string of characters. When the C function returns, the atom that has the printed representation specified by the string is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, then the call fails. If the C function does not overwrite the character pointer, then the result is undefined.

Prolog copies the string if required, so that it is not necessary for the C program to worry about retaining it. Beware, however, that the string must not be an `auto`, because in this case its storage may be reclaimed after the foreign function exits but before Prolog has managed to copy it.

```
Prolog:  [-string]
C        char *f(...)
           {
             char *x;
             return x;
           }
Pascal:  Not supported
FORTRAN: Not supported
```

No argument is passed to C. The return value from the C function is assumed to be a character pointer pointing to a null-terminated string of characters. The atom that has the printed representation specified by the string is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, the call fails.

Prolog copies the string if required, so that it is not necessary for the C program to worry about retaining it. Beware, however, that the string must not be an `auto`, because in this

case its storage may be reclaimed after the foreign function exits but before Prolog has managed to copy it.

10.3.7.3 Passing Atoms as Strings to/from Pascal or FORTRAN

This section describes passing atoms as pointers to fixed-length, blank-padded arrays of characters. This is the way to pass atoms as strings between Prolog and Pascal or FORTRAN. See Section 10.3.7.2 [fli-p2f-atm-spc], page 390 for how to pass atoms as null-terminated strings between Prolog and C.

> **Implementation note:** The foreign interface makes some assumptions about how string parameters are handled in Pascal and FORTRAN compilers. If a given Pascal or FORTRAN compiler has different conventions for the handling of string parameters, the interface will not work. The conventions are:
>
> - A string result is represented by a pointer to the character array followed by its size. These two values are passed before all the other arguments.
> - Other string parameters are also represented by a pointer to the characters and a size. In this case the pointer occupies the normal position in the argument list, and the size is passed after all the other arguments.

```
Prolog:  +string(N)
C:       Not supported
Pascal:  type stringN = packed array [1..N] of char;
         var x: stringN
FORTRAN: character*N
```

The argument must be instantiated to an atom, otherwise the call will signal an error. A character array, containing a copy of the characters of the atom, is passed by reference to the function. The text is truncated on the right or padded on the right with blanks to length *N*.

Note that the Pascal parameter is call-by-reference (var), the same as for the `-string(N)` case below.

```
Prolog:  -string(N)
C:       Not supported
Pascal:  type stringN = packed array [1..N] of char;
         var x: stringN
FORTRAN: character*N
```

A pointer to a character array of length *N*, initialized to all blanks, is passed to the function. It is assumed that the function will fill in this array. When the function returns, the atom that has the printed representation specified by the character array is unified with the corresponding argument of the Prolog call.

Trailing blanks in the character array are ignored. Thus if the foreign function sets a character array of length 6 to '`atom `', Prolog will convert the result to the atom `atom`. Leading blanks are significant: if the foreign function returns ' `this` ', the resulting atom is `' this'`.

The argument can be of any type; if it cannot be unified with the returned atom, then the call fails. If the function does not fill in the character array, then the result is the null atom `''`.

```
Prolog:  [-string(N)]
C:       Not supported
Pascal:  Not Supported
FORTRAN: character*N function
```

This argument specification is valid only for FORTRAN. The FORTRAN function result is initialized to a blank-filled character array of length *N*. It is assumed that the function will fill this array. The atom that has the printed representation specified by the character array is unified with the corresponding argument of the Prolog call.

Trailing blanks in the character array are ignored, as for the `-string(N)` case above.

The argument can be of any type; if it cannot be unified with the returned atom, then the call fails. If the function does not fill in the character array, then the result is the null atom `''`.

10.3.7.4 Converting between Atoms and Strings

Four functions are provided to enable foreign functions to translate from one representation of an atom to another. The first two functions are most useful for C: they convert between canonical atoms and null-terminated C strings. The other two functions are most useful for Pascal and FORTRAN: they convert between canonical atoms and blank-padded character arrays.

`QP_string_from_atom(atom)`

`atom` `QP_atom` (that is, an unsigned integer passed by value)

Returns: Pointer to a null-terminated string of characters (C convention for strings)

Returns a pointer to a string representing `atom`. This string should not be overwritten by the foreign function.

`QP_atom_from_string(string)`

`string` Pointer to a null-terminated string of characters (C convention for strings)

Returns: `QP_atom`

Returns the canonical representation of the atom whose printed representation is `string`. The string is copied, and the foreign function can reuse the string and its space.

```
QP_padded_string_from_atom(pointer_to_atom, pointer_to_padded_string,
pointer_to_length)
```

`pointer_to_atom`
: Pointer to a `QP_atom` (that is, an unsigned integer passed by reference)

`pointer_to_padded_string`
: Pointer to a character array

`pointer_to_length`
: Pointer to an integer (that is, an integer passed by reference)

Returns: integer

Fills in the character array of length `*pointer_to_length` with the string representation of the atom. The string is truncated or blank-padded to `*pointer_to_length` if the length of the atom is greater than or less than `*pointer_to_length`, respectively. The length of the atom (not `*pointer_to_length`) is returned as the function value.

```
QP_atom_from_padded_string(pointer_to_atom, pointer_to_padded_string,
pointer_to_length)
```

`pointer_to_atom`
: Pointer to a `QP_atom` (that is, an unsigned integer passed by reference)

`pointer_to_padded_string`
: Pointer to a character array

`pointer_to_length`
: Pointer to an integer (that is, an integer passed by reference)

Returns: integer

Sets `*pointer_to_atom` to the canonical representation of the atom whose printed representation is the string (less any trailing blanks) contained in the character array of length `*pointer_to_length`. Returns the length of the resulting atom (not `*pointer_to_length`) as the function value.

Below are C specifications of these functions. Note that the arguments of the last two functions are passed by reference. Hence, the last two functions can be called directly from Pascal or FORTRAN. The first two functions are designed to be called from C, in which all parameters are passed by value.

```
char * QP_string_from_atom(atom)
     QP_atom atom;

QP_atom QP_atom_from_string(string)
     char *string;

int QP_padded_string_from_atom(atom,string,length)
     QP_atom *atom;
     char *string;
     int *length;

int QP_atom_from_padded_string(atom,string,length)
     QP_atom *atom;
     char *string;
     int *length;
```

Canonical atoms are particularly useful as constants, to be used in passing back results from foreign functions. The above functions can be used to initialize tables of such constants.

These functions can only be called from languages other than C if those languages have a C-compatible calling convention for passing integers and pointers. For example, this is true for both Pascal and FORTRAN running under UNIX 4.2 BSD. See the appropriate Quintus Prolog Release Notes for any further details pertaining to your system.

10.3.8 Passing Prolog Terms

This section describes passing Prolog terms to a foreign function and receiving Prolog terms from a foreign function. For the current release this interface is supported only for C.

There is a difference between passing atomic objects (atoms, floats, db_reference and integers) and generic Prolog terms through the foreign interface. Generic Prolog terms passed to a C function (using `+term`) are not converted to any representation in C. Instead the foreign function in C gets a reference to the Prolog term, which is of type `QP_term_ref` (defined in '`<quintus/quintus.h>`'). Similarly when a generic term is returned (using `-term` or `[-term]`) from a foreign function there is no conversion of any data structures in the foreign language into an equivalent Prolog representation. The foreign function has to return a reference to a Prolog term, which it originally got from Prolog or from one of the functions/macros provided to manipulate Prolog terms such as the `QP_put*` and `QP_cons*` families of functions.

When Prolog terms are referred to from C, what the C function holds is an indirect reference to the Prolog term. There is a reason for this indirection. Prolog terms live in the Prolog global stack, and migrate when Prolog does garbage collection or stack shifting. If the C function held onto a direct reference to a Prolog term it would become invalid after one of these memory management operations. Prolog cannot update and relocate these references

that C is holding onto since it is impossible to distinguish between Prolog references and other integers and pointers that C holds onto.

The C code should also be aware of the scope (or lifetime) of the references to Prolog terms passed to it. Once you return to Prolog from a call to a foreign function, all the references to Prolog terms passed to the foreign function are invalid. All references to terms created by the foreign function are also invalid.

> *WARNING:* You should not store references to prolog terms into global variables in the foreign language.

The scope of references to terms are more restricted when C calls Prolog. If Prolog returns a term as a result of a C call to a Prolog predicate, that term is valid only till the call for the next solution from that Prolog predicate (using `QP_next_solution()`). This also holds true for terms created in C. If you create a term after one call to a Prolog predicate then the reference to that term is only valid till the call for next solution from that Prolog predicate. .

10.3.8.1 Passing a Prolog term to a Foreign Function

```
Prolog:  +term
C:       QP_term_ref
```

The argument can be any Prolog term. The C function gets an object of type `QP_term_ref` (defined in '`<quintus/quintus.h>`'). `QP_term_type()` and associated functions can be used to test the type of the term. And the `QP_get()` functions can be used to access the value associated with the term. `QP_unify()` can be used to used to unify terms or subterms of terms passed to C.

10.3.8.2 Returning a Prolog term from a Foreign Function

```
Prolog:  -term
C:       QP_term_ref x;
```

An initialized `QP_term_ref` (defined in '`<quintus/quintus.h>`') is passed to the C function. It is assumed that the function will assign a term to this `QP_term_ref` using one of the `QP_put()` functions. When the foreign function returns, the term that the `QP_term_ref` refers to is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the referred term, the call will fail.

10.3.8.3 A Prolog term returned as a value of a Foreign Function

```
Prolog:  [-term]
C:       QP_term_ref f(...);
           {
                QP_term_ref ref = QP_new_term_ref();
                return ref;
           }
```

No argument is passed to the foreign function. The return value from the function is assumed to be a reference to a Prolog term of type `QP_term_ref`. The term that the `QP_term_ref` refers to is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the referred term, the call will fail.

10.3.9 Passing Pointers

Pointers should be passed through the foreign interface using the specification

`address(`*typename*`)`

They could also be passed as integers, but there are two added advantages for using the address specification. The first is that a stand-alone tool could check for consistency between the foreign declarations and the foreign code. The second advantage is for possible optimizations on platforms whose pointers require more than 29 bits.

The *typename* is there so that a stand-alone tool could know what kind of argument to pass or what kind of result to demand and *typename* should be the name used in the foreign language to identify the type of object named by the pointer. It is sufficiently important to be able to check the `foreign/3` declarations that Prolog will issue a warning if the *typename* is not an atom, but it makes no other use of the *typename*. The *typename* can even be omitted entirely, using `address` as an argument specification.

is the argument type desired.]

Note that programs should not rely on numeric relations between foreign language pointers being true of the Prolog integers to which they are converted.

See Section 10.3.15.4 [fli-p2f-fex-poi], page 410 for an example of passing pointers through the foreign interface. For further examples, see `library(charsio)` and `library(vectors)`.

```
Prolog:  +address(typename)
C:       typename *x
Pascal:  type ptr = ^typename;
         x: ptr
FORTRAN: typename x(*)
```

The argument must be instantiated to an integer, otherwise the call fails. If the argument is 0, the foreign function will receive the NULL pointer. Otherwise the argument will be

converted to a pointer. The coding is system-dependent. All you can rely on is that NULL and "`malloc()` pointers" can be passed from the foreign language to Prolog and that Prolog can then pass the same pointers back to the foreign language.

FORTRAN programmers will note that `+address(integer)` and `+address(float)` parameters are useful for passing arrays to FORTRAN, but since FORTRAN has no pointer data type (and no equivalent of `malloc(3)`), address results are not possible. Therefore arrays cannot be constructed in FORTRAN and then passed to Prolog; they must be constructed in C or Pascal. Section 10.3.15.4 [fli-p2f-fex-poi], page 410 gives an example where arrays are constructed in C and later passed to a FORTRAN routine.

The *typename* must be an atom, but is otherwise ignored by Prolog. It is present for the benefit of stand-alone tools, which could check

that your Prolog `foreign/3` facts are compatible with your C source files.

```
Prolog:  -address(typename)
C:       typename **x;
         *x = ...
Pascal:  type ptr = ^typename;
         var x: ptr;
         x = ...
FORTRAN: Not supported
```

A pointer to a pointer is passed to the foreign function. It is assumed that the function will overwrite this variable with the result it wishes to return. This result should be either the NULL pointer or a `malloc()` pointer. When the function returns, the result is converted to a Prolog integer, which is then unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, the call fails. If the foreign function does not set the result, the result is undefined.

The *typename* must be an atom, but is otherwise ignored by Prolog. It is present for the benefit of stand-alone tools, which could check

that your Prolog `foreign/3` facts are compatible with your C source files.

```
Prolog:  [-address(typename)]
C:       typename *f(...)
           {
              typename *x;
              return x;
           }
Pascal:  type ptr = ^typename;
         function f(...): ptr;
           var x: ptr;
           begin
             f := x;
           end
FORTRAN: Not supported
```

No argument is passed to the foreign function. The return value from the foreign function is assumed to be a pointer to an object of the type indicated by *typename*. This pointer should be either NULL or a `malloc()` pointer. It is converted to a Prolog integer, which is then unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, the call fails.

The *typename* must be an atom, but is otherwise ignored by Prolog. It is present for the benefit of stand-alone tools, which could check

that your Prolog `foreign/3` facts are compatible with your C source files.

```
Prolog:  +address
C:       char *x
Pascal:  type charp = ^char;
         x: charp
FORTRAN: Not supported
```

This is equivalent to `+address(char)` (see `+address(`*typename*`)` above). Note that `+address(char)` is not useful in FORTRAN because FORTRAN will not accept a pointer to a character array as representing that array. Therefore `+address` is not allowed in FORTRAN. To pass a character array to FORTRAN use the `+string(`*N*`)` argument type as described in Section 10.3.7.3 [fli-p2f-atm-spf], page 392.

```
Prolog:  -address
C:       char **x
         *x = ...
Pascal:  type charp = ^char;
         var x: charp;
         x = ...
FORTRAN: Not supported
```

This is equivalent to `-address(char)` (see `-address(`*typename*`)` above).

```
Prolog:  [-address]
C:       char *f(...)
           {
             char *x;
             return x;
           }
Pascal:  type charp = ^char;
         function f(...): charp;
           var x: charp;
           begin
             f := x;
           end
FORTRAN: Not supported
```

This is equivalent to `[-address(char)]` (see `[-address(`*typename*`)]` above).

10.3.10 Important Prolog Assumptions

For information about memory allocation, see the discussion of `PROLOGINITSIZE`, `PROLOGMAXSIZE` and `PROLOGINCSIZE` in Section 8.12 [ref-mgc], page 256.

10.3.11 Debugging Foreign Code Routines

In order to debug foreign code in conjunction with Prolog code, it is necessary to statically link your program together with the Development Kernel as discussed in Section 9.1 [sap-srs], page 337. The resulting executable can then be debugged using any standard debugger, such as `gdb(1)`.

Note that it is often useful for debugging purposes to build an application that is linked with QUI, since then both the Prolog and the non-Prolog parts of the application can be debugged simultaneously, using the QUI debugger and the standard debugger respectively. See Section 9.1.10 [sap-srs-qui], page 354 for how to do this. If you do this, you may find that the standard debugger gets affected by the way that QUI uses the `SIGIO` signal. Most standard debuggers provide a way of ignoring specified signals, which is what is needed here. For example, under `gdb(1)` the command '`handle SIGIO noprint nostop pass`' should be issued before starting up the QUI with the '`run`' command.

> *WARNING:* Under source-level debuggers such as `gdb(1)`, single stepping out of a function that was called from Prolog does not work properly. You should always '`continue`' in such a situation.

10.3.12 Implementation of `load_foreign_executable/1`

This section gives some information on the implementation of `load_foreign_executable/1`, which may help in solving more difficult foreign code loading problems. This information applies when the foreign code is being loaded dynamically on top of the Development System. Refer to Section 9.1 [sap-srs], page 337 for information on how foreign code is linked into a stand-alone program.

`load_foreign_executable/1` loads a shared object file by calling the library function `dlopen(3)` (UNIX) or `LoadLibrary()` (Windows). This automatically loads any shared libraries that are stored as dependencies in the file.

10.3.13 Implementation of `load_foreign_files/2`

`load_foreign_files/2` is implemented by constructing a shared object file and then using the same mechanism to load the shared object file as for `load_foreign_executable/1`. Under UNIX, the shared object file is constructed by the linker with a command similar to:

```
% ld -G -o /tmp/qpnnnn.so LinkFile ListOfFiles ListOfLibraries -lc
```

Under Windows, the command is similar to:

```
C:\> link -dll -out:C:\tmp\qpnnnn.dll LinkFile ListOfFiles
ListOfLibraries qpeng.lib libqp.lib -defaultlib:msvcrt
```

If any libraries are specified in *ListOfLibraries* then a *LinkFile* is generated that references all routines to be accessed by Prolog so that, if any of the specified libraries are static libraries, all the relevant object files will be included in the shared object file. In many cases no additional libraries are required and so *ListOfLibraries* = [] and no *LinkFile* is generated.

Note that when using languages other than C, various specific libraries may need to be included (such as '`-lpc`' or '`-lF77`').

10.3.14 Library support for linking foreign code

The Structs and Objects packages (see Chapter 13 [str], page 655 and Chapter 14 [obj], page 665) allow Prolog to hold pointers to C data structures and arrays and access and store into fields in those data structures in a very efficient way.

Support for translating between Prolog terms and C data structures is provided by `library(terms)` and lists can be mapped to C arrays with `library(vectors)`. These are useful when you want to pass a complete copy of the data structure over to the other language. If you only want to access parts of a structure then the Structs and Objects packages are recommended.

10.3.15 Foreign Code Examples: UNIX

This section presents examples of incrementally loading C, Pascal and FORTRAN code into Prolog, using the foreign language interface under UNIX.

10.3.15.1 C Interface

If the C file '`c.c`' is compiled as shown below, then loading the Prolog file as shown will produce the indicated results.

c.c

```
/* c1(+integer, [-integer]) */
long int c1(a)
long int a;
{
   return(a+9);
}

/* c2(-integer) */
void c2(a)
long int *a;
{
   *a = 99;
}

/* c11(+atom, [-atom]) */
QP_atom c11(a)
QP_atom a;
{
   return(a);
}

/* c21(+atom, -atom) */
void c21(a,b)
QP_atom a;
QP_atom *b;
{
   *b = a;
}

/* c3(+float, [-float]) */
double c3(a)
double a;
{
   return(a+9.0);
}
/* c4(-float) */
void c4(a)
float *a;
{
   *a = 9.9;
}
/* c5(string, [-string]) */
char * c5(a)
char * a;
{
   return(a);
}
/* c6(-string) */
void c6(a)
char * *a;
{
```

At the command level:

```
% cc -c c.c
```

Produces the object file.

c.pl

```
foreign_file(c, [c1, c2, c11, c21, c3, c4, c5, c6]).

foreign(c1,  c, c1(+integer, [-integer])).
foreign(c2,  c, c2(-integer)).
foreign(c11, c, c11(+atom, [-atom])).
foreign(c21, c, c21(+atom, -atom)).
foreign(c3,  c, c3(+float, [-float])).
foreign(c4,  c, c4(-float)).
foreign(c5,  c, c5(+string,[-string])).
foreign(c6,  c, c6(-string)).

:- load_foreign_files([c], []),
   abolish(foreign_file,2),
   abolish(foreign,3).
```

Loading the Prolog file (see the reference pages for `foreign/3`, `foreign_file/2` and `load_foreign_files/2`) into Prolog and invoking the following query gives the following results:

```
| ?- c1(1,X1), c2(X2), c11(foo,X11), c21(foo,X21), c3(1.5,X3), c4(X4),
     c5(foo,X5), c6(X6).

X1 = 10,
X2 = 99,
X11 = X21 = X5 = foo,
X3 = 10.5,
X4 = 9.89999,
X6 = '99' ;



no
```

10.3.15.2 Pascal Interface

If the Pascal file '`p.p`' is compiled as shown below, then loading the Prolog file as shown will produce the indicated results.

p.p

```
type
alfa = packed array[1..10] of char;

(* p1(+integer, [-integer]) *)
function p1(a: integer32): integer32;
begin
  p1 := a + 9;
end;

(* p2(-integer) *)
procedure p2(var a: integer32);
begin
  a := 99;
end;

(* p11(+atom, [-atom]) *)
function p11(a: integer32) : integer32;
begin
  p11 := a;
end;

(* p21(+atom, -atom) *)
procedure p21(a: integer32; var b: integer32);
begin
  b := a;
end;

(* p3(+float, [-float]) *)
function p3(a: real) : real;
begin
  p3 := a + 9.0;
end;

(* p4(-float) *)
procedure p4(var a: real);
begin
  a := 9.9;
end;

(* p5(+string(10), -string(10)) *)
procedure p5(var s: alfa; var t: alfa);
begin
        t := s;
end;

(* p6(-string(10)) *)
procedure p6(var s: alfa);
begin
        s := 'output';
end;
```

At the command level:

```
% pc -c p.p
```

Produces the object file.

p.pl

```
foreign_file(p, [p1, p2, p11, p21, p3, p4, p5, p6]).

foreign(p1,  pascal, p1(+integer, [-integer])).
foreign(p2,  pascal, p2(-integer)).
foreign(p11, pascal, p11(+atom, [-atom])).
foreign(p21, pascal, p21(+atom, -atom)).
foreign(p3,  pascal, p3(+float, [-float])).
foreign(p4,  pascal, p4(-float)).
foreign(p6,  pascal, p5(+string(10),-string(10))).
foreign(p5,  pascal, p6(-string(10))).

:- load_foreign_files([p], ['-lpc']),
   abolish(foreign_file,2),
   abolish(foreign,3).
```

Loading the Prolog file (see `foreign/3`) into Prolog and invoking the following query gives the following results:

```
| ?- p1(1,X1), p2(X2), p11(foo,X11), p21(foo,X21), p3(1.5,X3), p4(X4),
     p5('parameter',X5), p6(X6).

X1 = 10,
X2 = 99,
X11 = X21 = foo,
X3 = 10.5,
X4 = 9.89999,
X5 = parameter,
X6 = output ;

no
```

Notes:

1. Passing of unsized strings (i.e. use of the `string` argument specification in a `foreign/3` fact) is not supported in this interface since `pc` does not have a convention for passing variable length arrays. Instead, padded strings (the `string(`*N*`)` argument specification) must be used. Notice that the corresponding parameter of `+string(`*N*`)` declaration is actually a call by reference parameter in Pascal procedures.
2. The linker option '`-lpc`' must be included in the call to `load_foreign_files/2` so that the foreign code routine will have access to the standard Pascal library.

10.3.15.3 FORTRAN Interface

If the FORTRAN file '`f.f`' is compiled as shown below, then loading the Prolog file as shown will produce the indicated results.

f.f

```
C       f1(+integer, [-integer])
        integer function f1(a)
            integer a
            f1 = a + 9
            return
        end

C       f2(-integer)
        subroutine f2(a)
            integer a
            a = 99
            return
        end

C       f11(+atom, [-atom])
        integer function f11(a)
            integer a
            f11 = a
            return
        end

C       f21(+atom, -atom)
        subroutine f21(a,b)
            integer a
            integer b
            b = a
            return
        end

C       f3(+float, [-float])
        real function f3(a)
            real a
            f3 = a + 9.0
            return
        end

C       f4(-float)
        subroutine f4(a)
            real a
            a = 9.9
            return
        end

C       f5(+string(10), [-string(10)])
        character*10 function f5(s)
        character*10 s
                f5 = s
                return
        end

C       f6(-string(10))
```

At the command level:

```
% f77 -c f.f
```

Produces the object file.

f.pl

```
foreign_file(f, [f1_, f2_, f11_, f21_, f3_, f4_, f5_, f6_]).

foreign(f1_,  fortran, f1(+integer, [-integer])).
foreign(f2_,  fortran, f2(-integer)).
foreign(f11_, fortran, f11(+atom, [-atom])).
foreign(f21_, fortran, f21(+atom, -atom)).
foreign(f3_,  fortran, f3(+float, [-float])).
foreign(f4_,  fortran, f4(-float)).
foreign(f5_,  fortran, f5(+string(10),[-string(10)])).
foreign(f6_,  fortran, f6(-string(10))).

:- load_foreign_files([f], ['-lF77']),
   abolish(foreign_file,2),
   abolish(foreign,3).
```

Loading the Prolog file (see `foreign/3`) into Prolog and invoking the following query gives the following results:

```
| ?- f1(1,X1), f2(X2), f11(foo,X11), f21(foo,X21), f3(1.5,X3), f4(X4),
     f5('parameter',X5), f6(X6).

X1 = 10,
X2 = 99,
X11 = X21 = foo,
X3 = 10.5,
X4 = 9.89999 ;
X5 = parameter ;
X6 = output ;

no
```

When you load FORTRAN code into a C program, you must ensure that any necessary FORTRAN run-time support code is loaded as well. The FORTRAN run-time library is divided into three parts in UNIX systems based on 4.2BSD:

- '`/usr/lib/libF77.a`' — this contains "mathematical" functions such as `sin()` and `catan()`, bit-handling functions, and support for character operations such as character assignment, concatenation, and comparison. You will almost always need to load this library file.
- '`/usr/lib/libI77.a`' — this contains the support routines for FORTRAN in-

put/output operations. If you are loading subroutines that do not perform FORTRAN input/output, you will not need to load this file. Note that there is currently no way of attaching a Prolog stream to a FORTRAN channel. We recommend that any FORTRAN subroutines to be loaded into Quintus Prolog perform input/output by calling C functions.

- '`/usr/lib/libU77.a`' — this contains interface routines that provide access to UNIX system calls. They are needed because the UNIX system calls expect strings in C format, which differs from FORTRAN format. If you are not calling any of the UNIX system calls from FORTRAN, you will not need to load this file.

UNIX systems based on System V have '`libF77.a`' and '`libI77.a`' but not '`libU77.a`'.

To ensure that these libraries will be loaded, use the linker options '`-lF77`', '`-lI77`', or '`-lU77`' respectively.

You should check your FORTRAN documentation for advice about combining FORTRAN subroutines with a C main program.

Notes:

1. Passing of unsized strings (for example, use of the `string` argument specification in a `foreign/3` fact) is not supported in this interface. Instead, padded strings (the `string(`*N*`)` argument specification) must be used.
2. The names of subroutines passed to the predicates `foreign_file/2` and `foreign/3` must end with an underscore ('`_`') to comply with the way in which `f77` generates external symbols.
3. The FORTRAN run-time library has been seen documented as '`-lf77`'. As case is significant in loader options, be sure to load this library using '`-lF77`'.

10.3.15.4 Passing pointers between Prolog and Foreign Code

Suppose we have a FORTRAN subroutine that multiplies a 3-element vector by a 3-by-3 matrix, returning a 3-element vector. This situation is then represented by the following code:

C code

```
typedef double vec_3[3];
typedef double mat_3_3[3][3];

vec_3 *make_vec(a, b, c)
    double a, b, c;
    {
        register vec_3 *x;

        x = (vec_3*)malloc(sizeof(vec_3));
        (*x)[0] = a, (*x)[1] = b, (*x)[2] = c;
        return x;
    }

mat_3_3 *make_mat(a0, a1, a2, b0, b1, b2, c0, c1, c2)
    double a0, a1, a2, b0, b1, b2, c0, c1, c2;
    {
        register mat_3_3 *x;

        x = (mat_3_3*)malloc(sizeof(mat_3_3));
        (*x)[0][0] = a0, (*x)[0][1] = a1, (*x)[0][2] = a2,
        (*x)[1][0] = b0, (*x)[1][1] = b1, (*x)[1][2] = b2,
        (*x)[2][0] = c0, (*x)[2][1] = c1, (*x)[2][2] = c2;
        return x;
    }
```

FORTRAN code

```
      subroutine matvec(mat, vec, ans)
          real mat(3,3), vec(3), ans(3)

          ans(1) = mat(1,1)*vec(1)+mat(2,1)*vec(2)+mat(3,1)*vec(3)
          ans(2) = mat(1,2)*vec(2)+mat(2,2)*vec(2)+mat(3,2)*vec(3)
          ans(3) = mat(1,3)*vec(3)+mat(2,3)*vec(2)+mat(3,3)*vec(3)
          return
      end
```

Prolog Code:

```
foreign(make_vec, c, make_vec(+float,+float,+float,
                              [-address(vec_3)])).
foreign(make_mat, c, make_mat(+float,+float,+float,
                              +float,+float,+float,
                              +float,+float,+float,
                              [-address(mat_3_3)])).
foreign(matvec_, fortran, matvec(+address(float),+address(float),
                                 +address(float))).      % note all +!

make_vec([A,B,C], X) :-
        make_vec(A, B, C, X).

make_mat([[A0,A1,A2],[B0,B1,B2],[C0,C1,C2]], X) :-
        make_mat(A0,A1,A2, B0,B1,B2, C0,C1,C2, X).

do_matvec(Vec, Mat, AnsObj) :-
        make_vec(Vec, VecObj),
        make_mat(Mat, MatObj),
        make_vec(0.0, 0.0, 0.0, AnsObj),
        matvec(VecObj, MatObj, AnsObj).
```

10.3.16 Summary of Predicates and Functions

Reference pages (in Chapter 18 [mpg], page 985) for the following provide further detail on the material in this section.

- `QP_atom_from_padded_string()`
- `QP_get*()`
- `QP_is*()`
- `QP_next_solution()`
- `QP_put*()`
- `QP_string_from_padded_atom()`
- `QP_term_type()`
- `QP_unify()`
- `foreign/3`
- `foreign_file/2`
- `load_foreign_executable/1`
- `load_foreign_files/2`

10.3.17 Library Support

-

- `library(vectors)`
- `library(terms)`

10.4 Foreign Functions Calling Prolog

10.4.1 Introduction

Quintus Prolog provides tools making it possible to call Prolog predicates from foreign languages. This is useful for a number of reasons.

- In the simplest case, a foreign function may want to get at information that is contained in the Prolog database. It might be inconvenient or unnatural for the foreign function to return to Prolog, Prolog to query its database and reinvoke foreign code.
- Foreign functions might also want to make use of Prolog's inferencing capabilities, again without having to contort algorithms.
- When Quintus Prolog is embedded in another application, that application invokes Prolog by calling it from foreign functions. This aspect of calling Prolog from foreign functions will be handled fully in the section on embedding Prolog (see Section 10.2 [fli-emb], page 365).

To take advantage of Quintus Prolog's ability to be called from foreign functions, you must first know how to do either of the following:

- load foreign code into Quintus Prolog; see Section 10.3 [fli-p2f], page 375
- embed Quintus Prolog in another application; see Section 10.2 [fli-emb], page 365

Currently, only functions written in C can call Prolog predicates directly; this restriction may be lifted in future releases of Quintus Prolog. However, other languages such as Pascal and FORTRAN can call Prolog by virtue of their ability to call C. foreign code!) calls Prolog. The FORTRAN and Pascal stuff will then help users call the C functions. It just wasn't making sense writing all of this in terms of generic "foreign languages", because I knew full well I'd have to eventually do something different for FORTRAN and Pascal. While this should be more clear for release 3, it may as a result require more rewriting in case we ever do make Prolog directly callable from FORTRAN and Pascal.}

The interface between C and Prolog supports direct exchange of Prolog's atomic data types (atoms, integers or database reference). The data is automatically converted between Prolog's internal representation and the internal representation of the foreign language. The interface also supports passing any Prolog term from C and returning any Prolog term to C.

10.4.1.1 Summary of steps

Following is a summary of the steps that enable you to call a Prolog predicate from a C function:

IN THE PROLOG CODE:

1. Use `extern/1` to declare the predicate callable from foreign functions, establishing an argument passing interface (see Section 10.4.2 [fli-ffp-ppc], page 414).

IN THE C CODE:

1. Look up the Prolog predicate by calling one of the functions `QP_predicate()` or `QP_pred()`. This provides C with a "handle" on the Prolog predicate that is used to make the actual call. (see Section 10.4.5.1 [fli-ffp-ccp-lcp], page 422).
2. If only a single solution is required, the predicate handle, together with C variables for input and output parameters, is passed to `QP_query()`. If `QP_query()` returns successfully, output results will have been left in C variables according to the specified interface.
3. If more than one solution is required, the predicate handle and C variables for input and output parameters are passed to `QP_open_query()`. This function initiates a query, returning a query identifier, which is passed to `QP_next_solution()`, which is called once for each solution requested. When a solution is returned, output results are left in C variables according to the specified interface. When sufficient solutions have been returned, or there are no more solutions, the query identifier is passed to either `QP_cut_query()` or `QP_close_query()` to terminate the query.

10.4.2 Making Prolog Procedures Callable by Foreign Functions

Any Prolog predicate can be made callable from foreign functions using the declaration `extern/1`. This includes built-in predicates, as well as predicates that are currently undefined and dynamic predicates that currently have no clauses.

An `extern/1` declaration not only makes the predicate known to foreign languages, but also specifies how arguments will be passed to and from it. When a predicate is declared callable using `extern/1` declaration, it becomes available to foreign functions as soon as the declaration is loaded. This is equally true of `extern/1` declarations occurring in files that are loaded from source form, pre-compiled QOF files whose source files contained `extern/1` declarations, and certain QOF files (e.g. those created using `save_program/1`) that retain callability information (see Section 8.5.4 [ref-sls-sst], page 196).

A Prolog predicate that has been made callable from foreign functions is not otherwise changed in any way. It can still be abolished or redefined just like any other predicate. There is no performance penalty associated with making a predicate callable from foreign functions. A predicate can be redeclared by loading a new or modified `extern/1` declaration.

When a predicate is made callable from foreign code, a new and closely related Prolog predicate called an *interface predicate* is created in the module in which `extern/1` was declared. The interface predicate has the same arity as the callable predicate, and its name is the name of the declared predicate with an underscore prepended to it. The interface predicate provides the link between foreign languages and its Prolog predicate. It can be abolished or saved just as any other predicate, but because they can only be created using an `extern/1` declaration, valid interface predicates cannot be made dynamic or multifile.

The purpose of the interface predicate is to supply an entry point into Prolog for foreign functions, and a handle on the property of callability of Prolog procedures for manipulation in Prolog. It is possible to call an interface predicate from Prolog, but the call will simply fail.

10.4.2.1 Specifying the Argument Passing Interface: `extern/1`

An `extern/1` declaration has the form

```
:- extern(+CallSpecification)
```

CallSpecification is a Prolog term specifying how calls from C will pass arguments to and receive arguments from the Prolog predicate. Handling arguments passed from other languages is discussed in Section 10.4.7 [fli-ffp-ppl], page 433.

CallSpecification is of the form:

```
PredicateName(ArgSpec, ArgSpec, ...)
```

where *PredicateName* is the name of the Prolog predicate and each *ArgSpec* is an argument specification for the corresponding argument of the predicate. *ArgSpec* must be one of the following list

```
+integer +float +single +double +atom +string +term +address(T)
-integer -float -single -double -atom -string -term -address(T)
```

where *T* is a foreign type name. The argument type `address` can always be used instead of `address(T)`.

Argument specifications used when declaring Prolog predicates callable from C are equivalent to those used when specifying an interface for C functions that are to be callable from Prolog.

Here are some example `extern/1` declarations:

Examples:

```
:- extern(write(+integer)).
:- extern(call(+term)).
:- extern(my_proc(+atom,+integer,-term,-integer)).
```

10.4.3 Passing Data to and from Prolog

The foreign function interface automatically converts between C's representation of data and the representation of atomic data types expected by Prolog. Thus the calling function does not need to know how Prolog represents atoms, integers, floats or addresses in order to communicate with Prolog. This feature simplifies the integration of Prolog with foreign code; in particular, it makes it easier to interface directly with already-written functions in libraries and other programs. It also allows for compatibility with later versions of Quintus Prolog and with versions of Quintus Prolog running on other hardware.

> **Asymmetry note:** When C calls Prolog, in contrast to Prolog calling foreign code, there is no Prolog datum passed as the function return value. Instead, the return value supplies the calling function with information as to whether the Prolog call succeeded or failed, or whether there was an exception raised.

Arguments are passed from C functions to Prolog predicates in the same order as they appear in the Prolog call. Prolog assumes that C functions will call Prolog predicates with the number and type of arguments as declared by the `extern/1` declarations; if it does not, the results are unpredictable. Certain types of inputs (for example, atoms) can be checked for validity when the query to Prolog is made, and an error value is returned if the type is incorrect. Outputs are passed to Prolog as pointers to storage for results. Prolog will internally create unbound variables with which to calculate the results. The outputs will then be automatically converted and written into the C storage according to the calling specification. If the result Prolog computes is inconsistent with the specified output type, an exception is signaled.

> **Asymmetry note:** When Prolog calls foreign code, outputs are unified with items supplied by the calling function; with C calling Prolog, assignment is used instead.

10.4.3.1 Passing Integers

```
Prolog:  +integer
C:       long int x;
```

The C long int is converted to a 32-bit Prolog integer, which is passed to the Prolog call. If the C integer contains garbage when it is passed, Prolog will receive that garbage as an integer.

```
Prolog:  -integer
C:       long int *x;
```

A pointer to a C long int is passed to the foreign interface. When Prolog returns a solution, a Prolog integer is expected in the corresponding argument of the call. The foreign interface converts that integer into a C long int and writes it at the location supplied. The previous contents of the location are destroyed. If the Prolog call does not return an integer in the appropriate position, a type error is raised and the contents of the location is unchanged.

10.4.3.2 Passing Floats

```
Prolog:  +float
C:       double x;
```

The C double-precision float is converted to a Prolog float, which is passed to the Prolog call. If the C double contains garbage, Prolog will receive that garbage as a double-precision floating-point number. Many C compilers will allow the parameter declaration to be `float` instead of `double` because they always convert single-precision floating-point arguments to double-precision. However, C compilers conforming to the new ANSI standard will not do this, so it is recommended that `double` be used.

```
Prolog:  +single
ANSI C:  float x;
```

The C single-precision float is converted to a Prolog float, which is passed to the Prolog call. If the C float contains garbage, Prolog will receive that garbage as a single-precision floating-point number.

Normally, this type of argument is not used; however, C compilers conforming to the new ANSI standard can pass single precision floats to Prolog without first converting them to double. It is not recommended that floats be passed as `single` until you have verified that your C compiler behaves as desired.

```
Prolog:  +double
C:       double x;
```

The C double-precision float is converted to a Prolog float, which is passed to the Prolog call. If the C float contains garbage, Prolog will receive that garbage as a double-precision floating-point number.

```
Prolog:  -float
C:       float *x;
```

A pointer to a C float is passed to the foreign interface. When Prolog returns a solution, a Prolog floating-point number is expected in the corresponding argument of the call. The foreign interface converts that number into a C float and writes it at the location supplied. The previous contents of the location are destroyed. If the Prolog call does not return a floating-point number in the appropriate position, a type error is raised and the contents of the location is unchanged.

```
Prolog:  -single
C:       float *x;
```

A pointer to a C float is passed to the foreign interface. When Prolog returns a solution, a Prolog floating-point number is expected in the corresponding argument of the call. The foreign interface converts that number into a C float and writes it at the location supplied. The previous contents of the location are destroyed. If the Prolog call does not return a

floating-point number in the appropriate position, a type error is raised and the contents of the location is unchanged.

When the foreign language calling Prolog is C, this type of argument is not normally used; however, C compilers conforming to the new ANSI standard can return single precision floats from Prolog without first converting them to double. It is not recommended that floats be passed as `single` until you have verified that your C compiler behaves as desired.

```
Prolog:  -double
C:       double *x;
```

A pointer to a C double is passed to the foreign interface. When Prolog returns a solution, a Prolog floating-point number is expected in the corresponding argument of the call. The foreign interface converts that number into a C double and writes it at the location supplied. The previous contents of the location will be destroyed. If the Prolog call does not return a floating-point number in the appropriate position, a type error is signaled and the contents of the location is unchanged.

It is assumed that the interface will overwrite this float with Prolog's result. When Prolog returns, its floating-point number is converted to double-precision and written onto the space for the foreign double. The previous contents of the C double will be lost. If the Prolog call does not return a floating-point number, a type error is raised and the result is unchanged.

10.4.3.3 Passing Atoms in Canonical Form

The foreign function interface allows Prolog atoms to be passed from C functions to Prolog either in a canonical form as unsigned integers, or as pointers to character strings.

This section describes passing atoms in canonical form. For each Prolog atom there is a single canonical representation. Programs can rely on the property that identical atoms have identical canonical representations. Note, however, that the canonical form of an atom is not necessarily identical across different invocations of the program. This means that canonical atom representations should not be used in files or interprogram communication. For these purposes strings should be used (see Section 10.4.4.1 [fli-ffp-a2s-str], page 419). Foreign functions can store canonical atoms in data structures, pass them around, access their strings using `QP_string_from_atom()` and pass them back to Prolog, but they should not attempt any other operations on them.

```
Prolog:  +atom
C:       QP_atom x;
```

The `QP_atom` must be a valid Prolog atom, otherwise the function attempting to pass the atom parameter (`QP_query()` or `QP_open_query()`) will return `QP_ERROR`. The C `QP_atom` is passed to Prolog, where it will appear as a normal Prolog atom. Atoms can be converted to strings using the functions `QP_string_from_atom()` or `QP_atom_from_padded_string()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393).

```
Prolog:  -atom
C:       QP_atom *x
```

A pointer to a C `QP_atom` is passed to the foreign interface. When Prolog returns a solution, a Prolog atom is expected in the corresponding argument of the call. This atom might be one obtained from Prolog, or one generated through the function `QP_atom_from_string()` or `QP_atom_from_padded_string()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393). The foreign interface simply writes that atom at the location supplied. The previous contents of the location are destroyed. If the Prolog call does not return an atom in the appropriate position, a type error is raised and the contents of the location is unchanged.

Also see Section 10.4.4 [fli-ffp-a2s], page 419 for discussion of conversion between atoms and strings.

10.4.4 Converting Between Atoms and Strings

10.4.4.1 Passing Atoms as Strings

The foreign function interface allows Prolog atoms to be passed from C functions to Prolog either in a canonical form as unsigned integers, or as pointers to character strings.

This section describes passing atoms as pointers to null-terminated character strings. Strings are always identical across different invocations of a program, so are the correct atom representation to use when writing to files or using interprogram communication. For other uses, atoms in the canonical form may be appropriate (see Section 10.4.3.3 [fli-ffp-dat-cat], page 418).

If, in a later release of Quintus Prolog, it is possible to call Prolog directly from FORTRAN or Pascal, it will additionally be possible to pass atoms as fixed-length, blank-padded strings (as when Prolog calls FORTRAN or Pascal).

```
Prolog:  +string
C:       char *x
```

The argument passed from the C function is a null-terminated character string. The foreign interface automatically converts the string to a Prolog atom, and passes it to the Prolog predicate.

```
Prolog:  -string
C:       char **x;
```

A pointer to a C string pointer is passed to the foreign interface. When Prolog returns a solution, a Prolog atom is expected in the corresponding argument of the call. This atom might be one obtained from Prolog, or one generated through the function `QP_atom_from_string()` or `QP_atom_from_padded_string()` (see Section 10.3.7.4 [fli-p2f-atm-a2s], page 393). The foreign interface writes a pointer to the string for that atom at the location

supplied. The previous contents of the location are destroyed. This string must not be overwritten by the C function. If the Prolog call does not return an atom in the appropriate position, a type error is raised and the contents of the location is unchanged.

See also Section 10.4.4 [fli-ffp-a2s], page 419 for discussion of conversion between atoms and strings.

10.4.4.2 Passing Terms

The foreign function interface allows Prolog terms to be passed from C functions. Like most of the simple data types that may be passed to and from Prolog, terms to be passed can originate either in Prolog or in C (see Section 10.3.8 [fli-p2f-trm], page 395). Like terms in Prolog, terms that are passed to C are automatically made safe from damage by operations that might change their absolute position in Prolog memory, like stack shifting and garbage collection.

```
Prolog:  +term
C:       QP_term_ref
```

The argument passed from the C function is a `QP_term_ref` initialized to a Prolog term. If something other than a term reference is passed to Prolog, the results are undefined.

```
Prolog:  -term
C:       QP_term_ref
```

A `QP_term_ref` is passed to the foreign interface. When Prolog returns a solution, the foreign interface writes a reference to the Prolog term in the corresponding argument of the call into the `QP_term_ref`. The previous contents of the `QP_term_ref` are destroyed. Unlike the other passing types, there are no associated type errors when passing terms.

Note that the output term, as well as the input term is represented in the C code by a `QP_term_ref`. This contrasts with other output types, which are usually represented in C as pointers to the corresponding input type.

10.4.4.3 Passing Addresses

Previous releases of Quintus Prolog had the restriction that integers of greater than 29 bits could not be represented as Prolog integers. Certain platforms, however, have pointers that use some of the four most significant bits; for these machines, pointers could not be represented as Prolog integers. This problem motivated the `address`, which could be treated specially, as a distinct data type that can be passed through the foreign interface.

With release 3 the restriction of integers to 29 bits has been lifted; however, the internal representation of integers of more than 29 bits is more bulky and somewhat slower to manipulate than that of smaller integers. While this is not a problem in normal programs, it could penalize programs that manipulate pointers in Prolog on certain platforms whose

pointers require more than 29 bits. We have chosen to retain the address data type in release 3 so that such penalties can be avoided where possible, as well as for backward compatibility. Addresses can be passed both to and from Prolog from foreign functions, and to and from foreign functions from Prolog. (See Section 10.3.9 [fli-p2f-poi], page 397.)

As when calling foreign code from Prolog, pointers should be passed through the interface using the type specification

```
address(typename)
```

as described in more detail below. *typename* should be the name used in the foreign language to identify the type of object named by the pointer.

```
Prolog:  +address(typename)
C:       typename *x;
```

The C pointer is converted to a 32-bit Prolog integer, which is passed to the Prolog call. If the C pointer contains garbage when it is passed, Prolog will receive that garbage as an integer.

```
Prolog:  -address(typename)
C:       typename **x;
```

A pointer to a C pointer is passed to the foreign interface. When Prolog returns a solution, a Prolog integer is expected in the corresponding argument of the call. If the argument is 0, the foreign function writes the NULL pointer at the location supplied. Otherwise, the foreign interface converts the integer into a C pointer and writes it at the location. The previous contents of the location will be destroyed. If the Prolog call does not return an integer in the appropriate position, a type error is signaled and the contents of the location is undefined.

```
Prolog:  +address
C:       char *x
```

This is equivalent to `+address(char)`.

```
Prolog:  -address
C:       char **x
```

This is equivalent to `-address(char)`.

Using `+address` in place of `+address(`*typename*`)`, or `-address` in place of `-address(`*typename*`)`, has no effect on the execution of the program; however, doing so can reduce the readability of the program and compromise program checking tools.

10.4.5 Invoking a Callable Predicate from C

A Prolog predicate that has been made callable from foreign code can be invoked in two ways: determinately or nondeterminately. A determinate query asks for the first solution and the first solution only. A nondeterminate query allows Prolog to backtrack, possibly returning multiple solutions to the calling foreign function.

Note that the terms *determinate* and *nondeterminate* do not refer to the Prolog predicate being called, but rather to the query. It is perfectly reasonable to ask for only the first solution of a Prolog predicate that is nondeterminate, or to attempt to return all solutions to a predicate that in fact has just one. Multiple solutions, if any, are returned in the Prolog solution order. When only a single solution is desired a determinate call is preferred, as it is more efficient and concise.

10.4.5.1 Looking Up a Callable Prolog Predicate

Before a Prolog predicate can be called from a foreign language it must be looked up. The C functions `QP_predicate()` and `QP_pred()` perform this function. The lookup step could have been folded into the functions that make the query, but if a predicate was to be called many times the redundant, if hidden, predicate lookup would be a source of unnecessary overhead. Instead, `QP_predicate()` or `QP_pred()` can be called just once per predicate. The result can then be stored in a variable and used as necessary.

Both `QP_predicate()` and `QP_pred` return a `QP_pred_ref`, which represents a Prolog predicate. The type definition for `QP_pred_ref` is found in '`<quintus/quintus.h>`'.

```
QP_pred_ref QP_predicate(name_string, arity, module_string)
char *name_string;
long int arity;
char *module_string;
```

`QP_predicate()` is the most convenient way of looking up a callable Prolog predicate. It is simply passed the name and module of the predicate to be called as strings, the arity as an integer, and returns a `QP_pred_ref`, which is used to make the actual call to Prolog.

`QP_predicate()` can only be used to look up predicates that have been declared callable from foreign code. If some other predicate is looked up, `QP_ERROR` is returned. Checking the return value protects you from attempting to call a predicate that isn't yet ready to be called.

```
QP_pred_ref QP_pred(name_atom, arity, module_atom)
QP_atom name_atom;
long int arity;
QP_atom module_atom;
```

`QP_pred()` is a less convenient, but faster, means of looking up a callable Prolog predicate. Unlike `QP_predicate()`, it has its name and module arguments passed as Prolog atoms. These may have been returned to C from Prolog, or may have been built in the foreign language using `QP_atom_from_string()`. One additional difference is that the name passed is *not* the name of the Prolog predicate to be called, but rather the name of the interface predicate constructed when the Prolog predicate was made callable from foreign code Section 10.4.2 [fli-ffp-ppc], page 414. Much of the cost of `QP_predicate()` is from having to look up Prolog atoms for its name and module arguments. By avoiding doing this unnecessarily, what `QP_pred()` gives up in convenience is returned in performance. Like `QP_predicate()`, `QP_pred()` returns a `QP_pred_ref`, which is used to make the actual call to Prolog. If a predicate that is not an interface predicate is looked up, `QP_pred()` returns `QP_ERROR`.

`QP_pred()` can only be used to look up predicates that have been declared callable from foreign code. If some other predicate, or a predicate that does not exist, is looked up, `QP_ERROR` is returned. This protects you from attempting to call a predicate that isn't yet ready to be called.

10.4.5.2 Making a Determinate Prolog Query

A determinate query can be made in a single C function call using `QP_query`. The first argument passed to `QP_query()` is a `QP_pred_ref` for the predicate to be called. Any arguments after the first represent parameters to be passed to and from the Prolog predicate.

The foreign language interface will interpret arguments passed to the Prolog predicate according to the call specification given when the predicate was made callable. Hence, it is important that the arguments to be passed to and from the Prolog predicate should correspond to that call specification. For certain parameter types (passing Prolog atoms in canonical form) it is possible to detect inconsistencies between data supplied to `QP_query()` and the call specification, but for the most part this is impossible. Calls that are inconsistent with their call specifications will produce undefined results.

`QP_query()` returns one of three values: `QP_SUCCESS`, indicating that the query was made and a solution to the query was computed; `QP_FAILURE`, meaning that the query was made but no solution could be found; and `QP_ERROR`, which says that either the query could not be made, or that an exception was signaled from Prolog but not caught. In this case, see the reference page for `QP_exception_term()`. Only when the return value is `QP_SUCCESS` should the values in variables passed as outputs from Prolog be considered valid. Otherwise, their contents are undefined.

It is important that a valid `QP_pred_ref` is passed to `QP_query()`; in particular, it is advisable to check for an error return from `QP_predicate()` or `QP_pred()` before calling `QP_query()`.

10.4.5.3 Initiating a Nondeterminate Prolog Query

For a nondeterminate query, multiple solutions to the query may be successively returned to the calling foreign function. Nondeterminate queries are made in three steps: the query is first initiated, or "opened", using `QP_open_query()`. Solutions are then requested using `QP_next_solution()`. When all desired solutions have been returned, or there are no more solutions, the query must be terminated by calling `QP_cut_query()` or `QP_close_query()`.

The C function `QP_open_query()` is used to initiate a nondeterminate Prolog query. The arguments passed to `QP_open_query()` are identical to those that would be passed to `QP_query()`; however, `QP_open_query()` does not compute a solution to the query. Its effect is to prepare Prolog for the computation of solutions to the query, which are requested by calls to the function `QP_next_solution()`. For consistency checking, `QP_open_query()` returns a `QP_qid`, which represents the Prolog query. The type definition for `QP_qid` is found in '`<quintus/quintus.h>`'. The `QP_qid` returned by a call to `QP_open_query()` must be passed to each call to `QP_next_solution()` for that query, as well as to `QP_cut_query()` or `QP_close_query()` when terminating the query. If an invalid `QP_qid` is passed to one of these functions, the function has no effect except to return `QP_ERROR`.

When requesting solutions from an open nondeterminate query, input and output parameters are *not* passed. The effect of `QP_open_query()` is to pass inputs to Prolog, which subsequently maintains them. It also tells Prolog where storage for outputs has been reserved. This storage will be written later, when solutions are returned.

If an error occurs when attempting to open a query, `QP_ERROR` is returned and the query is automatically terminated.

It is important that a valid `QP_pred_ref` is passed to `QP_open_query()`; in particular, it is advisable to check for an error return from `QP_predicate()` or `QP_pred()` before calling `QP_open_query()`.

10.4.5.4 Requesting a Solution to a Nondeterminate Prolog Query

The function `QP_next_solution()` is used to return a solution from an open nondeterminate Prolog query. Solutions are computed on demand, and multiple solutions are returned in the normal Prolog order. `QP_next_solution()` is passed the `QP_qid` returned by `QP_open_query()` when the nondeterminate query was opened. No additional input or output parameters are passed: after a call to `QP_open_query()`, Prolog manages inputs itself, and has been told where storage for outputs has been reserved. Each time `QP_next_solution()` computes a new solution it writes it on the output storage for the foreign function to use as it likes. Each new solution overwrites the old memory, destroying the previous solution, so it is important that the foreign function copies solutions elsewhere if it wants to accumulate them.

Important restriction: only the innermost, i.e. the most recent, open query can be asked to compute a solution. A new query can be made at any point whether or not other

nondeterminate queries are open; however, while the new query remains open only it will be able to return solutions.

10.4.5.5 Terminating a Nondeterminate Prolog Query

`QP_close_query()` and `QP_cut_query()` are functions that are used to terminate an open nondeterminate Prolog query. They differ only in their effect on the Prolog heap, which can be reflected in the solutions Prolog has returned to C.

The difference between `QP_close_query()` and `QP_cut_query()` can best be understood with reference to Prolog's control flow. `QP_close_query()` is equivalent to the Prolog call '`!, fail`'. The cut renders the current computation determinate, removing the possibility of future backtracking. The following call to `fail/0` then initiates backtracking to the next previous parent goal with outstanding alternatives. In doing so it pops the Prolog heap to its state when the parent goal succeeded, in effect throwing away any terms created since that parent goal.

In contrast, just calling '`!`' in Prolog renders the computation determinate without initiating backtracking. Any terms created since the parent goal are retained.

In the context of calling Prolog from foreign languages, terminating a query using `QP_close_query()` generally means throwing away the most recent solution that was calculated, unless that solution has been copied into a more permanent place. (Of course, any previous solutions must also be assumed to have been overwritten by subsequent solutions unless copied elsewhere!) The converse of this behavior is that closing a query using `QP_close_query()` automatically frees up the Prolog memory that holds the last solution.

Terminating a query using `QP_cut_query()` renders the computation determinate, but as it is not failed over the Prolog heap is not popped. Thus when terminating a query using `QP_cut_query()` more space is retained, but so is the most recent solution.

10.4.6 Examples

10.4.6.1 Calling Arbitrary Prolog Goals from C

Any Prolog predicate can be made callable from foreign code, including system built-ins. An especially useful case of this generally useful ability is making the built-in `call/1` callable. `call/1` is declared callable like any other predicate, and is passed the Prolog term to be called. The term may have originated in Prolog, or may have been constructed in C using the supplied term manipulation functions (see Section 10.3.8 [fli-p2f-trm], page 395).

In this particular example, we pass a term from Prolog to C, then C calls `call/1` with that term. This lets us concentrate on the calling rather than on the construction of the term to be called.

On the Prolog side of the interface, the following declaration is loaded:

```
:- extern(call(+term)).
```

On the C side, the following function is defined, compiled and either loaded into Prolog using the dynamic foreign interface or statically linked with Prolog:

```
#include <quintus/quintus.h>

call_prolog(t)
QP_term_ref t;
{
        QP_pred_ref call = QP_predicate("call", 1, "user");
        QP_query(call, t);
}
```

This done, any goal that can be called from Prolog can also be called from C by passing it to `call_prolog/1`.

10.4.6.2 Generating Fibonacci Numbers

Prolog and foreign languages are generally intercallable in Quintus Prolog; in particular, arbitrarily nested calling is permitted. The following example uses recursive calling between Prolog and C to generate Fibonacci numbers:

fib.pl

```
fib :-
    int(I),
    fib(I, F),
    write(fib(I,F)), nl,
    fail.

int(I) :- int(0, I).

int(I, I).
int(I, K) :-
    J is I+1,
    int(J, K).

fib(N, F) :-
    (   N =< 1 ->
        F = 1.0
    ;   N1 is N-1,
        N2 is N-2,
        c_fib(N1, F1),
        c_fib(N2, F2),
        F is F1+F2
    ).

:- extern(fib(+integer, -float)).

foreign(c_fib, c_fib(+integer, [-float])).
foreign_file(fib, [c_fib]).

:- load_foreign_files(fib, []).
```

fib.c

```
#include <quintus/quintus.h>

double c_fib(i)
    long int i;
    {
        float f1, f2;
        QP_pred_ref fib = QP_predicate("fib", 2, "user");

        if (i <= 1) {
            return 1.0;
        } else {
            QP_query(fib, i-1, &f1);
            QP_query(fib, i-2, &f2);
            return f1+f2;
        }
    }
```

10.4.6.3 Calling a Nondeterminate Predicate

This example shows how a nondeterminate query can be made from C. It also shows how you can get at the exception terms raised from a Prolog query from C.

brothers.pl

```
foreign(brothers, c, brothers).
foreign_file(brothers, [brothers]).
:- load_foreign_files(brothers, []),
        abolish([foreign/3, foreign_file/2]).

:- extern(write(+term)).        % Make write/1     callable from C
:- extern(karamazov(-atom)).    % Make karamazov/1 callable from C

karamazov('Fyodor').
karamazov('Dmitri').
karamazov('Ivan').
karamazov('Alyosha').
```

brothers.c

```
#include <quintus/quintus.h>

/* lookup_predicate() is just a wrapper around QP_predicate()
   that prints an error message if QP_predicate() fails.
   It returns 1 if QP_predicate() succeeds and 0 if
   QP_predicate() fails
*/
int lookup_predicate(name, arity, module, predref)
    char * name;
    int    arity;
    char * module;
    QP_pred_ref * predref;
    {
        *predref = QP_predicate(name, arity, module);

        if (*predref == QP_BAD_PREDREF) {
            printf("%s:%s/%-d is not callable from C\n",
                   module, name, arity);
            return 0;
        } else {
            return 1;
        }
    }
```

brothers.c

```
void brothers() /* brothers() is called from Prolog */
    {
        QP_pred_ref     karam, write;
        QP_qid          query;
        QP_atom         bro;
        int             status;

        if (!lookup_predicate("karamazov", 1, "user", &karam)) {
            return;
        }

        if ((query = QP_open_query(karam, &bro)) == QP_BAD_QID) {
            printf("Cannot open query\n");
            return;
        }

        /* Get solutions one at a time */
        while ((status = QP_next_solution(query)) == QP_SUCCESS) {
            printf("%10s is a Karamazov\n",
                   QP_string_from_atom(bro));
        }

        QP_close_query(query);

        if (status == QP_ERROR) {
            /* Query raised an exception */
            QP_term_ref error = QP_new_term_ref();

            printf("Query signalled an exception\n");
            if (QP_exception_term(error) == QP_ERROR) {
                printf("Could not get at exception term\n");
                return;
            }

            if (lookup_predicate("write", 1, "user", &write)) {
                /* Call Prolog builtin write/1 to print the
                   exception term */
                if (QP_query(write, error) != QP_SUCCESS) {
                    printf("Couldnt write exception term\n");
                } else {
                    return;
                }
            }
        }
    }
```

To test the `QP_exception_term()` part of this example add a clause for `karamazov/1` like:

```
karamazov(_) :- raise_exception(karamazov(error)).
```

10.4.6.4 Nested Prolog Queries

This example demonstrates how you can have nested queries to Prolog from C. For brevity sake, we don't check the statuses returned by all the calls to `QP_next_solution()` for error values. This is not advised in real applications. This example also shows the use of `QP_cut_query()`.

books.pl

```
foreign(print_books, c, print_books).
foreign_file(books, [print_books]).
:- load_foreign_files(books, []),
        abolish([foreign/3, foreign_file/2]).

:- extern(author(-atom)).
:- extern(book(+atom,-atom)).

author(hesse).
author(kafka).
author(dostoyevski).

book(dostoyevski, idiot).
book(dostoyevski, gambler).
book(hesse,  steppenwolf).
book(hesse,  sidhdhartha).
book(hesse,  demian).
book(kafka,  america).
book(kafka,  trial).
book(kafka,  castle).
book(kafka,  metamorphosis).
```

books.c

```
#include <quintus/quintus.h>

#define MAX_BOOKS 3

void print_books()
    {
        QP_pred_ref author, book;
        QP_qid      q1, q2;
        QP_atom     a, b;
        int         count;

        if (!(lookup_predicate("author", 1, "user", &author)))
          return;

        if (!(lookup_predicate("book", 2, "user", &book)))
          return;

        if ((q1 = QP_open_query(author, &a)) == QP_BAD_QID) {
            printf("Cant open outer query\n");
            return;
        }

        while (QP_next_solution(q1) ==  QP_SUCCESS) {
            /* For each solution returned by author(X) do */
            if ((q2 = QP_open_query(book, a, &b)) == QP_BAD_QID) {
                printf("Cant open inner query\n");
                break;
            }
            printf("Books by %s:\n", QP_string_from_atom(a));

            count = 0;
            while ((count < MAX_BOOKS) &&
                   (QP_next_solution(q2) == QP_SUCCESS)) {
                /* Find atmost MAX_BOOKS solns for books(X,Y) */
                printf("\t\t%s\n",QP_string_from_atom(b));
                count++;
            }
            QP_close_query(q2);
        }
        QP_close_query(q1);
    }
```

10.4.7 Calling Prolog from Pascal and FORTRAN

It is possible to call Prolog predicates from Pascal and FORTRAN using C as an intermediary language. Your Pascal or FORTRAN manual will tell you how to make your code call C. Then call Prolog from C using the procedures described in this section.

10.4.8 Summary of Predicates and Functions

Reference pages for the following provide further detail on the material in this section.

- `extern/1`
- `QP_close_query()`
- `QP_cut_query()`
- `QP_next_solution()`
- `QP_open_query()`
- `QP_pred()`
- `QP_predicate()`
- `QP_query()`

10.4.9 Library Support

-
- `library(vectors)`
- `library(terms)`

10.5 Quintus Prolog Input / Output System

This section describes how Prolog streams are represented as C data structures, how streams can be configured to handle various file formats and how to create a customized Prolog stream in C.

10.5.1 Overview

The Quintus Prolog input/output system is designed to handle various file formats, device-dependent I/O, and in particular, it enables you to create customized Prolog streams in C. File-related input/output operations of a Prolog program can be coded to be portable among different operating systems and the underlying formats of files.

A *Prolog stream* is an object storing the information about how to complete input/output operations to a file, device or other form of I/O channel. All Prolog input/output operations are performed through Prolog streams.

The embedding ("bottom") layer of the Quintus Prolog system provides a set of default functions for handling normal Prolog streams. However, *user defined streams* can be defined at runtime in such a way that Prolog built-in I/O operations work on other types of Prolog stream. Examples of user-defined streams include:

- a stream to inter-process communication
- a stream to a window in a graphic environment

There are three streams opened by the embedding layer I/O initialization functions when a Prolog process starts:

- user input stream: normal input channel
- user output stream: normal output channel
- user error stream: normal error message channel

Prolog also keeps track of two current streams,

- current input stream
- current output stream

A Prolog input/output built-in predicate or function that takes no stream argument is performed on the current input or current output stream.

In this section, we outline the Quintus Prolog input/output model and describe the Prolog stream structure, defining the different formats and options that can be associated with a stream. Then we discuss the method of creating a user-defined stream.

We also list a number of functions that enable Prolog streams to be manipulated in foreign code. Finally we discuss some compatability issues with the I/O system in previous versions of Quintus Prolog.

> **Please note:** The terms *record* and *line* have the same meaning in this section. A line terminated with ⟨LFD⟩ is just a type of record. However, a record (line) is not always terminated by ⟨LFD⟩.

10.5.2 Input/Output Model

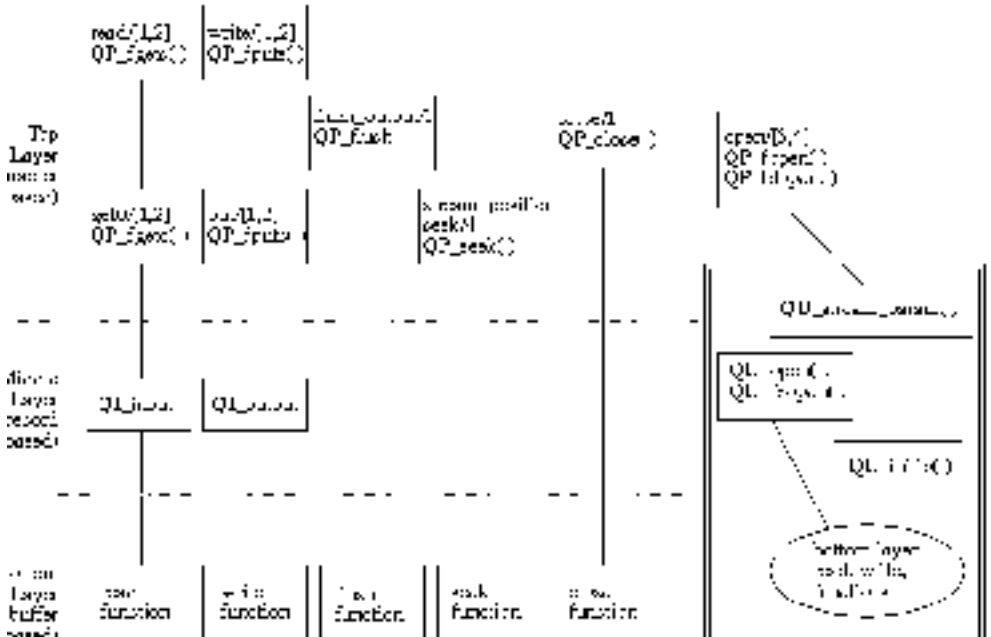
There are three layers of input/output operations visible in the Prolog system as illustrated in the figure "Input/Output Model".

The *top layer* is character based. It supports reading a character, writing a character and testing the state of a Prolog stream. `get0/[1,2]` and `put/[1,2]` are examples of the first layer operation.

The *middle layer* is record based. Its primary function is to keep the integrity of a record, through such operations as trimming a record, padding a record and handling output overflow. This layer is not visible to the user and cannot be changed by the user.

The *bottom layer* is buffer based. It performs the actual input from or output to the underlying device of a Prolog stream. The bottom layer is a collection of five functions associated with a stream: read, write, flush, seek and close functions. Typically the read function reads data from the underlying device into a buffer that it maintains and then passes this data up to the middle layer a record at a time. The write function provides buffer space for a record to be received from the middle layer and writes out the buffer to the underlying device.

The embedding open function `QU_open()` assigns the appropriate bottom layer functions for a stream created by `open/[3,4]`. A user-defined Prolog stream must supply its own bottom layer functions for the stream. Bottom layer functions are described in Section 10.5.6 [fli-ios-bot], page 451.



Input/Output Model

Writing to a QP_DELIM_LF record file stream demonstrates how the three layers work together. Each put/2 call on the stream simply stores a character in the record buffer of the stream. When the top layer predicate, nl/1, is called on the stream, the middle layer output function is called. The middle layer function stores a ⟨LFD⟩ in the record buffer and updates some counters for the stream. It then calls the bottom layer function of the stream. The bottom layer writes out the record to the output file.

> **Please note:** In addition, the top layer contains predicates and C functions to perform seek, flush output and close operations on a stream. There are no middle layer functions for these operations.

10.5.3 Stream Structure

The Prolog representation of a Prolog stream provides a way of retrieving information from its corresponding C stream structure. It is currently represented as a Prolog term in the form of '$stream'(N) where N is an integer to identify the stream. However, a user application should not explicitly create such a Prolog term as a Prolog stream, it should only be obtained through open/[3,4], open_null_stream/1, or stream_code/2. In C code, a Prolog stream is represented as a pointer to structure of type QP_stream. The formatting information along with bottom layer functions of a stream is stored in its QP_stream structure. A stream behavior depends on the values of fields in its QP_stream structure. The options specified in open/4 are converted and stored in the QP_stream structure created for the stream.

Selected fields in QP_stream that can be accessed and modified by an application are described in the remainder of the section. Most of these fields are option fields, which can be specified in open/4.

10.5.3.1 Filename of A Stream

```
open/[3,4]:    1st argument
QP_stream:     char *filename
```

The filename of a stream is recorded in the stream structure. If a stream does not have a filename, (e.g. a data communication channel), the filename field should be an empty string. After a stream is created, the Prolog system accesses this field only to get the corresponding filename of a stream.

10.5.3.2 Mode of A Stream

```
open/[3,4]:    2nd argument
QP_stream:     unsigned char mode
```

This field indicates whether the stream is created for input or output mode. The `mode` field is `QP_READ` for an input (read only) stream, `QP_WRITE` or `QP_APPEND` for an output stream. This field should not be changed for the lifetime of a stream.

10.5.3.3 Format of A Stream

```
open/4 option:  format(Format)
QP_stream:      unsigned char format
```

This field indicates the format of a stream. The `format` tells the middle layer functions how to wrap (unwrap) a record. Possible formats are:

`QP_VAR_LEN`
: specified as `format(variable)` in `open/4`. Each record in the file has its own length. The middle layer input function trims the trailing blank characters in each record if trimming is turned on for the stream.

  The input/output system does not alter any character in each record for a `QP_VAR_LEN` format stream with no trimming and no line border code.

`QP_DELIM_LF`
: specified as `format(delimited(lf))` in `open/4`. From an application program's point of view, each record in the file is terminated with a single ⟨LFD⟩. Under Windows, however, what's actually stored in the file is the sequence ⟨RET⟩⟨LFD⟩.

`QP_DELIM_TTY`
: specified as `format(delimited(tty))` in `open/4`. The file stream is a terminal device or a terminal emulator. What characters delimit each record depends on the host operating system. The Prolog input/output system treats this format like `QP_DELIM_LF` as far as record termination is concerned.

  The Prolog input/output system also automatically provides special services for streams with `QP_DELIM_TTY` format (see Section 10.5.4 [fli-ios-tty], page 444).

For a delimited record format stream, the middle layer input function removes the delimiting character at the end of each record and the line border code for the stream is returned to a top layer input predicate (function) when the end of a record is reached. The middle output layer output function adds the delimited character at the end of each record before the record is passed to the bottom layer write function.

The `format` field may be set to `QP_FMT_UNKNOWN` when a stream structure is created if the format to be used is not known yet, for example, because the underlying device is not yet opened. This format field must be set to a proper format before any I/O takes place on the stream.

An example of this is when opening a Prolog stream through `open/[3,4]` or `QP_fopen()` without specifying the format. The embedding open function, `QU_open()` is given the stream format `QP_FMT_UNKNOWN` and thus chooses an appropriate format for the stream based on the filename and options of the stream.

Depending on the host operating system, some formats may be used more frequently than others. `QP_DELIM_LF` and `QP_DELIM_TTY` are the most frequently used formats for a Prolog system running under UNIX or Windows. `QP_VAR_LEN` is more frequently used under VMS.

10.5.3.4 Maximum Record Length

```
open/4 option:  record(MaxRecLen)
QP_stream:      int max_reclen
```

This field indicates the maximum record length of the stream. It is usually also the buffer length of the stream. The value in this field is not changed once the stream is created. The maximum record length can be bigger than the value store in the `max_reclen` field for some operating systems allowing reading or writing partial records, such as the UNIX operating system.

To create an unbuffered output stream, the value in `max_reclen` must be set to 0. The bottom layer write function is then called for each character placed in the output stream. To create an unbuffered input stream, the value in `max_reclen` can be either 0 or 1.

10.5.3.5 Line Border Code

```
open/4 option:  end_of_line(LineBorder)
QP_stream:      int line_border
```

The `line_border` field can be any integer greater than or equal to `QP_NOLB` (-1). If the value is `QP_NOLB`, there is no line border code for the stream.

For an input stream, the line border code is the value to be returned in getting a character when a stream is positioned at the end of a record. (Notice that if the stream is a delimited record format stream, the delimited character has already been removed.) If there is no line border code, the first character in the next record is returned at the end of a record. Writing the line border code (i.e. with `put/[1,2]`) to an output stream signals the end of the current record. Instead of writing out the line border code, the record is appropriately wrapped up based on the format of the stream, and the wrapped record is handed to the bottom layer write function of the stream. This is just like a new line operation on the stream (e.g. `nl/[0,1]` or `QP_newln()`).

10.5.3.6 File Border Code

```
open/4 option:  end_of_file(FileBorder)
QP_stream:      int file_border
```

This is the code to be returned on reading at the end of file for an input stream. This field is not used for an output stream. The value in `file_border` field can be any integer greater than or equal to `QP_NOFB` (-2). If the value is `QP_NOFB`, there is no file border code for the stream. If the `file_border` field is `QP_NOFB`, reading at the end of file is the same as reading past the end of file.

10.5.3.7 Reading Past End Of File

```
open/4 option:  eof_action(Action)
QP_stream:      unsigned char peof_act
```

This field is only used for an input stream. There are three states for an input stream, normal, end of file, and past end of file. An input stream is in normal state until it reaches end of file, where the state is switched to end of file state. If there is no file border code or the file border code is consumed, the input stream is switched to past end of file state. The field `peof_act` specifies which action to take for reading from a stream at past end of file state. The value of `peof_act` can be one of the following.

`QP_SEEK_ERROR`
: specified as `seek(error)` in `open/4`. An error for any type of seeking in the stream.

`QP_PASTEOF_ERROR`
: specified as `eof_action(error)` in `open/4`. The `errno` field of the stream is set to `QP_E_EXHAUSTED`, and the read call fails.

`QP_PASTEOF_EOFCODE`
: specified as `eof_action(eof_code)` in `open/4`. Retuns the file border code for reading at the past end of file state. The state of the input stream does not change. If there is no file border code for the input stream, it is the same as setting the field to `QP_PASTEOF_ERROR`.

`QP_PASTEOF_RESET`
: specified as `eof_action(reset)` in `open/4`. Resets the state of an input stream to normal state and calls the bottom layer read function to get the input record for reading at past end of file state. Setting the field to this value may be useful for the stream with `QP_DELIM_TTY` format. It is possible to get more records from a tty device after the end of file character is typed.

Once an input stream reaches end of file state, its bottom layer read function will not be called unless the `peof_act` field is `QP_PASTEOF_RESET`.

10.5.3.8 Prompt String

```
QP_stream:      char *prompt
```

A prompt string is a character string that will be written out when a tty stream (a stream with `QP_DELIM_TTY` format) reads input at the beginning of a line. The prompt string is stored in this field. This field is only used for an input tty stream. The prompt string will not be written out if there are no output tty streams connected to the same tty as the input tty stream (see Section 10.5.4 [fli-ios-tty], page 444). A prompt string of a stream may be changed after the stream is created through `prompt/[2,3]` or by assigning a new character string to this field directly.

10.5.3.9 Record Trimming

```
open/4 option:  trim
QP_stream:      unsigned char trim
```

A non-zero value in `trim` field indicates trimming should be performed on each record of an input stream with

`QP_VAR_LEN` format. The trailing blank characters in each of the records are removed in the trimming operation. This field has no impact on an input stream with other formats or for any output stream.

10.5.3.10 Seek Type

```
open/4 option:  seek(SeekType)
QP_stream:      unsigned char seek_type
```

This field specifies which type of seeking can be performed on the stream. The possible values for `seek_type` are:

`QP_SEEK_ERROR`
: specified as `seek(error)` in `open/4`. It is an error to perform any type of seeking operation on the stream. Any call to `seek/4`, `stream_position/3`, `QP_seek()` or `QP_setpos()` is an error.

`QP_SEEK_PREVIOUS`
: specified as `seek(previous)` in `open/4`. The stream only seeks back to a previous read or written position. The previous position must be saved through `QP_getpos()` or `stream_position/[2,3]`. The seeking can only be performed through `stream_position/3` or `QP_setpos()`.

`QP_SEEK_BYTE`
: specified as `seek(byte)` in `open/4`. The stream can seek to any arbitrary byte offset in the file or to any previously saved position. In addition to the seeking method described in `QP_SEEK_PREVIOUS`, the stream can also seek through `seek/4` or `QP_seek()`.

`QP_SEEK_RECORD`
: specified as `seek(record)` in `open/4`. The stream can seek to an arbitrary record number in the file or to any previously saved position.

The seek function must be supplied for a user-defined stream with `seek_type` set to any value other than `QP_SEEK_ERROR`.

10.5.3.11 Flushing An Output Stream

```
open/4 option:  flush(FlushType)
QP_stream:      unsigned char flush_type
```

This field specifies whether or not the characters buffered in an output stream can be written out immediately as a partial record. It is not used for an input stream. Characters written to an output stream are buffered until the current record is terminated or the output buffer overflows. When an output record is terminated, it is passed to the bottom layer of the write function of the stream. The completed record is either written to the associated device of the stream or further buffered by the bottom layer write function. If the output buffer overflows, or `flush_output/1` is called, buffered characters may be forced out through flushing the output stream. The possible values for `flush_type` are:

`QP_FLUSH_ERROR`
: specified as `flush(error)` in `open/4`. It is an error to flush the output stream. Any call to `flush_output/1` or `QP_flush()` is an error.

`QP_FLUSH_FLUSH`
: specified as `flush(flush)` in `open/4`. Write out any characters in the output buffer to the associated device of the stream. The host system must permit writing out of partial records to support this option. The bottom layer flush function must be supplied for the output stream.

10.5.3.12 Output Stream Buffer Overflow

```
open/4 option:  overflow(OverFlowAction)
QP_stream:      unsigned char overflow
```

Written characters are stored in the buffer of an output stream by the middle layer output function until the current record is terminated, through a newline operation (such as `nl/[0,1]` or `QP_newln()`) or by writing the line border code of the stream. If a character is written when the buffer of an output stream is full, it overflows the output buffer. The `overflow` field specifies the action that the middle layer function should take if this happens. The possible values for `overflow` are:

`QP_OV_ERROR`
: specified as `overflow(error)` in `open/4`. It is an error when output stream buffer overflows.

`QP_OV_TRUNCATE`
: specified as `overflow(truncate)` in `open/4`. Keeps the characters in the buffer and throws away the current character that overflows the output buffer.

`QP_OV_FLUSH`
: specified as `overflow(flush)` in `open/4`. Pass the current buffer storing a partial record of the output stream to the bottom layer flush function to write out the buffer. The host operating system must support writing of partial records for the device associated with the stream.

Note that if an output stream is unbuffered (i.e. `max_reclen` is 0) then the middle layer function ignores the `overflow` field and calls the `write` function for each character written to the stream.

10.5.3.13 Storing Error Condition Of A Stream

```
QP_stream:      int errno;
```

An error code is stored in the `errno` field of a stream structure when an error is detected in any of the three layers of input/output functions. The top layer QP functions set the error code to `QP_errno` when an error occurs in the call to the functions. The error code stored in this field may not last more than two QP input/output functions calls since middle layer functions and some QP functions clear out this field before they call the bottom layer function. If an error is detected in the bottom layer function, the `errno` field should be given an appropriate error code before the function returns. The error code stored in `errno` field can be any of the host operating system error numbers, QP error numbers or user-defined error numbers.

10.5.3.14 System-Dependent Address In A File Stream

```
QP_stream:      union QP_cookie magic;
```

The system-dependent address of the current position in a stream is stored in the `magic` field of the stream structure. It is only used when there is any kind of seek to be performed on the stream. This field is a C type `union cookie`, which is defined in '`<quintus/quintus.h>`' as follows:

```
union QP_cookie {
      struct RFA {
             int  BlockNumber;
             short Offset;
      } vms_rfa;
      int mvs_rrn;
      int  cms_recno;
      off_t byteno;
      int user_defined[2];
} magic;
```

Depending on the host operating system, different field names of `union cookie` are used to store the position address of the stream depending on the host operating system. Under UNIX, `magic.byteno` is used to record the current location of the file pointer as an absolute byte offset from the beginning of the stream; `magic.vms_rfa` is used on VMS; `magic.mvs_rrn` is used on MVS; `magic.cms_recno` is used on CMS. `magic.user_defined` is used for a user's specific method of recording the current location of a stream. These values must be maintained in the bottom layer functions of a stream with seek permission.

10.5.3.15 Bottom Layer Functions

```
QP_stream:      int (*read)(), (*write)(), (*flush)(),
                    (*seek)(), (*close)();
```

These fields store the address of the corresponding bottom layer functions of the stream. See Section 10.5.6 [fli-ios-bot], page 451 for description about how to define these functions.

10.5.4 TTY Stream

A Prolog stream is a tty stream if the format of the stream is `QP_DELIM_TTY`. A tty stream is normally associated with a terminal device, a pseudo-terminal device or a terminal emulator. A set of tty streams can be grouped together through a distinct character string key for each group of tty streams. All the tty streams from the same tty device (emulator) should normally be grouped together. A tty stream registers itself to a tty stream group by calling `QP_add_tty()` with the specific character string key for the group.

There are two services provided automatically by Prolog I/O system to each tty stream group. When a tty stream is closed, it is automatically removed from its tty group.

Prompt Handling
: There must be at least one output stream in the tty stream group in order to write out the prompt string of an input stream in the group. When an input tty stream reads at the beginning of a line, the middle layer input function writes out the prompt of the input stream to the latest registered output stream in the tty group before the bottom layer of the read function of the input stream is called.

Stream Position
: Character count, line count and line position for each stream in the tty group are automatically adjusted for each tty group. When the buffer of an output stream is written out (such as the output line is terminated, the buffer overflows, or the stream output is flushed), the counts of all the output streams in the tty group are brought up to date. When an input streams reads input to its buffer, the counts of all the streams in its tty group are updated to the current counts. Linking the counts of tty streams in the tty group makes the count of a tty stream correspond to its physical appearance on the tty device.

A sample Prolog session demonstrates the special services performed for tty streams. The default `open/[3,4]` automatically registers tty streams to the tty group using filename as the key. After writing '`write\n`' to `Output1`, the counts for `Output1` and `Output2` are brought up to date. The counts in Input1 is not changed since counts in input stream are only updated when reading from the input stream. After reading from Input1, the counts for all the three streams are updated. The prompt '`INPUT>>` ' is written out either through `Output1` or `Output2`, so it is included in the counts. The count in Input1 is different from `Output1` and `Output2` since only character '`r`' is consumed in the input of '`read\n`'.

```
| ?- compile(user).
| write_count(Input, Output1, Output2) :-
     character_count(Input, C0), line_count(Input, L0),
         line_position(Input, P0),
     character_count(Output1, C1), line_count(Output1, L1),
         line_position(Output1, P1),
     character_count(Output2, C2), line_count(Output2, L2),
         line_position(Output2, P2),
     format('input   : ~d, ~d, ~d~n', [C0, L0, P0]),
     format('output1 : ~d, ~d, ~d~n', [C1, L1, P1]),
     format('output2 : ~d, ~d, ~d~n', [C2, L2, P2]).
| ^D
% user compiled in module user, 0.216 sec 384 bytes

yes
| ?- open('/dev/tty', read, Input), prompt(Input, _, 'INPUT>> '),
     open('/dev/tty', write, Output1),
     open('/dev/tty', write, Output2),
     format(Output1, 'write~n', []),
     write_count(Input, Output1, Output2),
     get0(Input, _), write_count(Input, Output1, Output2).
write
input   : 0, 1, 0
output1 : 6, 2, 0
output2 : 6, 2, 0
INPUT>> read
input   : 15, 2, 9
output1 : 19, 3, 0
output2 : 19, 3, 0
```

Notice that the I/O in the `user_input` and `user_output` are not included in the counts although both streams are connected to the same tty. The three default streams (`user_input`, `user_output` and `user_error`) are put into a different tty group in the embedding initialization function, `QU_initio()`.

10.5.5 Defining A Customized Prolog Stream

In addition to calling default open predicates or functions — such as `open/[3,4]` or `QP_fopen()` to create a stream, a user can define a stream through the method described in this section. A common reason to do this is to create a stream, which is not supported directly by the Prolog input/output system, such as a stream for inter-process communication. The created stream can be passed as the stream argument to all the input/output related Prolog predicates and functions. This section presupposes Section 10.5.3 [fli-ios-sst], page 437 on stream structure.

10.5.5.1 Summary of Steps

The following steps are required to create a user-defined stream in foreign code, such as C. The stream is represented in C as a pointer to a `QP_stream` structure. It can then be converted back to Prolog stream representation through `stream_code/2`. The predicate `stream_code/2` converts, in either direction, between Prolog and C representations of a stream.

1. Define the user-defined stream structure, containing fields required to operate the stream. (see Section 10.5.5.2 [fli-ios-cps-sst], page 446)
2. Prepare creation of the user-defined stream. This usually requires a function to perform the following steps:
   a. Open the I/O channel, e.g. open a file or set up inter-process communication. (see Section 10.5.5.3 [fli-ios-cps-opn], page 447)
   b. Allocate memory for the user-defined stream and set values in the fields of the allocated user-defined stream. (see Section 10.5.5.4 [fli-ios-cps-all], page 448)
   c. Set up the default values for the `QP_stream` part of the user-defined stream through `QU_stream_param()` and modify these values as necessary. (see Section 10.5.5.5 [fli-ios-cps-sqs], page 449)
   d. Initialize the remaining fields of `QP_stream` structure used internally by the Prolog system through `QP_prepare_stream()` and register the created stream through `QP_register_stream()`. (see Section 10.5.5.6 [fli-ios-cps-ire], page 450)
   e. If the stream is a tty stream, register the stream to its tty group through `QP_add_tty()`. (see Section 10.5.5.7 [fli-ios-cps-tty], page 451)
3. Implement the bottom layer functions to be used for the stream. These may include read, write, flush, seek and close functions. (see Section 10.5.6 [fli-ios-bot], page 451)

These steps are described in more detail in the remainder of this section. An example of creating a stream for a binary file in one of read, write or append modes is discussed. The example is written in C although it can also be written in other languages, such as Pascal or Fortran.

The example opens a file as a binary stream. The characters input from or output to the stream are exactly the same as stored in file. Seeking to a random byte position and flushing output are permitted in the stream. The first example lists complete source code (see Section 10.5.7 [fli-ios-uds], page 457). Note that binary streams are in fact supported in the system.

10.5.5.2 Defining a Stream Structure

The first field of the user-defined stream structure must be of type `QP_stream`. Other fields in the user-defined stream structure can be anything that is required to operate on the user-defined stream. The Prolog input/output system passes a `QP_stream` pointer as the first argument to the bottom layer functions; casting this to the user-defined stream structure

enables other fields in the user-defined stream to be accessed. The example below declares a binary stream structure as:

```
typedef struct
    {
        QP_stream qpinfo;
        int fd;                 /* UNIX file descriptor */
        int last_rdsize;        /* size of last returned record */
        unsigned char buffer[Buffer_Size];      /* I/O buffer */
    } BinStream;

#define CoerceBinStream(x) ((BinStream *)(x))
```

The field `qpinfo` stores information about the binary stream known to the Prolog input/output system. There is a `buffer` field in the structure since the I/O buffer is allocated by the user. The macro `CoerceBinStream` is used to convert a pointer to `QP_stream` into a pointer to `BinStream`. We use this macro to convert the pointer so that fields in the `BinStream` structure can be accessed.

10.5.5.3 Opening The User-Defined Stream

Depending on the specific user-defined stream, there are different operations needed for the stream. A stream that operates on a file needs to open the file; a stream that operates for inter-process communication needs to build the connection to different process. Our example stream operates on files, so we just open the file to get file descriptor. The parameters of our function and local variables in the function are also listed.

```
QP_stream *
open_bin(filename, modename, error_num)
    char   *filename, *modename;
    int    *error_num;
    {
        BinStream       *stream;
        QP_stream       *option;
        int             fd, mode;

        switch (*modename) {
        case 'r': mode = QP_READ;
                  fd = open(filename, O_RDONLY, 0000);
                  break;
        case 'w': mode = QP_APPEND;
                  fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC,
                                      0666);
                  break;
        case 'a': mode = QP_APPEND;
                  fd = open(filename, O_WRONLY|O_CREAT, 0666);
                  break;
        default:  *error_num = QP_E_BAD_MODE;
                  return QP_NULL_STREAM;
        }
        if (fd < 0) {
            *error_num = errno;
            return QP_NULL_STREAM;
        }

        ......  allocate space and set user-stream fields  ......
        ......  set up QP_stream structure fields  ......
        ......  register the created QP_stream  ......

        return &stream->qpinfo;
    }
```

This function can be called from Prolog using the Prolog calling C interface described in Section 10.3 [fli-p2f], page 375. The address returned by this function is converted into the Prolog representation of a stream using `stream_code/2`.

10.5.5.4 Allocating Space And Setting Field Values For the User-Defined Stream

The memory space for a user-defined stream structure and its buffer are controlled by the user application. It is recommended to use `QP_malloc()` to allocate the space for more efficient memory utilization in the Prolog system. In our example, the buffer is a field in

the `BinStream` structure so that only one `QP_malloc()` call allocates both buffer and stream space.

After memory is allocated, the fields in the stream structure are set appropriately. The fields of `QP_stream` part in the stream structure is set up in the next step.

```
if ((stream = (BinStream *) QP_malloc(sizeof(*stream)))
                == ((BinStream *) 0) ) {
    (void) close(fd);
    *error_num = QP_errno;
    return QP_NULL_STREAM;
}
stream->fd = fd;
stream->last_rdsize = 0;
```

10.5.5.5 Setting Up The QP_stream Structure

The default values in the fields of `QP_stream` part of the user-defined stream are set through the `QU_stream_param()` function. The declaration of `QU_stream_param()` is given as:

```
void QU_stream_param(filename, mode, format, option)
    char        *filename;
    int         mode;
    int         format;
    QP_stream   *option;
```

If the stream does not have a filename, the empty string `""` should be used. The parameter `mode` can be one of `QP_READ`, `QP_WRITE` or `QP_APPEND`. The parameter `format` can be one of the format types listed in Section 10.5.3.3 [fli-ios-sst-fmt], page 438. The default version of `QU_stream_param()` source code is shipped with Quintus Prolog (see Section 19.3.77 [cfu-ref-QU_stream_param], page 1472 also lists the source).

The fields in the `QP_stream` structure can then be modified based on the desired features of the user-defined stream. All the fields described in the Stream Structure section can be modified (see Section 10.5.3 [fli-ios-sst], page 437), but often the only modified fields are `max_reclen`, `seek_type` and bottom layer function fields.

In our example, the format `QP_VAR_LEN` is chosen for non-tty files, and the `line_border` field is reset so that the middle layer functions do not alter any of the input/output records. The fields `max_reclen` and `seek_type` are set to the right values for our stream. The bottom layer function fields are set based on the mode and the `seek_type` of the stream. If the stream is opened for append, the file pointer of the stream is moved to the end of file and the `magic` field is updated (`magic.byteno` is used since it is a UNIX file).

```
option = &stream->qpinfo;
QU_stream_param(filename, mode, QP_VAR_LEN, option);
option->max_reclen = Buffer_Size;
option->line_border = QP_NOLB;
if (isatty(fd)) {
    option->format = QP_DELIM_TTY;
    option->seek_type = QP_SEEK_ERROR;
} else {
    option->seek_type = QP_SEEK_BYTE;
    option->seek = bin_seek;
}
if (mode != QP_READ) {
    option->write = bin_write;
    option->flush = bin_write;
} else
    option->read = bin_read;
if (option->mode == QP_APPEND &&
                    option->format != QP_DELIM_TTY) {
    if ((option->magic.byteno=lseek(fd,0L,L_XTND)) < 0) {
        (void) close(fd);
        *error_num = errno;
        return QP_NULL_STREAM;
    }
}
option->close = bin_close;
```

10.5.5.6 Initialize and Register The Created Stream

The fields of `QP_stream` structure used internally by the Prolog system are initialized through `QP_prepare_stream()`. It should be called after other fields in `QP_stream` are properly set up. `QP_prepare_stream()` takes a pointer to `QP_stream` as its first parameter and the address of the input/output buffer for the stream as its second parameter.

`QP_register_stream()` is then called to register the user-defined stream so that the stream can be used in Prolog code. In our example, if the registration fails the bottom layer function is used to close the opened file and deallocate the memory space for the created stream and a null stream is returned.

```
QP_prepare_stream(&stream->qpinfo, stream->buffer);
if (QP_register_stream(&stream->qpinfo) == QP_ERROR)
{   (void) stream->qpinfo.close(&stream->qpinfo);
    *error_num = QP_errno;
    return QP_NULL_STREAM;
}
```

10.5.5.7 TTY Group For TTY Stream

This is an optional step only for tty streams. A tty stream needs to register to its group for special tty service (see Section 10.5.4 [fli-ios-tty], page 444).

Finally the pointer to the created stream is returned to Prolog, and converted to Prolog stream representation through `stream_code/2`. In our example, we use the filename of the stream as the key to register into its group.

```
if (option->format == QP_DELIM_TTY)
    (void) QP_add_tty(&stream->qpinfo, filename);

return &stream->qpinfo;
```

10.5.6 The Bottom Layer Functions

There are five bottom layer functions for a stream: read, write, flush, seek and close. However, not all of these functions are needed for every stream:

- The `seek` function is not required when the `seek_type` is `QP_SEEK_ERROR`.
- The `read` function is only required for an input stream.
- The `write` function is only required for an output stream.
- The `flush` function is not required when the `flush_type` is `QP_FLUSH_ERROR`.
- The `close` function should be supplied for every stream.

These functions usually only operate on the specific fields of a particular user-defined stream. (i.e. Fields other than the first field in a user-defined stream structure.) The `errno` field and `magic` field in `QP_stream` part may also be maintained by the bottom layer functions. The `mode` field and `max_reclen` field are also typically accessed by the bottom layer functions.

Except for the read function, all the bottom layer functions return `QP_SUCCESS` upon success, and return `QP_ERROR` and assign an error code to the `errno` field upon failure. These functions are described further in subsequent sub-sections.

10.5.6.1 The Bottom Layer Read Function

```
int <read function>(qpstream, bufptr, sizeptr)
       QP_stream        *qpstream;
       unsigned char   **bufptr;
       size_t           *sizeptr;

Return Values:   QP_FULL  : a complete record is read
                 QP_PART  : a partial  record is read
                 QP_EOF   : end of file is reached
                 QP_ERROR : a partial  record is read
```

The bottom layer read function returns a record of input to its caller. The returned record is buffered. The buffer address is returned through `*bufptr` parameter and the size of the returned record is stored in `*sizeptr` parameter. The `magic` field in `qpstream` should be updated to the system-dependent file address (see Section 10.5.3.14 [fli-ios-sst-sda], page 443) for the beginning of the returned record. If there is no seek permission for the stream, the `magic` field may be ignored. The `errno` field in `QP_stream` stores the error code if an error is detected in the function.

In our example, the read function does not return `QP_PART` since any length of input is chosen as a complete record.

```
static int
bin_read(qpstream, bufptr, sizeptr)
    QP_stream            *qpstream;
    unsigned char       **bufptr;
    size_t               *sizeptr;
    {
        int n;
        register BinStream *stream = CoerceBinStream(qpstream);

        qpstream->magic.byteno += stream->last_rdsize;
        stream->last_rdsize = 0;
        n = read(stream->fd, (char*) stream->buffer,
                             (int) qpstream->max_reclen);
        if (n > 0) {
            *bufptr  = stream->buffer;
            *sizeptr = n;
            stream->last_rdsize = n;
            return QP_FULL;
        } else if (n == 0) {
            *sizeptr = 0;
            return QP_EOF;
        } else {
            qpstream->errno = errno;
            return QP_ERROR;
        }
    }
```

10.5.6.2 The Bottom Layer Write Function

```
int <write function>(qpstream, bufptr, sizeptr)
       QP_stream         *qpstream;
       unsigned char    **bufptr;
       size_t            *sizeptr;

Return Values:   QP_SUCCESS
                 QP_ERROR
```

The bottom layer write function writes out a record from buffer address stored in `*bufptr` and the size of the record stored in `*sizeptr`. Upon successful return, `*sizeptr` stores the maximum record size and `*bufptr` stores the address of the beginning of the buffer for the next output record. The `magic` field in `qpstream` should be updated to the system-dependent file address (see Section 10.5.3.14 [fli-ios-sst-sda], page 443) for the beginning of the next output record. If there is no seek permission for the stream, the `magic` field may be ignored. The `errno` field in `QP_stream` stores the error code if an error is detected in the function and `QP_ERROR` is returned. The output record passed into the write function may be a partial record if output record overflows the output buffer for a stream that permits overflow.

```
static int
bin_write(qpstream, bufptr, sizeptr)
    QP_stream            *qpstream;
    unsigned char       **bufptr;
    size_t               *sizeptr;
    {
        BinStream  *stream = CoerceBinStream(qpstream);
        int        n, len=(int) *sizeptr;
        char        *buf = (char *) *bufptr;

        while ((n = write(stream->fd, buf, len)) > 0 && n < len) {
            buf += n;
            len -= n;
        }
        if (n >= 0) {
            qpstream->magic.byteno += *sizeptr;
            *sizeptr = qpstream->max_reclen;
            *bufptr  = stream->buffer;
            return QP_SUCCESS;
        } else {
            qpstream->errno = errno;
            return QP_ERROR;
        }
    }
```

10.5.6.3 The Bottom Layer Flush Function

```
int <flush function>(qpstream, bufptr, sizeptr)
       QP_stream         *qpstream;
       unsigned char    **bufptr;
       size_t            *sizeptr;

Return Values:   QP_SUCCESS
                 QP_ERROR
```

The parameters and the return values of the flush function have the same syntax and the same meaning as the write function. The write function may buffer the output record without writing the record out. The flush function should write out the output record immediately when it is called. The middle layer function will not call the write function with an empty record (`*sizeptr` is zero), but the flush function may be called with an empty record passed in. In general, the flush function can be the same as write function unless the write function buffers output records. An output stream needs a bottom layer flush function only if `flush_type` of the stream is not `FLUSH_ERROR`.

In our example, the bottom layer write function does not buffer output record and it can also handle writing an empty record, so the bottom layer flush function is the same as the write function.

10.5.6.4 The Bottom Layer Seek Function

```
int <seek function>(qpstream, qpmagic, whence, bufptr, sizeptr)
       QP_stream        *qpstream;
       union QP_cookie  *qpmagic;
       int               whence;
       unsigned char   **bufptr;
       size_t           *sizeptr;

Return Values:   QP_SUCCESS
                 QP_ERROR
```

The bottom layer seek function sets the stream `qpstream` to a new position based on the method `whence` and the system-dependent file address `qpmagic` specified in the parameters. The output parameter `*bufptr` stores the beginning of the buffer and `*sizeptr` stores the size of record. When the bottom layer seek function is called, the `magic` field of `qpstream` is the current stream position known to the user of the stream. (It does not include the unconsumed characters in the buffer.) Upon success, the seek function should return `QP_SUCCESS` and the `magic` field of `qpstream` should be updated to the new position. Upon failure, it returns `QP_ERROR` and suitable error code should be assigned to `error` field of `qpstream`.

The stream is set to a new position based on the `whence` value and `qpmagic` values. (see Section 10.5.3.14 [fli-ios-sst-sda], page 443)

- If `whence` is `QP_BEGINNING`, the `magic` field is the system-dependent address to be positioned from the beginning of the stream. If the `seek_type` of the stream is `QP_SEEK_PREVIOUS`, the `whence` value is always `QP_BEGINNING`.
- If `whence` is `QP_CURRENT`, seeking is to be performed from the current position. For instance, if `byteno` is used for `magic` field of `qpstream`, the stream should be set to the current position (`qpstream->magic.byteno`) plus the offset specified in `qpmagic->byteno`.
- If `whence` is `QP_END`, seeking is to be performed from the end of file position associated with `qpstream`. For instance, if `byteno` is used for `magic` field of `qpstream`, the stream

should be set to the size of the file associated with `qpstream` plus the offset specified in `qpmagic->byteno`.

Due to the buffering mechanism of a stream, the `magic` field in `qpstream` might be different from the actual position for the file (or other devices) associated with the stream. For example, if the current record of an input stream has a size of 10 and there are only 5 characters consumed, the `magic` field indicates the position at the sixth character rather than the 11th character of the current record. In short, the value in `magic` field of `qpstream` does not count any characters in the buffer that are not consumed in an input stream. For an output stream, the caller of the bottom layer seek function will call flush function to flush output prior to calling this function if `qpstream` permits flushing (`flush_type` is not `QP_FLUSH_ERROR`). The caller of this function does not permit any seeking in an output stream with no flush permission if there are characters in the current record (line position is not zero in the output stream). However, if the bottom layer of an output stream without flushing permission buffers more than one output record, it is possible for the bottom layer seek function to be called with records in the buffer. (This would be the case that there are no characters in the current record.) The bottom layer seek function should write out the records in the buffer for this case.

One effect of seeking is to clear out the buffer of a stream. This should be adhered to in implementing the bottom layer seek function. If `qpstream` is an input stream, `bufptr` and `sizeptr` have the same meaning as in the bottom layer read function. If `qpstream` is an output stream, `bufptr` and `sizeptr` have the same meaning as in the bottom layer write function. So for most of cases, `*bufptr` should be set to the beginning of the buffer of the stream and `*sizeptr` should be set as follows:

```
*sizeptr = (qpstream->mode == QP_READ) ? 0 : qpstream->max_reclen;
```

The bottom layer seek function for our example stream:

```
static int
bin_seek(qpstream, qpmagic, whence, bufptr, sizeptr)
     QP_stream          *qpstream;
     union QP_cookie    *qpmagic;
     int                whence;
     unsigned char      **bufptr;
     size_t             *sizeptr;
     {
        BinStream *stream = CoerceBinStream(qpstream);
        off_t        new_offset;

        switch (whence) {
        case QP_BEGINNING:
            new_offset = lseek(stream->fd,qpmagic->byteno,L_SET);
            break;
        case QP_CURRENT:
            /* The current location of file pointer is different
               from what the user thinks it is due to buffering.
               The magic field has been brought up to date by the
               caller of this function already, so just seek to
               that position first. */
            if (lseek(stream->fd, qpstream->magic.byteno, L_SET)
                                                  == -1) {
                qpstream->errno = errno;
                return QP_ERROR;
            }
            new_offset = lseek(stream->fd,qpmagic->byteno,L_INCR);
            break;
        case QP_END:
            new_offset = lseek(stream->fd,qpmagic->byteno,L_XTND);
            break;
        default:
            qpstream->errno = QP_E_INVAL;
            return QP_ERROR;
        }
        if (new_offset == -1) { /* error in seeking */
            qpstream->errno = errno;
            return QP_ERROR;
        }

        qpstream->magic.byteno = new_offset;
        *bufptr  = stream->buffer;
        *sizeptr = (qpstream->mode == QP_READ) ? 0
                                  : qpstream->max_reclen;
        stream->last_rdsize = 0;
        return QP_SUCCESS;
     }
```

10.5.6.5 The Bottom Layer Close Function

```
int <close function>(qpstream)
        QP_stream   *qpstream;

Return Value:   QP_SUCCESS
                QP_ERROR
```

The bottom layer close function performs the specific close operation of a user-defined stream and deallocates the memory space for the stream. It returns `QP_ERROR` and assigns an appropriate error code to the `errno` field of `QP_stream` if an error occurs in the function. In our example, we use `QP_free()` to deallocate memory space since the memory is allocated by `QP_malloc()`.

```
static int
bin_close(qpstream)
    QP_stream   *qpstream;
    {
        BinStream *stream = CoerceBinStream(qpstream);
        int fd = stream->fd;

        if (close(fd) < 0) {
            qpstream->errno = errno;
            return QP_ERROR;
        }
        (void) QP_free(qpstream);
        return QP_SUCCESS;
    }
```

10.5.7 Examples Of User-Defined Streams

Three examples of creating a user-defined stream are listed in this section. The foreign code examples are written in C language under UNIX operating system.

10.5.7.1 Creating A Binary Stream

This example creates a binary stream. All the characters read from the stream are exactly the same as the characters stored in the file of the stream. All the characters stored in the file of the stream are the same as the characters written to the stream. The stream permits flushing output and random seek to arbitrary byte position in the file. By choosing `QP_VAR_LEN` as the format of the stream and using the full buffer as a record to communicate between middle layer and bottom layer functions, a line is actually a full buffer of the stream. A newline operation does not output a ⟨LFD⟩ either so that the line count of the stream is not based on the number of ⟨LFD⟩ characters.

bin.pl

```
foreign(open_bin, c, open_bin(+string, +string, -integer, [-address])).
foreign_file('bin', [open_bin]).

:- load_foreign_files(['bin'],[]).

open_bin_file(FileName, ModeName, Stream) :-
        open_bin(FileName, ModeName, ErrorNum, CStream),
        ( CStream =:= 0 ->
                raise_exception(existence_error(
                    open_bin_file(FileName, ModeName, Stream),
                    1, file, FileName, errno(ErrorNum)))
        ; stream_code(Stream, CStream)
        ).
```

bin.c

```
#include <fcntl.h>
#include <errno.h>
#include <sys/file.h>           /* for seek */

#ifndef L_SET
#define L_SET   0
#endif
#ifndef L_INCR
#define L_INCR  1
#endif
#ifndef L_XTND
#define L_XTND  2
#endif

#include <quintus/quintus.h>

extern  char *QP_malloc();

/*  The following three functions support UNIX I/O on files
    without breaking things into records.  All the characters
    read from or written to the file are kept exactly the same.
*/

#define Buffer_Size             8192

typedef struct
    {
        QP_stream qpinfo;
        int fd;                 /* UNIX file descriptor */
        int last_rdsize;        /* size of last returned line */
        unsigned char buffer[Buffer_Size];      /* I/O buffer */
    } BinStream;

#define CoerceBinStream(x) ((BinStream *)(x))
```

bin.c

```
static int
bin_read(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        int n;
        register BinStream *stream = CoerceBinStream(qpstream);

        qpstream->magic.byteno += stream->last_rdsize;
        stream->last_rdsize = 0;
        n = read(stream->fd, (char*)stream->buffer,
                             (int)  qpstream->max_reclen);
        if (n > 0) {
            *bufptr  = stream->buffer;
            *sizeptr = n;
            stream->last_rdsize = n;
            return QP_FULL;
        } else if (n == 0) {
            *sizeptr = 0;
            return QP_EOF;
        } else {
            qpstream->errno = errno;
            return QP_ERROR;
        }
    }

static int
bin_write(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        BinStream  *stream = CoerceBinStream(qpstream);
        int        n, len=(int) *sizeptr;
        char       *buf = (char *) *bufptr;

        while ((n = write(stream->fd, buf, len)) > 0 && n < len) {
            buf += n;
            len -= n;
        }
        if (n >= 0) {
            qpstream->magic.byteno += *sizeptr;
            *sizeptr = qpstream->max_reclen;
            *bufptr  = stream->buffer;
            return QP_SUCCESS;
        } else {
            qpstream->errno = errno;
            return QP_ERROR;
        }
    }
```

bin.c

```
static int
bin_seek(qpstream, qpmagic, whence, bufptr, sizeptr)
     QP_stream          *qpstream;
     union QP_cookie    *qpmagic;
     int                whence;
     unsigned char      **bufptr;
     size_t             *sizeptr;
     {
        BinStream *stream = CoerceBinStream(qpstream);
        off_t           new_offset;

        switch (whence) {
        case QP_BEGINNING:
            new_offset = lseek(stream->fd,qpmagic->byteno,L_SET);
            break;
        case QP_CURRENT:
        /* The current location of file pointer is different from
           what the user thinks it is due to buffering.  The magic
           field has been brought up to date by the caller of this
           function, so just seek to that position first. */
            if (lseek(stream->fd, qpstream->magic.byteno, L_SET)
                                          == -1) {
                qpstream->errno = errno;
                return QP_ERROR;
            }
            new_offset = lseek(stream->fd,qpmagic->byteno,L_INCR);
            break;
        case QP_END:
            new_offset = lseek(stream->fd,qpmagic->byteno,L_XTND);
            break;
        default:
            qpstream->errno = QP_E_INVAL;
            return QP_ERROR;
        }
        if (new_offset == -1) { /* error in seeking */
            qpstream->errno = errno;
            return QP_ERROR;
        }
        qpstream->magic.byteno = new_offset;
        *bufptr  = stream->buffer;
        *sizeptr = (qpstream->mode == QP_READ) ? 0
                              : qpstream->max_reclen;
        stream->last_rdsize = 0;
        return QP_SUCCESS;
     }
```

bin.c

```
static int
bin_close(qpstream)
    QP_stream   *qpstream;
    {
        BinStream *stream = CoerceBinStream(qpstream);
        int fd = stream->fd;

        if (close(fd) < 0) {
            qpstream->errno = errno;
            return QP_ERROR;
        }
        (void) QP_free(qpstream);
        return QP_SUCCESS;
    }
```

bin.c

```
QP_stream *
open_bin(filename, modename, error_num)
    char   *filename, *modename;
    int    *error_num;
    {
        BinStream       *stream;
        QP_stream       *option;
        int             fd, mode;

        switch (*modename) {
        case 'r': mode = QP_READ;
                  fd = open(filename, O_RDONLY, 0000);
                  break;
        case 'w': mode = QP_APPEND;
                  fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC,
                                      0666);
                  break;
        case 'a': mode = QP_APPEND;
                  fd = open(filename, O_WRONLY|O_CREAT, 0666);
                  break;
        default:  *error_num = QP_E_BAD_MODE;
                  return QP_NULL_STREAM;
        }
        if (fd < 0) {
            *error_num = errno;
            return QP_NULL_STREAM;
        }
        if ((stream = (BinStream *) QP_malloc(sizeof(*stream)))
                      == ((BinStream *) 0) ) {
            (void) close(fd);
            *error_num = QP_errno;
            return QP_NULL_STREAM;
        }
        stream->fd = fd;
        stream->last_rdsize = 0;
```

bin.c

```
            /* obtain default values in QP_stream structure */
            /*     and modified fields for this stream */
        option = &stream->qpinfo;
        QU_stream_param(filename, mode, QP_VAR_LEN, option);
        option->max_reclen = Buffer_Size;
        option->line_border = QP_NOLB;
        if (isatty(fd)) {
            option->format = QP_DELIM_TTY;
            option->seek_type = QP_SEEK_ERROR;
        } else {
            option->seek_type = QP_SEEK_BYTE;
            option->seek = bin_seek;
        }
        if (mode != QP_READ) {
            option->write = bin_write;
            option->flush = bin_write;
        } else
            option->read = bin_read;
        if (option->mode == QP_APPEND
                  && option->format != QP_DELIM_TTY) {
            if ((option->magic.byteno=lseek(fd,0L,L_XTND)) < 0) {
                (void) close(fd);
                *error_num = errno;
                return QP_NULL_STREAM;
            }
        }
        option->close = bin_close;

        QP_prepare_stream(&stream->qpinfo, stream->buffer);
        if (QP_register_stream(&stream->qpinfo) == QP_ERROR) {
            (void) stream->qpinfo.close(&stream->qpinfo);
            *error_num = QP_errno;
            return QP_NULL_STREAM;
        }

        /* Use filename to register tty stream to its group */
        if (option->format == QP_DELIM_TTY)
            (void) QP_add_tty(&stream->qpinfo, filename);

        return &stream->qpinfo;
    }
```

10.5.7.2 Creating A Stream To Read An Encrypted File

This example creates an input stream to read from a file encrypted using a simple encryption algorithm. The key is stored in the first byte of the file. A character code stored in the file is the result of a logical exclusive-or operation of the output character and the key. The decryption of the input file is done in the bottom layer read function.

The input stream created only permits seeking to a previous read position. Notice the bottom layer read function defined (`decrypt_read()`) buffers more than one record. By doing this, the Prolog input/output system will maintain a correct line count and line position based on the new line character (‘`\n`’). There are also two user-defined error numbers used in this example (`DECRYPT_NO_KEY` and `DECRYPT_TTY_FILE`).

decrypt.pl

```
foreign(open_decrypt, c, open_decrypt(+string, -integer,
                                      [-address])).
foreign_file('decrypt', [open_decrypt]).

:- load_foreign_files(['decrypt'],['-lc']).

open_decrypt_stream(FileName, PrologStream) :-
        open_decrypt(FileName, ErrorNum, CStream),
        ( CStream =:= 0 ->
                raise_exception(existence_error(
                    open_decrypt_stream(FileName, PrologStream),
                    1, file, FileName, errno(ErrorNum)))
        ; stream_code(PrologStream, CStream)
        ).
```

decrypt.c

```
#include <fcntl.h>
#include <quintus/quintus.h>
extern int errno;

#define Buffer_Size             8192

typedef unsigned char           Key_Type;

typedef struct
    {
        QP_stream       qpinfo;
        int             fd;          /* file descriptor */
        int             last_rdsize; /* size of last record */
        int             left_size;   /* char. left unread */
        unsigned char   *left_ptr;   /* pointer to the unread */
        unsigned char   buffer[Buffer_Size+3];
        Key_Type        key;            /* decryption key */
    } DecryptStream;

#define CoerceDecryptStream(qpstream) \
                           ((DecryptStream *)(qpstream))

/* define user-defined error number */
#define DECRYPT_NO_KEY          QP_END_ECODE+1
#define DECRYPT_TTY_FILE        QP_END_ECODE+2
```

decrypt.c

```
/*
     To enable the Prolog system to maintain correct line count
     and line position, a whole buffer is read but only a line
     in the buffer is returned every time.
     The characters in the buffer are decrypted at once.

     The buffer is maintained as follows:

                    <- left_size ->
     +---------------+-------------+--+-------+
     | has been read | to be read  |\n| empty |
     +---------------+-------------+--+-------+
                    ^ left_ptr      ^ <pad '\n' character>
*/
static int
decrypt_read(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        register DecryptStream *stream =
                              CoerceDecryptStream(qpstream);
        register int            n;
        register unsigned char  *s, *s1;

        /* magic is the beginning byte offset of return record */
        qpstream->magic.byteno += stream->last_rdsize;

        if (stream->left_size <= 0) {
            register Key_Type   *kp, *kq, key;

            /* read a new buffer of input and decrypt characters*/
            n = read(stream->fd, (char *) stream->buffer,
                                  Buffer_Size);

            if (n > 0) {
                kp=(Key_Type *) stream->buffer;
                kq=(Key_Type *) &stream->buffer[n];
                for (key = stream->key; kp < kq ; ) /* decrypt */
                    *kp++ ^= key;
                stream->left_size = n;
                stream->left_ptr  = stream->buffer;
            } else if (n == 0) {
                stream->last_rdsize = stream->left_size = 0;
                *bufptr = stream->left_ptr = stream->buffer;
                *sizeptr = stream->last_rdsize = 0;
                return QP_EOF;
            } else {
                qpstream->errno = errno;
                stream->last_rdsize = stream->left_size = 0;
                return QP_ERROR;
```

decrypt.c

```
        /*  make next line of data available  */
        s = stream->left_ptr;
        se = s + stream->left_size;
        while (s < se) {
            if (*s++ == '\n') {         /* found end of record */
                break;
            }
        }

        *bufptr = stream->left_ptr;
        *sizeptr = stream->last_rdsize = s - stream->left_ptr;
        stream->left_ptr   = s;
        stream->left_size  = se - s;
        return (*--s == '\n') ? QP_FULL : QP_PART;
    }

/* Only QP_SEEK_PREVIOUS is allowed for the file, so 'whence'
   specified can only be QP_BEGINNING.  '*sizeptr' should always
   be set to 0 since there is only input stream.
*/
static int
decrypt_seek(qpstream, qpmagic, whence, bufptr, sizeptr)
     QP_stream          *qpstream;
     union QP_cookie    *qpmagic;
     int                whence;
     unsigned char      **bufptr;
     size_t             *sizeptr;
     {
        DecryptStream *stream = CoerceDecryptStream(qpstream);
        off_t           offset;

        switch (whence) {
        case QP_BEGINNING:
            if ((offset = lseek(stream->fd,qpmagic->byteno,L_SET))
                                            == -1) {
                qpstream->errno = errno;
                return QP_ERROR;
            }
            qpstream->magic.byteno = offset;
            *bufptr  = stream->buffer;
            *sizeptr = 0;
            stream->left_ptr  = stream->buffer;
            stream->left_size = stream->last_rdsize = 0;
            return QP_SUCCESS;
        case QP_CURRENT:
        case QP_END:
        default:
            qpstream->errno = QP_E_INVAL;
            return QP_ERROR;
        }
     }
```

decrypt.c

```
static int
decrypt_close(qpstream)
    QP_stream   *qpstream;
    {
        DecryptStream *stream = CoerceDecryptStream(qpstream);
        int fd = stream->fd;

        QP_free((char *)stream);
        if (close(fd) < 0) {
            qpstream->errno = errno;
            return QP_ERROR;
        }
        return QP_SUCCESS;
    }
```

decrypt.c

```
/* open_crypt_stream: open the specified non-tty 'filename' for
   reading.  The file is a simple crypted file with the first
   byte as the key.  It is crypted by logical exclusive-or
   operation of the key with every character in the file.
   Upon success, the opened stream is returned.
   Upon failure, QP_NULL_STREAM is returned and the error code
   is stored in the parameter 'error_num'.
*/
QP_stream *
open_decrypt(filename, error_num)
    char *filename;
    int  *error_num;
    {
        int             fd;
        Key_Type        key;
        DecryptStream   *stream;
        QP_stream       *option;

        if ((fd = open(filename, O_RDONLY)) < 0) {
            *error_num = errno;
            return QP_NULL_STREAM;
        }
        if (isatty(fd)) {        /* tty file is not accepted */
            (void) close(fd);
            *error_num = DECRYPT_TTY_FILE;
        }

        switch (read(fd, (char *) &key, sizeof(key)) ) {
        case sizeof(key):
                break;
        case 0:
                *error_num = DECRYPT_NO_KEY;
                (void) close(fd);
                return QP_NULL_STREAM;;
        default:
                *error_num = errno;
                (void) close(fd);
                return QP_NULL_STREAM;
        }
```

decrypt.c

```
    if (! (stream = (DecryptStream *)
                    QP_malloc(sizeof(*stream))) ) {
        (void) close(fd);
        *error_num = QP_errno;
        return QP_NULL_STREAM;
    }

    stream->fd = fd;
    stream->last_rdsize = 0;
    stream->left_size = 0;
    stream->key = key;

    option = &stream->qpinfo;
    QU_stream_param(filename, QP_READ, QP_DELIM_LF, option);

    option->max_reclen = Buffer_Size;
         /* Record the current byte offset in the file */
    option->magic.byteno = sizeof(key);
    option->read   = decrypt_read;
    option->seek   = decrypt_seek;
    option->close  = decrypt_close;

    QP_prepare_stream(&stream->qpinfo, stream->buffer);
    if (QP_register_stream(&stream->qpinfo) == QP_ERROR) {
        (void) stream->qpinfo.close(&stream->qpinfo);
        *error_num = QP_errno;
        return QP_NULL_STREAM;
    }
    return (QP_stream *) stream;
}
```

10.5.7.3 Creating A Stream Based On C Standard I/O Library

This example demonstrates creating a stream based on standard I/O library package. The stream is created as unbuffered for the Prolog I/O system (It is still buffered in the standard I/O package). By making the stream unbuffered, mixed I/O operations between Prolog code and C code using standard I/O library functions will work appropriately. In this case, line counts and character counts will be maintained for Prolog I/O predicates and QP functions only.

stdio.pl

```
foreign(open_stdio, c, open_stdio(+string, +string, -integer,
                                  [-address])).
foreign_file('stdio', [open_stdio]).

:- load_foreign_files(['stdio'],['-lc']).

open_stdio_file(FileName, ModeName, Stream) :-
        valid_open_mode(ModeName, Mode),
        open_stdio(FileName, Mode, ErrorNum, CStream),
        ( CStream =:= 0 ->
                raise_exception(existence_error(
                    open_stdio_file(FileName, ModeName, Stream),
                    1, file, FileName, errno(ErrorNum)))
        ; stream_code(Stream, CStream)
        ).

valid_open_mode(read,   r).
valid_open_mode(write,  w).
valid_open_mode(append, a).
```

stdio.c

```
#include <stdio.h>
#include <quintus/quintus.h>

/* Create a stream based on UNIX standard I/O library.
   This stream is created as an unbuffered stream so that
   mixed calls of Quintus Prolog I/O predicates (functions)
   and standard I/O on the stream will read/write the same
   sequence of bytes of the stream */

typedef struct
    {
        QP_stream       qpinfo;
        FILE            *fp;
        unsigned char   buffer[4];
    } StdioStream;

#define CoerceStdioStream(stream)       ((StdioStream *) stream)

extern int      errno;

static int
stdio_read(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        StdioStream     *stream = CoerceStdioStream(qpstream);
        register int    c;

        if ((c = getc(stream->fp)) < 0)
            return QP_EOF;
        stream->buffer[0] = (unsigned char) c;
        *bufptr = stream->buffer;
        *sizeptr = 1;
        /* '-1' because the magic field stores the beginning
                address of the returned buffer */
        qpstream->magic.byteno = ftell(stream->fp)-1;
        return (c == '\n') ? QP_FULL : QP_PART;
    }
```

stdio.c

```
static int
stdio_write(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        StdioStream *stream = CoerceStdioStream(qpstream);

        if (*sizeptr == 0) {
            *bufptr = stream->buffer;
            *sizeptr = 0;
            return QP_SUCCESS;
        }
        errno = 0;
        if (putc((char) stream->buffer[0], stream->fp) < 0) {
            qpstream->errno = (errno) ? errno : QP_E_CANT_WRITE;
            return QP_ERROR;
        }
        qpstream->magic.byteno = ftell(stream->fp);
        *bufptr = stream->buffer;
        *sizeptr = 0;            /* use 0 for unbuffered write */
        return QP_SUCCESS;
    }

static int
stdio_flush(qpstream, bufptr, sizeptr)
    QP_stream           *qpstream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        StdioStream *stream = CoerceStdioStream(qpstream);

        /* The stream is unbuffered so that there is no character
           in the buffer of stream->buffer */
        errno = 0;
        if (fflush(stream->fp) < 0) {
            qpstream->errno = (errno) ? errno : QP_E_CANT_FLUSH;
            return QP_ERROR;
        }
        qpstream->magic.byteno = ftell(stream->fp);
        *bufptr = stream->buffer;
        *sizeptr = 0;
        return QP_SUCCESS;
    }
```

stdio.c

```
static int
stdio_seek(qpstream, qpmagic, whence, bufptr, sizeptr)
    QP_stream           *qpstream;
    union QP_cookie     *qpmagic;
    int                 whence;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        StdioStream *stream = CoerceStdioStream(qpstream);
        int     rtn;

        errno = 0;
        /* fseek() should normally flush out the buffered input
           for stream->fp.  Use fflush() just to be safe */
        if (qpstream->mode != QP_READ)
            (void) fflush(stream->fp);
        switch (whence) {
        case QP_BEGINNING:
            rtn = fseek(stream->fp, qpmagic->byteno, 0);
            break;
        case QP_CURRENT:
            rtn = fseek(stream->fp,  qpmagic->byteno, 1);
            break;
        case QP_END:
            rtn = fseek(stream->fp, qpmagic->byteno, 2);
            break;
        default:
            qpstream->errno = QP_E_INVAL;
            return QP_ERROR;
        }
        if (rtn == -1) {
            qpstream->errno = (errno) ? errno : QP_E_CANT_SEEK;
            return QP_ERROR;
        }
        qpstream->magic.byteno = ftell(stream->fp);
        *bufptr  = stream->buffer;
        *sizeptr = (qpstream->mode == QP_READ) ? 0
                                   : qpstream->max_reclen;
        return QP_SUCCESS;
    }
```

stdio.c

```
static int
stdio_close(qpstream)
    QP_stream   *qpstream;
    {
        StdioStream *stream = CoerceStdioStream(qpstream);

        /* characters in fp buffer is flushed by fclose() */
        if (fclose(stream->fp) < 0) {
            qpstream->errno = errno;
            return QP_ERROR;
        }
        QP_free((char *) stream);
        return QP_SUCCESS;
    }
```

stdio.c

```
/* open_stdio() creates an instance of standard input/output
   based stream.  The function creates a file stream based
   on the 'filename' and 'modename' parameter.
   It returns the pointer to the created QP_stream structure
   upon success.  It returns QP_NULL_STREAM and sets
   error code in 'error_num' upon failure.
*/
QP_stream *
open_stdio(filename, modename, error_num)
    char        *filename, *modename;
    int         *error_num;
    {
        QP_stream    *option;
        FILE         *fp;
        StdioStream  *stream;
        int          mode, stdio_read(), stdio_write(),
                     stdio_flush(), stdio_seek(), stdio_close();

        switch (*modename) {
        case 'r':       mode = QP_READ;         break;
        case 'w':       mode = QP_WRITE;        break;
        case 'a':       mode = QP_APPEND;       break;
        default:        *error_num = QP_E_BAD_MODE;
                        return QP_NULL_STREAM;
        }
        if ((fp = fopen(filename, modename)) == NULL) {
            *error_num = errno;
            return QP_NULL_STREAM;
        }

                /* allocate space for the stream */
        stream = (StdioStream *) QP_malloc(sizeof(*stream));
                /* set values in the stream */
        stream->fp = fp;

        /* obtain default values in QP_stream structure */
        /*     and modified fields for this stream */
        option = &stream->qpinfo;
        QU_stream_param(filename, mode, QP_DELIM_LF, option);
        if (isatty(fileno(fp))) {
            option->format = QP_DELIM_TTY;
            option->seek_type = QP_SEEK_ERROR;
        } else {
            option->seek_type  = QP_SEEK_BYTE;
        }
        option->max_reclen = (mode == QP_READ) ? 1 : 0;
```

stdio.c

```
    if (mode != QP_READ) {
        option->write = stdio_write;
        option->flush = stdio_flush;
    } else {
        option->read = stdio_read;
        option->peof_act   = QP_PASTEOF_EOFCODE;
    }
    option->seek  = stdio_seek;
    option->close = stdio_close;
            /* sets correct value in magic field */
    if (option->mode != QP_APPEND)
        option->magic.byteno = 0;
    else
        option->magic.byteno = ftell(fp);

            /* set internal fields and register stream */
    QP_prepare_stream(&stream->qpinfo, stream->buffer);
    if (QP_register_stream(&stream->qpinfo) == QP_ERROR) {
        (void) stream->qpinfo.close(&stream->qpinfo);
        *error_num = QP_errno;
        return QP_NULL_STREAM;
    }

            /* register tty stream to its group */
    if (option->format == QP_DELIM_TTY)
        (void) QP_add_tty(&stream->qpinfo, filename);
    return (QP_stream *) stream;
}
```

10.5.8 Built-in C Functions And Macros For I/O

Several builtin functions and macros are defined to enable Prolog streams to be manipulated in foreign code. This section lists each of these functions. In this list, the character '`#`' is used to denote a C macro, which is defined in '`<quintus/quintus.h>`'. Full descriptions of each of these functions can be found in the individual reference pages.

Open a stream

`QP_fopen()`: open a text file or a binary file as a Prolog stream.

`QP_fdopen()`: create a text stream or a binary stream from a file descriptor.

Close a stream

`QP_close()`: close a Prolog stream

`QP_fclose()`: close a Prolog stream, same as `QP_close()`.

Input from a stream

`#QP_getchar()`: get a character from the Prolog current input stream.

`QP_getc()`: get a character from a Prolog input stream.

`QP_fgetc()`: get a character from a Prolog input stream.

`#QP_peekchar()`: look a character ahead from the Prolog current input stream.

`#QP_peekc()`: look a character ahead from a Prolog input stream.

`QP_fpeekc()`: look a character ahead from a Prolog input stream.

`QP_ungetc()`: unget the previous read character from a Prolog input input stream.

`#QP_skipline()`: skip the current input record of the Prolog current input stream.

`#QP_skipln()`: skip the current input record of a Prolog input stream.

`QP_fskipln()`: skip the current input record of a Prolog input stream.

`QP_fgets()`: get a string from a Prolog input stream.

`QP_fread()`: read several items of data from a Prolog input stream.

Output to a stream and flush output stream buffer:

`#QP_putchar()`: put a character on the Prolog current output stream.

`#QP_putc()`: put a character on a Prolog output stream.

`QP_fputc()`: put a character on a Prolog output stream.

`#QP_newline()`: terminates an output record for the Prolog current output stream.

`#QP_newln()`: terminates an output record for a Prolog output stream.

`QP_fnewln()`: terminates an output record for a Prolog output stream.

`QP_puts()`: put a character string on the Prolog current output stream.

`QP_fputs()`: put a character string on a Prolog output stream.

`QP_fwrite()`: write several items of data on a Prolog output stream.

`QP_tab()`: put the specified character the number of times specified on a Prolog output stream.

`QP_tabto()`: put the specified character up to the specified line position on a Prolog output stream.

`QP_printf()`: print formatted output on the Prolog current output stream.

`QP_fprintf()`: print formatted output on a Prolog output stream.

`QP_vfprintf()`: print formatted output of a varargs argument list on a Prolog output stream.

`QP_flush()`: flush output on a Prolog output stream

Get stream position and seek to a new position in a stream:

`QP_getpos()`: Get the current position for a Prolog stream.

`QP_setpos()`: position a Prolog stream back to a previous read/written position.

`QP_rewind()`: reposition a Prolog stream back to the beginning

`QP_seek()`: seek to a random position in a Prolog stream

Get counts in a stream:

`#QP_char_count()`: obtain the character count for a Prolog stream.

`#QP_line_count()`: obtain the line count for a Prolog stream.

`#QP_line_position()`: obtain the line position for a Prolog stream.

End of line (record) and End of file test:

`#QP_eoln()`: test end of record condition for an input stream.

`#QP_eof()`: test end of file condition for an input stream.

Set current stream:

`QP_setinput()`: set the Prolog current input stream.

`QP_setoutput()`: set the Prolog current output stream.

Error number related functions:

`QP_ferror()`: test error condition for a Prolog stream.

`QP_clearerr()`: clear the previous error on a Prolog stream.

`QP_errmsg()`: get the corresponding error message from a QP error number.

`QP_perror()`: print an error message based on a QP error number.

Finally, there are five global stream variables accessible in foreign code. These are streams, not file descriptors. It makes no sense to pass these to system calls that expect file descriptors. The values in these variables should not be changed by an assignment statement. These variables are:

`QP_stream *QP_stdin`
: user input stream, it is referred as `user_input` (or `user`) in Prolog.

`QP_stream *QP_stdout`
: user output stream, it is referred as `user_output` (or `user`) in Prolog.

`QP_stream *QP_stderr`
: user error stream, it is referred as `user_error` in Prolog.

`QP_stream *QP_curin`
: current input stream

`QP_stream *QP_curout`
: current output stream

10.5.9 Backward Compatibility I/O Issues

The Quintus Prolog input/output system is redesigned in release 3. C code written for Quintus Prolog application prior to Release 3.0 should also work on release 3 since the new design also maintains backward compatibility.

10.5.9.1 Default Stream

However, while the old Prolog I/O system is based on the C standard I/O library, the new Prolog I/O system is not. If an application performs mixed I/O operation in Prolog and foreign code on the three default Prolog streams, it might not work appropriately under the new I/O due to incompatibilities between the buffering mechanism in the C standard I/O stream and the Quintus Prolog stream.

Let's look at an example of a mixed output operation on a Prolog session under UNIX. Both the C standard output stream and the Prolog `user_output` stream write output to the same file descriptor, 1, which is a tty.

```
| ?- set_output(user_output), write(first), c_printf('FIRST'),
     write(second), c_printf('SECOND'), nl, c_nl.
```

The predicate `c_printf/1` and `c_nl/0` calls the following C functions:

```
void c_printf(atom)
     char *atom;
     {    printf("%s", atom); }
void c_nl() { putchar('\n'); }
```

The query yields the following output as expected prior to Quintus Prolog release 3.

```
firstFIRSTsecondSECOND
```

However it yields the following output on Quintus Prolog release 3 since each stream has its own buffer and no characters are actually written to the file descriptor 1 until new line operation is called.

```
firstsecond
FIRSTSECOND
```

This problem can be solved by supplying a different embedding `QU_initio()` function at the time of the installation of Quintus Prolog (or at the time of creating a statically linked Prolog system) to create the three default streams based on C standard I/O streams. How to create an unbuffered Prolog stream based on a C standard I/O stream has already been shown in the third example of creating customized Prolog streams (see Section 10.5.7.3 [fli-ios-uds-sst], page 472)

10.5.9.2 User_defined Streams

In Quintus Prolog releases prior to 3.0, `QP_make_stream()` was the function used to create a user-defined stream. Quintus Prolog 3.1 users should use the method described in Section 10.5.5 [fli-ios-cps], page 445. `QP_make_stream()` creates an unbuffered Prolog stream. This is not very efficient.

`QP_make_stream()` can still be used in release 3, but may not be supported in the future. Other old QP I/O functions that may not be available in future release are:

```
QP_sprintf()     QP_getc()       QP_sgetc()
QP_putc()        QP_sputc()      QP_sputs()
```

The naming convention of these functions does not match well with their counterparts on C standard I/O library. For instance, `QP_sprintf()` performs formatted output on a Prolog stream as the same operation for `fprintf(3)` on a C standard I/O stream. It is therefore renamed to be `QP_fprintf()` in release 3. For the same reason, `QP_sgetc()` is renamed as `QP_fgetc()`; `QP_sputc()` is renamed as `QP_fputc()`. `QP_getc()` and `QP_putc()` are now actually macros defined in '`<quintus/quintus.h>`'. However, all these functions are still available in release 3. If a user's foreign code calls either `QP_getc()` or `QP_putc()` without including '`<quintus/quintus.h>`', the old version of the function will be called. If '`<quintus/quintus.h>`' is included, the call is expanded to another function since both `QP_getc()` and `QP_putc()` are macros in '`<quintus/quintus.h>`'.

11 Inter-Process Communication

11.1 tcp: Network Communication Package

This package supplies the necessary primitives for network communication. This allows the user to take advantage of the computing power of a network of computers by allowing the construction of a set of cooperating processes running on different machines.

In general, the `tcp` package provides facilities to

- Send output to some connected process;
- Wait for input from some connected process, with or without a timeout; and
- Schedule wakeups.[1]

This package implements a *stream* socket with the TCP protocol providing the underlying communication support. A *stream* socket provides for bidirectional, reliable, sequenced and unduplicated flow of data without record boundaries. Two other types of sockets, the *datagram* socket and the *raw* socket, are not used here. TCP stands for the Internet Transmission Control Protocol.

This library package is intended for network communication, however, it does not require that each process be on a separate machine. It can be used to establish connections and communicate with processes on the same machine in just the same way that it would establish connections and communicate with processes on other machines.

Here is a simple example of the kind of thing you can do with this package. The example is the producer-filter-consumer problem, each running as a separate process. The producer produces successive terms and passes them on to the filter. The filter reads successive terms from the producer and then either passes them on to the consumer or discards them. The consumer reads and echos the terms passed to it by the filter. It is taken from the example program '`IPC/TCP/demo/ce.pl`'.

[1] Not available for C processes.

```
:-use_module(library(random)).
:-use_module(library(tcp)).

% on machine A we have the producer process:

producer:-
    tcp_address_from_file(filter, Address),
    tcp_connect(Address, Filter),
    repeat,
        random(X),
        tcp_send(Filter, X),
    fail.

% on machine B we have the filter process:

filter:-
    tcp_create_listener(AddressA, _),
    tcp_address_to_file(filter, AddressA),
    tcp_address_from_file(consumer, AddressC),
    tcp_connect(AddressC, Consumer),
    repeat,
        tcp_select(term(_,X)),
        0.2 =< X, X < 0.7,
        tcp_send(Consumer, X),
    fail.

% and on machine C we have the consumer process:

consumer:-
    tcp_create_listener(Address, _),
    tcp_address_to_file(consumer, Address),
    repeat,
        tcp_select(term(_,X)),
        format('The filtered number: ~d~n', [X]),
    fail.
```

11.1.1 The client/server relationship

Two processes have a client/server relationship when they cooperate with one another in such a way that one process responds to connection requests generated by the other.

It is important to remember that the client/server relationship has a very restricted meaning: it refers solely to how connections are established. For example, an X-Windows server controls the X-Windows client, whereas a NFS server is controlled by NFS clients. So saying processes have a client/server relationship says nothing about which process is the controlling process.

The distinction between clients and servers is based solely on how connections are established. Servers get connections by accepting connection requests, clients get connections by requesting a connection from a server.

server — a process that accepts connection requests. The server is said to listen for connections.

client — a process that requests a connection from a server.

A process can be a client to some processes, but a server to other processes. In the preceding example, the filter is a server to the producer and a client of the consumer.

A Prolog server is a Prolog process that calls the predicate `tcp_create_listener/2` and then accepts connection requests. A C server is a process that calls the C function `tcp_create_listener()` and then accepts connection requests.

A Prolog client is a Prolog process that calls the predicate `tcp_connect/2` to generate connection requests. In the same way a C client is a process that calls `tcp_connect()`.

Although designed principally with Prolog to Prolog communication in mind, this package can be used for any combination of C or Prolog servers or clients.

11.1.2 Using tcp

The tcp package is loaded by

```
:- use_module(library(tcp)).
```

This package relies on the flag `fileerrors` being set, which is the default. See `fileerrors/0` (Section 8.7.7.3 [ref-iou-sfh-sem], page 226) and `prolog_flag/3` (Section 8.10.1 [ref-lps-ove], page 245) for more about this flag. If `fileerrors` is not set, the behavior of this package is unpredictable. While we're on the subject of flags, it is also probably a good idea to set the flag `syntax_errors` to `error` (see Section 8.19.4.10 [ref-ere-err-syn], page 322). In summary, we recommend that the commands

```
prolog_flag(fileerrors,_,on),
prolog_flag(syntax_errors,_,error)
```

be issued before connections are established.

11.1.2.1 `tcp_trace(-OldValue, +On_or_Off)`

Causes a trace of the major events. All trace output is written to `user_error`.

`tcp_trace/2` raises a domain error if either of its arguments do not unify with either of the atoms `on` or `off`.

11.1.2.2 tcp_watch_user(*-Old, +On_or_Off*)

Not available under Windows.

This causes `tcp_select/[1,2]` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491) to return the atom `user_input` whenever input is available on `stdin`.

Unless `stdin` is unbuffered, the atom `user_input` will only be returned after a newline is received.

`tcp_watch_user/2` raises a domain error if either of its arguments do not unify with either of the atoms `on` or `off`.

11.1.2.3 tcp_reset

Resets the tcp software, killing all its sockets and dynamic predicates. All connected processes get an `end_of_file`.

11.1.3 Maintaining Connections

Described here are the various predicates for creating and destroying connections to other processes.

11.1.3.1 tcp_create_listener(*?Address, -PassiveSocket*)

`tcp_create_listener/2` creates a *passive socket* to listen for connections. If *Address* is unbound, `tcp_create_listener/2` establishes a listener on a *dynamic port* and binds *Address* to an *address term* of the form `address(Port,Host)`. Alternatively, `tcp_create_listener/2` will establish a listener on a *fixed port* if *Address* is bound to an address term with `Port` set to the specific port number. Note that only privileged (or '`root`') processes can use port numbers less than 1024.

`tcp_create_listener/2` returns immediately after creating the socket that is used for accepting connection requests and returns the socket identifier in *PassiveSocket*. Connection requests are accepted by the select predicates (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491).

11.1.3.2 tcp_destroy_listener(*+PassiveSocket*)

This predicate kills the passive socket created by `tcp_create_listener/2`.

11.1.3.3 `tcp_listener(?PassiveSocket)`

This predicate succeeds if the *PassiveSocket* argument is the socket identifier of a listener, or backtracks returning all passive socket identifiers if *PassiveSocket* is unbound.

11.1.3.4 `tcp_address_to_file(+ServerFile, +Address)`

This predicate writes the address term *Address* to the file *ServerFile*. This is useful when creating a listener on a dynamic port to enable clients to find out the server address by using the complementary predicate `tcp_address_from_file/2`.

11.1.3.5 `tcp_address_from_file(+ServerFile, -Address)`

A client uses this predicate to obtain the address of the server from the file *ServerFile*, written by the server using `tcp_address_to_file/2`. If the client and server are on different machines then *ServerFile* must be located on a network transparent filesystem (e.g. NFS) to be accessible to the client.

11.1.3.6 `tcp_address_from_shell(+Host, +ServerFile, -Address)`

Not available under Windows.

This is identical to `tcp_address_from_file/2` except that, instead of relying on a network transparent file system to be able to read *ServerFile*, it executes a remote shell command on *Host* to read the contents of the file. This is useful for applications that cannot rely on the presence of a network transparent file system.

11.1.3.7 `tcp_address_from_shell(+Host, +UserId, +ServerFile, -Address)`

This adds a *UserId* parameter, so that the machine that has the handle file need not have an account for every user that wishes to access it.

The *UserId* is an atom representing the login name of some account on the target machine *Host*.

11.1.3.8 `tcp_connect(+Address, -Socket)`

This is used by a Prolog client to connect to a server. *Socket* is unified with the active socket identifier created when the connection is established. The *Address* parameter is an address term, such as that returned by `tcp_address_from_file/2`.

Once a connection has been made to *Socket* later calls to `tcp_connect/2` succeed immediately, without attempting to re-establish a connection to *Socket*.

11.1.3.9 `tcp_connected(?Socket)`

This predicate succeeds if the *Socket* argument is the socket identifier of an active connection, or backtracks returning all currently active connections if *Socket* is unbound.

For example, this predicate can be used to shutdown all connections:

```
close_all_connections :-
        tcp_connected(X),
        tcp_shutdown(X),
        fail
    ;   true.
```

11.1.3.10 `tcp_connected(?Socket,?PassiveSocket)`

A server process can use this predicate to identify which active connections are associated with which listeners. This can be useful if a server process establishes multiple listeners, listening on different ports simultaneously.

11.1.3.11 `tcp_shutdown(+Socket)`

This kills the connection to *Socket*. *Socket* gets, through `tcp_select/[1,2]`, an `end_of_file`. If there is no connection to *Socket*, it silently fails.

Note that when `tcp_select/[1,2]` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491) returns `end_of_file(Socket)`, `tcp_shutdown(Socket)` has already been called.

`tcp_shutdown/1` raises an instantiation error when *Socket* is uninstantiated.

11.1.3.12 Short lived connections

The operating system limits the number of simultaneous connections open by a process at one time. The limit various with different operating systems but is typically 64 connections.

If the number is too small for your application, consider making connections persist only long enough for a send or receive. This way at most one connection would be alive at any time.

Here is an example of how you might implement sending and receiving in terms of short lived connections.

```
send(To,Term):-    % +To, +Term
    my_address(MyAddress),
    tcp_connect(To, Socket),
    tcp_send(Socket, MyAddress-Term),
    tcp_shutdown(Socket).

receive(From,Term):-  % -From, -Term
    repeat,
    tcp_select(term(Socket, From-Term)),
    !,
    tcp_shutdown(Socket).
```

In the above example, it is assumed that all processes are servers, and have asserted the address obtained from a call to `tcp_create_listener/2` into `my_address/1`.

The performance penalty of short lived connections is the time for making and breaking connections, which is actually quite fast. For comparison, the time it takes to create and then destroy a connection to a server is hundreds of times slower than sending a character to the server, but is comparable to the time it takes to send a large term to the server.

11.1.4 Sending and Receiving Terms

11.1.4.1 `tcp_select(-`*Term*`)`

The purpose of `tcp_select/1` is to process connection requests, return terms related to timing, and return in a round-robin fashion terms read from connected processes. *Term* is one of:

`connected(`*Socket*`)`
: is returned when the server has accepted a connection request from *Socket*. *Socket* is small integer. It is a socket file descriptor (a small integer).

`wakeup(`*Term*`)`
: is returned when the timer alarm specified in `tcp_schedule_wakeup/2` was delivered (see Section 11.1.5.3 [ipc-tcp-tim-schedule_wakeup2], page 494).

`user_input`
: is returned whenever there is input available on `stdin` and `tcp_watch_user(_,on)` has been called. See `tcp_watch_user/2`, Section 11.1.2.2 [ipc-tcp-utc-watch_user2], page 488.

`term(`*Socket*`,`*Term*`)`
: is returned when some process whose socket file descriptor is *Socket* has called `tcp_send/2` (see Section 11.1.4.3 [ipc-tcp-trm-send2], page 493). This is the result of a read from the socket.

`end_of_file(`*Socket*`)`
: is returned when the connection is lost to the process whose socket file descriptor is *Socket*. This is the result a read from the socket. The connection is shut down.

Windows caveats:

- `tcp_select/[1,2]` is not interruptible by `^C`. For this reason, calling `tcp_select/2` with infinite timeout is probably a bad idea. If called with infinite timeout and if there are no open sockets, then `tcp_select/2` will return immediately, indicating a timeout.
-

11.1.4.2 `tcp_select(+`*Timeout*`, -`*Term*`)`

This is the same as `tcp_select/1` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491), except `tcp_select/2` will return the atom `timeout` if the timeout interval expires before input is available or before a timer alarm is delivered (from `tcp_schedule_wakeup/2`, see Section 11.1.5.3 [ipc-tcp-tim-schedule_wakeup2], page 494).

Timeout is a floating point number indicating seconds.

The `tcp_select` predicates must deal with three events:

1. Timer alarms — are dealt with first. These are scheduled with `tcp_schedule_wakeup/2` (see Section 11.1.5.3 [ipc-tcp-tim-schedule_wakeup2], page 494).
2. ready input — are dealt with when (1) doesn't apply, that is, if no timer alarm is delivered; `tcp_select/2` will sleep until there is input available from some connected process.
3. timeout — when neither (1) or (2) apply, and the timeout interval specified has expired, then the atom `timeout` is returned.

A poll is effected by specifying 0 for *Timeout*.

11.1.4.3 `tcp_send(+Socket, +Term)`

This sends *Term* to the process whose socket identifier is *Socket*. *Socket* gets the term *term* (*From*,*Term*) from `tcp_select/[1,2]` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491).

Note that `tcp_send/2` can only be used to send terms to a Prolog server as *Term* is sent in an encoded form that is efficiently decoded by `tcp_select/[1,2]` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491).

Here is an example of how one might use `tcp_send/2` to implement a remote procedure call to some process whose file descriptor is *P*. It is assumed that the connections between the two processes have been established elsewhere.

```
:-use_module(library(basics), [member/2]).
:-use_module(library(freevars), [free_variables/4]).

% machine a has p_call/2

p_call(P, Goal):-
    free_variables(Goal, [], [], FreeVars),
    tcp_send(P, satisfy(FreeVars, Goal)),
    tcp_select(term(P, Bag)),
    member(FreeVars, Bag).

% machine b has slave/0

slave:-
    T = term(P, satisfy(FreeVars, Goal)),
    repeat,
        tcp_select(T),
        findall(FreeVars, Goal, Bag),
        tcp_send(P, Bag),
    fail.
```

The use of `library(freevars)` is to limit the amount of data being sent by the slave to just those variables that may be instantiated by calling `Goal`.

For many applications this is all that is required. It has the advantage of limiting both the frequency of messages sent (`findall/3`), and the size of the messages (`free_variables/4`). A better implementation of remote procedure call would allow the caller to respond to solutions from several different machines as soon as the solutions are generated, without waiting for the solutions to be assembled into a list. This is attempted in the example program '`IPC/TCP/demo/sibling.pl`' (see Section 11.1.9 [ipc-tcp-exa], page 505).

If you try to send to a broken socket, the "Broken pipe" exception is raised:

```
existence_error(_,_,_,_,errno(32))
```

11.1.5 Time Predicates

The tcp package supplies various operations on time. These predicates can be used independently of the rest of the package. The tcp package provides a timer scheduling facility and time conversion facilities. Time stamping and portrayal is supplied by another library package, namely `library(date)` (see Section 12.11.2 [lib-mis-date], page 637).

All the time predicates use an absolute time format called the `timeval/2` representation. It is a term of the form `timeval(`*`Seconds`*`,`*`MicroSeconds`*`)`, where *Seconds* and *MicroSeconds* are integers representing the absolute system time in seconds and microseconds, respectively.

11.1.5.1 `tcp_now(-`*`Timeval`*`)`

`tcp_now/1` returns the current absolute system time in a `timeval/2` structure.

11.1.5.2 `tcp_time_plus(?`*`Timeval1`*`, ?`*`DeltaTime`*`, ?`*`Timeval2`*`)`

This predicate is true when the interval between *Timeval1* and *Timeval2* is *DeltaTime*. *DeltaTime* is a floating point number representing seconds. Both *Timeval1* and *Timeval2* are `timeval/2` structures.

At least two of the arguments to `tcp_time_plus/3` must be ground.

11.1.5.3 `tcp_schedule_wakeup(+`*`Timeval`*`, +`*`Term`*`)`

This schedules a wakeup from `tcp_select/1` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491). For example,

```
tcp_now(Now),
tcp_time_plus(Now, 0.1, Timeval),
tcp_schedule_wakeup(Timeval, foo)
```

forces `tcp_select/[1,2]` to return the term `wakeup(foo)` a tenth of a second after the call to `tcp_now/1`. Any number of wakeups may be pending.

The wakeup mechanism is implemented in terms of timer alarms. If a timer alarm was delivered when the process was not waiting at select, the next call to select will indicate that an alarm was delivered.

The wakeup mechanism can be used independently of the rest of the tcp package. You need not set up communications with other processes to use the wakeup mechanism.

11.1.5.4 tcp_scheduled_wakeup(*?Timeval*, *?Term*)

`tcp_schedule_wakeup/2` backtracks through the list of scheduled wakeups.

11.1.5.5 Canceling Wakeups

Two predicates are provided:

`tcp_cancel_wakeup(+`*Timeval*`, +`*Term*`)`

`tcp_cancel_wakeups`

`tcp_cancel_wakeup/2` cancels the wakeup that was previously specified using `tcp_schedule_wakeup/2`. `tcp_cancel_wakeups/0` cancels all the pending wakeups.

11.1.5.6 tcp_daily(+*Hour*, +*Minute*, +*Seconds*, -*Timeval*)

This predicate is useful for scheduling daily events. The given time will be translated into the equivalent absolute time, and that time will definitely be within the next 24 hours. For example,

```
tcp_daily(13, 0, 0, Timeval)
```

unifies *Timeval* with the `timeval/2` representation for 1 pm. If the goal was submitted at noon, *Timeval* will represent an hour later. If the goal was submitted at 2 pm, *Timeval* will represent 1 pm the following day.

Hour, *Minute*, and *Second* must all be integers.

The following example wakes up every 24 hours

```
        ...
        tcp_daily(H, M, S, Timeval)
        tcp_schedule_wakeup(Timeval, time(H,M,S)),
        repeat,
            tcp_select(X),
            dispatch(X),
        fail.

    dispatch(wakeup(T)):-
        T = time(H,M,S),
        tcp_daily(H, M, S, Timeval),
        tcp_schedule_wakeup(Timeval, T),
        ...
```

11.1.5.7 `tcp_date_timeval(?Date, ?Timeval)`

This predicate is used to convert between the time format supplied by `library(date)` and the `timeval/2` representation. At least one of the arguments must be ground. The parameter *Date* is of the form `date(Year,Month,Day,Hour,Minute,Second)`.

`library(date)` has facilities for portraying time. `tcp_date_timeval/2` can be used with `library(date)` for portrayal:

```
:-use_module(library(addportray), [add_portray/1]).
:-use_module(library(date), [time_stamp/3]).
:-use_module(library(tcp), [tcp_date_timeval/2]).

:-initialization add_portray(portray_timeval).

portray_timeval(timeval(Seconds, MicroSeconds)):-
    tcp_date_timeval(Date,timeval(Seconds, MicroSeconds)),
    time_stamp(Date,'%y %02n %M %02d %W %02c:%02i:', Stamp),
    write(Stamp),
    Date=date(_, _, _, _, _, S),
    X is (S * 1.0e6 + MicroSeconds) / 1.0e6,
    (X < 10 -> write(0) ; true),
    format('~2f', X).
```

Which would result in the following:

```
| ?- tcp_now(X).

X = 1989 03 March 01 Wednesday 17:09:58.12
```

11.1.6 Using Prolog streams

Sometimes the format of data being exchanged between processes is not known in advance and it is not possible to assume that the data sent are valid Prolog terms. This package provides Prolog streams for each connection that can be read from or written to using the stream input/output predicates. For more information about Prolog streams, see Section 8.7.2 [ref-iou-str], page 215.

Although these streams can be written to or read from using the standard input/output predicates supplied by Prolog, they must be closed using `tcp_shutdown/1` instead of `close/1`, otherwise the database internal to `library(tcp)` will become inconsistent.

There is a subtle point about `end_of_file`: in a correct implementation of TCP it is possible to receive zero-length packets, so that a socket *should* be able to signal end-of-file repeatedly just like a terminal. So does `end_of_file(Socket)` mean "connection lost to *Socket*", or does it mean "zero-length packet received from *Socket*"? Using `tcp_select/1`

or `tcp_select/2`, `end_of_file` causes the connection to be shutdown. Reading the streams yourself lets you decide what `end_of_file` means on your sockets.

11.1.6.1 `tcp_select_from(-Term)`

This is similar to `tcp_select/1` (see Section 11.1.4.1 [ipc-tcp-trm-select1], page 491), but instead of reading the socket that has data available, it returns the term `from(Socket)`, where *Socket* is socket to some other process. The other terms it may return are

- `connected(Socket)`
- `wakeup(T)`
- `user_input`

as described in the section on `tcp_select/1`.

It is up to the caller to read from the stream associated with the socket file descriptor, *Socket*, by using `tcp_input_stream/2` and then reading from that stream using the standard stream input predicates, as in

```
...,
tcp_select_from(from(Socket)),
tcp_input_stream(Socket, S),
get0(S,X),
...
```

11.1.6.2 `tcp_select_from(+Timeout, -Term)`

The behavior of `tcp_select_from/2` is the same as `tcp_select_from/1`, except `tcp_select_from/2` will return the atom `timeout` if the timeout interval expires before input is available or a timer alarm is delivered.

Timeout is a floating point number indicating seconds

11.1.6.3 `tcp_input_stream(?Socket, -Stream)`

Returns the input stream *Stream* from the socket file descriptor *Socket*. You can read from the stream using the regular stream input predicates.

If you attempt to read from a broken socket, the "Connection reset by peer" exception is raised:

```
existence_error(_,_,_,_,errno(54))
```

When using `tcp_select/[1,2]`, this exception is caught and interpreted as an `end_of_file`, and the connection is shut down.

11.1.6.4 `tcp_output_stream(?Socket, -Stream)`

Returns the output stream *Stream* from the socket file descriptor term *Socket*. You can write to the stream using the regular stream output predicates.

Output on socket streams are buffered. If you want to send characters one at a time, you must follow each write with a flush, as in

```
tcp_output_stream(Socket, Stream),
put(Stream, Character),
flush_output(Stream)
```

If you try to write to a broken socket, the "Broken pipe" exception is raised:

```
existence_error(_,_,_,_,errno(32))
```

11.1.7 The Callback Interface

The tcp package supplies a callback interface, which is a way of arranging for a predicate to be called whenever some tcp event occurs. This is especially useful for applications that can not wait at one of the `tcp_select` predicates. A callback is a predicate that is called when some condition is met. Callbacks are called when your process is in some wait state. It uses the callback facility described in relation to the Quintus supplied C functions `QP_select()` and `QP_add_input()`.

For example, applications that use the callbacks defined in the Quintus X Toolkit interface (`xif`) use callbacks extensively. Before the callback interface to `library(tcp)` was provided, there was no way for a graphics program to both service its callbacks and service `library(tcp)` without polling.

Although this interface may seem primitive, it has the advantage of being able to be used by separate subcomponents of a larger system since the requirement that the application wait at `tcp_select/[1,2]` is lifted.

11.1.7.1 `tcp_create_input_callback(+Socket, +Goal)`

`tcp_create_input_callback/2` arranges that *Goal* is called whenever there is data available on *Socket*.

To arrange for the client to make a callback whenever there is input available on a given socket:

```
        ...
        tcp_connect(Address, Socket),
        tcp_create_input_callback(Socket, input_on(Socket)).
        ...

    input_on(Socket) :-
        tcp_input_stream(Socket, I),
        read(I, Term),
        ...
```

11.1.7.2 tcp_destroy_input_callback(+*Socket*)

`tcp_destroy_input_callback/1` destroys the callback associated with *Socket*.

11.1.7.3 tcp_input_callback(**Socket*, **Goal*)

`tcp_input_callback/2` backtracks through the list of *Socket-Goal* pairs maintained by the callback interface to the tcp package.

11.1.7.4 tcp_create_timer_callback(+*Timeval*, +*Goal*, -*TimerId*)

`tcp_create_timer_callback/3` arranges things so that *Goal* will be called when the absolute time value *Timeval* is called.

Note that this goal will only be called when your program is in a wait state. Therefore all the tcp package can do is guarantee that if your program goes to sleep at `QP_select()` and the absolute time specified is reached or passed, then the goal is called. It does not work using asynchronous operating system timers.

The output argument *TimerId* can be used for destroying the timer callback just created.

To arrange for a goal to be called at some absolute time:

```
        ...,
        tcp_create_timer_callback(Timeval, Goal, TimerId),
        ...
```

11.1.7.5 tcp_destroy_timer_callback(+*TimerId*)

`tcp_destroy_timer_callback/1` uses the *TimerId* obtained by calling `tcp_create_timer_callback/3` for removing the timer callback.

11.1.7.6 tcp_timer_callback(*Timerid*, *Goal*)

`tcp_timer_callback/2` backtracks through the list of *Timerid-Goal* pairs maintained by the callback interface to the tcp package.

11.1.7.7 tcp_accept(+*PassiveSocket*, -*Socket*)

`tcp_accept/2` is used to accept a connection request on *PassiveSocket* and binding a Prolog stream to it. The input and output streams created by calling this predicate can be obtained by using the *Socket* output argument when calling `tcp_input_stream/2` and `tcp_output_stream/2`, respectively.

To arrange for a server to make a callback whenever there is a connection request:

```
        ...
        tcp_create_listener(Port, Host, Passive),
        tcp_create_input_callback(Passive, accept(Passive)),
        ...

    accept(Passive) :-
        tcp_accept(Passive, Socket),
        ...
```

Probably you will want to make the newly created socket a callback as well, so then the clause for `accept/1` in the preceding example would be:

```
    accept(Passive) :-
        tcp_accept(Passive, Socket),
        tcp_create_input_callback(Socket, input_on(Socket)).
```

Where `input_on/1` is as defined in the example for `tcp_create_input_callback/2`.

11.1.8 The C functions

Communication between processes is effected through sockets. Sockets are referred to by small 32 bit integers, called file descriptors below (even though they are proper file descriptors only under UNIX).

The program using this package should be linked with the object file '`tcp_c.o`' (substitute the actual object file extension for '`.o`'). The sources to create this file, the C source files '`tcp.c`' and '`tcp.h`', are provided in the tcp library directory.[2] Furthermore, the source file '`tcp.h`' from '`IPC/TCP`' should be included in the C sources using the package.

[2] The object file is also created in the system directories of tcp at installation time. In the development system, the call `absolute_file_name(library(system('tcp_c.o')),AbsPath)` returns the full path to the object file '`tcp_c.o`' for the particular architecture being used.

Errors from using the C tcp package are written to `stderr` using the system function `perror(3)`, so whenever a tcp C function indicates an error, the error message has already been printed and errno has already been set.

Although sockets may be used like any other file descriptor, they must be killed using `tcp_shutdown()`, otherwise the behavior of `tcp_select()` will become unpredictable.

For a C client calling a Prolog server, see example '`cs.c`' (Section 11.1.9 [ipc-tcp-exa], page 505).

For a Prolog client calling a C server, see example '`c_server.pl, c_server.c`' (Section 11.1.9 [ipc-tcp-exa], page 505).

11.1.8.1 `tcp_create_listener()`

A C process can listen for connection requests generated by client processes by calling the C function `tcp_create_listener()`, as shown by the following program fragment:

```
int Port,Service;
char *Host;

if (tcp_create_listener(0, &Port,&Host,&Service) != 0)
        ... an error occurred.
```

The C function `tcp_create_listener()` is used by a C server to create a listener (*Service*). A passive socket is used to accept connection requests. It returns the port and hostname in *Port* and *Host* arguments respectively. The first argument specifies a fixed port number to listen on. If it is zero, as in this example, then a dynamic port number is allocated. The *Service* is used with later calls to the C function `tcp_accept()` to accept a connection request. A connection request is detected using the C function `tcp_select()`.

The example program '`c_server.c`' illustrates the use of this function (see Section 11.1.9 [ipc-tcp-exa], page 505).

11.1.8.2 `tcp_address_to_file()`

```
#include "tcp.h"

int tcp_address_to_file(ServerFile, Port, Host)
    char *ServerFile;
    int Port;
    char *Host;
```

This function writes the *Port* and *Host* to the file *ServerFile* to enable a client to find the server's address. This is useful when establishing a listener on a dynamic port.

11.1.8.3 `tcp_address_from_file()`

```
#include "tcp.h"

int tcp_address_from_file(ServerFile, Port, Host)
    char *ServerFile;
    int *Port;
    char **Host;
```

This reads the server's address from the file *ServerFile*, which was written by the server calling `tcp_address_to_file/2`.

Note that **Host* is a pointer to static storage that will be overwritten at the next call to `tcp_address_from_file()`.

`tcp_address_from_file()` returns zero upon successful completion.

11.1.8.4 `tcp_address_from_shell()`

```
#include "tcp.h"

int tcp_address_from_shell(Host1, UserId, ServerFile, Port, Host)
    char *Host1;
    char *UserId;
    char *ServerFile;
    int *Port;
    char **Host;
```

This is identical to `tcp_address_from_file()`, except that, instead of relying on a network transparent file system to be able read *ServerFile*, it executes a remote shell command to *Host1* to read the contents of the file. This is useful for applications that cannot rely on the presence of a network transparent file system.

**Host* is a pointer to static storage that will be overwritten at the next call to `tcp_address_from_shell()`.

The *UserId* argument may be specified as `""`, meaning login as myself. It is provided so that the machine that has the handle file need not have an account for every user that wishes to access it.

Note that *Host1* need not be the same string as **Host*, but typically is.

`tcp_address_from_shell()` returns zero upon successful completion.

11.1.8.5 tcp_connect()

The C function `tcp_connect()` is used by C clients to connect to some server running on machine *host* at some *port*. It returns the newly created socket. The following program fragment demonstrates the use of the C function `tcp_connect()` along with its companion C function `tcp_address_from_file()`:

```
#include "tcp.h"

int c,port;
char *host;

if(tcp_address_from_file(serverfile, &port, &host) != 0)
    ... an error occurred.
c = tcp_connect(host, port);
if (c == -1)  ... an error occurred.
```

A fuller example of the above can be found in the demonstration program '`cs.c`'.

11.1.8.6 tcp_accept()

```
#include "tcp.h"

int fd,Service;

fd = tcp_accept(Service);
if (fd == -1)  ... an error occurred.
```

The C function `tcp_accept()` is used to accept a connection request. It returns the file descriptor for the newly created socket (*fd*).

A connection request is recognized when the file descriptor returned by the C function `tcp_select()` is the file descriptor for the passive socket returned by `tcp_create_listener()`. In other words, `tcp_select()` indicates that the passive socket created by `tcp_create_listener()` has input available. Since it is impossible to read from a passive socket, this means that a connection request is pending, and it is time to call `tcp_accept()` to accept the connection request.

11.1.8.7 tcp_select()

```
#include "tcp.h"

int FD,Block;
double Timeout;

...

switch (tcp_select(Block, Timeout, &FD))
{
case tcp_ERROR:
    ... an error occurred.
case tcp_TIMEOUT:
    ... handle a timeout
case tcp_SUCCESS:
    ... input is ready on FD
}
```

`tcp_select()` is used to determine which file descriptor is ready for input. It returns 3 status values:

`tcp_ERROR`
: the error message is printed using the system function `perror(3)`.

`tcp_TIMEOUT`
: the time interval specified in Timeout (seconds) expired.

`tcp_SUCCESS`
: the FileDescriptor returned is ready to be read.

With `Block == tcp_BLOCK`, `tcp_select()` will ignore the Timeout parameter, and simply block until some data is available.

With `Block == tcp_POLL`, `tcp_select()` will sleep for Timeout seconds, or until data is available, whichever comes first.

11.1.8.8 tcp_shutdown()

```
#include "tcp.h"

int FD;

if (tcp_shutdown(FD) == -1)    ... an error occurred.
```

The C function `tcp_shutdown()` is used to kill a passive or active socket. It is important that this is used instead of the system function `close(2)`, since it affects the behavior of the C function `tcp_select()`.

11.1.9 Examples

Five examples have been provided in '`IPC/TCP/demo`'. Each example has detailed instructions on its use in its source file. A Makefile is provided in the '`demo`' directory.

Calling `absolute_file_name/2` is a convenient way of finding the path to a demonstration program. For example, to find the sibling demonstration, try issuing the command

```
| ?- absolute_file_name(demo('sibling.pl'),X).
```

Here is a list of the example tcp programs.

'`client.pl, server.pl`'
: The client/server example. This illustrates how connections are established and a very simple method of remote procedure call.

'`sibling.pl`'
: This file demonstrates the tcp software for three connected processes. Once connected, all three send and receive goals to each other as peers. It can detect reception of keyboard input, since it uses the `tcp_watch_user/2` predicate.

'`ce.pl`'
: An example of the producer, filter, consumer problem. The producer process sends random numbers between 0 and 1 to the filter process, which in turn sends copies of the numbers it received from the producer ranging between 0.2 and 0.7 to the consumer.

'`cs.c`'
: This is a simple way to call the Prolog server defined in '`server.pl`' from the c-shell. If the *ServerFile* (see `tcp_address_from_file/2`, Section 11.1.3.5 [ipc-tcp-mco-address_from_file2], page 489) for some server is `x`, then

  ```
  % cs x "write('hi there'),nl"
  ```

  causes the server to write the string `"hi there\n"` to its socket. The C program '`cs.c`' copies the socket output from the server to `stdout`.

'`c_server.c, c_server.pl`'
: This illustrates a Prolog client calling a C server.

11.2 IPC/RPC: Remote Predicate Calling

11.2.1 Overview

We recommend that if you are just starting out, do *not* use this package. The TCP package is much faster and more powerful. IPC/RPC is not available under Windows.

In releases prior to Quintus Prolog Release 2.5, this package was simply known as IPC. It is now called IPC/RPC to distinguish it from another interprocess communication package,

which is called IPC/TCP. That package is more general than this one since its facilities can be used to implement the functionality of this package.

This package has some interesting facilities for calling a Prolog servant from C. Before Release 3.0, this was the only way to call Prolog from C. Now Prolog can be fully embedded in a C application.

This Interprocess Communication (IPC) package provides tools for allowing programs written in Prolog or C to remotely call predicates in a Prolog program that is running as a separate process, possibly on a different machine. The communication between the processes is implemented using sockets or pipes to send goals to the remote process and to retrieve answers back.

We refer to the Prolog process that is being invoked by some other program as a *servant*, because it provides a goal evaluation service at the request of another program: it is given a goal to invoke, invokes it, and then returns the answers to the caller. It is called a servant, as opposed to a server, because it serves a single master. We refer to the program that is calling the servant as the *master*. The interface described here permits a program (the master) to call *one* servant and use it to evaluate many subgoals. The characteristics of the interface vary somewhat with the programming language of the master. If the master is itself a Prolog program, then the interface can be much more flexible than when the calling program is written in a procedural language, such as C. We divide the description of the interface into two parts: (1) when the master program is written in Prolog, and (2) when the master program is written in C. Although only C calling Prolog is documented, other languages can also call Prolog if they adhere to the specified protocol.

> **Please note:** On System V versions of UNIX or VMS, the master and server processes are currently required to run on the same machine, and the communication is via pipes rather than sockets. On UNIX systems based on BSD, such as SunOS 4.x, the user can choose to use either pipes or sockets, provided that the two processes are on the same machine. Sockets must be used when the processes are on separate machines. When sockets are used, there needs to be an entry in the '`/etc/hosts`' file(s) for each machine that is used.

11.2.2 Prolog Process Calling Prolog Process

When the master is a Prolog program, a very flexible interface is supported because the nondeterminacy of the two Prolog programs can be combined. Also, general Prolog data structures can be passed between the programs easily, since both programs support the same data types. Using this interface, a complex Prolog system can achieve significant parallel evaluation, by using a servant on another processor and communicating over a network. The routines described below allow a master to have only a single servant process. (They could be extended without much difficulty to support multiple servants and servants being masters of other servants, if that proves important.)

There are two sides to any interface: here we have the calling Prolog program (the master), and the called Prolog program (the servant). Each must perform certain functions that allow them to cooperate.

For a master to use a servant, the master must first create it. This is done by starting a Prolog process that will be the servant. The system creates that process by running a saved state previously created by the programmer. After the servant has been created and is running, the master may send it goals to evaluate using `call_servant/1` and `bag_of_all_servant/3`. All goals sent to the servant are evaluated in the database of the servant, which is disjoint from the database of the master. This means that all programs that the servant will execute must either already be in the saved state that was initially loaded, or a goal must be sent to the servant telling it to compile (or consult) the appropriate files. One could also use remote call to have the servant evaluate an `assert/1`.

For an example of using a Prolog servant from Prolog, see the '`IPC/RPC/demo`' library directory (`qplib('IPC/RPC/demo')`).

All the following predicates are defined in the module `qpcallqp`. To be able to use them, the master program must first load them by entering the directive:

```
:- use_module(library(qpcallqp)).
```

11.2.2.1 save_servant(+*SavedState*)

To be able to call a servant, you must first create (using `save_servant/1`) a saved state that is to be run as the servant. Run Prolog on the machine on which the servant is to be run, and load (that is, compile, consult, or assert) everything that the servant will need. Then call `save_servant(`*SavedState*`)`, where *SavedState* is the name of the file in which to save the state. (This saved state should not normally be started directly from a terminal by a user; when started it will automatically try to open and read a socket.)

11.2.2.2 create_servant(+*Machine*, +*SavedState*, +*OutFile*)

Before a master can use a servant, the servant must first be started up and connections to it must be made. This is done by a call to `create_servant/[2,3]`.

Machine is the name of the machine on which to run the servant. If *Machine* is omitted, or set to the null atom `''`, the servant is run on the same machine, and communication is via pipes. If *Machine* is the atom `local`, the servant is run on the same machine but communication is via sockets. If *Machine* names another machine, communication will be via sockets. You need to be able to use `rsh` on that machine.

SavedState is the name of the file that contains the Prolog saved state on that machine. It must have been previously created with `save_servant/1`.

OutFile is the name of the file to which output from the servant will be written. This file is on the local machine and it will be created if it does not already exist. *This file should be examined if there are problems with the communication to the servant.* Tracing information (if any, see Section 11.2.4 [ipc-rpc-tra], page 518) will also be written to this file.

If *OutFile* is the atom `user`, then all output will be sent to the standard output stream of the master. If it is the null atom `''`, the servant's standard output is discarded and its standard error is directed to the master's standard error.

11.2.2.3 `call_servant(+Goal)`

Once a servant has been created (by `create_servant/2`), goals can be sent to it for evaluation, by using `call_servant(Goal)`. This sends the goal *Goal* to the servant, which evaluates it (with respect to its own database) and sends all the answers back. The answers are returned as solutions of `call_servant/1`. The answers bind the variables in *Goal*. Answers after the first are obtained by backtracking into `call_servant/1`. Note that the servant computes and sends *all* answers back to the master, even if the caller uses a cut to throw away all but the first.

11.2.2.4 `bag_of_all_servant(?Template, +Goal, -Bag)`

If the servant is running on a different physical processor than the master, then it is desirable to be able to achieve some degree of parallelism, to have both machines doing useful work at the same time. This is not the case with `call_servant/1`, since the master Prolog process is waiting the entire time that the servant process is computing. The predicate `bag_of_all_servant/3` is provided to allow a sophisticated user to write some truly parallel applications. (See the demo program '`queensdemo`' for an example of using parallelism in a search problem.)

Semantically `bag_of_all_servant/3` is very similar to `bagof/3`. The reader should be familiar with the operation of `bagof/3` before reading further. The differences are:

1. `bag_of_all_servant/3` requires that there be no free variables in *Goal* that do not appear in *Template*. If there are, `bag_of_all_servant/3` will report an error. You may use the existential operator (`^`) as in `bagof/3`.
2. `bag_of_all_servant/3` succeeds with *Bag* bound to `[]` if *Goal* has no answers at all. This means that `bag_of_all_servant/3` always succeeds and returns in *Bag* exactly one answer: the list of instances of *Template*, one instance for each success of *Goal*.

The exact operation of `bag_of_all_servant/3` depends on the form of *Goal*. If *Goal* is a conjunction of the form `(Goal1, Goal2)` or a disjunction of the form `(Goal1; Goal2)`, then the first subgoal (*Goal1*) will be executed by the servant, and the second subgoal (*Goal2*) will be executed by the current process. The system will try to overlap local and remote evaluation as much as possible. If *Goal* is neither a conjunction nor a disjunction, then the entire goal will be sent to the servant to be executed.

There are several restrictions on how `bag_of_all_servant/3` can be used.

- *Goal2* may not contain any cuts.
- *Goal2* must not require the services of the servant to be evaluated. That is, it cannot call any predicate that uses `call_servant/1` or `bag_of_all_servant/3`.

11.2.2.5 `set_of_all_servant(?Template, +Goal, -Set)`

This is just like `bag_of_all_servant` except that duplicates are removed in the returned list.

11.2.2.6 `reset_servant`

There are occasions in which the communications between the master and the servant can get out of sync, in particular if the user generates an interrupt and then aborts a process in the middle of a remote goal service. In this case, the goal `reset_servant` attempts to return the servant to the top level and to flush the socket.

11.2.2.7 `shutdown_servant`

To close down a servant when it is no longer needed (or to reinitialize it or to connect to another servant with a different database), use `shutdown_servant/0`. This terminates the servant process.

11.2.3 C Process Calling Prolog Process

The support for calling a Prolog goal from a C program consists of Prolog predicates and C functions. The Prolog predicates allow you to create a saved state, which will be the servant. The C functions are used by your C program to create the Prolog process and communicate with it.

Before a C program can call a Prolog program, you must first create a Prolog saved state in which all the predicates to be called are defined. The saved state must also define the characteristics of the interface to each of the predicates to be called. Once this saved state is created, a C program can call the Prolog predicates by using C API functions. We first describe how to create the Prolog saved state. After that we describe how the C program calls a Prolog predicate.

```
:- use_module(library(ccallqp)).
```

11.2.3.1 The Prolog Side

The interface for calling a Prolog program from a C program is strictly typed. In the Prolog servant program, the user must declare which Prolog procedures can be called from the C program, the types of the data elements to be passed between them, and the direction the elements are to be sent. This is done in Prolog by defining `external/3` facts to provide this information. These facts are very similar to those for `foreign/3` and have the following form:

```
external(CommandName, Protocol, PredicateSpecification).
```

CommandName is the name by which the C program invokes this predicate. *Protocol* is the protocol to be used, which currently must be `xdr`. *PredicateSpecification* is a term that describes the Prolog predicate and the interface, and is of the form:

```
PredicateName(ArgSpec1, ArgSpec2, ...)
```

PredicateName is the name of the Prolog predicate (an atom). There is an *ArgSpec* for each argument of the predicate, and *ArgSpec* is one of:

```
+integer        +float          +atom           +string
-integer        -float          -atom           -string
```

Examples:

```
external(add, xdr, addtwoints(+integer,+integer,-integer)).
external(ancestor, xdr, ancestor(+string,-string)).

/* Define addtwoints/3 for use by C caller. */
addtwoints(X, Y, Z) :- Z is X+Y.

/* Define ancestor/2 for use by C caller */
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

The interface allows the simple Prolog data types (atoms, integers, and floating-point numbers) to be passed to and from a calling C program. The '`+`' annotation on an argument specification means that the corresponding value will be passed from the calling C program to the called Prolog predicate. A '`-`' annotation means that the value will be passed from the Prolog predicate back to the calling C program. The '`+`' and '`-`' annotations are always from the point of view of the master (or caller). In this case the C program is the master.

The argument specifications have the same meanings as they do in `foreign/3` facts, but note the directions implied by '`+`' and '`-`'. Also note that the '`...`' specifications are not allowed. The limitations on the sizes of integers, floats, and strings in Prolog are the same as for the interface to foreign routines.

The values passed as `atom` arguments will be treated as unsigned integers in the C program. Their uses must be restricted to the same invocation of the Prolog servant. These integers can be converted to and from the associated strings by using the C functions `QP_ipc_atom_from_string()` and `QP_ipc_string_from_atom()` below.

11.2.3.2 save_ipc_servant(+*SavedState*)

To be able to call a servant from C, you must first have created a saved state that will run as the servant. This is done using `save_ipc_servant/1`. Run Prolog on the machine on which the servant is to be run, and load (that is, compile, consult, or assert) everything that the servant will need. This includes all the `external/3` facts that define the interface, as well as the predicates that the C program will call. Then call `save_ipc_servant(`*SavedState*`)`, where *SavedState* is the name of the file in which to save the state.

11.2.3.3 The C Side

After the saved state containing the Prolog predicates and the interface declarations has been created, a C program can access those predicates by using the C functions described in the following sections. Your C program that uses these functions should #include the file '`IPC/RPC/ccallqp.h`' to include the extern definitions. The final linker step that creates the main executable must include the file '`IPC/RPC/`*system*`/ccallqp.o`'. This file contains the C object code that implements the C functions you will be using, and was created at the time your Quintus Prolog system was installed.

See the RPC demo for an example.

11.2.3.4 QP_ipc_create_servant()

```
int QP_ipc_create_servant(host, QP_save_state, QP_outfile)
    char *host, *QP_save_state, *QP_outfile;
```

host is the name of the machine on which the Prolog servant is to run. If *host* is the string `"local"`, then the servant is run on the same machine as the master. If *host* is the empty string `""`, the servant is run on the local machine but pipes are used for the interprocess communication rather than sockets, which are used otherwise.

QP_save_state is the name of a file containing a Prolog saved state that was created by `save_ipc_servant/1`. This saved state must contain the definitions of the predicates to be called and the `external/3` facts that specify the interface. If *QP_save_state* is not an absolute filename, it will be sought, on the specified machine, in the sequence of directories specified by your `PATH` environment variable. If this search fails, the current working directory will be tried, if it exists on that machine.

QP_outfile is the name of the file to which to route the Prolog servant's output (`stdout` and `stderr`). If *QP_outfile* is the string `"user"`, then the servant's output is routed to the

C program's `stdout`. If it is the empty string `""`, the servant's `stdout` stream is discarded, and its `stderr` stream is routed to the master's `stderr`.

This routine returns the file descriptor of the connecting socket if the connection was made successfully and -1 if not. This routine starts the Prolog servant and may take several seconds to complete.

11.2.3.5 QP_ipc_lookup()

```
int QP_ipc_lookup(name)
    char *name;
```

This routine finds and returns the command number associated with the external routine *name*. The command number is a nonnegative integer. The associated command must have appeared as the first field in an `external` declaration in the saved state started by the previous `QP_ipc_create_servant()` call. If the command is not found, -1 is returned.

11.2.3.6 QP_ipc_prepare()

```
int QP_ipc_prepare(command, arg1, ..., argn)
    int command;
    va_dcl
```

This function sends a request to the Prolog process to evaluate a goal. The command is identified by *command*, which must have been obtained by an earlier call to `QP_ipc_lookup()`. The arguments are the values to be sent to the command, that is those in the interface specification with a '`+`' annotation. They must be in left-to-right order as they appear in the specification. They must be of the corresponding types as indicated in the specification:

```
+integer:       int
+float:         float
+atom:          QP_atom
+string:        char *
```

See the example below in Section 11.2.3.12 [ipc-rpc-cpp-exa], page 514.

On successful completion, this routine returns 0. If an error has occurred, it returns -1.

11.2.3.7 QP_ipc_next()

```
int QP_ipc_next(command, arg1, ..., argm)
    int command;
    va_dcl
```

The C routine `QP_ipc_next()` retrieves an answer for a goal (command) that was initiated by a previous call to `QP_ipc_prepare()`. The command is identified by *command*. It must

be the same as the command given to `QP_ipc_prepare()`. The remaining arguments are variables that will be set by `QP_ipc_next()` to the values returned as the next answer to the goal. There is one argument for each field in the interface specification that was annotated with '`-`'. They must be in left-to-right order. The types of the arguments must correspond to the types declared in the `external/3` specification as follows:

```
-integer:       int *
-float:         float *
-atom:          QP_atom *
-string:        char **
```

For a returned value of type string, space must be provided by the calling routine to hold the value returned. The characters of the returned string will be copied over the string passed in.

If an(other) answer to the query has been obtained and the argument parameters have been set accordingly, `QP_ipc_next()` returns 0. If there are no more answers, it returns -1. If there is an error, it returns -2.

See the example below in Section 11.2.3.12 [ipc-rpc-cpp-exa], page 514.

11.2.3.8 `QP_ipc_close()`

```
int QP_ipc_close()
```

The `QP_ipc_close()` routine closes a query that was opened by `QP_ipc_prepare()` but did not have all of its answers retrieved by calls to `QP_ipc_next()`. (When `QP_ipc_next()` returns a -1, indicating no more answers, the query is automatically closed, and a subsequent call to `QP_ipc_close()` is an error.) `QP_ipc_close()` returns 0 if it has closed the query successfully, and -1 if there is an error.

11.2.3.9 `QP_ipc_shutdown_servant()`

```
int QP_ipc_shutdown_servant()
```

The routine `QP_ipc_shutdown_servant()` shuts down the servant. It sends a message to the servant that causes it to terminate. It returns 0 if the shutdown is successful, and -1 if there is some problem.

11.2.3.10 `QP_ipc_atom_from_string()`

```
QP_atom QP_ipc_atom_from_string(str)
    char *str;
```

The routine `QP_ipc_atom_from_string()` returns the unsigned integer that is the Prolog representation of the atom with the name *str*. This representation is valid for the lifetime of the servant. It must not be saved and used with a different invocation of the servant.

11.2.3.11 QP_ipc_string_from_atom()

```
void QP_ipc_string_from_atom(atom, str)
    QP_atom atom;
    char *str;
```

The routine `QP_ipc_string_from_atom()` can be used to get the name corresponding to an atom, given the Prolog internal unsigned integer representation of the atom. The string *str* will be overwritten with a null-terminated string that is the name of the atom. The caller must provide enough space to contain this string.

11.2.3.12 Examples

The first example shows how you can package a call to a Prolog goal that is known to be determinate. Here, the C function `fred` hides the call to Prolog. However, the servant must be initiated by a call to `QP_ipc_create_servant()` before it can be called.

Prolog Specification

```
external(fred, xdr, fred(+integer,-integer,+integer)).

fred(X, Y, Z) :- ...
```

C routine

```
int fred(i, j)
    int i, j;
    {
        static int fredp = -1;
        int k;

        if (fredp < 0) {        /* initialize */
            fredp = QP_ipc_lookup("fred");
            if (fredp < 0)
                DieBecause("couldn't find fred");
        }

        /* send the request */
        QP_ipc_prepare(fredp, i, j);

        /* get the answer back */
        if (QP_ipc_next(fredp, &k))
            DieBecause("fred failed");

        /* known determinate, so close request */
        QP_ipc_close();

        /* return the answer */
        return k;
    }
```

The second example shows an entire program and how all types of arguments are be passed. It also shows how `QP_ipc_atom_from_string()` and `QP_ipc_string_from_atom()` can be used. In terms of functionality, this is not a very interesting program, and the conversion between atoms and strings is just to give an example.

Prolog Specification

```
external(dupl, xdr, duplicate(-integer,+integer,-string,+string,
                      -float,+float,-atom,+atom)).

duplicate(A, A, B, B, C, C, D, D).
```

C program:

```
main()
{
    int pdupl;
    char host[20], savestate[50];
    int iint, oint;
    char istr1[20], istr2[20], ostr1[20], ostr2[20];
    float iflt, oflt;
    QP_atom iatom, oatom;

        printf("Enter host and savestate: ");
        scanf("%s%s", host, savestate);

        if (QP_ipc_create_servant(host,savestate,"servant_out"))
                DieBecause("Error starting up servant");

        pdupl = QP_ipc_lookup("dupl");
        if (pdupl < 0) DieBecause("dupl not defined");

        for (;;) {      /* loop until break */
            printf("Enter int, str, flt, str: ");
            if (scanf("%d%s%f%s",&iint,istr1,&iflt,istr2) != 4)
                break;
            /* get atom for the string typed in */
            iatom = QP_ipc_atom_from_string(istr2);

            /* send the request */
            if (QP_ipc_prepare(pdupl, iint, istr1, iflt, iatom))
                DieBecause("dupl prepare error");

            /* get answer back, and convert atom back to string */
            QP_ipc_next(pdupl, &oint, ostr1, &oflt, &oatom);
            QP_ipc_string_from_atom(oatom, ostr2);

            /* close request because we want only one answer */
            if (QP_ipc_close()) printf("ERROR closing\n");

            printf("Answer is: %d %s %G %s(%d)\n",
                        oint, ostr1, oflt, ostr2, oatom);
        }

        if (QP_ipc_shutdown_servant())
            DieBecause("Error shutting down servant");
}
```

The third example shows how to retrieve multiple answers:

Prolog Specification

```
external(table, xdr, table(-string,-integer)).

        table(samuel, 34).
        table(sarah, 54).
        ...
```

C program

```
main()
{
        char host[20], savestate[50];
        int ptable, ret;
        char strval[40];
        int intval;

        printf("Enter host and savestate: ");
        scanf("%s%s", host, savestate);

        if (QP_ipc_create_servant(host,savestate,"servant_out"))
                DieBecause("Error starting up servant");

        ptable = QP_ipc_lookup("table");
        if (ptable < 0) {
            printf("table not defined\n");
            return;
        }

        /* send the request */
        QP_ipc_prepare(ptable);

        /* retrieve and print ALL answers */
        while (!(ret = QP_ipc_next(ptable, strval, &intval)))
            printf("String: %s, Integer: %d\n", strval,intval);

        /* note no close, since we retrieved all the answers! */
        if (ret == -1) printf("All answers retrieved\n");
        else printf("Error retrieving answers\n");

        if (QP_ipc_shutdown_servant())
            DieBecause("Error shutting down servant");
}
```

The final example shows how one could write a C function to turn Prolog's message tracing (see Section 11.2.4 [ipc-rpc-tra], page 518) on and off.

Prolog Specification

```
external(settrace, xdr, settrace(+string)).

settrace(X) :- msg_trace(_,X).
```

C routine

```
void settrace(OnOff)
    char *OnOff;
    {
        static int psettrace = -1;
        int k;

        if (psettrace < 0) {
            psettrace = QP_ipc_lookup("settrace");
            if (psettrace < 0)
                DieBecause("couldn't find settrace");
        }

        QP_ipc_prepare(psettrace, OnOff);
        if (QP_ipc_next(psettrace))
            DieBecause("settrace failed");
        QP_ipc_close();
    }
```

11.2.4 Tracing

A simple tracing facility is available for determining what messages are received by and sent from the Prolog servant. When message tracing is on, messages sent or received cause a trace message to be written to the current output stream. It will normally be redirected to a file by `create_servant/3` or `QP_ipc_create_servant()`. The UNIX command *`tail -f`* may be helpful in looking at the trace messages. Each trace message indicates what the corresponding interprocess message was. The precise form of the trace information depends on whether the Prolog servant is serving a C program or another Prolog program.

Message tracing can be turned on and off by having the servant process call `msg_trace/2` which is described below. A master that is a Prolog process can use `call_servant/1` to cause the servant to call `msg_trace/2`. It can also call `msg_trace/2` directly to control tracing of its own messages.

To make a servant that serves a C master trace its message, it must either have had tracing turned on before its saved state was created, or it must provide an `external` routine that can be invoked by the C master to turn on tracing (see Example 4 in Section 11.2.3.12 [ipc-rpc-cpp-exa], page 514).

11.2.4.1 `msg_trace(-OldValue, +OnOrOff)`

The predicate `msg_trace/2` returns the current value of the message-trace flag (`on` or `off`) and resets its value. The message-trace flag has one of the values `on` or `off`. *OldValue* is bound to the previous value of the flag, and the flag is reset to the value of *OnOrOff*, which must be either `on` or `off`. The call `msg_trace(X,X)` returns the current value without changing it. When the message-trace flag is on, messages to and from the servant are traced. By executing `call_servant(msg_trace(_,on))`, tracing can be turned on in the servant.

11.2.5 Known Bugs

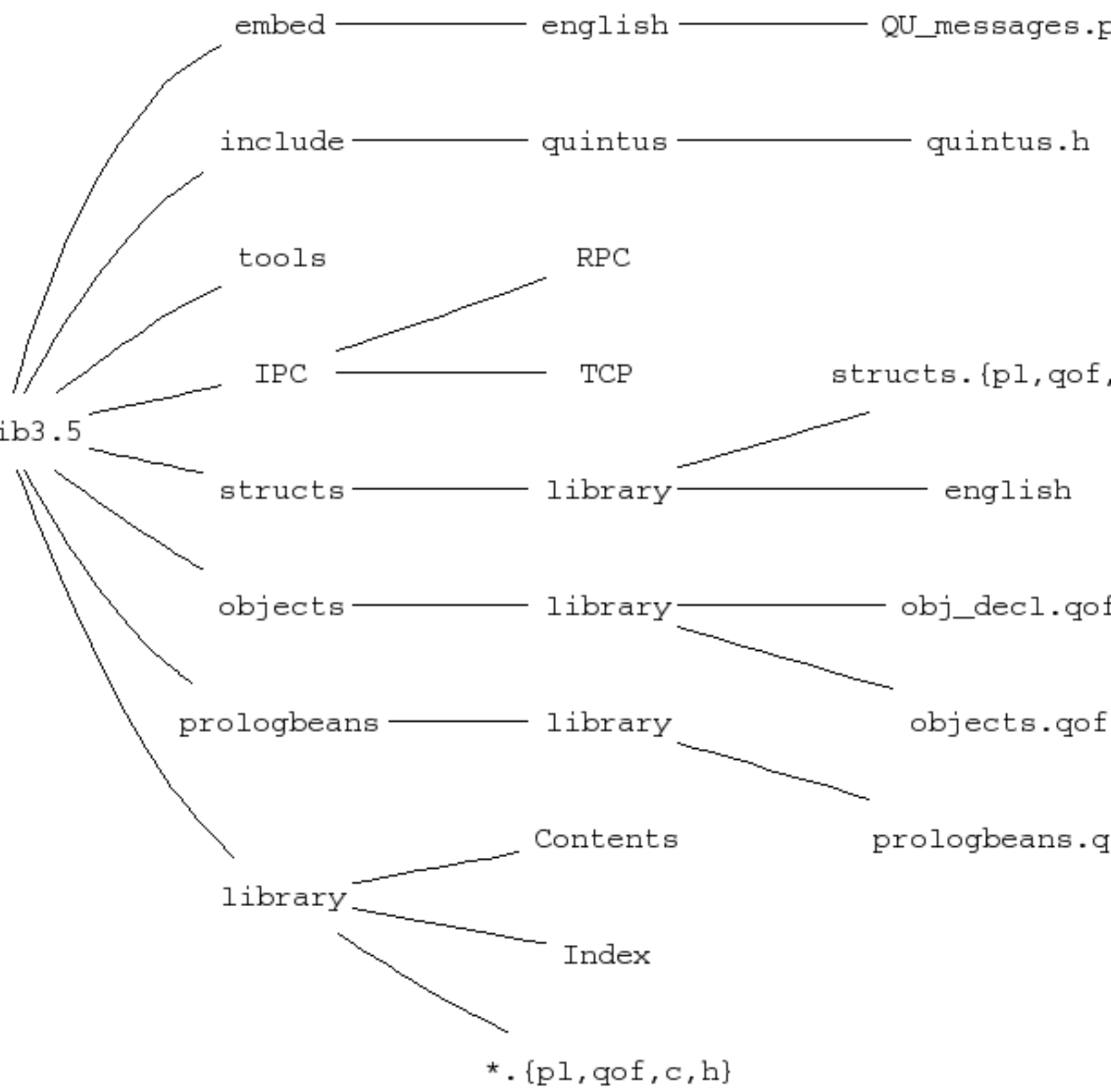
If the Prolog master process is interrupted while it is waiting for an answer from the servant process, the master process may crash.

12 Library

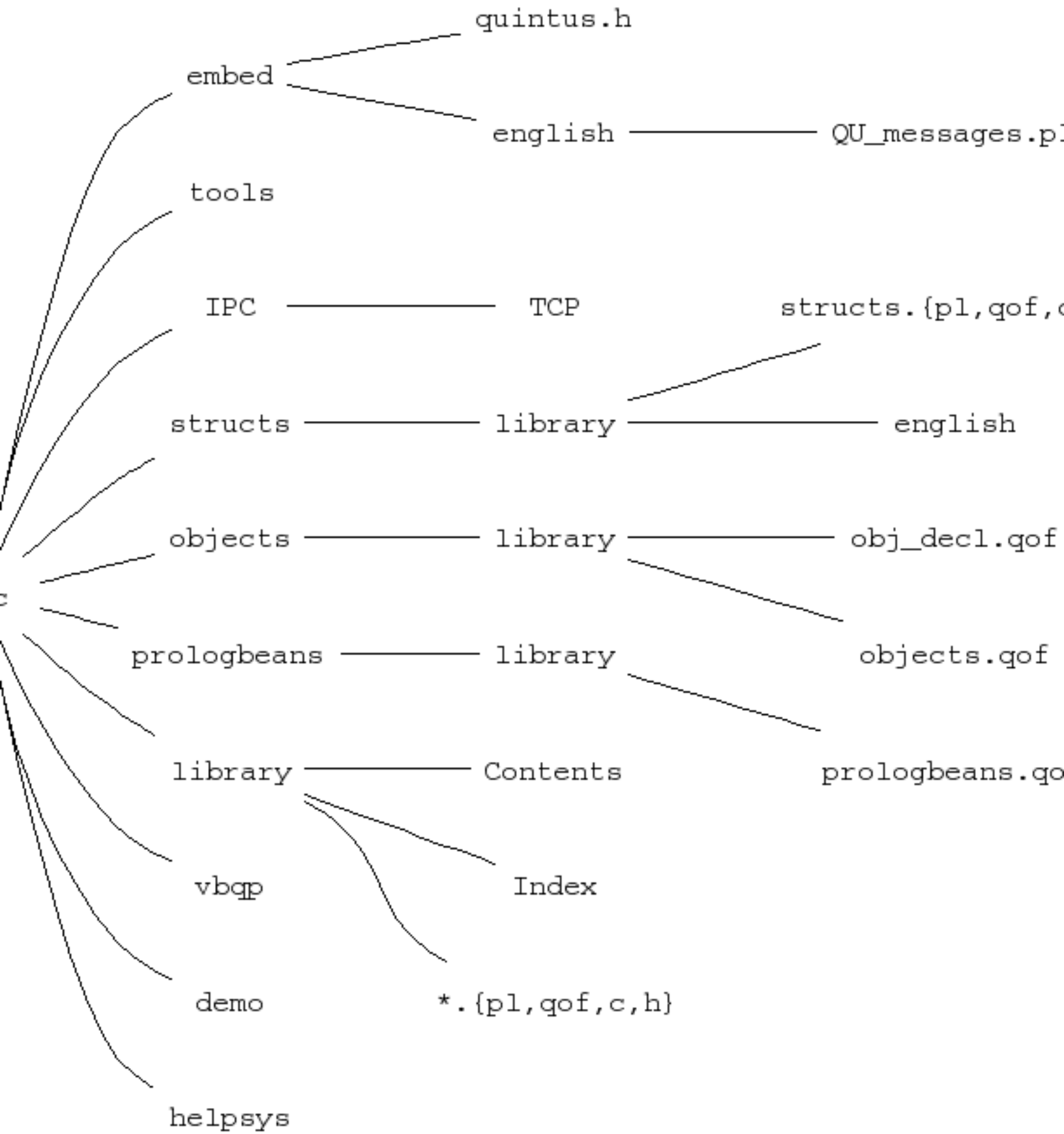
12.1 Introduction

12.1.1 Directory Structure

The Quintus Prolog Library directory (part of the installation directory described in Section 1.1 [int-man], page 1) contain files written in Prolog and C, which supplement the Quintus Prolog kernel. The structure of the Library Directory differs slightly between UNIX and Windows, as shown in the following figures.


embed
english
QU_messages.p
include
quintus
quintus.h
tools
RPC
IPC
TCP
structs.{pl,qof,
ib3.5
structs
library
english
objects
library
obj_decl.qof
prologbeans
library
objects.qof
prologbeans.q
Contents
library
Index
*.{pl,qof,c,h}

The Quintus Prolog Library Directory under UNIX, ‘`qplib3.5`’


quintus.h
embed
english
QU_messages.pl
tools
IPC
TCP
structs.{pl,qof,
structs
library
english
objects
library
obj_decl.qof
prologbeans
library
objects.qof
library
Contents
prologbeans.qo
vbqp
Index
demo
*.{pl,qof,c,h}
helpsys

The Quintus Prolog Library Directory under Windows, '`src`'

'`library`' contains a large number of predicates that can be regarded as extensions to the set of predicates that are built into the Prolog system. Both source and QOF versions are provided.

'`tools`' source files for Quintus-supplied development tools, independent programs that perform various functions such as determinancy checking and cross-referencing. They can be used to analyze your programs statically (that is, without running them) and possibly locate bugs.

'`structs`' contains the Structs Package, which allows access to C data structures from Prolog. The directory includes demos.

'`objects`' contains the Objects Package, which enables programmers to write object-oriented programs in Quintus Prolog. It can be regarded as a high-level alternative to the Structs Package.

'`prologbeans`'
contains the PrologBeans Package, which provides an interface from Java to Prolog.

'`vbqp`' An interface from Visual Basic to Prolog. Only available under Windows.

'`IPC`' two inter-process communication packages: Remote Predicate Call (RPC, not available under Windows) and Transmission Control Protocol (TCP).

'`include`' Contains '`<quintus/quintus.h>`', a header file containing #defines, typedefs, struct definitions, etc., which are needed to compile C code that needs to call API functions or use Quintus data structures. Under Windows, the '`include`' directory is placed directly under *quintus-directory*.

'`embed`' contains modules for user customization of the message handler (see Section 8.20 [ref-msg], page 325) and source code for the Embedding Layer.

The predicate `library_directory/1` has predefined clauses for the '`library`' and '`tools`' directories. These depend on the `file_search_path/2` definition of '`qplib`'. You can see these clauses by typing *`listing.`* or *`listing(library_directory).`* after starting up Prolog. This definition of `library_directory/1` means that you can refer, from within Prolog, to any file in any of these areas using the form `library(`*`File`*`)`. For example either (A) or (B) would load the file '`lists.qof`' from the '`library`' directory.

```
| ?- [library(lists)].                                    (A)

| ?- load_files(library(lists)).                          (B)
```

Library packages are typically loaded by doing (C) if the package is not a module-file or if it is a module and you want all the exported predicates.

```
| ?- ensure_loaded(library(addportray)).                  (C)
```

See the descriptions of `ensure_loaded/1` and `use_module/[1,2,3]`.

In addition to the loadable QOF files, source files ('`.pl`' or '`.c`') are provided for each package.

12.1.2 Status of Library Packages

The predicates described here have been tested and are believed to work as documented. If you want something slightly different from one of these predicates, it is strongly recommended that you *do not change* the existing definition. Instead, write a *new* predicate using the existing definition as a model. There are several reasons for not changing the definition of a predicate in the library:

1. It may confuse people reading your code who are familiar with the documented behavior of the library predicate.
2. If you use other library files, they may depend on the exact definition of this predicate.
3. You might have to redo your modification if you wished to run your program on some new release of Quintus Prolog.
4. We do not accept responsibility for any bugs introduced by a user's change to library code.

12.1.3 Documentation of Library Packages

All library packages include code comments, often extensive, which serve as documentation. Accessing these comments and other information available in the library directory is discussed in Section 12.1.3.1 [lib-bas-dlp-acc], page 526.

In addition, many packages are more fully documented:

- The IPC packages are documented in Chapter 11 [ipc], page 485.
- A large number are documented in Section 12.2 [lib-lis], page 528 through Section 12.11 [lib-mis], page 635 of this part of the manual.

The rest are abstracted in Section 12.13 [lib-abs], page 641. For these the information in the following section is particularly useful.

12.1.3.1 Accessing Code Comments

If you know the name of the library package in question, simply look at the source code in '`qplib3.5`'. Apart from code comments, there is information about predicates in two files that summarize the contents of the directory. They are called '`Contents`' and '`Index`'. (See the figure above.) If you do not know the name of the package the '`Index`' file will be helpful. If you know the name of the package, you can use the '`Contents`' file to gain further information about it.

The '`Index`' file contains one line for each exported predicate in the library. The predicates are listed in standard order, ignoring module names. A typical entry looks like this:

```
list_to_binary/4   flatten   ./flatten.pl
```

This means that there is a predicate called `list_to_binary/4` in the library, that it lives in a module called `flatten`, and that the file that contains it is '`flatten.pl`' in the same directory as the Index.

If you have set up an environment variable `QL` holding the name of the Quintus library directory, you could ask "what predicates are there to deal with files?" by issuing the command

```
% egrep files $QL/Index
```

The '`Contents`' file is organized by library files rather than by predicates. A typical entry in this file is a block of lines like this:

```
basics  + documented in manual
basics  % the basic list processing predicates
basics  - ./basics.pl
basics  : member/2
basics  : memberchk/2
basics  : nonmember/2
```

The first line means that `library(basics)` is one of the library packages that is fully documented in this manual. The second line is a short description of the contents. The third line says which file contains `library(basics)`; in this case it is '`basics.pl`' in the same directory as Contents. The remaining lines list the predicates exported by `library(basics)`. You could obtain this information by issuing the command

```
% egrep '^basics' $QL/Contents
```

These files are provided as a convenience, and do not have the same authority as the printed manual.

12.1.4 Notation

12.1.4.1 Character Codes

Many of the examples in this manual show lists of character codes being written as quoted strings. This actually happens if you load the library package `library(printchars)`. That package extends the predicate `portray/1` (using `library(addportray)`) so that `print/1`, the top-level, and the debugger will write lists of character codes as follows:

```
| ?- X = [0'a,0'b,0'c].
X = [97,98,99]

| ?- ensure_loaded(library(printchars)).

| ?- X = [0'a,0'b,0'c].
X = "abc"
```

12.1.4.2 Mode Annotations

A new system of representing the modes of arguments has been adapted in Release 3, is described in Chapter 18 [mpg], page 985. The library, including IPC, is still documented under the old system of mode annotations:

Each predicate definition is headed by a goal template such as

```
setof(?X,+Goal,-Set)
```

Here *X* and the others are meta-variables, which name the arguments so that we don't have to keep saying "its first argument" and so on. The characters that precede the meta-variables will seem familiar if you know the mode declarations of DEC-10 Prolog; their significance is as follows:

'+' This argument is an input to the predicate. It must initially be instantiated; otherwise, the predicate raises an error exception.

'-' This argument is an output. It is returned by the predicate. That is, the output value is unified with any value that was supplied for this argument. The predicate fails if this unification fails. If no value is supplied, the predicate succeeds, and the output variable is unified with the return value.

'?' This argument does not fall into either of the above categories. It is not necessarily an input nor an output, and it need not be instantiated.

Note that it is not an error to call a predicate with a '-' argument already instantiated. The value supplied will simply be unified with the result returned, and if that unification fails, the predicate fails.

12.2 List Processing

12.2.1 Introduction

While Prolog has data structures other than lists (and the library implements several others, such as priority queues, in terms of more primitive structures), list processing is still very important in Prolog. This section describes the library predicates that operate on lists.

12.2.2 What is a "Proper" List?

Several of the predicate descriptions below indicate that a particular predicate only works when a particular argument "is a proper list". A *proper list* is either the atom `[]` or else it is of the form `[_|L]` where L is a proper list. X is a *partial list* if and only if `var(X)` or X is `[_|L]` where L is a partial list. A term is a list if it is either a proper list or a partial list; that is, `[_|foo]` is not normally considered to be a list because its tail is neither a variable nor `[]`.

Note that the predicate `is_list(X)` defined in `library(lists)` really tests whether X is a *proper* list. The name is retained for compatibility with earlier releases of the library. Similarly, `is_set(X)` and `is_ordset(X)` test whether X is a proper list that possesses the additional properties defining sets and ordered sets.

The point of the definition of a proper list is that a recursive procedure working its way down a proper list can be certain of terminating. Let us take the case of `last/2` as an example. `last(X, L)` ought to be true when `append(_, [X], L)` is true. The obvious way of doing this is

```
last(Last, [Last]).
last(Last, [_|Tail]) :-
        last(Last, Tail).
```

If called with the second argument a proper list, this definition can be sure of terminating (though it will leave an extra choice point behind). However, if you call

```
| ?- last(X, L), length(L, 0).
```

where `L` is a variable, it will backtrack forever, trying ever longer lists. Therefore, users should be sure that only proper lists are used in those argument positions that require them.

12.2.3 Five List Processing Packages

There are five library files that are specifically concerned with list processing. They are

`library(basics)`
: contains very basic list processing operations.

`library(lists)`
: contains operations that view lists as sequences.

`library(sets)`
: contains operations that view lists as sets.

`library(ordsets)`
: contains operations that view lists as sets, but require that the elements of the lists be in standard order (see `compare/3` in the reference pages) so as to be much more efficient than `library(sets)` for any but the smallest sets.

`library(listparts)`
: establishes a common vocabulary for names of parts of lists.

As a general rule, if a predicate defined here has a counter (a non-negative integer) as one of its arguments, it will suffice for the counter argument to be instantiated. Otherwise, at least one of the list arguments must be a proper list. Failing this, the predicate may backtrack forever trying ever longer lists. When you look at the code you will see that some of the library routines use `same_length/2` or `same_length/3` to ensure termination.

12.2.4 Basic List Processing — `library(basics)`

12.2.4.1 Related Built-in Predicates

See also the built-in predicates `length/2` and `append/3`, which can be used to find the length of a proper list or to construct a proper list of a given length, and `append(*Head, *Tail, *List)`, which is used to combine lists and take lists apart.

12.2.4.2 `member(?Element, ?List)`

`member(?Element, ?List)` is true when *List* is a (possibly partial) list, and *Element* is one of its elements. It may be used to check whether a particular element occurs in a given list, or to enumerate all of the elements of a list by backtracking. `member/2` may also be used to generate a list.

```
| ?- member(a, [b,e,a,r]).

yes

| ?- member(e, [s,e,e,n]).

yes   /* this will succeed twice */

| ?- member(e, [t,o,l,d]).

no

| ?- member(X-Y, [light-dark,near-far,wet-dry]).

X = light,
Y = dark ;

X = near,
Y = far ;

X = wet,
Y = dry

| ?- member(a-X, [b-2,Y-3,X-Y]).

X = 3,
Y = a ;

X = a,
Y = a

| ?- member(a, L), member(b, L), member(c, L),
|    length(L, N).

L = [a,b,c],
N = 3
```

The last example will generate lists of increasing length whose first three members are `a`, `b`, and `c`.

If L is a proper list of length n, `member(X, L)` has at most n solutions, whatever X is. But if L is a partial list, `member/2` will backtrack indefinitely, trying to place X ever farther to the right. For example,

```
| ?- member(a, L).

L = [a|_879] ;

L = [_878,a|_881] ;

L = [_878,_880,a|_883] ;
   .
   .
   .
```

until you stop it.

In general, you should only use `member/2` when the second argument is a proper list. This list need not be ground; however, it must not end with a variable.

12.2.4.3 memberchk(+*Element*, +*List*)

In the previous section, it was pointed out that `member(e, [s,e,e,n])` succeeds twice. If you have a ground term (or one that is sufficiently instantiated) and you only want to know whether it occurs in a list or not, you would like the membership test to succeed only once. `memberchk/2` is a version of `member/2` that does this.

`memberchk(`*Element*`, `*List*`)` can only be used to test whether a known element occurs in a known list. It cannot be used to enumerate elements of the list. `memberchk/2` commits to the first match and does not backtrack.

Use `memberchk/2` in preference to `member/2`, but only where its restrictions are appropriate.

12.2.4.4 nonmember(+*Element*, +*List*)

`nonmember(+`*Element*`, +`*List*`)` is true when *Element* does not occur in the *List*. For `nonmember/2` to instantiate *Element* in any way would be meaningless, as there are infinitely many terms that do not occur in any given list.

`nonmember/2` should only be used when *List* and *Element* are sufficiently instantiated that you can tell whether *Element* occurs in *List* or not without instantiating any variables. If this requirement is not met, the answers generated may not be exactly what you would expect from the logic.

For example, some valid uses of `nonmember/2` are:

```
| ?- nonmember(a, [x,y,z]).

yes
| ?- nonmember(x, [x,y,z]).

no
```

In the following examples, `nonmember/2` is invalidly used with insufficiently instantiated arguments. In these cases it simply fails.

```
| ?- nonmember(X, [x,y,z]).

no
| ?- nonmember(x, [X]).

no
| ?- nonmember(x, X).

no
```

Use `nonmember/2` to check whether a known element occurs in a known list, in preference to '`\+ member/2`' or '`\+ memberchk/2`'.

12.2.5 Lists as Sequences — `library(lists)`

`library(lists)` provides a large number of list processing operations. See also Section 12.2.4 [lib-lis-basics], page 530, which describes the more basic list processing operations that are provided by `library(basics)`.

The predicates defined by this library file are:

`is_list(+List)`
: is true when *List* is instantiated to a proper list: that is, to either `[]` or `[_|Tail]` where *Tail* is a proper list. A variable, or a list that ends with a variable, will fail this test.

`append(+ListOfLists, ?List)`
: is true when *ListOfLists* is a list `[L1,...,Ln]` of lists, *List* is a list, and appending *L1*, ..., *Ln* together yields *List*. If *ListOfLists* is not a proper list, `append/2` will fail. Additionally, either List should be a proper list, or each of *L1*, ..., *Ln* should be a proper list. The behavior on non-lists is undefined. *ListOfLists* must be proper because for any given solution, infinitely many more can be obtained by inserting nils (`[]`) into *ListOfList*.

`append(?Prefix, ?Tail1, ?List1, ?Tail2, ?List2)`
: is logically equivalent to:

```
append(Prefix, Tail1, List1),
append(Prefix, Tail2, List2).
```

but is much more efficient. `append/5` is guaranteed to halt in finite time if any one of *Prefix*, *List1*, or *List2* is a proper list.

You can use `append/5` to add a common *Prefix* to the front of *Tail1* and *Tail2*, to remove a common *Prefix* from *List1* and *List2*, or in several other ways.

Here is an example of the use of `append/5`. The task is to check whether *Word1* and *Word2* are the same except for exactly one insertion, deletion, or transposition error.

```
spell(i, Word1, Word2) :-
   append(_, Suffix, Word1, [_|Suffix], Word2).
spell(d, Word1, Word2) :-
   append(_, [_|Suffix], Word1, Suffix, Word2).
spell(t, Word1, Word2) :-
   append(_, [X,Y|Suffix], Word1, [Y,X|Suffix], Word2).

| ?- spell(E, Word1, "fog"),
     print(E-Word1), nl, fail.
i-"og"
i-"fg"
i-"fo"
d-[_682,102,111,103]
d-[102,_682,111,103]
d-[102,111,_682,103]
d-[102,111,103,_682]
t-"ofg"
t-"fgo"
no
```

`correspond(?X, ?Xlist, ?Ylist, ?Y)`
: is true when *Xlist* and *Ylist* are lists, *X* is an element of *Xlist*, *Y* is an element of *Ylist*, and *X* and *Y* are in corresponding places in their lists. Nothing is said about the other elements of the two lists, nor even whether they are the same length. Only one solution is ever found, as the procedure for `correspond/4` contains a cut. For a logical predicate having similar effects (that is, one that finds all solutions), see `select/4`. Either *Xlist* or *Ylist* should be a proper list.

`delete(+List, +Elem, ?Residue)`
: is true when *List* is a list, in which *Elem* may or may not occur, and *Residue* is a copy of *List* with all elements equal to *Elem* deleted. To extract a single copy of *Elem*, use `select(Elem, List, Residue)`. For a given *Elem* and *Residue*, there are infinitely many *List*s containing *Elem* or not. Therefore, this predicate only works one way around: *List* must be a proper list and *Elem* should be instantiated. Only one solution is ever found.

`delete(+List, +Elem, +Count, ?Residue)`
: is true when *List* is a list, in which *Elem* may or may not occur, and *Count* is a non-negative integer. *Residue* is a copy of *List* with the first *Count* elements equal to *Elem* deleted. If *List* has fewer than *Count* elements equal to *Count*,

all of them are deleted. If *List* is not proper, `delete/4` may fail. *Elem* and the elements of *List* should be sufficiently instantiated for `\=` to be sound.

`keys_and_values(?KeyValList, ?KeyList, ?ValList)`
is true when all three arguments are lists of the same length (at least one of them should be a proper list), and are of the form

```
KeyValList = [K1-V1,K2-V2,...,Kn-Vn]
KeyList    = [K1,   K2   ,...,Kn   ]
ValList    = [   V1,   V2,...,   Vn]
```

That is, the *i*th element of *KeyValList* is a pair *Ki-Vi*, where *Ki* is the *i*th element of *KeyList* and *Vi* is the *i*th element of *ValList*. The main point of this, of course, is that *KeyValList* is the kind of list that the built-in predicate `keysort/2` sorts, where the *Ki* are the keys that are sorted on and the *Vi* go along for the ride. You can, for example, sort a list without discarding duplicate elements, using

```
msort(Raw, Sorted) :-
        keys_and_values(RawKV, Raw, _),
        keysort(RawKV, SortedKV),
        keys_and_values(SortedKV, Sorted, _).
```

`keys_and_values/3` can also be used for constructing the input (list) argument of `list_to_map/2` and for decomposing the result of `map_to_list/2` — see `library(maps)` (Section 12.13 [lib-abs], page 641).

```
| ?- keys_and_values([light-dark,
                      near-far,
                      wet-dry],
                      Keys, Vals).

Keys = [light,near,wet],
Vals = [dark,far,dry]

| ?- keys_and_values(Pairs, [light,near,wet],
|                           [dark,far,dry]).

Pairs = [light-dark,near-far,wet-dry]
```

`last(?Last, +List)`
is true when *List* is a list and *Last* is its last element. This could be defined as

```
last(X, L) :-
   append(_, [X], L).
```

`nextto(?X, ?Y, +List)`
is true when *X* and *Y* appear side-by-side in *List*. It could be defined as

```
nextto(X, Y, List) :- append(_, [X,Y|_], List).
```

`nextto/3` may be used to enumerate successive pairs from *List*. *List* should be a proper list.

`nth0(?N, ?List, ?Elem)`
: is true when *Elem* is the *N*th member of *List*, counting the first as element 0 (that is, throw away the first *N* elements and unify *Elem* with the next one). Note that the argument pattern resembles that of `arg/3`. Unlike `arg/3` (but like `genarg/3`; see Section 12.3.3 [lib-tma-arg], page 551), `nth0/3` can be used to solve for either *N* or *Elem*. If used to solve for *N*, *List* should be a proper list.

`nth0(?N, ?List, ?Elem, ?Rest)`
: unifies *Elem* with the *N*th element of *List*, counting from 0, and *Rest* with the remaining elements. `nth0/4` can be used to select the *N*th element of *List* (yielding *Elem* and *Rest*), or to insert *Elem* before the *N*th (counting from 0) element of *Rest*, (yielding *List*). Either *N* should be instantiated, or *List* should be a proper list, or *Rest* should be a proper list; any one is enough.

```
| ?- nth0(2, List, c, [a,b,d,e]).

List = [a,b,c,d,e]

| ?- nth0(2, [a,b,c,d,e], Elem, Rest).

Elem = c,
Rest = [a,b,d,e]
| ?- nth0(N, [a,b,c,d,e], c, Rest).

N = 2,
Rest = [a,b,d,e]

| ?- nth0(1, List, Elem, Rest).

List = [_973,Elem|_976],
Elem = _755,
Rest = [_973|_976]
```

`nth1(?N, ?List, ?Elem)`
: is the same as `nth0/3`, except that it counts from 1 so that, for example,

```
nth1(1, [H|T], H)
```

: is true. *List* should be a proper list.

`nth1(?N, ?List, ?Elem, ?Rest)`
: is the same as `nth0/4` except that it counts from 1. It can be used to select the *N*th element of *List* (yielding *Elem* and *Rest*), or to insert *Elem* before the *N* +1st element of *Rest*, when it yields *List*. Either *N* should be instantiated, or *List* should be a proper list, or *Rest* should be a proper list; any one is enough.

```
| ?- nth1(3, List, c, [a,b,d,e]).

List = [a,b,c,d,e]

| ?- nth1(3, [a,b,c,d,e], Elem, Rest).

Elem = c
Rest = [a,b,d,e]
| ?- nth1(N, [a,b,c,d,e], c, Rest).

N = 3
Rest = [a,b,d,e]

| ?- nth1(1, List, Elem, Rest).

List = [Elem|Rest],
Elem = _755,
Rest = _770
```

`perm(+List, ?Perm)`
: is true when *List* and *Perm* are permutations of each other. If you simply want to test this, the best way is to sort the two lists and see if the results are the same; use `samsort/2` from `library(samsort)` (Section 12.13 [lib-abs], page 641) in preference to `sort/2`.

  The point of `perm/2` is to generate permutations; it only works if *List* is a proper list. `perm/2` should not be used in new programs; use `permutation/2` instead.

`permutation(?List, ?Perm)`
: is true when *List* and *Perm* are permutations of each other. Unlike `perm/2`, it will work even when *List* is not a proper list. `permutation/2` will return reasonable results when *Perm* is also not proper, but will still backtrack forever unless one of the arguments is proper. Be careful: `permutation/2` is is quite efficient, but the number of permutations of an *N*-element list is *N!* (*N*-factorial). Even for a 7-element list that is 5040.

`perm2(?A, ?B, ?C, ?D)`
: is true when `[A, B]` is a permutation of `[C, D]`. `perm2/4` is very useful for writing pattern matchers over commutative operators. It is used more often than `perm/2`. `perm2/4` is not really an operation on lists. `perm2/4` is in `library(lists)` only because `permutation/2` is there.

`remove_dups(+List, ?Pruned)`
: removes duplicated elements from *List*, which should be a proper list. If *List* contains non-ground elements, *Pruned* may contain elements that unify. Two elements will be considered duplicates if and only if all possible substitutions cause them to be identical.

```
| ?- remove_dups([X,X], L).

X = _123
L = [X]

| ?- remove_dups([X,Y], L).

X = _123
Y = _126
L = [X,Y]

| ?- remove_dups([3,1,4,1], L).

L = [1,3,4]
```

`remove_dups/2` does not preserve the original order of the elements of *List*.

`rev(+`*List*`, ?`*Reversed*`)`
: is true when *List* and *Reversed* are lists with the same elements but in opposite orders. *List* must be supplied as a proper list; if *List* is partial, `rev/2` may find a solution, but if backtracked into will backtrack forever, trying ever longer lists. Use `rev/2` only when you know that *List* is proper; it is then twice as fast as calling `reverse/2`.

`reverse(?`*List*`, ?`*Reversed*`)`
: is true when *List* and *Reversed* are lists with the same elements but in opposite orders. Either *List* or *Reversed* should be a proper list: given either argument the other can be found. If both are partial, `reverse/2` will keep trying longer instances of both. If you want an invertible relation, use this. If you only want the reversal to work one way around, `rev/2` is adequate.

`same_length(?`*List1*`, ?`*List2*`)`
: is true when *List1* and *List2* are both lists and have the same number of elements. No relation between the elements of *List1* and *List2* is implied. This predicate may be used to generate either list given the other, or indeed to generate two lists of the same length, in which case the arguments will be bound to lists of length 0, 1, 2, and so on. `same_length/2` is supplied to make it easier to write invertible predicates, transferring the proper list status of either argument to the other. same_length(*List1, List2*) has the same effect as the following call to `same_length/3`:

```
same_length(List1, List2, _ /* any length */)
```

`same_length(?`*List1*`, ?`*List2*`, ?`*Length*`)`
: is true when *List1* and *List2* are both lists and have the same number of elements, and *Length* is an integer that is the common length of the two lists. No relation between the elements of *List1* and *List2* is implied. This predicate may be used to generate any of its arguments. If *Length* is given, or if either *List1* or *List2* is a proper list at the time of call, `same_length/3` is determinate and terminates. Otherwise it will backtrack forever, binding its arguments to lists of length 0, 1, 2, and so on. `same_length/3` is logically equivalent to

```
length(List1, Length),
length(List2, Length)
```

except that if *List2* is known and the other arguments are not, this code will not terminate if backtracked into, while `same_length/3` will terminate determinately.

`select(?X, ?Xlist, ?Y, ?Ylist)`
: is true when *X* is the *K*th element of *Xlist* and *Y* the *K*th element of *Ylist* for some *K*, and apart from that element *Xlist* and *Ylist* are the same. You can use `select/4` to replace *X* by *Y* or vice versa. Either *Xlist* or *Ylist* should be a proper list.

`selectchk(?X, ?Xlist, ?Y, ?Ylist)`
: is to `select/4` what `memberchk/2` is to `member/2` in `library(basics)`.

`shorter_list(?Short, ?Long)`
: is true when *Short* is a list strictly shorter than *Long*. No relation between the elements of *Short* and *Long* is implied. *Long* does not have to be a proper list provided it has one more element than *Short*. This can be used to generate lists shorter than *Long*; lengths 0, 1, 2, and so on will be tried, but backtracking will terminate with a list that is one element shorter than *Long*. `shorter_list/2` cannot be used to generate lists longer than *Short*, because it does not look at all the elements of the longer list.

`subseq(?Sequence, ?SubSequence, ?Complement)`
: is true when *SubSequence* and *Complement* are both subsequences of the list *Sequence* (the order of corresponding elements being preserved) and every element of *Sequence* that is not in *SubSequence* is in *Complement* and vice versa. Among other things, this means that

```
length(Sequence) = length(SubSequence) +
                   length(Complement)
```

and

```
subseq([1,2,3,4], [1,3,4], [2]).
```

`subseq/3` was written to generate subsets and their complements together from *Sequence*, but can also be used to interleave two lists *Subsequence* and *Complement* in all possible ways. Either *Sequence* should be a proper list, or both *SubSequence* and *Complement* should both be proper lists. Note that if *S1* is a subset of *S2*, it will be generated *before S2* as a *SubSequence* and *after* it as a *Complement*. To be specific, take *S1* = `[a]`, *S2* = `[a,c]`, *Sequence* = `[a,b,c]`:

```
| ?- subseq([a,b,c], Sub, Com).

Sub = [],
Com = [a,b,c] ;

Sub = [c],
Com = [a,b] ;

Sub = [b],
Com = [a,c] ;       % S2 generated as Complement

Sub = [b,c],
Com = [a] ;         % S1 generated as Complement
                     % (AFTER S2)
Sub = [a],           % S1 generated as SubSequence
Com = [b,c] ;       % (BEFORE S2)

Sub = [a,c],         % S2 generated as SubSequence
Com = [b] ;

Sub = [a,b],
Com = [c] ;

Sub = [a,b,c],
Com = [] ;

no                % these 8 are all the solutions.
```

Further examples of the use of `subseq/3` are:

```
| ?- subseq([1,2,3,4], X, [2]).

X = [1,3,4]

| ?- subseq([a,b], Subs, Comp).

Subs = [],
Comp = [a,b] ;

Subs = [b],
Comp = [a] ;

Subs = [a],
Comp = [b] ;

Subs = [a,b],
Comp = [] ;

no
| ?- subseq(Seq, [@], [#]).

Seq = [#,@] ;

Seq = [@,#] ;

no
```

subseq0(+*Sequence*, *?SubSequence*)

is true when *SubSequence* is a subsequence of *Sequence*, but may be *Sequence* itself. Thus:

```
| ?- subseq0([a,b], [a,b]).

yes
| ?- subseq0([a,b], [a]).

yes
```

Also,

```
| ?- setof(X, subseq0([a,b,c],X), Xs).

Xs = [[],[a],[a,b],[a,b,c],[a,c],[b],[b,c],[c]]

| ?- bagof(X, subseq0([a,b,c,d],X), Xs).

Xs = [[a,b,c,d],[b,c,d],[c,d],[d],[],[c],[b,d],
[b],[b,c],[a,c,d],[a,d],[a],[a,c],[a,b,d],[a,b],
[a,b,c]]
```

Sequence must be a proper list.

`subseq1(+Sequence, ?SubSequence)`
: is true when *SubSequence* is a proper subsequence of *Sequence*; that is, *SubSequence* contains at least one element less than *Sequence*. *Sequence* must be a proper list.

```
| ?- % note that [a,b,c] does NOT appear in Xs:
|    setof(X, subseq1([a,b,c],X), Xs).

Xs = [[],[a],[a,b],[a,c],[b],[b,c],[c]]

| ?- % note that [a,b,c,d] does NOT appear in Xs:
|    bagof(X, subseq1([a,b,c,d],X), Xs).

Xs = [[b,c,d],[c,d],[d],[],[c],[b,d],[b],[b,c],
[a,c,d],[a,d],[a],[a,c],[a,b,d],[a,b],[a,b,c]]
```

`sumlist(+Numbers, ?Total)`
: is true when *Numbers* is a proper list of numbers, and *Total* is their sum. Note that a list of arithmetic expressions will *not* work. If any of the *Numbers* is a floating-point number, *Total* will be a floating-point number; otherwise it will be an integer.

`transpose(?X, ?Y)`
: is true when X is a list of the form [[*X11*,. . .,*X1m*],. . .,[*Xn1*,. . .,*Xnm*]] and Y is its transpose, that is, Y = [[*X11*,. . .,*Xn1*],. . .,[*X1m*,. . .,*Xnm*]].

  To make the `transpose/2` invertible, all the sublists of the list being transposed must be of the same length. If they are not, it will fail.

12.2.6 Lists as Sets

12.2.6.1 Set Processing — `library(sets)`

The `library(sets)` package represents sets as lists with no repeated elements. Some of the predicates provided by this package may return sensible answers if given arguments that contain repeated elements, but that is a lucky accident. When in doubt, use `list_to_set/2` to convert from a list (with possibly repeated elements) to a set. For a list of predicates related to set manipulation that are not in the `library(sets)` package, see Section 12.2.6.2 [lib-lis-set-pre], page 546. For some applications, *ordered* sets are more appropriate; see Section 12.2.7 [lib-lis-ordsets], page 547 for more information.

The predicates defined in `library(sets)` are described below:

`add_element(+Elem, +Set1, -Set2)`
: is true when *Set1* and *Set2* are sets represented as unordered lists, and *Set2* = *Set1* `U` {*Elem*}. `add_element/3` may only be used to calculate *Set2* given *Elem*

and *Set1*. However, it is permissible for *Set1* to be a list with a variable at the end; in this case, `add_element/3` will add new elements to the end of *Set2*.

`del_element(+Elem, +Set1, -Set2)`
: is true when *Set1* and *Set2* are sets represented as unordered lists, and *Set2* = *Set1* \ {*Elem*}. `del_element/3` may only be used to calculate *Set2* given *Elem* and *Set1*. If *Set1* does not contain *Elem*, *Set1* and *Set2* will be equal. If *Set1* contains more than one copy of *Elem* (in which case *Set1* is not really a set representation), only the first copy of *Elem* will be removed. See `delete/3` in `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533) for a predicate that removes all copies of a given element. When *Set1* and *Set2* are identical, there are infinitely many *Elem*s that would make this predicate true, so we could not solve for *Elem*. Therefore, we do not attempt to solve for *Elem* in *any* case, which is why it is a '+' argument.

`is_set(+Set)`
: is true when *Set* is a proper list that contains no repeated elements (that is, a proper set). `is_set/1` does not check for any particular order. If *Set* is not a proper list, `is_set/1` fails.

`disjoint(+Set1, +Set2)`
: is true when *Set1* and *Set2* have no elements in common. `disjoint/2` is the opposite of `intersect/2` (below).

`select(?Element, ?Set, ?Residue)`
: is true when *Set* is a list, *Element* occurs in *Set*, and *Residue* is everything in *Set* except *Element* (the order of elements is preserved). To ensure termination, either *Set* or *Residue* should be proper. `select/3` works on lists as well as on sets.

  `select/3` is closely related to the predicate `select/4` in `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533). Although `select/3` is normally used to solve for *Element* and *Residue*, you can read '`select(X, S, Y, R)`' as dq"replace *X* by *Y* in *S* giving *R*", and '`select(X, S, R)`' can be read as "replace *X* by *nothing* in *S* giving *R*".

```
| ?- select(a, [a,r,a], R).

R = [r,a] ;

R = [a,r] ;

no

| ?- select(a, [a,r,a], e, R).

R = [e,r,a] ;

R = [a,r,e] ;

no
```

`selectchk(+Element, +Set, ?Residue)`
: is to `select/3` what `memberchk/2` is to `member/2` in `library(basics)`. That is, it locates the first occurrence of *Element* in *Set* and deletes it, returning the resulting list in *Residue*. It is steadfast in *Residue*.

`pairfrom(?Set, ?Element1, ?Element2, ?Residue)`
: is true when *Set* is a set, *Element1* occurs in *Set*, *Element2* occurs in *Set* after *Element1*, and *Residue* is everything in *Set* except *Element1* and *Element2*. The point of `pairfrom/4` is to select pairs of elements from a set without selecting the same pair twice in different orders. To ensure termination, either *Set* or *Residue* should be proper. `pairfrom/4` works on lists as well as on sets.

`intersect(+Set1, +Set2)`
: is true when *Set1* and *Set2* have a member in common. It assumes that both sets are known, and that you do not need to know which element it is that they share.

`intersect/3`
: is an obsolete predicate and should not be used in new programs.

`subset(+SubSet, +Set)`
: is true when each member of *SubSet* occurs in *Set*. `subset/2` can only be used to test two given sets; it cannot be used to generate subsets.

  To generate subsets, use `subseq0/[2,3]` or `subseq1/[2,3]` from `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533); they will generate each subset (or each proper subset) (and, for the three-argument versions, its complement) precisely once, but cannot be used for testing whether a given set is a subset of another. Note that they generate sub-*sequences*; to really generate sub-*sets* they would have to enumerate all the permutations of each subsequence, which would be quite costly.

`seteq(+Set1, +Set2)`
: is true when *Set1* is a subset of *Set2*, and vice-versa. Since set representations should not contain duplicates, we could check whether one is a permutation of

the other. The method used by `seteq/2` works even if *Set1* and *Set2* do contain duplicates.

`list_to_set(+List, ?Set)`
: is true when *List* and *Set* are lists, and *Set* contains the same elements as *List* in the same order, except that *Set* contains no duplicates. *List* and *Set* are thus equal when considered as sets. `list_to_ord_set/2` is faster at converting a list to a set, but the method used by `list_to_set/2` preserves as much of the original ordering as possible.

`intersection(+Set1, +Set2, ?Intersection)`
: is true when *Intersection* is the intersection of *Set1* and *Set2*, taken in a particular order. In fact it is precisely the elements of *Set1* taken in their original order, with elements not in *Set2* deleted. If *Set1* contains duplicates, so may *Intersection*.

`intersection(+Sets, ?Intersection)`
: is true when *Sets* is a proper list of sets, and *Intersection* is the intersection of all the sets in *Sets*. In fact, *Intersection* is precisely the elements of the head of *Sets*, with elements that do not occur in all of the other sets dropped. *Sets* must not be empty.

`subtract(+Set1, +Set2, ?Difference)`
: is like `intersection/3`, but here it is the elements of *Set1* that *are* in *Set2* that are deleted.

`symdiff(+Set1, +Set2, ?Diff)`
: is true when *Diff* is the symmetric difference of *Set1* and *Set2*; that is, if each element of *Diff* occurs in one of *Set1* and *Set2*, but not both. The construction method is such that the answer will contain no duplicates even if *Set1* and *Set2* do.

`setproduct(+Set1, +Set2, ?CartesianProduct)`
: is true when *Set1* is a set (list) and *Set2* is a set (list) and *CartesianProduct* is a set of *Elt1*-*Elt2* pairs, with a pair for each element *Elt1* of *Set1* and *Elt2* of *Set2*. For example,

```
| ?- setproduct([b,a], [1,2], Product).

Product = [[b-1],[b-2],[a-1],[a-2]]
```

`union(+Set1, +Set2, ?Union)`
: is true when *Union* is the elements of *Set1* that do not occur in *Set2*, followed by all the elements of *Set2*, that is, when the following are true:

```
subtract(Set1, Set2, Diff)
append(Diff, Set2, Union.
```

`union(+Sets, ?Union)`
: is true when *Sets* is a list of sets and *Union* is the union of all the sets in *Sets*. *Sets* must be a proper list, but it may be empty.

`union(+Set1, +Set2, ?Union, ?Difference)`
: added to keep '`sets.pl`' and '`ordsets.pl`' parallel. This predicate is true when the following are true:

```
union(Set1, Set2, Union),
subtract(Set1, Set2, Difference).
```

`power_set(?Set, ?PowerSet)`
: is true when *Set* is a list and *PowerSet* is a list of all the subsets of *Set*. The elements of *PowerSet* are ordered so that if *A* and *B* are subsets of *Set* and *B* is a subset of *A* (for example, *Set*=[1,2,3], *A*=[1,3], *B*=[3]) then *A* will appear before *B* in *PowerSet*. Note that `length(PowerSet) = 2^length(Set)`, so this is only useful for a small *Set*.

```
| ?- power_set([a,b], X).
X = [[a,b],[a],[b],[]]
```

12.2.6.2 Predicates Related to Sets

The following predicates are relevant to sets, but are not in `library(sets)`:

`length(-List, +Integer)`
: built-in predicate: do not use this if *Set* might contain duplicates. See Section 8.15 [ref-all], page 295 for more information.

`append(+*List1, +*List2, +*List3)`
: built-in predicate: only use `append/3` this way when *Set1* and *Set2* are known to be disjoint, and put a comment in your code explaining the hack.

`member(?Elem, ?Set)`
: in `library(basics)` (Section 12.2.4.2 [lib-lis-basics-member], page 530)

`memberchk(?Elem, ?Set)`
: in `library(basics)` (Section 12.2.4.3 [lib-lis-basics-memberchk], page 532)

`subseq0(+Set, ?SubSet)`
: in `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533): you can only use `subseq0/2` to generate subsets of a given *Set*, not to test whether a given *SubSet* is a subset of a given *Set*, because `subseq0/2` preserves the order of the elements, which is irrelevant to sets. However, you *can* use it to generate subsets of an ordered set, as the order of the elements *does* matter there.

`subseq1(+Set, ?ProperSubSet)`
: in `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533): you can only use `subseq1/2` to generate proper subsets of a given *Set*, not to test whether a given *SubSet* is a proper subset of a given *Set*, because `subseq1/2` preserves the order of the elements, which is irrelevant to sets. However, you *can* use it to generate proper subsets of an ordered set, as the order of the elements *does* matter there.

12.2.7 Lists as Ordered Sets — `library(ordsets)`

In this group of predicates, sets are represented by ordered lists with no duplicates. Thus {c,r,a,f,t} would be `[a,c,f,r,t]`. The ordering is defined by the `@<` family of term comparison predicates, and is the ordering used by the built-in predicates `sort/2` and `setof/3`. Note that `sort/2` and `setof/3` produce ordered sets as their results. See Section 8.15 [ref-all], page 295 for more information.

The benefit of the ordered representation is that the elementary set operations can be done in time proportional to the *sum* of the argument sizes rather than their *product*.

A number of predicates described elsewhere can be used on unordered sets. Examples are `length/2` (built-in; see Section 8.9 [ref-lte], page 238), `member/2` (from `library(basics)`; see Section 12.2.4.2 [lib-lis-basics-member], page 530), `subseq1/2` (from `library(lists)`; see Section 12.2.5 [lib-lis-lists], page 533), `select/3` (from `library(sets)`; see Section 12.2.6.1 [lib-lis-set-sets], page 542), and `sublist/3` (from `library(maplist)`; see Section 12.13 [lib-abs], page 641).

`is_ordset(+List)`
: is true when *List* is a proper list of terms [T1,T2, ...,Tn] and the terms are strictly increasing: `T1 @< T2 @< ... @< Tn`. The output of `sort/2` and `setof/3` always satisfies this test. Anything that satisfies this test can be given to the predicates in `library(ordsets)`, regardless of how it was generated.

`list_to_ord_set(+List, ?Set)`
: is true when *Set* is the ordered representation of the set designated by the unordered representation *List*. (This is in fact no more than an alias for `sort/2`.)

`ord_add_element(+Set1, +Element, ?Set2)`
: calculates *Set2* = *Set1* `U {`*Element*`}`. It only works this way around. `ord_add_element/3` is the ordered equivalent of `add_element/3` (Section 12.2.6.1 [lib-lis-set-sets], page 542).

`ord_del_element(+Set1, +Element, ?Set2)`
: calculates *Set2* = *Set1* `\ {`*Element*`}`. It only works this way around. `ord_del_element/3` is the ordered equivalent of `del_element/3` (Section 12.2.6.1 [lib-lis-set-sets], page 542).

`ord_disjoint(+Set1, +Set2)`
: is true when *Set1* and *Set2* have no element in common. It is not defined for unsorted lists.

`ord_intersect(+Set1, +Set2)`
: is true when *Set1* and *Set2* have at least one element in common. Note that the test is `==` rather than `=` .

`ord_intersect(+Set1, +Set2, ?Intersection)`
: is an obsolete synonym for `ord_intersection/3`. It should not be used in new programs.

`ord_intersection(+Set1, +Set2, ?Intersection)`
: is true when *Intersection* is the ordered representation of the intersection of *Set1* and *Set2*, provided that *Set1* and *Set2* are ordered sets.

`ord_intersection(+Sets, ?Intersection)`
: is true when *Intersection* is the ordered representation of the intersection of all the sets in *Sets* (which must be a non-empty proper list of ordered sets).

`ord_seteq(+Set1, +Set2)`
: is true when *Set1* and *Set2* represent the same set. Since they are assumed to be ordered representations, *Set1* and *Set2* must be identical.

`ord_setproduct(+Set1, +Set2, ?Product)`
: is true when *Product* is a sorted list of *Elt1-Elt2* pairs, with a pair for each element *Elt1* of *Set1* and each element *Elt2* of *Set2*. *Set1* and *Set2* are assumed to be ordered; if they are not, the result may not be.

```
| ?- list_to_ord_set([t,o,y], Set1),
|    list_to_ord_set([d,o,g], Set2),
|    ord_setproduct(Set1, Set2, Product).

Set1 = [o,t,y],
Set2 = [d,g,o],
Product = [o-d,o-g,o-o,t-d,t-g,t-o,y-d,y-g,y-o]

| ?- % but with unordered arguments:
|    ord_setproduct([t,o,y], [d,o,g], Product).

Product = [t-d,t-o,t-g,o-d,o-o,o-g,y-d,y-o,y-g]
```

`ord_subset(+Set1, +Set2)`
: is true when every element of the ordered set *Set1* appears in the ordered set *Set2*. To generate subsets, use a member of the `subseq0/2` family from `library(lists)` (Section 12.2.5 [lib-lis-lists], page 533).

`ord_subtract(+Set1, +Set2, ?Difference)`
: is true when *Difference* contains all and only the elements of *Set1* that are not also in *Set2*.

`ord_symdiff(+Set1, +Set2, ?Difference)`
: is true when *Difference* is the symmetric difference of *Set1* and *Set2*.

`ord_union(+Set1, +Set2, ?Union)`
: is true when *Union* is the union of *Set1* and *Set2*. Note that when an element occurs in both *Set1* and *Set2*, only one copy is retained.

`ord_union(+Sets, ?Union)`
: is true when *Union* is the ordered representation of the union of all the sets in *Sets* (which must be a proper list of ordered sets). This is quite efficient. In fact `ord_union/2` can be seen as a generalization of `sort/2`.

`ord_union(+OldSet, +NewSet, ?Union, ?ReallyNew)`
: is true when *Union* is *NewSet* U *OldSet*, and *ReallyNew* is *NewSet* \ *OldSet*. This is useful when you have an iterative problem, and you're adding some

possibly new elements (*NewSet*) to a set (*OldSet*), and as well as getting the updated set (*Union*) you would like to know which if any of the "new" elements didn't already occur in the set (*ReallyNew*).

If operations on ordered sets or ordered lists are useful to you, you may also find `library(ordered)` (Section 12.13 [lib-abs], page 641) or `library(ordprefix)` (Section 12.13 [lib-abs], page 641) of interest.

12.2.8 Parts of lists — `library(listparts)`

`library(listparts)` exists to establish a common vocabulary for names of parts of lists among Prolog programmers. You will seldom have occasion to use `head/2` or `tail/2` in your programs — pattern matching is clearer and faster — but you will often use these words when talking about your programs. The predicates provided are

`cons(?Head, ?Tail, ?List)`
: *Head* is the head of *List* and *Tail* is its tail; i.e. `append([Head, Tail, List)]`. No restrictions.

`last(?Fore, ?Last, ?List)`
: *Last* is the last element of *List* and *Fore* is the list of preceding elements, e.g. `append(Fore, [Last, List)]`. *Fore* or *Last* should be proper. It is expected that *List* will be proper and *Fore* unbound, but it will work in reverse too.

The remaining predicates are binary, and `part(Whole, Part)` is to be read as "*Part* is the/a *part* of *Whole*". When both `part/2` and `proper_part/2` exist, proper *part*s are strictly smaller than *Whole*, whereas *Whole* may be a *part* of itself. *N* is the length of the whole argument, assumed to be a proper list. This order is strictly in accord with the fundamental principle of argument ordering in Prolog: INPUTS BEFORE OUTPUTS.

`head(List, Head)`
: *List* is a non-empty list and *Head* is its head. A list has only one head. No restrictions.

`tail(List, Tail)`
: *List* is a non-empty list and *Tail* is its tail. A list has only one tail. No restrictions.

`prefix(List, Prefix)`
: *List* and *Prefix* are lists and *Prefix* is a proper prefix of *List*.

`proper_prefix(List, Prefix)`
: *List* and *Prefix* are lists and *Prefix* is a proper prefix of *List*. That is, *Prefix* is a prefix of *List* but is not *List* itself. It terminates if either argument is proper, and has at most *N* solutions. Prefixes are enumerated in ascending order of length.

`suffix(`*List*`, `*Suffix*`)`
: *List* and *Suffix* are lists and *Suffix* is a suffix of *List*. It terminates only if *List* is proper, and has at most *N*+1 solutions. Suffixes are enumerated in descending order of length.

`proper_suffix(`*List*`, `*Suffix*`)`
: *List* and *Suffix* are lists and *Suffix* is a proper suffix of *List*. That is, *Suffix* is a suffix of *List*. It terminates only if *List* is proper, and has at most *N*+1 solutions. Suffixes are enumerated in descending order of length.

`segment(`*List*`, `*Segment*`)`
: *List* and *Segment* are lists and *Segment* is a sublist of *List*.

`proper_segment(`*List*`, `*Segment*`)`
: *List* and *Segment* are lists and *Segment* is a proper sublist of *List*.

`sublist/2`
: same as `segment/2`

`proper_sublist/2`
: same as `proper_segment/2`

12.3 Term Manipulation

12.3.1 Introduction

There are two ways of looking at Prolog data structures. One is the proper "object-level" logical way, in which you think of arguments as values. The other is the "meta-logical" way, in which you see them not as lists or trees (or whatever your object-level data types are), but as "terms".

Prolog has the following built-in operations that operate on terms as such:

`functor(+`*Term*`, -`*Name*`, -`*Arity*`)`
: is true when *Term* is a term, and the principal function symbol of *Term* is *Name*, and the arity (number of arguments) of *Term* is *Arity*. Alternatively, you may think of this as being true when *Term* is a term and the principal functor of *Term* is *Name*/*Arity*. All constants, including numbers, are their own principal function symbols, so `functor(1.3, 1.3, 0)` is true. This may be used to find the functor of a given term, or to construct a term having a given functor.

`arg(+`*Argnum*`, +`*Term*`, -`*Arg*`)`
: is true when *Term* is a non-variable, *Argnum* is a positive integer, and *Arg* is the *Argnum*th argument of *Term*. Argument numbering starts at 1. This can only be used to find *Arg*; *Argnum* and *Term* must be given.

+-Term `=..` *+-List*
: is true when *Term* is a term, *List* is its principal function symbol and the list of the remaining arguments. Use of `=../2` can nearly always be avoided, and

should be whenever possible, as it is very slow and uses memory unnecessarily (see Section 2.5.7 [bas-eff-bdm], page 47).

`copy_term(+Term, -Copy)`
: unifies *Copy* with an alphabetic variant of *Term* that contains all new variables (see Section 12.3.8 [lib-tma-subsumes], page 562). That is, `copy_term/2` makes a copy of *Term* by replacing each distinct variable in *Term* by a new variable that occurs nowhere else in the system, and unifies *Copy* with the result.

`compare(-Order, +Term1, +Term2)`
: compares *Term1* and *Term2* with respect to *Order*, which may be one of `<`, `>`, or `=`. If *Order* is `=`, the comparison is actually done with respect to the `==/2` operator on terms.

The system also includes the term comparison predicates `==/2`, `\==/2`, `@</2`, `@>/2`, `@>/2`, `@=</2`. See Section 8.9 [ref-lte], page 238 for more details.

12.3.2 The Six Term Manipulation Packages

There are currently six library packages that extend Prolog's built-in set of operations on terms. They are

`library(arg)`
: some generalizations of `arg/3`

`library(changearg)`
: some operations for building new terms

`library(occurs)`
: testing whether a given term does or does not contain another term or variable

`library(samefunctor)`
: some generalizations of `functor/3`

`library(subsumes)`
: testing whether one term subsumes another

`library(unify)`
: sound unification

12.3.3 Finding a Term's Arguments — `library(arg)`

`library(arg)` defines seven predicates, all of which are generalizations of the built-in predicate `arg/3`.

`arg(+ArgNum, +Term, -Arg)`
: unifies *Arg* with the *ArgNum*th argument of *Term*. *Term* must not be a variable, but any other kind of term is acceptable. Even a number is acceptable as *Term*;

numbers are simply terms that happen to have no arguments. *ArgNum* must be instantiated to an integer. If *ArgNum* is less than 1 or greater than the number of arguments of *Term*, `arg/3` signals an error. Basically, `arg/3` pretends to be the infinite table

```
arg(1, a(X), X).
arg(1, a(X,_), X).
arg(2, a(_,X), X).
...
arg(5, zebra_finch(_,_,_,_,X,_,_,_), X).
...
```

except that it can only be used to find the *Arg* for a given *Index* and *Term*, and cannot find the *Index*. `arg/3` is a built-in predicate, and is described in the reference pages, not actually defined in `library(arg)`.

`arg0(+`*`Index`*`, +`*`Term`*`, ?`*`Arg`*`)`

unifies *Arg* with the *Index*th argument of *Term* if *`Index`* `> 0`, or with the principal function symbol of *Term* if *`Index`* `= 0`. This predicate is supplied because some other Prolog implementations have made `arg/3` do this, and this makes it easier to convert code originally written for those systems. The one reason you might use `arg0/3` is that it reports errors, while `arg/3`, for backwards compatibility with DEC-10 Prolog, does not. Examples:

```
| ?- arg0(2, f(o,x,y), X).

X = x

| ?- arg0(0, f(o,x,y), X).

X = f

| ?- arg0(N, f(o,x,y), X).
! Instantiation error in argument 1 of arg0/3
! goal:  arg0(_732,f(o,x,y),_767)

| ?- arg0(y, f(o,x,y), N).
! Type error in argument 1 of arg0/3
! integer expected, but y found
! goal:  arg0(y,f(o,x,y),_764)
```

`genarg(?`*`Index`*`, +`*`Term`*`, ?`*`Arg`*`)`

is a version of `arg/3` that is able to solve for *Index* as well as for *Arg*.

```
| ?- arg(N, f(a,b), X).

no

| ?- genarg(N, f(a,b), X).

N = 2,
X = b ;

N = 1,
X = a ;

no
| ?- genarg(N, f(1,b,2), X), atom(X).

N = 2,
X = b ;

no

| ?- genarg(3, f(1,b,2), X).

X = 2
```

If *Index* is instantiated, `genarg/3` generates the same result as `arg/3`. If *Index* is uninstantiated, `genarg/3` picks out each argument in turn. The order in which the arguments are tried is *not* defined; the current implementation works from right to left, but this order should *not* be relied upon.

`genarg0(?Index, +Term, ?Arg)`
: is a version of `arg0/3` that is able to solve for *Index* as well as *Arg*.

`args(?Index, +Terms, ?Args)`
: is true when *Terms* and *Args* are lists of the same length, each element of *Terms* is instantiated to a term having at least *Index* arguments, and `arg(Index, Term, Arg)` is true for each pair `<Term, Arg>` of corresponding elements of `<Terms, Args>`. *Index* is strictly positive, and only arguments are found, not principal function symbols. This is a generalization of `genarg/3`. For example,

```
| ?- args(1, [a+b,c-d,e*f,g/h], X).

X = [a,c,e,g]

| ?- args(2, [a+A,c-B,e*C,g/D], [b,d,f,h]).

A = b,
B = d,
C = f,
D = h
```

```
| ?- args(I, [1-a,2-b,3-c,4-d], X).

I = 2,
X = [a,b,c,d] ;

I = 1,
X = [1,2,3,4]
```

`args0(?Index, +Terms, ?Args)`
is like `args/3` except that *Index* = `0` selects the principal function symbol.

```
| ?- args0(0, [a+b,c-d,e*f,g/h,27], X).

X = [+,-,*,/,27]

| ?- args0(I, [1-a,2-b,3-c,4-d], X).

I = 2,
X = [a,b,c,d] ;

I = 1,
X = [1,2,3,4] ;

I = 0,
X = [-,-,-,-]
```

This is a generalization of `genarg0/3`.

`project(+Terms, ?Index, ?Args)`
is identical to `args0/3` except for the argument order. The argument order of `project/3` is not consistent with anything else in the library. This predicate is retained for backwards compatibility. Use `args0/3` instead in new programs.

`path_arg(?Path, +Term, ?SubTerm)`
unifies *SubTerm* with the subterm of *Term* found by following *Path*, where *Path* is a sequence of positive integers. For example, the goal

```
path_arg([I,J], MyTerm, MySubTerm)
```

unifies `MySubTerm` with the `J`'th argument of the `I`'th argument of `MyTerm`. In general, *Term* should be ground. `path_arg/3` may be regarded as a generalization of `genarg/3`. It can be used to find the *SubTerm* and a known *Path*, or to find a *Path* to a known *SubTerm*. It could have been defined as

```
path_arg([], Term, Term).
path_arg([Index|Indices], Term, SubTerm) :-
        genarg(Index, Term, Arg),
        path_arg(Indices, Arg, SubTerm).
```

The actual library program is rather more complicated because it contains error-reporting code. Examples of its use include:

```
/* Here is a sample table of all the subterms of
/* the quadratic formula "(a*x^2) + (b*x) + c = 0"
/*
[]          a*x^2+b*x+c=0
[1]         a*x^2+b*x+c
[1,1]       a*x^2+b*x
[1,1,1]     a*x^2
[1,1,1,1]   a
[1,1,1,2]     x^2
[1,1,1,2,1]   x
[1,1,1,2,2]     2
[1,1,2]           b*x
[1,1,2,1]         b
[1,1,2,2]           x
[1,2]                 c
[2]                     0
*/

| ?- path_arg([1,1,2,2],   a*x^2+b*x+c=0, X).

X = x                                 ^

| ?- path_arg([1,1,1,2,2], a*x^2+b*x+c=0, X).

X = 2                             ^

| ?- path_arg(Path,         a*x^2+b*x+c=0, b).

Path = [1,1,2,1]                    ^
```

This notation for locating subtrees of a tree is widely used throughout computer science.

Note that except for `project/3`, which is included only in the interests of backwards compatibility, all of these predicates have the same pattern of arguments:

- first *Index* (or its equivalent, *Path*)
- then *Term* (or *Terms*)
- and finally *Arg* (or *Args*)

For consistency, we recommend that you use this argument order for "selector" predicates generally: first the argument or arguments that constitute the selector or index, then the thing or things that are being selected from, and finally the result or results.

12.3.4 Altering Term Arguments — `library(changearg)`

The predicates in `library(changearg)` allow you to construct a new term that is identical to an old term except that one of its elements has been replaced or two of its elements have been swapped. Using these operations, you could use terms as one-dimensional arrays; however, though the elements of such arrays can be accessed in *O(1)* time using `arg/3`, changing an element takes *O(N)* time, where *N* is the arity of the term. See `library(logarr)` for a more efficient way of implementing arrays in Prolog.

Why then are these operations provided? To aid in the construction of term-rewriting systems. For example, suppose you have a set of rewrite rules expressed as a table

```
rewrite_rule(X*0, 0).
rewrite_rule(X*1, X).
rewrite_rule(K*X, X*K) :- integer(K).
rewrite_rule(X*(Y*Z), (X*Y)*Z).
   .
   .
   .
```

which you want exhaustively applied to a term. You could write

```
waterfall(Expr, Final) :-
        path_arg(Path, Expr, Lhs),
        rewrite_rule(Lhs, Rhs),
        change_path_arg(Path, Expr, Modified, Rhs),
        !,
        waterfall(Modified, Final).
waterfall(Expr, Expr).
```

Then

```
| ?- waterfall((a*b)*(c*0)*d, X).

X = 0

| ?- waterfall((1*a)*(2*b), X).

X = a*2*b
```

The predicates supplied by `library(changearg)` are as follows:

`change_arg(+`*Index*`, ?`*OldTerm*`, ?`*OldArg*`, ?`*NewTerm*`, ?`*NewArg*`)`
: is true when *OldTerm* and *NewTerm* are identical except that the *Index*th argument of *OldTerm* is *OldArg* and the *Index*th argument of *NewTerm* is *NewArg*. Either *OldTerm* or *NewTerm* should be supplied; the other term can then be found. `change_arg/5` is actually quite symmetric:

```
change_arg(K, O, X, N, Y)
```

and

```
change_arg(K, N, Y, O, X)
```

have exactly the same effect. For example:

```
| ?- change_arg(1, c(o,l,t), X, N, u).

X = o,
N = c(u,l,t)

| ?- change_arg(1, N, u, c(o,l,t), X).

N = c(u,l,t),
X = o
| ?- change_arg(3, SALE, E, s(a,l,t), T).

SALE = s(a,l,E),
E = _755,
T = t

| ?- change_arg(3, a+b, b, X, c).

no
```

`change_arg(+Index, ?OldTerm, ?NewTerm, ?NewArg)`
: is identical to `change_arg/5` except that the *OldArg* argument is omitted. **Please note:** this argument order may be surprising if you think about this predicate on its own; however, it makes sense in the context of the entire group.

`change_arg0/[4,5]`
: like `change_arg/[4,5]` except that *Index*=0 is allowed, in which case the principal function symbol is changed. Do not use this in new programs; use `change_arg/5` or `change_functor/5` directly. The order in which values for *Index* are enumerated is *not defined*.

`change_functor(?OldTerm, ?OldSymbol, ?NewTerm, ?NewSymbol, ?Arity)`
: is true when *OldTerm* and *NewTerm* are identical terms, except that the functor of *OldTerm* is *OldSymbol*/*Arity*, and the functor of *NewTerm* is *NewSymbol*/*Arity*. This is similar to `same_functor/3` in some respects (Section 12.3.7 [lib-tma-samefunctor], page 561), such as the fact that any of the arguments can be solved for. If *OldTerm* and *NewSymbol* are instantiated, or *NewTerm* and *OldSymbol* are instantiated, or *NewSymbol*, *OldSymbol*, and *Arity* are instantiated, that is enough information to proceed. Note that *OldSymbol* or *NewSymbol* may be a number, in which case *Arity* must be 0.

`swap_args(+Index1, +Index2, ?OldTerm, ?Arg1, ?NewTerm, ?Arg2)`
: is true when *OldTerm* and *NewTerm* are identical except that

```
                        at Index1       at Index2
in OldTerm                 Arg1            Arg2
```

```
in NewTerm          Arg2            Arg1
```

that is, the arguments at *Index1* and *Index2* have been swapped. As with `change_arg/5`, `swap_args/6` is symmetric; the following terms have exactly the same effect.

```
swap_args(I, J, O, X, N, Y)

swap_args(I, J, N, Y, O, X)
```

For example:

```
| ?- swap_args(1, 4, f(X,e,a,Y,e,r), r, T, d).

X = r,
Y = d,
T = f(d,e,a,r,e,r)
```

`swap_args(+`*Index1*`, +`*Index2*`, ?`*OldTerm*`, ?`*NewTerm*`)`
is identical to `swap_args/6` except that the *Arg1* and *Arg2* arguments are omitted.

```
| ?- swap_args(1, 4, f(r,e,a,d), X).

X = f(d,e,a,r)
```

`change_path_arg(+`*Path*`, ?`*OldTerm*`, ?`*OldSub*`, ?`*NewTerm*`, ?`*NewSub*`)`
is true when *OldTerm* and *NewTerm* are identical terms except that

```
path_arg(Path, OldTerm, OldSub),
path_arg(Path, NewTerm, NewSub)
```

That is, the subterm of *OldTerm* at *Path* was *OldSub* and is replaced by *NewSub* in *NewTerm*, and there are no other differences between *OldTerm* and *NewTerm*. This is to `change_arg/5` as `path_arg/3` is to `arg/3`.

`change_path_arg(+`*Path*`, ?`*OldTerm*`, ?`*NewTerm*`, ?`*NewSub*`)`
is identical to `change_path_arg/5` except that the *OldSub* argument is omitted.

```
| ?- OldTerm = this*is+an*example,
|    path_arg(Path, OldTerm, this),
|    change_path_arg(Path, OldTerm, NewTerm, it).

OldTerm = this*is+an*example,
Path = [1,1],
NewTerm = it*is+an*example
```

12.3.5 Checking Terms for Subterms — `library(occurs)`

The predicates in `library(occurs)` test whether a given term is a subterm of another or not. We define a subterm thus:

- T is a subterm of T
- S is a subterm of T if A is an argument of T and S is a subterm of A

A *proper* subterm of a term T would be any subterm of T other than T itself. There are no library predicates concerned with the "proper subterm" relationship, only the "subterm" relationship.

There are two questions we might ask:

- does S unify with (is it '=' to) some subterm of T? The predicates that ask this question have '`_term`' in their names.
- is S identical to (is it '==' to) some subterm of T? The predicates that ask this question have '`_var`' in their names.

When the predicates are applied to ground terms, both questions have the same answers.

Seven predicates are defined by `library(occurs)`:

`contains_term(+SubTerm, +Term)`
: is true when *Term* contains a subterm that unifies with ('=') *SubTerm*.

`contains_var(+SubTerm, +Term)`
: is true when *Term* contains a subterm that is identical to ('==') *SubTerm*. The reason for the name is that this predicate is normally used to check whether *Term* contains a particular *variable* *SubTerm*. But `contains_var/2` makes sense even when *SubTerm* is not a variable. In fact, if *Term* and *SubTerm* are both ground, `contains_term/2` and `contains_var/2` are the same test.

`free_of_term(+SubTerm, +Term)`
: is true when *Term* does *not* contain a subterm that unifies with ('=') *SubTerm*.

`free_of_var(+SubTerm, +Term)`
: is true when *Term* does *not* contain a subterm that is identical to ('==') *SubTerm*. This is the "occur check", which is needed for sound unification: a variable X should unify with a non-variable term T only if `free_of_var(X, T)`. See `library(unify)` (Section 12.3.9 [lib-tma-unify], page 562) for an example of the use of this predicate.

`occurrences_of_term(+SubTerm, +Term, ?Tally)`
: unifies *Tally* with the number of subterms of *Term* that unify with ('=') *SubTerm*.

`occurrences_of_var(+SubTerm, +Term, ?Tally)`
: unifies *Tally* with the number of subterms of *Term* that are identical to ('==') *SubTerm*.

`sub_term(-SubTerm, +Term)`
: enumerates the *SubTerm*s of *Term*. The order in that the subterms are enumerated is not fully defined, though each subterm will be reported before any of its own subterms. Be careful: terms tend to have *lots* of subterms.

```
| ?- sub_term(X, (a+b)*(c+d)), tab(8),
write(X), nl, fail.

        (a+b)*(c+d)
        c+d
        d
        c
        a+b
        b
        a
no
```

The order in which these terms are generated is subject to change, and should not be relied upon.

12.3.6 Note on Argument Order

All the predicates in `library(occurs)` have the same argument pattern:

```
{prefix}_{term|var}(SubTerm, Term[, Extra])
```

where *Extra* includes any other arguments. Why does the *SubTerm* argument appear first? The answer involves `library(call)` and `library(maplist)`. Here are some of the things you can do with the arguments this way around:

```
terms_containing_term(SubTerm, Terms, Selected) :-
        include(contains_term(SubTerm), Terms, Selected).
    %   if member(Term, Terms), then Term will be included in
    %   Selected iff contains_term(SubTerm, Term) succeeds.

terms_free_of_var(SubTerm, Terms, Selected) :-
        exclude(contains_var(SubTerm), Terms, Selected).
    %   if member(Term, Terms), then Term will be included in
    %   Selected iff contains_var(SubTerm, Term) fails.

tallies_of_term(SubTerm, Terms, Tallies) :-
        maplist(occurrences_of_term(SubTerm), Terms, Tallies).
    %   if nth1(N, Terms, Term), then nth1(N, Tally, Tallies)
    %   where occurrences_of_term(SubTerm, Term, Tally).
```

The same argument order is used for `sub_term/2` even though it is not used in this way, in order to preserve consistency.

12.3.7 Checking Functors — `library(samefunctor)`

This library package is supplied to solve the following problem: often you could write code that works more than one way around except that this requires calling `functor/3` twice, and one of the calls must therefore precede the other. For example,

```
reverse_terms(Term1, Term2) :-
        functor(Term1, F, N),   % ***
        functor(Term2, F, N),   % ***
        reverse_terms(N, 1, Term1, Term2).

reverse_terms(0, _, _, _) :- !.
reverse_terms(Z, A, Term1, Term2) :-
        arg(Z, Term1, Arg),
        arg(A, Term2, Arg),
        Y is Z-1, B is A+1,
        reverse_terms(Y, B, Term1, Term2).
```

As written, this can only be used to find *Term2* given *Term1*. If you swapped the two *** lines, you could find *Term1* given *Term2*, but then could not find *Term2* given *Term1*. You can make a bidirectional version of `reverse_terms/2` by using the predicate `same_functor/3` in place of the *** lines:

```
reverse_terms(Term1, Term2) :-
        same_functor(Term1, Term2, N),  % ***
        reverse_terms(N, 1, Term1, Term2).
```

`library(samefunctor)` defines the following predicates:

`same_functor(?Term1, ?Term2, ?Symbol, ?Arity)`
: is true when *Term1* and *Term2* have the same principal functor, the function symbol being *Symbol* and the arity being *Arity*. In other words,

  `same_functor(T1, T2, F, N)` if
  `functor(T1, F, N)` and
  `functor(T2, F, N)` are both true.

  Either *Term1*, or *Term2*, or *both Symbol* and *Arity* should be instantiated. This is the most general variant.

`same_functor(?Term1, ?Term2, ?Arity)`
: is true when *Term1* and *Term2* have the same principal functor, and *Arity* is their common arity. Either *Term1* or *Term2* should be instantiated, and the other arguments can then be found. Often, `same_functor/3` is appropriate and the greater generality of `same_functor/4` is not needed.

`same_functor(?Term1, ?Term2)`
: is true when *Term1* and *Term2* have the same principal functor. Either *Term1* or *Term2* should be instantiated, and the other argument can then be found.

Note that `same_functor/4` has the same argument order as `functor/3` except that it has *two* "term" arguments at the front. Whenever a predicate's arguments include a functor specification expressed as two arguments, the function symbol and its arity, those two arguments should always be adjacent, with the function symbol first and the arity immediately following. `functor/3` and `same_functor/4` obey this rule.

12.3.8 Term Subsumption — `library(subsumes)`

Term subsumption is a sort of one-way unification. Recall that terms S and T unify if they have a common instance, and that unification in Prolog instantiates both terms to that common instance. S subsumes T if T is already an instance of S. For our purposes, T is an *instance* of S if there is a substitution that leaves T unchanged and makes S identical to T.

Two terms are *alphabetic variants* if they are identical except for variable names, and all common variables appear in corresponding positions. For example,

`f(X,Y,X)` and `f(X,Z,X)` are alphabetic variants;
`f(X,Y,X)` and `f(Y,X,Y)` are not;
`f(X,Y,X)` and `f(X,Y,Y)` are not;
`f(X,Y,X)` and `f(Z,V,Z)` are

This file used to define `subsumes_chk/2`, but this is now a built-in predicate:

`subsumes_chk(+`*General*`, +`*Specific*`)`
: is true when *General* subsumes *Specific*; that is, when *Specific* is already an instance of *General*. It does not bind any variables in *General* or in *Specific*.

There are currently two predicates in this file:

`subsumes(?`*General*`, +`*Specific*`)`
: is true when *General* subsumes *Specific*, and instantiates *General* so that it becomes identical to *Specific*. It does not further instantiate *Specific*.

`variant(?`*Term1*`, ?`*Term2*`)`
: is true when *Term1* and *Term2* are alphabetic variants. That is, `variant(`*Term1*`, `*Term2*`)` holds precisely when `subsumes_chk(`*Term1*`, `*Term2*`)` and `subsumes_chk(`*Term2*`, `*Term1*`)` both hold.

12.3.9 Unification — `library(unify)`

This file defines only one predicate.

`unify(?Term1, ?Term2)`
: is true when *Term1* unifies with *Term2*. The only difference between this predicate and the built-in predicate `Term1 = Term2` is that `unify/2` applies the "occur check" and the built-in predicate `=/2` does not. This means that according to ordinary logic, a variable X should *not* unify with a term containing X. `unify/2` does this correctly and `=/2` does not. Thus

```
| ?- unify(X, [X]).

no

| ?- X = [X].

X = [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[...
```

Whenever `unify(X, Y)` succeeds, `X = Y` would have succeeded and made the same variable bindings.

12.3.10 `library(termdepth)`

Many resolution-based theorem provers impose a depth bound on the terms they create. Not the least of the reasons for this is to prevent infinite loops. `library(termdepth)` provides five predicates:

`term_depth(+Term, -Depth)`
: *Depth* is unified with the depth of *Term*, calculated according to the following definition:

```
term_depth(Var) = 0
term_depth(Const) = 0
term_depth(F(T1,...,Tn)) =
    1 + max(term_depth(T1), ..., term_depth(Tn))
```

`depth_bound(+Term, +Bound)`
: This succeeds when the depth of *Term*, as calculated by `term_depth/2`, is less than or equal to *Bound*. *Bound* is assumed to be a strictly positive integer. Note that `depth_bound/2` will terminate even if *Term* is cyclic.

`term_size(+Term, -Size)`
: Unify *Size* with the size of *Term*, where the size is the number of constants and function symbols in the term. (Note that this is a lower bound on the size of any term instantiated from *Term*, and that instantiating any variable to a non-variable must increase the size. This latter property is why variables are counted as 0 rather than as 1.) The definition is

```
term_size(Var) = 0
term_size(Const) = 1
term_size(F(T1,...,Tn)) =
    1 + term_size(T1) + ... + term_size(Tn).
```

`size_bound(+Term, +Bound)`
: This succeeds when the size of *Term*, as calculated by `term_size/2`, is less than or equal to *Bound*. *Bound* is assumed to be a non-negative integer. Note that `size_bound/2` will always terminate even if *Term* is cyclic.

`length_bound(+List, +Bound)`
: is true when *List* is a list having at most *Bound* elements. *Bound* must be instantiated. If *List* ends with a variable, it will be instantiated to successively longer proper lists, up to the length permitted by *Bound*. Note that the depth of a list of constants is its length.

12.4 Text Processing

12.4.1 Introduction — `library(strings)`

Quintus Prolog has two data types that can be used to represent sequences of characters. For each sequence of up to N characters for some implementation-defined limit (N is 65532 in Quintus Prolog) there is exactly one atom with that character sequence as its name. Further, each atom has exactly one name.

Atoms provide a convenient and storage-efficient way of handling modest amounts of character data. Two atoms can be compared for identity very quickly using the built-in predicate `==/2`.

The second data type is *chars*. A convention in Prolog is that a list of values all of which belong to the data type *thing* is said to be of type *things*. A further convention is that integers representing character codes (the range is 1..255) are said to be of type *char*. So a *chars* value is a (possibly empty) list of character codes. A list can be of any length. Two lists can be compared for identity in time proportional to their length by using the built-in predicate `==/2`.

In the text that follows, the term *text object* can generally be taken to refer to an atom.

This section describes how to use the predicates defined in `library(strings)`. If you plan to do extensive text processing, you should consider using lists of character codes rather than atoms, because you will then be able to use an exceptionally powerful pattern-matching language for recognizing patterns and also for constructing new chars. See Section 8.16 [ref-gru], page 298 for more about this facility. It is highly recommended that you use grammar rules freely for list and text processing, and that you always consider whether grammar rules can be used clearly to accomplished a desired text processing effect before using the operations in `library(strings)`.

12.4.1.1 Access to operating system — `system/1`

`system(ListOfTextObjects)`
: is a version of `unix(shell(_))` that builds the command up out of pieces without the cost of interning a new atom, which may never be used again. It does whatever the C function `system(3)` does.

12.4.2 Type Testing

`name(+Constant)`
: is true when *Constant* is a text object. Same as `atom/1`.

Note particularly:

```
| ?- name("chars").

no

| ?- name("").

yes

| ?- atom("").

yes
```

12.4.3 Converting Between Constants and Characters

Quintus Prolog currently supports the following kinds of constants:

- atoms
- integers
- floating-point numbers

The data type "list of character codes" is called *chars*. You can convert between atoms, numbers, and chars using the following predicates:

- `name(+Constant, -Chars)`
- `atom_chars(?Atom, ?Chars)`
- `number_chars(?Number, ?Chars)`
- `char_atom(?Char, ?Atom)`

`name/2` is retained for compatibility with DEC-10 Prolog, C-Prolog, and earlier releases of Quintus Prolog. All the other predicates have names that follow a rule, which you would do well to follow in your own code.

Suppose you have two data types *foo* and *baz*, and a predicate that converts from one type to another. If each value of type *foo* corresponds to exactly one value of type *baz*, and if each value of type *baz* corresponds to at most one value of type *foo*, the predicate should be called

```
foo_baz(?The_foo, ?The_baz)
```

As an example, let *foo* be the data type "character code" (we call this "char"), and let *baz* be the data type "atom". Given any char *C*, there is exactly one atom whose name is [*C*]. Given any atom, either its name contains one single character *C*, or it contains some other number of characters, in which case there is no unique character to which it corresponds. Therefore, a predicate that converts between character codes and single-character atoms will have the name

```
char_atom(?Char, ?Atom)
```

Remember that this pattern means that we can *always* solve for the second argument given a value for the first, and that we *may* be able to solve for the first argument given a value for the second.

If a *foo* can be converted to a unique *baz*, but a *baz* might correspond to more than one *foo*, the predicate is to be called

```
foo_to_baz(+The_foo, ?The_baz)
```

The '`_to_`' tells you that the conversion only works one way around. For example, given an atom or number, there is a unique list of character codes that can be made from it, but a given list of character codes such as '`"0"` could have come from an atom or an integer. Therefore a predicate that converts between arbitrary constants and character codes should be called

```
constant_to_chars(+Constant, ?Chars)
```

In fact this operation is called `name/2`, because that is what it was called in DEC-10 Prolog.

All the new data type conversion predicates follow these naming rules. Now let us look at them.

12.4.3.1 `name(?Constant, ?Chars)`

If *Constant* is supplied, it should be an atom or number. *Chars* is unified with a list of character codes representing the "name" of the constant. These are precisely the characters that `write/1` would write if asked to write *Constant*.

If *Constant* is a variable, *Chars* should be a proper list of character codes. If *Chars* looks like the name of a number, *Constant* will be unified with that number. The syntax for numbers that is accepted by `name/2` is exactly the one that `read/1` accepts. If *Chars* does not look like the name of a number, *Constant* will be unified with an atom.

This attempt to guess what sort of constant you want means that there are atoms that can be constructed by `read/1` and by `atom_chars/2`, but not by `name/2`. `name/2` is retained for backwards compatibility with DEC-10 Prolog, C-Prolog, and earlier versions of Quintus Prolog. New programs should use `atom_chars/2` or `number_chars/2`, whichever is appropriate.

`name/2` is a built-in predicate. `library(strings)` contains a predicate `name1/2`, which is identical to `name/2` except that it reports errors in the same way as other library predicates.

12.4.3.2 atom_chars(*?Atom, ?Chars*)

Chars is the list of character codes comprising the printed representation of *Atom*. Initially, either *Atom* must be instantiated to an atom, or *Chars* must be instantiated to a proper list of character codes.

If *Atom* is initially instantiated to an atom, *Chars* is unified with a list of the character codes that make up its printed representation. If *Atom* is uninstantiated and *Chars* is initially instantiated to a list of characters, *Atom* is instantiated to an atom containing exactly those characters, even if the characters look like the printed representation of a number. If the arguments to `atom_chars/2` are both uninstantiated, an error is signalled.

`atom_chars/2` was built into the system in Release 2.0. Before that, it was a library predicate. `library(strings)` still contains the old code under the name `atom_chars1/2`.

12.4.3.3 number_chars(*?Number, ?Chars*)

Chars is the list of character codes comprising the printed representation of *Number*. Initially, either *Number* must be instantiated to a number, or *Chars* must be instantiated to a proper list of character codes.

If *Number* is initially instantiated to a number, *Chars* is unified with the list of character codes that make up its printed representation. If *Number* is uninstantiated and *Chars* is initially instantiated to a list of characters that corresponds to the correct syntax of a number (either integer or float), *Number* is bound to that number; otherwise `number_chars/2` will simply fail. If the arguments to `number_chars/2` are both uninstantiated, it signals an error.

`number_chars/2` was built into the system in Release 2.0. Before that, it was a library predicate. `library(strings)` still contains the old code under the name `number_chars1/2`.

12.4.3.4 char_atom(*?Char*, *?Atom*)

`char_atom/2` converts between character codes and atoms. *Atom* must be an atom or a variable. *Char* must be a character code or a variable. Character codes are integers in the range 1..255.

If either argument is instantiated, but *Char* is not a valid character code or *Atom* is not an atom, `char_atom/2` fails silently.

If *Char* is a valid character code, *Atom* is unified with the atom whose name is [*Char*].

If *Atom* is an atom, and its name contains a single character, *Char* is unified with the code of that character.

`char_atom(`*Char*`, `*Atom*`)` is true when *Char* is a character code, *Atom* is an atom whose name contains exactly one character, and *Code* is the code of that character. If *Atom*'s name has no characters, or more than one, this predicate is simply false. Error exceptions and efficiency aside, `char_atom/2` could be defined as

```
char_atom(Char, Atom) :-
        atom_chars(Atom, [Char]).
```

12.4.4 Comparing Text Objects

If you have two atoms, two character codes, or two lists of character codes to compare, the following built-in predicates can be used:

`@</2` lexicographically less than

`@>=/2` not less than

`@>/2` lexicographically greater than

`@=</2` not greater than

`==/2` identical to

`\==/2` not identical to

`compare/3`
three-way comparison

For example,

```
| ?- a @< b.

yes

| ?- a @> b.

no

| ?- compare(R, "fred", "jim").

R = <

| ?- 0'a @< 0'z.

yes
```

There are several points to note about these built-in comparison predicates:

1. They are sensitive to alphabetic case
2. If the two terms being compared are of different types, it is the types that are compared (integer < atom < list).
3. They behave as though the two operands were converted to character lists and the shorter operand were padded on the right with -1's.

It would be useful to have routines that ignored alphabetic case. `to_lower/2` and `to_upper/2` in `library(ctypes)` (Section 12.4.10 [lib-txp-ctypes], page 588) may be useful in writing your own.

`library(strings)` provides two string comparison predicates that address the other two points.

`compare_strings(-`*Relation*`, +`*Text1*`, +`*Text2*`)`
: takes two text objects and compares them, binding *Relation* to

  `<` if *Text1* lexicographically precedes *Text2*

  `=` if *Text1* and *Text2* are identical (but for type)

  `>` if *Text1* lexicographically follows *Text2*

`compare_strings(-`*Relation*`, +`*Text1*`, +`*Text2*`, +`*Pad*`)`
: is the same as `compare_strings/3`, except that it takes an additional parameter, which is a character code (an integer). In effect, it pads the shorter of *Text1* or *Text2* on the right with the padding character *Pad*, and calls `compare_strings/3` on the result.

We could have defined `compare_strings/[3,4]` this way:

```
compare_strings(Relation, Text1, Text2) :-
        name(Text1), name(Text1, Name1),
        name(Text2), name(Text2, Name2),
        compare(Relation, Name1, Name2).
compare_strings(Relation, Text1, Text2, Pad) :-
        name(Text1), name(Text1, Name1),
        name(Text2), name(Text2, Name2),
        pad(Name1, Name2, Pad, Full1, Full2),
        compare(Relation, Full1, Full2).

pad(Name1, [], Pad, Name1, Full2) :- !,
        pad(Name1, Pad, Full2).
pad([], Name2, Pad, Full1, Name2) :-
        pad(Name2, Pad, Full1).
pad([Char1|Name1], [Char2|Name2], Pad,
    [Char1|Full1], [Char2|Full2]) :-
        pad(Name1, Name2, Pad, Full1, Full2).

pad([], _, []).
pad([_|X], Pad, [Pad|Y]) :-
        pad(X, Pad, Y).
```

The point of `compare_strings/4` is that some programming languages define string comparison to use blank padding (`Pad`=32), while others define it to use NUL padding (`Pad`=0), and yet others use lexicographic comparison (`Pad`= -1) as `compare/3` and `compare_strings/3` do; `compare_strings/4` allows you to specify whichever is most useful for your application.

The host language function used to implement these operations is considerably more general. You may want to experiment with it.

Here are some examples:

```
| ?- % illustrating the difference between compare/3
|    % and compare_strings/3
|    compare(R1, fred, jim),
|    compare(R2, "fred", "jim").

R1 = <,
R2 = <

| ?- compare_strings(R1, fred, jim).

R1 = <
```

```
| ?- compare_strings(R2, "fred", "jim").
! Type error in argument 2 of compare_strings/3
! atom expected, but [102,114,101,100] found
! Goal:  compare_strings(_286,[102,114,101,100],
                              [106,105,109])

| ?- % illustrating compare_strings/4
|    Space is " ",
|    compare_strings(R, ' ', ''),
|    compare_strings(S, ' ', '', Space).

Space = 32,
R = <,
S = =
```

Another convention is sometimes used, in which the lengths of the atoms are compared first, and the text is examined only for atoms of the same length. You could program it thus:

```
xpl_compare(Relation, Text1, Text2) :-
        /* this is not in library(strings) */
        string_length(Text1, Length1),
        string_length(Text2, Length2),
        (   Length1 =:= Length2 ->
            compare_strings(Relation, Text1, Text2)
        ;   compare(Relation, Length1, Length2)
        ).
```

12.4.5 Concatenation

There are two approaches to concatenation. One is to provide a concatenation *function* that takes some number of text objects and yields their concatenation. The other is to provide a concatenation *relation*.

Quintus Prolog provides a built-in concatenation relation for lists, namely `append/3`. This concatenation relation can perforce be applied to lists of character codes.

```
| ?- ensure_loaded(library(printchars)).

| ?- append("app", "end", X).

X = "append"

| ?- append(X, "end", "intend").

X = "int"

| ?- append(_, [C|_], "to be written"),
|    put(C), fail.

to be written
no
```

`library(strings)` contains a concatenation relation for text objects. This relation was inherited from the DEC-10 Prolog library. The original code was written to support `gensym/2` (described in Section 12.4.9 [lib-txp-ato], page 587) and then generalized.

`concat(?Text1, +Constant2, ?Text3)`
: is true when *Text1* and *Text3* are the same kind of text object, *Constant2* is any sort of constant, and

```
name(Text1, Name1),
name(Constant2, Name2),
name(Text3, Name3),
append(Name1, Name2, Name3)
```

: is true. It can be used to solve for *Text1* given the other two arguments or to solve for *Text3* given the other two arguments, but unlike `append/3` it cannot be used to solve for *Constant2*.

This definition is retained for backwards compatibility with the DEC-10 Prolog and C-Prolog libraries, and with earlier versions of the Quintus library. `concat/3` may be removed from future versions of the Quintus library.

There is a proper concatenation relation that is exactly analogous to `append/3`:

`string_append(?A, ?Z, ?AZ)`
: is true when A, Z, and AZ are all atoms, and

```
name(A, NameA),
name(Z, NameZ),
name(AZ, NameAZ)
append(NameA, NameZ, NameAZ)
```

is true. It can be used to solve for any one of its arguments given the other two.

As a point of interest, `string_append/3` could have been defined using `midstring/4`, which is defined below.

```
append_strings(A, Z, AZ) :-
        midstring(AZ, A, Z, 0).
```

Examples:

```
| ?- concat(app, end, X).
X = append

| ?- string_append(app, end, X).
X = append

| ?- concat(X, end, append).
X = app

| ?- string_append(X, end, append).
X = app

| ?- concat(app, X, append).  % SURPRISE!
no

| ?- string_append(app, X, append).
X = end

| ?- concat(app, 137, X).
X = app137

| ?- string_append(app, 137, X).
no

| ?- concat(X, Y, ab).  % SURPRISE!
no

| ?- string_append(X, Y, ab).
X = '', Y = ab ;
X = a, Y = b ;
X = ab, Y = '' ;
no
```

12.4.5.1 Concatenation Functions

`library(strings)` defines a set of concatenation functions. Each of them takes a list of constants as its first argument, and returns the concatenation of the names of the constants as its second argument. They are

`concat_atom(+`*ListOfConstants*`, -`*Atom*`)`
: unifies *Atom* with the atom whose name is the concatenation of the names of the *ListOfConstants*.

`concat_atom(+`*ListOfConstants*`,` *Separator*`, -`*Atom*`)`
: like `concat_atom/2`, except that the elements of *Atom* are separated by *Separator*.

`concat_chars(+`*ListOfConstants*`, -`*Chars*`)`
: unifies *Chars* with the list of character codes that is the concatenation of the names of the *ListOfConstants*.

`concat_chars(+`*ListOfConstants*`, +`*Separator*`, -`*Chars*`)`
: like `concat_chars/2`, except that the elements of *Chars* are separated by `+`*Separator*.

Simplified versions of these predicates could have been defined thus:

```
concat_atom(Constants, Atom) :-
        concat_chars(Constants, Chars),
        atom_chars(Atom, Chars).

concat_chars([], []).
concat_chars([Constant|Constants], Chars0) :-
        name(Constant, Name),
        append(Name, Chars1, Chars0),
        concat_chars(Constants, Chars1).
```

There is one additional "feature": in place of a constant, you may supply a non-empty list of character codes. For example,

```
| ?- concat_atom([fred_,27], X).

X = fred_27
```

and

```
| ?- concat_atom([fred,"_",27], X).

X = fred_27
```

both work. Beware: an empty list of character codes, `""`, is in fact the atom written `[]`. Because of this ambiguity it is not possible to write a predicate that will accept *any* atom and *any* list of character codes, because `"" = []` is both. '`[]`' is the atom `[]`, which has two punctuation marks in its name. This is for compatibility with other Edinburgh Prologs. So while you might expect

```
| ?- concat_atom([fr,"",ed], fred).

no
```

you will in fact get

```
| ?- concat_atom([fr,"",ed], X).

X = 'fr[]ed'
```

This "feature" of allowing non-empty lists of character codes is thus sufficiently confusing that it is likely to be withdrawn in future releases of the Quintus library, and is retained in this release for backward compatibility with earlier releases of the library. The concatenation functions themselves will remain.

12.4.6 Finding the Length and Contents of a Text Object

There are two predicates for determining the length of a text object:

`string_size(+`*Text*`, -`*Length*`)`
`string_length(+`*Text*`, -`*Length*`)`
: *Length* is unified with the number of characters in the name of *Text*, which must be an atom.

These two predicates are identical except that `string_length/2` will report an error if its first argument is not a text object.

There are versions of Quintus Prolog on stock hardware that support Kanji. Those versions currently represent Kanji by *pairs* of characters. Beware of this difference. This is likely to change.

There are two predicates for extracting a character from a text object:

`string_char(?`*Index*`, +`*Text*`, ?`*Char*`)`
: unifies *Char* with the character code of the character at position *Index* (counting from 1) in *Text*. Being a selector predicate, its arguments follow the convention of being in the same order as those of `arg/3`; see the description of `library(args)`, Section 12.3.3 [lib-tma-arg], page 551. *Text* must be instantiated to a text object. *Index*, if instantiated, must be an integer. If *Index* is less than one or greater than the length of *Text*, `string_char/3` fails quietly. If *Index* is a variable, `string_char/3` will enumerate suitable values for *Index* and *Char*.

`nth_char(?`*Offset*`, +`*Text*`, ?`*Char*`)`
: is the same as `string_char/3` except that *Offset* counts from 0 rather than from 1. This predicate was added in this release to simplify conversion from another dialect, which is why it is inconsistent with Prolog conventions. We recommend that you use `string_char/3` in new programs instead.

```
| ?- string_size(fred, X).

X = 4

| ?- string_size(47, X).

no

| ?- string_length(fred, X).

X = 4

| ?- string_length(47, X).
! Type error in argument 1 of string_length/2
! symbol expected, but 47 found
! goal:  string_length(47,_43)

| ?- X is " ".

X = 32

| ?- string_char(3, 'an example', X).

X = 32

| ?- nth_char(2, 'an example', X).

X = 32

| ?- string_char(I, 'an example', 0'a).

I = 1 ;
I = 6 ;
no

| ?- nth_char(I, 'an example', 0'a).

I = 0 ;
I = 5 ;
no
```

We shall see in the next section that `nth_char/3` could have been defined by

```
nth_char(Offset, Text, Char) :-
        subchars(Text, [Char], Offset, 1, _).
```

If you wanted a predicate like `nth_char/3` but that counted from the right-hand end of the text instead of the left-hand end, you could define

```
nth_char_from_right(Offset, Text, Char) :-
        /* this is not in library(strings) */
        subchars(Text, [Char], _, 1, Offset).
```

12.4.7 Finding the width of a term — `library(printlength)`

`library(printlength)` provides four predicates to determine how wide a term would be if written:

`print_length(+`*Command*`, -`*Length*`)`
: This succeeds when *Command* would write *Length* characters to the current output stream, none of them being newline characters. `print_length/2` allows you to determine how many columns an atom (or other term) would take if printed according to *Command*. The length of the output of any command at all can be determined this way, provided that it writes to the current stream, and not to a stream argument. `print_length/2` fails if *Command* fails.

`print_length(+`*Command*`, ?`*StartColumn*`, ?`*EndColumn*`)`
: This succeeds when executing *Command* would write *EndColumn*-*StartColumn* characters. Either *StartColumn* or *EndColumn* should be instantiated to an integer. Then one can solve for the other argument. Quintus Prolog numbers columns starting from 0 (think of '`line_position`' as "the number of characters that have already been read from/written on this line"), so `print_length/3` will fail if *StartColumn* is negative. `print_length/3` fails if *Command* fails.

`print_lines(+`*Command*`, -`*Lines*`)`
: is true when Command would write *Lines* new-line characters to the current output stream. One use of this is to tell whether there would be any point in calling `print_length/2`.

`tab_to(+`*Column*`)`
: Ensure that `line_position(`*Current_output*`, `*Column*`)` is true by writing 0 or 1 newlines and at most *Column* spaces to the *Current_output* stream.

12.4.8 Finding and Extracting Substrings

The beauty of Prolog as a text processing language is that definite clause grammars (DCG's) are not only part of it, but almost an inevitable part, and may be used for constructing and decomposing pieces of text as well as matching them.

As an example of the power of definite clause grammars, suppose we want to take American-style dates apart. Here is a grammar:

```
usa_date(Y, M, D, MDY) :-
        usa_date(Y, M, D, MDY, "").

usa_date(Y, M, D) -->
        digits(M), "/", digits(D), "/", digits(Y).

digits([D|Ds]) -->
        [D], {is_digit(D)},
        (   digits(Ds)
        ;   {Ds = []}
        ).
```

With this definition, we can take dates apart:

```
| ?- usa_date(Y, M, D, "12/25/86").

Y = "86",
M = "12",
D = "25"

| ?- usa_date(Y, M, D, "2/1/87").

Y = "87",
M = "2",
D = "1".

| ?- usa_date(Y, M, D, "1-feb-87").

no
```

We can also put dates together:

```
| ?- usa_date("86", "12", "25", Date).

Date = "12/25/86"

| ?- usa_date("87", "2", "1", Date).

Date = "2/1/87"
```

Thanks to the fact that non-terminals in a DCG can take arguments, and with a little care, you can write quite complicated grammars that can be used for composition as well as decomposition.

If you want to do any sort of text processing in Prolog, you should learn how to use grammar rules. A well-written grammar requires less mental decoding than a program using the operations in `library(strings)`. Here are versions of `usa_date/4` written using the operations in `library(strings)`. `cons_date/4` can only build a date, and `dest_date/4`

can only take one apart. Both are simplified, and do not check that *Y*, *M*, and *D* are made of digits.

```
cons_date(Y, M, D, Date) :-
        concat_atom([Y,/,M,/,D], Date).

dest_date(Y, M, D, Date) :-
        substring(Date, /, I, 1),       % find left /
        substring(Date, /, J, 1, R),    % find right /
        J > I,
        substring(Date, Y, 0, I),       % extract Y
        substring(Date, D, _, R, 0),    % extract D
        P is I+1, Q is R+1,             % widen fringes
        substring(Date, M, P, _, Q).    % extract M
```

It is not immediately obvious what this does, whereas the version using grammar rules is considerably clearer.

The argument is sometimes raised that, while grammar rules may be more elegant, string operations are more efficient. However, it is the daily experience of Prolog programmers that "clean" and "efficient" tend to describe the same code.

The following relative times were measured using Quintus Prolog on an MC68020:

```
cons_date('86', '12', '25', _)
------------------------------ = 1.5
usa_date( "86", "12", "25", _)

dest_date(_, _, _, '86/12/25')
------------------------------ = 4.5
usa_date( _, _, _, "86/12/25")
```

In both cases, processing lists of character codes using grammar rules was more efficient than using "string" operations. If the "string" operations were built in, rather than being part of the library, they could be faster than they are. Even so, using grammar rules would still be the preferred method.

12.4.8.1 `midstring/[3,4,5,6]`

The `midstring/N` family has four members:

- `midstring(`*ABC*`, `*B*`, `*AC*`, `*LenA*`, `*LenB*`, `*LenC*`)`
- `midstring(`*ABC*`, `*B*`, `*AC*`, `*LenA*`, `*LenB*`)`
- `midstring(`*ABC*`, `*B*`, `*AC*`, `*LenA*`)`
- `midstring(`*ABC*`, `*B*`, `*AC*`)`

The principal routine for taking apart an atom is `midstring/6`.

`midstring/6`
is true when

- *ABC*, *B*, and *AC* are all atoms.
- *ABC* can be broken into three pieces *A*, *B*, and *C*,
- *AC* is the concatenation of *A* and *C*,
- *LenA* is the length of *A*,
- *LenB* is the length of *B*, and
- *LenC* is the length of *C*.

Either *ABC* or both *B* and *AC* should be instantiated. Apart from this restriction, `midstring/6` is a genuine relation, and all the other arguments can be solved for.

How can you be expected to remember all these arguments? The arguments fall into two groups: three text objects and three integers. The three integers form a little picture of what you want.

```
                |   LenA    |     LenB    |  LenC   |
        ABC=     a a a a a a B B B B B B B c c c c c
```

So the order of the three integer arguments is the same as the order of the substrings whose lengths they are. Note that none of these arguments is a "position" in any string: all three of them are to be understood as *lengths* of strings.

The order of the text arguments was chosen to extend the order used by `substring/5` (see Section 12.4.8.2 [lib-txp-sub-substring], page 582). Generally, you are more likely to know *ABC* than *B* or *AC*, and you are more likely to know *B* than *AC*. For example, a common use of the `midstring/[3,4,5,6]` family is to ask "if *B* is deleted from *ABC*, what results?", which can be done by asking

```
| ?- midstring(this_is_abc, is, AC, _, _, _).

AC = th_is_abc ;

AC = this__abc ;

no
```

Earlier in this section we saw `midstring/4` used to append two strings. Now we can see how that works:

```
| ?- midstring(BC, B, C, 0).
```

is true when *BC* can be broken into three parts, *A*, *B*, and *C*, such that 0 is the length of *A*, the lengths of *B* and *C* are unconstrained, and *C* is the concatenation of *A* and *C*.

Another way to see this is that

```
| ?- midstring (ABC, B, AC, N).
```

is true when *ABC* is obtained by inserting *B* just after the *N*th character of *AC*.

Some other examples using the `midstring/N` family:

- To delete 7 characters from `I"m going, Tom`, retaining the first 4 characters:

```
| ?- midstring('I''m going, Tom', _, Answer, 4, 7).

Answer = 'I''m Tom'
```

  To search for the text `ab` in `abracadabra`:

```
| ?- midstring(abracadabra, ab, _, Offset).

Offset = 0 ;
Offset = 7 ;
no
```

- To divide *Whole* into *Front* and *Back*, where the length of *Front* is known:

```
| ?- midstring(Whole, Front, Back, 0, FrontLength).
```

- To divide *Whole* into *Front* and *Back*, where the length of *Back* is known:

```
| ?- midstring(Whole, Front, Back, 0, _, BackLength).
```

- To insert *Part* into *Fringes* at a given *Offset* from the left, yielding *Whole*:

```
| ?- midstring(Whole, Part, Fringes, Offset).
```

- To insert *Part* into *Fringes* at a given *Offset* from the right, yielding *Whole*:

```
| ?- midstring(Whole, Part, Fringes, _, _, Offset).
```

- To delete *Drop* characters from *Whole*, starting at a given *Offset* from the left, yielding Short:

```
| ?- midstring(Whole, _, Short, Offset, Drop).
```

- To delete *Drop* characters from *Whole*, starting at a given *Offset* from the right, yielding *Short*:

```
| ?- midstring(Whole, _, Short, _, Drop, Offset).
```

- To determine the *Length* of *Text*:

```
| ?- midstring(Text, Text, '', 0, Length, 0).
```

It would not be useful to try to memorize these examples. Instead, remember the *picture*:

```
ABC,  B,  AC,  LenA,  LenB,  LenC
```

Note that `midstring/[3,4,5,6]` has been carefully designed so that you can extract and insert from either the left or the right with equal facility, and so that successive calls to extract related fragments will require a minimum of arithmetic. For example, suppose we want to break a text *ABCD* into four pieces, where the lengths of *B* and *D* are known. We think about the picture

```
ABCD = '' (ABC) D
ABC  = A (B) C
AC   = A (C) ''
```

and then write

```
four_parts(ABCD, A, B, C, D, LenB, LenD) :-
        /* this is an example */
        midstring(ABCD, ABC, D, 0, _, LenD),
        midstring(ABC, B, AC, LenA, LenB),
        midstring(AC, C, A, LenA, _, 0).
```

12.4.8.2 substring/[4,5]

Very often you are not interested in the third argument that `midstring/[3,4,5,6]` return. This argument is not returned or constructed by `substring/[4,5]`. In fact, `substring/[4,5]` were designed first, and `midstring/[3,4,5,6]` were developed from it to provide string insertion and deletion.

- `substring(+`*`ABC`*`, ?`*`B`*`, ?`*`LenA`*`, ?`*`LenB`*`, ?`*`LenC`*`)`
- `substring(+`*`ABC`*`, ?`*`B`*`, ?`*`LenA`*`, ?`*`LenB`*`)`

are true when

- *ABC* and *B* are both atoms,
- *ABC* can be broken into three pieces *A*, *B*, and *C*,
- *LenA* is the length of *A*,
- *LenB* is the length of *B*, and
- *LenC* is the length of *C*.

The *ABC* argument must be instantiated. The *B* argument may be instantiated (this provides a string search facility) but need not be. This is the reason for the argument order: arguments that are strict inputs should precede other arguments, and `substring/5` has precisely one strict input.

The point of `substring/5` is to let you work from the right-hand end of the text as easily as from the left-hand end. But the fact that Prolog is based on relations rather than functions means that this one operation can replace both substring and string search operations.

Thus to determine where `a` occurs in `abracadabra`,

```
| ?- substring(abracadabra, a, Offset, _).

Offset = 0 ;
Offset = 3 ;
Offset = 5 ;
Offset = 7 ;
Offset = 10 ;
no
```

This is the preferred way of searching for a substring in Quintus Prolog. Note that if you use `substring/5` to search for a substring, you can then extract the preceding or following characters thus:

```
| ?- substring(Text, Pattern, L_Before, L_Pattern, L_After),
|    substring(Text, Before, 0, L_Before, _),
|    substring(Text, After, _, L_After, 0).
```

Again, this is not guesswork, but is arrived at by thinking about the picture

```
Text = Before (Pattern) After
     = '' (Before) Pattern After
     = Before Pattern (After) ''
```

There are two other predicates in this family:

```
index(Pattern, String, Offset) :-
        substring(String, Pattern, Offset, _).

string_search(Pattern, String, Offset) :-
        substring(String, Pattern, Offset, _).
```

`index/3` is retained for compatibility with earlier releases of `library(strings)`. It will be withdrawn in a future release. If you have code that uses `index/3`, you should replace calls to `index/3` by calls to `string_search/3`, which is an exact synonym for it.

12.4.8.3 subchars/[4,5]

`subchars/[4,5]` are identical to `substring/[4,5]` (Section 12.4.8.2 [lib-txp-sub-substring], page 582) except that the second argument, B, is a list of character codes, not a text object.

`subchars(+`*ABC*`, ?`*B*`, ?`*LenA*`, ?`*LenB*`, ?`*LenC*`)`
`subchars(+`*ABC*`, ?`*B*`, ?`*LenA*`, ?`*LenB*`)`
are true when

- ABC is a text object,
- B is a list of character codes (it may be empty),
- ABC can be broken into three pieces A, B, and C,
- $LenA$ is the length of A,
- $LenB$ is the length of B, and
- $LenC$ is the length of C.

```
| ?- subchars(frederica, Name, 4, 4).

Name = "eric"
```

These predicates avoid constructing an atom that you may have no further use for.

12.4.8.4 The "span" family

The midstring, substring, and subchars families give you a way of taking strings apart when you know the lengths of the substrings you want, or when you know a particular substring.

The "span" family gives you another way of taking strings apart. The family contains three sub-families: `span_left/[3,4,5]`, which scan from the left, `span_right/[3,4,5]`, which scan from the right, and `span_trim/[3,4,5]`, which scans from both ends towards the middle.

`span_left(+`*Text*`, +`*Set*`, ?`*LenA*`, ?`*LenB*`, ?`*LenC*`)`
`span_left(+`*Text*`, +`*Set*`, ?`*LenA*`, ?`*LenB*`)`
`span_left(+`*Text*`, +`*Set*`, ?`*LenA*`)`

are true when

- *Text* is a text object,
- *Set* specifies a set of characters (see below), and
- *Text* can be broken into three pieces *A*, *B*, *C*, such that
  - *LenA* is the length of *A*,
  - *LenB* is the length of *B*,
  - *LenC* is the length of *C*,
  - no character in *A* belongs to the *Set*,
  - every character in *B* belongs to the *Set*,
  - *B* is not empty (so some character of *Text* must belong to the *Set*),
  - *C* contains the rest of *Text* (it may contain characters from *Set*), and
  - *A* and *B* are as long as possible.

The *Set* is

- an atom *A*. A character belongs to such a *Set* if and only if it occurs in the name of A. The atom `''` represents an empty *Set*.
- a non-empty list of character codes [*C1*,. . . ,*Cn*]. A character belongs to set a *Set* if and only if it occurs among the character codes *C1*,. . . ,*Cn*.
- `not(`*X*`)`, where *X* is an atom or non-empty list of characters. A character belongs to such a *Set* if and only if it does *not* belong to the set *X*.

The first two arguments must be instantiated. Given them, the remaining three arguments are uniquely determined. The last three arguments give you a picture of how the text is divided:

```
          |   LenA    |    LenB     |  LenC   |
Text=    a a a a a a B B B B B B B c c c c c
                        \____Set____/
```

where *Set* embraces the characters in the *B* substring. By design, the *Set* argument occupies the same position in the argument list of this predicate that *B* does in the argument list

of `substring/[4,5]` or `midstring/[3,4,5,6]`. The fact that the last three arguments of `span_left/5` follow this convention means that you can use `midstring/[3,4,5,6]`, `substring/[4,5]`, or `subchars/[4,5]` to extract whichever substring interests you.

For example, to skip leading spaces in *String*, yielding *Trimmed*, you would write

```
| ?- span_left(String, not(" "), Before),
|    substring(String, Trimmed, Before, _, 0).
```

Note that this fails if there are no non-blank characters in *String*. To extract the first blank-delimited *Token* from *String*, yielding a *Token* and the *Rest* of the string, you would write

```
| ?- span_left(String, not(" "), Before, Length, After),
|    substring(String, Token, Before, Length, After),
|    substring(String, Rest, _, After, 0).
```

`span_right(+`*Text*`, +`*Set*`, ?`*LenA*`, ?`*LenB*`, ?`*LenC*`)`
`span_right(+`*Text*`, +`*Set*`, ?`*LenB*`, ?`*LenC*`)`
`span_right(+`*Text*`, +`*Set*`, ?`*LenC*`)`
: are true when

  - *Text* is a text object,
  - *Set* specifies a set of characters, and
  - *Text* can be broken into three pieces A, B, C, such that
    - *LenA* is the length of A,
    - *LenB* is the length of B,
    - *LenC* is the length of C,
    - no character in C belongs to the *Set*,
    - every character in B belongs to the *Set*,
    - B is not empty (so some character of *Text* must belong to the *Set*), and
    - C and B are as long as possible.

These three predicates are exactly like `span_left/[3,4,5]` except that they work from right to left instead of from left to right. In particular, the picture

```
                |   LenA    |    LenB     |  LenC   |
        Text=    a a a a a a B B B B B B B c c c c c
                             \____Set____/
```

applies.

Finally, there are predicates that scan from both ends:

`span_trim(+`*Text*`, +`*Set*`, ?`*LenA*`, ?`*LenB*`, ?`*LenC*`)`
: is true when

- *Text* is a text object,
- *Set* specifies a set of characters, and
- *Text* can be broken into three pieces *A*, *B*, *C*, such that
  - *LenA* is the length of *A*,
  - *LenB* is the length of *B*,
  - *LenC* is the length of *C*,
  - every character in *A* belongs to the *Set*,
  - every character in *C* belongs to the *Set*,
  - *A* and *C* are as long as possible, and
  - *B* is not empty.

The *Set* argument of `span_trim/5` has the same form as the *Set* argument of `span_left/[3,4,5]` or `span_right/[3,4,5]`, but there is an important difference in how it is used: in `span_trim/5` the *Set* specifies the characters that are to be trimmed away. The picture is

```
             |   LenA    |    LenB     |  LenC   |
    Text=     a a a a a a B B B B B B B c c c c c
              \___Set___/              \__Set__/
```

There is a special case of `span_trim/5` that enables you to strip particular characters from both ends of a string. These unwanted characters are designated in *Set* in `span_trim/3`:

```
span_trim(String, Set, Trimmed) :-
        span_trim(String, Set, Before, Length, After),
        substring(String, Trimmed, Before, Length, After).
```

A further specialization, `span_trim/2`, is intended for trimming blanks from fixed-length records:

```
span_trim(String, Trimmed) :-
         span_trim(String, " ", Before, Length, After),
         substring(String, Trimmed, Before, Length, After).
```

For example,

```
| ?- span_trim('  abc    ', " ", B, L, A).
B = 2
L = 3
A = 4

| ?- substring('  abc    ', Trimmed, 2, 3, 4).
Trimmed = abc

| ?- span_trim(' an   example ', Trimmed).
Trimmed = 'an   example'
```

Note that the last example leaves the group of three internal blanks intact. There are no predicates in `library(strings)` for compressing such blanks.

In manipulating text objects, do not neglect the possibility of combining the "span" family with `subchars/[4,5]` or `midstring/[3,4,5,6]`.

12.4.9 Generating Atoms

`library(strings)` defines three predicates for generating atoms.

`gensym(+`*`Prefix`*`, -`*`Atom`*`)`
: is equivalent to `concat_atom([`*`Prefix`*`, `*`N`*`], `*`Atom`*`)`, where *N* is the next number in the sequence associated with *Prefix*. Notionally, each *Prefix* has its own counter, starting with zero. *Prefix* must be an atom. Examples:

```
| ?- gensym(a, X).

X = a1

| ?- gensym(a, X).

X = a2

| ?- gensym(b, X).

X = b1

| ?- gensym(1, X).

no
```

`gensym(-`*`Atom`*`)`
: uses the *Prefix* '`%`'. Example:

```
| ?- gensym(X).

X = '%58'
```

`cgensym(+`*`Prefix`*`, ?`*`Atom`*`)`
: if *Atom* is a variable, it generates a new atom using `gensym/2`. Otherwise, *Prefix* and *Atom* should be atoms. The name is to be read as "**c**onditionally **gen**erate **sym**bol".

These predicates are included for compatibility with the DEC-10 Prolog library, which has contained them for several years. `cgensym/2` will do nothing to *Atom* if it is already bound; otherwise it is just like `gensym/2`.

12.4.10 Case Conversion — `library(ctypes)`

There are no case conversion operations in the supported `library(strings)` package. In the ASCII and EBCDIC character sets, case conversion is well defined, because each letter is either an uppercase letter (which has a unique lowercase equivalent) or a lowercase letter (which has a unique uppercase equivalent). Not all of the character sets have this property.

There are case conversion operations in `library(ctypes)`. They are

`to_lower(?`*Upper*`, ?`*Lower*`)`
: which is true when *Upper* and *Lower* are valid character codes, and either *Upper* is the code of an uppercase letter that has a unique lowercase equivalent, and *Lower* is the code of that unique lowercase equivalent, or *Upper* is the code of some other character, and *Lower* is the same as *Upper*. Note that this means that if *Lower* is the code of a lowercase letter that is the unique equivalent of some uppercase letter, there are two solutions for *Upper*.

`to_upper(?`*Lower*`, ?`*Upper*`)`
: which is true when *Lower* and *Upper* are valid character codes, and either *Lower* is the code of a lowercase letter that has a unique uppercase equivalent; and *Upper* is the code of that unique uppercase equivalent, or *Lower* is the code of some other character, and *Upper* is the same as *Lower*. Note that this means that if *Upper* is the code of an uppercase letter that is the unique equivalent of some lowercase letter, there are two solutions for *Lower*.

In the ASCII and EBCDIC character sets, these definitions behave as one would expect. But consider the case of Greek.

Capital sigma is undeniably an uppercase letter; yet it has two lowercase equivalents: one for use at the end of words and one for use elsewhere. This means that `to_upper/2` would map both medial and final lowercase sigma to uppercase sigma, but that `to_lower/2` would leave uppercase sigma unchanged. A similar problem exists in German, where '`ß`' is a lowercase letter whose uppercase equivalent is the *pair* of letters '`SS`'.

Because of such problems, `library(caseconv)` is only adequate for ASCII or EBCDIC. This package defines two groups of predicates. The predicates in the first group test the case of a name. Those in the second group convert the case of a name or a non-empty list of character codes.

`lower(+`*Text*`)`
: is true if *Text* contains no uppercase letters.

`upper(+`*Text*`)`
: is true if *Text* contains no lowercase letters.

`mixed(+`*Text*`)`
: is true if *Text* contains at least one lowercase letter and and least one uppercase letter.

In each case, *Text* may contain other things than letters. If `mixed(`*Text*`)` is true, then `lower(`*Text*`)` and `upper(`*Text*`)` must both be false. However, `lower(`*Text*`)` and `upper(`*Text*`)` can both be true if *X* contains no letters at all. Examples:

```
| ?- lower(a).

yes

| ?- lower(quixotic).

yes

| ?- lower('Quixotic').

no

| ?- lower(**).

yes

| ?- upper(a).

no

| ?- upper('QUIXOTIC').

yes

| ?- upper('Quixotic').

no

| ?- upper(**).

yes

| ?- mixed(quixotic).

no
```

```
| ?- mixed('QUIXOTIC').

no

| ?- mixed('!$Quixotic<<<').

yes

| ?- mixed(**).

no
```

`lower(+`*Given*`, ?`*Lower*`)`
: unifies *Lower* with a lowercase version of *Given*. Uppercase letters are converted to lowercase, and no other changes are made. `lower(`*Lower*`)` is true.

`upper(+`*Given*`, ?`*Upper*`)`
: unifies *Upper* with an uppercase version of *Given*. Lowercase letters are converted to uppercase, and no other changes are made. `upper(`*Upper*`)` is true.

`mixed(+`*Given*`, ?`*Mixed*`)`
: unifies *Mixed* with a mixed-case version of *Given*. In each block of consecutive letters, the first letter is converted to uppercase and the following letters are converted to lowercase. No other changes are made. `mixed(`*Mixed*`)` is true if and only if *Given* contained at least two adjacent letters; otherwise `upper(`*Mixed*`)` is true.

In each of these predicates, *Given* may be an atom or a non-empty list of character codes. If *Given* is a number, these predicates will quietly fail. The action of these predicates on other terms is not defined. The second argument is unified with a term of the same type as *Given*, containing the same number of characters.

Examples (assuming that `library(printchars)` has been loaded):

```
| ?- lower("Are other character sets a REAL problem?", X).

X = "are other character sets a real problem?"

| ?- upper('Yes, they are!', X).

X = 'YES, THEY ARE!'

| ?- mixed('what a nuisance', X).

X = 'What A Nuisance'
```

```
| ?- upper(1.2e3, X).

no

| ?- lower('1.2E3', X).

X = '1.2e3'
```

Note that numbers cannot be converted by these predicates.

12.4.11 Note

12.5 XML Parsing and Generation

`library(xml)` is a package for parsing XML with Prolog, which provides Prolog applications with a simple "Document Value Model" interface to XML documents. A description of the subset of XML that it supports can be found at: http://homepages.tesco.net/binding-time/xml.pl.html

The package, originally written by Binding Time Ltd., is in the public domain and unsupported. To use the package, enter the query:

```
| ?- use_module(library(xml)).
```

The package represents XML documents by the abstract data type *document*, which is defined by the following grammar:

document	::= `xml(`*attributes*`,`*content*`)`	{ well-formed document }
	| `malformed(`*attributes*`,`*content*`)`	{ malformed document }
attributes	::= `[]`	
	| `[`*name*`=`*chardata*`|`*attributes*`]`	
content	::= `[]`	
	| `[`*cterm*`|`*content*`]`	
cterm	::= `pcdata(`*chardata*`)`	{ text }
	| `comment(`*chardata*`)`	{ an XML comment }
	| `namespace(`*URI*`,`*prefix*`,`*element*`)`	{ a Namespace }
	| `element(`*tag*`,`*attributes*`,`*content*`)`	{ <*tag*>..</*tag*> encloses *content* or <*tag* /> if empty }
	| `instructions(`*name*`,`*chardata*`)`	{ A PI <? *name chardata* ?> }
	| `cdata(`*chardata*`)`	{ <![CDATA[*chardata*]]> }

	`| doctype(`*tag*`,`*doctypeid*`)`	{ DTD <!DOCTYPE .. > }
	`| unparsed(`*chardata*`)`	{ text that hasn't been parsed }
	`| out_of_context(`*tag*`)`	{ *tag* is not closed }
tag	::= *atom*	{ naming an element }
name	::= *atom*	{ not naming an element }
URI	::= *atom*	{ giving the URI of a namespace }
chardata	::= *chars*	{ see Section 18.1.4.2 [mpg-ref-aty-ety], page 988 }
doctypeid	::= `public(`*chardata*`,`*chardata*`)`	
	`| system(`*chardata*`)`	
	`| local`	

The following predicates are exported by the package:

`xml_parse(+`*Chars*`, -`*Document*`[, +`*Options*`])`
: Parses *Chars*, a *chars*, to *Document*, a *document*. *Chars* is not required to represent strictly well-formed XML.

  Options is a list of zero or more of the following, where *Boolean* must be `true` or `false`:

  `format(`*Boolean*`)`
  : Indent the element content (default `true`).

  `extended_characters(`*Boolean*`)`
  : Use the extended character entities for XHTML (default `true`).

  `remove_attribute_prefixes(`*Boolean*`)`
  : Remove namespace prefixes from attributes when it's the same as the prefix of the parent element (default `false`).

`xml_parse(-`*Chars*`, +`*Document*`[, +`*Options*`])`
: Generates *Chars*, a *chars*, from *Document*, a *document*. If *Document* is not a valid *document* term representing well-formed XML, an exception is raised.

  In this usage of the predicate, the only option available is `format/1`.

`xml_subterm(+`*Term*`, ?`*Subterm*`)`
: Unifies *Subterm* with a sub-term of *Term*, a *document*. This can be especially useful when trying to test or retrieve a deeply-nested subterm from a document.

`xml_pp(+`*Document*`)`
: "Pretty prints" *Document*, a *document*, on the current output stream.

12.6 Negation

12.6.1 Introduction — `library(not)`

This section describes the predicates provided by `library(not)`: `not/1`, `\=/2`, `~=/2`, and `once/1`. For comparison purposes, the negation facilities that are built into the Prolog system are also described.

12.6.2 The "is-not-provable" Operator

DEC-10 Prolog, C-Prolog, and Quintus Prolog all provide an "is-not-provable" operator '`\+`'. The meaning of

```
\+ Goal
```

is that the Prolog system is unable to find any solution for Goal as it currently stands.

In operational terms, we have

```
\+ Goal :-
    ( call(Goal) -> fail
    ; true
    ).
```

In DEC-10 Prolog and C-Prolog, this is exactly how '`\+`' is implemented. In Quintus Prolog there is a slight difference: the Quintus Prolog compiler supports '`(`*`if`* `->` *`then`* `;` *`else`*`)`' directly, so a clause of the form

```
p :-
     q,
     \+ r,
     s.
```

is compiled as if you had written

```
p :-
     q,
     ( r -> fail ; true ),
     s.
```

If '`\+`' were not known to the compiler, the form *r* would be built as a data structure and passed to '`\+`' to interpret, which would be slower and would require more memory. The extra efficiency of having '`\+`' handled directly by the compiler is well worth having.

12.6.3 "is-not-provable" vs. "is-not-true" — not(*Goal*)

If the difference between the Prolog "is-not-provable" operator ('\+') and the standard negation operator of logic is not taken into account, you may find that some of your programs will behave in an unexpected manner. Here is an example:

```
man(john).
man(adam).

woman(sue).
woman(eve).

married(adam, eve).

married(X) :-
        married(X, _).
married(X) :-
        married(_, X).

human(X) :-
        man(X).
human(X) :-
        woman(X).
```

The question

```
| ?- \+ married(Who).
```

is a perfectly good one, to which one might at first glance expect the response to be `john` or `sue`. However, the meaning of this clause to Prolog is

> is it the case that the term `married(Who)` has
> no provable instances?

to which the answer is '`no`', as `married(adam)` is an instance of `married(Who)` and is provable by Prolog. The question

```
| ?- \+ dead(X).
```

is also a perfectly good one, and after perhaps complaining that `dead/1` is undefined, Prolog will report the answer '`yes`', because `dead(X)` has no instance that can be proven from this database. In effect, this means "for all X, `dead(X)` is not provable". Even though "`dead(adam)` is not provable" is a true consequence of this statement, Prolog will *not* report '`X = adam`' as a solution of this question. This is not the function of `\+/1`.

Note also that "`dead(adam)` is false" is *not* a valid consequence of this database, even though "`dead(adam)` is not provable" is true. To deduce one from the other requires the

use of the "Closed World Assumption", which can be paraphrased as "anything that I do not know and cannot deduce is not true". See any good book on logic programming (such as *Foundations of Logic Programming* by John Lloyd, Springer-Verlag 1984) for a fuller explanation of this.

We would very often like an operation that corresponds to logical negation. That is, we would like to ask

```
| ?- not married(X).
```

and have Prolog tell us the names of some people who are not married, or to ask

```
| ?- not dead(X).
```

and have Prolog name some people who are not dead. The unprovability operator will not do this for us. However, we *can* use `\+/1` as if it were negation, but only for certain tasks under some conditions that are not very restrictive.

The first condition is that if you want to simulate '`not(`*p*`)`' with '`\+(`*p*`)`', you must first ensure that you have complete information about *p*. That is, your program must be such that every true ground instance of *p* can be proved in finite time, and that every false ground instance of *p* will fail in finite time. Database programs often have this property.

Even then, given a non-ground instance of *p*, '`not(`*p*`)`' would be expected to bind some of the variables of *p*. But by design, '`\+(`*p*`)`' *never* binds any variables of *p*. Therefore the second condition is that when you call '`\+(`*p*`)`', *p* should be ground, or '`\+(`*p*`)`' will not simulate '`not(`*p*`)`' properly.

Checking the first condition requires an analysis of the entire program. Checking that *p* is ground is relatively simple. Therefore, the library file `library(not)` defines an operation

```
not Goal
```

which checks whether its *Goal* argument is ground, and if it is, attempts to prove '`\+ Goal`'. Actually, the check done is somewhat subtler than that. The simulation can be sound even when some variables remain; for example, if `left_in_stock(Part, Count)` has at most one solution for any value of `Part`, then

```
\+ (left_in_stock(Part,Count), Count < 10)
```

is perfectly sound provided you do not use `Count` elsewhere in the clause. You can tell `not/1` that you take responsibility for a variable's being safe by existentially quantifying it (see the description of `setof/3`), so

```
not Count^(left_in_stock(Part,Count), Count < 10)
```

demands only that `Part` must be ground. Even so, this is not particularly good style, and you would be better off adding a predicate

```
fewer_in_stock_than(Part, Limit) :-
        left_in_stock(Part, Count),
        Count < Limit.
```

and asking the question

```
not fewer_in_stock_than(Part, 10)
```

If you want to find instances that do not satisfy a certain test, you can generally enumerate likely candidates and check each one. For example,

```
| ?- human(H), not married(H).

H = john ;

H = sue

| ?- man(M), not dead(M).

M = john ;

M = adam
```

The present library definition of `not/1` warns you when you would get the wrong answer, and offers you the choice of aborting the computation or accepting possible incorrect results.

12.6.4 Inequality

DEC-10 Prolog, C-Prolog, and Quintus Prolog provide two inequality operations:

Term1 `\==` *Term2*
: *Term1* is not currently identical to *Term2*

Expr1 `=\=` *Expr2*
: *Expr1* and *Expr2* are arithmetic expressions with different values

`=\=/2` is reasonably straightforward. Either it is true, and will remain true, or it is false, and will remain false, or the Prolog system will report an error if it cannot determine which. Thus `1 =\= 2` is true, `1 =\= 1.0` is false, and `_ =\= 3` will result in an error exception.

`\==/2` is not really a logical relation, but a meta-logical relation like `var/1`. It tests whether two terms are exactly identical, down to having the same variables in the same places. It either succeeds or fails; and if it fails it will continue to do so, but if it succeeds it may fail later on. For example,

```
| ?- X \== Y,  % succeeds
|    X  =  Y,  % succeeds, unifying X and Y
|    X \== Y.  % FAILS, now that X and Y are unified

no
```

It is safe to use `==/2` and `\==/2` to test the equality of terms when you know in advance that the terms will be ground by the time the test is made.

12.6.4.1 *Term1* \= *Term2*

`library(not)` defines another inequality predicate, as do the C-Prolog and DEC-10 Prolog libraries:

Term1 `\=` *Term2*
: *Term1* and *Term2* do not unify

This means exactly the same thing as

```
\+ Term1 = Term2
```

and is subject to the same problems as the use of unprovability to simulate negation. However, when the terms are ground, it is more efficient than `\==/2`.

12.6.4.2 *Term1* ~= *Term2*

Obviously, if we can have a version of `\+/2` that checks whether it is safe to proceed, we can have a version of `\=/2` that does the same. So `library(not)` also defines a "sound inequality" predicate

Term1 `~=` *Term2*
: *Term1* and *Term2* are not equal

There are three cases: it may succeed, or fail, or warn you that there is not enough information yet to tell. Note that `~=/2` is a bit more clever than `not(T1 = T2)` would be:

```
f(X, a) ~= f(Y, b)
```

will succeed (correctly) even though

```
not(f(X,a) = f(Y,b))
```

would complain about the unbound variables `X` and `Y`.

As `~=/2` is sound and `\=/2` is not, we recommend that you use `~=/2` rather than `\=/2` in your Prolog code.

12.6.5 Forcing Goal Determinacy — `once(`*Goal*`)`

We have seen that

```
\+ Goal
```

and

```
( Goal -> false ; true )
```

have the same effect. The form

```
( Goal -> true ; false )
```

is also useful: it commits to the first solution of Goal.

This operation has a name:

```
once Goal
```

`once/1` is useful when you have a nondeterminate definition for *Goal*, but in this case you want to solve it determinately. For example, to find out if there are any married couples, you could use

```
once married(_, _)
```

If the predicate is really determinate in concept, it is better to make it determinate in definition as well, rather than to use `once/1`.

12.6.6 Summary

`library(not)` defines

```
not Goal
Term1 \= Term2
Term1 ~= Term2
once(Goal)
```

The tests

```
\+ Goal
Term1 \== Term2
Expr1 =\= Expr2
```

are built into Quintus Prolog already, and are described in the reference pages.

12.7 Operations on Files

12.7.1 Introduction — `library(files)`

The package `library(files)` provides operations on individual files, such as renaming, deleting, opening, and checking for existence.

You may also find `library(directory)` of interest (see Section 12.8 [lib-lfi], page 607).

12.7.2 Built-in Operations on Files

The following operations on files are described in the reference pages.

`absolute_file_name(+RelFileName, ?AbsFileName)`
: takes a filename *RelFileName* (typically typed in by the user of a program) and unifies it with the normalized *AbsFileName*.

  Beware: `absolute_file_name/2` mimics the filename resolution done by commands such as `compile/1`. It is meant primarily for looking up Prolog source files. If you want to find the absolute filename of any other file, `absolute_file_name/2` may not be appropriate. See `file_member_of_directory/2` in Section 12.8.2 [lib-lfi-fdi], page 608.

`close(+FileNameOrStream)`
: if *FileNameOrStream* is an atom, it refers to a DEC-10-compatible stream connected to the file of that name; otherwise it is a stream object. In either case, the associated stream is closed.

`current_stream(?AbsFileName, ?Mode, ?Stream)`
: is true if *Stream* is a stream connected to file *AbsFileName*, and currently open in mode *Mode*. `current_stream/2` can be used to list currently open streams and the files to which they are connected.

`library_directory(?Directory)`
: unifies *Directory* with an entry from the user-modifiable table of directories to be searched for library files.

`open(+FileName, +Mode, -Stream)`
: opens a new *Stream* connected to the file named by *FileName*. If *Mode* is `write` (`append`) it will (may) create a new file.

`see(+FileNameOrStream)`
: opens the file or stream *FileNameOrStream* for input if it is not already open.

`tell(+FileNameOrStream)`
: opens the file or stream *FileNameOrStream* for output if it is not already open.

`unix(cd(+`*Directory*`))`

selects *Directory* as the default directory for relative file names. This is also known as the current working directory. To find out what the default directory currently is, type

```
| ?- absolute_file_name(., CurrentDirectory).
```

12.7.3 Renaming and Deleting Files

`library(files)` defines four predicates pertaining to deleting and renaming files. `rename/2` and `dec10_rename/2` are identical replacements for the DEC-10 Prolog/C-Prolog `rename/2` command. They should only be used to convert old code to Quintus Prolog. New programs should use `delete_file/1` and `rename_file/2`.

`delete_file(+`*FileName*`)`

FileName should be an atom naming a file that currently exists and can be deleted. If so, the file it names is deleted, and the command succeeds. If not, an error exception is raised. and the command fails. Examples:

```
| ?- delete_file('ask.otl').

yes

| ?- delete_file('does_not_exist').
! Existence error in delete_file/1
! file nosuch does not exist
! O/S error : No such file or directory
! goal:  delete_file(does_not_exist)
| ?- unix(system('cat </dev/null >search.d/tmp')),
     unix(system('chmod a-w search.d')),
     delete_file('search.d/tmp').
! O/S error : Permission denied
! goal:  delete_file('search.d/tmp')
| ?- delete_file("tmp").
! Type error in argument 1 of delete_file/1
! symbol expected, but [116,109,112] found
! goal:  delete_file([116,109,112])

| ?- unix(system('rm tmp')).
rm: override protection 444 for tmp? n
yes             % did NOT delete the file

| ?- delete_file(tmp).

yes             % **DID** delete the file
```

This last example is important: the `rm` command (see `rm(1)`) checks the permission bits of the file (see `chmod(1)`) and asks you whether you really want to delete a file that you do not have write permission for, even if you have permission to delete it. `delete_file/1` does not do this.

`rename_file(+`*`OldName`*`, +`*`NewName`*`)`

OldName should be an atom naming a file that currently exists and can be renamed, and *NewName* should be a valid filename to which the file can be renamed. If so, the file will be renamed, and the command will succeed. If not, an error exception will be raised. Examples:

```
| ?- rename_file(does_not_exist, imaginary).
! Existence error in rename_file/2
! file does_not_exist does not exist
! O/S error : No such file or directory
! goal:  rename_file(does_not_exist,imaginary)
```

`rename_file/2` and `delete_file/1` have no effect on currently open streams, whether opened by `open/3`, `see/1`, or `tell/1`.

What will happen if you continue to use streams that used to be connected to files affected by these commands is system-dependent. Under UNIX, input will continue to come from a file as if it had not been renamed, and output will continue to go to a file as if it had not been renamed. For example:

```
% prolog
| ?- compile(library(files)).
<output of compile/1>
yes

| ?- open(fred, write, OutputStream),
     open(fred, read,  InputStream ),
     delete_file(fred),
     format(OutputStream, 'foo.~n', []),
     flush_output(OutputStream),
     read(InputStream, Term),
     close(OutputStream),
     close(InputStream).

OutputStream = '$stream'(10,3),
InputStream = '$stream'(11,4),
Term = foo
```

`rename(+`*`OldName`*`, +`*`NewName`*`)`

This command is identical to `dec10_rename/2` below.

`dec10_rename(+`*`OldName`*`, +`*`NewName`*`)`

This predicate is similar, but not identical, to the DEC-10 Prolog/C-Prolog command `rename/2`, and is provided solely for the sake of compatibility. If you are converting existing DEC-10 Prolog or C-Prolog code to Quintus Prolog, the fact that `rename/2` does close the file and is sensitive to the `fileerrors` flag should be useful. In new programs we recommend the use of `rename_file/2`

and `delete_file/1`. *OldName* and *NewName* must be atoms, otherwise an error is reported and the command fails (this is not affected by the setting of the `fileerrors` flag). If *NewName* is `[]`, the file named by *OldName* is deleted, otherwise it is renamed to *NewName*. If the rename cannot be performed, what happens next depends on the setting of the `fileerrors` flag (see the reference page for `prolog_flag/3`). If `fileerrors` is `on`, an error exception is raised. If `fileerrors` is `off`, the command fails quietly. Examples:

```
| ?- prolog_flag(fileerrors, Setting).

Setting = on

| ?- dec10_rename(2, 1).
! Type error in argument 1 of dec10_rename/2
! symbol expected, but 2 found
! goal:  dec10_rename(2,1)
| ?- dec10_rename(does_not_exist, []).
! Existence error in dec10_rename/2
! file does_not_exist does not exist
! O/S error : No such file or directory
! goal:  dec10_rename(does_not_exist,[])

| ?- prolog_flag(fileerrors, _, off).

yes
| ?- dec10_rename("old", "new").
! Type error in argument 1 of dec10_rename/2
! symbol expected, but [111,108,100] found
! goal:  dec10_rename([111,108,100],[110,101,119])

| ?- dec10_rename(does_not_exist, []).

no
```

12.7.4 Checking To See If A File Exists

No matter what programming language you are using, all multiple-access file systems have the problem that the system may correctly report that a file *does* exist, and then when you attempt to use it you may find that it does *not*, because someone has deleted or renamed it in the meantime.

The operations below can help you avoid problems, but in the final analysis, the only way to tell whether you can open a file is to try to open it.

`file_exists(+FileName)`

FileName must be an atom. `file_exists/1` succeeds if a file of that name exists. If there is something of that name, but it is a directory, `file_exists/1` fails. You need sufficient rights to the file to be able to determine whether it is a directory (see `stat(2)`). Named pipes and devices are accepted as files.

`file_exists(+FileName, +Permissions)`

FileName must be an atom, and *Permissions* must be one of the following, or a list of them:

`exists`	does the file exist?
`read`	can the file be read?
`write`	can the file be *over*-written?
N	*N* is an integer(*N*); see `access(2)`

`file_exists/2` succeeds when there is a file (not a directory) named *FileName* and you have each of the *Permissions* you named.

If *Permission* is an integer, it is interpreted the way that the argument to the system call `access(2)` is interpreted, namely (the file must exist)

```
        + 1 * ('execute' permission is wanted)
        + 2 * ('write' permission is wanted)
        + 4 * ('read' permission is wanted)>
```

This is allowed so that a C programmer who is used to writing

```
    if (!access(FileName, 6)) {
        can_read_and_write(FileName);
    } else {
        cannot_access_file(FileName);
    }
```

can write

```
    (   file_exists(FileName, 6) ->
        can_read_and_write(FileName)
    ;    /* otherwise */
        cannot_access_file(FileName)
    )
```

We recommend, however, that you code this example as

```
    (   file_exists(FileName, [read,write]) ->
        can_read_and_write(FileName)
    ;       /* otherwise */
        cannot_access_file(FileName)
    )
```

Under operating systems that do not support version numbers (as UNIX and Windows do not), `file_exists/2` could fail (because there is no such *FileName*) and `can_open_file/2` could succeed (because you are allowed to create one). Conversely, `file_exists/2` could succeed (because there is such a *FileName*) and `can_open_file/2` fail (because you have so many files open that you cannot open another).

`file_must_exist(+`*`FileName`*`)`
succeeds when `file_exists(`*`FileName`*`)` succeeds; otherwise, it raises an error exception.

```
| ?- file_must_exist(fred).
! Existence error in file_must_exist/1
! file fred does not exist
! O/S error : No such file or directory
! goal:  file_must_exist(fred)
```

`file_must_exist(+`*`FileName`*`, +`*`Permission`*`)`
succeeds when `file_exists(`*`FileName`*`, `*`Permission`*`)` succeeds; otherwise, it raises an error exception.

```
| ?- unix(system('ls -l files.o')),
     file_must_exist('files.o', write).

-r--r--r--  1 ok       746 Jan 24 17:58 files.o
! Permission error: cannot access file 'foo.o'
! O/S error : Permission denied
! goal:  file_must_exist('foo.o',write)
```

`can_open_file(+`*`FileName`*`, +`*`Mode`*`, +`*`Quiet`*`)`
FileName is a filename. *Mode* is `read`, `write`, or `append`, just as for the `open/3` command. `can_open_file/2` fails quietly if the file cannot be opened. The *Quiet* parameter controls the raising of an error exception when the file cannot be opened: if *Quiet* is `fail`, `can_open_file/3` fails quietly, whereas if *Quiet* is `warn`, it raises an error exception. If *Mode* is

`read` *FileName* must exist and be readable

`append` *FileName* must exist and you must have permission to append to it, or *FileName* must be nonexistent in a directory in which you have permission to create a new file

`write` the same conditions apply as for `append`

This predicate actually attempts to open the file. It will, for example, create a file in order to determine whether it *can* create it. But if that happens, it immediately deletes the file again, so there should be no permanent effect on the file system.

`can_open_file(+`*`FileName`*`, +`*`Mode`*`)`
equivalent to `can_open_file(`*`FileName`*`, `*`Mode`*`, fail)`.

`open_file(+`*`FileName`*`, +`*`Mode`*`, -`*`Stream`*`)`
is the same as the built-in predicate `open/3` (which is described in the reference pages), except that it always raises an error exception if it cannot open the file, and is not sensitive to the `fileerrors` flag.

`current_dec10_stream(?FileName, ?See_or_Tell)`
is true when *See_or_Tell* is `see` and *FileName* is a file that was opened by `see(`*`FileName`*`)` and has not yet been closed, or when *See_or_Tell* is `tell` and

FileName is a file that was opened by `tell(FileName)` and has not yet been closed. It is a version of `current_stream/3`, which just tells you about the Dec-10-compatible streams. It relies on two facts: (1) *all* the streams you opened are in the `current_stream/3` table. (2) `seeing/1` (`telling/1`) return an atom if and only if the current input (output) stream was opened by `see/1` (`tell/1`), and the atom it returns is the one given to `see/1` (`tell/1`).

`close_all_streams`
: closes all the current streams except the standard streams.

None of the predicates described in this section is affected by the `fileerrors` flag. Indeed, they exist so that you can check for errors *before* they happen.

See the summary description of `library(ask)` (Section 12.9 [lib-uin], page 612) for two useful predicates that use `can_open_file/3`.

12.7.5 Other Related Library Files

Several other library files do things to or with files. This section lists those files and the file-related predicates in them. `library(ask)` and `library(directory)` are documented in this manual. The remaining files contain in-line comments, which you can read.

12.7.5.1 `library(aropen)`

`ar_open(+Archive, +Member, -Stream)`
: opens a stream reading a particular member of a UNIX archive. See `ar(1)`.

12.7.5.2 `library(ask)`

`ask_file(+Prompt, -FileName)`
: Reads *FileName* from the terminal, having prompted for it with *Prompt*. It continues prompting until a *FileName* is read for which `can_open_file(FileName, read, warn)` is true, or until an empty line is typed (in the latter case it fails quietly).

`ask_file(+Prompt, +Mode, -FileName)`
: Reads *FileName* from the terminal, having prompted for it with *Prompt*. It continues prompting until a *FileName* is read for which `can_open_file(FileName, Mode, warn)` is true, or until an empty line is typed (in the latter case it fails quietly).

12.7.5.3 `library(big_text)`

This is a package for keeping large chunks of text in files. See Section 12.13 [lib-abs], page 641.

12.7.5.4 library(crypt)

`crypt_open(+FileName, +Password, +Mode, -Stream)`
: is the same as `open(FileName, Mode, Stream)` except that the external file is encrypted, using *Password* as the key.

`crypt_open(+FileName, +Mode, -Stream)`
: prompts for *Password* and proceeds as `crypt_open/4`.

These predicates do not use any of the encryption features of the operating system, so a separate C program for managing encrypted files is included.

12.7.5.5 library(directory)

This module provides operations for finding files in directories and for finding properties of files and directories.

See Section 12.8 [lib-lfi], page 607 for details.

12.7.5.6 library(fromonto)

This package provides a suite of I/O redirection operators, which allow the user to execute a goal with the current input or output stream temporarily redirected to a specified stream, file, or list of characters. See Section 12.13 [lib-abs], page 641.

12.7.5.7 library(unix)

This package provides a set of UNIX-like commands. They take character lists as well as atoms.

```
| ?- cd.           % same as cd "~".
| ?- cd Dir.       % Dir is atom or chars.

| ?- csh.          % runs an interactive /bin/csh
| ?- csh Cmd.      % interprets Cmd with /bin/csh
| ?- ls.           % runs /bin/ls with no arguments.

| ?- pg F.         % same as sh('/usr/bin/pg F').

| ?- sh.           % runs an interactive /bin/sh
| ?- sh Cmd.       % interprets Cmd with /bin/sh
| ?- shell.        % runs an interactive $SHELL
| ?- shell Cmd.    % interprets Cmd with $SHELL
```

12.8 Looking Up Files

12.8.1 Introduction — `library(directory)`

For the most part, Prolog programs have little need to examine directories or to inquire about file properties. However, the need does occasionally arise. For example, an expert-system shell might offer the option of either loading a single file into its knowledge base, or of loading all the files in a directory having a particular extension. The Quintus Prolog library file `library(directory)` provides the tools you need to do this. For example, we might define

```
kb_load(File) :-
    (  directory_property(File, searchable) ->
          forall(file_member_of_directory(File,'*.kb',_,Full),
                   kb_load(Full))
    ;  file_property(File, readable) ->
          kb_load_file(File)
    ;  format(user_error, '~N! cannot read ~w.~n', [File]),
       fail
    ).
```

The routines in this package were designed to be a complete toolkit for safely wandering around a UNIX-like file system. Although there are quite a few of them, they do actually fit together in a coherent group. For information on operations relating to individual files rather than to directories, see `library(files)` (Section 12.7 [lib-ofi], page 599).

The following principles have been observed:

- An absolute distinction is drawn between files and directories. The set of operations one can usefully perform on a directory is different from the set one can perform on a file: for example, having write permission to a directory allows the user to create new files in it, not to rewrite the entire directory! If any routine in this package tells you that a "file" exists, you can be sure that it means a "regular" file.
- The directory scanning routines do not actually open the files they find. Thus finer discriminations, such as that between source and object code, are not made.
- The predicate names are made up of complete English words in lowercase, separated by underscores, with no abbreviations.
- Every predicate acts like a genuine logical relation insofar as it possibly can.
- Like those in the `library(unix)` package, if anything goes wrong, the predicates in `library(directory)` raise an error exception. Any time that a predicate fails quietly, it should mean "this question is meaningful, but the answer is no"; any exception to this should be regarded as a bug.
- The directory scanning routines insist that the directory argument name a searchable directory. But the "property" routines are to be read as "there exists a thing of such a type with such a property", and quietly fail if there is no such file or directory.

12.8.2 Finding Files in Directories

The basic directory scanning routine is

`file_member_of_directory(?Directory, ?FileName, ?FullName)` is true when

1. *Directory* is an atom naming a directory;
2. *FileName* is an atom conforming to the rules for file names without directory components;
3. *Directory* contains an entry with name *FileName*, and the current process is allowed to know this fact; and
4. *FullName* is an atom naming the file, combining both the *Directory* and the *FileName*, and *FullName* names a regular file.

Directory can be an absolute filename or a relative one. *FullName* will be absolute if *Directory* is absolute, relative if *Directory* is relative.

We return the *FileName* component because that is the component on which pattern matching is generally done. We return the *FullName* component in order to remove from the user the burden of manipulating the (system-dependent) rules for putting together a directory name and a file name.

This predicate acts as much like a logical relation as it can. Here are some of the ways of using it:

```
| ?- file_member_of_directory(foo, Name, Full),
     write(Name=Full), nl, fail.
  %  to enumerate members of the directory

| ?- file_member_of_directory(baz, 'ugh.pl', Full).
  %  to test whether a file 'ugh.pl' is visible in
  %  directory 'baz', and if so return the full name

| ?- file_member_of_directory(Dir, Nam, 'baz/jar.log').
  % if there is a visible regular file baz/jar.log,
  % to return its directory in Dir and name in Nam.
```

`file_member_of_directory/3` has two variants:

`file_member_of_directory(?FileName, ?FullName)`
: is the same as `file_member_of_directory/3`, except that it checks the current directory. You could obtain this effect quite easily by calling `file_member_of_directory/3` with first argument '`.`', but in other operating systems the current directory is denoted differently. This provides an operating-system-independent way of searching the current directory. There is one other difference, which is of great practical importance: '`.`' is a *relative* directory name, but `file_member_of_directory/2` uses the *absolute* name for the current directory, so that the *FullName* you get back will also be absolute. See the description of `absolute_file_name/2` in the reference pages. Note the difference between calling

```
absolute_file_name(FileName, FullName)
```

and calling

```
file_member_of_directory(FileName, FullName)
```

The former will accept any filename, but the *FileName* must be instantiated. The latter will only accept simple file names with no directory component, and insists that the file must already exist, but in return will generate *FileName*.

`file_member_of_directory(?Directory, ?Pattern, ?FileName, ?FullName)`
is the same as `file_member_of_directory/3`, except that it filters out all the *FileName*s that do not match *Pattern*. *Pattern* is an atom that may contain '`?`' and '`*`' wildcards. '`?`' matches any character and '`*`' matches any sequence of characters (cf. UNIX `csh(1)` and `sh(1)`). The main use for this routine is to select files with a particular extension. Thus,

```
| ?- file_member_of_directory(foo,'*.pl',Short,Full).
```

matches files '`foo/*.pl`'.

To summarize, the three routines discussed so far are

```
file_member_of_directory([Directory, [Pattern, ]]Short, Full)
```

They enumerate *FileName*-*FullName* pairs one at a time: in alphabetic order, as it happens.

There is another set of three predicates finding exactly the same solutions, but returning them as a *set* of *FileName*-*FullName* pairs. We follow here the general convention that predicates that return one "thing" have '`thing`' in their name, and predicates that return the *set* of "things" have '`things`' in their name.

`file_members_of_directory([?Directory, [?Pattern, ]]?Set)`
unifies *Set* with a list of *FileName*-*FullName* pairs that name visible files in the given *Directory* matching the given *Pattern*. Thus, instead of

```
| ?- file_member_of_directory(foo, '*.pl', S, F).

S = 'bat.pl',
F = 'foo/bat.pl' ;

S = 'fly.pl',
F = 'foo/fly.pl' ;

no
```

one would find

```
| ?- file_members_of_directory(foo, '*.pl', Set).

Set = ['bat.pl'-'foo/bat.pl', 'fly.pl'-'foo/fly.pl']
```

12.8.3 Finding Subdirectories

Corresponding to the above set of six predicates for matching files in a particular directory, there is another set of six for matching subdirectories. They have the forms

`directory_member_of_directory([`*`?Directory`*`, [`*`?Pattern`*`, ]]`*`?Short`*`, `*`?Full`*`)`
`directory_members_of_directory([`*`?Directory`*`, [`*`?Pattern`*`, ]]`*`?Set`*`)`

They are exactly like the '`file_member...`' predicates in every way, except that they insist that the files thus located should instead be proper subdirectories of *Directory*. This means that not only should *Full* name a directory, but also *Short* should *not* be '`.`' or '`..`'. The reason for this is to allow you to easily write routines that explore an entire directory tree, as in

```
explore(Directory, FullName) :-
   file_member_of_directory(Directory, _, FullName).
explore(Directory, FullName) :-
   directory_member_of_directory(Directory, _, SubDir),
   explore(SubDir, FullName).

| ?- explore(., FullName), write(FullName), nl, fail.
```

If the self ('`.`') and parent ('`..`') entries were not concealed from the search, this code would go into an infinite loop exploring '*`Directory`*`/./././././././.`' and so on. Note that this does not preclude using '`.`' and '`..`' in the *Directory* name itself.

12.8.4 Finding Properties of Files and Directories

Once you have obtained a file or directory name, you can ask about the properties of that file or directory. The set of properties available is inherently operating-system-dependent. This section describes the facilities currently available under UNIX and the restrictions in the Windows version.

Properties fall into several classes. The current classes are

`boolean` having values `true` and `false`

`integer` having non-negative integer values

In the Windows version no '`_id`' information, such as `user_id`, is obtained.

`who` (UNIX only) values are sub*sequences* of `[user,group,other]` — that is, the order of the elements must be preserved. Sub*sets* will not do — `[other,user]` is not a possible value.

`date` values are `date(`*`Year`*`, `*`Month`*`, `*`Day`*`)` terms. The arguments are this way round so you can use them for sorting.

`time` values are `date(`*`Year`*`, `*`Month`*`, `*`Day`*`, `*`Hour`*`, `*`Minute`*`, `*`Second`*`)` terms, with *Hour* on a 24-hour clock. The arguments are in this order so you can use them for sorting. The times given are *local* times, not GMT times. See `ctime(3)`.

`user` (UNIX only) values are user names

`group` (UNIX only) values are group names

The properties, with their types, are

```
readable        : boolean
writable        : boolean
executable      : boolean       file only
searchable      : boolean       directory only
set_user_id     : boolean       file only
set_group_id    : boolean       file only
save_text       : boolean       file only
only_one_link   : boolean       file only
who_can_read    : who
who_can_write   : who
who_can_execute : who           file only
who_can_search  : who           directory only
access_date     : date
modify_date     : date
create_date     : date
access_time     : time
modify_time     : time
create_time     : time
owner_user_name : user
owner_group_name: group
owner_user_id   : integer
owner_group_id  : integer
number_of_links : integer       file only
size_in_bytes   : integer       file only
size_in_blocks  : integer       file only
block_size      : integer       file only
```

The properties `readable`, `writable`, `executable`, and `searchable` ask the question "can this process do such-and-such to the file". For more information, see `access(2)`. For more information on the other properties, see `stat(2)`.

The basic routine for determining the properties of files is

`file_property(+`*`File`*`, `*`?Property`*`, `*`?Value`*`)`
: `file_property/3` is true when *File* is the name of a visible regular file, *Property* is one of the properties listed in the table above, other than those specific to directories, and *Value* is the actual value of *Property* for that *File*. *File* must be specified; there may be tens of thousands of files with a particular

attribute! However, you can enumerate all the properties of a given *File* (by leaving *Property* uninstantiated) if you like.

`file_property/3` has the following variant:

`file_property(+File, ?Property)`
: is only allowed when *Property* is a boolean property, and is otherwise equivalent to the call

```
file_property(File, Property, true)
```

(Note that if there is a user with login name `true`, `file_property(F,owner_user_name,true)` is possible, but `file_property(F,owner_user_name)` is not; the *Property* really must be a boolean property, not just have `true` as its value.) In particular, a quick way to check whether *File* names a file that this process can read is to ask

```
file_property(File, readable)
```

See also `can_open_file/3` in `library(files)`. Note that

```
file_property(File, writable)
```

asks whether a writable *File* already exists; if you want to know whether `open(File, write, Stream)` will be allowed, use `can_open_file/3` (see Section 12.7 [lib-ofi], page 599).

To match these two predicates, which access properties of files, there are two predicates for asking about the properties of directories:

- `directory_property(?Directory, ?Property, ?Value)`
- `directory_property(?Directory, ?Property)` % boolean only

12.8.5 Summary

`library(directory)` provides Prolog routines for scanning directories and checking properties of files. See also `absolute_file_name/2` in the reference pages, and `library(files)`.

12.9 Obtaining User Input

12.9.1 Introduction

Quintus Prolog, DEC-10 Prolog, SICstus Prolog, and other similar Prolog systems offer only two methods of input:

1. reading Prolog terms using `read/1`
2. reading single characters using `get0/1`

There is a large gap between the two, and sometimes the input requirements of application programs lies in the gap. The Prolog library contains two sets of packages to fill the gap.

- `library(readin)` and `library(readsent)` are for reading English sentences. They return a list of words, which you can then parse using a Definite Clause Grammar (built into the Prolog system).
- `library(ctypes)`, `library(prompt)`, `library(readconst)`, `library(continued)`, `library(lineio)`, and `library(ask)` are more general in purpose.

12.9.2 Classifying Characters — `library(ctypes)`

One of the problems facing anyone who uses Prolog on more than one system is that different operating systems use different characters to signal the end of a line or the end of a file. We have

Dialect	DEC-10 Prolog	SICStus Prolog	Quintus Prolog
OS	(TOPS-10)	(UNIX,Windows)	(UNIX,Windows)
end-of-line	31 (^_)	10 (LF, ^J)	10 (LF, ^J)
end-of-file	26 (^Z)	-1	-1

Windows note: From an application program's point of view, each line in the file is terminated with a single ⟨LFD⟩. However, what's actually stored in the file is the sequence ⟨RET⟩⟨LFD⟩.

A prudent Prolog programmer will try to avoid writing these constants into his program. Indeed, a prudent Prolog programmer will try to avoid relying too much on the fact that Prolog uses the ASCII character set.

Quintus Prolog addresses these problems by imitating the programming language C. The package `library(ctypes)` defines predicates that recognize or enumerate certain types of characters. Where possible, the names and the character sets have been borrowed from C.

Except as indicated, all of the predicates in `library(ctypes)` check the type of a given character, or backtrack over all the characters of the appropriate type if given a variable.

`is_endfile(-`*Char*`)`

Char is the end-of-file character. There is only one such character. If `get0/1` returns it, the end of the input file has been reached, and the file should not be read further. No special significance is attached to this character on output; it might not be a valid output character at all (as in Quintus Prolog) or it might simply be written out along with other text.

The need for this predicate is largely obviated by the built-in predicate `at_end_of_file/[0,1]` in Release 3.

`is_newline(-Char)`

Char is the end-of-line character. There is only one such character. You can rely on it not being space, tab, or any printing character. It is returned by `get0/1` at the end of an input line. The end-of-line character is a valid output character, and when written to a file ends the current output line. It should not be used to *start* lines, only to *end* them.

The need for this predicate is largely obviated by the built-in predicate `skip_line/[0,1]` in Release 3.

`is_newpage(-Char)`

Char is the end-of-page character. There is at most one such character, and when it is defined at all it is the ASCII "formfeed" character. On some systems there may be no end-of-page character. This character is returned by `get0/1` at the end of an input page. It is a valid output character, and when written to a file ends the current output page. It should not be used to *start* pages, only to *end* them.

`is_endline(+Char)`

Some systems permit more than one end-of-line character for terminal input; one of them is always C's "newline" character, another is the end-of-file character (`^D` or `^Z`) if typed anywhere but as the first character of a line, and the last is the "eol" character, which the user can set with the `stty(1)` command.

`is_endline/1` accepts most ASCII control characters, but not space, tab, or delete, which covers all the line terminators likely to arise in practice. It should only be used to recognize line terminators; if passed a variable, it will raise an error exception.

The need for this predicate is largely obviated by the built-in predicate `at_end_of_line/[0,1]` in Release 3.

`is_alnum(?Char)`

is true when *Char* is the ASCII code of a letter or digit. It may be used to recognize alphanumerics or to enumerate them. Underscore '_' is *not* an alphanumeric character. (See `is_csym/1` below.)

`is_alpha(?Char)`

is true when *Char* is the ASCII code of a letter. It may be used to recognize letters or to enumerate them. Underscore '_' is *not* a letter. (See `is_csymf/1` below.)

`is_ascii(?Char)`

is true when *Char* is in the range 0..127. If *Char* is a variable, `is_ascii/1` (like most of the predicates in this section) will try to bind it to each of the acceptable values in turn (that is, it will enumerate them). Whether the end-of-file character satisfies `is_ascii/1` or not is system-dependent.

`is_char(?Char)`

is true when Char is a character code in whatever the range happens to be. (In this version: ISO 8859/1.)

`is_cntrl(?Char)`
: is true when *Char* is an ASCII control character; that is, when *Char* is the code for DEL (127) or else is in the range 0..31. Space is *not* a control character.

`is_csym(?Char)`
: is true when *Char* is the code for a character that can appear in an identifier. C identifiers are identical to Prolog identifiers that start with a letter. Put another way, *Char* is a letter, digit, or underscore. There are C compilers that allow other characters in identifiers, such as '`$`'. In such a system, C's version of `iscsym/1` will accept those additional characters, but Prolog's will not.

`is_csymf(?Char)`
: is true when *Char* is the code for a character that can appear as the first character of a C or Prolog identifier. Put another way, *Char* is a letter or an underscore.

`is_digit(?Char)`
: is true when *Char* is the code for a decimal digit; that is, a character in the range 0..9.

`is_digit(?Char, ?Weight)`
: is true when *Char* is the character code of a decimal digit, and *Weight* is its decimal value.

`is_digit(?Char, ?Base, ?Weight)`
: is true when *Char* is the code for a digit in the given *Base*. *Base* should be an integer in the range 2..36. The digits (that is, the possible values of *Char*) are 0..9, A..Z, and a..z, where the case of a letter is ignored. *Weight* is the value of *Char* considered as a digit in that base, given as a decimal number. For example,

```
is_digit(97 /* a */, 16, 10)
is_digit(52 /* 4 */, 10,  4)
is_digit(70 /* F */, 16, 15)
```

  This is a genuine relation; it may be used all possible ways. You can even use it to enumerate all the triples that satisfy the relation. Each argument must be either a variable or an integer.

`is_graph(?Char)`
: is true when *Char* is the code for a "graphic" character, that is, for any printing character other than space. The graphic characters are the letters and digits, plus

```
!  "  #  $  %  &  '  (  )  *  ;  <  =  >  ?  @
[  \  ]  ^  _  `  {  |  }  ~  +  ,  -  .  /  :
```

`is_lower(+Char)`
: is true when *Char* is the code for a lowercase letter, a..z.

`is_paren(?Left, ?Right)`
: is true when *Left* and *Right* together form one of the delimiter pairs '`(`' and '`)`', '`[`' and '`]`', or '`{`' and '`}`'.

`is_period(?Char)`

`is_period/1` recognizes each of the three punctuation marks that can end an English sentence. That is, `is_period(Char)` is true when *Char* is an exclamation point (‘`!`’), a question mark (‘`?`’), or a period (‘`.`’). Note that if you want to test specifically for a period character, you should use the goal

```
Char is "."
```

`is_print(?Char)`

is true when *Char* is any of the ASCII “printing” characters, that is, anything except a control character. All the “graphic” characters are “printing” characters, and so is the space character. When written to ordinary terminals, each printing character takes exactly one column, and Prolog code for lining up output in nice columns is entitled to rely on this. The width of a tab, and the depiction of other control characters than tab or newline, is not defined.

`is_punct(?Char)`

is true when *Char* is the code for a non-alphanumeric printing character; that is, *Char* is a space or one of the characters listed explicitly under `is_graph/1`. Note that underscore is a “punct” and so is the space character. The reason for this is that C defines it that way, and this package eschews innovation for innovation’s sake.

`is_quote(?Char)`

is true when *Char* is one of the quotation marks ‘`‘`’ (back-quote), ‘`’`’ (single-quote), or ‘`"`’ (double-quote).

`is_space(?Char)`

is true when *Char* is the code for a white space character. This includes tab (9, `^`I), linefeed (10, `^`J), vertical tab (11, `^`K), formfeed (12, `^`L), carriage return (13, `^`M), and space (32). These constitute the C definition of white space. For compatibility with DEC-10 Prolog, `is_space/1` also accepts the (31, `^_`) character.

`is_upper(?Char)`

is true when *Char* is the code for an uppercase letter, A..Z.

`is_white(?Char)`

is true when *Char* is a space or a tab. The reason for distinguishing between this and `is_space/1` is that if you skip over characters satisfying `is_space/1` you will also be skipping over the ends of lines and pages (though at least you will not run off the end of the file), while if you skip over characters satisfying `is_white/1` you will stop at the end of the current line.

`to_lower(?Char, ?Lower)`

is true when *Char* is any ASCII character code, and *Lower* is the lowercase equivalent of *Char*. The lowercase equivalent of an uppercase letter is the corresponding lowercase letter. The lowercase equivalent of any other character is the same character. If you have a string (list of character codes) X, you can obtain a version of X with uppercase letters mapped to lowercase letters and other characters left alone by calling the library routine

```
maplist(to_lower, X, LowerCasedX)
```

In normal use of `to_lower/2`, *Char* is bound. If *Char* is uninstantiated, `to_lower/2` will still work correctly, but will be less efficient. If you want to convert a lowercase letter *Kl* to its uppercase version *Ku*, *do not* use `to_lower/2`; `to_lower(Ku, 97)` has two solutions: 65 (A) and 97 (a). Use `to_upper/2` instead.

`to_upper(?Char, ?Upper)`
: is true when *Char* is any ASCII character code, and *Upper* is the uppercase equivalent of *Char*. The uppercase equivalent of a lowercase letter is the corresponding uppercase letter. The uppercase equivalent of any other character is the same character. If you have a string (list of character codes) *X*, you can obtain a version of *X* with lowercase letters mapped to uppercase and other characters left alone by calling the library routine

```
maplist(X, to_upper, UpperCasedX)
```

The System V macro `isxdigit()` is not represented in this package because `isdigit/3` subsumes it. The System V macros `_tolower()` and `_toupper()` are not represented because `to_lower/2` and `to_upper/2` subsume them.

The predicates needed for portability between operating systems are

- `is_endfile/1`
- `is_endline/1`
- `is_newline/1`
- `is_newpage/1`

Remember: `is_endfile/1` and `is_endline/1` are for recognizing the end of an input file or the end of an input line, while `is_newline/1` and `is_newpage/1` return the character that you should give to `put/1` to end a line or page of output.

12.9.3 Reading and Writing Lines — `library(lineio)`

`library(lineio)` defines some commands for reading and writing lines of text.

`get_line(-Chars, -Terminator)`
: reads characters from the current input stream until it finds an end-of-line character. *Chars* is unified with a list containing all the character codes other than the end-of-line character, and *Terminator* is unified with the end-of-line character. This allows you to check which character ended the line; in particular you should be prepared to handle the `is_endfile(Terminator)` case. When the end of a file is encountered, there may have been a partial line preceding it; so when `is_endfile(Terminator)`, *Chars* may or may not be the empty list.

  `get_line/2` is normally called with *Chars* unbound. A call to `get_line/2` with *Chars bound* will behave similarly to `get0/1` in that even if the line of characters does not unify with *Chars*, nevertheless the entire line is consumed

and is irretrievable. Thus, if you call `get_line("fred", Eol)` and the next line of input is in fact '`jim`' or '`frederica`', the entire line will have been read before the call to `get_line/2` fails. Only call `get_line/2` with *Chars* bound when you want the line to be thrown away if it does not match. For example, if you want to skip until you encounter a line containing only a single '`.`' (a convention some editors and some mailers use for the end of terminal input), you can write

```
...
skip_through_line(".")
```

where

```
skip_through_line(X) :-
        repeat,
            get_line(X, _),
        !.
```

(`skip_through_line/1` is not in the library.)

`get_line(-`*Chars*`)`
: is used for the common case in which you are uninterested in what the end-of-line character was, provided it was not end-of-file. `get_line/1` reads a whole line, just like `get_line/2`, then checks that the line terminator was not the end-of-file character, and unifies the list of character codes with *Chars*. If *Chars* is instantiated and does not match the line that is read, or if the line terminator was end-of-file, `get_line/1` fails quietly (with the same consequences regarding the loss of the non-matching text as with `get_line/2` above).

`fget_line(+`*Stream*`, ?`*Chars*`, ?`*Terminator*`)`
: like `get_line/2` except that *Stream* is specified.

`fget_line(+`*Stream*`, ?`*Chars*`)`
: like `get_line/1` except that *Stream* is specified.

`put_chars(+`*Chars*`)`
: is a generalization of `put/1`. *Chars* should be instantiated to a (possibly empty) list of character codes. The corresponding characters are written to the current output stream. If you know the characters in advance, it is better to use `write/1`; for example,

```
put_chars("they do the same thing")
```

and

```
write('they do the same thing')
```

both write exactly the same characters to the current output stream, but the latter is more efficient. Use `put_chars/1` when you already have the text you want to write as a list of character codes, `write/1` when you have the text as an atom.

`put_line(+`*Chars*`)`
: writes the list of character codes *Chars*, then writes a newline character. It produces exactly the same output that

```
put_chars(Chars), nl
```

would, but is generally more convenient. If you are reading lines from one file using `get_line/1` and writing them to another, `put_line/1` is the predicate to use.

12.9.4 Reading Continued Lines — `library(continued)`

`library(continued)` is an extension of `library(lineio)`. It defines two commands for reading continued lines.

`read_oper_continued_line(-Line)`

reads a line of text, using a convention rather like that of BCPL: an input line that ends with '<*op*, `newline`>' where *op* is a left parenthesis '`(`', left bracket '`[`', left brace '`{`', or a binary infix character from the set

```
+ * - / # = < > ^ | & : ,
```

is taken to be continued; the *op* character is included in the combined *Line*, but the newline is *not* included.

```
| ?- compile([library(printchars),library(continued)]).
          . . .
| ?- read_oper_continued_line(Line).
|: command /option1=value1,
|:    /option2=value2

Line = "command /option1=value1,   /option2=value2"

| ?- read_oper_continued_line(Line).
|: Not continued!

Line = "Not continued!"

| ?- read_oper_continued_line(Line).
|: x^2+
|: 2*x+
|: 1

Line = "x^2+2*x+1"
```

It is likely that this will not be exactly the set of characters you want to act as continuation indicators, and you may want some *op* characters retained and others discarded. That is why we make the source code available: this file is intended mainly as an example.

`read_unix_continued_line(-Line)`

uses the UNIX convention (understood by `sh`, `csh`, `cc`, and several other programs) that a line terminated by a '<`backslash, newline`>' pair is continued,

and the backslash and newline do *not* appear in the combined *Line*. For example,

```
| ?- read_unix_continued_line(Line).
|: ab\
|: cde\
|: f

Line = "abcdef"
```

The following example is an extract from '`/etc/termcap`':

```
| ?- unix(system('cat termcap-extract')).

dw|vt52|dec vt52:\
    :cr=^M:do=^J:nl=^J:bl=^G:\
    :le=^H:bs:cd=\EJ:ce=\EK:cl=\EH\EJ:cm=\EY%+ %+ :\
    :co#80:li#24:nd=\EC:ta=^I:pt:sr=\EI:up=\EA:\
    :ku=\EA:kd=\EB:kr=\EC:kl=\ED:kb=^H:
dx|dw2|decwriter II:\
    :cr=^M:do=^J:nl=^J:bl=^G:\
    :kb=^h:le=^H:bs:co#132:hc:os:

| ?- see('termcap-extract'),
|    read_unix_continued_line(Line),
|    seen.

Line = "dw|vt52|dec vt52:    :cr=^M:do=^J:nl=^J:bl=^G:
    :le=^H:bs:cd=\EJ:ce=\EK:cl=\EH\EJ:cm=\EY%+ %+ :
 :co#80:li#24:nd=\EC:ta=^I:pt:sr=\EI:up=\EA:    :ku=\E
A:kd=\EB:kr=\EC:kl=\ED:kb=^H:"
```

Note that only the backslashes at the ends of the lines have been discarded, and that the spaces at the beginning of the following lines have been retained.

12.9.5 Reading English Sentences

12.9.5.1 Overview

There are two library files for reading sentences. One of them is `library(readin)`, which defines the single predicate `read_in/1`. `library(readin)` was written to be used.

`library(readsent)` was originally written to be read and modified, as everyone has a different idea of how sentence reading should be done. Nevertheless, you may find `read_sent/1` to be quite useful.

Both sentence readers work by reading characters until some termination condition and then parsing a list of character codes using a Definite Clause Grammar. You can have any

number of grammars in one Prolog program. You will probably find that `read_in/1` does most of what you want, but if you want to do something different you may find it easier to modify `read_sent/1`.

12.9.5.2 `library(readin)`

`read_in(-`*Sentence*`)`

reads characters from the current input stream until it finds end-of-file or a sentence terminator (see `is_period/1`) at the end of a line. There may be any number of tabs and spaces between that stop and the end of the line. It then breaks the characters up into "words", where a "word" is

- a sequence of letters (see `is_alpha/1`), which is converted to lowercase and returned as a Prolog atom
- a sequence of decimal digits (see `is_digit/1`), which is returned as a Prolog integer (plus and minus signs become separate atoms).
- any graphic character other than a letter or digit, which is returned as a single-character Prolog atom

The resulting list is returned in *Sentence*. The punctuation mark that terminated the sentence *is* included in the list. Here is an example:

```
| ?- read_in(X).
|: This is an example.  An example of read-in.  In
|: it there are +00003 sentences!

X = [this,is,an,example,.,an,example,of,read,-,
     in,.,in,it,there,are,+,3,sentences,!]
```

Note that the end-of-line character, and any spaces and tabs following the sentence terminator, are consumed. It is important that the end-of-line character be consumed; otherwise subsequent prompts will behave unpredictably.

12.9.5.3 `library(readsent)`

`read_until(?`*Delimiters*`, -`*Answer*`)`

reads characters from the current input until a character in the *Delimiters* string is read. The characters are accumulated in the *Answer* string, and include the closing delimiter. The end-of-file character always acts as a delimiter, even if it is not in the list of characters you supply.

`trim_blanks(+`*RawInput*`, ?`*Cleaned*`)`

removes leading and trailing layout characters from *RawInput*, and replaces internal groups of layout characters by single spaces.

`chars_to_words(+Chars, ?Words)`

parses a list of characters (read by read_until) into a list of tokens, where a token is

X *X* a full stop or other punctuation mark

`atom(`*X*`)` *X* a sequence of letters, e.g. `atom(the)`

`integer(`*X*`)`
X a sequence of digits, e.g. `integer(12)`

`apost` ‘’’

`aposts` ‘’’s

`string(`*X*`)`
X `"..`*sequence of any*`.."`

Thus the string ‘`the "Z-80" is on card 12.`’ would be parsed as `[atom(the),string('Z-80'),atom(is),atom(on),atom(card), integer(12),'.']`. It is up to the sentence parser to decide what to do with these. Note that the final full stop, if any, is retained, as the parser may need it.

`case_shift(+`*Mixed*`, ?`*Lower*`)`
converts all the upper case letters in *Mixed* to lower case. Other characters (not necessarily letters!) are left alone. If you decide to accept other characters in words only `chars_to_atom` has to alter. See also `lower/2` in `library(caseconv)`.

`read_line(-Chars)`
reads characters up to the next newline or the end of the file, and returns them in a list, including the newline or end-of-file. Usually you want multiple spaces conflated to one, and the newline dropped. To do this, call `trim_blanks` on the result. For a routine that does not include the newline character in the result, see the predicate `get_line/1` in `library(lineio)`.

`read_sent(-`*Sentence*`)`
reads a single sentence from the current input stream. It reads characters up to the first sentence terminator (as defined by `is_period/1`) it finds, then throws characters away until it has reached the end of a line. The characters read are then broken up into “words”, where a “word” is

- a sequence of letters, which is converted to lowercase and returned as a compound term `atom(`*Word*`)`. For example, ‘`THIS`’ would be returned as `atom(this)`
- a sequence of decimal digits, which is returned as a compound term `integer(`*Value*`)`. For example, ‘`0123`’ would be returned as `integer(123)`. Plus and minus signs become separate atoms.
- a sequence of characters in double quotes, which is returned as a compound term `string(`*X*`)`, where *X* is an atom containing the characters between the quotes. Two adjacent quotes are read as one, so the input string ‘`Double "" Quote`’ is returned as `string('Double " Quote')`.
- apostrophe s (‘`'s`’) is returned as the atom `aposts`.
- apostrophe not followed by s (‘`''`’) is returned as the atom `apost`.
- any other graphic character is returned as a single-character Prolog atom.

The resulting string is returned in *Sentence*. Here is an example.

```
| ?- read_sent(X).
|: The predicate read_sent/1 accepts sentences
|: that span more than 1 line, but not lines
|: that contain +2 or more sentences.  trash trash

X = [atom(the),atom(predicate),string(read_sent),
     atom(accepts),atom(sentences),atom(that),
     atom(span),atom(more),atom(than),integer(1),
     atom(line),',',atom(but),atom(not),atom(lines),
     atom(that),atom(contain),+,integer(2),atom(or),
     atom(more),atom(sentences),.]
```

This is more unwieldy than the output of `read_in/1`, but it does mean that your parser can tell the difference between words, numbers, and strings by pattern matching rather than having to use the *meta-logical* predicates `atom/1`, `integer/1`, and so forth.

12.9.6 Yes-no Questions, and Others — `library(ask)`

The file `library(ask)` defines a set of commands for asking questions whose answer is a single character, and for asking for file names (see Section 12.7.5.2 [lib-ofi-oth-ask], page 605).

`library(ask)` uses several commands from `library(prompt)`, but if you want to use them in your program you should explicitly include the directive

```
:- ensure_loaded(library(prompt)).
```

in your program. The principal such command is `prompt/1`, which is used to print the question or prompt.

`yesno(+`*`Question`*`)`
: writes *Question* (using `write/1`) to the terminal, regardless of the current output stream, and reads an answer. The prompt is followed by '`? `', so you should not put a question mark in the question yourself. The answer is the first character typed in response; anything following on the same line will be thrown away. If the answer is *`y`* or *`Y`*, `yesno/1` succeeds. If the answer is *`n`* or *`N`*, `yesno/1` fails. Otherwise it repeats the question. The user has to explicitly type a *`y`* or *`n`* before it will stop. Because the rest of the line is thrown away, the user can type *`yes`*, *`Yes`*, *`You'd better not`*, and so forth with exactly the same effect as a plain *`y`*. If the user just presses ⟨RET⟩, that is *not* taken as *`yes`*.

`yesno(+`*`Question`*`, +`*`Default`*`)`
: is like `yesno/1` except that

  - *Default* may be an atom (the first character of whose name will be used), a string (whose first character will be used) or an ASCII code, and will be written in brackets before the question mark; and
  - if the user just presses ⟨RET⟩, *Default* will be used as the answer.

For example,

```
yesno('Do you want an extended trace', yes)
```

prints

```
Do you want an extended trace [y]? _
```

and leaves the terminal's cursor where the underscore is. If the user presses ⟨RET⟩, this call to `yesno/1` will succeed. If the user answers *`yes`* it will succeed. If the user answers *`no`* it will fail. If the first non-layout character of the user's answer is neither *`n`*, *`N`*, *`y`*, nor *`Y`*, the question will be repeated.

`ask(+`*`Question`*`, -`*`Answer`*`)`
: writes *Question* to the terminal as `yesno/1` would, and reads a single character *Answer*. *Answer* must be a "graphic" character (a printing character other than space). `ask/2` will continue asking until it is given such a character. The remainder of the input line will be thrown away.

`ask(+`*`Question`*`, +`*`Default`*`, -`*`Answer`*`)`
: uses *Default* as the default character the way that `yesno/2` does, and mentions the default in brackets just before the question mark. If the user presses carriage return, *Default* will be returned as his *Answer*. *Answer* can be instantiated, in which case the call to `ask/2` or `ask/3` will fail if the user does not give that answer. For example, `yesno/2` could (almost) have been defined as

```
yesno(Question, Default) :-
        ask(Question, Default, 0'y).
```

`ask_chars(+`*`Prompt`*`, +`*`MinLength`*`, +`*`MaxLength`*`, -`*`Answer`*`)`
: writes *Prompt* to the terminal, and reads a line of characters from it. This response must contain between *MinLength* and *MaxLength* characters inclusive, otherwise the question will be repeated until an answer of satisfactory length is obtained. Leading and/or trailing layout characters are retained in the result, and are counted when determining the length of the answer. The list of character codes read is unified with *Answer*. Note that a colon and a space (`': '`) are added to the Prompt, so don't add such punctuation yourself. The end-user can find out what sort of input is required by typing a line that starts with a question mark. Therefore it is not possible to read such a line as data. See `prompted_line/2` in `library(prompt)`.

  Examples:

```
| ?- ask_chars('Label', 1, 8, Answer).
Label: 213456789
Please enter between 1 and 8 characters.
Do not add a full stop unless it is part of the answer.
Label: four

Answer = "four"

| ?- ask_chars('Heading', 1, 30, Answer).
Heading: ?
Please enter between 1 and 30 characters.
Do not add a full stop unless it is part of the answer.
Heading:       three leading spaces

Answer = "   three leading spaces"
```

`ask_number(+`*Prompt*`, +`*Default*`, -`*Answer*`)`

writes *Prompt* on the terminal, and reads a line from it in response. If, after "garbage" characters are thrown away, the line read represents a Prolog number, that number is unified with *Answer*. The "garbage" characters that are thrown away are layout characters (including spaces and tabs), underscores '`_`', and plus signs '`+`'. For example, the input '`+ 123_456`' would be treated as if the user had typed '`123456`'. The conversion is done by `number_chars/2`. If the user entered an integer, *Answer* will be unified with an integer. If the user entered a floating-point number, *Answer* will be unified with a floating-point number. No conversion is done. If the line contains only "garbage" characters and there is a *Default* argument, *Answer* is unified with *Default*. This happens regardless of whether or not *Default* is a number. If the input is unacceptable, the question will be repeated after an explanation of what is expected. The user can type `?` for help. Examples:

```
| ?- ask_number('Pick a number', X).
Pick a number: ?
Please enter a number followed by RETURN
Pick a number: 27

X = 27

| ?- ask_number('Say cheese', X).
Say cheese:
Please enter a number followed by RETURN
Say cheese: 3 . 141 _ 593

X = 3.14159

| ?- ask_number('Your guess', '100%', X).
Your guess [100%]: 38.
Please enter a number followed by RETURN
Your guess [100%]: 38

X = 38

| ?- ask_number('Your guess', '100%', X).
Your guess [100%]: ⟨RET⟩

X = '100%'
```

ask_number(+*Prompt*, +*Lower*, +*Upper*[, +*Default*], -*Answer*)

These two predicates are a combination of `ask_between/[4,5]` and `ask_number/[2,3]`. They write the prompt to the terminal, read a line from it in response, throw away "garbage" characters, try to parse the result as a number, and check that it is between the *Lower* and *Upper* bounds. *Lower* and *Upper* may severally be integers or floating point numbers. *Answer* will be unified with an integer if the user typed an integer, with a floating-point number if the user typed a floating-point number, or with whatever *Default* happens to be if there is a *Default* and the user entered an empty line. If you want a floating-point result whatever the user typed, you will have to do your own conversion with `is/2`. Examples:

```
| ?- ask_number('Enter temperature in Fahrenheit',
                 32.0, 212.0, 77.0, Temp).
Enter temperature in Fahrenheit [77.0]: 10
Please enter a number between 32.0 and 212.0 followed
by RETURN
Enter temperature in Fahrenheit [77.0]: 68

Temp = 68
```

ask_file(+*Question*, -*Filename*)

same as `ask_file/3`.

`ask_file(+`*Question*`, +`*Mode*`, -`*FileName*`)`

writes *Question* to the terminal and reads a filename from the terminal, regardless of the current I/O streams. If the user presses ⟨RET⟩, `ask_file/3` just fails; an empty filename is taken as an indication that the user has finished entering file names. A reply beginning with a question mark will cause a brief help message to be printed (explaining that a filename is wanted, and how to enter one), and the question will be repeated. Otherwise, `ask_file/3` checks that the file can be opened in the mode specified by *Mode* (`read`, `write`, or `append`). If it is not possible to open the file in mode *Mode*, the operating system's error result is reported and the question is repeated. If it is possible to open the file in this mode, the name of the file is returned as *FileName*. However, `ask_file/3` does *not* open the file for you, it simply checks that it is *possible* to open the file. Here is an example "dialogue":

```
| ?- ask_file('Where should the cross-reference go? ',
     write, File).
Where should the cross-reference go? ?
Please enter the name of a file that can be opened
in write mode, followed by ⟨RET⟩.  To end this
operation, just type ⟨RET⟩ with no filename.
Where should the cross-reference go? call.pl

! Permission error: cannot access file 'call.pl'
! O/S error : Permission denied
! goal:  can_open_file('call.pl',write,warn)
| ?- ask_file('Where should the cross-reference go? ',
     write, File).
Where should the cross-reference go? call.xref

File = 'call.xref'

| ?- ask_file('Next file: ', read, File).
Next file: call.pl

! Permission error: cannot access file 'call.pl'
! O/S error : Permission denied
! goal:  can_open_file('call.pl',read,warn)
| ?- ask_file('Next file: ', read, File).
Next file: call.xref

! Existence error in can_open_file/3
! file call.xref does not exist
! O/S error : No such file or directory
! goal:  can_open_file('call.xref',read,warn)
```

Points to note:

- `ask_file/3` does not add a question mark and space to the prompt; you must put them in the question yourself.

- Although the first call to `ask_file/3` found that it was *possible* to open '`call.xref`' for output, it *did not* open it, so the second call to `ask_file/3` could not find any such file.

`ask_between(+Prompt, +Lower, +Upper[, +Default], -Answer)`
writes *Prompt* on the terminal, and reads a line in response. If the line read represents a Prolog integer between *Lower* and *Upper* inclusive, this line is unified with *Answer*. The line may contain only digits and perhaps a leading minus sign. If the line is empty and there is a *Default* argument, *Answer* is unified with *Default*. This happens regardless of whether *Default* is an integer or in the indicated range. If the answer read is not acceptable, the user is told what sort of answer is wanted and is prompted again. For example, after defining

```
p(X) :-
   ask_between('Number of samples',1,20,
               [none],X),
   integer(X).
```

the following conversation might take place.

```
| ?- p(X).
Number of samples [none]: ?
Please enter an integer between 1 and 20
Do not add a full stop.
Number of samples [none]: 0
Please enter an integer between 1 and 20
Do not add a full stop.
Number of samples [none]: 9

X = 9

| ?- p(X).
Number of samples [none]: ⟨RET⟩
no
```

The prompt that is printed is '*Prompt* [*Default*]: ' if there is a *Default* argument, '*Prompt*: ' otherwise, so that you can use the same prompt whether or not there is a default argument.

`ask_oneof(+Prompt, +Constants[, +Default], -Answer)`
prints *Prompt* on the terminal, and reads a line in response. *Constants* should be a list of constants (terms that are acceptable as the first argument of `name/2`). If the user's response is the full name of one of the constants, *Answer* is unified with that constant. Failing that, if the user's response is a prefix of exactly one of the constants, *Answer* is unified with that constant. If the response is just ⟨RET⟩, and there is a *Default* argument, *Answer* is unified with *Default* (which need not be a constant, nor need it be an element of *Constants*). If nothing else works, the user is told what sort of response is wanted, and is prompted again.

The prompt that is printed is '*Prompt* [*Default*]: ' if there is a *Default* argument, '*Prompt*: ' otherwise, so that you can use the same prompt whether or not there is a default argument.

You should find it straightforward to define your own simple queries using this kit. As a general rule, try to arrange things so that if the user types a question mark s/he is told what sort of response is wanted. All the queries defined in this section do that.

The commands for reading English sentences do nothing special when their input is a single question mark. Here is an example of how you can build a query from them that does something sensible in this case.

```
ask_sentence(Prompt, Sentence) :-
    repeat,
        prompt(Prompt),
        read_in(X),
        (   X = [?] ->
            format(user_output,
                'Please enter an English sentence.~n', []),
            fail
        ;   true
        ),
    !,
    Sentence = X.
```

12.9.7 Other Prompted Input — `library(prompt)`

`library(prompt)` defines several commands for reading prompted input from the terminal. In fact, `library(ask)` is built on top of this package.

`prompt(+`*Prompt*`)`
: is used by all the commands in `library(ask)` to print the prompt or question. You may find it useful in constructing your prompted input commands. If *Prompt* is any term except a list, it is written to the terminal using `write/1`; if *Prompt* is a list of terms, each element of the list is written to the terminal using `write/1`, with no additional layout or punctuation. After writing *Prompt* to the terminal, the terminal output is flushed, so that *Prompt* will appear on the terminal before the user has to type an answer. `prompt/1` ensures that *Prompt* always starts at the beginning of a line.

`prompted_char(+`*Prompt*`, -`*Char*`)`
: writes *Prompt* to the terminal, and reads a line of characters from it. The first of these characters is returned as *Char*; the rest are discarded. The original case of the character is preserved. Note that *Char* might be a newline character or the end-of-file character.

`prompted_line(+`*Prompt*`, -`*Chars*`)`
`prompted_line(+`*Prompt*`, -`*Chars*`, -`*Terminator*`)`

These predicates write *Prompt* to the terminal, regardless of the current output stream; and read a list of character codes from the terminal, regardless of the current input stream. *Prompt* is written using `write/1`; it is normally a single atom, and should never be a list of character codes. `prompted_line/3` also returns the end-of-line character, while `prompted_line/2` simply checks that the end-of-line character is not end-of-file. When you want to ask the user of your program a question, use `prompted_line/[2,3]` instead of changing I/O streams yourself, or using `ttyget0/1` and its associates. In order to ask the user of your program "Do you really want to stop?" and stop if the user says *yes* or anything else beginning with a lowercase *y*, simply write

```
conditional_halt :-
   prompted_line('Do you really want to stop? ',
                 [0'y|_]),
   halt.
```

You can use `prompted_line/[2,3]` without worrying what the current I/O streams are, or whether you need to call `ttyflush/0` or not. Also, as with `get_line/[1,2]`, an input line ends with the line terminator character. The user does *not* have to end `prompted_line/[2,3]` input with a `.`, just with ⟨RET⟩.

12.9.8 Pascal-like Input — `library(readconstant)`

`library(readconst)` provides a set of Pascal-like input commands. The commands are

- `read_constant(X)`
- `read_constant(Stream,X)`
- `read_constants([X1,...,Xn])`
- `read_constants(Stream,[X1..Xn])`
- `prompted_constant(Prompt, X)`
- `prompted_constants(Prompt, [X1,...,Xn])`
- `skip_constant`
- `skip_constant(Stream)`
- `skip_constants(N)`
- `skip_constants(Stream,N)`

The idea is that these commands consume some number of "tokens" from the input stream (for `read_constant/1`, `skip_constant/0`, `read_constants/1`, and `skip_constants/1` this is the current input stream; for `read_constant/2`, `skip_constant/1`, `read_constants/2`, and `skip_constants/2` it is the *Stream* argument; for `prompted_constant/2` and `prompted_constants/2` it is the `user_input` stream). `prompted_constant/1` and `prompted_constants/2` resemble Pascal's `readln` command; the others resemble Pascal's `read` command.

These commands skip initial layout (spaces and control characters). There are two kinds of tokens: quoted tokens and unquoted tokens.

A *quoted token* starts with a single quote (‘’’) or a double quote (‘"’). It ends with the same character that it starts with. To include a quote in such a token, write the quote twice. This is the same as Prolog with the `character_escapes` flag off. (There is currently no way of making `read_constant/1` use C-style character escapes.) A token that starts with a single quote will be returned as a Prolog atom, even if it looks like a number. Again, this is the same as Prolog. For example, ‘`5`’ will be returned as the atom `5`, not the integer 5. A token that starts with a double quote will be returned as a list of character codes. For example, ‘`"""""`’ (four double quotes) will be returned as the list [34]. Both ‘`''`’ (two single quotes) and ‘`""`’ (two double quotes) are acceptable tokens, meaning the atom with no characters in its name (`''`) and the empty list (`[]`) respectively.

The character following the closing quote of a quoted token is always discarded. This character is normally a space, tab, newline, or comma.

An unquoted token is anything else. Characters are read until a layout character or a comma is found. The comma or layout character that terminates the token is discarded. The other characters are given to the built-in predicate `name/2`, so the token will be returned as a number if it looks like a number; otherwise it will be returned as an atom. The syntax of numbers is perforce identical to the syntax of numbers in Quintus Prolog.

In both cases, we have leading layout, which is skipped, the token proper, and a terminating character, which is discarded. If, for example, the input looks like

```
fred, 1.2 ' ', last⟨LFD⟩
```

and we call ‘`read_constants([A,B,C,D])`’, the bindings will be ‘`A=fred, B=1.2, C=' ', D=last`’, and the entire line will have been consumed. But if the input looks like

```
fred, 1.2 ' ', last⟨SPC⟩⟨SPC⟩⟨LFD⟩
```

the last ⟨SPC⟩ and ⟨LFD⟩ will be left in the input stream.

The input stream can contain end-of-line comments, which begin with a percent sign (‘`%`’) just as they do in Prolog. A comment will terminate an unquoted token, and will be skipped. Suppose you want a data file that contains a number and a filename. The file could look like this:

```
% This is the data file.
137 % is the number
foobaz/other-file % is the filename
```

You could read it by calling

```
| ?- see('the-file'), read_constants([Nbr,File]), seen.
```

The following predicates are defined in `library(readconst)`:

`read_constant(-`*`Constant`*`)`
: reads a single constant from the current input stream, then unifies *Constant* with the result.

`read_constant(+`*`Stream`*`, -`*`Constant`*`)`
: reads a single constant from *Stream*, then unifies *Constant* with the result.

`read_constants(-`*`[X1,...,XN]`*`)`
: The argument must be a proper list. *N* constants are read from the current input stream, then *[X1,. . .,XN]* is unified with a list of the results.

`read_constants(+`*`Stream`*`, -`*`[X1,...,XN]`*`)`
: The second argument must be a proper list. *N* constants are read from *Stream*, then *[X1,. . .,XN]* is unified with a list of the results.

`skip_constant`
: reads a single constant from the current input stream, then throws it away. This produces the same effect as calling `read_constant(_)`, but is more efficient, as it doesn't convert the constant from character form to Prolog form before discarding it.

`skip_constant(+`*`Stream`*`)`
: reads a single constant from *Stream* and discards it. This produces the same effect as calling `read_constant(Stream, _)` but is more efficient.

`skip_constants(+`*`N`*`)`
: reads *N* constants from the current input stream and discards them. *N* must be a non-negative integer.

`skip_constants(+`*`Stream`*`, +`*`N`*`)`
: reads *N* constants from *Stream* and discards them. *N* must be a non-negative integer.

`prompted_constant(+`*`Prompt`*`, -`*`Constant`*`)`
: writes *Prompt* to the terminal (to `user_output`) and reads one constant from it (from `user_input`) in response, then unifies *Constant* with the result. This command will flush the rest of the input line after it has read *Constant*, just like the commands in `library(ask)`. Here is an example:

```
| ?- prompted_constant('Guess the magic number: ', X),
|    integer(X).
Guess the magic number: 27 is my guess

X = 27
```

  The words '`is my guess`' and the new-line are discarded.

`prompted_constants(+`*`Prompt`*`, -`*`[X1,...,XN]`*`)`
: writes a prompt to the terminal (to `user_output`) and reads *N* constants from it (from `user_input`) in response, then unifies *[X1,. . .,XN]* with a list of the results. This command will flush the rest of the input line after it has read *[X1,. . .,XN]*, just like the commands in `library(ask)`.

12.10 Interface to Math Library

12.10.1 Introduction — `library(math)`

`library(math)` is an interface to the `math(3m)` library.

All of the predicates in this module take some number of numeric arguments and yield a numeric result as their last argument. For example, the predicate call `sin(X, Result)` unifies *Result* with `sin(X)`. The predicates provided are

`sign(X, Result)`
: *Result* is +/-1, agreeing with sign of *X*.

`sign(X, Y, AbsX_SignY)`
`abs(X, Result)`
: *Result* has same type as *X*

`fabs(X, Result)`
: *Result* is a float

`hypot(X, Y, Result)`
: *Result* is `sqrt(X*X+Y*Y)`

`max(X, Y, Result)`
: If *X* and *Y* are different types,

`min(X, Y, Result)`
: *Result* may be of either type; see NOTE #1, below.

`log(X, Result)`
: natural logarithm

`log10(X, Result)`
: common logarithm; see NOTE #2, below.

`pow(X, Y, Result)`
: *Result* is *X****Y*; see NOTE #2, below.

`exp(Y, Result)`
: *Result* is e***Y*

`sqrt(X, Result)`
: *Result* is *X***0.5

`scale(X, Integer, Result)`
: *X* and *Result*) are floats, *Result* = *X**2^*N*.

`floor(+X, ?Result)`
`floor(+X, Result, Fraction)`
`ffloor(X, Result)`
: *Result* is `float(floor(X))`

`ffloor(X, Result, Fraction)`
`truncate(X, I)`
`truncate(X, I, F)`
`ftruncate(X, I)`
`ftruncate(X, I, F)`
`ceiling(+X, ?Result)`
`ceiling(+X, ?Result, Fraction)`
: like `ceiling/2`, with a fraction present.

`fceiling(X, Result)`
: *Result* is `float(ceiling(X))`

`fceiling(X, Result, Fraction)`
: like `fceiling/2`, with a fraction present.

`round(X, Result)`
`round(X, Result, Fraction)`
`fround(X, Result)`
`fround(X, Result, Fraction)`
`fremainder(X, Y, Remainder)`
`decode_float(?Number, ?Sign, ?Significand, ?Exponent)`
`j0(X, Result)`
: These are Bessel functions,

`y0(X, Result)`
: see `j0(3m)`

`j1(X, Result)`
`y1(X, Result)`
`jn(N, X, Result)`
: *N* is an integer.

`yn(N, X, Result)`
: See NOTE #3, below.

`sin(X, Result)`
`asin(X, Result)`
`sinh(X, Result)`
`asinh(X, Result)`
: See NOTE #4, below.

`cos(X, Result)`
`acos(X, Result)`
`cosh(X, Result)`
`acosh(X, Result)`
: See NOTE #4, below.

`tan(X, Result)`
`atan(X, Result)`
`atan2(X, Y, Result)`
`tanh(X, Result)`
`atanh(X, Result)`
: See NOTE #4, below.

`gamma(X, Result)`
This is really `ln(gamma(X))`; see NOTE #5, below.

Notes:

1. If an arithmetic operation like '`X+Y`' is given mixed arguments, it will convert the integer argument to a floating-point number. In this release, `max(X, Y, Max)` and `min(X, Y, Min)` do not do this. For example,

```
| ?- min(1, 2.3, X).

X = 1
```

   This is likely to change.
2. The names of these functions are copied from C. In a later release, the functions will be

   `^(X, N, Result)`
   Result is *X*^*N* (*N* integer)

   `exp(Y, Result)`
   Result is e**Y

   `exp(X, Y, Result)`
   Result is *X***Y*, as exp(*Y**log(*X*))

   `log(X, Y, Result)`
   Result is the base *X* logarithm of *Y*

   `log(Y, Result)`
   Result is the base e logarithm of *Y*
3. The Bessel functions will be moved into another file. They are not generally available on non-UNIX versions of Quintus Prolog. They are retained as is in this release for compatibility with previous releases.
4. The inverse hyperbolic trigonometric functions are not provided in most systems. We have provided C code to compute them. It is suggested that you satisfy yourself as to their accuracy before relying on them.
5. The function that UNIX calls `gamma()` is in fact the natural logarithm of the gamma function. It is not generally available on non-UNIX versions of Quintus Prolog. It is retained in this release for compatibility with previous releases.

12.11 Miscellaneous Packages

12.11.1 `library(ctr)`

`library(ctr)` provides an array of 32 global integer variables. It was written some time ago for compatibility with another dialect of Prolog. The operations provided on these variables are

`ctr_set(+`*Ctr*`, +`*N*`)`
: ctr[*Ctr*] := *N*

`ctr_set(+`*Ctr*`, +`*N*`, ?`*Old*`)`
: *Old* is ctr[*Ctr*], ctr[*Ctr*] := *N*

`ctr_inc(+`*Ctr*`)`
: ctr[*Ctr*] := ctr[*Ctr*] + 1

`ctr_inc(+`*Ctr*`, +`*N*`)`
: ctr[*Ctr*] := ctr[*Ctr*] + *N*

`ctr_inc(+`*Ctr*`, +`*N*`, ?`*Old*`)`
: *Old* is ctr[*Ctr*], ctr[*Ctr*] := ctr[*Ctr*] + N

`ctr_dec(+`*Ctr*`)`
: ctr[*Ctr*] := ctr[*Ctr*] - 1

`ctr_dec(+`*Ctr*`, +`*N*`)`
: ctr[*Ctr*] := ctr[*Ctr*] - *N*

`ctr_dec(+`*Ctr*`, +`*N*`, ?`*Old*`)`
: *Old* is ctr[*Ctr*], ctr[*Ctr*] := ctr[*Ctr*] - *N*

`ctr_is(+`*Ctr*`, ?`*Old*`)`
: *Old* is ctr[*Ctr*]

If you want to use these counters in a nestable construct, remember to reset them properly; for example,

```
count_solutions(Goal, Count) :-
        ctr_set(17, 0, Old),
        (call(Goal), ctr_inc(17), fail ; true),
        ctr_set(17, Old, X),
        Count = X.
```

This will work even if *Goal* contains a call to `count_solutions/2`, because the old counter value is saved on entry to the clause, and restored on exit. Contrast this with the following example:

```
count_solutions(Goal, Count) :-
        ctr_set(17, 0),
        (call(Goal), ctr_inc(17), fail ; true),
        ctr_set(17, X),
        Count = X.
```

In this example, if Goal contains a call to `count_solutions/2`, the inner call will clobber the counter of the outer call, and the predicate will not work.

This file is provided mainly to allow you to experience (by doing your own timing tests) that the foreign interface, *not* the database, is the tool for hacking global variables in Prolog, provided that the global variables take only constants as values.

12.11.2 `library(date)`

`library(date)` is a time-stamp package.

- Dates are records of the form `date(`*`Year`*`,` *`Month`*`,` *`Day`*`)`.
- Times are records of the form `time(`*`Hour`*`,` *`Minute`*`,` *`Second`*`)`.
- Datimes are records of the form `date(`*`Year`*`,` *`Month`*`,` *`Day`*`,` *`Hour`*`,` *`Minute`*`,` *`Second`*`)`.
- "When" values are 32-bit time-stamps representing the number of seconds since the beginning of January 1st 1970, represented as integers.

The parameter ranges are

Year	year-1900	(e.g. 1987 -> 87)
Month	0..11	(e.g. January -> 0, September -> 8)
Day	1..31	(e.g. 27 -> 27)
Hour	0..23	(e.g. midnight -> 0, noon -> 12)
Minute	0..59	
Second	0..59	

These parameter ranges are compatible with the library function `localtime(3)`. Note that the range for months is not what you might expect.

The predicates provided are:

- `now(`*`?When`*`)`
- `date(`*`-DateNow`*`)`
- `date(+`*`When`*`,` *`-DateThen`*`)`
- `time(`*`-TimeNow`*`)`
- `time(+`*`When`*`,` `-TimeThen`*`)`
- `datime(`*`-DatimeNow`*`)`
- `datime(+`*`When`*`,` *`-DatimeThen`*`)`
- `datime(`*`?Datime`*`,` *`?Date`*`,` *`?Time`*`)`
- `date_and_time(`*`-DateNow`*`,` *`-TimeNow`*`)`
- `date_and_time(+`*`When`*`,` *`-DateThen`*`,` *`-TimeThen`*`)`
- `portray_date(+`*`TimeStamp`*`)`
- `time_stamp(+`*`Format`*`,` *`-TimeStamp`*`)`
- `time_stamp(+`*`When`*`,` *`+Format`*`,` *`-TimeStamp`*`)`

For example,

```
| ?- date(X), portray_date(X).
11-Jan-90
X = date(90,0,11)
```

Note that if you want both the current date and time, you should call either `datime/1` or `date_and_time/2`. It is an error to obtain the date and time in separate calls, because midnight could intervene and put you nearly 24 hours out.

Dates and datimes are also returned by `directory_property/3` and `file_property/3` (see `library(directory)`). All these records can be compared using term comparison.

The predicates `time_stamp/[2,3]` provide a way of creating a time-stamp atom using a special kind of format string. For example,

```
| ?- time_stamp('%W, %d %M %y',Date).

Date = 'Thursday, 11 January 1990'
```

The details of the format strings are explained in a comment in the sources. Please note that, in the interests of internationalization, `time_stamp/[2,3]` are likely to be superseded in a future release by something based on the ANSI C operation `strftime(3)`. The other predicates in this package will *not* change at that time.

12.11.3 Arbitrary Expressions — `library(activeread)`

Languages such as Lisp allow you to read an expression and to evaluate it, returning a data structure. Prolog provides "evaluation" only for arithmetic expressions, and then only in certain argument positions. `library(activeread)` provides a new experimental facility for reading an arbitrary "expression" and "evaluating" it.

```
| ?- active_read(InputTerm).
```

reads a term from the current input stream. If this term has the form

```
X | Goal.
```

then *Goal* is called and *InputTerm* is unified with *X*. Otherwise, *InputTerm* is unified with the term that was read. Note that *Goal* may backtrack, in which case `active_read/1` will also backtrack.

EXAMPLES:

```
| ?- active_read(X).
|: T | append([1,2],[3,4], T).
X = [1,2,3,4]
yes
```

```
| ?- active_read(X).
|: Front+Back | append(Front, Back, [1,2,3,4]).
X = []+[1,2,3,4] ;
X = [1]+[2,3,4] ;
X = [1,2]+[3,4] ;
X = [1,2,3]+[4] ;
X = [1,2,3,4]+[] ;
no

| ?- active_read(X).
|: abort.
X = abort
yes
```

Please note: `library(activeread)` is not a module-file, but it is sufficiently small that there should be no problem with including a separate copy in each module where it is required.

12.11.4 `library(addportray)`

`library(addportray)` makes the use of `portray/1` more convenient. In DEC-10 Prolog and C Prolog, a program could contain clauses like

```
portray(X) :-
        should_be_handled_here(X),
        print_it_this_way
```

scattered through any number of files. In Quintus Prolog, this does not work, because each file will wipe out every other file's clauses for `portray/1`; in any case, a clause for `portray/1` in a module will do nothing at all, because it is `user:portray/1` that you must define. DEC-10 Prolog and C Prolog had a similar problem in that if you reconsulted a file containing such clauses, you lost all the other clauses for `portray/1`.

Now, in order to add a link to `portray/1` clauses to your program, you can do the following:

```
:- use_module(library(addportray)).

local_portray(X) :-
        should_be_handled_here(X),
        print_it_this_way(X).

:- add_portray(local_portray).
```

To cancel such a link, you can call

```
:- del_portray(local_portray).
```

Note that if you use this package, you should not define `portray/1` in any other way; otherwise, these links will be lost.

You can link to other user-defined predicates (such as `term_expansion/2`) this way too. Suppose the other predicate to be linked to is `user:`*`Pred/Arity`*. Then

```
:- add_linking_clause(Link, Pred, Arity).
```

ensures that there is a clause

```
Pred(X1,...,XArity) :- Link(X1,...,XArity).
```

in module `user`, where *Link*/*Arity* is called in the module from which `add_linking_clause/3` is called, and

```
:- del_linking_clause(Link, Pred, Arity).
```

ensures that there is no such clause. For example, you can add a case to `term_expansion/2` by adding the following directive to a module:

```
:- add_linking_clause(local_expander, term_expansion, 2).
```

12.12 Tools

12.12.1 The 'tools' Directory

12.12.1.1 Overview

The '`tools`' directory, `qplib(tools)` is a subdirectory of the library directory, (see Section 12.1 [lib-bas], page 521). The primary tools that it contains are:

`qpxref` A Prolog cross-referencer

`qpdet` determinacy checker

12.12.2 The Cross-Referencer — `qpxref`

]

]

The main purpose of the `qpxref` program is to find undefined predicates and unreachable code. It can also aid in the formation of `module/2` and `use_module/2` statements by file, and list all cross-reference for each predicate.

See Section 2.5.9 [bas-eff-xref], page 51 for detailed information on using this tool.

12.12.3 Determinacy Checker — `qpdet`

]

`qpdet` is a tool to help you write efficient, determinate code. It is intended to be used to look for unwanted nondeterminacy in programs that are intended to be mostly determinate. Unintended nondeterminacy should be eradicated because

1. it may give you wrong answers on backtracking
2. it may cause a lot of memory to be wasted

See Section 2.5.5 [bas-eff-det], page 39 for detailed information on using this tool.

12.13 Abstracts

The following abstracts are meant to describe the functionality of each package, not to serve as documentation. Whatever documentation exists is included in comments within each package. Refer to Section 12.1.3.1 [lib-bas-dlp-acc], page 526 to find out how to locate the source files if you wish to read the code comments. All the files abstracted in the following pages are found in the library directory.

`library(aggregate)`
: defines `aggregate/3`, an operation similar to `bagof/3`, which lets you calculate sums. For example, given a table `pupil(Name, Class, Age)`, to calculate the average age of the pupils in each class, one would write

```
| ?- aggregate( sum(Age)/sum(1),
                 Name^pupil(Class, Name, Age),
                 Expr),
     call(Average_Age is Expr).
```

`library(antiunify)`
: Anti-unification is the mathematical dual of unification: given two terms *T1* and *T2* it returns the most specific term that generalizes them, *T*. *T* is the most specific term that unifies with both *T1* and *T2*. A common use for this is in learning; the idea of using it that way comes from Gordon Plotkin.

  The code here is based on a routine called `generalise/5` written by Fernando Pereira. The name was changed because there are other ways of generalizing things, but there is only one dual of unification.

  `anti_unify(+Term1, +Term2, -Term)`
  : binds *Term* to a most specific generalization of *Term1* and *Term2*. When you call it, *Term* should be a variable.

`anti_unify(+`*Term1*`, +`*Term2*`, -`*Subst1*`, -`*Subst2*`, -`*Term*`)`
: binds *Term* to a most specific generalization of *Term1* and *Term2*, and *Subst1* and *Subst2* to substitutions such that

```
Subst1(Term) = Term1
Subst2(Term) = Term2
```

Substitutions are represented as lists of *Var*=*Term* pairs, where *Var* is a Prolog variable, and *Term* is the term to substitute for *Var*. When you call `anti_unify/5`, *Subst1*, *Subst2*, and *Term* should be variables.

`library(arity)`
: Provides support for Arity code translated by `arity2quintus`.

`library(aritystring)`
: provides support for Arity's string operations.

`library(aropen)`
: lets you open a member of a UNIX archive file (see UNIX `ar(1)`) without having to extract the member. You cannot compile or consult such a file, but you can read from it. This may be useful as an example of defining Prolog streams from C. Not available under Windows.

`library(arrays)`
: provides constant-time access and update to arrays. It involves a fairly unpleasant hack. You would be better off using `library(logarr)` or `library(trees)`.

`library(assoc)`
: A binary tree implementation of "association lists".

`library(avl)`
: AVL trees in Prolog.

`library(bags)`
: provides support for the data type *bag*.

`library(benchmark)`
: Users can easily obtain information about the performance of goals: time and memory requirements.

`library(between)`
: provides routines for generating integers on backtracking.

`library(big_text)`
: Defines a *big_text* data type and several operations on it. The point of this module is that when writing an interactive program you often want to display to (or acquire from) the user large amounts of text. It would be inadvisable (though possible) to store the text in Prolog's database. With this package you can store text in a file, copy text to a stream, acquire new text from a stream, and/or have Emacs edit a big text file. See the file '`big_text.txt`' in the library area for more details.

`library(bitsets)`
: Operations on sets of integers (*bitsets*). Contains analogs for most operations in `library(ordsets)`.

`library(break)`
: Prints an error message and enters a break level (if possible), avoiding the problem of `break/0` in QPC.

`library(call)`
: provides a number of predicates that are useful in programs that pass goals as arguments.

`library(caseconv)`
: is mainly intended as an example of the use of `library(ctypes)`. Here you'll find predicates to test whether text is in all lowercase, all uppercase, or mixed-case, and to convert from one case to another.

`library(charsio)`
: lets you open a list of character codes for input as a Prolog stream and, having written to a Prolog stream, collect the output as a list of character codes. There are three things you can do with `library(charsio)`:

  1. You can open an input stream reading from a (ground) list of characters. This is the predicate `chars_to_stream`.
  2. You can run a particular goal with input coming from a (ground) list of characters. The predicates `with_input_from_chars/[2,3]` do this.
  3. You can run a particular goal with output going to a list of characters (the unification is done after the goal has finished). The `with_output_to_chars/[2,3]` predicates do this.

`library(clump)`
: Group adjacent related elements of lists.

`library(count)`
: The purpose is to count predicate calls. Instead of loading a program by calling `compile/1`, use `count/1`. The program will do what it always used to, except that it may run twice as slowly. The output of `library(count)` is a file that contains a record of predicate calls, and is suitable for processing by `awk(1)` or similar utilities.

`library(critical)`
: provides a critical-region facility.

`library(crypt)`
: defines two operations similar to `open/3`:

  `crypt_open(+`*`FileName`*`[, +`*`Password`*`, +`*Mode*`, -`*Stream*`)]`

  If you do not supply a *Password*, `crypt_open/3` will prompt you for it. Note that the password *will* be echoed. If there is demand, this can be changed. The *Stream* will be clear text as far as Prolog is concerned, yet encrypted as far as the file system is concerned.

  '`encrypt.c`' is a stand-alone program (which is designed to have its object code linked to *three* names: `encrypt`, `decrypt`, and `recrypt`), and can be used to read and write files using this encryption method.

  This encryption method was designed, and the code was published, in Edinburgh, so it is available outside the USA.

`library(decons)`
: provides a set of routines for recognizing and building Prolog control structures. The only predicate that is likely to be useful is `prolog_clause(`*`Clause`*`, `*`Head`*`, `*`Body`*`)`.

`library(demo)`
: Defines the `demo file_search_path`.

`library(det)`
: Aids in determinacy checking by locating places where cuts are really necessary.

`library(environ)`
: provides access to the operating system's *environment variables*. `environ(`*`?Varname`*`, `*`?Value`*`)` is a genuine relation. Note that if you include this file in a saved state, the values of environment variables are those current when the saved state was run, not when it was saved. There is also an `argv/1` in this file, which is superseded by `unix(argv(_))`.

`library(environment)`
: Portability aid for UNIX (BSD, System V), Windows, VMS, VM/SP (CMS), MVS, MS-DOS, Macintosh.

`library(expansion)`
: This library "takes over" `term_expansion/2` and provides more powerful hooks that enable multiple, "simultaneously active" and recursive program transformations to be achieved in an effcient manner.

`library(fft)`
: Performs a fast fourier transform in Prolog. This file was written to demonstrate that a FFT could be written in Prolog with the same *O(N**log*(N))* *asymptotic* cost as in Fortran. There are several easy things that could be done to make it faster, but you would be better off for numerical calculations like this using `library(vectors)` to call a Fortran subroutine.

`library(filename)`
: Portable filename manipulation. Documentation on '`filename.txt`'.

`library(flatten)`
: provides predicates for flattening binary trees into lists.

`library(foreach)`
: defines two iteration forms.

```
forall(Generator, Test)
foreach(Generator, Test)
```

`forall/2` is the standard double-negation "there is no proof of *Generator* for which *Test* is not provable", coded as '`\+ (`*`Generator`*`, \+ `*`Test`*`)`'.

`foreach/2` works in three phases: first each provable instance of *Generator* is found, then each corresponding instance of *Test* is collected in a conjunction, and finally the conjunction is executed.

If, by the time a *Test* is called, it is always ground — apart from explicitly existentially quantified variables — the two forms of iteration are equivalent,

and `forall/2` is cheaper. But if you want *Test* to bind some variables, you must use `foreach/2`.

`library(freevars)`
: This is an internal support package. Users will probably have no direct use for it.

`library(fromonto)`
: defines some "pretty" operators for input/output redirection. Examples:

```
| ?- (repeat, read(X), process(X))
     from_file 'fred.dat'.

| ?- read(X) from_chars "example. ".

X = example

| ?- write(273.4000) onto_chars X.

X = "273.4"
```

`library(gauss)`
: Gaussian elimination.

`library(getfile)`
: defines `get_file(+FileName, -ListOfLines)`, which reads an entire file into memory in one go.

`library(graphs)`
: a collection of utilities for manipulating mathematical graphs. The collection is incomplete. Please let us know which operations in this collection are most useful to you, and which operations that you would find useful have not been included.

  The P-representation of a graph is a list of (from-to) vertex pairs, where the pairs can be in an arbitrary order. This form is convenient for input and output.

  The S-representation of a graph is a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by `keysort/2`) and the neighbors of each vertex are also in standard order (as produced by `sort/2`). This form is convenient for many calculations.

  See also `library(mst)` (Section 12.13 [lib-abs], page 641), which is soon to be merged into `library(graphs)`.

`library(heaps)`
: provides support for the data type *heap* (heaps are also known as priority queues).

`library(knuth_b_1)`
: is a table of constants taken from Appendix B1 of D.E. Knuth's *The Art of Computer Programming*, Volume 1. The point is not to provide the constants — you could have calculated them yourselves easily enough — but to illustrate the recommended way of building such constants into your programs.

`library(listparts)`
: exists to establish a common vocabulary for names of parts of lists among Prolog programmers.

`library(lpa)`
: Compatibility library for LPA Prologs. See also '`quintus.mac`', '`quintus.dec`'.

`library(logarr)`
: is an implementation of "arrays" as 4-way trees. See also `library(trees)`.

`library(long)`
: This is a rational arithmetic package.

  `rational(N)` recognizes arbitrary-precision rational numbers: this includes integers, `infinity`, `neginfinity`, & `undefined`. `whole(N)` recognizes arbitrary precision integers. `eval(Expr, Result)` evaluates an expression using arbitrary precision rational arithmetic; it does *not* accept floats at all. `{eq,ge,gt,le,lt,ne}/2` are infix predicates like `</2` that compare rationals (or integers, *not* expressions). `succ/2`, `plus/3`, and `times/3` are relational forms of arithmetic, which work on rational numbers (*not* floats). To have rational numbers printed nicely, put the command

```
:- assert((portray(X) :- portray_number(X)))
```

  in your code. See '`long.doc`' and the comments in '`long.pl`'.

`library(mapand)`
: provides mapping routines over &-trees. See also '`maplist.pl`'.

`library(maplist)`
: is built on top of `library(call)`, and provides a collection of meta-predicates for applying predicates to elements of lists.

`library(maps)`
: implements functions over finite domains, which functions are represented by an explicit data structure.

`library(menu)`
: illustrates how to drive the Emacs interface from Prolog. The sample application involves choosing items from a menu. See also the '`menu_example.pl`' program in the demo directory. Not available under Windows.

`library(mst)`
: is a preliminary version of a minimal spanning tree package, that will eventually be merged into `library(graphs)`.

  `library(mst)` currently provides two predicates:

  `first_mst(+Nodes, +Cost, -Root, -MST)`
  `mst(+Nodes, +Cost, -Root, -MST)`

  - *Nodes* is a list of nodes.
  - *Cost* is a predicate that takes three extra arguments. A predicate from `library(call)`, `call(Cost, X, Y, Dist)` calculates the distance *Dist* between nodes *X* and *Y*.

- *Root* is the root of a minimal spanning tree and *MST* is a list of the arcs in that minimal spanning tree.

Please note: `mst/4` has been carefully written so that it will find all the minimal spanning trees of a graph. `mst/4` finds many trees, especially as it is blind to redundant representations of isomorphic trees. If you will be satisfied with any MST at all, use `first_mst/4` instead. `first_mst/4` will try to keep the arcs in the same order as the nodes if at all possible.

`library(multil)`
: provides multiple-list routines.

`library(newqueues)`
: provides support for the *queue* data type. The `library(newqueues)` package replaces `library(queues)`, and should be used in new programs.

`library(nlist)`
: Interface to the UNIX library function `nlist(3)`. Not available under Windows.

`library(note)`
: The built-in predicates and commands pertaining to the "recorded" (or "internal") database have an argument called the "key". All that matters about this key is its principal functor. That is, `fred(a, b)` and `fred(97, 46)` are regarded as the same key.

  `library(note)` defines a complete set of storing, fetching, and deleting commands where the "key" is a ground term *all* of which is significant, using the existing recorded database. Note that this package is no better indexed than the existing recorded database.

`library(order)`
: The usual convention for Prolog operations is INPUTS before OUTPUTS. The built-in predicate `compare/3` violates this. This package provides an additional interface to provide comparison predicates with the usual order. The package contains predicates to compare numbers, terms, sets and ordered lists.

`library(ordered)`
: is a collection of predicates for doing things with a list and an ordering predicate. See also `library(ordsets)` (Section 12.2.7 [lib-lis-ordsets], page 547), `library(ordprefix)` below, and `library(samsort)` (Section 12.13 [lib-abs], page 641).

`library(ordprefix)`
: is for extracting initial runs from lists, perhaps with a user-supplied ordering predicate. See also `library(ordered)` above.

'`quintus.mac`'
: version of '`lpa.pl`' to be used on Mac.

'`quintus.dec`'
: version of '`lpa.pl`' to be used on DEC.

`library(pipe)`

Quintus streams may be connected to pipes using `library(pipe)`, which provides a single predicate:

`popen(+`*`Command`*`, +`*`Mode`*`, -`*`Stream`*`)`

Mode may be either:

`read` *Stream* will be bound to a new input stream, connected to the standard output of the *Command*. The standard input stream of the *Command* is left the same as the standard input stream of Prolog. So we have

`user_input -> `*`Command`*` -> `*`Stream`*

`write` *Stream* will be bound to a new output stream, connected to the standard input of the *Command*. The standard output stream of the *Command* is left the same as the standard output stream of Prolog. So we have

`Stream` `-> `*`Command`*` -> user_output`

The behavior of `popen/3` is defined by the system function `popen(3S)`. There is no special `pclose/1` command: the existing `close/1` will call `pclose(3S)`. Commands are executed by `sh(1)` under UNIX and by the default command interpreter under Windows, e.g. `cmd.exe` under Windows XP. Under Windows, the underlying `popen()` C library function, and therefore also `popen/3`, only works in console applications, e.g. in `prolog` but not in `qpwin`.

`library(plot)`

This package generates UNIX `plot(5)` files.

`library(pptree)`

This file defines pretty-printers for (parse) trees represented in the form

```
<tree> --> <node label>/[<son>,...<son>]
         |  <leaf label>          -- anything else
```

Two forms of output are provided: a human-readable form and a Prolog term form for reading back into Prolog.

`pp_tree(+`*`Tree`*`)`

prints the version intended for human consumption, and

`pp_term(+`*`Tree`*`)`

prints the Prolog-readable version.

There is a new command `ps_tree/1`, which prints trees represented in the form

```
<tree> --> <node label>(<son>,...,<son>)
         |  <leaf>                -- constants
```

The output of `ps_tree/1` is readable by Prolog and people both. You may find it useful for things other than parse trees.

`library(printchars)`
: extends `portray/1` (using `library(addportray)`) so that lists of character codes are written by `print/1`, by the top level, and by the debugger, between double quotes.

```
| ?- X = "fred".

X = [102,114,101,100]

| ?- use_module(library(printchars)),
     X = "fred".

X = "fred"
```

`library(printlength)`
: provides predicates for determining how wide a term would be if written.

`library(putfile)`
: Uses C stream functions to copy the contents of a file to the the current output stream. This is the fastest known method for copying the contents of a file to the current output stream.

`library(qerrno)`
: Defines error codes specific to Quintus Prolog, which do not have any standard errno assignment.

`library(qsort)`
: provides a stable version of quicksort. Note that quicksort is *not* a good sorting method for a language like Prolog. If you want a good sorting method, see `library(samsort)` below.

`library(queues)`
: provides support for the *queue* data type. This library has been made obsolete in release 3 by the introduction of `library(newqueues)`. It is retained for backward compatibility, but should not be used in new programs.

`library(random)`
: provides a random number generator and several handy interface routines. The random number generators supplied by various operating systems are all different. It is useful to have a random number generator that will give the same results in all versions of Quintus Prolog, and this is the one.

`library(ranstk)`
: This is a Prolog implementation of the algorithms in Eugene W. Myers' *An Applicative Random-Access Stack*.

`library(read)`
: This code was originally written at the University of Edinburgh. David H. D. Warren wrote the first version of the parser. Richard A. O'Keefe extracted it from the Dec-10 Prolog system and made it use only user-visible operations. He also added the feature whereby '`P(X,Y,Z)`' is read as `call(P,X,Y,Z)`. Alan

Mycroft reorganized the code to regularize the functor modes. This is easier to understand (there are no more '?'s), and it also fixes bugs concerning the curious interaction of cut with the state of parameter instantiation. O'Keefe then took it over again and made a number of other changes.

There are three intentional differences between this library and the Dec-10 Prolog parser:

- "Predicate variables" serve as syntactic saccharine for `call/N`.
- When there is a syntax error, DEC-10 Prolog will backtrack internally and read the next term. This fails. If you call `portable_read/1` with an uninstantiated argument, failure means a syntax error. You can rely on it.
- '`, ..`' is not accepted in place of '`|`'. This was always a parser feature, not a tokeniser feature: any amount of layout and commentary was allowed between the '`,`' and the '`..`'. It wouldn't be hard to allow this again.

`library(retract)`

This file adds more predicates for accessing dynamic clauses and the recorded database. The built-in predicate `retract/1` will backtrack through a predicate, expunging each matching clause until the caller is satisfied. *This is not a bug.* That is the way `retract/1` is *supposed* to work. But it is also useful to have a version that does not backtrack.

`library(retract)` defines, among many other commands, `retract_first/1`, which is identical to `retract/1` except that it expunges only the first matching clause, and fails if asked for another solution.

`library(samsort)`

provides a stable sorting routine, which exploits existing order, both ascending and descending. (It is a generalization of the natural merge.) `samsort(`*`Raw`*`, `*`Sorted`*`)` is like `sort(`*`Raw`*`, `*`Sorted`*`)` except that it does not discard duplicate elements. `samsort(`*`Order`*`, `*`Raw`*`, `*`Sorted`*`)` lets you specify your own comparison predicate, which the built-in sorting predicates `sort/2` and `keysort/2` do not. This file also exports two predicates for merging already-sorted lists: `merge/3` and `merge/4`. See also `library(ordered)` and `library(qsort)`.

`library(setof)`

provides additional predicates related to the built-in predicate `setof/3`. Note that the built-in predicates `bagof/3` and `setof/3` are much more efficient than the predicates in this file. See also `library(findall)`.

`library(show)`

The built-in command `listing/1` displays dynamic predicates. But there is no built-in command for displaying the terms recorded under a given key. `library(show)` defines two predicates: `show(`*`Key`*`)` displays all the terms recorded under the given *Key*, and `show/0` displays all the *Keys* and terms in the recorded database.

`library(showmodule)`

provides a command for displaying information about a loaded module. `show_module(`*`Module`*`)` prints a description of the *Module*, what it exports, and what it imports. The command

```
| ?- show_module(_), fail ; true.
```

will print a description of every loaded module. To backtrack through all current modules and print information about the predicates they define, import, and export, use

```
| ?- ensure_loaded(library(showmodule)),
     show_module(Module).
```

To print information about a particular module *m*, use

```
| ?- show_module(m).
```

`library(statistics)`

The `full_statistics/[0,2]` predicates are exactly like the built-in `statistics/[0,2]` predicates except that

- Integers are written out with commas every three digits.
- The number of page faults is reported (if known).

`library(stchk)`

This package allows local style-check modifications in a file. This module provides an alternative interface to the style check flags. The idea is that a file that uses it will look like

```
<usual heading>

:- push_style.
:- set_style(StyleFlag, Value).
...

<clauses>

:- pop_style.
```

Some combination of this with the existing style check interface will be safe: no matter what style check changes are made, the original values will be restored.

The initial state (assumed) is that all checks are ON.

`library(terms)`

The foreign code interface provides means of passing constants between Prolog and C, FORTRAN, Pascal, etc.

`library(terms)` lets you pass copies of terms from Prolog to C, and receive copies of terms from C. For example, the new built-in predicate `copy_term/2` could have been defined this way:

```
'copy term'(Term, Copy) :-
        prolog_to_c(Term, Pointer_to_C_version),
        c_to_prolog(Pointer_to_C_version, Temp),
        erase_c_term(Pointer_to_C_version),
        Copy = Temp.
```

The C code in '`terms.c`' is just as much a part of this package as the Prolog code. In particular, the comments in that file describe the representation used

on the C side of the interface and there are routines and macros (see '`terms.h`') for accessing terms-in-C.

`library(termdepth)`
: Many resolution-based theorem provers impose a depth bound on the terms they create — not least to prevent infinite loops. `library(termdepth)` provides predicates that find the depth, length and size of a term, which can even be used on cyclic terms.

`library(tokens)`
: This package is a public-domain tokeniser in reasonably standard Prolog. It is meant to complement the library READ routine. It recognizes Dec-10 Prolog with the following exceptions:

  - '`%(`' is not accepted as an alternative to '`{`'
  - '`%)`' is not accepted as an alternative to '`}`'
  - '`,..`' is not accepted as an alternative to '`|`'
  - large integers are not read in as `xwd(`*`Top18Bits`*`,`*`Bottom18Bits`*`)`
  - After a comma, '`(`' is read as ' `(`' rather than '`(`'. This does the parser no harm at all, and the Dec-10 tokeniser guarantees never to return '`(`' except immediately after an atom, yielding ' `(`' everywhere else.

  BEWARE: this file does not recognize floating-point numbers.

`library(trees)`
: is an implementation of arrays as binary trees.

`library(types)`
: This file is support for the rest of the library, and is not really meant for general use. The type tests it defines are almost certain to remain in the library or to migrate to the system. The error checking and reporting code is certain to change. The library predicates `must_be_compound/3`, `must_be_proper_list/3`, `must_be_var/3`, and `proper_list/1` are new in this release.

`library(update)`
: provides utilities for updating "database" relations.

`library(vectors)`
: The Quintus Prolog foreign code interface provides means of passing scalars between Prolog and C, FORTRAN, Pascal, etc.

  `library(vectors)` provides routines you can use to pass one-dimensional numeric arrays between Prolog and C, Pascal, or FORTRAN. See the comments in the code. Briefly,

  `list_to_vector(+`*`ListOfNumbers`*`, +`*`Type`*`, -`*`Vector`*`)`
  : creates a vector, which you can pass to C. C will declare the argument as *Type**, and Prolog will declare the argument as `+address(`*`Type`*`)`. FORTRAN will declare the argument as an array of *Type*.

  `make_vector(+`*`Size`*`, +`*`Type`*`, -`*`Vector`*`)`
  : creates a vector, which the foreign routine is to fill in. C will declare the argument as *Type**, and Prolog will declare the argument as

`+address(`*Type*`)`. FORTRAN will declare the argument as an array of *Type*.

`vector_to_list(+`*Vector*`, ?`*List*`)`
extracts the elements of the *Vector* as a list of numbers; if the *Vector* contains chars or ints, the *List* will contain integers, otherwise it will contain floating-point numbers.

`kill_vector(+`*Vector*`)`
frees a vector. Don't forget to do this! You can still call `vector_to_list/2` on a dead vector, until the next time memory is allocated. All that you can really rely on is that it is safe to create some vectors, call a C routine, kill all the vectors, and then extract the contents of the interesting ones before doing anything else.

`library(writetokens)`
This package converts a term to a list of tokens. This is essentially the same as the public-domain '`write.pl`', except that instead of writing characters to the current output stream, it returns a list of tokens. There are three kinds of tokens: punctuation marks, constants, and atoms. There is nothing to indicate spacing; the point of this package is to let the caller do such formatting.

`library(xml)`
is a package for parsing XML with Prolog, which provides Prolog applications with a simple "Document Value Model" interface to XML documents.

13 The Structs Package

The `structs` package allows Prolog to hold pointers to C data structures, and to access and store into fields in those data structures. Currently, the only representation for a pointer supported by Quintus Prolog is an integer, so it isn't possible to guarantee that Prolog can't confuse a pointer with an ordinary Prolog term. What this package does is to represent such a pointer as a term with the type of the structure or array as its functor and the integer that is the address of the actual data as its only argument. We will refer such terms as *foreign terms*.

Important caveats:

You should not count on future versions of the struct package to continue to represent foreign terms as compound Prolog terms. In particular, you should never explicitly take apart a foreign term using unification or `functor/3` and `arg/3`. You may use the predicate `foreign_type/2` to find the type of a foreign term, and `cast/3` (casting a foreign term to address) to get the address part of a foreign term. You may also use `cast/3` to cast an address back to a foreign term. You should use `null_foreign_term/2` to check if a foreign term is null, or to create a null foreign term of some type.

It should never be necessary to explicitly take apart foreign terms.

13.1 Foreign Types

There are two sorts of objects that Prolog may want to handle: *atomic* and *compound*. *Atomic* objects include numbers and atoms, and *compound* objects include data structures and arrays. To be more precise about it, an atomic type is defined by one of the following. A long integer is 64 bits on DEC Alpha platforms and 32 bits on other Quintus Prolog platforms. Long integers are however truncated to 32 bits (sign-extended) by the Prolog system:

`long` long signed integer (but see above)

`integer` 32 bit signed integer

`short` 16 bit signed integer

`char` 8 bit signed integer

`unsigned_long`
long unsigned integer (but see above)

`unsigned_integer`
32 bit unsigned integer (but Prolog can only handle 31 bits unsigned)

`unsigned_short`
16 bit unsigned integer

`unsigned_char`
: 8 bit unsigned integer

`float`
: 32 bit floating-point number

`double`
: 64 bit floating-point number

`atom`
: 32 bit Prolog atom number. Unique for different atoms, but not consistent across Prolog sessions.

`string`
: long pointer to 0-terminated character array. Represented as an atom in Prolog.

`address`
: an untyped address. Like `pointer(_)`, but `structs` does no type checking for you. Represented as a Prolog integer.

`opaque`
: Unknown type. Cannot be represented in Prolog. A pointer to an opaque object may be manipulated.

And compound types are defined by one of:

`pointer(`*Type*`)`
: a long pointer to a thing of type *Type*.

`array(`*Num*`,`*Type*`)`
: A chunk of memory holding *Num* (an integer) things of type *Type*.

`array(`*Type*`)`
: A chunk of memory holding some number of things of type *Type*. This type does not allow bounds checking, so it should be used with great care. It is also not possible to use this sort of array as an element in an array, or in a struct or union.

`struct(`*Fields*`)`
: A compound structure. *Fields* is a list of *Field*_name:*Type* pairs. Each *Field* _name is an atom, and each *Type* is any valid type.

`union(`*Members*`)`
: A union as in C. *Members* is a list of *Member*_name:*Type* pairs. Each *Member* _name is an atom, and each *Type* is any valid type. The space allocated for one of these is the maximum of the spaces needed for each member. It is not permitted to store into a union (you must get a member of the union to store into, as in C).

C programmers will recognize that the kinds of data supported by this package were designed for the C language. They should also work for other languages, but programmers must determine the proper type declarations in those languages. The table above makes clear the storage requirements and interpretation of each type.

Note that there is one important difference between the `structs` package and C: the `structs` package permits declarations of pointers to arrays. A pointer to an array is distinguished from a pointer to a single element. For example

```
pointer(array(char))
```

is probably a more appropriate declaration of a C string type than

```
pointer(char)
```

which is the orthodox way to declare a string in C. Note that the `structs_to_c` tool described below does generate proper (identical) C declarations for both of these `structs` declarations.

13.1.1 Declaring Types

Programmers may declare new named data structures with the following procedure:

```
:- foreign_type
    Type_name = Type,
    ...,
    Type_name = Type.
```

Where *Type_name* is an atom, and *Type* defines either an atomic or compound type, or is a previously-defined type name.

In Prolog, atomic types are represented by the natural atomic term (integer, float, or atom). Compound structures are represented by terms whose functor is the name of the type, and whose only argument is the address of the data. So a term `foo(123456)` represents the thing of type `foo` that exists at machine address 123456. And a term `integer(123456)` represents the integer that lives in memeory at address 123456, *not* the number 123456.

For types that are not named, a type name is generated using the names of associated types and the dollar sign character (‘`$`’), and possibly a number. Therefore, users should not use ‘`$`’ in their type names.

13.2 Using Structs with QPC

The `structs` package is divided into two parts:

- the part necessary to process foreign declarations
- the part that defines all the `structs` predicates (other than the `foreign_type/2` declaration predicate).

The former file is not (usually) needed while your application is running, while the latter part certainly is. By separating `structs` into two files, you may avoid including the `structs` declaration code in your application.

In order to declare a foreign type or use foreign types in a foreign function declaration, you must first load the file `library(structs_decl)`. Ordinarily, you would probably do this by including an `ensure_loaded/1` or `use_module/[1,2,3]` call in your file. Unfortunately, this will not allow `qpc` to compile your file. In order both to use your file in the development

system, and to compile it with `qpc`, put the following line in your files that define foreign types or use foreign types in foreign function declarations:

```
:- load_files(library(structs_decl),
              [when(compile_time),if(changed)]).
```

The `when(compile_time)` tells `qpc` to load `library(structs_decl)` into `qpc`, and not to record a dependency on it. This means that `library(structs_decl)` will not be part of your statically linked application.

If you accidentally use

```
 :- ensure_loaded(library(structs_decl)).
```

it will compile in the development system, but when you `qpc` the file you will get a warning.

Files that just *use* `structs` are much simpler. Just add this to those files:

```
:- ensure_loaded(library(structs)).
```

There is another important complication. If you have type declarations in one file (call it A) that use types declared in another file (B), you must declare (at least) a compile_time dependency. So in file A, you'd need to have the line:

```
:- load_files('B', [when(compile_time),if(changed)]).
```

This does not allow predicates in A to call predicates in B. If you need this, too, you should instead include in file A the line:

```
:- load_files('B', [when(both),if(changed)]).
```

You will also need to ensure that B is compiled to a QOF file before trying to `qpc` A. This requires that if A is a module-file, so must B be. If A is not a module-file, then B need not be a module-file (but it may be). If you use the `make` utility to maintain object files, you might then want to add the following line to your '`Makefile`':

```
A.qof:      B.qof
```

13.3 Checking Foreign Term Types

The type of a foreign term may determined by the goal

```
foreign_type(+Foreign_term, -Type_name)
```

Note that `foreign_type/2` will fail if *Foreign_term* is not a foreign term.

13.4 Creating and Destroying Foreign Terms

Prolog can create or destroy foreign terms using

```
new(+Type, -Datum),
new(+Type, +Size, -Datum) and
dispose(+Datum)
```

where *Type* is an atom specifying what type of foreign term is to be allocated, and *Datum* is the foreign term. The *Datum* returned by `new/[2,3]` is not initialized in any way. `dispose/1` is a dangerous operation, since once the memory is disposed, it may be used for something else later. If *Datum* is later accessed, the results will be unpredictable. `new/3` is only used to allocate arrays whose size is not known beforehand, as defined by `array(`*Type*`)`, rather than `array(`*Type*`,`*Size*`)`.

13.5 Accessing and Modifying Foreign Term Contents

Prolog can get or modify the contents of a foreign term with the procedures

```
get_contents(+Datum, +Part, -Value)
get_contents(+Datum, *Part, *Value)
put_contents(+Datum, +Part, +Value).
```

It can also get a pointer to a field or element of a foreign term with the procedure

```
get_address(+Datum, +Part, -Value).
get_address(+Datum, *Part, *Value).
```

For all three of these, *Datum* must be a foreign term, and *Part* specifies what part of *Datum* *Value* is. If *Datum* is an array, *Part* should be an integer index into the array, where 0 is the first element. For a pointer, *Part* should be the atom `contents` and *Value* will be what the pointer points to. For a struct, *Part* should be a field name, and *Value* will be the contents of that field. In the case of `get_contents/3` and `get_address/3`, if *Part* is unbound, then `get_contents/3` will backtrack through all the valid parts of *Datum*, binding both *Part* and *Value*. A C programmer might think of the following pairs as corresponding to each other:

```
get_contents(Foo, Bar, Baz)
Baz = Foo->Bar

put_contents(Foo, Bar, Baz)
Foo->Bar = Baz

get_address(Foo, Bar, Baz)
Baz = &Foo->Bar.
```

The hitch is that only atomic and pointer types can be got and put by `get_contents/3` and `put_contents/3`. This is because Prolog can only hold pointers to C structures, not the structures themselves. This isn't quite as bad as it might seem, though, since usually structures contain pointers to other structures, anyway. When a structure directly contains another structure, Prolog can get a pointer to it with `get_address/3`.

Access to most fields is accomplished by peeking into memory (see Section 8.8.4.2 [ref-ari-aex-pee], page 236), so it is very efficient.

13.6 Casting

Prolog can "cast" one type of foreign term to another. This means that the foreign term is treated just as if it where the other type. This is done with the following procedure:

```
cast(+Foreign0, +New_type, -Foreign)
```

Foreign is the foreign term that is the same data as *Foreign0*, only is of foreign type *New_type*. *Foreign0* is not affected. This is much like casting in C.

Casting a foreign term to `address` will get you the raw address of a foreign term. This is not often necessary, but it is occasionally useful in order to obtain an indexable value to use in the first argument of a dynamic predicate you are maintaining. An `address` may also be casted to a proper foreign type.

This predicate should be used with great care, as it is quite easy to get into trouble with this.

13.7 Null Foreign Terms

"NULL" foreign terms may be handled. The predicate

```
null_foreign_term(+Term, -Type)
null_foreign_term(-Term, +Type)
```

holds when *Term* is a foreign term of *Type*, but is NULL (the address is 0). At least one of *Term* and *Type* must be bound. This can be used to generate NULL foreign terms, or to check a foreign term to determine whether or not it is NULL.

13.8 Interfacing with Foreign Code

Foreign terms may be passed between Prolog and other languages through the foreign interface.

To use this, all foreign types to be passed between Prolog and another language must be declared with `foreign_type/2` before the `foreign/[2,3]` clauses specifying the foreign functions.

The `structs` package extends the foreign type specifications recognized by the foreign interface. In addition to the types already recognized by the foreign interface, any atomic type recognized by the `structs` package is understood, as well as a pointer to any named `structs` type.

For example, if you have a function

```
char nth_char(string, n)
    char *string;
    int n;
    {
        return string[n];
    }
```

You might use it from Prolog as follows:

```
:- foreign_type cstring = array(char).

foreign(nth_char, c, nth_char(+pointer(cstring), +integer,
        [-char])).
```

This allows the predicate `nth_char/3` to be called from Prolog to determine the nth character of a C string.

Note that all existing foreign interface type specifications are uneffected, in particular `address/[0,1]` continue to pass addresses to and from Prolog as plain integers.

13.9 Examining Type Definitions at Runtime

The above described procedures should be sufficient for most needs. This module does, however, provide a few procedures to allow programmers to access type definitions. These may be a convenience for debugging, or in writing tools to manipulate type definitions.

The following procedures allow programmers to find the definition of a given type:

```
type_definition(+Type, -Definition)
type_definition(*Type, *Definition)
type_definition(+Type, -Definition, -Size)
type_definition(*Type, *Definition, *Size)
```

Type is an atom naming a type, *Definition* is the definition of that type, and *Size* is the number of bytes occupied by a foreign term of this type. *Size* will be the atom `unknown` if the size of an object of that type is not known. Such types may not be used as fields in

structs or unions, or in arrays. However, pointers to them may be created. If *Type* is not bound at call time, these procedures will backtrack through all current type definitions.

A definition looks much like the definition given when the type was defined with `type/1`, except that it has been simplified. Firstly, intermediate type names have been elided. For example, if `foo` is defined as `foo=integer`, and `bar` as `bar=foo`, then `type_definition(bar, integer)` would hold. Also, in the definition of a compound type, types of parts are always defined by type names, rather than complex specifications. So if the type of a field in a struct was defined as `pointer(fred)`, it will show up in the definition as `'$fred'`. Of course, `type_definition('$fred', pointer(fred))` would hold, also.

The following predicates allow the programmer to determine whether or not a given type is atomic:

```
atomic_type(+Type)
atomic_type(*Type)

atomic_type(+Type, -Primitive_type)
atomic_type(*Type, *Primitive_type)

atomic_type(+Type, -Primitive_type, -Size)
atomic_type(*Type, *Primitive_type, *Size)
```

Type is an atomic type. See Section 13.1 [str-fty], page 655 for the definition of an atomic type. *Primitive_type* is the primitive type that *Type* is defined in terms of. *Size* is the number of bytes occupied by an object of type *Type*, or the atom `unknown`, as above. If *Type* is unbound at call time, these predicates will backtrack through all the currently defined atomic types.

13.10 Structs to C

Included with `structs` is the program `structs_to_c`. This program reads in a Prolog file containing `structs` declarations, and generates a '`.h`' file containing equivalent declarations to be #included in your C programs. Each type you declare in your '`.pl`' file will have a corresponding `typedef` in the '`.h`' file.

If you wish to use this tool, you will have to build an executable yourself, as the default installation procedure doesn't build `structs_to_c`, in order to save space. To build `structs_to_c`, visit the `structs` library directory, and type

```
% make structs_to_c
```

This will make a host and operating system specific executable file `structs_to_c`, which you should move to an appropriate directory, for example '`/usr/local/bin`'.

13.11 Tips

1. Most important tip: don't subvert the `structs` type system by looking inside foreign terms to get the address, or use `functor/3` to get the type. This has two negative effects: firstly, if the `structs` package should change its representation of foreign terms, your code will not work. But more importantly, you are more likely to get type mismatches, and likely to get unwrapped terms or even doubly wrapped terms where you expect wrapped ones.
2. Remember that a foreign term `fred(123456)` is not of type `fred`, but a pointer to `fred`. Looked at another way, what resides in memory at address 123456 is of type `fred`.
3. The wrapper put on a foreign term signifies the type of that foreign term. If you declare a type to be `pointer(opaque)` because you want to view that pointer to be opaque, then when you get something of this type, it will be printed as `opaque(456123)`. This is not very informative. It is better to declare

```
fred = opaque,
thing = struct([...,
                part:pointer(fred),
                ...
               ]).
```

   so that when you get the contents of the `part` member of a `thing`, it is wrapped as `fred(456123)`.

14 The Quintus Objects Package

The Quintus Objects package enables programmers to write object-oriented programs in Quintus Prolog. The objects in Quintus Objects are modifiable data structures that provide a clean and efficient alternative to storing data in the Prolog database.

14.1 Introduction

The Quintus Objects package enables programmers to write object-oriented programs in Quintus Prolog. The objects in Quintus Objects are modifiable data structures that provide a clean and efficient alternative to storing data in the Prolog database.

This user's guide is neither an introduction to object-oriented programming nor an introduction to Quintus Prolog. A number of small, sample programs are described in this manual, and some larger programs are in the '`demo`' directory.

14.1.1 Using Quintus Objects

One of the basic ideas of object-oriented programming is the encapsulation of data and procedures into objects. Each object belongs to exactly one class, and an object is referred to as an instance of its class. A class definition determines the following things for its objects:

- slots, where an object holds data
- messages, the commands that can be sent to an object
- methods, the procedures the object uses to respond to the messages

All interaction with an object is by sending it messages. The command to send a message to an object has the form

```
Object MessageOp Message
```

where *Object* is an object, *MessageOp* is one of the message operators ('`<<`', '`>>`', or '`<-`') and *Message* is a message defined for the object's class. Roughly speaking, the '`>>`' message operator is used for extracting information from an object, '`<<`' is for storing information into an object, and '`<-`' is for any other sort of operation.

For example, using the point class defined in the next section, it would be possible to give the following command, which demonstrates all three message operators.

```
| ?- create(point, PointObj),
     PointObj >> x(InitX),
     PointObj >> y(InitY),
     PointObj << x(2.71828),
     PointObj << y(3.14159),
     PointObj <- print(user),
     nl(user).

(2.71828,3.14159)
PointObj = point(23461854),
InitX = 1.0,
InitY = 2.0
```

First it binds the variable `PointObj` to a newly created `point` object. Then, the two get messages (sent with the '`>>`' operator) fetch the initial values of the point's `x` and `y` slots, binding the variables `InitX` and `InitY` to these values. Next, the two put messages (sent with the '`<<`' operator) assign new values to the object's `x` and `y` slots. Finally, the send message (sent with the '`<-`' operator) instructs the point object to print itself to the `user` stream, followed by a newline. Following the goal, we see the point has been printed in a suitable form. Following this, the values of `PointObj`, `InitX`, and `InitY` are printed as usual for goals entered at the Prolog prompt.

Because this goal is issued at the Prolog prompt, the values of the variables `PointObj`, `InitX` and `InitY` are not retained after the command is executed and their values are displayed, as with any goal issued at the Prolog prompt. However, the point object still exists, and it retains the changes made to its slots. Hence, objects, like clauses asserted to the Prolog database, are more persistent than Prolog variables.

Another basic idea of object-oriented programming is the notion of inheritance. Rather than defining each class separately, a new class can inherit the properties of a more general superclass. Or, it can be further specialized by defining a new subclass, which inherits its properties. (C++ uses the phrase "base class" where we use "superclass." It also uses "derived class" where we use "subclass.")

Quintus Objects uses term expansion to translate object-oriented programs into ordinary Prolog. (This is the same technique that Prolog uses for its DCG grammar rules.) As much as possible is done at compile time. Class definitions are used to generate Prolog clauses that implement the class's methods. Message commands are translated into calls to those Prolog clauses. And, inheritance is resolved at translation time.

Quintus Objects consists of two modules, `obj_decl` and `objects`. The `obj_decl` module is used at compile time to translate the object-oriented features of Quintus Objects. Any file that defines classes or sends messages should include the command

```
:- load_files(library(obj_decl),
              [when(compile_time), if(changed)]).
```

The `objects` module provides runtime support for Quintus Objects programs. A file that sends messages or asks questions about what classes are defined or to what class an object belongs should include the command:

```
:- use_module(library(objects)).
```

You will probably include both in most files that define and use classes.

You must have a license to use the `obj_decl` module, but you may include the `objects` module in programs that you will distribute.

14.1.2 Defining Classes

A class definition can restrict the values of any slot to a particular C-style type. It can specify whether a slot is *private* (the default, meaning that it cannot be accessed except by that methods of that class), *protected* (like *private*, except that the slot can also be accessed by subclasses of the class), or *public* (meaning get and put methods for the slot are generated automatically), and it can specify an initial value. The class definition also may contain method clauses, which determine how instances of the class will respond to messages. A class definition may also specify one or more superclasses and which methods are to be inherited.

The point object created in the previous example had two floating point slots, named `x` and `y`, with initial values of 1.0 and 2.0, respectively. As we have seen, the `point` class also defined put and get methods for `x` and `y`, as well as a send method for printing the object. The put and get methods for `x` and `y` can be automatically generated simply by declaring the slots `public`, but the `print` method must be explicitly written. In addition, in order to be able to create instances of this class, we must define a `create` method, as explained in Section 14.2.3.4 [obj-scl-meth-credes], page 679. We also provide a second `create` method, taking two arguments, allowing us to specify an `x` and `y` value when we first create a point object.

```
:- class point =
        [public x:float = 1.0,
         public y:float = 2.0].

Self <- create.

Self <- create(X, Y) :-
        Self << x(X),
        Self << y(Y).

Self <- print(Stream) :-
        Self >> x(X),
        Self >> y(Y),
        format(Stream, '(~w,~w)', [X,Y]).

:- end_class point.
```

The variable name `Self` in these clauses is arbitrary—any variable to the left of the message operator in the head of a method clause refers to the instance of the class receiving the message.

14.1.3 Using Classes

Given this definition, the following command creates an instance of the point class, assigning values to its `x` and `y` slots, and prints a description of the point.

```
| ?- create(point(3,4), PointObj),
     PointObj <- print(user).
```

The print message prints `(3.0,4.0)`. The variable `PointObj` is bound to a Prolog term of the form

```
point(Address)
```

where *Address* is essentially a pointer to the object.

In general, an object belonging to a class *ClassName* will be represented by a Prolog term of the form

```
ClassName(Address)
```

The name *ClassName* must be an atom. This manual refers to such a term as if it were the object, not just a pointer to the object. Users are strongly discouraged from attempting to do pointer arithmetic with the address.

After execution of this command, the point object still exists, but the variable `PointObj` can no longer be used to access it. So, while objects resemble clauses asserted into the Prolog database in their persistence, there is no automatic way to search for an object.

Objects are not automatically destroyed when they are no longer needed. And, there is no automatic way to save an object from one Prolog session to the next. It is the responsibility of the programmer to keep track of objects, perhaps calling the `destroy/1` predicate for particular objects that are no longer needed or asserting bookkeeping facts into the Prolog database to keep track of important objects.

14.1.4 Looking Ahead

The next few sections of this manual describe the Quintus Objects package in greater detail. In particular, they describe how to define classes, their methods and their slots, and how to reuse class definitions via inheritance. Small sample programs and program fragments are provided for most of the features described.

Experienced Prolog programmers may choose to skip over these sections and look at the sample programs in this package's demo directory, referring to the reference pages as necessary. Everyone is encouraged to experiment with the sample programs before writing their own programs.

14.2 Simple Classes

This section is about simple classes that inherit nothing—neither slots nor methods—from more general superclasses. Everything about these classes is given directly in their definitions, so they are the best starting point for programming with Quintus Objects.

The use of inheritance in defining classes is described in the next section. Classes that inherit properties from superclasses are called derived classes in some systems, such as C++. In general, the use of inheritance extends the properties of the simple classes in this section.

14.2.1 Scope of a Class Definition

A simple class definition begins with a statement of the form

```
:- class ClassName = [SlotDef, ...].
```

The class's slots are described in the list of *SlotDef* terms. It is possible, though not often useful, to define a class with no slots, by specifying the empty list. In that case the '`=`' and the list may be omitted.

The class's methods are defined following the `class/1` directive, by Prolog clauses. Most of this section is about defining and using methods.

The class definition ends with any of the following:

```
:- end_class ClassName.
```

or

```
:- end_class.
```

or the next `class/1` directive or the end of the file. The *ClassName* argument to `end_class/1` must match the class name in the corresponding `class/1` directive. It is not possible to nest one class definition inside another.

14.2.2 Slots

A slot description has the form

Visibility SlotName:*SlotType* = *InitialValue*

where *Visibility* and '= *InitialValue*' are optional. Each slot of a class must have a distinct name, given by the atom *SlotName*. The *Visibility*, *SlotType* and *InitialValue* parts of the slot description are described separately.

14.2.2.1 Visibility

A slot's visibility is either private, protected, or public. If its visibility is not specified, the slot is private. The following example shows all four possibilities:

```
:- class example = [w:integer,
                    private   x:integer,
                    protected y:integer,
                    public    z:integer]
```

Slot `z` is public, `y` is protected, and both `x` and `w` are private.

Direct access to private slots is strictly limited to the methods of the class. Any other access to such slots must be accomplished through these methods. Making slots private will allow you later to change how you represent your class, adding and removing slots, without having to change any code that uses your class. You need only modify the methods of the class to accomodate that change. This is known as *information hiding*.

Protected slots are much like private slots, except that they can also be directly accessed by subclasses. This means that if you wish to modify the representation of your class, you will need to examine not only the class itself, but also its subclasses.

Public slots, in contrast, can be accessed from anywhere. This is accomplished through automatically generated get and put methods named for the slot and taking one argument. In the example above, our `example` class would automatically support a get and put method named `z/1`. Note, however, that unlike other object oriented programming languages that support them, public slots in Quintus Objects do not violate information hiding. This is because you may easily replace a public slot with your own get and put methods of the

same name. In this sense, a public slot is really only a protected slot with automatically generated methods to fetch and store its contents.

Within a method clause, any of the class's slots can be accessed via the `fetch_slot/2` and `store_slot/2` predicates. These are the only way to access private and protected slots. They may be used to define get and put methods for the class, which provide controlled access to the protected slots. But, they can only be used within the method clauses for the class, and they can only refer to slots of the current class and protected and public slots of superclasses.

In the slot description, `public`, `protected` and `private` are used as prefix operators. The `obj_decl` module redefines the prefix operator `public`, as follows:

```
:- op(600, fy, [public]).
```

Unless you use the obsolete `public/1` directive in your Prolog programs, this should cause no problems.

14.2.2.2 Types

A slot's type restricts the kinds of values it may contain. The slot is specified in the slot description by one of the following Prolog terms with the corresponding meaning. Most of these will be familiar, but the last four, `address`, `term`, *Class* and `pointer(`*Type*`)`, require some additional explanation:

Type	Description
`integer`	32-bit signed integer
`short`	16-bit signed integer
`char`	8-bit signed integer
`unsigned_short`	16-bit unsigned integer
`unsigned_char`	8-bit unsigned integer
`float`	32-bit floating point number
`double`	64-bit floating point number
`atom`	Prolog atom (32-bit pointer)
`address`	32-bit address The address type is intended for use with foreign code. A slot of this type might store an address returned from a foreign function. That address might, in turn, be used in calling another foreign function or with the `assign/2` predicate or with arithmetic operators such as `integer_at`. Hence, most Prolog programmers can safely ignore this type.

`term` Prolog term

The term type is for general Prolog terms. Such a slot can hold any of the other types. However, if you know a slot will be used to hold only values of a particular type, it is more efficient to specify that type in the class definition.

Storing a term containing free variables is similar to asserting a clause containing free variables into the Prolog database. The free variables in the term are replaced with new variables in the stored copy. And, when you fetch the term from the slot, you are really fetching a copy of the term, again with new variables.

Class where *Class* is the name of a defined class

The class type is for any object in a class defined with Quintus Objects. Such a slot holds an object of its class or one of that class's descendants, or the `null` object (see Section 14.2.2.4 [obj-scl-slt-null], page 673).

`pointer(`*Type*`)`
where *Type* is an atom

The pointer type is intended for use with the Structs Package, ProXT or ProXL. It is similar to the `address` type, except that access to this slot yields, and update to this slot expects, a term of arity 1 whose functor is Type and whose argument is the address. Again, most Prolog programmers can safely ignore this type.

Please note that there is no `unsigned_int` or `unsigned_long` type, because Prolog itself currently cannot represent such a number. You should represent such numbers as type `integer` with care. Arithmetic operations on unsigned integers represented this way will work as expected, however comparisons will not! This is inherent in using Prolog to manipulate 32 bit unsigned numbers in general; it is not specific to Quintus Objects.

14.2.2.3 Initial Values

A slot description may optionally specify an initial value for the slot. The initial value is the value of the slot in every instance of the class, when the object is first created. The initial value must be a constant of the correct type for the slot.

If an initial value is not specified, a slot is initialized to a value that depends on its type. All numbers are initialized to 0, of the appropriate type. Atom and term slots are initialized to the empty atom (`''`). Addresses and pointers are initialized to null pointers. And, objects are initialized to the `null` object (see Section 14.2.2.4 [obj-scl-slt-null], page 673).

More complicated initialization—not the same constant for every instance of the class—must be performed by create methods, which are described later.

14.2.2.4 The `null` object

The `null` object is a special object that is not an instance of any class, but that can be stored in a slot intended for any class of object. This is very much like the `NULL` pointer in C. This is useful when you do not yet have an object to store in a particular slot.

In Prolog, the `null` is represented by the atom `null`.

Note that because the `null` object is not really an object of any class, you cannot determine its class with `class_of/2`. Unless noted otherwise, when we write of an *object* in this document, we do not include the `null` object.

14.2.3 Methods

Some methods are defined by method clauses, between the `class/1` directive and the end of the class's definition. Others are generated automatically. There are three kinds of messages in Quintus Objects, distinguished by the message operator they occur with:

'`>>`' A get message, which is typically used to fetch values from an object's slots.

'`<<`' A put message, which is typically used to store values in an object's slots.

'`<-`' A send message, which is used for other operations on or involving an object.

Quintus Objects automatically generates some get and put methods. And, it expects particular message names with the send operator for create and destroy methods. For the most part, however, you are free to use any message operators and any message names that seem appropriate.

A method clause has one of these message operators as the principal functor of its head. Its first argument, written to the left of the message operator, is a variable. By convention, we use the variable `Self`. Its second argument, written to the right of the message operator, is a term whose functor is the name of the message and whose arguments are its arguments.

For example, in the class whose definition begins as follows, a 0-argument send message named `increment` is defined. No parentheses are needed in the clause head, because the precedence of the '`<-`' message operator is lower than that of the '`:-`' operator.

```
:- class counter = [public count:integer = 0].

Self <- increment :-
        Self >> count (X0),
        X1 is X0 + 1,
        Self << count (X1).
```

Its definition uses the automatically generated get and put methods for the public slot `count`.

It may look as though this technique is directly adding clauses to the `>>/2`, `<</2` and `<-/2` predicates, but the method clauses are transformed by term expansion, at compile time. However, the method clauses have the effect of extending the definitions of those predicates.

Methods are defined by Prolog clauses, so it is possible for them to fail, like Prolog predicates, and it is possible for them to be nondeterminate, producing multiple answers, upon backtracking. The rest of this section describes different kinds of methods.

14.2.3.1 Get and Put Methods

Get and put methods are generated automatically for each of a class's public slots. These are 1-argument messages, named after the slots.

In the point class whose definition begins with

```
:- class point =
        [public x:float=0,
         public y:float=0].
```

the get and put methods are automatically generated for the `x` and `y` slots. If the class defines a `create/0` method, then the command

```
| ?- create(point, PointObj),
     PointObj >>  x(OldX),
     PointObj >>  y(OldY),
     PointObj <<  x(3.14159),
     PointObj <<  y(2.71828).
```

creates a point object and binds both `OldX` and `OldY` to 0.0E+00, its initial slot values. Then, it changes the values of the `x` and `y` slots to 3.14159 and 2.71828, respectively. The variable `PointObj` is bound to the point object.

It is possible, and sometimes quite useful, to create get and put methods for slots that do not exist. For example, it is possible to add a polar coordinate interface to the point class by defining get and put methods for `r` and `theta`, even though there are no `r` and `theta` slots. The get methods might be defined as follows:

```
Self >> r(R) :-
        Self >> x(X),
        Self >> y(Y),
        R2 is X*X + Y*Y,
        sqrt(R2, R).

Self >> theta(T) :-
        Self >> x(X),
        Self >> y(Y),
        A is Y/X,
        atan(A, T).
```

This assumes that `library(math)`, which defines the `sqrt/2` and `atan/2` predicates, has been loaded. The put methods are left as an exercise.

In the rational number class whose definition begins with

```
:- class rational =
        [public num:integer,
         public denom:integer].
```

get and put methods are automatically generated for the `num` and `denom` slots. It might be reasonable to add a get method for `float`, which would provide a floating point approximation to the rational in response to that get message. This is left as an exercise.

It is also possible to define get and put methods that take more than one argument. For example, it would be useful to have a put method for the point class that sets both slots of a point object. Such a method could be defined by

```
Self << point(X,Y) :-
        Self << x(X),
        Self << y(Y).
```

Similarly, a 2-argument get method for the rational number class might be defined as

```
Self >> (N/D) :-
        Self >> num(N),
        Self >> denom(D).
```

Note that the name of the put message is `(/)/2`, and that the parentheses are needed because of the relative precedences of the '`>>`' and '`/`' operators.

Put messages are used to store values in slots. Get messages, however, may be used either to fetch a value from a slot or to test whether a particular value is in a slot. For instance, the following command tests whether the `do_something/2` predicate sets the point object's `x` and `y` slots to 3.14159 and 2.71828, respectively.

```
| ?- create(point, PointObj),
     do_something(PointObj),
     PointObj >> x(3.14159),
     PointObj >> y(2.71828).
```

The `fetch_slot/2` predicate can similarly be used to test the value of a slot.

The effects of a put message (indeed, of any message) are not undone upon backtracking. For example, the following command fails:

```
| ?- create(point, PointObj),
     PointObj << x(3.14159),
     PointObj << y(2.71828),
     fail.
```

But, it leaves behind a point object with `x` and `y` slots containing the values 3.14159 and 2.71828, respectively. In this, storing a value in an object's slot resembles storing a term in the Prolog database with `assert/1`.

Some care is required when storing Prolog terms containing unbound variables in term slots. For example, given the class definition that begins with

```
:- class prolog_term = [public p_term:term].

Self <- create.
```

the following command would succeed:

```
| ?- create(prolog_term, TermObj),
     TermObj << p_term(foo(X,Y)),
     X = a,
     Y = b,
     TermObj >> p_term(foo(c,d)).
```

The reason is that the free variables in `foo(X,Y)` are renamed when the term is stored in the `prolog_term` object's `p_term` slot. This is similar to what happens when such a term is asserted to the Prolog database:

```
| ?- retractall(foo(_,_)),
     assert(foo(X,Y)),
     X = a,
     Y = b,
     foo(c,d).
```

However, this goal would fail, because `c` and `d` cannot be unified:

```
| ?- create(prolog_term, TermObj),
     TermObj << p_term(foo(X,X)),
     TermObj >> p_term(foo(c,d)).
```

14.2.3.2 Direct Slot Access

Get and put methods are not automatically generated for private and protected slots. Those slots are accessed by the `fetch_slot/2` and `store_slot/2` predicates, which may only appear in the body of a method clause and which always operate on the object to which the message is sent. It is not possible to access the slots of another object with these predicates.

You may declare a slot to be private or protected in order to limit access to it. However, it is still possible, and frequently useful, to define get and put methods for such a slot.

For example, if numerator and denominator slots of the rational number class were private rather than public, it would be possible to define put methods to ensure that the denominator is never 0 and that the numerator and denominator are relatively prime. The get methods merely fetch slot values, but they need to be defined explicitly, since the slots are private. The new definition of the rational number class might start as follows:

```
:- class rational =
        [num:integer=0,
         denom:integer=1].

Self >> num(N) :-
        fetch_slot(num, N).

Self >> denom(D) :-
        fetch_slot(denom, D).

Self >> (N/D) :-
        Self >> num(N),
        Self >> denom(D).
```

One of the put methods for the class might be

```
Self << num(NO) :-
        fetch_slot(denom, DO)
        reduce(NO, DO, N, D),
        store_slot(num, N),
        store_slot(denom, D).
```

where the `reduce/4` predicate would be defined to divide `NO` and `DO` by their greatest common divisor, producing `N` and `D`, respectively.

The definition of `reduce/4` and the remaining put methods is left as an exercise. The put methods should fail for any message that attempts to set the denominator to 0.

14.2.3.3 Send Methods

Messages that do something more than fetch or store slot values are usually defined as send messages. While the choice of message operators is (usually) up to the programmer, choosing them carefully enhances the readability of a program.

For example, print methods might be defined for the point and rational number classes, respectively, as

```
Self <- print(Stream) :-
        Self >> x(X),
        Self >> y(Y),
        format(Stream, "(~w,~w)", [X, Y]).
```

and

```
Self <- print(Stream) :-
        fetch_slot(num, N),
        fetch_slot(denom, D),
        format(Stream, "~w/~w", [N, D]).
```

These methods are used to access slot values. But, the fact that the values are printed to an output stream makes it more reasonable to define them as send messages than get messages.

Frequently send methods modify slot values. For example, the point class might have methods that flip points around the x and y axes, respectively:

```
Self <- flip_x :-
        Self >> y(Y0),
        Y1 is -1 * Y0,
        Self << y(Y1).

Self <- flip_y :-
        Self >> x(X0),
        X1 is -1 * X0,
        Self << x(X1).
```

And, the rational number class might have a method that swaps the numerator and denominator of a rational number object. It fails if the numerator is 0.

```
Self <- invert :-
        fetch_slot(num, N)
        N =\= 0,
        fetch_slot(denom, D)
        store_slot(num, D),
        store_slot(denom, N).
```

These methods modify slot values, but they do not simply store values that are given in the message. Hence, it is more reasonable to use the send operator.

It is possible for a method to produce more than one answer. For example, the class whose definition begins with

```
:- class interval =
        [public lower:integer,
         public upper:integer].
```

might define a send method

```
Self <- in_interval(X) :-
        Self >> lower(L),
        Self >> upper(U),
        between(L, U, X).
```

which uses the `between/3` predicate from `library(between)`. The `in_interval` message will bind `X` to each integer, one at a time, between the lower and upper slots, inclusive. It fails if asked for too many answers.

The rest of this section describes particular kinds of send messages.

14.2.3.4 Create and Destroy Methods

Objects are created with the `create/2` predicate. When you define a class, you must specify all the ways that instances of the class can be created. The simplest creation method is defined as

```
Self <- create.
```

If this method were defined for *Class*, then the command

```
| ?- create(Class, Object).
```

would create an instance of *Class* and bind the variable `Object` to that instance. All slots would receive their (possibly default) initial values.

More generally, if the definition for *Class* contains a create method

```
Self <- create(Arguments) :-
        Body.
```

then the command

```
| ?- create(Class(Arguments), Object).
```

will create an instance of *Class* and execute the *Body* of the create method, using the specified *Arguments*. The variable *Object* is bound to the new instance.

If a simple class definition has no create methods, then it is impossible create instances of the class. While the absence of create methods may be a programmer error, that is not always the case. Abstract classes, which are classes that cannot have instances, are often quite useful in defining a class hierarchy.

Create methods can be used to initialize slots in situations when specifying initial slot values will not suffice. (Remember that initial values must be specified as constants at compile time). The simplest case uses the arguments of the create message as initial slot values. For example, the definition of the point class might contain the following create method.

```
Self <- create(X,Y) :-
        Self << x(X),
        Self << y(Y).
```

If used as follows

```
| ?- create(point(3.14159, 2.71828), PointObj),
     PointObj >> x(X),
     PointObj >> y(Y).
```

it would give `X` and `Y` the values of 3.14159 and 2.71828, respectively.

In some cases, the create method might compute the initial values. The following (partial) class definition uses the `date/1` predicate from `library(date)` to initialize its year, month and day slots.

```
:- class date_stamp =
        [year:integer,
         month:integer,
         day:integer].

Self <- create :-
        date(date(Year, Month, Day)),
        store_slot(year, Year),
        store_slot(month, Month),
        store_slot(day, Day).
```

All three slots are private, so it will be necessary to define get methods in order to retrieve the time information. If no put methods are defined, however, the date cannot be modified after the `date_stamp` object is created (unless some other method for this class invokes `store_slot/2` itself).

Create methods can do more than initialize slot values. Consider the `named_point` class, whose definition begins as follows:

```
:- class named_point =
        [public name:atom,
         public x:float=1,
         public y:float=0].

Self <- create(Name, X, Y) :-
        Self << name(Name),
        Self << x(X),
        Self << y(Y),
        assert(name_point(Name, Self)).
```

Not only does the `create/3` message initialize the slots of a new `named_point` object, but it also adds a `name_point/2` fact to the Prolog database, allowing each new object to be found by its name. (This create method does not require the `named_point` object to have a unique name. Defining a `uniq_named_point` class is left as an exercise.)

An object is destroyed with the `destroy/1` command. Unlike `create/2`, `destroy/1` does not require that you define a `destroy` method for a class. However, `destroy/1` will send a destroy message (with no arguments) to an object before it is destroyed, if a `destroy` method is defined for the object's class.

If a `named_point` object is ever destroyed, the address of the object stored in this name `point/2` fact is no longer valid. Hence, there should be a corresponding destroy method that retracts it.

```
Self <- destroy :-
        Self >> name(Name),
        retract(name_point(Name, Self)).
```

Similar create and destroy methods can be defined for objects that allocate their own separate memory or that announce their existence to foreign code.

14.2.3.5 Instance Methods

Instance methods allow each object in a class to have its own method for handling a specified message. For example, in a push-button class it would be convenient for each instance (each push-button) to have its own method for responding to being pressed.

The declaration

```
:- instance_method Name/Arity, ....
```

inside a class definition states that the message *Name/Arity* supports instance methods. If the class definition defines a method for this message, it will be treated as a default method for the message.

The `define_method/3` predicate installs a method for an object of the class, and the `undefine_method/3` predicate removes that method.

Suppose that the `date_stamp` class, defined earlier, declared an instance method to print the year of a `date_stamp` instance.

```
:- instance_method print_year/1.

Self <- print_year(Stream) :-
        Self >> year(Y0),
        Y1 is Y0 + 1970,
        format(Stream, "~d", [Y1]).
```

The arithmetic is necessary because UNIX dates are based on January 1, 1970.

If a particular `date_stamp` object's date were to be printed in Roman numerals, it could be given a different `print_year` method, using the `define_method/3` predicate.

```
| ?- create(date_stamp, DateObj),
     define_method(DateObj,
   print_year(Stream),
   print_roman_year(Stream, DateObj)).
```

If this `date_stamp` object is created in 1994, then a `print_year` message sent to it would print the current year as

```
MCMXCIV
```

Defining the predicate `print_roman_year/2` is left as an exercise. It must be able to access the `year` slot of a `date_stamp` object. Because it is not defined by a method clause within the class definition, `print_roman_year/2` cannot use the `get_slot/2` predicate.

None of `instance_method/1`, `define_method/3`, `undefine_method/3` specify a message operator. Instance methods can only be defined for send messages.

14.3 Inheritance

This section describes the additional features (and the additional complexity) of defining classes with inheritance in Quintus Objects. Most of what was said about classes in the previous section remains true in these examples.

14.3.1 Single Inheritance

The simplest case is when a new class inherits some properties (slots and methods) from a single superclass. That superclass may, in turn, be defined in terms of its superclass, etc. The new class, its superclass, its superclass's superclass (if any) and so on are all ancestors of the new class.

14.3.1.1 Class Definitions

The definition of a class with a single superclass begins with a `class/1` directive of the form

```
:- class ClassName = [SlotDef, ...] +  SuperClass.
```

where the list of *SlotDef* descriptions may be empty. In that case, the definition can simplified to

```
:- class ClassName = SuperClass.
```

The class *SuperClass* must be a defined class when this definition is given.

In Quintus Objects, a subclass inherits all the slots of its superclass. And, by default, it inherits all the methods of its superclass. The remainder of this section describes what the programmer can do to control this inheritance.

14.3.1.2 Slots

A class's slots are a combination of those explicitly defined in its slot description list and the slots it inherits from its superclass. In Quintus Objects, a class inherits all the slots of its superclass. It follows that a class inherits all the slots of all its ancestors.

The programmer's control over inheritance of slots is limited. It is not possible to rename an inherited slot, nor is it possible to change its type, unless it is a class slot. It is possible to change a slot's initial value. And, it is possible to effectively change a slot's visibility.

To change the initial value or the type (when allowed) of a slot, include a new *SlotDef* in the list of slot descriptions for the class, with the same slot name and a new type or initial value. The type of a class slot can only be changed to a subclass of the type of the superclass's slot. The new initial value must still be a constant of the appropriate type.

The `named_point` class, defined earlier, could have better been defined from the point class, which began as follows:

```
:- class point =
        [public x:float=0,
         public y:float=0].
```

The definition of the `named_point` class would then begin with

```
:- class named_point =
        [public name:atom,
         public x:float=1.0] + point.
```

This `named_point` class has public slots named `name`, `x` and `y`, with the same types and initial values as the earlier `named_point` definition, which did not use inheritance. This

`named_point` class also inherits all the methods of the `point` class, which saves us from having to write them again (and maintain them).

A slot that was private or protected in a superclass may be defined as public. This will cause get and put methods to be generated in the subclass. A slot that was public in a superclass may be defined as protected or private, but this does not prevent it from inheriting the get and put methods of the superclass. For that, the `uninherit/1` directive, defined below, is needed.

14.3.1.3 Methods

In Quintus Objects, by default, a class inherits all the methods of its superclass. The programmer has more control over the inheritance of methods than the inheritance of slots, however. In particular, methods can be uninherited and they can be redefined.

To prevent a method from being inherited, use the `uninherit/1` directive. For example, suppose that the class `point` is defined as before. That is, its definition begins as follows:

```
:- class point =
        [public x:float=0,
         public y:float=0].
```

Because both slots are public, a put method is automatically generated for each, which allows their values to be changed.

The definition of a new class `fixed_point` might begin as follows:

```
:- class fixed_point = point.

:- uninherit
        point << (x/1),
        point << (y/1).

Self <- create(X, Y) :-
        store_slot(x, X),
        store_slot(y, Y).
```

The parentheses are necessary because of the precedences of the '`<<`' and '`/`' operators.

Because the put methods from `point` are not inherited, no instance of the `fixed_point` class can change its `x` and `y` values once created—unless the class definition contains another method for doing so. The get methods are inherited from `point`, however.

To redefine a method, simply include method clauses for its message within a class's definition. The new method clauses replace, or shadow, the inherited method clauses for this class.

Another way to prevent the `x` and `y` slots of the `fixed_point` class from being modified would be to shadow the put methods. For example, they might be redefined as

```
Self << x(_) :-
        format(user_error, "cannot modify x slot value.~n.", []),
        fail.

Self << y(_) :-
        format(user_error, "cannot modify y slot value.~n", []),
        fail.
```

Now attempts to modify the `x` or `y` values of a fixed point object generate a specific error message and fail. A more complicated version would raise an appropriate exception.

14.3.1.4 Send Super

Even when a superclass's method is shadowed or uninherited, it is possible to use the superclass's method inside a method clause for the new class. This makes it possible to define a "wrapper" for the superclass's method, which invokes the superclass's method without having to duplicate its code. This technique works with all message types.

Sending a message to a superclass is done with a command of the form

```
super MessageOp Message
```

where *MessageOp* is one of the message operators ('`<<`', '`>>`' or '`<-`') and *Message* is a message defined for the superclass. A generalization of this syntax may be used to specify which superclass to send the message to. This is discussed in Section 14.3.2.3 [obj-inh-mih-meth], page 686.

Sending a message to a class's superclass can only be done within a message clause.

14.3.2 Multiple Inheritance

It is possible for a class to be defined with more than one superclass. Because the class inherits properties from multiple superclasses, this is referred to as multiple inheritance.

Multiple inheritance is a complex and controversial topic. What should be done about conflicting slot or method definitions? (This is sometimes called a "name clash.") What should be done about slots that are inherited from two or more superclasses, but that originate with a common ancestor class? (This is sometimes called "repeated inheritance".) Different systems take different approaches.

Quintus Objects supports multiple inheritance in a limited but still useful way. It does not allow repeated inheritance, and it places all the responsibility for resolving name clashes

on the programmer. This section describes the multiple inheritance features of Quintus Objects.

14.3.2.1 Class Definitions

The definition of a class with multiple superclasses begins with a `class/1` directive of the form

```
:- class ClassName = [SlotDef, ...] + SuperClass + ....
```

The list of slot descriptions and the superclasses to the right of the '`=`' can appear in any order, without changing the class being defined. In fact, the slot descriptions can be partitioned into more than one list, without changing the class. However, it is best to adopt a fairly simple style of writing class definition and use it consistently.

Just as the slot names in a list of slot descriptions must be distinct, superclass names should not be repeated.

14.3.2.2 Slots

In Quintus Objects, the programmer has no control over multiple inheritance of slots. All slots from all superclasses are inherited. And, the superclasses should have no slot names in common.

As a consequence, in Quintus Objects no superclasses of a class should have a common ancestor. The only exception would be the unusual case where that common ancestor has no slots.

14.3.2.3 Methods

By default, all methods are inherited from all superclasses. Any of the superclasses' methods can be uninherited, as described earlier, by using the `uninherit/1` directive.

If the same message is defined for more than one superclass, however, you must choose at most one method to inherit for the message. You may choose none. You may do this by defining a new method for the message (shadowing the superclasses' methods), or by using the `uninherit/1` directive, or by using the `inherit/1` directive.

The following is considered a classic example of multiple inheritance.

```
:- class toy.              % no slots in this class

Self >> size(small).

Self >> rolls(false).

:- end_class toy.

:- class truck.          % no slots in this class

Self >> size(large).

Self >> rolls(true).

:- end_class truck.
```

The idea expressed in these definitions is that most toys are small and do not roll. On the other hand, most trucks are large, but they do roll. A toy truck shares one feature with each class, but we can hardly expect a compiler to choose the correct one.

The definition of a new class, toy_truck, might begin with

```
:- class toy_truck = toy + truck.
```

Rather than redefine the get methods for `size` and `rolls`, we can specify which to inherit in two ways. One way is positive, stating which to inherit, and the other way is negative, stating which not to inherit.

The positive version would be

```
:- inherit
        toy >> (size/1),
        truck >> (rolls/1).
```

This is more convenient when a message is defined in several superclasses, because all but the chosen method are uninherited. And, it is probably easier to understand.

The negative version would be

```
:- uninherit
        toy >> (rolls/1),
        truck >> (size/1).
```

The `toy_truck` class would exhibit the same behavior with either definition.

It is possible to define methods that access the shadowed or uninherited methods of the superclasses, by sending the message to the superclasses. In the case of multiple inheritance, however, it may be necessary to specify which superclass to send the message to.

The `toy_truck` class, for example, might define these methods:

```
Self >> uninherited_size(S) :-
        super(truck) >> size(S).

Self >> uninherited_rolls(R) :-
        super(toy) >> rolls(R).
```

They provide access to the unchosen methods from `toy_truck`'s superclasses.

While these examples with the toy_truck class are clearly "toy" examples, the same techniques can be used in more realistic cases.

14.3.2.4 Abstract and Mixin Classes

While Quintus Objects only supports a limited form of multiple inheritance, its facilities are sufficient for working with so-called *mixin classes*.

The idea is to construct similar classes by first defining a class that contains the things the desired classes have in common. Typically, this will be an *abstract class*, which will have no instances itself. Then, provide the features that differentiate the desired classes with a set of mixin classes

Mixin classes that have nothing in common can safely be mixed together, to build the desired classes. The mixin classes will usually be abstract classes, also, because they are too specialized for their instances to be useful on their own.

The date_stamp class defined earlier would make a good mixin class. A similar `time_stamp` class might be (partially) defined as follows:

```
:- class time_stamp =
        [hour:integer,
         minute:integer,
         second:integer].

Self <- create :-
        time(time(Hour, Minute, Second)),
        store_slot(hour, Hour),
        store_slot(minute, Minute),
        store_slot(second, Second).
```

Another mixin class might be used to "register" objects in the Prolog database.

```
:- class registry = [name:atom].

Self <- create(Name) :-
        Self << name(Name),
        assert(registered(Name, Self)).

Self <- destroy :-
        Self >> name(Name),
        retract(registered(Name, Self)).
```

The `registry` mixin class could have been used with the `point` class to define the `named_point` class, which was an example from an earlier section.

The ability to send a message to an object's superclass is useful when working with mixin classes. Suppose the definition of a new class begins with

```
:- NewClass = OldClass + date + time + registry.
```

where *OldClass* is some previously defined class that lacks the features provided by the `date`, `time` and `registry` classes. (In fact, they should not have any slot names in common.) Then its create method can be defined by

```
Self <- create(Name) :-
        super(OldClass) <- create,
        super(date) <- create,
        super(time) <- create,
        super(registry) <- create(Name).
```

This avoids the need to duplicate the code in the create methods of OldClass and all three mixin classes.

14.3.3 Asking About Classes and Objects

It is possible to determine, at run time, what classes are defined, how they are related by inheritance, what class an object belongs to, etc. This section describes the predicates used for those purposes. Most of the predicates involve the class hierarchy, so they are properly described in the section on inheritance. But, several can be useful even in programs that use only simple classes.

Most of these predicates come in pairs, where one predicate involves one class or its direct superclasses, and the other predicate involves all ancestors. For example, the `class_superclass/2` and `class_ancestor/2` predicates connect a currently defined class to its superclass(es) and to all its ancestors, respectively.

In all of these predicates, the ancestors of a class include not only superclasses and their ancestors, but also the class itself. A class cannot be a superclass of itself, by the rules of defining classes. However, it is convenient to consider every class an ancestor of itself,

because then we may say that every property of a class is defined in one of its ancestors, without having to say "the class itself or a superclass or a superclass of a superclass, etc."

14.3.3.1 Objects

The `class_of/2` predicate is used to test whether an object is of a particular type or to determine the type of an object. Similarly, the `descendant_of/2` predicate relates an object to all ancestors of its class. (Remember that the object's class is, itself, an ancestor class of the object.)

Both require the first argument (the object) to be instantiated. That is, the predicates cannot be used to find objects of a given class. If you need to search among all the objects of a class, you must provide a way to do it. One way to do this is to assert a fact connecting the class name to every object, when it is created. The named_point example of the previous section took that idea a step further by allowing each object to have a different name.

The `pointer_object/2` predicate relates an object's address (a pointer) to the object. Remember that an instance of *Class* is represented by a term of the form

```
Class(Address)
```

The `pointer_object/2` predicate requires that one of its arguments be instantiated, but it may be either one. Hence, just by knowing the address of an object (which possibly was returned by a foreign function) it is possible to determine the object's type.

Most Prolog programmers can safely ignore the `pointer_object/2` predicate, unless they are using Quintus Objects with foreign functions or with the Structs package.

14.3.3.2 Classes

The `current_class/1` predicate is used to ask whether a class is currently defined or to get the names of all currently defined classes.

The `class_superclass/2` predicate is used to test whether one class is a superclass of another, or to find a class's superclasses, or to find a class's subclasses, or to find all subclass-superclass pairs. The `class_ancestor/2` predicate is used in the same ways for the ancestor relation between currently defined classes.

As an example, the following goal finds all the ancestors of each currently defined class.

```
| ?- setof(C-As,
   (current_class(C),
    setof(A, class_ancestor(C,A), As)),
   L).
```

It binds `L` to a list of terms of the form *Class-AncestorList*, with one term for each currently defined class.

Arguably, this predicate violates the principle of information hiding, by letting you ask about how a class is defined. Therefore, you should generally avoid it. It may be useful, however, in debugging and in building programmer support tools.

14.3.3.3 Messages

The `message/4` predicate is used to ask whether a message is defined for a class or to find what messages are defined for a class, etc. It does not distinguish between messages whose methods are defined in the class itself and those that are inherited from a superclass.

The `direct_message/4` predicate is used to ask whether a message is not only defined for a class, but whether the method for that message is defined in the class itself. It can also be used to determine which methods are defined in a class. This ability to look inside a class definition makes `direct_message/4` an egregious violator of the principle of information hiding. Thus it, like `class_ancestor/2`, should mainly be confined to use in programmer support applications.

Both `message/4` and `direct_message/4` take the message operator as an argument, along with the class, message name and arity. Hence it is possible to use these predicates to ask about get, put or send messages.

It is not possible to ask about a class's slots, nor should it be. However, it is possible (and quite reasonable) to ask about the get and put messages that are defined for a class. For example, the following goal finds all the 1-argument messages that are defined for both the get and put message operators in the class *Class*.

```
| ?- setof(Message,
     (message(Class, <<, Msg, 1),
      message(Class, >>, Msg, 1)),
     L).
```

There may or may not be slots corresponding to these messages; that detail is hidden in the definition of *Class*. However, it should be possible to use *Class* as if the slots were there.

As an example, recall the polar coordinate interface to the point class, which defined get and put methods for `r` and `theta`, even though data was represented inside an object by rectangular coordinates `x` and `y`.

14.4 Term Classes

Sometimes it is convenient to be able to send messages to ordinary Prolog terms as if they were objects. Prolog terms are easier to create than objects, and unlike objects, they are automatically garbage collected (see Section 14.5.2.2 [obj-tech-lim-gc], page 697). Of

course, unlike objects, Prolog terms cannot be modified. However, when a particular class of objects never needs to be dynamically modified, and doesn't need to be subclassed, it may be appropriate to define it as a *term class*.

A term class is defined much like an ordinary class: it begins with a '`:- class`' directive defining the class and its slots, follows with clauses defining the methods for this class, and ends with an '`:- end_class`' directive, the end of the file, or another '`:- class`' directive. The only difference is in the form of the '`:- class`' directive introducing a term class definition.

14.4.1 Simple Term Classes

The simplest sort of term class declaration has the following form:

```
:- class ClassName = term(Term).
```

This declares that any term that unifies with *Term* is an instance of class *ClassName*. For example, you might declare:

```
:- class rgb_color = term(color(_Red,_Green,_Blue)).

color(R,_G,_B) >> red(R).
color(_R,G,_B) >> green(G).
color(_R,_G,B) >> blue(B).

:- end_class rgb_color.
```

This would declare any term whose principal functor is `color` and arity is three to be an object of class `rgb_color`. Given this declaration, entering the goal

```
color(0.5, 0.1, 0.6) >> blue(B)
```

would bind `B` to 0.6.

Note that you cannot use `create/2` to create a term class instance. Since they are just ordinary terms, you can create them the same way you'd create any ordinary Prolog term. Similarly, you cannot modify an existing term class instance.

You may specify a term class as the type of a slot of an ordinary class. This is effectively the same as specifing the type to be `term`. In particular, fetching and storing term class slots is not very efficient. Also, the default value for slots of term class type is `''`; this is because not enough is known about a simple term class to determine a better default. For an explanation of how to avoid these pitfalls, see Section 14.4.3 [obj-tcl-tce], page 693.

14.4.2 Restricted Term Classes

The intention of the `rgb_color` class presented above is to represent a color as a triple of floating point numbers between 0.0 and 1.0. But the above definition does not restrict the arguments of the `color` term in any way: *any* `color/3` term is considered to be an instance of the `rgb_color` class.

The second form of term class declaration allows you to specify constraints on instances of a term class. The form of such a declaration is as follows:

```
:- class ClassName = term(Term, Constraint).
```

This declares that any term that unifies with *Term* and satisfies *Constraint* is an instance of class *ClassName*. The *Constraint* term is an ordinary Prolog goal, which will usually share variables with *Term*.

To extend our `rgb_color` class example so that only `color/3` terms whose arguments are all floats between 0.0 and 1.0 are instances of `rgb_color`, we would instead begin the definition as follows:

```
:- class rgb_color =
         term(color(Red,Green,Blue),
              (float(Red),   Red >= 0.0,   Red =< 1.0,
               float(Green), Green >= 0.0, Green =< 1.0,
               float(Blue),  Blue >= 0.0,  Blue =< 1.0)).
```

Note the parentheses around the constraint in this example. Whenever the constraint contains multiple goals separated by commas, you will need to surround the goal with parentheses.

With this definition of the `rgb_color` class, only `color/3` terms whose arguments are all floating point numbers between 0 and 1 inclusive will be considered to be instances of `rgb_color`.

14.4.3 Specifying a Term Class Essence

As mentioned above, it is possible to specify a term class as the type of a slot of some other object. For example, we might declare

```
:- class colored_rectangle = [
        public origin:point,
        public size:size,
        public color:rgb_color].
```

This will store an `rgb_color` object (i.e., a `color/3` term) in the `color` slot of each `colored_rectangle` object. Unfortunately, though, Quintus Objects cannot tell what is the best way to store a term object, and therefore it stores it the same way it stores a slot declared to be

of `term` type: using the Prolog database. This has all the efficiency disadvantages of `term` slots. In this case, however, we know that all that really needs to be saved in order to save an `rgb_color` object is the three arguments. We also know that each of these arguments is a floating point number, and because precision isn't terribly critical in representating colors, each of these numbers can be stored as a `float`, rather than a `double`. In effect, we know that the *essence* of a `rgb_color` object is these three numbers; if we have them, we can easily construct the `color/3` term. If we provide this information in the declaration of the `rgb_color` class, then Quintus Objects can store instances of the `rgb_color` class as 3 separate floats, rather than as a term, significantly improving the performance of creating or destroying a `colored_rectangle` object, as well as accessing or modifying its `color` slot.

The essence of a term class is specified with the following form of `class` declaration:

```
:- class ClassName = term(Term, Constraint, Essence).
```

where *Essence* is of the form

```
[Name1:Type1=i[Variable1], Name2:Type2=i[Variable2], ...]
```

and each *Name* is a distinct atom naming a slot, each *Type* is a slot type as specified in Section 14.2.2.2 [obj-scl-slt-typ], page 671, and each *Variable* is an unbound variable appering in *Term*. Providing a term essence not only makes storage of terms in ordinary object slots more efficient, it also gives a name to each "essential" slot of the term class. This allows you to use `fetch_slot` to fetch the slots of this class.

To extend our `rgb_color` example, we might introduce the `rgb_color` class with this declaration:

```
:- class rgb_color =
         term(color(Red,Green,Blue),
              (float(Red),   Red >= 0.0,   Red =< 1.0,
               float(Green), Green >= 0.0, Green =< 1.0,
               float(Blue),  Blue >= 0.0,  Blue =< 1.0),
              [red:float=Red, green:float=Green, blue:float=Blue]).
```

This declaration defines the `rgb_color` class exactly as the example declaration of the previous section: every `color/3` term whose arguments are all floating point numbers between 0.0 and 1.0 inclusive are instances of `rgb_color`. The difference is that with this declaration, ordinary classes that have slots of type `rgb_color`, such as the `colored_rectangle` example above, will be stored more efficiently, and their `rgb_color` slots will be accessed and modified much more efficiently. Also, it will be possible to use `fetch_slot(red, Red)` in the methods of the `rgb_color` class to fetch to red component of the message recipient, and similarly for `green` and `blue`.

14.5 Technical Details

This section will be expanded in future versions of Quintus Objects. For now, it provides a BNF grammar for the syntax of class definitions and a short list of some limitations of Quintus Objects.

14.5.1 Syntax of Class Definitions

The following BNF grammar gives a concise description of the syntax of class definitions. It assumes an understanding of Prolog syntax for the following items: *variable*, *atom*, *compound_term*, and *constant*. Slot types, particularly the `address`, `class` and `pointer` types, were discussed in an earlier section.

class_def ::= *class_begin* `{` *clause* `|` *method* `}` *class_end*

class_begin ::= `:- class` *class_name opt_class_spec* `.`

opt_class_spec ::= *empty* `|` `=` *class_spec*

class_spec ::= *multi_parent_or_slots* `|` *term_class_spec*

clause ::= *head opt_body* `.`

head ::= *atom* `|` *compound_term* `.`

method ::= *message_head opt_body* `.`

message_head ::= *message_goal*

class_end ::= `:- end_class` *opt_class_name* `.`
`|` *empty* /* if followed by *class_begin* or *eof* */

message ::= *atom* `|` *compound_term*

multi_parent_or_slots ::= *parent_or_slots* `{` `+` *parent_or_slots* `}`

parent_or_slots ::= *class_name* `|` `[]` `|` `[` *slot_def* `{,` *slot_def* `}` `]`

slot_def ::= *opt_visibility slot_name* `:` *slot_type opt_init_value*

opt_visibility ::= *empty* `|` `private` `|` `protected` `|` `public`

opt_init_value ::= *empty* `|` `=` *constant*

term_class_spec ::= `term(`*term opt_goal_essence*`)`

opt_goal_essence ::= *empty* | `,` *goal opt_essence*

opt_essence ::= *empty* | `,` *essence*

essence ::= `[` *variable* `:` *slot_type* `{ , }` *variable* `:` *slot_type* `} ]`

opt_body ::= *empty* | `:-` *body*

body ::= *message_or_goal* `{ , }` *message_or_goal* `}`

message_or_goal ::= *message_goal* | *goal*

message_goal ::= *variable message_operator message*

message_operator ::= `<<` | `>>` | `<-`

opt_class_name ::= *empty* | *class_name*

class_name ::= *atom*

slot_name ::= *atom*

slot_type ::= `integer`
| `short`
| `char`
| `unsigned_short`
| `unsigned_char`
| `float`
| `double`
| `atom`
| `address`
| `term`
| *class_name*
| `pointer(`*atom*`)`

14.5.2 Limitations

This section summarizes the current limitations of Quintus Objects.

14.5.2.1 Debugging

When you debug Quintus Objects programs that were compiled using the `obj_decl` module, you are tracing the translated version of your code. This includes all method clauses and (some) message sending commands.

QUI's source-level debugger cannot connect compiled Quintus Objects code with the source code. This is similar to the problem of tracing Prolog's DCG grammar rules.

14.5.2.2 Garbage Collection

There is no garbage collection of objects. It is the responsibility of the programmer to keep track of unused objects. In particular, avoid doing the following:

```
| ?- create(Class, Object).
```

Unless the create message for *Class* made some provision for finding the new object again, it is now lost. It cannot be used, and it cannot be destroyed.

14.5.2.3 Multiple Inheritance

The provisions for multiple inheritance in this version of Quintus Objects are limited. In particular, there is no control over the inheritance of slots, which makes repeated inheritance impossible. However, it does support the mixin style of multiple inheritance.

14.5.2.4 Persistence

While objects are more persistent than Prolog variables, there is no automatic way to save objects from one execution of your program to the next. Hence they are less persistent than the clauses in the Prolog database.

If you need to save a set of objects from one Prolog session to another, copy the objects to the Prolog database as terms, and save them to a QOF file. Then, after you reload the QOF file, rebuild the objects. Keep in mind that addresses are not valid from one session to another.

In short, there is no way to avoid initializing objects at run time.

14.6 Exported Predicates

The following reference pages, alphabetically arranged, describe the exported Quintus Objects predicates. They can be imported by an embedded command:

```
:- use_module(library(objects)).
```

14.6.1 `<-/2`

Synopsis

`+`*Obj* `<-` `+*`*Mesg*

Arguments

Obj	*object*
Mesg	*term*

Description

Sends *Mesg* to *Obj*. A send message. The class of *Obj* must have a method defined for this message.

A clause with `<-/2` as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Exceptions

`instantiation_error`
: either argument is unbound.

`domain_error`
: *Mesg* is not *callable* or *Obj* is not a valid object.

`existence_error`
: *Mesg* is not a defined message for *Obj*.

Caveat

For reasons of efficiency, an `existence_error` exception will only be raised if the code that sends the message is compiled with debugging enabled (see `debug_message`), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the `<-/2` predicate will be compiled into more efficient code if the `obj_decl` module is loaded at compile time.

See Also

`<</2`, `>>/2`, `direct_message/4`, `message/4`

14.6.2 <</2

Synopsis

+*Obj* << +*Att*

Arguments

Obj	*object*
Att	*term*

Description

Send a message to `Obj` to store the value of `Att` in the object. A put message. `Att` must be an attribute that can be stored in objects of `Obj`'s class.

A clause with `<</2` as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Put methods are automatically generated for public slots.

Exceptions

`instantiation_error`
: either argument is unbound.

`domain_error`
: *Mesg* is not *callable* or *Obj* is not a valid object.

`existence_error`
: *Mesg* is not a defined message for *Obj*.

Caveat

For reasons of efficiency, an `existence_error` exception will only be raised if the code that sends the message is compiled with debugging enabled (see `debug_message`), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the `<</2` predicate will be compiled into more efficient code if the `obj_decl` module is loaded at compile time.

See Also

`<-/2`, `>>/2`, `direct_message/4`, `message/4`, `store_slot/2`

14.6.3 >>/2

Synopsis

+*Obj* >> +-*Att*

Arguments

Obj	*object*
Att	*term*

Description

Send a message to *Obj* that fetches the value of *Att* from the object. A get message. *Att* must be an attribute to fetch from *Obj*'s class.

A clause with >>/2 as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Get methods are automatically generated for public slots.

Exceptions

`instantiation_error`
: either argument is unbound.

`domain_error`
: *Mesg* is not *callable* or *Obj* is not a valid object.

`existence_error`
: *Mesg* is not a defined message for *Obj*.

Caveat

For reasons of efficiency, an `existence_error` exception will only be raised if the code that sends the message is compiled with debugging enabled (see `debug_message`), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the >>/2 predicate will be compiled into more efficient code if the `obj_decl` module is loaded at compile time.

See Also

`<-/2`, `<</2`, `direct_message/4`, `message/4`, `fetch_slot/2`

14.6.4 `class/1` directive

Synopsis

```
:- class ClassName.

:- class ClassName = [SlotDef, ...].

:- class ClassName = Super.

:- class ClassName = [SlotDef, ...] + Super + ....

:- class ClassName = term(Term).

:- class ClassName = term(Term, Goal).

:- class ClassName = term(Term, Goal, Essence).
```

Arguments

ClassName	*atom*
SlotDef	*term*
Super	*atom*

Description

The definition of class *ClassName* begins with this `class/1` directive and ends with the next `class/1` directive, the next `end_class/[0,1]` directive, or the end of the file, whichever comes first. All clauses that look like method definitions within the scope of the class definition (that is, which have one of `<-/2`, `<</2` or `>>/2` as the principal functors of their heads) are considered method definitions of the class.

You may provide as many slot definitions (*SlotDef*) and superclasses (*Super*) as you like. All superclasses must be previously defined classes.

A slot definition (SlotDef) has the form

```
Visibility SlotName:Type = InitialValue
```

where *Visibility* and '`= InitialValue`' are optional.

Visibility is either `public`, `protected`, or `private`. If it is omitted, the slot is private.

SlotName must be an atom.

SlotType must be one of the following:

`integer` 32-bit signed integer

`short` 16-bit signed integer

`char` 8-bit signed integer

`unsigned_short`
16-bit unsigned integer

`unsigned_char`
8-bit unsigned integer

`float` 32-bit floating point number

`double` 64-bit floating point number

`atom` Prolog atom (32-bit pointer)

`address` 32-bit address

`term` Prolog term

Class 32-bit pointer to an instance of *Class*, which must be a previously defined class

`pointer(`*Type*`)`
like `address`, except that access to this slot yields, and update of this slot expects, a unary term whose functor is *Type*

InitialValue may be any constant appropriate for the slot's type.

Term, if specified, is any compound Prolog term. Class declarations of any of the last three forms introduce a *term class*, which defines any term that unifies with *Term* as an instance of the class being defined.

Goal, if specified, is any Prolog goal. This goal may be used to restrict which terms that unify with *Term* will be considered to be instance of the class being defined. The default *Goal* is `true`. Other than when it is `true`, *Goal* will usually share variables with *Term*.

Essence, if specified, is a list of terms of the form

```
Variable:Type
```

where *Variable* is a variable apprearing somewhere in *Term* and *Type* is one of the possible *Slottype* types listed above. There should be a *Variable*:*Type* pair for every variable in *Term*. By specifying an essence, you permit much more space- and time-efficient storage of and access to term slots.

Caveat

Note that every class for which you want to be able to create instances must define at least one create method.

Examples

The following class definition is for a class named `point`, with two public slots, named `x` and `y`. Both slots are of type `integer` and have initial values of 1 and 2, respectively.

```
:- class point =
        [public x:integer=1,
         public y:integer=2].

Self <- create.
:- end_class point.
```

Because the slots are public, they have get and put methods generated automatically. Because the class has a create method defined, it is possible to create an instance with the command

```
| ?- create(point, PointObj).
```

which creates a `point` object and binds the variable `PointObj` to it.

Using the `point` class, we could create a class, `named_point`, which has an extra public slot, `name`.

```
:- class named_point =
        [public name:atom] + point.

Self <- create(Name, X, Y) :-
        Self << name(Name),
        Self << x(X),
        Self << y(Y).

:- end_class named_point.
```

The only way to create a `named_point` object requires specifying values for all three slots.

See Also

`end_class/[0,1]`

Section 14.2 [obj-scl], page 669, Section 14.4 [obj-tcl], page 691.

14.6.5 `class_ancestor/2`

Synopsis

`class_ancestor(*Class, *Anc)`

Arguments

Class	*atom*
Anc	*atom*

Description

Anc is *Class* or an ancestor class of *Class*.

See Also

`class_superclass/2`

14.6.6 `class_method/1` directive

Synopsis

```
:- class_method +Name/+Arity, ... .
```

Arguments

Name	*atom*
Arity	*integer*

Description

Declares that a class's method for send message *Name*/*Arity* is an ordinary method, not an instance method.

Used when the class being defined inherits an instance method from a superclass, to allow the class to define a non-instance method for the message. A descendent class may still declare this to be an instance method, so the same message may be an instance method for some classes and an ordinary class method for others.

Must occur within the scope of the class definition. Only applies to send messages.

See Also

`instance_method/1`

14.6.7 `class_superclass/2`

Synopsis

`class_superclass(*Class, *Super)`

Arguments

Class	*atom*
Super	*atom*

Description

Class is an immediate subclass of *Super*.

See Also

`class_ancestor/2`

14.6.8 `class_of/2`

Synopsis

`class_of(+Obj, -Class)`

Arguments

Obj	*object*
Class	*atom*

Description

Class is the class of *Obj*.

Exceptions

`instantiation_error`
: *Obj* is unbound.

`type_error`
: *Obj* is not a valid object.

See Also

`pointer_object/2`

14.6.9 `create/2`

Synopsis

`create(+`*`Descriptor`*`,-`*`Obj`*`)`

Arguments

Descriptor	*term*
Obj	*object*

Description

Obj is a newly created and initialized object. Descriptor is a term describing the object to create. After memory is allocated and any slot initializations are performed, a create message is sent to the object.

The functor of *Descriptor* indicates the class to create. The arguments of the create message are the arguments of *Descriptor*.

Exceptions

`instantiation_error`
: *Descriptor* is unbound.

`domain_error`
: *Descriptor* is not a valid `create` descriptor.

`resource_error`
: unable to allocate enough memory for object.

Caveat

You must have a `create/`*`N`* method for every arity *N* you want to be able to use in creating instances of a class. This includes arity 0. If no such method exists, a domain error will be raised.

Examples

Given the class definition

```
:- class point =
                [public x:integer=1,
                 public y:integer=2].

Self <- create.
Self <- create(X, Y) :-
                Self << x(X),
                Self << y(Y).
:- end_class point.
```

the command

```
| ?- create(point, Point1).
```

creates a `point` object, with the default slot values for `x` and `y`, and binds variable `Point1` to the new object. The command

```
| ?- create(point(10,15), Point2).
```

creates a `point` object with values 10 and 15 for slots `x` and `y`, respectively, and binds variable `Point2` to the new object.

See Also

`destroy/1`

14.6.10 `current_class/1`

Synopsis

`current_class(*`*`Class`*`)`

Arguments

Class *atom*

Description

`Class` is the name of a currently defined class.

14.6.11 `debug_message/0` directive

Synopsis

```
:- debug_message.
```

Description

Prolog clauses following this directive will be compiled to send messages "carefully."

That is, a message sent to an object that does not understand the message will raise an exception, which describes both the message and the object receiving it. This also catches attempts to send an unbound message, to send a message to an unbound object, and similar errors.

See Also

`nodebug_message/0`

14.6.12 `define_method/3`

Synopsis

`define_method(+`*`Obj`*`, +`*`Message`*`, +`*`Body`*`)`

Arguments

Obj	*object*
Message	*term*
Body	*callable*

Description

Install *Body* as the method for *Message* in the instance *Obj*. Following the execution of this goal, sending *Message* to *Obj* will execute *Body*, rather than the default method or a method previously defined with `define_method/3`.

Message must have been declared to be an instance method for the class of *Obj*.

Exceptions

`instantiation_error`
: any argument is unbound.

`type_error`
: *Obj* is not a compound term, or *Message* or *Body* is not *callable*.

`domain_error`
: *Message* does not specify an instance method for the class of *Obj*, or *Body* include a goal to fetch or store a non-existent slot.

See Also

`instance_method/1`, `undefine_method/3`

14.6.13 `descendant_of/2`

Synopsis

`descendant_of(+Obj, *Class)`

Arguments

Obj	*object*
Class	*atom*

Description

Obj is an instance of *Class* or of a descendant of *Class*.

Exceptions

`instantiation_error`
: *Obj* is unbound.

`type_error`
: *Object* is not a valid object.

See Also

`class_ancestor/2`, `class_of/2`, `class_superclass/2`

14.6.14 `destroy/1`

Synopsis

`destroy(+`*`Obj`*`)`

Arguments

Obj *object*

Description

Dispose of *Obj*.

First send a `destroy` message to *Obj*, if such a message is defined for its class. A `destroy` message takes no argument. Unlike `create/2`, it is possible to destroy instances of a class even if it defines no `destroy` methods.

Exceptions

`instantiation_error`
: *Obj* is unbound.

`type_error`
: *Object* is not a valid object.

See Also

`create/2`

14.6.15 `direct_message/4`

Synopsis

`direct_message(*Class, *Op, *Name, *Arity)`

Arguments

Class	*atom*
Op	*message_operator*
Name	*atom*
Arity	*integer*

Description

Name/*Arity* is an *Op* message directly understood (defined rather than inherited) by instances of *Class*. This predicate is used to test whether a message is defined for a class.

Op is one of `<-`, `>>`, or `<<`, specifying the kind of message.

This predicate violates the principle of information hiding by telling whether the method for a message is defined within a class or inherited. Hence its use in ordinary programs is discouraged. It may be useful, however, during debugging or in developing programming support tools.

See Also

`<-/2`, `<</2`, `>>/2`, `message/4`

14.6.16 `end_class/[0,1]` directive

Synopsis

```
:- end_class.
```

```
:- end_class +ClassName.
```

Arguments

ClassName
: *atom*

Description

A class definition continues until the next `end_class/[0,1]` directive, the next `class/1` directive, or the end of the file, whichever comes first.

It is not possible to nest one class definition within another.

All clauses that look like method definitions (that is, which have one of `<-/2`, `<</2` or `>>/2` as the principal functors of their heads) are considered to be method definitions for the class.

Caveat

The argument to `end_class/1`, if specified, must match the class name of the preceding `class/1` directive.

See Also

`class/1`

14.6.17 `fetch_slot/2`

Synopsis

`fetch_slot(+`*`SlotName`*`, -`*`Value`*`)`

Arguments

SlotName	*atom*
Value	*term*

Description

Fetch *Value* from the slot specified by *SlotName*.

This predicate may only appear in the body of a method clause, and it always operates on the object to which that message is sent. It cannot be used to directly access the slots of another object.

Exceptions

`instantiation_error`
: *Slot* is unbound.

`domain_error`
: *Slot* is not the name of a slot of the current class.

`permission_error`
: *Slot* is a private slot of a superclass.

See Also

`>>/2`, `store_slot/2`

14.6.18 `inherit/1` directive

Synopsis

`:- inherit` *+ClassName* *+Op* *+Name/+Arity*`, ....`

Arguments

ClassName	*atom*
Op	*message_operator*
Name	*atom*
Arity	*integer*

Description

ClassName names the class from which the message should be inherited, *Op* indicates which kind of message it is, and *Name* and *Arity* indicate the name and arity of the message to be inherited. You may include several inheritance specifications in one directive.

Caveat

Be careful of the precedences of the message operator and the `/` operator. You may need to use parentheses.

Examples

Suppose classes `toy` and `truck` are defined as follows:

```
:-class toy.
Self <- create.
Self >> size(small).
Self >> rolls(false).
:- end_class toy.

:- class truck.
Self <- create.
Self >> size(small).
Self >> rolls(true).
:- end_class truck.
```

Then `toy_truck` inherits its size from `toy` and the fact that it rolls from `truck`:

```
:- class toy_truck = toy + truck.
:- inherit
        toy <- (create/0),
        toy <- (size/1),
        truck <- (rolls/1).
:- end_class toy_truck.
```

Note that this is just a toy example.

See Also

`uninherit/1`

14.6.19 `instance_method/1` directive

Synopsis

```
:- instance_method +Name/+Arity.
```

Arguments

Name	*atom*
Arity	*integer*

Description

The message *Name*/*Arity* is declared to support instance methods in a class. This means that instances of this class, and its descendants, may each define their own methods for this message.

A method defined for this message by the class is considered the default method for the message. An instance that does not define its own method uses the default. Defining a new method overrides this default method; there is no need to explicitly remove it.

An instance method is installed in an instance of the class with the `define_method/3` predicate. An instance method is removed from an instance of the class, reverting to the default method, with the `undefine_method/3` predicate.

Must occur within the scope of the class definition. Only applies to send messages.

See Also

`class_method/1`, `define_method/3`, `undefine_method/3`

14.6.20 `message/4`

Synopsis

`message(*Class, *Op, *Name, *Arity)`

Arguments

Class	*atom*
Op	*message_operator*
Name	*atom*
Arity	*integer*

Description

Name/*Arity* is an *Op* message understood by instances of *Class*. This predicate is used to test whether a message is either defined for or inherited by a class.

Op is one of `<-`, `>>`, or `<<`, specifying the kind of message.

See Also

`<-/2`, `<</2`, `>>/2`, `direct_message/4`

14.6.21 `nodebug_message/0` directive

Synopsis

```
:- nodebug_message.
```

Description

Prolog clauses following this directive are no longer compiled to send messages "carefully."

See Also

`debug_message/0`

14.6.22 `pointer_object/2`

Synopsis

`pointer_object(+`*`Addr`*`,-`*`Obj`*`)`

`pointer_object(-`*`Addr`*`,+`*`Obj`*`)`

Arguments

Addr	*integer*
Obj	*object*

Description

Addr is the address of object *Obj*. This can be used to get the address of an object or to get an object given its address.

Exceptions

`instantiation_error`
: both *Obj* and *Addr* are unbound.

`type_error`
: *Addr* is not an integer.

14.6.23 `store_slot/2`

Synopsis

`store_slot(+`*`SlotName`*`, +`*`NewValue`*`)`

Arguments

SlotName *atom*

NewValue *term*

Description

Store *NewValue* in the slot specified by *SlotName*.

This predicate may only appear in the body of a method clause, and it always operates on the object to which that message is sent. It cannot be used to directly modify the slots of another object.

Exceptions

`instantiation_error`
: either argument is unbound.

`type_error`
: *NewValue* is not of the appropriate type for *Slotname*.

`domain_error`
: *Slotname* is not the name of a slot of the current class.

`permission_error`
: *Slotname* is a private slot of a superclass.

See Also

`<</2`, `fetch_slot/2`

14.6.24 `undefine_method/3`

Synopsis

`undefine_method(+`*`Obj`*`, +`*`Name`*`, +`*`Arity`*`)`

Arguments

Obj	*object*
Name	*atom*
Arity	*integer*

Description

Remove *Obj*'s current instance method for the *Name*/*Arity* message. After executing this goal, sending this message to *Obj* executes the class's default method for the message.

Name/*Arity* must have been declared to be an instance method for the class of *Obj*.

If *Obj* has no current instance method for the *Name*/*Arity* message, the predicate has no effect.

Exceptions

`instantiation_error`
: any argument is unbound.

`type_error`
: *Obj* is not a compound term, *Name* is not an atom, or *Arity* is not an integer.

`domain_error`
: *Message* does not specify an instance method for the class of *Obj*.

See Also

`define_method/3`, `instance_method/1`

14.6.25 `uninherit/1` directive

Synopsis

```
:- uninherit +Class +Op +Name/+Arity, ... .
```

Arguments

Class	*atom*
Op	*message_operator*
Name	*atom*
Arity	*integer*

Description

This prevents the class within whose scope this directive appears from inheriting the *Name* /*Arity* method of type *Op* from ancestor *Class*.

If *Class* is unbound, the specified message is uninherited from all ancestors that define it.

Caveat

Note that if you define a message for your class, you do not need to uninherit that message from its superclasses: it will automatically be shadowed.

Be careful of the precedences of the message operator and the `/` operator. You may need to use parentheses.

Examples

```
:- uninherit someclass << (foo/1),
             someclass >> (foo/1).
```

This prevents the get and put methods for the slot `foo` from being inherited from any ancestors of class `someclass`. In effect, it makes the `foo` slot a protected slot for this class.

See Also

`inherit/1`

14.7 Glossary

abstract class
: A class that cannot have instances. Abstract classes are helpful in designing a class hierarchy, to contain the common parts of several concrete classes.

ancestor
: One of a class's superclasses, one of its superclasses's superclasses, etc. Sometimes, for convenience, ancestor includes the class itself, along with its proper ancestors.

child
: A synonym for subclass.

class
: A class is defined by a description of the information its instances contain and the messages they respond to. Every object is an instance of one and only one class.

concrete class
: A class that can have instances. Most classes are concrete.

create method
: Specifies what actions should be taken when an instance of a class is created. A create method frequently provides initial slot values or specifies an action to be performed by the new object. A create message is sent to each new object by the `create/2` predicate. A create message is a kind of send message.

descendant
: One of a class's subclasses, one of its subclasses's subclasses, etc. Sometimes the word descendant includes the class itself, along with its proper descendants.

destroy method
: Specifies what actions should be taken when an instance of a class is destroyed. A destroy message is sent to an object by the `destroy/1` predicate. A destroy message is a kind of send message.

direct slot access
: Fetching or storing a slot value without sending a message to the object. This should be used with care!

  Quintus Objects allows direct access to a class's slots only within its method definitions, via the `fetch_slot/2` and `store_slot/2` predicates.

get message
: A message that inquires about some aspect of an object. Typically used to fetch slot values. Get methods are automatically generated for public slots. Get messages are written with the '`>>`' operator.

inheritance
: The process by which a class's slots and methods are determined from an ancestor.

initial value
: The value a slot is initialized to when an object is created. Every slot has a default initial value, which depends upon its type. You may specify different initial values in a class definition.

instance Another word for object. The word instance draws attention to the class of which the object is an instance.

instance method
A method that may be defined differently for each instance of a class. The class may have a default method for this message, which is overridden by installing an instance method for a particular object.

message A command to an object to perform an operation or to modify itself, or an inquiry into some aspect of the object. In Quintus Objects, a message is either a get message, a put message or a send message. The syntax for sending a message to an object is

```
Object Operator Message
```

where *Operator* is one of the following:

`>>` get message

`<<` put message

`<-` send message

method A class's implementation of a particular message. You send messages to an object, but you define methods for a class.

method clause
A Prolog clause used to define a method for a class. A method clause has one of `<-/2`, `<</2` or `>>/2` as the principal functor of its head, and it can only appear within the scope of its class's definition. A method's definition may contain more than one message clause.

mixin class
A class that is intended to be combined (mixed in) with other classes, via multiple inheritance, to define new subclasses.

multiple inheritance
When a class names more than one superclass. Typically, it inherits slots and methods from each. In Quintus Objects, two different superclasses should not use the same slot name. And, if a message is defined by more than one superclass, the class definition must specify which method to inherit.

object A modifiable data item that holds information and responds to messages. Another word for instance.

parent class
A synonym for superclass.

private slot
A private slot is, by default, only accessible within methods of the class itself. Not even the descendants of the class may access its private slots, except through the class's methods. Get and put methods are not automatically generated for a private slot, so it is only accessed via the methods you define. If the visibility of a slot is not specified, it is private, rather than public or protected.

protected slot
: A protected slot is, by default, only accessible within methods of the class itself and its descendants. Get and put methods are not automatically generated for a protected slot, so it is only accessed via the methods you define. If the visibility of a slot is not specified, it is private, rather than public or protected.

  Quintus Objects `protected` is similar to `protected` in C++.

public slot
: A public slot is accessible via its own get and put methods, which are generated for it automatically. If no visibility is specified, a slot is private, rather than public or protected.

put message
: A message that modifies some aspect of an object. Typically used to store slot values. Put methods are automatically generated for public slots. Put messages are written with the '`<<`' operator.

send message
: The most common sort of message. Used for performing an operation on an object or for performing an action that depends upon an object. Send messages are written with the '`<-`' operator.

send super
: When a method for a class executes a shadowed superclass's method. This allows a class to put a "wrapper" around its superclass's method, making it unnecessary to duplicate the method just to make a small extension to it.

shadow
: When a class defines its own method for a message defined by one of its ancestors, the new method hides or "shadows" the ancestor's method. The new class's descendants will inherit its method for that message, rather than its ancestors. That is, a class always inherits the "closer" of two methods for a message.

slot
: A part of an instance that holds an individual datum. Like a member of a C struct or a field of a Pascal record.

subclass
: A class that is a more specific case of a particular class. This is the opposite of superclass. A class does not name its subclasses; they are inferred.

superclass
: A class that is a more general case of a particular class. Each class lists its superclasses.

term class
: A class whose instances are represented as ordinary Prolog terms. The functor of these objects need not be the name of the class, and the arity need not be one.

term slot
: A slot that can hold any Prolog term.

uninherit
: Specify that a method from a superclass should not be inherited. This is similar to shadowing the superclass's method, but does not specify a replacement for it.

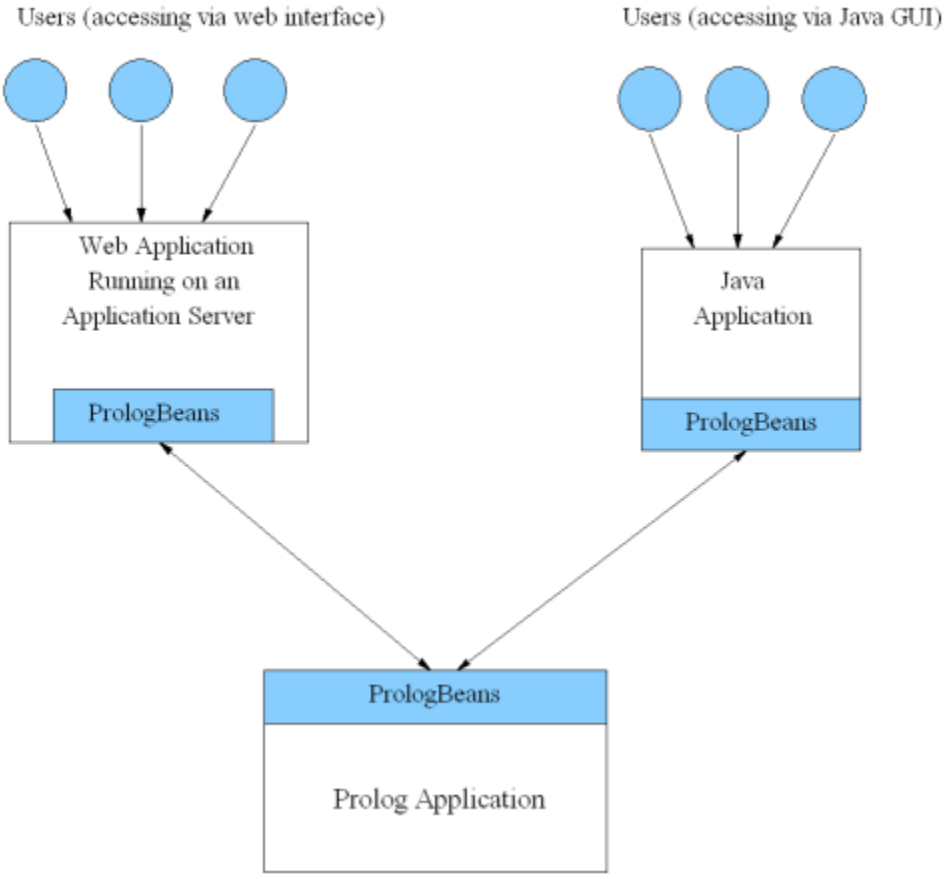
visibility
: A slot may be defined to be either `public`, `protected`, or `private`. By default, if no visibility is specified, a slot is private.

15 The PrologBeans Package

15.1 Introduction

PrologBeans is a package for integrating Java and Prolog applications. The main idea is to let Java and Prolog run in separate processes. It is usually a bad idea to let Java and Prolog coexist in the same process, as their respective virtual machines tend to compete over resources such as memory and UNIX signals.

The current version of the package is designed to be used when Java applications need to send queries to a Prolog server (and less intended for showing a GUI from a Prolog program). One typical application is to connect Java based web applications to a Prolog server (see examples later).



PrologBeans setup where the Prolog application serves several users accessing both via a web application server and a Java GUI.

The PrologBeans package is split into the file '`prologbeans.jar`', to be used in the Java application, and the '`library(prologbeans)`' module, to be used in the Prolog part of the application, i.e. the *Prolog server*.

All PrologBeans examples can be found in the '`qplib('prologbeans/demo')`' directory, which is one of the directories covered by the '`demo`' file search path.

15.2 Features

The current version of PrologBeans is designed to be used mainly as a connection from Java to Prolog. Current features are:

- Socket based communication
- Allows Java application and Prolog server to run on different machines
- Multiple Java applications can connect to same Prolog server
- Java applications can make use of several Prolog servers
- Allows Java Applets to access Prolog server
- Platform independent (e.g. any platform where Prolog and Java exist)
- Simplifies the use of Prolog in Java application servers (Tomcat, etc)
- Prohibits unwanted use of Prolog server by host control (only specified hosts can access the Prolog server)
- Supports Java servlet sessions
- Supports JNDI lookup (Java Naming and Directory Interface)

Coming features:

- Connection pooling (several connections in application servers and several running Prolog servers for better performance)
- More advanced options for querying the Prolog server
- Better support for communication from Prolog to Java (e.g. a Java server that the Prolog application can connect to)
- Support for launching Prolog and loading Prolog programs from a Java application

15.3 A First Example

This section provides an example to illustrate how PrologBeans can be used. This application has a simple Java GUI where the user can enter expressions that will be evaluated by an expression evaluation server.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import se.sics.prologbeans.*;

public class EvaluateGUI implements ActionListener {

  private JTextArea text = new JTextArea(20, 40);
  private JTextField input = new JTextField(36);
  private JButton evaluate = new JButton("Evaluate");
  private PrologSession session = new PrologSession();

  public EvaluateGUI() {
    JFrame frame = new JFrame("Prolog Evaluator");
    Container panel = frame.getContentPane();
    panel.add(new JScrollPane(text), BorderLayout.CENTER);
    JPanel inputPanel = new JPanel(new BorderLayout());
    inputPanel.add(input, BorderLayout.CENTER);
    inputPanel.add(evaluate, BorderLayout.EAST);
    panel.add(inputPanel, BorderLayout. SOUTH);
    text.setEditable(false);
    evaluate.addActionListener(this);
    input.addActionListener(this);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    try {
      Bindings bindings = new Bindings().bind("E",
                          input.getText() + '.');
      QueryAnswer answer =
        session.executeQuery("evaluate(E,R)", bindings);
      Term result = answer.getValue("R");
      if (result != null) {
        text.append(input.getText() + " = " + result + '\n');
        input.setText("");
      } else {
        text.append("Error: " + answer.getError() + "\n");
      }
    } catch (Exception e) {
      text.append("Error when querying Prolog Server: " +
                  e.getMessage() + '\n');
    }
  }

  public static void main(String[] args) {
    new EvaluateGUI();
  }
}
```

The Java code above first sets up the GUI with a text area for showing results, a text field for entering expressions, and a button for requesting an evaluation (the constructor `EvaluateGUI()`). It will also add itself as `ActionListener` on both the text field and the button. The method `actionPerformed(ActionEvent event)` will be called whenever the user has pressed ⟨RET⟩ or clicked on the button. `actionPerformed` first binds the variable E to the value of the text field, and then sends the query to the Prolog server with `session.executeQuery("evaluate(E,R)", bindings);`. If everything goes well, the Prolog server will return an answer (bound to R), which will be appended to the text area.

```
:- module(evaluate,[main/0,my_predicate/2]).
:- use_module(library(prologbeans)).
:- use_module(library(charsio), [read_from_chars/2]).

%% Register acceptable queries and start the server (using default port)
main:-
    register_query(evaluate(C,P), my_predicate(C,P)),
    start.

%% We have received a list of characters,
%% which needs to be converted into an expression
my_predicate(Chars, P) :-
    read_from_chars(Chars, X),
    P is X.
```

The Prolog code above first defines the module and imports the needed modules. Then, in the `main/0` predicate, it configures the server to answer queries on the form `evaluate(C,P)` and starts the server. The last few lines defines the predicate `my_predicate(Chars, P)`, which is the predicate that performs the evaluation. Note that the expression to evaluate is represented as a list of characters and must be converted into a term before evaluation.

Please note: the environment variable `QP_PATH` as used here is meant to be a shorthand for *quintus-directory*, and does not need to be set explicitly.

To start the example, first start the Prolog server by going to the '`qplib('prologbeans/demo')`' directory and type:

```
% prolog +l evaluate.pl
| ?- main.
```

To start the GUI type (from the same directory as above):

```
> java -
classpath "%QP_PATH%\java3.5\prologbeans.jar;." EvaluateGUI
```

(Windows), or

```
% java -
classpath "$QP_PATH/java3.5/prologbeans.jar:." EvaluateGUI
```

(UNIX)

15.4 Java Interface

The Java interface is centered around the class `PrologSession`, which represents a connection (or session) to a Prolog server. `PrologSession` contains static methods for looking up named `PrologSession` instances using JNDI (Java Naming and Directory Interface) as well as methods for querying the Prolog server. Other important classes are: `QueryAnswer`, which contains the answer for a query sent to the Prolog server; `Term`, which represents a Prolog term; and `Bindings`, which supports stuffing of variable values used in queries.

General information about Java, Servlets and JNDI is available at the Java Technology site: http://java.sun.com/

A brief description of the methods in the provided Java classes are presented below.

The `PrologSession` object is the connection to the Prolog server. The constructor `PrologSession()` creates a `PrologSession` with the default settings (`host = localhost`, `port = 8066`. `PrologSession` contains the following methods:

static PrologSession `getPrologSession (String name)` [Method on `PrologSession`]

returns the `PrologSession` registered in JNDI with the given name. Use this method in application servers where services are registered using JNDI. **Please note**: the application server must be configured to register the `PrologSession` with the given name for this method to work. See Tomcat configuration in Section 15.6.3 [pbn-exs-tomcat], page 747.

static PrologSession `getPrologSession (String name, HTTPSession session)` [Method on `PrologSession`]

returns the `PrologSession` registered in JNDI with the given name. The `PrologSession` will make use of sessions and the session id will be the same as in the `HTTPSession`. Use this method in web application servers with support for servlets and `HTTPSession` (and when support for sessions is desired).

String *getHost* () [Method on `PrologSession`]

returns the host of the Prolog server (exactly as registered in `setHost`).

void *setHost* (`String prologServerHost`) [Method on `PrologSession`]

sets the host of the Prolog server (default `localhost`). Either IP-address or host name is allowed.

int *getPort* () [Method on `PrologSession`]

returns the port of the Prolog server.

void *setPort* (`int port`) [Method on `PrologSession`]

sets the port of the Prolog server (default `8066`).

void *connect* () [Method on `PrologSession`]
connects to the Prolog server. By default the `executeQuery` will automatically connect to the server when called.

void *setAutoConnect* (`boolean autoConnect`) [Method on `PrologSession`]
sets the connection mode of this `PrologSession`. If set to `true` it will ensure that it is connected to the Prolog server as soon as a call to `executeQuery` or anything else causing a need for communication happens. This is by default set to `true`

boolean *isAutoConnecting* () [Method on `PrologSession`]
returns the state of the AutoConnect mode.

QueryAnswer *executeQuery* (`String query`) [Method on `PrologSession`]
sends a query to the Prolog server and waits for the answer before returning the `QueryAnswer`. Anonymous variables (underscore, '_'), will be ignored, and thus not accessible in the `QueryAnswer`. `executeQuery` throws `IOException` if communication problems with the server occurs. **Please note**: `executeQuery` will only return one answer.

QueryAnswer *executeQuery* (`String query, Bindings bindings`) [Method on `PrologSession`]
sends a query to the Prolog server and waits for the answer before returning the `QueryAnswer`. `Bindings` are variable bindings for the given *query* and will ensure that the values are stuffed correctly. An example:

```
QueryAnswer answer =
   executeQuery("evaluate(In,Out)",
                new Bindings().bind("In","4*9."));
```

The `QueryAnswer` contains the answer (new bindings) for a query (or the error that occurred during the query process). `QueryAnswer` inherits from `Bindings`, and extends and modifies it with the following methods:

Term *getValue* (`String variableName`) [Method on `QueryAnswer`]
returns the value of the given variable. If there is a value a `Term` (a parsed Prolog term) is returned, otherwise `null` is returned. All bindings from the query are available in the `QueryAnswer`.

boolean *queryFailed* () [Method on `QueryAnswer`]
returns `true` if the query failed (e.g. the Prolog returned `no`). In this case, there will be no answers (no new bindings, and `isError` will return `false`).

boolean *isError* () [Method on `QueryAnswer`]
returns `true` if there was an error.

String *getError* () [Method on `QueryAnswer`]
returns the error message (which is only set if there was an error, otherwise it will be `null`).

The `Term` object is for representing parsed Prolog terms, and has the following methods:

boolean *isAtom* () [Method on `Term`]
returns `true` if the `Term` is an atom.

boolean *isInteger* () [Method on `Term`]
returns `true` if the `Term` is an integer.

boolean *isFloat* () [Method on `Term`]
returns `true` if the `Term` is a floating-point number.

boolean *isCompound* () [Method on `Term`]
returns `true` if the `Term` is a compound term.

boolean *isList* () [Method on `Term`]
returns `true` if and only if `Term` is a compound term with principal functor `./2`.

boolean *isString* () [Method on `Term`]
returns `true` if the `Term` an instance of `PBString` (which can be used for fast string access by a type-cast to `PBString` and the use of the method `getString()` that returns the string).

boolean *isVariable* () [Method on `Term`]
returns `true` if the `Term` is a variable.

int *intValue* () [Method on `Term`]
returns the `int` value of the integer.

long *longValue* () [Method on `Term`]
returns the `long` value of the integer.

float *floatValue* () [Method on `Term`]
returns the `float` value of the floating-point number.

double *doubleValue* () [Method on `Term`]
returns the `double` value of the floating-point number.

String *getName* () [Method on `Term`]
returns the functor name of the `Term` (see `functor/3`). If the Term represents a variable (`isVariable()` returns `true`), the variable name is returned.

int *getArity* () [Method on `Term`]
returns the number of arguments of this term (e.g. `parent(A1,A2)` would return 2) (see `functor/3`). If the term is not a compound term, `getArity()` will return 0.

Term *getArgument* (`int index`) [Method on `Term`]
returns the `Term` representing the argument at the position given by *index*. If there are no arguments, or if an argument with the specified index does not exist, `IndexOutOfBoundsException` will be thrown. The first argument has index one (see `arg/3`).

`Bindings` is used for binding variables to values in a query sent to the Prolog. The values will be automatically stuffed before they are sent to the Prolog server.

void *bind* (`String name, int value`) [Method on `Bindings`]
binds the variable with the given name to the given value. **Please note**: this method is also available for values of type `long`, `float`, `double`, and `Term`.

void *bind* (`String name, String value`) [Method on `Bindings`]
binds the variable with the given name to the given value. The value will be seen as a list of `UNICODE` character codes in Prolog.

void *bindAtom* (`String name, String value`) [Method on `Bindings`]
binds the variable with the given name to the given value. **Please note**: this method will encode the String as an atom when querying the Prolog server.

15.5 Prolog Interface

The Prolog interface is based on the idea of a Prolog server that provides its service by answering queries from external applications (typically Java applications). The Prolog interface in PrologBeans is defined in '`library(prologbeans)`', which implements the Prolog server and exports the following predicates:

`start`
`start(+`*`Options`*`)`
starts the Prolog server using the options specified. *Options* should be a list of zero or more of:

`port(?`*`Val`*`)`
an integer denoting the port number of the Prolog server (default: 8066). If *`Val`* is a variable then some unused port will be selected by the OS, the actual port number can be obtained with `get_server_property/1`, typically from a `server_started` event listener.

`accepted_hosts(+`*`Val`*`)`
a list of atoms denoting the hosts (in form of IP-addresses) that are accepted by the Prolog server (default: `['127.0.0.1']`).

`session_timeout(+Val)`
: an integer denoting the duration of a session in seconds. The session will be removed if it has been inactive more than this timeout when the session garbage collect starts. If the session timeout is set to zero there will be no garbage collect on sessions (default: `0`).

`session_gc_timeout(+Val)`
: an integer denoting the minimum time in seconds between two consecutive session garbage collections. If the timeout is set to zero there will be no garbage collect on sessions (default: `0`).

For example:

```
:- start([port(7500),
          accepted_hosts(['127.0.0.1','99.8.7.6'])]).
```

`shutdown`
`shutdown(+Mode)`
: shuts down the server and closes the sockets and the streams after processing all available input. There are three modes:

`now`
: as soon as possible (default).

`no_sessions`
: after all sessions have ended (all sessions have either been explicitly removed by request of the client application, or they have been garbage collected). **Please note**: there can still be connections to the Prolog server even when all sessions have ended.

`no_connections`
: after all connections to the Prolog server are closed. **Please note**: there can still be user sessions left when all connections have been closed.

`register_query(+Query, :PredicateToCall, +SessionVar)`
: registers a query and the corresponding predicate. Before the registration any previously registered query matching `Query` will be removed (as if `unregister_query(Query)` was called). The predicate, `PredicateToCall` will be called, as if by `once(PredicateToCall)`, when a query matching `Query` is received. Before calling the query, the variable *SessionVar*, if given, is bound to the id of the current session. Session ids are typically generated in web applications that track users and mark all consecutive web-accesses with the same session id.

`unregister_query(+Query)`
: unregisters all queries matching *Query*.

`session_get(+SessionID, +ParameterName, +DefaultValue, -Value)`
: returns the value of a given parameter in a given session. If no value exists, it will return the default value. Arguments:

SessionID
: is the id of the session for which values have been stored

ParameterName
: an atom, is the name of the parameter to retrieve

DefaultValue
is the value that will be used if no value is stored

Value is the stored value or the default value if nothing was stored

`session_put(+SessionID, +ParameterName, +Value)`
stores the value of the given parameter. **Please note**: any pre-existing value for this parameter will be overwritten. Note that `session_put/3` will not be undone when backtracking (the current implementation is based on `assert`). Arguments:

SessionID is the id of the session for the values to store

ParameterName
an atom, is the name of the parameter to store

Value the value to be stored

`register_event_listener(+Event, :PredicateToCall, -Id)`
`register_event_listener(+Event, :PredicateToCall)`
Registers *PredicateToCall* to be called (as if by `once(PredicateToCall)`) when the event matching *Event* occurs (event matching is on principal functor only). If the goal fails or raises an exception a warning is written to `user_error` but the failure or exception is otherwise ignored. Arguments:

Event is the event template, see below.

PredicateToCall
an arbitrary goal.

Id becomes bound to a (ground) term that can be used with `unregister_event_listener/1` to remove this event listener.

The predefined events are as follows:

`session_started(+SessionID)`
called before the first call to a query for this session

`session_ended(+SessionID)`
called before the session is about to be garbage collected (removed)

`server_started`
called when the server is about to start (enter its main loop)

`server_shutdown`
called when the server is about to shut down

Attempt to register an event listener for other events than the predefined events will throw an exception.

More than one listeners can be defined for the same event. They will be called in some unspecified order when the event occurs.

`unregister_event_listener(+Id)`
Unregister a previously registered event listener. The *Id* is the value returned by the corresponding call to `register_event_listener/3`. It is an error to attempt to unregister an event listener more than once.

15.6 Examples

15.6.1 Embedding Prolog in Java Applications

If you have an advanced Prolog application that needs a GUI you can write a stand-alone Java application that handles the GUI and set up the Prolog server to call the right predicates in the Prolog application.

An example of how to do this can be found in the '`qplib('prologbeans/demo')`' directory ('`evaluate.pl`', see the example code in Section 15.3 [pbn-exa], page 736).

Another example of this is '`pbtest.pl`', which illustrates several advanced features like:

- registering several queries
- listening to server events (`server_started`)
- shutting down the Prolog server from Java
- starting up the Prolog server from Java
- using dynamic (OS assigned) ports for the Java/Prolog communication

The example is run by executing the Java program `PBTest`:

> `> java -classpath "%QP_PATH%\java3.5\prologbeans.jar;." PBTest` (Windows), or
> `% java -classpath "$QP_PATH/java3.5/prologbeans.jar:." PBTest` (UNIX)

15.6.2 Application Servers

If you want to get your Prolog application to be accessible from an intranet or the Internet you can use this package to embed the Prolog programs into a Java application server such as Tomcat, WebSphere, etc.

An example of how to do this is provided in '`sessionsum.pl`'. This example uses sessions to keep track of users so that the application can hold a state for a user session (as in the example below, remember the sum of all expressions evaluated in the session).

```
<% page import = "se.sics.prologbeans.*" %>
<html>
<head><title>Sum Calculator</title></head>
<body bgcolor="white">
<font size=4>Prolog Sum Calculator, enter expression to evaluate:
<form><input type=text name=query></form>
<%
   PrologSession pSession =
   PrologSession.getPrologSession("prolog/PrologSession", session);

   String evQuery = request.getParameter("query");
   String output = "";
   if (evQuery != null) {
     Bindings bindings = new Bindings().bind("E",evQuery + '.');
     QueryAnswer answer =
        pSession.executeQuery("sum(E,Sum,Average,Count)", bindings);
     Term average = answer.getValue("Average");
     if (average != null) {
        Term sum = answer.getValue("Sum");
        Term count = answer.getValue("Count");

        output = "<h4>Average =" + average + ", Sum = "
        + sum + " Count = " + count + "</h4>";
     } else {
        output = "<h4>Error: " + answer.getError() + "</h4>";
     }
  }
%>
<%= output  %><br></font>
<p><hr>Powered by Quintus Prolog
</body></html>
```

The example shows the code of a JSP (Java Server Page). It makes use of the method `PrologSession.getPrologSession(String jndiName, HTTPSession session)`, which uses JNDI to look up a registered `PrologSession`, which is connected to the Prolog server. The variable *session* is in a JSP bound to the current `HTTPSession`, and the variable *request* is bound to the current `HTTPRequest`. Since the `HTTPSession` object `session` is specified all queries to the Prolog server will contain a session id. The rest of the example shows how to send a query and output the answer.

Example usage of sessions is shown below, and is from '`sessionsum.pl`':

```
:- module(sessionsum,[main/0,sum/5]).
:- use_module(library(prologbeans)).
:- use_module(library(charsio), [read_from_chars/2]).

%% Register the acceptable queries (session based)
main:-
    register_query(sum(C,Sum,Average,Count),
                   sum(C,Session,Sum,Average,Count),
                   Session),
    start.

%% The sum predicate which gets the information from a session database,
%% makes some updates and then stores it back in to the session store
%% (and returns the information back to the application server)
sum(ExprChars, Session, Sum, Average, Count) :-
    session_get(Session, sum, 0, OldSum),
    session_get(Session, count, 0, OldCount),
    read_from_chars(ExprChars, Expr),
    Val is Expr,
    Sum is OldSum + Val,
    Count is OldCount + 1,
    Average is Sum / Count,
    session_put(Session, sum, Sum),
    session_put(Session, count, Count).
```

In this example a query `sum/4` is registered to use a predicate `sum/5` where one of the variables, *Session* will be bound to the session id associated to the query. The `sum/5` predicate uses the `session_get/4` predicate to access stored information about the particular session, and then it performs the evaluation of the expression. Finally, it updates and stores the values for this session.

15.6.3 Configuring Tomcat for PrologBeans

This section will briefly describe how to set up a Tomcat server so that is it possible to test the example JSPs. Some knowledge about how to run Tomcat and how to set up your own web application is required. Detailed information about Tomcat is available at http://jakarta.apache.org/tomcat/.

Assuming that you are positioned in the Tomcat installation directory, do the following:

1. Add the '`prologbeans.jar`' to the '`common/lib/`' directory. Note that this will give all Tomcat applications access to the PrologBeans system. There are other options for importing '`prologbeans.jar`' that might be better for your type of application.
2. In the '`conf/server.xml`' file add the following (after the Tomcat Root Context tags - shown as the first lines below):

```
[...]
<!-- Tomcat Root Context -->
<!--
  <Context path="" docBase="ROOT" debug="0"/>
-->


<DefaultContext>
   <Resource name="prolog/PrologSession" auth="Container"
             type="se.sics.prologbeans.PrologSession"/>

   <ResourceParams name="prolog/PrologSession">
     <parameter>
       <name>factory</name>
       <value>org.apache.naming.factory.BeanFactory</value>
     </parameter>
     <parameter>
       <name>port</name>
       <value>8066</value>
     </parameter>
   </ResourceParams>
</DefaultContext>
[...]
```

This will register a `PrologSession` instance under the name `prolog/PrologSession` so that it is possible to do a JNDI lookup.

3. In your application's '`web.xml`' file, found in Tomcat's '`webapps/your_application/WEB-INF`' directory, you need the following resource reference:

```
<resource-ref>
  <res-ref-name>prolog/PrologSession</res-ref-name>
  <res-type>se.sics.prologbeans.PrologSession</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

4. Copy the example JSP files that you want to use ('`sessionsum.jsp`' and '`evaluate.jsp`') into the Tomcat web application directory ('`webapps/your_application`').
5. Start the Tomcat server.
6. Start the example that you want to test (Section 15.3 [pbn-exa], page 736 shows how to start one of the examples).

16 The ProXL Package

16.1 Introduction

ProXL is an interface between Quintus Prolog and the X Window System (Version 11 Release 2 or later). ProXL permits the rapid development of X-based applications using Prolog's interactive development facilities. Essentially all functionality of X is made available through Prolog calls and callbacks. ProXL is built directly on the standard Xlib interface, and ProXL code is inter-operable with Xlib C code.

ProXL is not available under Windows.

16.1.1 User Benefits

- **For the X window-application developer:** ProXL provides the power of Prolog to test and debug application code interactively.
- **For the Prolog programmer:** ProXL adds the functionality of the X window system to permit Prolog to control interactive window-oriented applications.
- **For the newcomer to X:** ProXL provides an excellent and succinct attribute-oriented interface to the basic features of X.
- **For the application vendor:** ProXL enables X development in combination with Prolog's rapid-prototyping capabilities. Interactive access to other standard packages is also available through Quintus Prolog's C language interface and Database Interface product.

16.1.2 ProXL Features

- Support for all X protocol functionality:
  - multiple displays and screens
  - nested windows
  - colors
  - all X visual types
  - pixmaps and bitmaps
  - fonts, both fixed and variable width
  - cursors, standard and custom
  - all X drawing primitives
  - all X graphics context functionality
  - window and font properties
  - window manager interface
  - selections and cut buffers

  - events
- ProXL objects have state:
  - Objects know their display and screen; no need to specify separately.
  - Windows and Pixmaps know their graphics context (GC); there is no need to specify GC when drawing.
  - Graphics Contexts remember their state.
- Primitives *get* and *put* operate on lists of attributes:
  - Attributes allow accessing or setting only desired information.
  - All attributes have defaults.
  - Access from Prolog is uniform across objects.
  - X asymmetries are smoothed and regularized.
- Inference built-in to locate correct X resources:
  - Specify a window, pixmap, or GC where font is needed.
  - Specify a window or pixmap where a GC is needed.
  - Specify a screen or display where a colormap is needed.
  - Specify anything where a display is needed.
  - Default screen and display simplify coding.
- Callbacks in Prolog:
  - Specify Prolog code to call when certain events happen.
  - Call Prolog procedures with any arguments.
  - Flexible mechanism for terminating event-handling loop.
  - Pass context-specific data to callback procedure.
  - Return result from event-handling loop.

    Callbacks are executed even while Prolog is waiting for you to type a goal.
- Program in Prolog:
  - Prolog's interactive environment simplifies code prototyping and testing.
  - Programs are much more compact than in other languages (ProXL code typically uses one-third as much code as C)
- Interoperability with C code:
  - Extend existing C/Xlib programs in Prolog with ProXL.
  - Integrate further C/Xlib code with ProXL applications.

This manual is addressed to people who know how to use the X window system, but not necessarily how to program it. For some readers, much of introduction will be familiar. You might want to read those sections that describe features peculiar to ProXL: Section 16.1.10 [pxl-bas-attr], page 755, Section 16.1.9 [pxl-bas-inf], page 755, and Section 16.1.11.1 [pxl-bas-tyi-cbk], page 756, and then skip ahead to the tutorial in Section 16.2 [pxl-tut], page 757. If you have written programs in Xlib, you might also find Section 16.16 [pxl-xlib], page 890 helpful.

X is a very subtle system, and the ProXL manual does not try to document it all. Therefore, it may occasionally be necessary to refer to a good reference on X. A major source of

information on X is the X manual for your system. Since X programming is often done in the C language, a good place to turn for detail on some of the esoteric features of X is a reference on Xlib. Another good way to learn about X is to experiment with ProXL.

16.1.3 Windows

A *window* is a rectangular area of the screen. X supports nested windows, where child windows are only visible insofar as they are within the boundary of their parent. Every application window has a *parent window*, which the window is physically inside. Every screen has a single *root window*, which is the entire contents of the screen. The root window does not have a parent, of course. A window's descendents are called its *inferiors*. All drawing into a window is confined, or *clipped*, to the window's boundaries, including all drawing into the window's inferiors. Sibling windows can overlap, in which case the *stacking order* of the windows becomes important, as one window can cover, or *occlude*, all or part of another. Drawing into a window will not affect parts of the window that are occluded.

A window can be *mapped*, that is, put on the screen, or not. If a window is not mapped, it cannot be seen, nor can any of its inferiors, whether or not they are mapped. This means that a window can be created once, and then mapped and unmapped at will without having to destroy and later recreate it. A window is said to be *viewable* if all of its ancestors are mapped. A window is said to be *visible* if it is viewable, and furthermore, it is not occluded by any siblings.

The window manager is distinct from the X server. In X, the user (by using a window manager) almost always has control over where and how large top level windows appear on the screen. Windows nested within top level windows, on the other hand, are completely under program control. The window manager, in fact, is simply another X program running alongside your applications. This means that the user can change the look and feel of the windows at any time by changing to another window manager.

16.1.4 Drawing and filling lines and shapes

Drawing in ProXL is fairly straightforward, and quite powerful. ProXL has primitives to

- draw points, lines, polygons, rectangles, arcs, ellipses (and circles)
- fill closed figures with arbitrary patterns
- draw or fill multiple items
- clear windows
- copy part of one drawable into another.

Programming in ProXL requires a certain way of looking at drawing and graphics in general. Think of a line or an ellipse not as an permanent object, but as a temporary pattern of pixels on the screen. When you draw a line, you are turning bits on. The next thing you do may

turn them off. Or they may be turned off when a user overlays your window with another. The line that you draw has no permanence. This property of X, call it *procedural drawing*, makes for a programming style in which you first open the windows you will need, and then do all drawing in callback routines (Section 16.1.11.1 [pxl-bas-tyi-cbk], page 756). If you want the line to seem to stay around, you have to explicitly provide for it to be repainted whenever it is damaged or obscured by a later event; see Section 16.1.11.2 [pxl-bas-tyi-rfr], page 756 on refreshing windows.

The primitives for drawing or filling a single shape take just the arguments you would expect:

- the destination window
- the parameters determining the shape to be drawn.

The primitives for drawing or filling multiple objects are also fairly simple. They each take

- the destination window
- a list of Prolog terms, each specifying the parameters of a shape to be drawn or filled.

In all of the drawing commands, and elsewhere in ProXL, the X coordinates are given relative to the left edge of the enclosing window , and Y coordinates are given relative to the *top* edge.

16.1.5 Drawing text

Drawing text in ProXL is like drawing lines or shapes. This means that every command to draw text must supply a position at which the text is to be drawn. ProXL does not maintain a *current* drawing position in a window, so you cannot use the standard Prolog primitives such as `write/[1,2]`, or `format/[2,3]`[1] for putting text into an X window.

Furthermore, note that the position in a drawable at which text is to be drawn is given in pixel coordinates, not character coordinates. There are three reasons for this:

- Using pixel coordinates gives you much greater flexibility in positioning and spacing of text.
- The font in which you are printing may not be fixed width (see Section 16.1.7.1 [pxl-bas-graf-font], page 753), so you cannot reliably determine where to print by multiplying the width of a character by the number of characters.
- X provides for fonts that write from right to left, so it is not always sufficient to add the pixel width of a string to its origin position to determine its right side. In this case, the origin point is at the right end of the string.

[1] Of course, the Prolog text output primitives will continue to work in the window you are running Prolog in. This is usually a terminal emulator window maintained by a program such as xterm; X does not automatically do this.

ProXL provides primitives for determining how large a string will be when it is displayed, including a primitive that will determine how many pixels left, right, above, and below the origin point a text string will occupy.

16.1.6 Drawing Pixmaps and drawing into Pixmaps

Apart from drawing shapes or text into a window, we often wish to import complex pictures from an inventory. In ProXL these are stored in *pixmaps*.

A *pixmap* is a rectangular area that can be drawn into, but cannot be seen. It may be copied into a window to be seen. It is common to draw frequently used complicated drawings into a pixmap, and then to copy the pixmap to a window whenever it is needed. Pixmaps are also used for representing textures, in which case they are *tiled*, that is, copied repeatedly like floor tiles, when they are displayed.

Pixmaps, like windows, are *drawables*, that is, something that you can draw into, not, as you might expect, something that is drawn. Most drawing primitives take a drawable as their destination argument as they can draw into either windows or pixmaps.

16.1.7 Graphics attributes of drawables

Certain parameters are needed often when drawing, for example, the color in which to draw, the width of lines, the texture with which to fill areas. These are examples of what we call *graphics attributes*. In X, graphics attributes are associated with drawing procedures. Rather than mention the same attributes over and over again, ProXL applications handle these graphics parameters by associating a set of graphics attributes with each drawable, and changing them at any time. So if you want to draw a line of width 2, then another of width 5, your program would have to change the line width graphics attribute of the drawable. In fact, this is not necessary, as we show in Section 16.1.7.3 [pxl-bas-graf-gc], page 754.

16.1.7.1 Fonts

Fonts determine what characters will look like in a window. A *font* is a graphics attribute of a drawable. It is a mapping from a character code to a bit pattern to be drawn in a drawable. Fonts may be *fixed width* (each character is the same width, like a typewriter or conventional computer terminal) or *proportionally spaced* (each character takes only the space needed for that character, like a normal typeset book). Fonts come in many sizes, shapes, and styles. They can even write from right to left! But X does not provide for rotated fonts, or fonts that write vertically.

16.1.7.2 Color and colormaps

A color is a triple of Red, Green, Blue values. Each *pixel*, or point, in a window or pixmap may be a different color. On each hardware platforms, there will be a restriction on the number of colors that may be displayed in a window at a time, and also a limitation on the number of different colors that are possible at all. For example, many screens are capable of showing approximately 16 million colors (256 different shades each of red, green, and blue), but can only show 256 colors at one time. This means that *pixel values*, the numbers assigned to each pixel, will be between 0 and 255. Each window, therefore, must have a way to know what color is associated with each pixel value. This is what a *colormap* does: it maps from pixel values to actual colors.

Notice that a colormap is associated with each window, but not with pixmaps. The pixel values in a pixmap are uninterpreted until the pixmap is copied to a window. So a pixmap copied into two different windows may appear in different colors if those windows have different colormaps.

16.1.7.3 Graphics contexts (GCs)

As we have seen in Section 16.1.7 [pxl-bas-graf], page 753, the look of drawings is determined by attaching graphics attributes to a drawable. It is often desirable to have various drawables use the same list of attributes. ProXL provides a data structure called *graphics context* (GC) to handle this situation.

To determine a figure's properties, an application can attach a list of attributes to each drawable. Alternatively, the application can create a GC, and specify the desired parameters there. Accordingly, there are two versions of each drawing primitive:

- one that uses the destination drawable's graphics attributes
- one that takes a GC as an argument.

These two approaches can be freely intermixed, and used with great power and flexibility, as explained in Section 16.6.3.3 [pxl-graf-cre-use], page 831.

16.1.8 Cursors

A *cursor* is the visual representation on the screen of the pointer device (mouse). A cursor is *not* the visual representation of the type-in position; X does not have a name for that. Each X window may specify what shape the cursor will take when the cursor is in that window. If a window doesn't specify a cursor in its attribute list, it uses the same one as its parent window.

16.1.9 Inferring arguments

In X, a *drawable* is an object into which you can draw, either a window or pixmap. ProXL drawing procedures take an argument of type "Drawable". Therefore, you can pass either a window or a pixmap to these drawing procedures.

ProXL extends this concept to include *displayables*, *screenables*, *windowables*, *colormapables*, *gcables*, and *fontables*. If, for example, a ProXL procedure requires a screen as an argument, you can pass either a screen or any other object which uniquely determines a screen. One object can uniquely determine another in two ways, either directly or via a default. For example, a window is directly associated with only one screen. Therefore, anywhere a screen is required any window associated with that screen can be specified. On the other hand, a display may have more than one screen associated with it, but it has only one default screen. Therefore, anywhere a screen is required the screen's display can be specified, if the display's default screen is the screen in question.

Following is a complete listing of the *-ables* supported by ProXL. The parenthesized associations below indicate that unique determination is made via a default value.

Object required
: **Objects fulfilling the requirement**

displayable
: display, screen, window, pixmap, colormap, gc, font, cursor

screenable
: display (default screen), screen, window, pixmap, colormap, gc

windowable
: display (root window of default screen), screen (root window), window

colormapable
: display (default colormap of default screen), screen (default colormap), window, colormap

gcable
: window, pixmap, gc

fontable
: window, pixmap, gc, font

Therefore, subsequent sections in this manual may describe an argument of a ProXL procedures as, for example, a GCable. In this case you can specify either a window, pixmap or GC as the argument.

16.1.10 Attributes: Specifying properties of ProXL objects

ProXL makes heavy use of what we call *attributes*. Graphics attributes, discussed above, are an example. An attribute is one aspect of the current state of a ProXL object. Formally, it is a Prolog term whose functor determines which attribute it is, and whose arguments determine the current state of that aspect of the object.

Attributes are used to

- specify the initial state of a newly created object
- change an object
- inquire about the current state of an object.

All procedures that use attributes accept a list, so many independent aspects of the state of an object may be examined or changed in a single operation. Some examples of attributes are `size(`*`Width`*`,`*`Height`*`)` for windows, pixmaps, and screens, and `line_width(`*`Width`*`)` for GCs, windows and pixmaps, `mapped(`*`State`*`)` for windows.

16.1.11 Handling keyboard and mouse input

X is *asynchronous*: X commands will be executed in the order they are specified, but may be executed some time later. Commands are queued up, and only transmitted to the X server when the buffer fills up or when the buffer is flushed. This affects programming style in various ways: It has consequences for error handling, callbacks, and window refreshing. It sometimes necessitates "flushing", though ProXL always flushes all ProXL displays whenever it is going to wait for user input, including input to the Prolog prompt. This greatly cuts down on the need to explicitly flush a display.

16.1.11.1 Callbacks

ProXL allows the user to register *callbacks* for each window, that is, Prolog routines that are called when certain *events* occur. These routines may be passed any arguments, including information particular to that event. For example, you may register a callback on a window for mouse button press events, and have your routine receive the position in the window of the mouse at the time the button was pressed.

Callbacks are the usual mechanism for ProXL programs to listen to and handle events, although ProXL programs may get and handle events directly if they wish.

16.1.11.2 Refreshing windows

People might expect lines or other shapes they have put on the screen to stay put. In X they do not. Another way to look at this is that the X window system does not automatically refresh windows when they become *damaged*; that is, when a window that was occluding another window is moved or unmapped, it leaves the occluded window with the wrong contents. It is *always* the responsibility of each application to keep its window up to date.

This is a case where procedural drawing (see Section 16.1.4 [pxl-bas-lin], page 751) determines programming style in ProXL. You start by writing one setup call, which determines all aspects of a window, creates that window, attaches callbacks, and maps the window. Then when the window is damaged, your callback will be executed to refresh the window. The tutorial ends with a simple example of such a program.

Also note that attempts to draw into a window before it is mapped will quietly be ignored. So if you write a program that creates a window, maps it, and then starts drawing in it, you'll often be surprised to find that there's nothing drawn in the window. It is essential to wait for the window to become mapped. The easiest way to do this is to do the drawing in a callback, which will automatically be executed as soon as the window becomes visible (as well as every time it is damaged).

16.1.11.3 Errors

Because of asynchronicity, you may have executed many commands when you discover that a previous command has caused an error. A ProXL feature described in Section 16.13.5.2 [pxl-eh-eho-synchronize], page 866 enables you to get around this problem during debugging.

There are two types of errors in X, *fatal errors* and *recoverable errors*. *Fatal errors* are handled by the server and terminate program execution. *Recoverable errors* are handled by the ProXL error handler, or a user-specified error handler.

16.1.12 Displays and Screens

A ProXL *screen* is what it sounds like: a physical screen on a monitor. Normally each CPU has one screen associated with it. However, there can be more; for instance, you might have both a color and a monochrome screen. ProXL allows programs to use all available screens.

X is *network transparent*, meaning that a program may open windows on any screen that is accessible through a network. This is possible because commands to X are sent via interprocess communication (IPC) to a separate *X server* process, which actually does the work. A ProXL *display* is essentially a connection between your Prolog program and the X server. Think of a *display* as one CPU on a network. Each display has a fixed number of *screens*, one of which is opened by default upon opening X. Also, given a display, you can enumerate its screens, as well as find its default screen.

Unless network transparency is being exploited, a ProXL programmer will not normally have to deal with displays explicitly. Screens are almost always handled by a default mechanism. Consult an Xlib manual for an overview of how screens and displays are used and why they exist.

16.2 Tutorial

This section briefly introduces each of the major aspects of Quintus ProXL through an interactive session. Starting from a primitive version, we will add features until the session covers the breadth of Quintus ProXL.

This session shows how ProXL programs can be developed interactively using Quintus Prolog. This is one of the great advantages of ProXL over most other approaches to X

Window system application development. We encourage you to bring up a ProXL system and actually type in the following session.

In Section 16.2.10 [pxl-tut-hello], page 771 we present a ProXL program whose results are the same as the interactive session. This session and program are not themselves useful, but they should give you enough of an idea of how ProXL works so that you can begin to write your own programs.

In order to understand the session, we assume that you know how to use Quintus Prolog and your X window manager and, in order to run the session and program, we assume that you have already installed both the X window system and Quintus ProXL.

ProXL refers to an object by means of a handle. An example of a handle is, `window(1787368)`. The functor `window/1` denotes the type of the object and the integer 1787368 is a unique identifier for the particular object. Your ProXL program should not depend upon the form of either the type or the identifier. Rather it should simply treat the handle as an opaque Prolog term (just like a Prolog stream or database reference). The unique object identifiers generated by ProXL will change from session to session, therefore, when you run your session the object identifiers returned by ProXL may be different than the ones printed here.

16.2.1 Displaying a Window on the Screen

First, let's put up a window. Type the command, 'prolog', into a shell window running under the X window system, and load the ProXL library:

```
% prolog

Quintus Prolog Release 3.5 (Sun 4, SunOS 5.5)

| ?- [library(proxl)].
```

You will be greeted with many messages about files being loaded, and finally will get another Prolog prompt. Now let's create a window:

```
| ?- create_window(Window, [mapped(true)]),
     retractall(win(_)),
     assert(win(Window)).

Window = window(2376568)
```

After a few moments, a window should appear on your screen. Some window managers may require you to choose a position for the window.

That's all there is to putting a window on the screen. You will notice that we didn't specify very much. We didn't say how big to make the window, nor where on the screen to put it, nor what color to make it, etc. All these parameters, and many more, are defaulted for

us by ProXL. The only thing we did specify was `mapped(true)`. In X, *mapping* a window makes it visible if its parent window is visible. Since we didn't specify a parent window for the window we're creating, the root window of the default screen is its parent, and the root window is always mapped. So by specifying the attribute `mapped(true)`, we make the new window visible.

The second argument of `create_window/2` is a list; this is called an *attribute* list. We could have put many different attributes in this list, to specify any of the attributes of the window mentioned above as having been defaulted. A complete list of the attributes supported by windows can be found in Section 16.3 [pxl-win], page 774, but for this example we won't need to use very many of them.

We use `assert/1` to remember the window we've created so that we can use it later. This is a good idea while you are experimenting with ProXL, but isn't usually necessary in actual programs. Using `retractall/1` to initialize the table first is just standard Prolog wisdom, avoiding any problems if this tutorial is run more than once.

Notice that this window is quite small; the default window size is 100 by 100 pixels. If we're going to display a message in this window, we'll need it to be bigger.

```
| ?- win(Window),
     put_window_attributes(Window, [size(250,100)]).

Window = window(2376568)
```

This makes the window 250 by 100 pixels, more than big enough for our purposes. The call to `win/1` gets back the window we just created, and `put_window_attributes/2` changes the window's size.

`put_window_attributes/2` is much like `create_window/2`, except that it allows you to specify new attributes for an existing window, and `create_window/2` lets you specify attributes for a window to be newly created. We could have specified `size(250,100)` in the call to `create_window/2`, if we had thought of it then.

If you try this, one of two things may happen: the window may be changed to the right size, or it may not. The reason is that X allows the window manager to have control over top level window sizes and positions. An application can try to change these things, but the window manager decides whether to allow it or not. The theory behind this is that the *user* should be in control of his windows, and the window manager is his tool in implementing his wishes. If this last goal didn't work for you, don't worry. Try to get your window manager to give the test window enough space for the message, and ignore the fact that the message isn't centered. The final version of this program will center the message.

16.2.2 Displaying Text in the Window

Now let's load a big font and write a message in the window.

```
| ?- load_font('*-times-bold-i-normal--24-*', Font),
     win(Window),
     put_graphics_attributes(Window, [font(Font)]),
     draw_string(Window, 10, 50, 'Hello, world!!').

Font = font(1787304),
Window = window(2376568)
```

`load_font/2` does just what it sounds like. The atom, `'*-times-bold-i-*-240-*'` is the specification of a font that is known to the X11 server.[2] To see all of the fonts known to your X11 server, use the shell command `xlsfonts`.

`put_graphics_attributes/2` will change graphics attributes of its first argument (aspects you might expect to specify when drawing). In this case, we're specifying that we want to use the font we just loaded when we draw text in this window. Then we call `draw_string/4` to write the message '`Hello, world!!`' into the window starting at the pixel position (10,50).

Notice that we didn't use `put_window_attributes/2` to specify the font for the window, but instead used `put_graphics_attributes/2`. The difference is important. ProXL has many `put_`*something*`_attributes/2` procedures, and in all cases the *something* refers to the attributes, not to the object being changed. In this case, `put_window_attributes/2` changes *window* attributes of the window you pass it. But `put_graphics_attributes/2` changes *graphics* attributes of the argument you pass it. Here that argument is a window; in Section 16.2.4 [pxl-tut-dbg], page 762 we will use `put_graphics_attributes/2` to change a pixmap.

The text is positioned at the coordinates (10,50). This means that the *origin point* for the first character is at that position, but what is the origin point? Actually, each type of font has its own way of positioning text relative to the origin point. For the typical latin fonts (fonts for the ASCII character set), however, all of a character is usually printed to the right of the origin point, and only descenders go below the origin point. So you can usually think of the origin as the left end of the baseline for the text being drawn.

16.2.3 Making the Window the Right Size

Make the window the right size for the message:

[2] If you are using X11 Release 2, then use the font name `'vbg-25'`. The way to specify fonts changed from Release 2 to Release 3.

```
| ?- win(Window),
        text_extents(Window, 'Hello, world!!', L, R, W,
                Ascent, Descent),
        Width is W+4,
        Height is Ascent+Descent+4,
        put_window_attributes(Window, [size(Width,Height)]),
        TextLeft is L+2,
        TextBase is Ascent+2.

Window = window(2376568),
L = 0,
R = 145,
W = 146,
Ascent = 17,
Descent = 3,
Width = 150,
Height = 24,
TextLeft = 2,
TextBase = 19
```

First we retrieve our window. The second line, the call to `text_extents/7`, computes how big the message is when printed in the window. The first argument to `text_extents/7` is a specification of the font we are using. Here, since we've already specified the font to use for drawing into this window, the window uniquely determines the font. So ProXL accepts the window in place of the font argument.

This unique determination of argument types is an important concept. ProXL tries to infer the right object from the object you give it. In this case, it wants a font, but it accepts a window. Previously, we associated a font with this window, so ProXL knows which font we mean. The table in Section 16.1.9 [pxl-bas-inf], page 755 spells out which entities are predictable.

Returning to our example, the remaining arguments to `text_extents/7` are:

Msg	the string whose size we're trying to find
L	left bearing: the number of pixels this string will occupy left of the origin
R	right bearing: the number of pixels this string will occupy right of the origin
W	total width
Ascent	the number of pixels above the origin
Descent	the number of pixels below the origin

The next two goals compute the width and height of the window. We allow 2 pixels of blank space all around the string, yielding the 4 pixels we add in to determine both width and height. Note that the height of a string is its ascent plus it descent.

In the fifth line, we specify the new size for the window as we did before, only now with a more precise size.

The last two lines compute the origin point for drawing into the window. *TextLeft* allows 2 pixels to the left of the string, and *TextBase* leaves 2 pixels above. For this tutorial, we'll just remember these results and use them when needed. In an actual program, we would pass them as arguments to the program doing the drawing.

Notice that resizing the window erases our '`Hello, world!!`' message. This points up an important aspect of X: it is *always* the responsibility of the application to refresh its windows. Later we'll worry about making that work automatically; for now, just manually refresh the window:

```
| ?- win(Window),
     draw_string(Window, 2, 19, 'Hello, world!!').

Window = window(2376568)
```

There shouldn't be any surprises here. The position (2,19) simply the TextLeft and TextBase we computed earlier.

16.2.4 Drawing a Textured Background

The window would look more interesting with a textured background. We decide that a 4x4 pixmap with two lines between opposite corners forming an X would make an interesting background texture.

```
| ?- create_pixmap(Pix, [size(4,4)]),
        draw_segments(Pix, [segment(0,0,3,3), segment(0,3,3,0)]),
        win(Window),
        put_window_attributes(Window, [background(Pix)]).

Pix = pixmap(2383840),
Window = window(2376568)
```

The first line is as simple as it looks: it creates a 4x4 pixel pixmap. The second line draws the X. Note that coordinates are given with (0,0) as the northwest corner pixel, so (3,3) is the southeast corner. Then the third line installs the new pixmap as the background pattern.

You might be wondering why the window didn't change. The reason is simple: the background is used to fill parts of the window that have been cleared. The easiest way to get the new background displayed is to use your window manager to either iconify and then uniconify the window, or to push it behind another window and then pull it back to the front. You could also do it by calling

```
| ?- win(Window),
     clear_window(Window).

Window = window(2376568)
```

Again we need to refresh the window:

```
| ?- win(Window),
     draw_string(Window, 2, 19, 'Hello, world!!').

Window = window(2376568)
```

That doesn't look terribly interesting. Try drawing the string in white instead of black.

```
| ?- win(Window),
     get_screen_attributes([white_pixel(Pix)]),
     put_graphics_attributes(Window, [foreground(Pix)]),
     draw_string(Window, 2, 19, 'Hello, world!!').

Window = window(2376568),
Pix = 0
```

The second line determines the pixel value for white on our screen. X does not establish a standard pixel value for black and white, so you must determine it explicitly. In fact, X doesn't even guarantee that the `black_pixel/1` and `white_pixel/1` attributes will yield black and white, but they are *logical* black and white.

The third line sets the foreground color, the color to draw in, to white. Finally, we draw our message. The default drawing color in ProXL is `black_pixel`; this is why you didn't have to specify it earlier. But now that we've specified white, all drawing will be done in white until we change it again.

16.2.5 Drawing a Drop Shadow

The white text also needs highlighting, so let's try a drop shadow. Drop shadows are done by printing the text in black first, and then again in white, offset a little bit horizontally and vertically.

Since the window is just big enough for the message, you should use the window manager to make it a bit bigger so it can accommodate both the message and its shadow. Then go ahead and draw the drop shadow.

```
| ?- win(Window),
     clear_window(Window),
     get_screen_attributes([black_pixel(Black),
                            white_pixel(White)]),
     put_graphics_attributes(Window, [foreground(Black)]),
     draw_string(Window, 7, 27, 'Hello, world!!'),
     put_graphics_attributes(Window, [foreground(White)]),
     draw_string(Window, 2, 19, 'Hello, world!!').

Window = window(2376568),
Black = 1,
White = 0
```

Note that we've determined both `black_pixel/1` and `white_pixel/1` of the screen at the same time here. You can put as many attributes in this list, and in fact in any attribute list, as you like. Other than that, this example is pretty straightforward. We drew the message first in black, then again in white five pixels to the left and eight pixels above the shadow.

The resulting displacement for the shadow seems a bit too much, at least vertically. The displacement should probably be relative to the size of the characters in the font, so let's find out how big the characters in the font are.

```
| ?- win(Window),
     get_font_attributes(Window, [height(H), max_width(W)]).

Window = window(2376568),
H = 25,
W = 23
```

The attribute `height/1` is the nominal height of the font, that is, its *declared* height. Individual characters can be taller than this, but they usually are not. The attribute `max_width/1` represents the width of the widest character in this font. Remember, different characters can have different widths. Notice that we've asked for the font attributes of a window. Once again, we take advantage of ProXL's ability to infer a font from a window.

Given these numbers, let's say that one fifth of the font's `max_width` and `height` are about the right horizontal and vertical offset for the drop shadow. So now recalculate the size of the window:

```
| ?- win(Window),
     get_font_attributes(Window, [height(Fh), max_width(Fw)]),
     Xdisplacement is Fw//5,
     Ydisplacement is Fh//5,
     text_extents(Window, 'Hello, world!!', L, R, W, A, D),
     Width is W+Xdisplacement+4,
     Height is A+D+Ydisplacement+4,
     put_window_attributes(Window, [size(Width,Height)]).

Window = window(2376568),
Fh = 25,
Fw = 23,
Xdisplacement = 4,
Ydisplacement = 5,
L = 0,
R = 145,
W = 146,
A = 17,
D = 3,
Width = 154,
Height = 29
```

This looks a bit complicated, but is actually just a combination of things we've done before. First we find the font's `max_width` and `height`, as above, and compute the X and Y displacement to be one fifth of those values. Then we find the size of the string, as before. The width and height of the window should be just as before, with the X and Y displacement added in. Finally, we resize the window. Not surprisingly, we lose the message again; let's put it back.

```
| ?- win(Window),
     Xdisplacement = 4,
     Ydisplacement = 5,
     A = 20,
     get_screen_attributes([black_pixel(Black),
                white_pixel(White)]),
     X0 is 2+Xdisplacement,
     Y0 is 2+A+Ydisplacement,
     put_graphics_attributes(Window, [foreground(Black)]),
     draw_string(Window, X0, Y0, 'Hello, world!!'),
     X1 is 2,
     Y1 is 2+A,
     put_graphics_attributes(Window, [foreground(White)]),
     draw_string(Window, X1, Y1, 'Hello, world!!').

Window = window(2376568),
Xdisplacement = 4,
Ydisplacement = 5,
A = 20,
Black = 1,
White = 0,
X0 = 6,
Y0 = 27,
X1 = 2,
Y1 = 22
```

Xdisplacement, *Ydisplacement*, and *A* come from previous goals, so we don't have to recompute them. We're also assuming the the left bearing of our message is 0. So the X position of our message is just 2, and the Y position is 2 plus the ascent of the message (A). And the X and Y position of the shadow message is offset from this by the X and Y displacement. The rest of this example is just like what we've seen before.

16.2.6 Specifying a Title for the Window

If you use a window manager that displays window titles, you probably would like to be able to give the window a title. In ProXL, this is done with the `property/2` window attribute.

```
| ?- win(Window),
     put_window_attributes(Window,
             [property('WM_NAME', hello)]).

Window = window(2376568)
```

This puts the string '`hello`' as the value of the *WM_NAME* property of our window. This property is watched by window managers that display window titles, and when it changes,

the window manager updates the title. There are many other properties that are significant to window managers, all discussed in Section 16.3.2 [pxl-win-wmi], page 779.

Some window managers do not display window titles, for example, `uwm`. If your window is like this, don't worry, the properties will not do any harm. And setting this property will make your programs friendlier for users with other window managers.

16.2.7 Color

Now let's add color. If you don't have a color machine, this will still work. Of course, you won't be able to see the colors, so if you can find a color display to run this example on, that would be best.

```
| ?- alloc_color(goldenrod, Pixel1),
     alloc_color(forestgreen, Pixel2),
     alloc_color(cyan, Pixel3),
     alloc_color(black, Pixel4),
     retractall(colors(_,_,_,_)),
     assert(colors(Pixel1,Pixel2,Pixel3,Pixel4)).

Pixel1 = 6,
Pixel2 = 7,
Pixel3 = 8,
Pixel4 = 1
```

Each line here allocates a color in the default colormap of the default screen. When you give `alloc_color/2` a valid color, it always returns a pixel value, even for black and white screens. If it can't allocate the color you ask for, it will give you the closest one it can. In this case, we have chosen the colors so that on a monochrome system the two background colors (*Pixel1* and *Pixel2*) will be different, and likewise the two foreground colors (*Pixel3* and *Pixel4*), so that the window will look reasonable. It wouldn't do if all the colors were the same.

If you're following along typing in these examples, you probably didn't get the same pixel values as we did here. That's why we assert the values: so we can get the right pixel values later when we need them.

Now let's construct a new background that uses these colors.

```
| ?- colors(Pixel1, Pixel2, _, _),
     create_pixmap(Pix, [size(4,4)]),
     put_graphics_attributes(Pix, [foreground(Pixel1)]),
     fill_rectangle(Pix, 0, 0, 3, 3),
     put_graphics_attributes(Pix, [foreground(Pixel2)]),
     draw_segments(Pix, [segment(0,0,3,3), segment(0,3,3,0)]),
     win(Window),
     put_window_attributes(Window,[background(Pix)]).

Pixel1 = 6,
Pixel2 = 7,
Pix = pixmap(586904)
```

The only thing here that's new is the call to `fill_rectangle/5`. We call it here to fill the pixmap with the appropriate background color. The rest of this has been discussed before in Section 16.2.4 [pxl-tut-dbg], page 762.

Finally, let's put the message back, only in color.

```
| ?- win(Window),
     colors(_, _, Pixel3, Pixel4),
     put_graphics_attributes(Window, [foreground(Pixel4)]),
     draw_string(Window, 6, 24, 'Hello, world!!'),
     put_graphics_attributes(Window, [foreground(Pixel3)]),
     draw_string(Window, 2, 19, 'Hello, world!!').

Window = window(2376568),
Pixel3 = 8,
Pixel4 = 1,
```

This is exactly what we did before, only now we specify `foreground/1` for the message, and the shadow, to take on the newly allocated colors.

16.2.8 Specifying a Cursor for the Window

The last complication we want to add to this example is a special cursor. Let's arrange for the cursor to look like gumby when it is in our window.

```
| ?- create_cursor(gumby, Cursor),
     put_window_attributes(window(1787368), [cursor(Cursor)]).

Cursor = cursor(1787472)
```

This is pretty much what one would expect. X predefines many cursors; Section 16.10 [pxl-crs], page 848 lists them. Gumby just happens to be one of them. We could custom design a cursor if we wished, but gumby should be fine for this example.

Try moving your mouse cursor into the hello window, and you will see that it becomes Gumby (or a reasonable facsimile).

16.2.9 Specifying a Callback Procedure for a Window Event

A callback is a Prolog procedure that is invoked when a particular window system event occurs. Examples of window system events are key presses, mouse clicks, mouse motion and window exposure.

A common ProXL application will create and map all of its windows and register a callback with each event to which the application must respond. Whenever Prolog is waiting for input, it watches for ProXL events and executes the registered callbacks.

16.2.9.1 Redrawing a window using a callback procedure

In our tutorial so far, every time our window has been damaged we have had to manually redraw it. The application program is always responsible for redrawing its windows when they have been damaged. This can be accomplished by associating a callback with the window telling ProXL what to do when the window needs to be redrawn.

The steps required to automatically redraw our window are:

- Define a callback procedure that draws the current contents of the window[3]
- Register that callback with the `expose` event of the window, so that the callback is invoked when the window needs to be redrawn

First, we must define the callback procedure to redraw our window:

[3] Since an `expose` event is sent to a window when it is first mapped (appears on the screen), it is not necessary to write separate code that initially draws the window and subsequently redraws the window.

```
| ?- compile(user).
| redraw(Window, Pixel3, Pixel4) :-
        Xdisplacement = 4,
        Ydisplacement = 5,
        A = 20,
        X0 is 2+Xdisplacement,
        Y0 is 2+A+Ydisplacement,
        X1 is 2,
        Y1 is 2+A,
        put_graphics_attributes(Window, [foreground(Pixel4)]),
        draw_string(Window, X0, Y0, 'Hello, world!!'),
        put_graphics_attributes(Window, [foreground(Pixel3)]),
        draw_string(Window, X1, Y1, 'Hello, world!!').
| ^D
% user compiled, 0.383 sec 380 bytes

yes
```

Next we must register the callback for `expose` events sent to our window:

```
| ?- win(Window),
     colors(_, _, Pixel3, Pixel4),
     put_window_attributes(Window,
            [callback(expose,
                    [count(0)],
                    redraw(Window,Pixel3,Pixel4))]).

Window = window(2376568),
Pixel3 = 8,
Pixel4 = 1
```

Here we have told ProXL that when the X server determines that it is necessary to redraw the window, by sending `expose` events, the procedure `redraw/3` should be called. The X server may send several `expose` events, each specifying a different part of the window to be redrawn. Since this is a simple example, we don't bother to redraw the window part by part. We just redraw the entire window whenever any part of it needs to be redrawn. The second argument, `count(0)`, accomplishes this by telling ProXL that it should call `redraw/3` only if this is the last `expose` message in the group.[4]

Now if you do something to force X to redraw our window, like iconifying and then uniconifying it, or hiding it behind another window and then exposing it, you will find that the window is automatically refreshed. Try it.

[4] If we had wanted to do more precise redrawing, we would have put `[position(X,Y),size(W,H)]` as the second argument in our callback specification, in place of `[count(0)]`, and passed X, Y, W, and H to the callback procedure. The callback procedure would then only redraw the area of the window thus specified.

16.2.9.2 `handle_events` and Terminating a Dispatch Loop

The last thing we need to learn to complete the example is how to wait for a ProXL condition. The example we have so far will keep our window refreshed indefinitely, but to make this a stand-alone demo we need to wait, handling refresh events, until the user indicates he is finished with this demo. Let's say when you click a mouse button in the window, we destroy the window and exit.

This is the role of the `handle_events/[0,1,2]` predicates. The simplest, `handle_events/0`, will wait, handling all ProXL events, until all windows with callbacks are destroyed. `handle_events/[1,2]` may be made to return before all windows have been destroyed, may be used to get information back from callbacks, and to pass context information to a callback (so it can behave differently in different contexts). But for this simple example, `handle_events/0` is perfectly adequate.

So all we need to do is arrange for our window to be destroyed when a mouse button is pressed in it:

```
| ?- win(Window),
     put_window_attributes(Window,
             [callback(button_press,
                    [],
                    destroy_window(Win))]).

Window = window(2376568)
```

Upon receiving a `button_press` event, the ProXL procedure `destroy_window/1` is invoked, destroying our window. At this point, `handle_events/0` will return.

Let's try it out:

```
| ?- handle_events.
```

Now press a mouse button while the cursor is in our window. The window should go away and `handle_events/0` should return.

16.2.10 The 'hello.pl' Program

The ProXL source code file listed below encapsulates all of the concepts presented in the previous session as a single program. This program appears in the file `demo('hello.pl')`.

hello.pl

```
:- module(proxl_hello, [hello/0]).

:- use_module(library(proxl)).


%  hello
%  test program for message_window/7.

hello :-
        chosen_font(Fontspec),
        current_font(Fontspec, Fontname),
        !,
        message_window('Hello, world!!', Fontname,
                goldenrod, forestgreen, cyan, black, _).


%  message_window(+Message, +Fontname, +Bg1, +Bg2, +Letters, +Shadow,
%       -Window)
%  Window is a window with Message, a Prolog atom, centered in it
%  in Fontname, an atom naming a font.  Bg1, Bg2, Letters and Shadow
%  are atoms naming colors.

message_window(Message, Fontname, Bg1, Bg2, Letters, Shadow, Window) :-
        load_font(Fontname, Font),
        message_size(Message, Font, Window_width, Window_height,
                Xoffset, Yoffset, Xdisplacement, Ydisplacement),
        alloc_color(Letters, Letters_pix),
        alloc_color(Shadow, Shadow_pix),
        background_pixmap(Bg1, Bg2, Bg),
        create_cursor(gumby, Cursor),
        create_window(Window,
                [   size(Window_width,Window_height), mapped(true),
                    border_width(2), background(Bg), cursor(Cursor),
                    property('WM_NAME', hello),
                    callback(expose, [count(0)],
                            expose_message(Window,Message,Letters_pix,
                                    Shadow_pix,Xoffset,Yoffset,
                                    Xdisplacement,Ydisplacement)),
                    callback(button_press, [], destroy_window(Window))
                ], [font(Font)]).
```

hello.pl

```
% message_size(+Message, +Font, -Window_width, -Window_height, -Xoffset,
%               -Yoffset, -Xdisplacement, -Ydisplacement)
%  Window_width and Window_height are the size of the smallest window
%  that will accomodate Message drawn with a drop shadow using font
%  Font.  Xoffset and Yoffset are the offset from the center of the
%  window at which we want to draw the string.  Since we want to keep
%  the message centered even when the window is resized, it's most
%  convenient to remember the offset from the center of the window,
%  which won't change.  Xdisplacement and Ydisplacement are the
%  distance the shadow should be displaced from the primary image,
%  computed as 1/5 of the font width and height, respectively.

message_size(Message, Font, Window_width, Window_height,
                Xoffset, Yoffset, Xdisplacement, Ydisplacement) :-
        get_font_attributes(Font, [height(Hei), max_width(Wid)]),
        Xdisplacement is Wid//5,
        Ydisplacement is Hei//5,
        text_extents(Font, Message, Lbearing, Rbearing, _, Asc, Desc),
        Xoffset is Lbearing-(Lbearing+Rbearing+Xdisplacement)//2,
        Yoffset is Asc-(Asc+Desc+Ydisplacement)//2,
        %  X and Y offset
        Window_width is Lbearing+Rbearing+Xdisplacement+4,
        Window_height is Asc+Desc+Ydisplacement+4.


%  background_pixmap(+Bg1, +Bg2, -Pixmap)
%  Pixmap is a newly allocated 4x4 background pixmap filled with our
%  background pattern.  Bg1 and Bg2 are the names of the colors to
%  use for this pixmap.

background_pixmap(Bg1, Bg2, Pixmap) :-
        alloc_color(Bg1, Bg1_pix),
        alloc_color(Bg2, Bg2_pix),
        create_pixmap(Pixmap, [size(4,4)], [foreground(Bg1_pix)]),
        fill_rectangle(Pixmap, 0, 0, 3, 3),
        put_graphics_attributes(Pixmap, [foreground(Bg2_pix)]),
        draw_segments(Pixmap, [segment(0,0,3,3),segment(0,3,3,0)]).
```

hello.pl

```
%  expose_message(+Window, +Message, +Letters_pix, +Shadow_pix,
%       +Xoffset, +Yoffset, +Xdisplacement, +Ydisplacement)
%  Redisplay the contents of Window.  Window is a window created by
%  message_window/7, and Message is the message displayed in it.
%  Letters_pix and Shadow_pix are the pixel values to draw the
%  letters and shadow in, respectively.  Xoffset and Yoffset are
%  the pixel offset from the center of the window at which Message
%  should be drawn.  And Xdisplacement and Ydisplacement are the
%  pixel offset from the message at which the shadow should be drawn.

expose_message(Window, Message, Letters_pix, Shadow_pix, Xoffset, Yoffset,
                Xdisplacement, Ydisplacement) :-
        get_window_attributes(Window, [size(Width,Height)]),
        X is Width//2 + Xoffset,        % compute position for message
        Y is Height//2 + Yoffset,
        Shadow_x is X+Xdisplacement,
        Shadow_y is Y+Ydisplacement,
        clear_window(Window),
        put_graphics_attributes(Window, [foreground(Shadow_pix)]),
        draw_string(Window, Shadow_x, Shadow_y, Message),
        put_graphics_attributes(Window, [foreground(Letters_pix)]),
        draw_string(Window, X, Y, Message).


%  chosen_font(-Fontname)
%  table of fonts to try.

chosen_font('*-times-bold-i-*-24-*').   % First choice, for X11R3
chosen_font('vgb-25').                  % Second coice, or on X11R2
chosen_font('fixed').                   % Last choice ...


%  user:runtime_entry(+Context)
%  The main program for runtime systems.

user:runtime_entry(start) :-
        hello,
        handle_events.                  % process callbacks till
                                        % hello window is destroyed
```

16.3 Windows

16.3.1 Window Attributes

The appearance and behavior of a window is largely determined by window attributes. The ProXL primitives `create_window/2`, `put_window_attributes/2`, and `get_window_attributes/2` form a family whose first argument is a windowable and whose second argument is a list of attributes. The first two primitives give attributes to a window, and the last allows you to inquire about the current state of a window.

The available attributes are listed here. The right hand column describes the attribute, lists the values of the variable (describing them when necessary), and indicates the default value of V.

Attribute **Description and values**

`parent(V)`
: This window's parent window. Defaults to the root window of default screen on window creation.

`x(V)`
: Left edge of window, relative to parent window's inside left edge, in pixels. Default is 0.

`y(V)`
: Top edge of window, relative to parent window's inside top edge, in pixels. Default is 0.

`position(X,Y)`
: *X* and *Y* of window relative to parent upper left corner, in pixels. Same as `x(X)`, `y(Y)`.

`width(V)`
: Inside width of window in pixels. Default is 100.

`height(V)`
: Inside height of window in pixels. Default is 100.

`size(W,H)`
: Same as `width(W)`, `height(h)`.

`depth(V)`
: Bits per pixel. Default is parent's depth. This attribute cannot be modified once a window is created.

`border_width(V)`
: width of window's border, in pixels. Default is 0.

`class(V)`
: Can this window be drawn in, or is it only for getting input? *V* is either:
  - `input_output` (default)
  - `input_only`

  This attribute cannot be modified once a window is created.

`visual(V)`
: The window's visual. Default is the visual of the window's parent's screen. See Section 16.8 [pxl-col], page 838. This attribute cannot be modified once a window is created.

`background(V)`

The window's background. *V* may be

- a pixel
- a pixmap
- `none` (default)
- `parent_relative`

Due to the design of the X window system itself, this attribute cannot be determined, but only modified.

`border(V)`

The pattern to be displayed in the window's border. *V* may be

- a pixel
- a pixmap
- `copy_from_parent` (default)

Due to the design of the X window system itself, this attribute cannot be determined, but only modified.

`bit_gravity(V)`

Where to put the contents of the window if it is resized. *V* may be one of

- `north_west`
- `north`
- `north_east`
- `west`
- `center`
- `east`
- `south_west`
- `south`
- `south_east`
- `static`
- `forget` (default)

`win_gravity(V)`

Where to put the subwindows of a window that has been resized. *V* may be one of

- `north_west`
- `north`
- `north_east`
- `west`
- `center`
- `east`
- `south_west`
- `south`

- `south_east`
- `static`
- `unmap` (default)

`backing_store(V)`
: Should the contents of this window be saved by the server when it is occluded by another window? *V* should be one of:

  - `not_useful` (default)
  - `when_mapped`
  - `always`

  Note that some screens don't support backing store, and even those that do may not always provide it when asked. Your application must *always* be prepared to repaint its own windows.

`backing_bit_planes(V)`
: If `backing_store` is used, which planes should be saved? *V* is a bit mask. The default is to save all planes (i.e., *V* = -1).

`backing_pixel(V)`
: Specifies the bit values to put into unbacked planes when restoring from backing store. Default is 0.

`save_under(V)`
: Should what is under this window be saved so that when the window is moved or unmapped, the newly exposed part of the screen can be refreshed without asking any applications to do the work? Like `backing_store(V)`, not all screens support this, and those that do might not support it whenever asked to. *V* may be either:

  - `true`
  - `false` (default)

`event_mask(V)`
: An integer bitmask indicating which events this window wants to handle. Note that if you use the ProXL callback mechanism, you should not modify this attribute directly.

`do_not_propagate_mask(V)`
: An integer bitmask indicating which events not to propagate. Any event destined for this window not specified by either `event_mask(V)` or `do_not_propagate_mask(V)` will be forwarded to this window's parent.

`override_redirect(V)`
: Override redirection of map and configure requests? Possible values of *V* are

  `false` (default) A program, usually a window manager, can specify that requests to map or reconfigure a child of a certain window, usually the root, should be sent to it. This allows a window manager to decide not to allow top level windows to be moved or reshaped, and to put title bars and special borders on top level windows.

`true` Map and configure requests should go directly to the server. This allows you to pop up a window, say a menu, where you want it, and without any adornments the window manager would put on it.

`colormap(V)`
V may be either:

- the colormap for this window
- `copy_from_parent` (default)

`cursor(V)`
V may be either:

cursor the cursor to display when the cursor is in this window

`none` the parent window's cursor will be displayed.

Due to the design of the X window system itself, this attribute cannot be determined, but only modified.

`mapped(V)`
Is this window on the screen, providing that its parent is? Possible values are

`false` (default)

`true`

`viewable` the window *is* on the screen; that is, the window is mapped, and its parent is viewable. The root window is always viewable.

`gc(V)` The default graphics context for this window. The default is the screen's default gc. See Section 16.6 [pxl-graf], page 827 for more information.

`property(N,V)`
V is the value of the *N* property of the window. If *N* is unbound, it will backtrack through all the properties of the window. If *N* is bound, it should be an atom. See Section 16.3.2 [pxl-win-wmi], page 779 for information on how to use the `property(N,V)` attribute to tell a window manager how to handle your windows.

`callback(E,F,V,C,G)`
Register *G* as the goal to call when event *E* is received by the window. *F* is a list of fields of the event to be accessed before calling *G*, and *V* is a variable, which may be passed back to the `handle_events` procedure in order to exit the `handle_events` loop. If *V* is `none`, then no result will be passed to `handle_events`. *C* is the context of the call, which is specified as the second argument to `handle_events/2`. This allows a callback's behavior to depend on the context in which it occurs. See Section 16.4 [pxl-ev], page 789 for more information.

`callback(E,F,V,G)`
Equivalent to `callback(E,F,V,_,G)`.

`callback(E,F,G)`
Equivalent to `callback(E,F,none,_,G)`.

16.3.2 Window Manager Interaction: Properties

In the X Window System, window managers largely control how an application's windows behave. Applications inform the currently active window manager of important facts and hints about a window by storing properties of certain names and types on that window. Each window manager determines what it will do with this information by itself; your application does *not* have any control over it.

X does establish many conventions for what window managers are expected to do, and you are encouraged to use those conventions. However, at this stage in the development of the X Window System, most window managers do not follow all these conventions. Keep this in mind when you are testing code that makes demands of window managers: your test may be failing due to no fault in your program, but simply because the window manager you are using does not support the feature you require.

All interaction with the window manager is done through window properties. As described above (see Section 16.3.1 [pxl-win-atts], page 775), window properties are window attributes of the form

```
property(Name,Value)
```

These can be set with `create_window` and `put_window_attributes`.

16.3.2.1 Giving the Window a Name

The attribute `property('WM_NAME',Name)` specifies that *Name*, which should be a Prolog atom, will be the name of the window it is an attribute of. Remember that not all window managers will display a window's name.

16.3.2.2 Giving the Window's Icon a Name

The attribute `property('WM_ICON_NAME',Name)` specifies that *Name*, which should be a Prolog atom, will be the name of the window's icon.

16.3.2.3 Suggesting a Size and Shape for the Window

There are many possible ways to suggest to a window manager the screen size and position of your window. First, you may specify two separate sets of size and position hints. The *normal* size hints are specified by the attribute

```
property('WM_NORMAL_HINTS',Hints)
```

The *zoomed* size hints, that is, the large or *full screen* size hints, are specified by the attribute

```
property('WM_ZOOM_HINTS',Hints)
```

In both cases, the *Hints* specified should be a term of the form

```
wm_size_hints(Position,Size,Min_size,Max_size,Resize_inc,Aspect)
```

The meaning of the arguments of the `wm_size_hints/6` term is as follows:

Argument **Meaning**

Position Specifies the window's position. It may be either:

`none` No hint is being made about the window's position.

`user_position(X,Y)`
This means that `position` *(X,Y)* is suggested, and that the suggestion comes from the user.

`program_position(X,Y)`
This means that `position` *(X,Y)* is suggested, and that the suggestion comes from the program.

Size Specifies the window's size. It may be either:

`none` meaning that no hint is being made about the window's size.

`user_size(W,H)`
W by *H* is the suggested size, and the suggestion comes from the user.

`program_size(W,H)`
W by *H* is the suggested size, and the suggestion comes from the program.

Min_size Specifies the minimum suggested size for the window. The window manager may choose to make the window smaller than this, if it likes. This may be either:

`none` No hint is being made about the window's minimum size.

`size(W,H)`
W by *H* is the suggested minimum size.

Max_size Specifies the maximum suggested size for the window. The window manager may choose to make the window larger than this, if it likes. This may be either:

`none` No hint is being made about the window's maximum size.

`size(W,H)`
W by *H* is the suggested maximum size.

Resize_inc Specifies the increments in which your application would like the window to be resized. This may be either:

`none` No hint is being made about the window's resize increment.

`size(W,H)`
The window should be made wider or narrower in *W* pixel increments and taller or shorter in *H* pixel increments.

Aspect Specifies the minimum and maximum aspect ratio you would like the window to have. This may be either:

`none` No hint is being made about the window's aspect ratio.

`Min-Max` The window's aspect ratio should be between *Min* and *Max*. *Min* and *Max* should be (possibly floating point) numbers specifying the ratio of width to height.

You may specify as many or as few of these arguments as you like, using `none` to avoid specifying a value.

16.3.2.4 Suggesting Icon, Initial State, and Other Features

To specify whether your window expects to get focus (keyboard ownership) from the window manager, how your window should appear initially, an picture to use in an icon, the icon's position, and what window group your window is in, specify the attribute

```
property('WM_HINTS',Hints)
```

Hints should be a term of the form

```
wm_hints(Input,Initial,Icon_pix,Icon_win,Icon_pos,Icon_mask,Window_group
)
```

The meaning of the arguments of the `wm_hints/7` term is as follows:

Argument **Meaning**

Input Specifies whether your window expects to get focus (keyboard ownership) from the window manager. Possible values are:

`none` No hint is being made about your about your application's input needs.

`true` Your application expects the window manager to give it focus.

`false` The application will grab the focus when it wants it, or that it never needs focus.

Initial Specifies the desired initial state of your window. Possible values are:

`none` No hint is being made about the initial state of your application.

`dont_care`
The application does not care how it starts up.

`normal` The window wants to come up in *normal* state, using its normal hints.

`zoom` The window would like to start out *zoomed*.

`iconic` The window would like to start out iconified.

`inactive` The application is not often used, so your window manager may want to put it in a special *inactive* menu.

Icon_pix Specifies the image you'd like to have used in your window's icon. It may be either:

`none` You are not specifying an icon mask.

a one plane pixmap

See Section 16.3.2.6 [pxl-win-wmi-isz], page 783 below for information about how to find out how big to make your pixmap.

Icon_win Specifies a window you'd like to have used as your window's icon. It may be either:

`none` You are not specifying an icon window.

a window

Icon_pos Specifies your suggested initial icon position. It may be either:

`none` You are not specifying an icon position.

`position(X,Y)`

Icon_mask Specifies a suggested mask to use in conjunction with *Icon_pix* to determine the icon's image. This allows for non-rectangular icons on window managers that support them. It may be either:

`none` You are not specifying an icon image.

a one plane pixmap

Window_group
Specifies a group of windows that should be iconified together. It may be either:

`none` You are not specifying a window group.

a window

16.3.2.5 Transient windows

To tell your window manager that a particular window is transient, for example a dialog box, you should give it an attribute

```
property('WM_TRANSIENT_FOR',Window)
```

This means that your window is transient, and, in some sense, *belongs to Window*.

16.3.2.6 Icon Sizes

Some window managers will only support certain icon sizes. These window managers store an attribute

```
property('WM_ICON_SIZE', wm_icon_size(List))
```

on the screen's root window. You may use `get_window_attribute` to find out these icon sizes. *List* will be a list of terms of the form

```
size_range(Min_w,Min_h,Max_w,Max_h,W_inc,H_inc)
```

where *Min_w* and *Min_h* specify a minimum icon width and height, *Max_w* and *Max_h* specify a maximum icon width and height, and *W_inc* and *H_inc* specify a width and height increment. For example, if *List* were

```
[size_range(64,64,64,64,0,0), size_range(32,32,48,48,8,8)]
```

this would mean that legal sizes are 32 by 32, 40 by 40, 48 by 48, and 64 by 64.

You should not need to set this attribute, though you could if you wanted to.

16.3.2.7 Other Window Properties

The `property(N,V)` window attribute is not limited to communicating with window managers. It may be used for attaching arbitrary data to a window. It is important to note that getting window properties is expensive, as it requires a round trip to the X server. However properties are an effective way for multiple applications in separate address spaces to communicate.

As described in Section 16.3.1 [pxl-win-atts], page 775, the name argument *N* of a property attribute may be any atom. The value argument *V* may be a pixmap, a colormap, a cursor, a font, an integer, a pixmap, an atom, a visual term, a window, or one of the following terms:

Value	Meaning
`arc/6`	Describes an arc. See Section 16.5.6 [pxl-prim-arcs], page 824 for an explanation.
`atom(Term)`	*Term* must be an atom. To Prolog, this is much the same as simply specifying *Term*, but other applications may distinguish between atoms and strings.

`cardinal(`*Int*`)`
: *Int* must be a non-negative integer. To Prolog this is much like simply specifying *Int*, but other applications may distinguish between signed and unsigned integers.

`point/2`
: Describes a point or position. See Section 16.5.2 [pxl-prim-pnt], page 821 for an explanation.

`rectangle/4`
: Describes a rectangle or region. See Section 16.5.5 [pxl-prim-rect], page 823 for an explanation.

`wm_hints/7`
: As described in Section 16.3.2.4 [pxl-win-wmi-wini], page 781.

`wm_size_hints/6`
: As described in Section 16.3.2.3 [pxl-win-wmi-wsiz], page 779.

`wm_icon_size/1`
: As described in Section 16.3.2.6 [pxl-win-wmi-isz], page 783.

16.3.3 Creating and Destroying Windows

The remainder of the section describes ProXL primitives.

16.3.3.1 `create_window/[2,3]`

```
create_window(-Window, +Attribs)
create_window(-Window, +Attribs, +Graphics_attribs)
```

Window is a newly-created window, and *Attribs* is a ground list of window attributes. If *Graphics_attribs* is given, it is a list of graphics attributes to be given to the window. The predicate `create_window/3` is as if defined by

```
create_window(Window, Attribs, Graphics_attribs):-
        create_window(Window, Attribs),
        put_graphics_attributes(Window, Graphics_attribs).
```

but it is more efficient.

See Section 16.6 [pxl-graf], page 827 for information about graphics attributes.

16.3.3.2 `destroy_window/1`

```
destroy_window(+Window)
```

Deallocate space for *Window*. *Window* should not be referred to anymore. Note that *Window* is not actually deallocated, but only put on a dead list, which can later be cleaned up by `clean_up/0`. Also note that many things can be destroyed, but windows are special

in that they are not marked as destroyed by this procedure, but are marked when they get an event indicating that *Window* has been destroyed. This is because a window may be destroyed by other procedures.

16.3.3.3 `destroy_subwindows/1`

```
destroy_subwindows(+Window)
```

Deallocate space for all subwindows of *Window*. As with `destroy_window/1`, destroyed subwindows are put on a dead list. See Section 16.3.3.2 [pxl-win-cre-destroy_window], page 784

16.3.4 Finding and Changing Window Attributes

16.3.4.1 `get_window_attributes/[2,3]`

```
get_window_attributes(+Windowable, +Attribs)
get_window_attributes(+Windowable, +Attribs, +Graphics_attribs)
```

Windowable is a windowable, and *Attribs* is a proper list of window Attribute settings. Note that *Attribs must* be a proper list. If an element of *Attribs* is an unbound variable, `get_window_attributes` will backtrack through all the window attributes of *Windowable*.

If *Graphics_attribs* is given, it is a proper list of graphics attributes of *Windowable*. The predicate `get_window_attributes/3` is as if defined by the following procedure, but it is more efficient.

```
get_window_attributes(Window, Attribs, Graphics_attribs):-
     get_window_attributes(Window, Attribs),
     get_graphics_attributes(Window, Graphics_attribs).
```

See Section 16.6 [pxl-graf], page 827 for information about graphics attributes.

16.3.4.2 `put_window_attributes/[2,3]`

```
put_window_attributes(+Windowable, +Attribs)
put_window_attributes(+Windowable, +Attribs, +Graphics_attribs)
```

Windowable is a windowable, and *Attribs* is a ground list of attributes. The window associated with *Windowable* is changed such that *Attribs* is a list of its attributes. There are a few attributes of a window that may not be changed once the window is created. These are listed above, with the descriptions of the actual attributes. Of course, if you specify a screen or display for *Windowable*, it doesn't make sense to change very many attributes, and it is recommended that you not change any.

If *Graphics_attribs* is given, it is a list of graphics attributes to be given to the window. The predicate `put_window_attributes/3` is as if defined by the following procedure, but it is more efficient.

```
put_window_attributes(Windowable, Attribs, Graphics_attribs):-
     put_window_attributes(Windowable, Attribs),
     put_graphics_attributes(Window, Graphics_attribs).
```

See Section 16.6 [pxl-graf], page 827 for information about graphics attributes.

16.3.4.3 `rotate_window_properties/[2,3]`

```
rotate_window_properties(+Properties, +Rotation)
rotate_window_properties(+Windowable, +Properties, +Rotation)
```

Rotate the values of *Properties*, a list of property names (atoms) on the window associated with *Windowable* (default is the root window of the default screen). Items are moved toward the front of the list. For example, if *Rotation* is 2, and *Properties* is

```
[prop1,prop2,prop3,prop4,prop5]
```

rotation changes *Properties* to

```
[prop3,prop4,prop5,prop1,prop2]
```

This predicate might be used, for example, to implemement a ring of selection service cut buffers.

16.3.4.4 `delete_window_properties/[1,2]`

```
delete_window_properties(+Properties)
delete_window_properties(+Windowable, +Properties)
```

Remove *Properties*, a list of window property name atoms, from the window associated with *Windowable*, which defaults to the default screen.

16.3.4.5 `map_subwindows/1`

```
map_subwindows(+Window)
```

Map all subwindows of *Window*.

16.3.4.6 `unmap_subwindows/1`

```
unmap_subwindows(+Window)
```

Unmap all subwindows of *Window*.

16.3.5 Miscellaneous Window Primitives

16.3.5.1 `restack_window/2`

```
restack_window(+Window, +Stackmode)
```

Move *Window* in stacking order as indicated by the value of *Stackmode*:

Value	Meaning
`top`	Move *Window* to the top of stack.
`above(Win2)`	Move *Window* just above *Win2*.
`bottom`	Move *Window* to the bottom of stack.
`below(Win2)`	Move *Window* just below *Win2*.
`top_if`	Move *Window* to the top of stack if any other window occludes it.
`top_if(Win2)`	Move *Window* to the top if *Win2* occludes it.
`bottom_if`	Move *Window* to the bottom of stack if it occludes any other window.
`bottom_if(Win2)`	Move *Window* to the bottom if it occludes *Win2*.
`opposite`	Move *Window* to the top if any window occludes it, otherwise move it to the bottom if it occludes any window.
`opposite(Win2)`	Move *Window* to the top if *Win2* occludes it, otherwise move *Window* to the bottom if it occludes *Win2*.

16.3.5.2 `window_children/[1,2]`

```
window_children(-Children)
window_children(+Windowable, -Children)
```

Children is the list of all *Windowable*'s children, in top-to-bottom stacking order. *Windowable* defaults to the root window of the default screen.

16.3.5.3 `current_window/[1,2]`

```
current_window(?Window)
current_window(?Window, ?Display)
```

Tells whether *Window* is a window that is known to ProXL. If *Display* is specified, *Window* is on that display. If *Window* is not bound, `current_window` backtracks through all windows.

A window is *known to ProXL* if it was created by ProXL, or if it was passed to ProXL by the X server or a foreign X procedure. For example, if you call `current_window/1` in a fresh ProXL session, few windows will be returned. If you call `window_children/1` before calling `current_window/1`, many more windows will be returned, because the call to `window_children/1` asks ProXL to find all the top level windows on the default screen, most of which ProXL won't have seen before.

16.3.6 Selections

16.3.6.1 `set_selection_owner/[2,3,4]`

```
set_selection_owner(+Selection, +Owner)
set_selection_owner(+Selection, +Owner, +Time)
set_selection_owner(+Displayable, +Selection, +Owner, +Time)
```

Set the owner of *Selection* on *Displayable* to be *Owner* at *Time*. *Selection* must be an atom, which names the selection to be owned. *Owner* must be a valid window. *Time* must be `current_time` or a timestamp in milliseconds. *Time* defaults to `current_time`.

16.3.6.2 `get_selection_owner/[2,3]`

```
get_selection_owner(+Selection, -Window)
get_selection_owner(+Displayable, +Selection, -Window)
```

Get the owner of *Selection* on *Displayable*. *Displayable* defaults to the default display.

16.3.6.3 `convert_selection/[4,5,6]`

```
convert_selection(+Selection, +Target, +Property, +Requestor)
convert_selection(+Selection, +Target, +Property, +Requestor, +Time)
convert_selection(+Displayable, +Selection, +Target, +Property, +Re-
questor, +Time)
```

Convert *Selection* on *Displayable* to *Target* type, storing it under *Property* on window *Requestor* at *Time*. *Time* must be `current_time` or a timestamp in milliseconds. *Time* defaults to `current_time`. *Displayable* defaults to the default display.

16.3.7 Checking Window Validity

16.3.7.1 valid_window/1

```
valid_window(+Window)
```

Window is a valid window. I.e., it has not been destroyed.

16.3.7.2 valid_windowable/2

```
valid_windowable(+Windowable, -Window)
```

Window is the valid window associated with *Windowable*. *Windowable* must be a window, screen, or display.

16.3.7.3 ensure_valid_window/2

```
ensure_valid_window(+Window, +Goal)
```

Window is a valid window. If it is not, an error message mentioning *Goal* is printed, and execution aborts.

16.3.7.4 ensure_valid_windowable/3

```
ensure_valid_windowable(+Windowable, -Window, +Goal)
```

Window is the valid window associated with *Windowable*. *Windowable* must be a valid window, screen, or display. If it is not, an error message mentioning *Goal* is printed, and execution aborts.

16.4 Events and Callbacks

In X11, the server communicates changes in the environment to the clients by sending *event* messages to them. Clients indicate specific interest in certain events by *selecting* them. Each X11 window has a bitmask that indicates the events it is interested in receiving and a bitmask that indicates the events it wants discarded.

ProXL extends the notion of X11 events by allowing the user to register Prolog *callback* routines with each window. In this section we discuss the basic notions of events and the callback mechanism.

16.4.1 Introduction

Under X11, events are selected on a per window basis by ORing individual *event select* bitmasks into the `event_mask` attribute of the window. To determine which window gets

an event, the X server searches the window hierarchy bottom up, starting with the innermost window where the event logically happened. If a window *selects* that particular event, it is sent to it. If the window *discards* that event, the event is thrown away and the search stops. If the window neither selects nor discards the event, the search resumes with its parent. Any event that propagates all the way up to the root window without being selected, is discarded anyway.

When using ProXL, in most cases, the user does not need to know about bitmasks. Events are selected by name, and Prolog routines are attached to Windows to handle the conditions.

The ProXL event handler mechanism receives X11 events, extracts user-specified values and calls the appropriate callback routine.

If there is no registered callback for a given event (including *default* handlers, as specified later), the event is quietly discarded.

If multiple callbacks are registered for an event in a Window (as might be the case for `button_press` or `button_release` events), they are tried sequentially until one succeeds, at which point the callback is considered satisfied and no more alternatives are tried. If all the registered callbacks fail, the event is quietly discarded.

Callbacks are Window attributes, and are established using `put_window_attributes/N`, or `create_window/N`, using one of the following attribute formats:

```
callback(+EventSpec, +EventValues, +ExitVar, +Context, +Goal)
callback(+EventSpec, +EventValues, +ExitVar, +Goal)
callback(+EventSpec, +EventValues, +Goal)
```

where *EventSpec* is a description of the event that should cause the user-supplied callback *Goal*, a Prolog goal, to be called.

EventValues is either a list of the event fields that the user wants to be supplied as arguments to *Goal* , or the term `xevent(E)`, which instructs ProXL to deliver the whole event structure, in the same internal format used by the event handling functions described in Section 16.12 [pxl-evf], page 856.

ExitVar, if supplied, is either a term that is bound by *Goal* when the user wants to exit the event handling loop and return, or the atom `none`. If omitted, it defaults to `none` and *Goal* is assumed to not contain an exit variable.

Context, if supplied, is a term that will be unified with the *Context* argument supplied to `handle_events/[2,3]` when the callback *Goal* is executed. If omitted, it defaults to `none`. If the user calls `handle_events/[0,1]`, *Goal* will be called with *Context* unbound.

16.4.2 Event Specification

Basically, events can be classified under four categories:

1. Those in which a single mask uniquely selects an event.
2. Those in which a single mask selects a number of events at once.
3. Those in which a number of masks might be required to select an event.
4. Those that are always selected and do not require a mask.

This section describes the valid terms that can be used as an *EventSpec* and their meanings. ProXL provides some additional discrimination mechanisms on top of X11 events that allow succinct specification of commonly selected events.

Note that some servers might not supply some of the events, if the underlying hardware can not support it, e.g. `key_release`. Event records have a set of common fields and event-specific fields. This section briefly describes each event, and its ProXL specification. The next section describes in detail the event fields.

16.4.2.1 Events uniquely selected by a single mask

Key Press Event. The server generates a `key_press` event for every key that is pressed, including modifier keys.

Specify with:

```
key_press
```

Key Release Event. The server generates a `key_release` event for every key that is released, including modifier keys.

Specify with:

```
key_release
```

Button Press Event. The server generates a `button_press` event for every mouse button that is pressed.

If specified without options, it indicates any mouse button press:

```
button_press
```

To select any of a particular set of mouse buttons pressed, use:

```
button_press(+ButtonList)
```

Where *ButtonList* is a list that can contain the integers `1` to `5`. As a special case, the empty list degenerates into the first specification.

Button Release Event. The server generates a `button_release` event for every mouse button that is released.

If specified without options, it selects any mouse button release:

```
button_release
```

To select any of a particular set of mouse buttons released, use:

```
button_release(+ButtonList)
```

Where *ButtonList* is a list that can contain the integers `1` to `5`. As a special case, the empty list degenerates into the first specification.

Enter Notify Event. The server generates an `enter_notify` event when the mouse pointer enters a window or it is *virtually entered* by the pointer moving between two windows that do not have a parent-child relationship.

Specify with:

```
enter_notify
```

Leave Notify Event. The server generates a `leave_notify` event when the mouse pointer leaves a window or it is *virtually leaved* by the pointer moving between two windows that do not have a parent-child relationship.

Specify with :

```
leave_notify
```

Keymap Event. The server generates a `keymap_notify` event immediately after each `enter_notify` or `focus_in` event, as a way for the application to read the keyboard state.

Specify with:

```
keymap_notify
```

Expose Event. The server generates `expose` events when a window becomes visible or previously invisible parts of a window become visible.

Specify with:

```
expose
```

Colormap Notify Event. The server generates a `colormap_notify` event when the colormap changes.

Specify with:

```
colormap_notify
```

Property Notify Event. The server generates a `property_notify` event when a window property changes.

Specify with:

```
property_notify
```

Visibility Notify Event. The server generates a `visibility_notify` event when there is any change in the visibility of the specified window.

Specify with:

```
visibility_notify
```

Resize Request Event. The server generates a `resize_request` event when there is any attempt to change the size of a window. This event is usually selected by the Window Manager to intercept resize attempts and modify the request according to its policies.

Specify with:

```
resize_request
```

16.4.2.2 Events that come in *pairs* selected by a single mask

In this case, when selecting the event, you get a complementary pair automatically. In other words, if you are interested in one of them, you probably are interested (or should be) in the other too.

Focus In and Focus Out events. Both of these events are selected internally with a single mask, `focus_change`, so even if you register a callback for only one, your window will also get the other one. You should always register a callback for both.

The server generates a `focus_in` event when the keyboard focus window changes as a result of an explicit `set_input_focus` call. The window that receives this event has the keyboard focus and will be receiving all keyboard input until it loses the focus. Specify with:

```
focus_in
```

The server generates a `focus_out` event when the keyboard focus window changes as a result of an explicit `set_input_focus` call. The window that receives this event has lost the keyboard focus and therefore will not be receiving any more keyboard input. Specify with:

```
focus_out
```

Graphics Events. These events present a special problem, as they are not selected by the window's `event_mask` attribute, but by the GC `graphics_exposure`. Also, they apply to both Windows and Pixmaps.

These callbacks are **not** part of the of the Window, Pixmap or GC attributes . They are established and de-established dynamically as an optional parameter to the `copy_area` and `copy_plane` calls.

The `graphics_expose` event is generated by `copy_area` or `copy_plane` when the source area is not available because the region is clipped or obscured. Specify with

```
graphics_expose
```

The server generates a `no_expose` event when the source area for a `copy_area` or `copy_plane` request was completely available, and therefore the request was carried out successfully. Specify with:

```
no_expose
```

16.4.2.3 Multiple events selected by a single mask

In this case, there is a whole family of events that are selected by a single mask, `substructure_redirect`. By registering a callback for one you will also get the others.

These events are typically selected by the Window Manager to intercept and modify top level window requests according to its policies.

Circulate Request Event. The server generates a `circulate_request` event when there is any attempt to change the stacking order of a window. Specify with:

```
circulate_request
```

Configure Request Event. The server generates a `configure_request` event when there is any attempt to change the configuration of a window. Specify with:

```
configure_request
```

Map Request Event. The server generates a `map_request` event when the functions `map_raised` and `map_window` are called, so mapping attempts can be intercepted. Specify with:

```
map_request
```

16.4.2.4 Multiple events selected by different masks

Two different masks, `structure_notify` and `substructure_notify` select essentially the same set of events, except that in the first case, the user gets the events that occur in the Window and in the second, the events that occur in its subwindows.

To accommodate this, the event specification has an extra argument, whose value is either `self` or `child`. If omitted, it defaults to `self`, except for the case of `create_notify` as explained below.

Remember that registering a callback for any of these events means that the Window will also be getting the others.

Circulate Notify Event. The server generates a `circulate_notify` event when a window is restacked.

To receive this event when the window itself is affected, specify either:

```
circulate_notify
circulate_notify(self)
```

To receive this event when a child window is affected, specify:

```
circulate_notify(child)
```

Configure Notify Event. The server generates a `configure_notify` event when there is any change to a window's configuration, i.e. position, size, etc.

To receive this event when the window itself is affected, specify either:

```
configure_notify
configure_notify(self)
```

To receive this event when a child window is affected, specify:

```
configure_notify(child)
```

Destroy Notify Event. The server generates a `destroy_notify` event when a window is destroyed.

To receive this event when the window itself is affected, specify either:

```
destroy_notify
destroy_notify(self)
```

To receive this event when a child window is affected, specify:

```
destroy_notify(child)
```

Gravity Notify Event. The server generates a `gravity_notify` event when a window is moved because its parent's size changed.

To receive this event when the window itself is affected, specify either:

```
gravity_notify
gravity_notify(self)
```

To receive this event when a child window is affected, specify:

```
gravity_notify(child)
```

Map Notify Event. The server generates a `map_notify` event when a window changes state from unmapped to mapped.

To receive this event when the window itself is affected, specify either:

```
map_notify
map_notify(self)
```

To receive this event when a child window is affected, specify:

```
map_notify(child)
```

Unmap Notify Event. The server generates an `unmap_notify` event when a window changes state from mapped to unmapped.

To receive this event when the window itself is affected, specify either:

```
unmap_notify
unmap_notify(self)
```

To receive this event when a child window is affected, specify:

```
unmap_notify(child)
```

Reparent Notify Event. The server generates a `reparent_notify` event when the parent of a window changes.

To receive this event when the window itself is affected, specify either:

```
reparent_notify
reparent_notify(self)
```

To receive this event when a child window is affected, specify:

```
reparent_notify(child)
```

Create Notify Event. The server generates a `create_notify` event when a window is created. A newly created window does not receive this event, its parent does. Specify with either:

```
create_notify
create_notify(child)
```

16.4.2.5 Single events selected by multiple masks

The `motion_notify` event is selected by a number of masks, including the `motion_hint` mask [5].

In ProXL they are separated into two different specifications, which are mutually exclusive.

Motion Notify Event. In this case, the server will steadily generate `motion_notify` events while the mouse is moving, or the program warps it, but only if the motion of the pointer begins and ends in the same window. The number of events generated can easily overwhelm your application, if it has to do any significant amount of computation for each one.

[5] Specifies that the user is only interested in knowing when there has been a movement, but does not need to know the complete path of the mouse pointer

If specified without options, the events will be generated independently of the state of the mouse buttons:

```
motion_notify
```

To specify mouse movements only while any of a particular set of mouse buttons are pressed, use:

```
motion_notify(+ButtonList)
```

Where *ButtonList* is a list that can contain the integers `1` to `5`. As a special case, the empty list degenerates into the first specification.

Motion Notify Hint Event. In this case, the server will generate only one `motion_notify` event to let your application know that the mouse pointer moved. You have to explicitly query the pointer using `get_pointer_attributes` to find out where the mouse pointer ended up.

If specified without options, the event will be generated independently of the state of the mouse buttons:

```
motion_notify_hint
```

To specify mouse movement only while any of a particular set of mouse buttons are pressed, use:

```
motion_notify_hint(+ButtonList)
```

Where *ButtonList* is a list that can contain the integers `1` to `5`. As a special case, the empty list degenerates into the first specification.

16.4.2.6 Events that are always selected

This category includes all the events that X11 will send to your application, regardless of whatever you want them or not. Well behaved programs must be able to handle them appropriately.

Mapping Notify Event. The server generates a `mapping_notify` event when any of the following are changed by another client:

- The mapping between keyboard *keycodes* and *keysyms*.
- The mapping between physical modifier keys and logical modifiers.
- The mapping between physical mouse buttons and logical buttons.

ProXL handles the necessary adjustments to your environment automatically, but you should be aware that it has changed. Specify with:

```
mapping_notify
```

Client Message Event. A `client_message` event is generated when clients use `send_event`, to communicate with your application. Specify with:

```
client_message
```

Selection Clear Event. The server generates a `selection_clear` event when a new owner is defined for a property. The current owner of the selection gets this event. Specify with:

```
selection_clear
```

Selection Notify Event. A `selection_notify` event is sent by a client, not the server. The owner of a selection sends it to a requester to announce that the selection has been converted to the appropriate format and stored as a property, or that the conversion could not be performed.

Specify with:

```
selection_notify
```

Selection Request Event. A `selection_request` event is generated when another client requests the selection owned by the window.

Specify with:

```
selection_request
```

16.4.3 Event Fields

Callbacks can optionally specify a list of field values that are to be unified with the contents of the X11 event record, and/or used as arguments to the callback goal. ProXL handles all the conversion between the C objects and data structures that the X server delivers, and the Prolog objects and data structures that your callback is given. Thus, your application is given symbolic atoms instead of numbers whenever possible, and furthermore, it is safe for your application to assert these objects in the data base.

This section describes the field specifications that can be used for each event and the possible values that these values can take. See the X11 documentation for more details on what individual values mean.

16.4.3.1 `button_press` and `button_release` Events

The fields that can be unified in `button_press` and `button_release` events are:

1. `type(`*T*`)` unifies *T* with either, `button_press` or `button_release`, depending on the event.
2. `serial(`*S*`)` unifies *S* with the serial number of the last request processed by the server.

3. `send_event(B)` unifies B, a boolean value, with one of:

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that the event is reported to.
6. `root(R)` unifies R with the root ProXL Window that the event occurred under.
7. `subwindow(S)` unifies S with the child ProXL window where the event occurred or `none`If the event occurred in the window itself.
8. `time(T)` unifies T with the server time, in milliseconds, when the event occurred.
9. `x(X)` unifies X with the x pointer coordinate, relative to the window origin.
10. `y(Y)` unifies Y with the y pointer coordinate, relative to the window origin.
11. `position(X, Y)` unifies X and Y with the pointer x and y coordinates, respectively, relative to the window origin.
12. `x_root(X)` unifies X with the x coordinate, relative to the root window origin.
13. `y_root(Y)` unifies Y with the y coordinate, relative to the root window origin.
14. `root_position(X, Y)` unifies X and Y with the x and y coordinates, respectively, relative to the root window origin.
15. `state(Buttons, Modifiers)` unifies *Buttons* with a term of the form:

    ```
    buttons(B1, B2, B3, B4, B5)
    ```

    where each argument of the term is unified with the state of the corresponding pointer button just before the event, and has the value `up` or `down`.

    Unifies *Modifiers* with a term of the form:

    ```
    modifiers(Shift, Control, Lock,
              Mod1, Mod2, Mod3, Mod4, Mod5)
    ```

    where each argument of the term is unified with the state of the corresponding modifier key just before the event, and has the value `up` or `down`.
16. `button(B)` unifies B with an integer between `1` and `5` corresponding to the button that changed state.
17. `same_screen(B)` unifies B, a boolean value, with one of

    `true` If the mouse pointer is currently on the same screen as the window receiving the event.

    `false` If the mouse pointer was actively grabbed by a client, before the automatic grab could take place.

16.4.3.2 `circulate_notify` Event

The fields that can be unified in `circulate_notify` events are:

1. `type(T)` unifies T with `circulate_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of:

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window receiving the event.
6. `window(W)` unifies W with the ProXL Window that was restacked. This can be the same as the window receiving the event, or one of its children.
7. `place(P)` unifies P with one of

   `on_top` If the window was raised to the top of the stack

   `on_bottom`
   If the window was placed at the bottom of the stack

16.4.3.3 `circulate_request` Event

The fields that can be unified in `circulate_request` events are:

1. `type(T)` unifies T with `circulate_request`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `parent(P)` unifies P with the parent ProXL Window of the window being restacked. This is the window that selected the event.
6. `window(W)` unifies W with the ProXL Window that is being restacked.
7. `place(P)` unifies P with one of

   `on_top` If the window was raised to the top of the stack

   `on_bottom`
   If the window was placed at the bottom of the stack

16.4.3.4 `client_message` Event

The fields that can be unified in `client_message` events are:

1. `type(T)` unifies T with `client_message`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.

3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window receiving the event.
6. `message_type(M)` unifies M with a ProXL Atom that specifies to the receiving client how to interpret the data.
7. `format(F)` unifies F with one of the integers `8`, `16` or `32` to specify the format of the data.
8. `data(D)` unifies D with a list of the data sent in the event. This will be one of:

   A list of 20 C `char` elements
   If the format is `8`.

   A list of 10 C `short` elements
   If the format is `16`.

   A list of 20 C `int` elements
   If the format is `32`.

16.4.3.5 `colormap_notify` Event

The fields that can be unified in `colormap_notify` events are:

1. `type(T)` unifies T with `colormap_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window whose associated colormap or attribute changes.
6. `colormap(C)` unifies C with the ProXL Colormap being installed or the constant `none` if the Colormap was destroyed.
7. `new(N)` unifies N, a boolean value, with one of

   `true` If the `colormap` attribute has been changed.

   `false` If the colormap is installed or uninstalled.
8. `state(S)` unifies S with one of

   `installed`
   If the colormap is installed.

   `uninstalled`
   If the colormap is uninstalled.

16.4.3.6 `configure_notify` Event

The fields that can be unified in `configure_notify` events are:

1. `type(T)` unifies T with `configure_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window that selected the event.
6. `window(W)` unifies W with the ProXL Window whose configuration has changed.
7. `x(X)` unifies X with the x coordinate, relative to the window's parent origin.
8. `y(Y)` unifies Y with the y coordinate, relative to the window's parent origin.
9. `position(X, Y)` unifies X and Y with the x and y coordinates respectively, relative to the window's parent origin.
10. `width(W)` unifies W with the width in pixels of the window.
11. `height(H)` unifies H with the height in pixels of the window.
12. `size(W, H)` unifies W and H with the width and height in pixels, respectively, of the window.
13. `border_width(W)` unifies W with the width of the window's border in pixels.
14. `above(A)` unifies A with the sibling ProXL Window that the window is immediately on top of, or the constant `none` if the window is on the bottom of the stack with respect to its siblings.
15. `override_redirect(B)` unifies B with the boolean value of the window's `override_redirect` attribute, one of:

    `true` If the client wants the window to be exempt from interception of the request by the Window Manager.

    `false` If the Window Manager is allowed to modify the request.

16.4.3.7 `configure_request` Event

The fields that can be unified in `configure_request` events are:

1. `type(T)` unifies T with `configure_request`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.

4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `parent(P)` unifies P with the ProXL Window that is the parent of the window being reconfigured. This is the window that selected the event.
6. `window(W)` unifies W with the ProXL Window being reconfigured.
7. `x(X)` unifies X with one of :

   An integer The requested x coordinate value, relative to the window's parent origin.

   `none` If no value was specified by the requestor.
8. `y(Y)` unifies Y with one of:

   An integer The requested y coordinate value, relative to the window's parent origin.

   `none` If no value was specified by the requestor.
9. `position(X, Y)` unifies X and Y with:

   An integer The requested coordinate value, respectively, relative to the window's parent origin.

   `none` If no value was specified by the requestor.
10. `width(W)` unifies W with one of:

    An integer The requested width value of the window, in pixels.

    `none` If no value was specified by the requestor.
11. `height(H)` unifies H with one of:

    An integer The requested height value of the window, in pixels.

    `none` If no value was specified by the requestor.
12. `size(W, H)` unifies W and H with:

    An integer The requested width and height values, respectively, in pixels.

    `none` If no value was specified by the requestor.
13. `border_width(W)` unifies W with one of:

    An integer The requested width value of the window's border, in pixels.

    `none` If no value was specified by the requestor.
14. `above(A)` unifies A with one of:

    A Window
    The sibling ProXL Window to be used for the requested re-stacking operation.

    `none` If the requestor, either specified `none` as the sibling window, or did not specify it.
15. `detail(D)` unifies D with one of:

    `above` If the window requests to be placed above the given sibling, or at the top of the stack (in this case, sibling must be `none`).

    `below` If the window requests to be placed just below the given sibling, or at the bottom of the stack (in this case, sibling must be `none`).

`top_if` If the window requests to be placed at the top of the stack if the given sibling obscures it. A value of `none` for sibling means any sibling.

`bottom_if`
If the window requests to be placed at the bottom of the stack if it obscures the given sibling. A value of `none` for sibling means any sibling.

`opposite` If the window requests to be placed at the top of the stack if the given sibling obscures it, or at the bottom of the stack if it obscures the given sibling. A value of `none` for sibling means any sibling.

`none` If no value was specified by the requestor.

16. `value_mask(V)` unifies V with a list of the fields in original request that were specified. The values that it can contain are `x, y, width, height, border_width, sibling, stack_mode`.

16.4.3.8 `create_notify` Event

The fields that can be unified in `create_notify` events are:

1. `type(T)` unifies T with `create_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `parent(P)` unifies P with the parent ProXL Window of the created window.
6. `window(W)` unifies W with the ProXL Window just created.
7. `x(X)` unifies X with the x coordinate of the created window, relative to its parent.
8. `y(Y)` unifies Y with the y coordinate of the created window, relative to its parent.
9. `position(X, Y)` unifies X and Y with the x and y coordinates, respectively, of the created window, relative to its parent.
10. `width(W)` unifies W with the width in pixels of the window.
11. `height(H)` unifies H with the height in pixels of the window.
12. `size(W, H)` unifies W and H with the width and height in pixels, respectively, of the window.
13. `border_width(W)` unifies W with the width of the window's border in pixels.
14. `override_redirect(B)` unifies B with the boolean value of the `override_redirect` attribute of the window, one of:

    `true` If the client wants the window to be exempt from interception of the request by the Window Manager.

    `false` If the Window Manager is allowed to modify the request.

16.4.3.9 `destroy_notify` Event

The fields that can be unified in `destroy_notify` events are:

1. `type(T)` unifies T with `destroy_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window that selected the event.
6. `window(W)` unifies W with the Window that was destroyed.

16.4.3.10 `enter_notify` and `leave_notify` Events

The fields that can be unified in `enter_notify` and `leave_notify` events are:

1. `type(T)` unifies T with `enter_notify` or `leave_notify` according to the event.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that receives the event.
6. `root(R)` unifies R with the root Window that the event occurred under.
7. `subwindow(S)` unifies S with the child ProXL window where the event occurred or `none` if the event occurred in the window itself.
8. `time(T)` unifies T with the server time, in milliseconds, when the event occurred.
9. `x(X)` unifies X with the x coordinate, relative to the window origin.
10. `y(Y)` unifies Y with the y coordinate, relative to the window origin.
11. `position(X, Y)` unifies X and Y with the x and y coordinates respectively, relative to the window origin.
12. `x_root(X)` unifies X with the x coordinate, relative to the root window origin.
13. `y_root(Y)` unifies Y with the y coordinate, relative to the root window origin.
14. `root_position(X, Y)` unifies X and Y with the x and y coordinates, respectively, relative to the root window origin.
15. `mode(M)` unifies M with one of

    `normal` If the event was caused by a normal mouse pointer movement or a pointer warp.

`grab` If the event was caused by a grab.

`ungrab` If the event was caused by an ungrab.

16. `detail(`*D*`)` unifies *D* with one of

`ancestor` If the movement came from/ended up in a ProXL Window who is a direct ancestor of the receiving window.

`virtual` If the movement just *passed through* the receiving window because of its position in the hierarchy on its way to a window in the same hierarchy.

`inferior` If the movement came from/ended up in a ProXL Window who is an inferior of the receiving window.

`nonlinear`
If the movement came from/ended up in a sibling or *cousin* window.

`nonlinear_virtual`
If the movement *passed through* this window on its way to a sibling or *cousin* window.

17. `same_screen(`*B*`)` unifies *B*, a boolean value, with one of

`true` If the mouse pointer is currently on the same screen as the window receiving the event.

`false` If the mouse pointer was actively grabbed by a client, before the automatic grab could take place.

18. `focus(`*F*`)` unifies *F*, a boolean, with one of

`true` If the receiving window is the focus window, or an inferior of the focus window.

`false` If the focus is assigned to another window hierarchy.

19. `state(`*Buttons*`, `*Modifiers*`)` unifies *Buttons* with a term of the form:

```
buttons(B1, B2, B3, B4, B5)
```

where each argument of the term is unified with the state of the corresponding pointer button just before the event, and has the value `up` or `down`.

Unifies *Modifiers* with a term of the form:

```
modifiers(Shift, Control, Lock,
          Mod1, Mod2, Mod3, Mod4, Mod5)
```

where each argument of the term is unified with the state of the corresponding modifier key just before the event, and has the value `up` or `down`.

16.4.3.11 `expose` Event

The fields that can be unified in `expose` events are:

1. `type(T)` unifies T with `expose`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window receiving the event.
6. `x(X)` unifies X with the x coordinate of the upper-left corner of the exposed region, relative to the window origin.
7. `y(Y)` unifies Y with the y coordinate of the upper-left corner of the exposed region, relative to the window origin.
8. `position(X, Y)` unifies X and Y with the x and y coordinates, respectively, of the upper-left corner of the exposed region, relative to the window origin.
9. `width(W)` unifies W with the width in pixels of the exposed region.
10. `height(H)` unifies H with the height in pixels of the exposed region.
11. `size(W, H)` unifies W and H with the width and height in pixels, respectively, of the exposed region.
12. `count(C)` unifies C with an integer giving the approximate number of remaining contiguous `expose` events that were generated as a result of a single function call. If it is zero, no more `expose` events for this window follow.

16.4.3.12 `focus_in` and `focus_out` Events

The fields that can be unified in `focus_in` and `focus_out` events are:

1. `type(T)` unifies T with `focus_in` or `focus_out` according to the event.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window receiving the event.
6. `mode(M)` unifies M with one of

   `normal` If the keyboard is not grabbed.

   `grab` If the event is triggered by the start of a keyboard grab.

   `ungrab` If the event is triggered by the keyboard being ungrabbed.

   `while_grabbed`
   If the focus changes while the keyboard is grabbed.

7. `detail(D)` unifies D with an atom identifying the relationship between the window that receives the event, the window that lost the focus, the window that got the focus, and the window that contained the pointer at the time of the focus change . One of:
   - `ancestor`
   - `virtual`
   - `inferior`
   - `nonlinear`
   - `nonlinear_virtual`
   - `pointer`
   - `pointer_root`
   - `detail_none`

16.4.3.13 `graphics_expose` Event

The fields that can be unified in `graphics_expose` events are:

1. `type(T)` unifies T with `graphics_expose`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `drawable(D)` unifies D with either the ProXL Window or ProXL Pixmap that was the destination of the graphics request.
6. `x(X)` unifies X with the x coordinate of the upper-left corner of the region, relative to the drawable origin.
7. `y(Y)` unifies Y with the y coordinate of the upper-left corner of the region, relative to the drawable origin.
8. `position(X, Y)` unifies X and Y with the x and y coordinates, respectively, of the upper-left corner of the region, relative to the drawable origin.
9. `width(W)` unifies W with the width in pixels of the region.
10. `height(H)` unifies H with the height in pixels of the region.
11. `size(W, H)` unifies W and H with the width and height in pixels, respectively, of the exposed region.
12. `count(C)` unifies C with an integer giving the approximate number of remaining contiguous `graphics_expose` events that were generated as a result of a single request.
13. `major_code(M)` unifies M with the name of the graphics request that produced the event . One of
    - `copy_area`
    - `copy_plane`
14. `minor_code(M)` unifies M with an integer giving the request minor code.

16.4.3.14 `no_expose` Event

The fields that can be unified in `no_expose` events are:

1. `type(T)` unifies T with `no_expose`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `drawable(D)` unifies D with either the ProXL Window or ProXL Pixmap that was the destination of the graphics request.
6. `major_code(M)` unifies M with the name of the graphics request that produced the event . One of
   - `copy_area`
   - `copy_plane`
7. `minor_code(M)` unifies M with an integer giving the request minor code.

16.4.3.15 `gravity_notify` Event

The fields that can be unified in `gravity_notify` events are:

1. `type(T)` unifies T with `gravity_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window that receives the event.
6. `window(W)` unifies W with the ProXL Window that was moved because of its `win_gravity` attribute.
7. `x(X)` unifies X with the new x coordinate of the window, relative to the its parent.
8. `y(Y)` unifies Y with the new y coordinate of the window, relative to its parent.
9. `position(X, Y)` unifies X and Y with the new x and y coordinates of the window, respectively, relative to its parent.

16.4.3.16 `keymap_notify` Event

The fields that can be unified in `keymap_notify` events are:

1. `type(T)` unifies T with `keymap_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that was reported in the immediately preceding `enter_notify` or `focus_in` event.
6. `key_vector(K)` unifies K with a list of 32 8-bit integers, for a total of 256 bits, where each bit represents the state of the corresponding keycode. Use `key_state/2` to interpret the results.

16.4.3.17 `key_press` and `key_release` Events

The fields that can be unified in `key_press` and `key_release` events are:

1. `type(T)` unifies T with `key_press` or `key_release` according to the event.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that receives the event.
6. `root(R)` unifies R with the root Window that the event occurred under.
7. `subwindow(S)` unifies S with the child ProXL window where the event occurred or `none` if the event occurred in the window itself.
8. `time(T)` unifies T with the server time, in milliseconds, when the event occurred.
9. `x(X)` unifies X with the x pointer coordinate, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise it is 0.
10. `y(Y)` unifies Y with the y pointer coordinate, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise it is 0.
11. `position(X, Y)` unifies X and Y with the x and y pointer coordinates, respectively, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise they are 0.
12. `x_root(X)` unifies X with the x pointer coordinate, relative to the root window origin, if the receiving window is on the same screen as the root window, otherwise with 0.
13. `y_root(Y)` unifies Y with the y pointer coordinate, relative to the root window origin, if the receiving window is on the same screen as the root window, otherwise with 0.
14. `root_position(X, Y)` unifies X and Y with the pointer x and y coordinates, respectively, relative to the root window origin, if the receiving window is on the same screen as the root window. Otherwise they are both 0.

15. `state(`*`Buttons`*`, `*`Modifiers`*`)` unifies *Buttons* with a term of the form:

    ```
    buttons(B1, B2, B3, B4, B5)
    ```

    where each argument of the term is unified with the state of the corresponding pointer button just before the event, and has the value `up` or `down`.

    Unifies *Modifiers* with a term of the form:

    ```
    modifiers(Shift, Control, Lock,
              Mod1, Mod2, Mod3, Mod4, Mod5)
    ```

    where each argument of the term is unified with the state of the corresponding modifier key just before the event, and has the value `up` or `down`.
16. `keycode(`*`K`*`)` unifies K with the server-dependent integer keycode associated with the physical key.
17. `keysym(`*`K`*`)` unifies K with the server-independent keysym associated with the physical key.
18. `chars(`*`C`*`)` unifies C with the list of ASCII characters associated with the physical key, if there is one, or the empty list, if there isn't one.
19. `length(`*`L`*`)` unifies L with the length of the ASCII string associated with the physical key.
20. `same_screen(`*`B`*`)` unifies B, a boolean value, with one of

    `true` If the mouse pointer is currently on the same screen as the window receiving the event.

    `false` If the mouse pointer and the window are on different screens.

16.4.3.18 `map_notify` Event

The fields that can be unified in `map_notify` events are:

1. `type(`*`T`*`)` unifies T with `map_notify`.
2. `serial(`*`S`*`)` unifies S with the serial number of the last request processed by the server.
3. `send_event(`*`B`*`)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(`*`D`*`)` unifies D with the ProXL Display the event was read from.
5. `event(`*`E`*`)` unifies E with the ProXL Window that selected the event.
6. `window(`*`W`*`)` unifies W with the ProXL Window that is being mapped.
7. `override_redirect(`*`B`*`)` unifies B with the boolean value of the `override_redirect` attribute of the window, one of

`true` If the client wants the window to be exempt from interception of the request by the Window Manager.

`false` If the Window Manager is allowed to modify the request.

16.4.3.19 `unmap_notify` Event

The fields that can be unified in `unmap_notify` events are:

1. `type(T)` unifies T with `unmap_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window that selected the event.
6. `window(W)` unifies W with the ProXL Window that is being mapped.
7. `from_configure(F)` unifies F, a boolean value, with one of

   `true` If the event was generated as a result of a resizing of the window's parent and the window itself has a `win_gravity` attribute of `unmap`.

   `false` Otherwise.

16.4.3.20 `mapping_notify` Event

The fields that can be unified in `mapping_notify` events are:

1. `type(T)` unifies T with `mapping_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window receiving the event.
6. `request(R)` unifies R with one of

   `modifier` If the modifier keys have been remapped.

   `keyboard` If the keyboard has been remapped.

   `pointer` If the pointer buttons have been remapped.
7. `first_keycode(F)` unifies F with the first in a range of keycodes with new mappings, if the value of *request* is `modifier` or `keyboard`. For `pointer`, its value is 0.

8. `count(C)` unifies C with an integer indicating the number of keycodes with altered mappings, if the *request* is `modifier` or `keyboard`. For `pointer`, its value is 0.

16.4.3.21 `map_request` Event

The fields that can be unified in `map_request` events are:

1. `type(T)` unifies T with `map_request`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `parent(P)` unifies P with the ProXL parent Window of the window being mapped.
6. `window(W)` unifies W with the ProXL Window requesting to be mapped.

16.4.3.22 `motion_notify` Event

The fields that can be unified in `motion_notify` events are:

1. `type(T)` unifies T with `motion_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that receives the event.
6. `root(R)` unifies R with the root Window that the event occurred under.
7. `subwindow(S)` unifies S with the child ProXL window where the event occurred or `none` if the event occurred in the window itself.
8. `time(T)` unifies T with the server time, in milliseconds, when the event occurred.
9. `x(X)` unifies X with the x pointer coordinate, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise it is 0.
10. `y(Y)` unifies Y with the y pointer coordinate, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise it is 0.
11. `position(X, Y)` unifies X and Y with the x and y pointer coordinates, respectively, relative to the window origin, if the receiving window is on the same screen as the root window. Otherwise they are 0.
12. `x_root(X)` unifies X with the x pointer coordinate, relative to the root window origin, if the receiving window is on the same screen as the root window, otherwise with 0.

13. `y_root(Y)` unifies Y with the y pointer coordinate, relative to the root window origin, if the receiving window is on the same screen as the root window, otherwise with 0.
14. `root_position(X, Y)` unifies X and Y with the pointer x and y coordinates, respectively, relative to the root window origin, if the receiving window is on the same screen as the root window. Otherwise they are both 0.
15. `state(Buttons, Modifiers)` unifies *Buttons* with a term of the form:

    ```
    buttons(B1, B2, B3, B4, B5)
    ```

    where each argument of the term is unified with the state of the corresponding pointer button just before the event, and has the value `up` or `down`.

    Unifies *Modifiers* with a term of the form:

    ```
    modifiers(Shift, Control, Lock,
              Mod1, Mod2, Mod3, Mod4, Mod5)
    ```

    where each argument of the term is unified with the state of the corresponding modifier key just before the event, and has the value `up` or `down`.
16. `is_hint(H)` unifies H with one of

    `normal` If the event is a normal `motion_notify` event.

    `hint` If the event is a hint that the mouse pointer moved. This will always be the case if the event selected is `motion_notify_hint`.
17. `same_screen(B)` unifies B, a boolean value, with one of

    `true` If the mouse pointer is currently on the same screen as the window receiving the event.

    `false` If the the mouse pointer and the window are on different screens.

16.4.3.23 `property_notify` Event

The fields that can be unified in `property_notify` events are:

1. `type(T)` unifies T with `property_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window who owns the property that changed.
6. `atom(A)` unifies A with the name of the property that changed.
7. `time(T)` unifies T with the server time, in milliseconds, when the event occurred.

8. `state(S)` unifies S with one of

   `new_value`
   : If the property has a new value.

   `delete`
   : If the property has been deleted.

16.4.3.24 `reparent_notify` Event

The fields that can be unified in `reparent_notify` events are:

1. `type(T)` unifies T with `reparent_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true`
   : If the event was sent by another client.

   `false`
   : If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `event(E)` unifies E with the ProXL Window that receives the event.
6. `window(W)` unifies W with the ProXL Window that has been reparented.
7. `parent(P)` unifies P with the ProXL Window that is the new parent.
8. `x(X)` unifies X with the x coordinate of the upper-left corner of the window, relative to its new parent's origin.
9. `y(Y)` unifies Y with the y coordinate of the upper-left corner of the window, relative to its new parent's origin.
10. `position(X, Y)` unifies X and Y with the x and y coordinates, respectively, of the upper-left corner of the window, relative to its new parent's origin.
11. `override_redirect(B)` unifies B with the boolean value of the `override_redirect` attribute of the window, one of

    `true`
    : If the client wants the window to be exempt from interception of the request by the Window Manager.

    `false`
    : If the Window Manager is allowed to modify the request.

16.4.3.25 `resize_request` Event

The fields that can be unified in `resize_request` events are:

1. `type(T)` unifies T with `resize_request`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true`
   : If the event was sent by another client.

   `false`
   : If the event was sent by the server.

4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window whose size is being changed.
6. `width(W)` unifies W with the requested width in pixels of the window.
7. `height(H)` unifies H with the requested height in pixels of the window.
8. `size(W, H)` unifies W and H with the requested width and height in pixels, respectively, of the window.

16.4.3.26 `selection_clear` Event

The fields that can be unified in `selection_clear` events are:

1. `type(T)` unifies T with `selection_clear`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that receives the event, and is losing the selection.
6. `selection(S)` unifies S with the atom giving the name of the selection.
7. `time(T)` unifies T with the last-changed time, in milliseconds, recorded for the selection.

16.4.3.27 `selection_notify` Event

The fields that can be unified in `selection_notify` events are:

1. `type(T)` unifies T with `selection_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `requester(R)` unifies R with the ProXL Window that requested the selection.
6. `selection(S)` unifies S with the atom name of the requested selection.
7. `target(T)` unifies T with the atom name for the data type that the selection was converted to.
8. `property(P)` unifies P with the atom name of the Window property where the converted selection was stored, or `none` if the conversion could not be performed.
9. `time(T)` unifies T with the server time, in milliseconds, when the selection was stored.

16.4.3.28 `selection_request` Event

The fields that can be unified in `selection_request` events are:

1. `type(T)` unifies T with `selection_request`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `owner(O)` unifies O with the ProXL Window that owns the selection.
6. `requester(R)` unifies R with the ProXL Window that requests the selection.
7. `selection(S)` unifies S with the atom name of the selection requested
8. `target(T)` unifies T with the atom name for the data type the selection is requested in.
9. `property(P)` unifies P with the atom name of the property on which the translated data should be stored.
10. `time(T)` unifies T with the server time, in milliseconds, when the selection was requested.

16.4.3.29 `visibility_notify` Event

The fields that can be unified in `visibility_notify` events are:

1. `type(T)` unifies T with `visibility_notify`.
2. `serial(S)` unifies S with the serial number of the last request processed by the server.
3. `send_event(B)` unifies B, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(D)` unifies D with the ProXL Display the event was read from.
5. `window(W)` unifies W with the ProXL Window that receives the event.
6. `state(S)` unifies S with one of:

   `unobscured`
   If the window is viewable and completely unobscured.

   `partially_obscured`
   If the window is viewable and partially obscured.

   `fully_obscured`
   If the window is fully obscured by another window.

16.4.3.30 `default` Event

ProXL allows the user to register a default callback that will be executed if there is no registered callback for the event received, or if all the registered callbacks failed. This is mostly a convenience feature for debugging purposes, as only a few of the event fields can be used.

The fields that can be unified in `default` events are:

1. `type(`*T*`)` unifies *T* with the actual event name.
2. `serial(`*S*`)` unifies *S* with the serial number of the last request processed by the server.
3. `send_event(`*B*`)` unifies *B*, a boolean value, with one of

   `true` If the event was sent by another client.

   `false` If the event was sent by the server.
4. `display(`*D*`)` unifies *D* with the ProXL Display the event was read from.

16.4.4 Activating the callback mechanism

ProXL provides three mechanisms for handling events and activating callbacks. First of all, events are handled whenever Prolog is waiting for input. This includes while Prolog is reading input from the terminal, even under the Quintus User Interface. The also includes while debugging. Events are not handled while Prolog computations are under way (Prolog is not interruped to handle events).

In situations where the program needs to wait while events are being handled, the predicates `handle_events/N` are appropriate. For those cases where more flexibility is required, there are many predicates described in Section 16.12 [pxl-evf], page 856 that will return events. These can be used in conjunction with `dispatch_event/N`, described below, to handle callbacks. Alternatively, callbacks need not be used: you can handle the event any way you like.

16.4.4.1 `handle_events/[0,1,2,3]`

To wait while callbacks are being handled, use one of:

```
handle_events
handle_events(-ExitCond)
handle_events(-ExitCond, +Context)
```

`handle_events` will enter a failure-driven loop listening for events on all open ProXL displays, and executing callbacks. It exits when either any of the callbacks binds its *exit variable* to a value that unifies with *ExitCond*, or when all ProXL Windows that have associated callbacks have been destroyed (in which case ExitCond will be unified with `none`).

If the *Displayable* is omitted, the default Display is used.

If *ExitCond* is omitted, it defaults to `none`.

Context is passed to any callbacks that want it, and may be used to cause callbacks to behave differently in different contexts. If omitted, it defaults to an anonymous variable.

For backward compatibility,

```
handle_events(+Displayable, -ExitCond, +Context)
```

is equivalent to `handle_events(ExitCond, Context)`. The Displayable is ignored.

16.4.4.2 `dispatch_event/[1,2,3]`

To activate the callbacks registered for a single X event, use one of:

```
dispatch_event(+XEvent)
dispatch_event(+XEvent, -ExitCond)
dispatch_event(+XEvent, -ExitCond, +Context)
```

`dispatch_event(+XEvent)` will trigger any ProXL callbacks associated with the supplied *XEvent* and succeed, even if there were no callbacks registered for the event.

`dispatch_event(+XEvent, -ExitCond)` triggers the ProXL callbacks associated with the supplied *XEvent* and succeeds either if all the ProXL Windows associated with the event's Display were destroyed, or if any of the callbacks binds its *exit variable* to a value that unifies with *ExitCond*. If the callback uses contexts, `dispatch_event/2` will pass an anonymous variable.

`dispatch_event(+XEvent, -ExitCond, +Context)` is like `dispatch_event/2`, but gives the *Context* argument to the callback, for further discrimination.

16.4.4.3 Exit Variables

The ProXL callback mechanism allows the user to specify an *exit variable* associated with each callback routine. After the successful execution of a callback routine, ProXL examines the exit variable, if it exists.

If the callback succeeded and the exit variable was bound by the callback to a value that unifies with the *ExitCond* given, `handle_events/[1,2,3]`, and `dispatch_event/[2,3]` succeed, with *ExitCond* bound to the value given by the callback routine.

16.5 Drawing Primitives

This section documents ProXL's drawing primitives.

In most of these procedures, a graphics context may optionally be supplied. If one is not, the current graphics attributes of the destination drawable determine the behavior of the drawing.

16.5.1 Clearing and Copying Areas

16.5.1.1 clear_area/[5,6]

```
clear_area(+Window, +X, +Y,
                +Width, +Height)
clear_area(+Window, +X, +Y,
                +Width, +Height, +Exposures)
```

Clear the region of *Window* defined by *X*, *Y*, *Width*, and *Height*. *Exposures* is either `true` or `false` (default `false`) indicating whether exposure events should be generated. The cleared portion of the window is filled with the window's background.

16.5.1.2 clear_window/1

```
clear_window(+Window)
```

Clear *Window*. The window is filled with its background.

16.5.1.3 copy_area/[8,9]

```
copy_area(+Dst, +Dst_x, +Dst_y, +Src, +Src_x, +Src_y,
          +Width, +Height)
copy_area(+Dst, +Src, +Gc, +Dst_x, +Dst_y, +Src_x, +Src_y,
          +Width, +Height)
```

Copy a region of *Src* into *Dst*. *Src_x*, *Src_y*, *Width*, and *Height* define the part of *Src* that is to be copied, and *Dst_x* and *Dst_y* specify where in *Dst* the are is to be copied.

16.5.1.4 copy_plane/[9,10]

```
copy_plane(+Dst, +Dst_x, +Dst_y, +Src, +Src_x, +Src_y,
           +Width, +Height, +Plane)
copy_plane(+Dst, +Src, +Gc, +Dst_x, +Dst_y, +Src_x, +Src_y,
           +Width, +Height, +Plane)
```

Copy some bit planes from a region of *Src* into *Dst*. *Src_x*, *Src_y*, *Width*, and *Height* define the part of *Src* that is to be copied, and *Dst_x* and *Dst_y* specify where in *Dst* the are is to be copied. *Plane* is a bit mask (i.e. an integer) specifying which planes to copy.

16.5.2 Drawing Points

16.5.2.1 draw_point/[3,4]

```
draw_point(+Drawable, +X, +Y)
draw_point(+Drawable, +Gc, +X, +Y)
```

Draw a single pixel point at *X*, *Y* in *Drawable*.

16.5.2.2 draw_points/[2,3]

```
draw_points(+Drawable, +Points)
draw_points(+Drawable, +Gc, +Points)
```

Draw single pixel points in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying the location of one point.

16.5.2.3 draw_points_relative/[2,3]

```
draw_points_relative(+Drawable, +Points)
draw_points_relative(+Drawable, +Gc, +Points)
```

Draw single pixel points in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying an x and y offset from the previous point. The first point is absolute.

16.5.3 Drawing Lines

16.5.3.1 draw_line/[5,6]

```
draw_line(+Drawable, +X1, +Y1, +X2, +Y2)
draw_line(+Drawable, +Gc, +X1, +Y1, +X2, +Y2)
```

Draw a line in *Drawable* from (*X1*,*Y1*) to (*X2*,*Y2*).

16.5.3.2 draw_lines/[2,3]

```
draw_lines(+Drawable, +Points)
draw_lines(+Drawable, +Gc, +Points)
```

Draw connected lines in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying the location of one vertex.

16.5.3.3 draw_lines_relative/[2,3]

```
draw_lines_relative(+Drawable, +Points)
draw_lines_relative(+Drawable, +Gc, +Points)
```

Draw connected lines in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying an x and y offset from the previous point. The first point is absolute.

16.5.3.4 draw_segments/[2,3]

```
draw_segments(+Drawable, +Segments)
draw_segments(+Drawable, +Gc, +Segments)
```

Draw disconnected line segments in *Drawable*. *Segments* is a list of `segment(X1,Y1,X2,Y2)` terms. Lines are drawn from each *X1*, *Y1* to each corresponding *X2*, *Y2*.

16.5.4 Drawing and Filling Polygons

16.5.4.1 draw_polygon/[2,3]

```
draw_polygon(+Drawable, +Points)
draw_polygon(+Drawable, +Gc, +Points)
```

Draw a polygon in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying the location of one vertex of the polygon. This procedure does not correspond directly to any Xlib function.

16.5.4.2 draw_polygon_relative/[2,3]

```
draw_polygon_relative(+Drawable, +Points)
draw_polygon_relative(+Drawable, +Gc, +Points)
```

Draw a polygon in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying the location of one vertex of the polygon relative to the previous point. This procedure does not correspond directly to any Xlib function.

16.5.4.3 fill_polygon/[3,4]

```
fill_polygon(+Drawable, +Points, +Shape)
fill_polygon(+Drawable, +Gc, +Points, +Shape)
```

Draw a filled polygon in *Drawable*. *Points* is a list of `point(X,Y)` terms, each specifying the location of one vertex of the polygon. *Shape* is either `complex`, meaning the polygon may contain intersecting edges, `nonconvex`, meaning no edges intersect, or `convex`, meaning that no edges intersect, and further, the polygon is wholly convex.

16.5.4.4 fill_polygon_relative/[3,4]

```
fill_polygon_relative(+Drawable, +Points, +Shape)
fill_polygon_relative(+Drawable, +Gc, +Points, +Shape)
```

Draw a filled polygon. *Points* is a list of `point(X,Y)` terms, each specifying the location of one vertex of the polygon relative to the previous point. *Shape* is either `complex`, meaning the polygon may contain intersecting edges, `nonconvex`, meaning no edges intersect, or `convex`, meaning that no edges intersect, and further, the polygon is wholly convex.

16.5.5 Drawing and Filling Rectangles

16.5.5.1 draw_rectangle/[5,6]

```
draw_rectangle(+Drawable, +X, +Y, +Width, +Height)
draw_rectangle(+Drawable, +Gc, +X, +Y, +Width, +Height)
```

Draw a rectangle in *Drawable*. The upper left corner of the rectangle is at *X*, *Y*, and its size is *Width* x *Height*.

16.5.5.2 draw_rectangles/[2,3]

```
draw_rectangles(+Drawable, +Rectangles)
draw_rectangles(+Drawable, +Gc, +Rectangles)
```

Draw rectangles in *Drawable*. *Rectangles* is a list of `rectangle(X,Y,Width,Height)` terms specifying the position and size of each rectangle.

16.5.5.3 fill_rectangle/[5,6]

```
fill_rectangle(+Drawable, +X, +Y, +Width, +Height)
fill_rectangle(+Drawable, +Gc, +X, +Y, +Width, +Height)
```

Draw a filled rectangle in *Drawable*. The upper left corner of the rectangle is at *X*, *Y*, and its size is *Width* x *Height*.

16.5.5.4 fill_rectangles/[2,3]

```
fill_rectangles(+Drawable, +Rectangles)
fill_rectangles(+Drawable, +Gc, +Rectangles)
```

Draw filled rectangles in *Drawable*. *Rectangles* is a list of `rectangle(X,Y,Width,Height)` terms specifying the position and size of each rectangle.

16.5.6 Drawing and Filling Arcs

16.5.6.1 draw_arc/[7,8]

```
draw_arc(+Drawable, +X, +Y, +Width, +Height, +Theta1, +Theta2)
draw_arc(+Drawable, +Gc, +X, +Y, +Width, +Height, +Theta1, +Theta2)
```

Draw an arc in *Drawable*. The arc is specified in terms of the rectangle that would enclose the ellipse of which this arec is a part, and the angles, in degrees from the center right of the ellipse, to draw between. *X*, *Y*, *Width*, and *Height* specify the bounding rectangle, *Theta1* specifies the starting angle, and *Theta2* specifies the ending angle. Angles may be integers or floats.

16.5.6.2 draw_arcs/[2,3]

```
draw_arcs(+Drawable, +Arcs)
draw_arcs(+Drawable, +Gc, +Arcs)
```

Draw arcs in *Drawable*. Each arc is specified in terms of the rectangle that would enclose the ellipse of which this arec is a part, and the angles, in degrees from the center right of the ellipse, to draw between. *Arcs* is a list of `arc(X,Y,Width,Height,Theta1,Theta2)` terms, where *X*, *Y*, *Width*, and *Height* specify the bounding rectangle, Theta1 specifies the starting angle, and Theta2 specifies the ending angle. Angles may be integers or floats.

16.5.6.3 fill_arc/[7,8]

```
fill_arc(+Drawable, +X, +Y, +Width, +Height, +Theta1, +Theta2)
fill_arc(+Drawable, +Gc, +X, +Y, +Width, +Height, +Theta1, +Theta2)
```

Draw a filled arc in *Drawable*. The arc is specified in terms of the rectangle that would enclose the ellipse of which this arec is a part, and the angles, in degrees from the center right of the ellipse, to draw between. *X*, *Y*, *Width*, and *Height* specify the bounding rectangle, *Theta1* specifies the starting angle, and *Theta2* specifies the ending angle. Angles may be integers or floats.

16.5.6.4 `fill_arcs/[2,3]`

```
fill_arcs(+Drawable, +Arcs)
fill_arcs(+Drawable, +Gc, +Arcs)
```

Draw filled arcs in *Drawable*. Each arc is specified in terms of the rectangle that would enclose the ellipse of which this arec is a part, and the angles, in degrees from the center right of the ellipse, to draw between. *Arcs* is a list of `arc(X,Y,Width,Height,Theta1,Theta2)` terms, where *X*, *Y*, *Width*, and *Height* specify the bounding rectangle, Theta1 specifies the starting angle, and Theta2 specifies the ending angle. Angles may be integers or floats.

16.5.7 Drawing and Filling Ellipses and Circles

An ellipses is specified by its bounding rectangle, much the same as the primitives for drawing and filling rectangles in Section 16.5.5 [pxl-prim-rect], page 823 above. To draw or fill a circle, just specify a square bounding rectangle.

16.5.7.1 `draw_ellipse/[5,6]`

```
draw_ellipse(+Drawable, +X, +Y, +Width, +Height)
draw_ellipse(+Drawable, +Gc, +X, +Y, +Width, +Height)
```

Draw an ellipse in *Drawable*. *X*, *Y*, *Width*, and *Height* specify the rectangle that would enclose the ellipse.

16.5.7.2 `draw_ellipses/[2,3]`

```
draw_ellipses(+Drawable, +Rectangles)
draw_ellipses(+Drawable, +Gc, +Rectangles)
```

Draw ellipses in *Drawable*. *Rectangles* is a list of `rectangle(X,Y,Width,Height)` terms, each specifying the rectangle that would enclose an ellipse.

16.5.7.3 `fill_ellipse/[5,6]`

```
fill_ellipse(+Drawable, +X, +Y, +Width, +Height)
fill_ellipse(+Drawable, +Gc, +X, +Y, +Width, +Height)
```

Draw a filled ellipse in *Drawable*. *X*, *Y*, *Width*, and *Height* specify the rectangle that would enclose the ellipse.

16.5.7.4 `fill_ellipses/[2,3]`

```
fill_ellipses(+Drawable, +Rectangles)
fill_ellipses(+Drawable, +Gc, +Rectangles)
```

Draw filled ellipses in *Drawable*. *Rectangles* is a list of `rectangle(X,Y,Width,Height)` terms, each specifying the rectangle that would enclose an ellipse.

16.5.8 Drawing Text

This section describes the commands for drawing text strings. The commands for determining how much space will be occupied by a given string are documented in Section 16.7.6 [pxl-font-siz], page 836. Fonts in general are documented in Section 16.7 [pxl-font], page 832.

In all of these primitives, the string to be drawn may be either a Prolog atom or a *chars*, that is a list of character codes. This later form is quite handy for code that builds up strings to be drawn by appending together strings, since it is not necessary to turn them into atoms in order to draw them.

16.5.8.1 `draw_string/[4,5]`

```
draw_string(+Drawable, +X, +Y, +String)
draw_string(+Drawable, +Gc, +X, +Y, +String)
```

Draw a text string in *Drawable*. *String* is drawn with the origin point of the first character at postion *X*, *Y*. *String* may be either a Prolog atom, or a list of character codes.

Note that this operation affects only pixels in *Drawable* where the characters *are*; the *background* part of the characters does not affect *Drawable*. If you want the entire area of *Drawable* occupied by string to be effected, use `draw_image_string`.

16.5.8.2 `draw_image_string/[4,5]`

```
draw_image_string(+Drawable, +X, +Y, +String)
draw_image_string(+Drawable, +Gc, +X, +Y, +String)
```

Draw a text string in *Drawable*. *String* is drawn with the origin point of the first character at postion *X*, *Y*. *String* may be either a Prolog atom, or a list of character codes. This drawing operation sets the background bits of each character to *Drawable*'s background color (or *Gc*'s background color, if *Gc* is specified).

16.5.8.3 `draw_text/[4,5]`

```
draw_text(+Drawable, +X, +Y, +Textitems)
draw_text(+Drawable, +Gc, +X, +Y, +Textitems)
```

Draw strings in *Drawable*, with mixed fonts and flexible inter-string spacing. *Textitems* is a list of `textitem(String,Delta)` and `textitem(String,Delta,Font)` terms, where *String* is the string (Prolog atom or list of character codes) to be printed, *Delta* is the number of extra pixels to skip over horizontally before drawing string, and, for `textitem/3` terms,

Font is the font in which to draw String and any following strings in *Textitems* until Font is changed again. Text printed after the call to `draw_text` will not be affected by the call.

Note that this will not print text on multiple lines. This is equivalent to doing several calls to `draw_string`, possibly with calls to `put_graphics_attributes/3` between, but more efficient. There is no similar primitive that is equivalent to multiple calls to `draw_image_string`.

16.6 Graphics Attributes and Graphics Contexts

The X window system requires one to set up many parameters to drawing commands before using the drawing primitives. For example, the command to draw a line in X does not have a way to specify the width of the line, or whether it will be dashed. These parameters must be specified ahead of time. Fortunately, there are defaults for all of these parameters, as discussed below, so you only need to worry about the parameters whose defaults don't suit you.

In ProXL, there are two ways to do this. You may specify the drawing parameters for the object you are drawing into, or you may specify them in a separate data structure, called a *graphics context*, or *gc*, and give the gc as an argument to the drawing commands. You may do whichever is more convenient at the time. You may mix these methods as you like.

16.6.1 Graphics Attributes

Much of the behavior of the drawing commands described in Section 16.5 [pxl-prim], page 820 is determined by the graphics attributes of the destination drawable (or of the gc, if one is supplied as argument). Following is a list of the graphics attributes and their meaning.

`function(V)`
: How source and destination are combined. Possible values are `clear`, `and`, `and_reverse`, `copy`, `and_inverted`, `noop`, `xor`, `or`, `nor`, `equiv`, `invert`, `or_reverse`, `copy_inverted`, `or_inverted`, `nand`, and `set`. Default is `copy`.

`plane_mask(V)`
: A bitmask specifying which planes of destination are affected by operations. Default is all planes (i.e., -1).

`foreground(V)`
: Foreground pixel value. Default is the pixel value for black.

`background(V)`
: Background pixel value. Default is the pixel value for white.

`line_width(V)`
: Width of drawn lines, in pixels. A value of 0 means *thin* lines. These may be drawn somewhat faster than width 1 lines. Default is 0.

`line_style(V)`
: How (and if) lines are to be dashed. Possible values are `solid`, `double_dash`, or `on_off_dash`. Default is `solid`.

`cap_style(V)`
: How wide lines are to be capped. Possible values are `not_last`, `butt`, `round`, and `projecting`. Default is `butt`.

`join_style(V)`
: How wide connecting lines are to be joined. Possible values are `miter`, `round`, and `bevel`. Default is `miter`.

`fill_style(V)`
: How filling is to be done. Possible values are `solid`, `tiled`, `opaque_stippled`, and `stippled`. Default is `solid`.

`fill_rule(V)`
: How to decide which parts of a figure are to be filled when the lines specifying the figure cross. Possible values are `odd_even_rule` or `winding_rule`. Default is `odd_even_rule`.

`arc_mode(V)`
: How arcs are to be filled. Possible values are `pie_slice` or `chord`. Default is `pie_slice`.

`tile(V)`
: The pixmap for tiling operations. This is only used if `file_style(tiled)` is selected. Default is a pixmap of unspecified size filled with the foreground color.

`stipple(V)`
: A 1 plane pixmap (a bitmap) for stipple operations. This is only used if `file_style(opaque_stippled)` or `file_style(stippled)` is selected. The default is a bitmap of unspecified size filled with 0.

`ts_x_origin(V)`
: X offset for tile and stipple operations. Default is 0.

`ts_y_origin(V)`
: Y offset for tile and stipple operations. Default is 0.

`ts_origin(X, Y)`
: Same as `ts_x_origin(X)`, `ts_y_origin(Y)`.

`font(V)`
: The font used for text display. The default is implementation dependent.

`subwindow_mode(V)`
: Should drawing operations affect mapped subwindows of a windows being drawn into? Possible values are `clip_by_children` and `include_inferiors`. Default is `clip_by_children`.

`graphics_exposures(V)`
: Should graphics exposure events be generated when copying from a window? Possible values are `true` and `false`. Default is `true`.

`clip_x_origin(V)`
: X origin for clipping. Default is 0.

`clip_y_origin(V)`
: Y origin for clipping. Default is 0.

`clip_origin(X, Y)`
: Same as `clip_x_origin(X)`, `clip_y_origin(Y)`.

`clip(V)`
: Specifies how to restrict drawing operations to the destination. V may be `none`, a bitmap, or a list of `rectangle(X, Y, Width, Height)` terms. If it is a bitmap, only pixels in the destination corresponding to 1 bits in the bitmap will be affected by drawing. If a list of `rectangle/4` terms, only pixels in the destination that fall within one of the rectangles will be affected. Default is `none`.

`clip(List, Order)`
: same as `clip(List)`, except that *List* must be a list of `rectangle(X, Y, Width, Height)` terms, and further, *Order* describes the order of the terms in the list. Possible values are `unsorted`, `y_sorted`, `y_x_sorted`, and `y_x_banded`. The default is that this attribute doesn't apply, since by default there is no clipping.

`dashes(V)`
: A list of alternating integer on and off lengths for dashed lines. This attribute is only used when `line_style(double_dash)` or `line_style(on_off_dash)` are selected. Default value is `[4, 4]` (which is the same as `[4]` since the list is used cyclically).

`dash_offset(V)`
: An integer specifying where in the `dashes(V)` list to begin drawing. Default is 0.

For more detailed information on particular attributes, see any good book on programming the X Window System. There are several good references on Xlib, the C interface to the X Window System.

If you are still confused about graphics attributes, the easiest, and often most effective, way to understand them is to experiment. ProXL makes this very easy. Simply open a window and use the drawing commands to draw into it. Then experiment with changing graphics attributes until you get the effect you want.

16.6.2 Finding and Changing Graphics Attributes

These primitives take a *gcable* as argument, and return or change the graphics attributes of that gcable. A gcable is either a window, a pixmap, or a gc. By specifying a window or pixmap in calls to these primitives, you are finding or changing the behavior of drawing commands on that window or pixmap when no gc is specified in the call.

16.6.2.1 `get_graphics_attributes/2`

```
get_graphics_attributes(+Gcable, ?Attribs)
```

Gcable is a gcable, and *Attribs* is a list of (some of) its current Attribute settings.

16.6.2.2 `put_graphics_attributes/2`

```
put_graphics_attributes(+Gcable, +Attribs)
```

Gc is a gc, and *Attribs* is a list of new Attribute settings. It is important that the user *not* change a default GC. All valid gcs except for default gcs have the next field set to NULL (0).

16.6.2.3 Example

This is very important, so it's worth taking a moment to give an example. If you have a window that you wish to drawing 5-pixel-wide solid (not dashed) lines in, and you want the ends of the line segments to be rounded, you might do:

```
graphics_attributes(Window, [line_width(5), cap_style(round)])
```

That's all there is to it. You don't need to specify the `line_style(solid)` attribute, because this is the default. You may count on the defaults; they are often what you want anyway. It is perfectly possible to create a window and start drawing into it without setting any graphics attributes, because the defaults may be good enough.

16.6.3 Creating and Destroying GCs

The following procedures may be used to create and destroy GCs.

16.6.3.1 `create_gc/[2,3]`

```
create_gc(-Gc, +Attribs)
create_gc(-Gc, +Drawable, +Attribs)
```

Gc is a newly-created gc with attributes specified by *Attribs*. *Drawable* is a drawable, which indicates things like the appropriate depth; *Drawable* defaults to the root window of the default display.

16.6.3.2 `release_gc/1`

```
release_gc(+Gc)
```

Inform ProXL that *Gc* is no longer being "held onto" by the programmer. As soon as no drawables refer to it, *Gc* will be destroyed.

Warning: A gc is considered to be "held onto" when it is created, or when it is got from a drawable (e.g. `get_window_attributes`). It is the responsibility of the programmer to ensure that she is not releasing a gc that is being held by another branch of her code. For example, If a gc is created, asserted somewhere, and put into a drawable, and later is accessed through the drawable, the programmer *must not* release it at this point, since it is still in the database and may be used. *Be careful.*

16.6.3.3 Using Gcs

As we have said, using gcs and using graphics attributes of drawables may be freely intermixed. For example, you may have an application that usually wants a certain set of graphics attributes for a window, but occasionally wants a very different set. You could handle this by setting up the graphics attributes of the window, and changing them when necessary. Or you could create two gcs, one with each of the needed sets of graphics attributes, and specify the correct gc in each drawing command. Or you could set the graphics attributes of the window to the most often needed configuration, and create a separate gc with the alternate set of attributes, and only specify a gc argument in drawing commands when you need the alternate set. Each of these approaches has its advantages and disadvantages; which you choose is up to you.

It is also possible to set the gc a drawable will use when no gc is specified in a drawing command. If you need to create many windows or pixmaps with the same graphics attributes, this is an efficient way to do it. You create the gc with the attributes needed by all these windows, and then when you create the windows, you specify this as their gc.

16.6.3.4 Sharing and Cloning of Gcs

Internally, each drawable has a gc. All of the drawing primitives that allow an optional gc will use this gc when no gc is specified. By setting a drawable's gc, it is possible to have several drawables share a common gc. Then, by changing this gc, you have changed the drawing behavior (when no gc is specified in the drawing command) of all the drawables that share this gc.

It is important, however, to understand the difference between changing the graphics attributes of a drawable and changing the graphics attributes of a gc. When you change the graphics attributes of a drawable, if any other drawable shares that gc, then the gc is cloned before modifying it. This means that changing the graphics attributes of a drawable *cannot* change the graphics attributes of any other drawable. However, changing the graphics attributes of a gc *will* change the graphics attributes of any drawable that uses that gc.

In fact, when you create a drawable but don't specify any graphics attributes, that drawable shares a gc with any other drawables that that haven't specified any graphics attributes. But the first time you specify graphics attributes for that drawable, its gc will be cloned. From then on, unless you explicitly give it a gc, its gc will not be shared, so any changes you make to its graphics attributes will be entirely private to that drawable.

16.6.4 Checking GC validity

The following primitives may be used to check whether a gc or a gcable is valid, and to find the gc associated with a gcable.

16.6.4.1 `valid_gc/1`

```
valid_gc(+Gc)
```

Gc is a valid gc. I.e., it has not been destroyed.

16.6.4.2 `ensure_valid_gc/2`

```
ensure_valid_gc(+Gc, +Goal)
```

Gc is a valid gc. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.6.4.3 `valid_gcable/2`

```
valid_gcable(+Gcable, -Gc)
```

Gc is a valid gcable. I.e., it has not been destroyed. *Gc* is the real gc.

16.6.4.4 `ensure_valid_gcable/3`

```
ensure_valid_gcable(+Gcable, -Gc, +Goal)
```

Gcable is a valid gcable. If it's not, an error message mentioning *Goal* is printed, and execution is aborted. *Gc* is the real gc.

16.7 Fonts

Fonts determine how text will look when drawn in a drawable. The set of fonts actually available at any time is implementation dependent.

16.7.1 Font Attributes

Font attributes are:

`direction(V)`
: Does the font draw from left to right or right to left? This is just a hint. Possible values are `left_to_right` or `right_to_left`.

`min_char(V)`
: Character code of the lowest represented character in this font.

`max_char(V)`
: Character code of highest represented character in this font.

`min_charset(V)`
: The lowest represented character set in this font. This is an integer between 0 and 255.

`max_charset(V)`
: The highest represented character set in this font. This is an integer between 0 and 255.

`all_chars_exist(V)`
: Do all characters between the `min_char` and the `max_char` in character sets between the `min_charset` and the `max_charset` in this font have nonzero size? Possible values are `true` and `false`.

`default_char(V)`
: The character code of the character printed for missing characters.

`ascent(V)`
: The number of pixels in the font above the base line. This is the nominal ascent for the font; some characters may write above this point.

`descent(V)`
: The number of pixels in the font below at or below the base line. This is the nominal ascent for the font; some characters may write below this point.

`height(V)`
: The font's ascent `+` descent. This is the nominal height for the font; some characters may be taller.

`property(N, V)`
: *V* is the value of the *N* property of the font. If *N* is unbound, backtrack through all the properties of the font. If *N* is bound, it should be an atom. *V* will be bound to an integer. Consult a good reference on X for information about font properties.

`max_lbearing(V)`
: The maximum number of pixels left of the base point of any character in this font.

`max_rbearing(V)`
: The maximum number of pixels at or to the right of the base point of any character in this font.

`max_width(V)`
: The width of widest character in this font.

`max_ascent(V)`
: The largest height above baseline of any character in this font.

`max_descent(V)`
: Largest height at or the below baseline of any character in this font.

`min_lbearing(V)`
: Minimum number of pixels to the left of the base point of any character in this font.

`min_rbearing(V)`
: Minimum number of pixels at or to the right of the base point of any character in this font.

`min_width(V)`
: Width of narrowest character in this font.

`min_ascent(V)`
: Smallest height above baseline of any character in this font.

`min_descent(V)`
: Smallest height at or below the baseline of any character in this font.

`char_lbearing(C, V)`
: *V* is the number of pixels left of the base point for the character whose character code is *C*.

`char_rbearing(C, V)`
: *V* is the number of pixels right of the base point of character *C*.

`char_width(C, V)`
: *V* is width of character *C*.

`char_ascent(C, V)`
: *V* is height above the baseline of character *C*.

`char_descent(C, V)`
: *V* is height at or below the baseline of character *C*.

`char_height(C, V)`
: *V* is the ascent plus the descent of character *C*.

`char_attribute_bits(C, V)`
: *V* is the attribute bits, represented as an integer, of character *C*. The meaning of the attributes is not defined by X.

Font attributes are *read only*, it is not possible to change them.

16.7.2 Loading and Unloading Fonts

Before a font can be used, it must be loaded. Then it must be specified as the value of the *font* graphics attribute of the destination drawable or the gc to be used for drawing.

16.7.2.1 load_font/[2,3]

```
load_font(+Name, -Font)
load_font(+Display, +Name, -Font)
```

Font is the font whose name is *Name* living on *Display*, which defaults to the default display.

16.7.2.2 release_font/1

```
release_font(+Font)
```

Inform ProXL that *Font* is no longer being "held onto" by the programmer. As soon as no gcs refer to it, *Font* will be unloaded.

Warning: A font is considered to be "held onto" when it is created, or when it is got from a gc (e.g., by `get_graphics_attributes`). It is the responsibility of the programmer to ensure that she is not releasing a font that is being held by another branch of her code. For example, If a font is created, asserted somewhere, and put into a gc, and later is accessed through the gc, the programmer *must not* release it at this point, since it is still in the database and may be used. *Be careful.*

16.7.3 Finding Font Attributes

It is not possible to change font attributes, only to examine them.

16.7.3.1 get_font_attributes/2

```
get_font_attributes(+Fontable, +Attributes)
```

Attributes is a list of attributes of *Fontable*.

16.7.4 The Font Search Path

It is possible to examine and change the path that will be searched when a font is to be loaded. Notice that the font search path applies to *all* clients, not just you, so setting it should be done only with great care.

16.7.4.1 get_font_path/[1,2]

```
get_font_path(-Directories)
get_font_path(+Display, -Directories)
```

Directories is a list of Prolog atoms specifying the font search path for *Display* (defaults as usual).

16.7.4.2 set_font_path/[1,2]

```
set_font_path(+Directories)
set_font_path(+Display, +Directories)
```

Directories is a list of Prolog atoms to become the font search path for *Display* (defaults as usual). Note this changes the search path for all clients using that display!

16.7.5 What Fonts Are Available?

16.7.5.1 current_font/[1,2,3,4]

```
current_font(-Name)
current_font(+Pattern, -Name)
current_font(+Limit, +Pattern, -Name)
current_font(+Display, +Limit, +Pattern, -Name)
```

Name is the name of a currently available font on *Display* (which defaults as usual). If *Pattern* is given, it restricts *Name* to fonts matching it. If *Limit* is given, it is an upper limit on the number of fonts that will be backtracked through (defaults to one million).

16.7.5.2 current_font_attributes/[2,3,4,5]

```
current_font_attributes(-Name, +Attributes)
current_font_attributes(+Pattern, -Name, +Attributes)
current_font_attributes(+Limit, +Pattern, -Name, +Attributes)
current_font_attributes(+Display, +Limit, +Pattern, -Name, +Attributes
)
```

This is just like `current_font/[1,2,3,4]`, except that *Attributes* is a list of attributes of the font named *Name*. The first three call the last with the appropriate defaults. Any valid font attributes may be used in *Attributes*, except for '`char_`' attributes. This is a limitation of X11.

16.7.6 The Size of a String

16.7.6.1 text_width/3

```
text_width(+Fontable, +String, -Width)
```

Width is the width in pixels of *String* when drawn in *Fontable*. *String* may be either an atom or list of character codes.

16.7.6.2 text_extents/[7,8]

```
text_extents(+Fontable, +String, -Lbearing, -RBearing,
             -Width, -Ascent, -Descent)
text_extents(+Fontable, +String, -Lbearing, -RBearing,
             -Width, -Ascent, -Descent, -Attribute_bits)
```

How much space would be occupied if *String* were drawn in in *Fontable*? *Lbearing* is the number of pixels to the left of the origin point; *RBearing* is the number of pixels to the right of the origin point; *Ascent* is the number of pixels above the origin point; and *Descent* is one greater than the number of pixels below the origin point. The reason *Descent* is one greater than what you would expect is that it allows you to add *Ascent* and *Descent* to determine the height of *String*.

String may be either an atom or list of character codes.

16.7.6.3 query_text_extents/[7,8]

```
query_text_extents(+Fontable, +String, -Lbearing, -RBearing,
                   -Width, -Ascent, -Descent)
query_text_extents(+Fontable, +String, -Lbearing, -RBearing,
                   -Width, -Ascent, -Descent, -Attribute_bits)
```

How much space will be occupied if *String* were drawn in *Fontable*? *Lbearing* is the number of pixels to the left of the origin point; *RBearing* is the number of pixels to the right of the origin point; *Ascent* is the number of pixels above the origin point; and *Descent* is one greater than the number of pixels below the origin point. The reason *Descent* is one greater than what you would expect is that it allows you to add *Ascent* and *Descent* to determine the height of *String*.

String may be either an atom or list of character codes.

This differs from `text_extents` only in that it does not force ProXL to load the sizes of all the characters in the font from the server. If this information has already been loaded, `query_text_extents` will use it. It is usually better to use `text_extents`, since once the information is loaded from the server, it is *much* faster to determine the size of a string. So if you will ever want to find the size of another string in the same font, you should probably use `text_extents`.

16.7.7 Checking Font Validity

The following procedures may be used to check whether a font or fontable is valid, and to find the font associated with a fontable.

16.7.7.1 valid_font/1

```
valid_font(+Font)
```

Font is a valid font. I.e., it has not been destroyed.

16.7.7.2 ensure_valid_font/2

```
ensure_valid_font(+Font, +Goal)
```

Font is a valid font. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.7.7.3 valid_fontable/2

```
valid_fontable(+Fontable, -Font)
```

Fontable is a valid fontable. A fontable is either a font, or something from which we can determine a font. This means a gc, or a window or pixmap, which have an associated gc. *Font* is the real font corresponding to *Fontable*.

16.7.7.4 ensure_valid_fontable/3

```
ensure_valid_fontable(+Fontable, -Font, +Goal)
```

Fontable is a valid fontable. If it's not, an error message mentioning *Goal* is printed, and execution is aborted. *Font* is the real font corresponding to *Fontable*.

16.8 Colors and Colormaps

This section describes several parts of the X window system related to color. It describes colors: red, green, blue triples; it describes colormaps: mappings from pixel values to colors; and it describes visuals: specifications of how colors will be appear physically on the screen.

16.8.1 Color Specifications

Colors are specified in ProXL as `color(R,G,B)` terms, where R, G, and B are (usually floating point) numbers between 0 and 1 inclusive. Alternately, a color can be specified as an atom naming a color (case is not significant), such as `blue` or `plum`, or as an atom of the form `'#RGB'` or `'#RRGGBB'` or `'#RRRGGGBBB'` or `'#RRRRGGGGBBBB'` where R, G, and B are hex digits. For example, `'#05F'` would represent a color with no red, 5/15 (or 0.333333) green, and full blue. In `put_color` and `put_colors`, *Red*, *Green*, and/or *Blue* in a `color(Red,Green,Blue)` term can be the atom `none`, which means that that primary won't be changed in the colormap.

Pixel values are simply integers.

16.8.2 Visuals

Visuals are Prolog terms of one of the following forms:

```
gray(Writable, Depth, Size)

direct_color(Writable, Depth, Size,Bits_per_RGB, Rmask, Gmask, Bmask)

pseudo_color(Writable, Depth, Size, Bits_per_RGB)
```

Visual terms describe the color capability of a screen or window. A `gray/3` visual means that pixel values in a window specify an entry in a colormap, but only one of the primary colors is actually used to drive the grayscale (or monochrome) display [6]. `direct_color/7` means that the window or screen support color hardware where separate bits in a pixel value specify entries in separate colormaps for each of the three primary colors. `pseudo_color/4` means that each pixel value specifies an entry in a colormap, which specifies all three primary colors. *Writable* is either `true` or `false`, indicating whether or not is is possible to allocate and change cells in a colormap. *Depth* specifies the number of bitplanes that are supported on this screen or window. *Size* is the number of user-accessible colormap entries; this will always be less than or equal to 2 to the power of *Depth*. *Bits_per_RGB* is the number of bits used to specify each primary in a color specification. In some sense, this is the *precision* of color specifications. And finally, *Rmask*, *Gmask*, and *Bmask* are bit masks (i.e. integers) that specify which bits of a `direct_color` visual pixel specify the red, green, and blue primary, respectively.

16.8.3 Using Colors

There are five different ways you can use color in ProXL, in approximate order of increasing difficulty:

1. You can use the `black_pixel` and `white_pixel` of your screen. `black_pixel` is the default foreground color, so if you don't change your drawable's foreground graphics attribute, you will be drawing in black. And the default window background is its screen's `white_pixel`, so if you don't change that, you will be drawing in black on a white background, which is often good enough.
2. You can allocate *colors*. If you want to have color images, this is usually the best approach. Allocating a color never fails, even on a monochrome screen, it just gives you the *closest* color it can. So if you ask for yellow and navy blue, you will get

[6] X does not specify which color drives the display, so *R*, *G* and *B* should all be the same for grayscale visuals.

white and black, respectively. This allows your application to run on monochrome (or grayscale) screens, as well as color.

When you allocate colors, you share these colors with other applications, too. This means that if you ask for blue, and another application is already using blue, you will get the same pixel value. Obviously this is good citizenship on a color screen with a limited number of pixel values available. It also means that you cannot change the color of a pixel value you get this way, because it would confound the other applications using this pixel value.

3. You can allocate *color cells* and *planes*. This gives you private pixel values for you to play with. You can set these pixel values to any colors you like, and change them as often as you like. Allocating color planes also allows you to control the actual bit patterns of the pixels you use, allowing you to do sophisticated tricks with images.

4. You can use a *standard colormap*. Standard colormaps provide a number of preselected colors, and can be shared with other applications. You must rely on your window manager to create standard colormaps for you, and, unfortunately, not all window managers do this.

5. You can allocate a *colormap*. While this might seem harmless enough, since your color table is yours alone, and doesn't affect the colormaps used by other applications, in practice few screens can support more than one colormap at a time. This means that when you colormap is installed, *all the other windows on the screen may display the wrong colors*. Also, if you use a private colormap, you must make sure that your colormap is installed. In theory, your window manager is responsible for installing your colormap when the pointer is in your window. In practice, few window managers do this now. This will probably change, but at the moment, this is a serious problem.

16.8.4 Allocating and Freeing Colors

16.8.4.1 `alloc_color/[2,3,4,5]`

```
alloc_color(+Color, -Pixel)
alloc_color(+Colormapable, +Color, -Pixel)
alloc_color(+Colormapable, +Color, -Pixel, -Actual_color)
alloc_color(+Colormapable, +Color, -Pixel, -Actual_color, -Exact_color
)
```

Pixel is a (possibly newly allocated) pixel value for the colormap associated with *Colormapable* that is as close as possible to *Color*. *Color* is a color spec, as described in Section 16.8.1 [pxl-col-cs], page 838. *Actual_color* is the actual color of *Pixel* in *Colormapable*, and *Exact_color* is the color that should have been allocated (the same as *Color*, if it is a `color/3` term). *Colormapable* defaults to the default screen (which means the color is allocated in the default screen's default colormap).

16.8.4.2 parse_color/[2,3]

```
parse_color(+Color, -Exact_color)
parse_color(+Screen, +Color, -Exact_color)
```

Exact_color is the color associated with *Color* on *Screen* (or default screen). Fails if *Color* is not a valid color spec.

16.8.4.3 free_colors/[2,3]

```
free_colors(+Colormapable, +Pixels)
free_colors(+Colormapable, +Pixels, +Planes)
```

Pixels, a list of pixel values allocated from the colormap associated with *Colormapable*, is freed, so that those pixel values can be allocated later. *Colormapable* defaults to the default colormap of the default screen. *Planes*, if supplied, is a list of plane masks, which is also freed.

16.8.5 Standard Colormaps

16.8.5.1 get_standard_colormap/[2,3]

```
get_standard_colormap(+Name, -Colormap)
get_standard_colormap(+Screen, +Name, -Colormap)
```

Colormap is a standard colormap on *Screen* that you may use, and that may be shared with other applications running on the same screen. *Name* is the name of a standard colormap. *Screen* defaults to the default screen.

These procedures rely on another program, usually the window manager, to actually create the standard colormaps. If your window manager doesn't do this, then these procedures will report an error.

16.8.6 Allocating Color Cells and Planes

16.8.6.1 alloc_color_cells/5 and alloc_contig_color_cells/5

```
alloc_color_cells(+Colormapable, +NColors, -Pixels,
                  +NPlanes, -Planes)
alloc_contig_color_cells(+Colormapable, +NColors, -Pixels,
                         +NPlanes, -Planes)
```

Pixels is a list of *NColors* pixel values (integers), and *Planes* is a list of *NPlanes* plane masks that have been allocated from the colormap associated with *Colormapable*. *Colormapable*

defaults to the default colormap of the default screen. For `alloc_contig_color_cells/5`, *Pixels* are sequential numbers.

16.8.6.2 `alloc_color_planes/[8,9]` and `alloc_contig_color_planes/[8,9]`

```
alloc_color_planes(+NColors, -Pixels, +NReds, +NGreens, +NBlues,
                   -RMask, -GMask, -BMask)
alloc_color_planes(+Colormapable, +NColors, -Pix-
els, +NReds, +NGreens, +NBlues,
                   -RMask, -GMask, -BMask)
alloc_contig_color_planes(+NColors, -Pix-
els, +NReds, +NGreens, +NBlues,
                          -RMask, -GMask, -BMask)
alloc_contig_color_planes(+Colormapable, +NColors, -Pix-
els, +NReds, +NGreens, +NBlues,
                          -RMask, -GMask, -BMask)
```

Pixels is a list of *NColors* pixel values (integers), and *RMask*, *GMask*, and *BMask* indicated *NReds*, *NGreens*, and *NBlues* planes, respectively, allocated from the colormap associated with *Colormapable*. *Colormapable* defaults to the default colormap of the default screen. For `alloc_contig_color_planes/9`, *RMask*, *GMask*, and *BMask* have contiguous bits turned on.

16.8.6.3 Freeing Color Cells and Planes

Color cells and planes are freed by `free_colors/[2,3]`, as described in Section 16.8.4.3 [pxl-col-alc-free_colors], page 841 above.

16.8.7 Finding and Changing Colors

The colors of shared pixels cannot be changed.

16.8.7.1 `put_color/[2,3]`

```
put_color(+Pixel, +Color)
put_color(+Colormapable, +Pixel, +Color)
```

Install *Color* as the color of *Pixel* in the colormap associated with *Colormapable*. *Color* is a color specification, as described above. For these procedures, colors may be specified as `color(R,G,B)` terms, where any of *R*, *G*, and/or *B* may be the atom `none`, in which case this component of the color is not set. *Colormapable* defaults to the default colormap of the default screen.

16.8.7.2 put_colors/[1,2]

```
put_colors(+Pixel_colors)
put_colors(+Colormapable, +Pixel_colors)
```

Install colors in the colormap associated with *Colormapable* as specified by *Pixel_colors*. *Pixel_colors* is a list of *Pixel-Color* terms, where *Color* is to be the color of Pixel in the colormap associated with *Colormapable*. Color may be color specification, as described above. For these procedures, colors may also be specified as `color(R,G,B)` terms, where any of *R*, *G*, and/or *B* may be the atom `none`, in which case this component of the color is not set. *Colormapable* defaults to the default colormap of the default screen.

16.8.7.3 get_color/[2,3]

```
get_color(+Pixel, -Color)
get_color(+Colormap, +Pixel, -Color)
```

Color is the color of *Pixel* in the colormap associated with Colormapable. *Color* is specified as a `color(R,G,B)` term, and *Pixel* must be an integer. Colormapable defaults to the default colormap of the default screen.

16.8.7.4 get_colors/[1,2]

```
get_colors(+Pixel_colors)
get_colors(+Colormapable, +Pixel_colors)
```

Pixel_colors is a list of *Pixel-Color* terms, where *Pixel* is bound at call time, and *Color* will be bound to a `color(R,G,B)` term indicating the color of Pixel in the colormap associated with *Colormapable*. *Colormapable* defaults to the default colormap of the default screen.

16.8.8 Creating and Freeing Colormaps

16.8.8.1 create_colormap/[1,2,3]

```
create_colormap(-Colormap)
create_colormap(?Visual, -Colormap)
create_colormap(+Screen, ?Visual, -Colormap)
```

Colormap is a newly created colormap on *Screen* using *Visual*. *Screen* defaults to the default screen. *Visual* defaults to the *Screen*'s default visual.

16.8.8.2 create_colormap_and_alloc/[1,2,3]

```
create_colormap_and_alloc(-Colormap)
create_colormap_and_alloc(+Visual, -Colormap)
create_colormap_and_alloc(+Screen, +Visual, -Colormap)
```

Colormap is a newly created colormap on *Screen* using *Visual*. All of the color cells in *Colormap* are allocated for your use. *Screen* defaults to the default screen. *Visual* defaults to the default screen's default visual.

16.8.8.3 free_colormap/1

```
free_colormap(+Colormap)
```

Free *Colormap*. If *Colormap* is the colormap of any windows, it will be yanked out from under them. If *Colormap* is the default colormap of a screen, it is not really freed. *Be careful*.

16.8.8.4 copy_colormap_and_free/2

```
copy_colormap_and_free(+Old_cmap, -New_cmap)
```

New_cmap is a newly allocated, non-shared, colormap containing all the colors you have allocated out of *Old_cmap* (which is probably shared).

16.8.9 Colormap Installation

Remember, installing your own colormap is very antisocial. This is the window manager's job. But if the window manager isn't doing its job. . . .

16.8.9.1 install_colormap/1

```
install_colormap(+Colormap)
```

Make sure *Colormap* is installed on its screen.

16.8.9.2 uninstall_colormap/1

```
uninstall_colormap(+Colormap)
```

Remove *Colormap* from the required list for its screen.

16.8.9.3 installed_colormap/[1,2]

```
installed_colormap(-Colormap)
installed_colormap(+Screen, -Colormap)
```

Colormap is a colormap that is installed on *Screen*.

16.8.10 Checking Colormap Validity

16.8.10.1 valid_colormap/1

```
valid_colormap(+Colormap)
```

Colormap is a valid colormap, which hasn't been destroyed.

16.8.10.2 valid_colormapable/2

```
valid_colormapable(+Colormapable, -Colormap)
```

Colormapable is a valid colormapable, which hasn't been destroyed.

16.8.10.3 ensure_valid_colormap/2

```
ensure_valid_colormap(+Colormap, +Goal)
```

Colormap is a valid colormap. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.8.10.4 ensure_valid_colormapable/3

```
ensure_valid_colormapable(+Colormapable, -Colormap, +Goal)
```

Colormap is the valid colormap associated with *Colormapable*. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.9 Pixmaps and Bitmaps

16.9.1 Pixmap Attributes

Following is a list of pixmap attributes:

`*width(`*W*`)`
: The width of pixmap, in pixels. Default is 100.

`*height(`*H*`)`
: The height of pixmap, in pixels. Default is 100.

`*size(`*W, H*`)`
: The same as `width(W), height(H)`.

`*depth(`*D*`)`
: The number of bits per pixel. Default is the screen's depth.

`*screen(`*S*`)`
: The screen on which this pixmap can be used. Default is the default screen.

`gc(`*V*`)`
: Default graphics context for drawing on this pixmap. Default is the default gc for the the pixmap's screen.

* — starred items cannot be modified once a pixmap is created.

16.9.2 Finding and Changing Pixmap Attributes

16.9.2.1 `get_pixmap_attributes/[2,3]`

```
get_pixmap_attributes(+Pixmap, +Attributes)
get_pixmap_attributes(+Pixmap, +Attributes, +Graphics_attribs)
```

Attributes are a subset of *Pixmap*'s current attributes. *Graphics_attribs* is a subset of *Pixmap*'s graphics attributes.

16.9.2.2 `put_pixmap_attributes/[2,3]`

```
put_pixmap_attributes(+Pixmap, +Attributes)
put_pixmap_attributes(+Pixmap, +Attributes, +Graphics_attribs)
```

Pixmap is modified so that *Attributes* are a subset of its attributes. The only pixmap attribute that may be changed is its gc. If *Graphics_attribs* is specified, it is a list of graphics attributes to be given to the pixmap. `put_pixmap_attributes(P,A,G)` is equivalent to `put_pixmap_attributes(P,A), put_graphics_attributes(P,G)`, but is slightly more efficient. But mainly it's more convenient.

16.9.3 Creating and Freeing Pixmaps

16.9.3.1 `create_pixmap/[2,3]`

```
create_pixmap(-Pixmap, +Attributes)
create_pixmap(-Pixmap, +Attributes, +Graphics_attribs)
```

Pixmap is a newly created pixmap having the specified attributes. If *Graphics_attribs* is given, it is a list of graphics attributes to be given to the pixmap. `create_pixmap(P,A,G)` is equivalent to `create_pixmap(P,A), put_graphics_attributes(P,G)`, but slightly more efficient. But mainly it's more convenient.

16.9.3.2 `free_pixmap/1`

```
free_pixmap(+Pixmap)
```

Pixmap is freed. It can no longer be used.

16.9.4 Reading and Writing Bitmap Files

These routines allow you to read and write files containing pixmaps of depth 1. Unfortunately, there is no established file format for pixmaps of greater depth.

16.9.4.1 `read_bitmap_file/[2,3,4,5]`

```
read_bitmap_file(+Filename, -Pixmap)
read_bitmap_file(+Filename, -Pixmap, -X_hot, -Y_hot)
read_bitmap_file(+Filename, +Screen, -Pixmap)
read_bitmap_file(+Filename, +Screen, -Pixmap, -X_hot, -Y_hot)
```

Pixmap is the bitmap encoded in file *Filename*. If *Screen* is supplied, it is *Pixmap*'s screen, if not it defaults to the default screen. If *X_hot* and *Y_hot* are asked for, they are the X and Y components of *Pixmap*'s hotspot, if it is specified in the file, and -1 if not specified in the file.

16.9.4.2 `write_bitmap_file/[2,4]`

```
write_bitmap_file(+Filename, +Pixmap)
write_bitmap_file(+Filename, +Pixmap, +X_hot, +Y_hot)
```

Writes *Pixmap* out to file *Filename* in X11 standard encoding. If *X_hot* and *Y_hot* are supplied, they specify the hotspot for the pixmap, else there is no hotspot.

16.9.5 Checking Pixmap Validity

16.9.5.1 `valid_pixmap/1`

```
valid_pixmap(+Pixmap)
```

Pixmap is a valid pixmap. I.e., it has not been destroyed.

16.9.5.2 ensure_valid_pixmap/2

```
ensure_valid_pixmap(+Pixmap, +Goal)
```

Pixmap is a valid pixmap. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.10 Cursors

Cursors do not have attributes, as there is nothing about a cursor that can be determined.

16.10.1 Creating and Freeing Cursors

16.10.1.1 create_cursor/[2,3,4,5]

```
create_cursor(+Cursor_spec, -Cursor)
create_cursor(+Display, +Cursor_spec, -Cursor)
create_cursor(+Cursor_spec, -Cursor, +Foreground_color, +Back-
ground_color)
create_cursor(+Display, +Cursor_spec, -Cursor, +Foreground_color,
              +Background_color)
```

Cursor is a newly created cursor on *Display*, as specified by *Cursor_spec*. If *Display* is not specified, the display of the default screen is used. If *Foreground_color* and *Background_color* are specified, they must be color specifications (see Section 16.8 [pxl-col], page 838) indicating the color to give to the two parts of the cursor, otherwise black and white are used. Note that *Foreground_color* and *Background_color* are *not* pixel values.

Cursor_spec must be one of these:

An atom which names a font cursor.

`pixmap_cursor(`*Source*`, `*Mask*`, `*X_hot*`, `*Y_hot*`)`
: Where *Source* and *Mask* are pixmaps of the same size, and *X_hot* and *Y_hot* are integers specifying the hot spot of the cursor, relative to the upper right corner of the pixmaps.

`glyph_cursor(`*Source_font*`, `*Source_char*`)`
: Where *Source_font* specifies a font and *Source_char* is the character code of the character in that font to be used as the image of the cursor.

`glyph_cursor(`*Source_font*`, `*Source_char*`, `*Mask_char*`)`
: Where *Source_font* a font and *Source_char* and *Mask_char* are the character codes of the characters in that font to be used as source and mask image of the cursor.

`glyph_cursor(`*`Source_font`*`, `*`Source_char`*`, `*`Mask_font`*`, `*`Mask_char`*`)`
: Where *Source_font* specifies a font and *Source_char* is the character code of the character in that font to be used as the foreground image of the cursor, and *Mask_font* and *Mask_char* similarly specify a mask image.

Obviously, specifying a cursor by name is simplest. In the other cases, the cursor is specified as a source and mask bitmap, and a hotspot. The bits that are turned off in the mask bitmap are transparent in the cursor. The bits turned on in both the mask and source bitmaps will appear in the cursor's foreground color. The remaining bits appear in the cursor's background color. For `glyph_cursor/2` terms, the same image is used as both source and mask, so there is no background.

16.10.1.2 `free_cursor/1`

```
free_cursor(+Cursor)
```

Free *Cursor*

16.10.2 Cursor Utilities

16.10.2.1 `recolor_cursor/3`

```
recolor_cursor(+Cursor, +Foreground_color, +Background_color)
```

Change the color of *Cursor*

16.10.2.2 `query_best_cursor/[4,5]`

```
query_best_cursor(+Width, +Height, -Best_width, -Best_height)
query_best_cursor(+Screen, +Width, +Height, -Best_width, -Best_height)
```

Best_width and *Best_height* are the *best* size for a cursor on *Screen* that is closest to *Width* and *Height*.

16.10.3 Checking Cursor Validity

16.10.3.1 `valid_cursor/1`

```
valid_cursor(+Cursor)
```

Cursor is a valid ProXL cursor.

16.10.3.2 ensure_valid_cursor/2

```
ensure_valid_cursor(+Cursor, +Goal)
```

Cursor must be a valid cursor. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.11 Displays and Screens

This section describes displays and screens for ProXL. We also discuss the concepts of displayables and screenables, and the default screen and display.

Both displays and screens have many attributes that can be examined by the `get_display_attributes/[1,2]` and `get_screen_attributes/[1,2]` predicates. It is not possible to set any attributes of displays or screens.

16.11.1 Display Attributes

Display attributes include:

`bitmap_bit_order(`*V*`)`
: Is leftmost bit in bitmap least or most significant? Either `lsb_first` or `msb_first`.

`bitmap_pad(`*N*`)`
: Number of bits each scanline must be padded to.

`bitmap_unit(`*N*`)`
: Size of the bitmap's unit, in bits.

`connection_number(`*N*`)`
: Connection number for display (`fd` under UNIX).

`default_screen(`*V*`)`
: The default screen for this display.

`display_string(`*S*`)`
: The name of this display, as an atom.

`image_byte_order(`*V*`)`
: Is leftmost byte in scanline least or most significant? Either `lsb_first` or `msb_first`.

`protocol_revision(`*N*`)`
: Minor X protocol revision number.

`protocol_version(`*N*`)`
: The major X protocol version (usually 11).

`q_length(N)`
: Number of events in display's input queue.

`screen(N, S)`
: S is Screen number N of this display (0 origin).

`screen_count(N)`
: Number of screens available on this display.

`server_vendor(V)`
: The name of the supplier of this server, as an atom.

`vendor_release(V)`
: The vendor's release number for this server.

16.11.1.1 get_display_attributes/[1,2]

```
get_display_attributes(+Attribs)
get_display_attributes(+Display, +Attribs)
```

Attribs is a list of attribute specifications. Accepted display attributes are listed above. *Display* defaults to the default display.

16.11.2 Opening and Closing Displays

16.11.2.1 open_display/2

```
open_display(+Displayname, -Display)
```

Display is the newly opened display named *Displayname* (an atom). Fails if it can't open *Displayname*.

16.11.2.2 close_display/1

```
close_display(+Display)
```

Close *Display*. Note that *Display* must be an actual display, not a displayable

16.11.3 Flushing and Syncing Displays

16.11.3.1 flush/[0,1]

```
flush
flush(+Display)
```

Flush the output buffer to *Display*. If *Display* is omitted, flush output to all open X displays.

16.11.3.2 `sync/[0,1]` and `sync_discard/[0,1]`

```
sync
sync(+Display)
sync_discard
sync_discard(+Display)
```

Flush the output buffer of *Display* and wait until all request have been processed by the server. If *Display* is omitted, we default to syncing all open displays. `sync_discard/[0,1]` also throw away all queued events.

16.11.4 Finding Currently Open Displays

16.11.4.1 `current_display/1`

```
current_display(?Display)
```

Display is a currently open ProXL display.

16.11.4.2 `default_display/1`

```
default_display(-Display)
```

Display is the display of the default screen.

16.11.5 Checking Display Validity

16.11.5.1 `valid_display/1`

```
valid_display(+Display)
```

Succeeds if *Display* is a valid, open, display.

16.11.5.2 `valid_displayable/2`

```
valid_displayable(+Displayable, -Display)
```

Display is the ProXL display associated with *Displayable*, a displayable. A displayable is any ProXL resource that has a unique display associated with it, that is, *any* ProXL resource. Fails unless *Display* is a valid, open, display.

16.11.5.3 ensure_valid_display/2

```
ensure_valid_display(+Display, +Goal)
```

Display is a valid display. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.11.5.4 ensure_valid_displayable/3

```
ensure_valid_displayable(+Displayable, -Display, +Goal)
```

Display is the valid display associated with *Displayable*. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.11.6 Screen Attributes

Screen attributes include:

`black_pixel(V)`
: Pixel value for black on this screen. An integer.

`white_pixel(V)`
: Pixel value for white on this screen. An integer.

`cells(N)` Number of entries in default color map.

`colormap(C)`
: The default color map for this screen.

`depth(D)` Number of planes for default root window.

`default_visual(V)`
: Default way pixel values are shown on this screen. See the documentation of visuals in Section 16.8.2 [pxl-col-vis], page 839.

`visual(V)`
: A possible way pixel values can be shown on this screen. Multiple results are possible, one of which will be the `default_visual`.

`does_backing_store(V)`
: Is this screen capable of saving the obscured part of windows so applications don't have to repaint them when they become exposed? Possible values are `when_mapped`, `not_useful`, or `always`.

`does_save_unders(B)`
: Is this screen capable of saving whatever is hidden by a window so when the window is unmapped, the exposed area need not be repainted? Possible values are `true` or `false`.

`display(D)`
: The display of this screen.

`planes(`*N*`)`
: Number of bits/pixel in root window.

`width(`*W*`)` Width of screen, in pixels.

`width_mm(W)`*V*
: Width of screen, in millimeters.

`height(`*H*`)`
: Height of screen, in pixels.

`height_mm(`*H*`)`
: Height of screen, in millimeters.

`size(`*W, H*`)`
: Width and height of screen, in pixels.

`size_mm(`*W, H*`)`
: Width and height of screen in millimeters.

`root(`*V*`)` Root window of this screen.

`screen_number(`*N*`)`
: The number of this screen on its display.

16.11.6.1 `get_screen_attributes/[1,2]`

```
get_screen_attributes(+Attribs)
get_screen_attributes(+Screen, +Attribs)
```

Attribs is a list of screen attributes, as listed above, of the screen uniquely associated with Screenable. *Screen* defaults the current default screen.

16.11.7 The Default Screen

16.11.7.1 `default_screen/2`

```
default_screen(-OldScreen, +NewScreen)
```

OldScreen is the old default screen, and *NewScreen* becomes the new default screen.

16.11.8 Checking Screen Validity

16.11.8.1 `valid_screen/1`

```
valid_screen(+Screen)
```

Succeeds if *Screen* is a screen on a valid, open, display.

16.11.8.2 valid_screenable/2

```
valid_screenable(+Screenable, -Screen)
```

Screen is the ProXL screen associated with *Screenable*, a screenable. A screenable is any ProXL resource that has a unique screen associated with it, that is, a display (which has a default screen), gc, colormap pixmap, or window. Fails unless *Screen* is a screen on a valid, open, display.

16.11.8.3 ensure_valid_screen/2

```
ensure_valid_screen(+Screen, +Goal)
```

Screen is a valid screen. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.11.8.4 ensure_valid_screenable/3

```
ensure_valid_screenable(+Screenable, -Screen, +Goal)
```

Screen is the valid screen associated with *Screenable*. If it's not, an error message mentioning *Goal* is printed, and execution is aborted.

16.11.9 Interfacing with Foreign Code

In order to interface your ProXL code with code written in other languages using the Xlib interface, it will often be necessary to find the *XID* of a ProXL object, or to create a new ProXL object from an XID. These primitives allow you to do this.

16.11.9.1 proxl_xlib/[3,4]

```
proxl_xlib(?ProXLobj, ?Type, ?Xid)
proxl_xlib(?ProXLobj, ?Type, ?Xid, ?Display)
```

ProXLobj is a ProXL object of *Type* on *Display* whose XID is *Xid*. If *Display* is not specified, it is assumed to be the display of the current default screen. *Type* is one of `window`, `pixmap`, `font`, `cursor`, or `colormap`. Either *ProXLobj* or all three of *Type*, *Xid*, and *Display* must be bound. `proxl_xlib/3` fails if *ProXLobj* is bound to a ProXL object on a display other than the display of the default screen.

16.11.9.2 display_xdisplay/2

```
display_xdisplay(?Display, ?Xdisplay)
```

Display is the ProXL display corresponding to the X display *Xdisplay*. May be used to find the ProXL display given an X display, or to find the X display given a ProXL display. But one or the other must be bound.

16.11.9.3 screen_xscreen/2

```
screen_xscreen(?Screen, ?Xscreen)
```

Screen is the ProXL screen corresponding to the X screen *Xscreen*. May be used to find the ProXL screen given an X screen, or to find the X screen given a ProXL screen. But one or the other must be bound.

16.11.9.4 visual_id/[2,3]

```
visual_id(?Visual, ?Visual_id)
visual_id(+Screenable, ?Visual, ?Visual_id)
```

Visual_id is the X visual id corresponding to the visual term *Visual* on the screen associated with *Screenable*. *Screenable* defaults to the default screen.

16.12 Event Handling Functions

Even though the ProXL callback and event handling mechanism is very powerful and easy to use, there are occasions when the user wants to interact directly with the Display connection and handle the incoming events without using the callback mechanism.

This section documents the functions that allow the user to bypass the callback mechanism, as well as a number of *utility* functions to examine the state of the Display event queue.

16.12.1 active_windows/[0,1]

```
active_windows
active_windows(+Displayable)
```

Succeed if there are any ProXL Windows with currently registered callbacks on the given *Displayable*, which is the default Display, if omitted.

16.12.2 events_queued/[2,3]

```
events_queued(+Mode, -Number)
events_queued(+Displayable, +Mode, -Number)
```

Unify *Number* with an integer giving the number of events queued for the given *Displayable*, which if omitted, is the default Display.

If there are currently any events in Xlib's queue, the predicate returns immediately, unifying *Number*, regardless of the value of *Mode*. Otherwise, it behaves according to the value of *Mode*:

`already` Binds *Number* to 0 and succeeds.

`after_reading`
: Attempts to read more events from the Display connection to the X server, without flushing the output buffer, and unifies *Number* to the number of events read.

`after_flushing`
: Like `after_reading`, but also flushes the output buffer.

16.12.3 pending/[1,2]

```
pending(-Number)
pending(+Displayable, -Number)
```

Unify *Number* with the number of events that have been received from the server, but have not been processed yet, for the given *Displayable*, which if omitted, is the default Display.

If there are no events in the queue, the output buffer is flushed and *Number* is unified with the number of events transferred to the input queue as a result of the flush.

16.12.4 new_event/[1,2]

Under ProXL, the event structures used by X11 are not Prolog terms, but ProXL-specific data structures whose implementation details are not visible to the user, and they have to be explicitly allocated and de-allocated[7].

ProXL provides routines to examine and set the contents of these structures. The user should not hang on to any of these structures, and in particular, should not assert them into the data base.

All of the event handling predicates documented in this section have an *XEvent* argument, which is an X11 event structure.

```
new_event(-XEvent)
new_event(-XEvent, +EventValues)
```

Will create a new private, X11 event structure and unify *XEvent* with it.

`new_event/1` creates a new, uninitialized event structure.

`new_event/2` creates a new event structure, and destructively set its fields from the values given in *EventValues*, which must be a list of event fields in the style of the callback

[7] In the current implementation, they are not garbage-collected automatically

mechanism, and unify *XEvent* with the initialized result. The list of event values must specify at least a `type`. If `display` is not specified, the default Display is used.

16.12.5 dispose_event/1

```
dispose_event(+XEvent)
```

Disposes of the given X11 event structure and returns the storage associated with it to the system. References to *XEvent* after it has been disposed of, will certainly cause disasters. *Be careful.*

16.12.6 next_event/[2,3]

```
next_event(-Type, ?XEvent)
next_event(+Displayable, -Type, ?XEvent)
```

Return the next event from the given *Displayable* event queue. If *Displayable* is omitted, the next event on any ProXL display is returned. *Type* is unified to the type of event, and the event is removed from the head of the queue.

If there are no events in the queue, `next_event/[2,3]` flushes the output queue(s) and blocks until an event is received. If *Displayable* is specified, and events for which callbacks are registered arrive on other displays, they will be handled, and `next_event/3` will continue to wait for an event on the specified *Displayable*.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `next_event/[2,3]` will unify *XEvent* with a private, local structure, which the user cannot hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

16.12.7 peek_event/[2,3]

```
peek_event(-Type, ?XEvent)
peek_event(+Displayable, -Type, ?XEvent)
```

Peek at the next event from the given *Displayable* event queue. If *Displayable* is omitted, the next event on any ProXL display is examined. *Type* is unified to the type of event. The event is not removed from the head of the queue.

If there are no events in the queue, `peek_event/[2,3]` flushes the output queue(s) and blocks until an event is received. If *Displayable* is specified, and events for which callbacks are registered arrive on other displays, they will be handled, and `peek_event/3` will continue to wait for an event on the specified *Displayable*.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `peek_event/[2,3]`

will unify *XEvent* with a prvate, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to `peek_event/[2,3]`.

16.12.8 `window_event/4`

```
window_event(+Window, +EventMask, -Type, ?XEvent)
```

Searches the event queue for the *Window*'s Display, and removes the first event that is intended for *Window* and is selected by the given *EventMask*. Unifies *Type* with the type of event removed. Other events in the queue are not discarded.

If there is no qualifying event in the queue, `window_event/4` flushes the output queue and blocks until one is received. If events for which callbacks are registered arrive on other displays, they will be handled, and `window_event/4` will continue to wait for an event on *Window*'s display.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `window_event/4` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

EventMask is an integer bitmask, that specifies the selected events. The predicate `event_list_mask/2` is useful to translate between X11 event mask names and bitmasks.

16.12.9 `check_window_event/4`

```
check_window_event(+Window, +EventMask, -Type, ?XEvent)
```

Is like, `window_event/4`, but fails if there is no matching event in the queue, and does not block waiting for one. The output buffer is flushed only on failure.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `check_window_event/4` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

EventMask is an integer bitmask, that specifies the selected events. The predicate `event_list_mask/2` is useful to translate between X11 event mask names and bitmasks.

16.12.10 `mask_event/[3,4]`

```
mask_event(+EventMask, -Type, ?XEvent)
mask_event(+Displayable, +EventMask, -Type, ?XEvent)
```

Search the event queue for the given *Displayable* and remove the first event that is selected by the given *EventMask*. If *Displayable* is omitted, the queues for all ProXL displays are

searched. Unifies *Type* with the type of event removed. Other events in the queue are not discarded.

If there is no qualifying event in the queue, `mask_event/[3,4]` flushes the output queue and blocks until one is received. If *Displayable* is specified, and events for which callbacks are registered arrive on other displays, they will be handled, and `mask_event/3` will continue to wait for an event on the specified *Displayable*.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `mask_event/[3,4]` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

EventMask is an integer bitmask, that specifies the selected events. The predicate `event_list_mask/2` is useful to translate between X11 event mask names and bitmasks.

16.12.11 check_mask_event/[3,4]

```
check_mask_event(+EventMask, -Type, ?XEvent)
check_mask_event(+Displayable, +EventMask, -Type, ?XEvent)
```

Are like `mask_event/[3,4]`, but fail if there is no matching event in the queue, and do not block waiting for one. The output buffer is flushed only on failure.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `check_mask_event/[3,4]` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

EventMask is an integer bitmask, that specifies the selected events. The predicate `event_list_mask/2` is useful to translate between X11 event mask names and bitmasks.

16.12.12 check_typed_event/[2,3]

```
check_typed_event(+Type, ?XEvent)
check_typed_event(+Displayable, +Type, ?XEvent)
```

Succeed only if there is an event of the given *Type* in the event queue for the given *Displayable*, which if omitted, is the default Display. Other events in the queue are not discarded. If there is no matching event in the queue, they fail, and do not block waiting for events. The output buffer is flushed only on failure.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `check_typed_event/[2,3]` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

16.12.13 check_typed_window_event/3

```
check_typed_window_event(+Window, +Type, ?XEvent)
```

Succeeds only if there is an event in the event queue for the *Window*'s Display, that is intended for *Window* and is of the given *Type*. Other events in the queue are not discarded. If there is no matching event in the queue, it fails and does not block waiting for events. The output buffer is flushed only on failure.

The user can pass an existing *XEvent* X11 structure, which is then destructively filled with the received event's contents, or an unbound variable. In the latter case, `check_typed_window_event/3` will unify *XEvent* with a private, local structure, which the user can not hang on to, but is guaranteed not to change its value until the next call to an event getting routine.

16.12.14 put_back_event/[1,2]

```
put_back_event(+XEvent)
put_back_event(+Displayable, +XEvent)
```

Pushes back the given *XEvent* event structure at the front of the event queue for the given *Displayable*, which if omitted, is the default Display. No checking of values is performed. *Be careful.*

16.12.15 send_event/[4,5]

```
send_event(+WindowSpec, +Propagate, +EventMask, +XEvent)
send_event(+Displayable, +WindowSpec, +Propagate, +EventMask, +XEvent)
```

Asks the server to send the event *XEvent*, an X11 event structure, to the specified window. Fail if *XEvent* could not be converted into the server format, usually a sign that the event `type` is invalid. The delivered event will have a value of `true` in its `send_event` field.

The argument *WindowSpec* can be:

A valid ProXL Window
: In this case the event will be sent to the Window and its Display is used as the Display connection. If *Displayable* was specified, it must have the same Display as the Window.

`pointer_window`
: In this case the event will be send to the window that the pointer is in. If *Displayable* was not specified, the default Display is used.

`input_focus`
: In this case, if the focus window contains the pointer, the event will be sent to the window that contains the pointer. Otherwise, the event will be sent to the focus window. If *Displayable* was not specified, the default Display is used.

Propagate is a boolean value and *EventMask* an integer bitmask that specifies the selected events.

`send_event/[4,5]` uses the *Propagate* and *EventMask* arguments to determine which clients should receive the specified events as follows:

If *EventMask* is 0
: The event is sent to the client that created the receiving window. If that client no longer exists, no event is sent.

If *Propagate* is `false`
: The event is sent to every client selecting any of the event types from *EventMask* on the receiving window.

If *Propagate* is `true`
: The event propagates up the receiving window hierarchy in the normal way.

The predicate `event_list_mask/2` is useful to translate between X11 event mask names and bitmasks.

No checking on the validity of the event contents is performed.

16.12.16 `send/[4,5]`

```
send(+WindowSpec, +Propagate, +EventMask, +EventValues)
send(+Displayable, +WindowSpec, +Propagate, +EventMask, +EventValues)
```

Are analogous to `send_event/[4,5]`, except that instead of taking an *XEvent* event structure argument, take *EventValues*, a list of event field values in the style of the callback mechanism.

The elements of *EventValues* will be used to fill an X11 event structure to send, and must contain at least a `type` field. If no `display` is specified, the default Display is used.

No checking on the validity of the event contents is performed.

16.12.17 `get_event_values/2`

```
get_event_values(+XEvent, +EventValues)
```

Unifies the elements of *EventValues*, a list of event field values in the style of the callback mechanism, with the contents of the given *XEvent* event structure.

16.12.18 `put_event_values/2`

```
put_event_values(+XEvent, +EventValues)
```

Destructively sets the contents of the given *XEvent* event structure to the values given by the elements of *EventValues*, a list of event field values in the style of the callback mechanism.

EventValues must at least contain a `type` element, and if the `display` is not given, the default Display is used.

No checking on the validity of the event contents is performed.

16.12.19 `get_motion_events/4`

If your server supports a motion history buffer, the predicate:

```
get_motion_events(+Window, +Start, +Stop, -TimeEvents)
```

Unifies *TimeEvents* with a list of terms of the form `time_coord(Time, X, Y)`.

This are all the events in the motion history buffer that fall between the specified timestamps *Start* and *Stop* (inclusive), and have coordinates that lie within the specified *Window*.

The *Start* and *Stop* arguments should be timestamps, in milliseconds, or the constant `current_time`.

16.13 Handling Errors Under ProXL

This section discusses how errors are handled under ProXL, how to set up the action that you want the system to take when an error occurs and how to install your own error handler.

16.13.1 Introduction

Under X11, error messages are usually asynchronous, because of the nature of the network connections and the need to batch requests to improve performance.

The server generates error events that are sent to the Display connection and handled, as soon as they arrive, by one of two error handlers, depending on the severity of the error. It is not possible to register callback routines for error handling, because of the fundamentally different way from other events that errors are handled by under X.

16.13.2 Recoverable Errors

Recoverable errors are handled by the ProXL Error handler and are any of the following types:

`bad_request`
: If the request made to the X Server was invalid.

`bad_value`
: If an integer argument is out of range.

`bad_window`
: If a Window argument is invalid.

`bad_pixmap`
: If a Pixmap argument is invalid.

`bad_atom` If an Atom argument is invalid.

`bad_cursor`
: If a Cursor argument is invalid.

`bad_font` If a Font argument is invalid.

`bad_match`
: If there is an argument mismatch.

`bad_drawable`
: If an argument is not a Window or Pixmap.

`bad_access`
: If the operation can't be performed.

`bad_alloc`
: If there are insufficient resources.

`bad_color`
: If there is no such colormap.

`bad_gc` If a GC argument is invalid.

`bad_id_choice`
: If the choice is not in the appropriate range or is already in use.

`bad_name` If the named Font or Color do not exist.

`bad_length`
: If the request length is incorrect.

`bad_implementation`
: If your server is defective.

ProXL routines do extensive checking of their arguments to try and detect invalid parameters before actually making a call to the X server. However, it is impossible to cover all the cases, and sometimes recoverable errors are signaled by the server.

Recoverable errors invoke a user-definable error handler. Under ProXL, the default error handler provided prints an error message and presents a list of options to the user.

The user can provide her own error handler routine, but it *must* be written in C and follow the guidelines set up for handlers under X11. The user should read the Xlib X11 documentation on errors and error handling before attempting to do this.

16.13.3 Fatal Errors

Fatal errors under X11 include I/O errors or system call errors, such as irrecoverable network problems, attempts to establish a connection to non-existent, or non-accessible server, etc.

Since X automatically exits after detecting a fatal error and gives no real chance of correcting the problem, ProXL uses the default X fatal error handler, which just prints a message and exists.

If you feel you need to change this handler, we suggest that you consult the X11 documentation.

16.13.4 The ProXL Error Handler

The ProXL error handler, if installed, will print a reasonably intelligent error message and then act according to the user-specified options. By default, it presents the following prompt and waits for user input:

```
ProXL Error Handler (h for help)?
```

Th options available are:

```
ProXL Error options:
 c continue - do nothing
 t trace    - debugger will start creeping
 d debug    - debugger will start leaping
 a abort    - cause Prolog abort
 e exit     - irreversible exit from Prolog
 A Abort    - cause Prolog abort and set this mode as the default
 h help     - print this message
```

16.13.5 Error Handling Options

The user can set error handling modes and enter synchronize mode when debugging.

16.13.5.1 `error_action/[2,3]`

Under ProXL, if the ProXL error handler is installed, it is possible to select a per-Display action using:

```
error_action(+Displayable, -Old, +New)
error_action(-Old, +New)
```

If *Displayable* is omitted, the default Display is used.

Old is unified with the previous action for the *Display* and the new value is set from *New*.

The available options for `error_action` are:

`user` If the user is installing her own C error handler.

`xhandler` If the user wants the X11 default error handler.

`continue` If the user wants the ProXL error handler installed in 'continue' mode for the given Display.

`trace` If the user wants the ProXL error handler installed in trace mode for the given Display.

`debug` If the user wants the ProXL error handler installed in debug mode for the given Display.

`abort` If the user wants the ProXL error handler installed in abort mode for the given Display.

`exit` If the user wants the ProXL error handler installed in exit mode for the given Display.

`menu` If the user wants the ProXL error handler installed and the show menu mode set for the given Display.

`default` Same as `menu`.

16.13.5.2 `synchronize/[1,2]`

Because of the asynchronous nature of X, localizing the source of errors can be very difficult. By putting the Display connection in *synchronous* mode, the user is certain that requests are carried out immediately and that errors will be reported as soon as the offending requests are finished. The performance of the ProXL system will be severly degraded when operating under this mode, so it should only be used for debugging purposes.

```
synchronize(+Displayable, +Goal)
synchronize(+Goal)
```

If *Displayable* is omitted, the default Display is used.

When using `synchronize/[1,2]`, the *Displayable* connection is put in synchronous mode while the given *Goal* is executed. Synchronization is turned off when *Goal* finishes, or if execution is aborted.

16.14 Window Manager Functions

Most of the functions provided here are most often used by Window Managers, and not normal applications. Nevertheless, some are useful in other situations.

These functions allow the user to:

- Control the lifetime of a window.
- Grab the pointer.
- Grab the Keyboard.

- Grab the server.
- Control event processing.
- Manipulate the keyboard and pointer settings.
- Control the screen saver.

16.14.1 Controlling the Lifetime of a Window

The save set of a client is a list of other client's windows, which if they are inferiors of one of the client's windows at connection close, should not be destroyed, but reparented.

16.14.1.1 `change_save_set/[2,3]`

```
change_save_set(+Displayable, +Window, +SaveSetMode)
change_save_set(+Window, +SaveSetMode)
```

Adds or deletes the given *Window* from the given *Displayable*'s save set. If *Displayable* is omitted, the default Display is used.

SaveSetMode specifies the action:

`insert` If *Window* should be added to the Display's save set.

`delete` If *Window* should be deleted from the Display's save set.

Window must have been created by another client.

16.14.2 Grabbing the Pointer

Normally mouse events are delivered as soon as they occur to the appropriate window and client, as determined by the window event masks and the input focus. The routines described in this section allow the user to grab the mouse. When a grab is in effect, events are sent to the grabbing client, rather than to the normal client who would have received the event. If the keyboard or pointer is put in asynchronous mode, further mouse and keyboard events will continue to be processed. If the keyboard or pointer is put in synchronous mode, no further events will be processed until the grabbing client allows them (see `allow_events`).

There are two kinds of grabs: active and passive. An active grab occurs when a single client grabs the pointer explicitly. Clients can also grab a particular pointer button in a window, this is called a *passive grab* and activates when the button is actually pressed.

Many of the operations take a *Time* argument, which can be `current_time` (the X server current time) or an actual timestamp in milliseconds.

16.14.2.1 grab_pointer/9

```
grab_pointer(+Window,
        +OwnerEvents, +EventsMask, +PointerMode,
        +KeyboardMode,
        +WindowConfine, +Cursor, +Time,
        -GrabStatus)
```

This predicate actively grabs control of the pointer and generates `enter_notify` and `leave_notify` events as appropriate.

Window is the ProXL window to which events will be reported while the grab is in effect.

OwnerEvents is one of:

`true` If pointer events should be reported normally.

`false` If pointer events should be reported only to the the grab window.

EventsMask is an integer bitmask specifying the events that are selected and should be reported to the client. The valid event mask names are:

- `button_press`
- `button_release`
- `enter_window`
- `leave_window`
- `pointer_motion`
- `pointer_motion_hint`
- `button1_motion`
- `button2_motion`
- `button3_motion`
- `button4_motion`
- `button5_motion`
- `button_motion`
- `keymap_state`

The predicate `event_list_mask/2` is useful to translate between event mask names and integer bitmasks.

PointerMode controls further processing of pointer events:

`async` Pointer event processing continues normally.

`sync` The state of the pointer, as seen by applications, appears to freeze. No further pointer events are generated until the grabbing client calls `allow_events/[1,2,3]` or the pointer grab is released.

KeyboardMode controls further processing of keyboard events:

`async` Keyboard event processing continues normally.

`sync` The state of the keyboard, as seen by applications, appears to freeze. No further keyboard events are generated until the grabbing client calls `allow_events/[1,2,3]` or the keyboard grab is released.

WindowConfine is the ProXL window to which the pointer will be confined while the grab is in effect, or the atom `none`.

Cursor is the ProXL cursor to be displayed during the grab, or the atom `none`.

Time is a timestamp in milliseconds (from an event) or the atom `current_time`.

GrabStatus is unified by `grab_pointer/9` with one of:

`success` If the grab was successful.

`already_grabbed`
If the attempt is unsuccessful because the pointer is already actively grabbed by some other client.

`invalid_time`
If the attempt is unsuccessful because the specified *Time* is earlier than the last-pointer-grab time, or later than the current server time.

`not_viewable`
If the attempt is unsuccessful because either of the grabbing *Window* or the *WindowConfine* is not viewable.

`frozen` If the attempt is unsuccessful because the pointer is frozen by an active grab of another client.

16.14.2.2 `grab_button/9`

```
grab_button(+ButtonGrab, +ModifiersMask,
        +GrabWindow, +OwnerEvents, +EventsMask,
        +PointerMode, +KeyboardMode, +WindowConfine,
        +Cursor)
```

This predicate establishes a passive grab on the pointer, activated when a specified button and set of modifiers are pressed while the pointer is in the grab window.

ButtonGrab is the pointer button to be grabbed, possible values are:

1 to 5 The button number.

`any_button`

ModifiersMask is one of:

An integer A bitmask giving the state of the modifier masks. The predicate `modifiers_mask/2` is useful to translate between modifier specifications and bitmasks.

`any_modifier`

GrabWindow is the ProXL window to which events will be reported while the grab is in effect.

OwnerEvents is one of:

`true` If pointer events should be reported normally.

`false` If pointer events should be reported only to the grab window.

EventsMask is an integer bitmask specifying the events that are selected and should be reported to the client. The valid event mask names are:

- `button_press`
- `button_release`
- `enter_window`
- `leave_window`
- `pointer_motion`
- `pointer_motion_hint`
- `button1_motion`
- `button2_motion`
- `button3_motion`
- `button4_motion`
- `button5_motion`
- `button_motion`
- `keymap_state`

The predicate `event_list_mask/2` is useful to translate between event mask names and integer bitmasks.

PointerMode controls further processing of pointer events:

`async` If pointer event processing continues normally.

`sync` If the state of the pointer, as seen by applications, appears to freeze. No further pointer events are generated until the grabbing client calls `allow_events/[1,2,3]` or the pointer grab is released.

KeyboardMode controls further processing of keyboard events:

`async` If keyboard event processing continues normally.

`sync` If the state of the keyboard, as seen by applications appears to freeze. No further keyboard events are generated until the grabbing client calls `allow_events/[1,2,3]` or the keyboard grab is released.

WindowConfine is the ProXL window to which the pointer will be confined during the grab, or the atom `none`.

Cursor is the ProXL cursor to be displayed during the grab, or the atom `none`.

16.14.2.3 ungrab_button/3

```
ungrab_button(+ButtonUngrab, +ModifiersMask, +UngrabWindow)
```

Releases the passive button/key combination grab on the specified window, if it was grabbed by the client. It has no effect on an active grab.

ButtonUngrab is the pointer button to be released, possible values are:

1 to 5

`any_button`

ModifiersMask is one of:

An integer A bitmask giving the state of the modifier masks. The predicate `modifiers_mask/2` is useful to translate between modifier specifications and bitmasks.

`any_modifier`

UngrabWindow is the ProXL window where the grab is in effect.

16.14.2.4 ungrab_pointer/[0,1,2]

```
ungrab_pointer
ungrab_pointer(+Time)
ungrab_pointer(+Displayable, +Time)
```

Release the pointer and any queued events if this client has it actively grabbed, unless the *Time* specified is earlier that the last-pointer-grab time or later than the current server time. It also generates `focus_in` and `focus_out` events.

Displayable is the ProXL Displayable. If omitted, the default Display is used.

Time is a timestamp in milliseconds (from an event) or `current_time`. If omitted, `current_time` is used.

16.14.2.5 change_active_pointer_grab/[3,4]

```
change_active_pointer_grab(+Displayable, +EventsMask, +Cursor, +Time)
change_active_pointer_grab(+EventsMask, +Cursor, +Time)
```

Modify the specified dynamic parameters of a grab, if the pointer is actively grabbed by the client and the specified *Time* is no earlier than the last-pointer-grab and no later than the current X server time.

Displayable is the ProXL Displayable. If omitted, the default Display is used.

EventsMask is an integer bitmask specifying the events that are selected and should be reported to the client. The valid event mask names are:

- `button_press`
- `button_release`
- `enter_window`
- `leave_window`
- `pointer_motion`
- `pointer_motion_hint`
- `button1_motion`
- `button2_motion`
- `button3_motion`
- `button4_motion`
- `button5_motion`
- `button_motion`
- `keymap_state`

The predicate `event_list_mask/2` is useful to translate between event mask names and integer bitmasks.

Cursor is the ProXL cursor to be displayed during the grab, or the atom `none`.

Time is a timestamp in milliseconds (from an event), or the atom `current_time`.

16.14.3 Grabbing the Keyboard

Usually, keyboard events will be delivered as soon as they occur to the appropriate window and client, which is determined by the window event masks and input focus. With these routines it is possible to grab the keyboard keys; in this case, events will be sent to the grabbing client, rather than the normal client who would have received the event. If the keyboard or pointer is in asynchronous mode, further mouse and keyboard events will continue to be processed. If the keyboard or pointer is in synchronous mode, no further events will be processed until the grabbing client allows them.

There are two kinds of grabs: **active** and **passive**. An active grab occurs when a single client grabs the keyboard explicitly. Clients can also grab a particular keyboard key in a window, this is called a *passive grab* and it activates when the key is actually pressed.

Many of the operations take a *Time* argument, which can be `current_time` (the X server current time) or an actual timestamp in milliseconds.

16.14.3.1 grab_keyboard/6

```
grab_keyboard(+Window, +OwnerEvents,
        +PointerMode, +KeyboardMode,
        +Time, -GrabStatus)
```

Actively grabs control of the keyboard and generates `focus_in` and `focus_out` events.

Window is the ProXL window to which events will be reported while the grab is in effect.

OwnerEvents is one of:

`true` If keyboard events should be reported normally.

`false` If keyboard events should be reported only to the grab window.

PointerMode controls further processing of pointer events:

`async` If pointer event processing continues normally.

`sync` If the state of the pointer, as seen by applications appears to freeze. No further pointer events are generated until the grabbing client calls `allow_events/[1,2,3]` or the pointer grab is released.

KeyboardMode controls further processing of keyboard events:

`async` If keyboard event processing continues normally.

`sync` If the state of the keyboard, as seen by applications appears to freeze. No further keyboard events are generated until the grabbing client calls `allow_events/[1,2,3]` or the keyboard grab is released.

Time is a timestamp in milliseconds or `current_time`.

GrabStatus is unified by `grab_keyboard/6` with one of:

`success` If the grab was successful.

`already_grabbed`
If the attempt is unsuccessful because the keyboard is already actively grabbed by some other client.

`invalid_time`
If the attempt is unsuccessful because the specified *Time* is earlier than the last-keyboard-grab time, or later than the current server time.

`not_viewable`
If the attempt is unsuccessful because the grabbing *Window* is not viewable.

`frozen` If the attempt is unsuccessful because the keyboard is frozen by an active grab of another client.

16.14.3.2 `ungrab_keyboard/[0,1,2]`

```
ungrab_keyboard(+Displayable, +Time)
ungrab_keyboard(+Time)
ungrab_keyboard
```

Release the keyboard and any queued events if this client has it actively grabbed, unless the *Time* specified is earlier that the last-keyboard-grab time or later than the current server time. It also generates `focus_in` and `focus_out` events.

Displayable is the ProXL Displayable. If omitted, the default Display is used.

Time is a timestamp in milliseconds (from an event) or `current_time`. If omitted, `current_time` is used.

16.14.3.3 `grab_key/6`

```
grab_key(+Key, +ModifiersMask, +GrabWindow,
        +OwnerEvents, +PointerMode, +KeyboardMode)
```

Establishes a passive grab on the keyboard, to be activated when the given key and modifiers are pressed while the pointer is in the window.

Key is the key that should be pressed. Possible values are:

An integer Giving the keycode.

A valid key name
As given by `keysym/2`.

`any_key`

ModifiersMask is one of:

An integer A bitmask giving the state of the modifier masks. The predicate `modifiers_mask/2` is useful to translate between modifier specifications and bitmasks.

`any_modifier`

Window is the ProXL window to which events will be reported while the grab is in effect.

OwnerEvents is one of:

`true` If keyboard events should be reported normally.

`false` If keyboard events should be reported only to the grab window.

PointerMode controls further processing of pointer events:

`async` If pointer event processing continues normally.

`sync` If the state of the pointer, as seen by applications appears to freeze. No further pointer events are generated until the grabbing client calls `allow_events/[1,2,3]` or the pointer grab is released.

KeyboardMode controls further processing of keyboard events:

`async` If keyboard event processing continues normally.

`sync` If the state of the keyboard, as seen by applications appears to freeze. No further keyboard events are generated until the grabbing client calls `allow_events/[1,2,3]` or the keyboard grab is released.

16.14.3.4 ungrab_key/3

```
ungrab_key(+Key, +ModifiersMask, +UngrabWindow)
```

Releases the passive grab started by the key combination on the specified window if it was grabbed by this client. Has no effect on an active grab.

Key is the key that should be ungrabbed. One of:

An integer Giving the keycode.

A valid key name
As given by `keysym/2`.

`any_key`

ModifiersMask is one of:

An integer A bitmask giving the state of the modifier masks. The predicate `modifiers_mask/2` is useful to translate between modifier specifications and bitmasks.

`any_modifier`

UngrabWindow is the ProXL window where the grab is in effect.

16.14.3.5 allow_events/[1,2,3]

```
allow_events(+Displayable, +EventMode, +Time)
allow_events(+EventMode, +Time)
allow_events(+EventMode)
```

Release some of the queued events, if the client has caused a device to freeze.

Displayable is a valid ProXL Displayable. If omitted, the default Display is used.

EventMode is one of:

- `async_pointer`
- `sync_pointer`
- `replay_pointer`
- `async_keyboard`
- `sync_keyboard`
- `replay_keyboard`
- `sync_both`
- `async_both`

Time is a timestamp in milliseconds (from an event) or `current_time`. If omitted, `current_time` is used.

16.14.4 Grabbing the Server

Grabbing the server is antisocial, as it does not allow other clients access. The use of this predicates is highly discouraged.

16.14.4.1 `grab_server/[0,1]`

```
grab_server(+Display)
grab_server
```

Allows a client to grab the server, disabling processing of any other requests.

Display is a ProXL Display. If omitted, the default Display is used.

16.14.4.2 `ungrab_server/[0,1]`

```
ungrab_server(+Display)
ungrab_server
```

Release the server to other connections.

Display is a ProXL Display. If omitted, the default Display is used.

16.14.5 Miscellaneous Control Functions

This section discusses how to:

- Move the pointer arbitrarily.
- Control the input focus.
- Kill clients.

16.14.5.1 warp_pointer/8

```
warp_pointer(+SrcWindow, +DestWindow,
        +SrcX, +\SrcY, +SrcW, +SrcH,
        +DestX, +DestY)
```

Moves the pointer to the coordinates specified by *DestX* and *DestY*, relative to *DestWindow* 's origin. It generates events just as if the user had moved the pointer.

SrcWindow is one of:

A ProXL Window
: If the move should only take place if the pointer is currently inside *SrcWindow*, and in a visible portion of the rectangle specified by *SrcX*, *SrcY*, *SrcW*, and *SrcH*.

`none`
: If the move is independent of the current pointer position.

DestWindow is one of:

A ProXL Window
: If the final destination of the move is relative to the origin of this window.

`none`
: If the move is relative to the current position of the pointer.

SrcX, *SrcY*, *SrcW*, *SrcH* are integers that specify the region, if any, inside *SrcWindow* where the pointer must be prior to the move.

DestX, *DestY* are integers giving the coordinates of the final pointer destination.

`warp_pointer/8` allows a lot of freedom for specifying pointer movements, but its use is rarely necessary because `put_pointer_attributes/2` handles the usual cases.

16.14.5.2 set_input_focus/3

```
set_input_focus(+WindowSpec, +RevertTo, +Time)
```

Changes the input focus and the last-focus-change time. It has no effect if the specified *Time* is earlier than the current last-focus-change-time or later than the current X server time. It generates `focus_in` and `focus_out` events.

WindowSpec is one of:

A ProXL Window
: The window that will acquire the input focus.

`pointer_root`
: If the focus window should be dynamically taken to be the root window of whatever screen the pointer is on at each keyboard event.

`none` If all keyboard arguments should be discarded until a new focus window is set.

RevertTo specifies which window that the input focus should revert to, if the current focus becomes not viewable. One of:

- `parent`
- `pointer_root`
- `none`

Time is a timestamp in milliseconds (from an event) or `current_time`.

16.14.5.3 `get_input_focus/[2,3]`

```
get_input_focus(+Displayable, -Focus, -RevertTo)
get_input_focus(-Focus, -RevertTo)
```

Obtain the current input focus window and the revert_to state.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Focus returns the current focus window. One of:

- The ProXL focus window.
- `pointer_root`.
- `none`

RevertTo returns the current focus reverts state. One of :

- `parent`
- `pointer_root`
- `none`

16.14.5.4 `set_close_down_mode/[1,2]`

```
set_close_down_mode(+Displayable, +CloseMode)
set_close_down_mode(+CloseMode)
```

Defines what happens to the client's resources when the connection is closed.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

CloseMode is one of:

- `destroy_all`
- `retain_permanent`
- `retain_temporary`

16.14.5.5 kill_client/[0,1,2]

```
kill_client(+Display, +Resource)
kill_client(+Resource)
kill_client
```

Force a close down of the client that created the associated resource.

Display is a ProXL Display. If omitted, the default Display is used.

Resource is one of:

Any ProXL resource associated with the client.

`all_temporary`
: If required to destroy resources of all clients that have terminated in `retain_temporary` mode.

16.14.6 Pointer Control

ProXL uses the concept of pointer attributes to query and control the state of the pointer.

16.14.6.1 get_pointer_attributes/[1,2]

```
get_pointer_attributes(+AttributeList)
get_pointer_attributes(+Screenable, +AttributeList)
```

Allows the user to find the state of the pointer.

Screenable is a ProXL Screenable. If omitted, the default Screen is used.

AttributeList is a List of pointer attributes, whose elements will be unified with the actual state of the pointer. Valid attributes are:

`acceleration(A)`
: Unifies *A* with a term of the form *N/D*, where *N* and *D* are integers, the **numerator** and **denominator** of the pointer acceleration multiplier.

`threshold(T)`
: Unifies *T* with the integer value of the pointer threshold parameter.

`mapping(B1, B2, B3, B4, B5)`
: Unifies *B1* to *B5* with integers in the range `0` to `5`, denoting the current mapping of physical pointer buttons to logical pointer buttons. The nominal mapping is $Bi = i$. A value of `0`, means that the button is disabled.

`root_position(X, Y)`
: Unifies *X* and *Y* with the current pointer coordinates, relative to the root window origin.

`window(W, X, Y)`
: Unifies *W* with a ProXL window the pointer is inside of, and *X* and *Y* to the pointer coordinates, relative to the window's origin. If *W* is unbound, `get_pointer_attributes` will backtrack over all windows in the hierarchy that contain the pointer, except for the root window.

`deepest(W, X, Y)`
: Unifies *W* with the innermost ProXL window the pointer is inside of, and *X* and *Y* to the pointer coordinates, relative to the window's origin.

`state(B, M)`
: Unifies *B* with a term describing the state of the pointer buttons, and *M* with a term describing the state of the modifier keys. The predicates `state_mask/2`, `buttons_mask/2` and `modifiers_mask/2` are useful for translating between buttons and modifiers representations and bitmasks.

16.14.6.2 put_pointer_attributes/[1,2]

```
put_pointer_attributes(+AttributeList)
put_pointer_attributes(+Screenable, +AttributeList)
```

Allows the user to change the state of the pointer.

Screenable is a ProXL Screenable. If omitted, the default Screen is used.

AttributeList is a List of pointer attributes, whose elements will be used to change the state of the pointer. Valid attributes are:

`acceleration(A)`
: Where *A* is either *N/D*, with *N* and *D* are non-negative integers, or `default`, to restore the pointer default acceleration.

`threshold(T)`
: *T* is either a non-negative integer, or `default`, to restore the pointer default threshold.

`mapping(B1, B2, B3, B4, B5)`
: *B1* to *B5* are non-duplicate integers in the range `1` to `5`, or `0`, denoting the desired mapping of physical pointer buttons to logical pointer buttons. A value of `0` means that the button should be disabled.

`root_position(X, Y)`
: *X* and *Y* are non-negative integers. Has the effect of warping the pointer to the given coordinates, relative to the root window origin.

`window(W, X, Y)`
: Warps the pointer to window *W*, at the position given by the coordinates *X* and *Y*. If *W* is a valid ProXL window, the coordinates are relative to the window's origin. If *W* is `none`, the coordinates are interpreted as offsets relative to the current pointer position.

16.14.7 Keyboard Control

ProXL uses the concept of keyboard attributes to query and control the state of the keyboard.

16.14.7.1 `get_keyboard_attributes/[1,2]`

```
get_keyboard_attributes(+AttributeList)
get_keyboard_attributes(+Displayable, +AttributeList)
```

Allows the user to find the state of the keyboard.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

AttributeList is a List of keyboard attributes, whose elements will be unified with the actual state of the keyboard. Valid attributes are:

`key_click_percent(C)`
: Unifies C with an integer between `0` and `100`, giving the volume for key clicks.

`bell_percent(B)`
: Unifies B with an integer between `0` and `100`, giving the base volume for the bell.

`bell_pitch(P)`
: Unifies P with a non-negative integer that gives the bell pitch, in Hz.

`bell_duration(D)`
: Unifies D with a non-negative integer that gives the duration of the bell, in milliseconds.

`led_mask(L)`
: Unifies L with an integer mask, where each bit set to 1 indicates that the corresponding led is on. The least significant bit of L corresponds to led 1.

`global_auto_repeat(G)`
: Unifies G with one of `on` or `off`.

`auto_repeats(A)`
: Unifies A with a list of bytes that indicate what keyboard keys have auto repeat enabled. The predicate `key_auto_repeat/2` is useful for interpreting the result.

`keymap(K)`
: Unifies K with a list of bytes that indicate the logical state of the keyboard. The predicate `key_state/2` is useful for interpreting the result.

16.14.7.2 `put_keyboard_attributes/[1,2]`

```
put_keyboard_attributes(+AttributeList)
put_keyboard_attributes(+Displayable, +AttributeList)
```

Allows the user to change the state of the keyboard.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

AttributeList is a List of keyboard attributes, whose elements will be used to change the state of the keyboard. Valid attributes are:

`key_click_percent(`*C*`)`
: *C* is either an integer between `0` and `100`, or `default`, to restore the keyboard default.

`bell_percent(`*B*`)`
: *B* is either an integer between `0` and `100`, or `default`, to restore the keyboard default.

`bell_pitch(`*P*`)`
: *P* is either a non-negative integer, or `default`, to restore the keyboard default.

`bell_duration(`*D*`)`
: *D* is either a non-negative integer, or `default`, to restore the keyboard default.

`led(`*N*`)` *N* specifies the led number, an integer between `1` and `32`.

`led_mode(`*M*`)`
: *M* is either `on` or `off`.

`auto_repeat_mode(`*R*`)`
: *R* is either `on`, `off` or `default`.

16.14.7.3 `bell/[1,2]`

```
bell(+Displayable, +Percent)
bell(+Percent)
```

Rings the keyboard bell.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Percent is an integer between `-100` and `100`, specifying the volume, relative to the base volume

16.14.8 Screen Saver Control

16.14.8.1 `set_screen_saver/[4,5]`

```
set_screen_saver(+Displayable, +Timeout, +Interval, +Blanking, +Expo-
sures)
set_screen_saver(+Timeout, +Interval, +Blanking, +Exposures)
```

Modify the screen saver parameters for the given *Displayable*.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Timeout specifies the screen saver timeout value. One of:

An integer Giving the timeout in seconds before the screen saver activates.

`default` To install default value.

`disable` To disable the screen saver.

Interval is An integer giving the interval between screen saver invocations.

Blanking Specifies screen blanking mode. One of:

- `dont_prefer`
- `prefer`
- `default`

Exposures Specifies the screen save control. One of:

- `dont_allow`
- `allow`
- `default`

16.14.8.2 `force_screen_saver/[1,2]`

```
force_screen_saver(+Displayable, +Mode)
force_screen_saver(+Mode)
```

Set the screen saver on or off.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Mode sets The screen saver mode. One of:

`reset` To deactivate.

`active` To activate.

16.14.8.3 `get_screen_saver/[4,5]`

```
get_screen_saver(+Displayable, -Timeout, -Interval, -Blanking, -Expo-
sures)
get_screen_saver(-Timeout, -Interval, -Blanking, -Exposures)
```

Allows the user to check the state of the screen saver.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Timeout is the integer specifying the timeout in seconds before the screen saver activates.

Interval is the integer specifying the interval, in seconds, between screen saver invocations.

Blanking is the current screen blanking preference. One of:

- `dont_prefer`
- `prefer`
- `default`

Exposures is the current screen saver control value. One of:

- `dont_allow`
- `allow`
- `default`

16.15 Utility Functions

This section describes various utility predicates.

16.15.1 Bitmask Handling

16.15.1.1 `state_mask/2`

```
state_mask(?State, ?Mask)
```

Translates between events `state` fields that consist of button and modifier specifications and the corresponding integer bitmasks.

State is unified with a term of the form `state(Buttons, Modifiers)`, where *Buttons* is the buttons specification used by `buttons_mask/2` and *Modifiers* is the modifiers specification used by `modifiers_mask/2`.

Mask is the integer bitmask representing the given set of buttons and modifiers.

16.15.1.2 `buttons_mask/2`

```
buttons_mask(?Buttons, ?Mask)
```

Translates between button specifications and the corresponding integer bitmask.

Buttons is a term of the form `buttons(B1, B2, B3, B4, B5)`, where button number *i* has argument position *i* in the term and the value of each argument is either `up` or `down`.

Mask is the integer bitmask representing the given set of buttons.

16.15.1.3 modifiers_mask/2

```
modifiers_mask(?Modifiers, ?Mask)
```

Translates between modifier specifications and the corresponding integer bitmask.

Modifiers is a term of the form `modifiers(Shift, Control, Lock, Mod1, Mod2, Mod3, Mod4, Mod5)`, where the value of each argument is either `up` or `down`.

Mask is the integer bitmask representing the given set of modifiers.

16.15.1.4 event_list_mask/2

```
event_list_mask(?EventList, ?Mask)
```

Translates between a list of events names and the corresponding integer bitmask.

EventList is a list of event mask names, taken from the following:

- key_press
- key_release
- button_press
- button_release
- enter_window
- leave_window
- pointer_motion
- pointer_motion_hint
- button1_motion
- button2_motion
- button3_motion
- button4_motion
- button5_motion
- button_motion
- keymap_state
- exposure
- visibility_change
- structure_notify
- resize_redirect
- substructure_notify
- substructure_redirect
- focus_change
- property_change
- colormap_change

- `owner_grab_button`

Mask is the corresponding integer bitmask.

16.15.1.5 `bitset_composition/3`

```
bitset_composition(?Mask1, ?Mask2, ?Mask3)
```

Take any of two mutually exclusive bitmasks and produce a third one, such that the arguments obey *Mask3* `is` *Mask1* `\/` *Mask2*.

16.15.2 Key Handling

16.15.2.1 `rebind_key/[3,4]`

```
rebind_key(+Displayable, +Key, +ModifiersList, +Atom)
rebind_key(+Key, +ModifiersList, +Atom)
```

Rebind a key, with a possible set of modifiers, to a new atom.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Key is a valid key name, as given by `keysym/2`.

ModifiersList is a list containing any of the following modifier key names:

- `'Shift_L'`
- `'Shift_R'`
- `'Control_L'`
- `'Control_R'`
- `'Caps_Lock'`
- `'Shift_Lock'`
- `'Meta_L'`
- `'Meta_R'`
- `'Alt_L'`
- `'Alt_R'`
- `'Super_L'`
- `'Super_R'`
- `'Hyper_L'`
- `'Hyper_R'`

Atom is an atom giving the new binding of *Key*

16.15.2.2 key_keycode/[3,4]

```
key_keycode(+Displayable, ?Key, ?Keycode, ?Index)
key_keycode(?Key, ?Keycode, ?Index)
```

Translates between keys and keycodes.

Under X11, physical keys are mapped to unique server-dependent keycodes and keycodes are mapped to a list of server-independent keysyms.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Key is a key name, as given by `keysym/2`.

Keycode is an integer, between `8` and `255`, corresponding to the *Key*.

Index is an integer, typically `0` or `1`, that identifies which of the keysyms attached to *Keycode* corresponds to *Key*. The usual case is that Keysym number `0` corresponds to the lower case variant of the key, and Keysym number `1` to the upper case variant, if it exists.

16.15.2.3 keysym/[1,2]

```
keysym(-KeysymSet)
keysym(?Keysym, ?Key)
```

`keysym/1` is true when *KeysymSet* is an atom giving the name of a pre-loaded keysym. The preloaded keysym sets are:

- `miscellany`
- `latin1`
- `latin2`
- `latin3`
- `latin4`
- `greek`

`keysym/2` is true when *Keysym* is the integer, server-independent keysym for the the key named *Key*, which is an atom. In general, the name of a key is just the atom (quoted, if necessary). See the Xlib documentation on keysyms for a list of the keysyms and key names.

16.15.2.4 is_key/[2,3]

```
is_key(?Type, +Key, -Keysym)
is_key(?Type, +Key)
```

Identify various subclasses of keys and their keysyms.

Type is one of:

`keypad` For keypad keys.

`cursor` For cursor control keys.

`pf` For pf keys.

`function` For function keys.

`misc_function`
For various other function keys.

`modifier` For modifier keys.

Key is the key name.

Keysym is the integer, server-independent, keysym.

16.15.2.5 `key_state/[3,4]`

```
key_state(+Displayable, +Keymap, -Key, -State)
key_state(+Keymap, -Key, -State)
```

Allows the user to find out the state of a key in a keymap.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Keymap is a keymap, typically obtained by `get_keyboard_attributes/N`.

Key is the name of a key.

State is either `up` or `down`.

16.15.2.6 `key_auto_repeat/[3,4]`

```
key_auto_repeat(+Displayable, +AutoRepeatMap,
                -Key, -State)
key_auto_repeat(+AutoRepeatMap, -Key, -State)
```

Allows the user to find out if a key has auto repeat enabled or not.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

AutoRepeatMap is an auto repeat keymap, typically obtained by

`get_keyboard_attributes/N`.

Key is the name of a key.

State is either `on` or `off`.

16.15.3 Application Preferences

16.15.3.1 get_default/[3,4]

```
get_default(+Displayable, +Program, +Option, -Value)
get_default(+Program, +Option, -Value)
```

Provides a simple interface for clients that want to find out about application preferences without using the Resource Manager. Defaults are usually loaded into the *RESOURCE_MANAGER* property on the root window at login. If no such property exists, the resource file '`~/.Xdefaults`' is loaded. Additional defaults are obtained from the filename specified by the environment variable `XENVIRONMENT`.

Displayable is a ProXL Displayable. If omitted, the default Display is used.

Program is an atom specifying the name of the program.

Option is an atom specifying the option name.

Value is an atom giving the preferred value for the option.

16.15.3.2 parse_geometry/5

```
parse_geometry(+Geometry, -X, -Y, -W, -H)
```

Parses standard X11 geometry descriptions that describe size and placement.

Geometry is an atom of the form

```
'=<width>x<height>{+-}<xoffset>{+-}<yoffset>'
```

X is unified with the x coordinate value from the atom.

Y is unified with the y coordinate value from the atom.

W is unified with the width value from the atom.

H is unified with the height value from the atom.

16.15.3.3 geometry/[12,13]

```
geometry(+Displayable, +ScreenNum,
        +Position, +Default, +BW, +FH, +FW,
        +XAdd, +YAdd, -X, -Y, -W, -H)
geometry(+ScreenNum, +Position, +Default,
        +BW, +FH, +FW, +XAdd, +YAdd, -X,
        -Y, -W, -H)
```

Parses a window geometry given an specification geometry, and a default geometry. The arguments are:

Displayable
: a ProXL Displayable. If omitted, the default Display is used.

ScreenNum
: an integer specifying which screen the window is on.

Position
: an atom specifying a possible incomplete geometry description in standard X format. See `parse_geometry/5` for details.

Default
: an atom specifying a complete geometry description in standard X format.

BW
: an integer specifying the border width.

FH
: an integer specifying the font height in pixels.

FW
: an integer specifying the font width in pixels.

XAdd
: an integer specifying additional horizontal padding, in pixels, needed in the window.

YAdd
: an integer specifying additional vertical padding, in pixels, needed in the window.

X
: an integer giving back the x coordinate, or unbound.

Y
: an integer giving back the y coordinate, or unbound.

W
: an integer giving back the width, or unbound.

H
: an integer giving back the height, or unbound.

16.16 ProXL for Xlib speakers

This section documents some of the most important differences between ProXL and Xlib. We assume that you are familiar with Xlib and, have read Section 16.1 [pxl-bas], page 749 and Section 16.2 [pxl-tut], page 757 from this manual.

16.16.1 Naming Conventions

ProXL has many primitives that are very similar (though none that are identical) to Xlib functions. Unfortunately, Xlib's naming conventions are not very convenient in Prolog, so ProXL uses different names. ProXL names are all lowercase, with underscores used to separate words. Also, ProXL names do not have prefixes or suffixes to specify the context they are to be used in; in Prolog, context is sufficient to disambiguate. For example, the line style called `LineDoubleDash` by Xlib is called simply `double_dash` by Prolog, since the context in which it is used will make clear that it is a line style.

To be a little more formal about it, here's a rough algorithm for translating names:

1. Strip off the X prefix from a function or structure name, or the context prefix or suffix from a #defined symbol.
2. Insert an underscore before the last in any sequence of one or more uppercase letters.
3. Make all letters lowercase.

So `XOpenDisplay` would become `open_display`, `NorthWestGravity` would become `north_west`, and `LSBFirst` would become `lsb_first`.

Note that not all X functions have direct ProXL counterparts. In some cases, a single ProXL procedure fills the role of many Xlib functions. For example, ProXL has no `set_fill_rule` procedure, instead you use `put_graphics_attributes/2`. Also, in a few cases, Xlib functions that take boolean arguments have been split into two different ProXL procedures, one for each boolean state.

16.16.2 Arguments

Arguments and argument order to ProXL primitives are not always directly predictable from their Xlib counterparts, but some rules of thumb can be helpful:

1. Almost all Xlib functions take a display as first argument. ProXL doesn't need this argument at all if any X resource (i.e., a server-side resource, like a window or GC or font) appears in the call. So few ProXL primitives take a display argument.
2. All the ProXL primitives that DO require a display have a version that allows the display to be defaulted. Also, several ProXL primitives require a screen as argument, and these, too, allow that argument to be defaulted. See the section on the default screen below for details.
3. Several Xlib functions take a window as argument (or some other X resource) when all they really want to know is a screen. In these cases, the analogous ProXL primitive takes a screen as argument. Note that rule 2 above applies in these cases.

ProXL doesn't always use the same argument order as Xlib. There are a few reasons for different argument order. Firstly, ProXL has defaults for some arguments, so sometimes argument orders are changed to facilitate defaults. ProXL also rearranges arguments to make parallels among families of procedures more obvious. In most cases, however, the argument order for ProXL procedures is the same as their Xlib counterparts.

16.16.3 Data Structures

Xlib has three basic kinds of data structures: documented C structures, undocumented C structures, and XIDs. The documented C structures are used as any C structure would be; the undocumented C structures are accessed through standard macros and functions; and the XIDs are used only as arguments to Xlib functions.

ProXL doesn't have these types, instead it has only *foreign terms* and regular Prolog terms. Foreign terms are Prolog terms that represent X objects. These objects are analogous to undocumented C structures in that their contents are not directly accessible, but it is not necessary to send a message to the server to get the contents.

Foreign terms, at the moment, are simply unary Prolog terms whose functor indicates the type of the foreign term and whose argument specifies the location of the contents of the data structure. Note that this implementation may be changed at any time, so you should not count on it.

One useful property of the current implementation is that the printed representation of a foreign term may be read back in, and will yield the same data structure. This is useful when debugging and when prototyping, since it means you can grab results of a goal you just ran and feed them in as arguments to the next goal you want to run.

All foreign terms representing XIDs contain not only the XID of the X resource, but also the display, and, where appropriate, the screen, on which this XID lives. This means that where Xlib requires both a display and an XID, ProXL only needs the foreign term. Therefore, very few ProXL procedures take a display as argument.

16.16.4 Prolog Terms

Prolog terms represent a few of the documented C structures. Most of the documented C structures are not needed in ProXL, for example the XWindowChanges structure. The documented structures that are represented in ProXL by Prolog terms are XPoint, XSegment, XRectangle, XArc, and XColor. The functor of these terms is the structure name translated by the naming conventions above. The first four kinds of terms have the same arguments as the corresponding C struct members, except that the angles in an arc specification are given in degrees, rather than 64ths of a degree, as in Xlib. Color terms have only the red, green, and blue components, and these are given as numbers between 0 and 1. For example, a 30 by 40 pixel rectangle at location (10, 20) would be represented by the term `rectangle(10,20,30,40)`, and the color magenta would be represented by `color(1,0,1)`. See the section of the ProXL manual on colors for information.

16.16.5 Convenience Functions

Xlib provides many *convenience functions* to make it simpler to change attributes of X resources. ProXL doesn't provide these functions, since the `put_*_attributes/2` procedures are quite simple to use, so there is no need for them in Prolog. For example, Xlib provides separate functions to change each GC attribute, as well as a general function that can change all the attributes of a GC at once; ProXL only provides the procedure to change all attributes at once, since it is easy to use. Leaving out all these unnecessary convenience functions makes ProXL simpler and smaller than Xlib.

16.16.6 Caching

It's also important to know what is *not* in a foreign term. Foreign terms do not contain any information about an X resource that may be changed. The sole exception to this rule is graphics contexts, which are discussed below. For example, a ProXL window may remember its depth (which cannot be changed once the window is created), but it won't remember its width or height, since they can change. This means that it is possible for other programs, running in the same or different address space, to share X resources with ProXL. ProXL will have no problems with window managers moving or resizing windows, since it goes to the server when it wants to know a window's size.

An unfortunate result of this is that not all attributes that can be changed can be gotten. This is because the X protocol does not provide a way to get all information about every X resource. For example, it is possible to set the background of a window, but it is not possible to get it. There aren't many such attributes, and they are documented in the ProXL manual.

16.16.7 Default Screen and Display

ProXL maintains a default screen, which the user may set. It starts out as the default screen on the default display. This default screen is used in many primitives when an optional screen argument is not specified. The display on which the default screen is present is considered to be the default display. The default display is used when an optional display argument is not given. The default display cannot be set directly, it is always determined by the default screen.

16.16.8 Graphics Contexts

ProXL, like Xlib, caches the current state of each graphics context. Therefore, it is possible for a ProXL application to determine the current state of a GC. Therefore, ProXL does not permit its GCs to be shared with foreign code in the same or different address space.

16.16.9 Default GCs

To make simple graphics easier, ProXL associates a default graphics context with each drawable. Thus you may omit the GC argument to all the drawing primitives, in which case the destination drawable's default GC is used. Alternatively, you may specify a GC in drawing primitives and ignore the default GC. You may also replace a drawable's default GC at any time.

The concept of a drawable's GC allows you to forget about GCs as separate entities, and instead think of drawables as having graphics attributes. ProXL manages this for you. So

if you want to draw in a window in green, you can set the window's foreground to green, and then draw. You needn't worry about GCs at all.

16.16.10 Modifying GCs

The primitive `put_graphics_attributes/2` is used to modify GCs. There are two ways to change a GC: it may be modified directly, by calling `put_graphics_attributes/2` with a GC as argument, or through a drawable, by passing a drawable as argument.

16.16.11 Sharing and Cloning of GCs

When you create a drawable without specifying a GC or any GC attributes, the drawable gets the default GC for that depth and screen. This default GC has all the default GC attributes, except that foreground is black and background is white. So in fact, you can often create a drawable and draw into it without worrying about GCs or graphics attributes at all. You are not permitted to modify the default GC.

When a drawable gets the default GC, it is actually sharing that GC with all the other drawables that use the default. And if you specify a GC when creating a drawable, or if you change the GC of a drawable, you are sharing that GC with any other drawables that use that GC. If you should modify a shared GC, all the drawables that use that GC will feel the effect. However, if you modify the graphics attributes of a drawable, only that drawable will feel the effect. This is accomplished by cloning the shared GC, modifying the new copy as specified, and installing the clone as the drawable's GC.

It helps to think of two different levels of use. If you never create a GC, and only modify graphics attributes of drawables, then you can think of the graphics attributes as belonging to drawables directly. The graphics attributes of a drawable will only change when you do it directly. However, if you create GCs, install them in drawables, and modify them directly, then a drawable's graphics attributes can change out from under it.

This can be very useful. You can change many drawables' graphics attributes all at once by changing their shared GC. And you can still change one of these drawables' graphics attributes directly, without changing the others. However, once you do this, that drawable will no longer share GCs with the other, so changes to the shared GC will no longer be felt by that drawable.

16.16.12 Memory Management

Memory management is a complex issue for ProXL. In C, programmers are used to having to keep track of their resources; in Prolog, however, resource reclamation is done by the Prolog garbage collector.

Unfortunately, it's not that simple. The X Window System does not make this possible. For example, there is no way to tell whether or not a given colormap is used by any windows, nor is there a way to tell the X server that a colormap should be reclaimed when no more windows require it. The only function provided by X actually destroys the colormap immediately, and any windows that use that colormap find themselves without any colormap.

Fonts are another matter. Since the only X resource that can refer to a font is a GC, and since ProXL GCs are not shared with other languages or other processes, ProXL can be careful about freeing a font that is relied on by some GC.

There is another difficulty, though: ProXL has no way of knowing whether or not the user's Prolog code is holding onto a font explicitly. For example, the user may assert it into the Prolog database, or pass it around as an argument throughout his program. To Prolog, this font looks just like any other term; it has no way of knowing that it represents a foreign term. This means that the Prolog garbage collector cannot collect foreign terms.

All this means that ProXL has two different mechanisms for freeing resources. Most resources can only be freed by explicit action, and the resource is freed immediately (the next time the output buffer is flushed). GCs and fonts are instead released. *Releasing* an X resource means that (you are proclaiming that) Prolog does not have any references to the resource. When no X resources refer to the released resource, it will be freed. Fonts can only be referred to by GCs. GCs can be referred to by drawables.

16.16.13 Mixed Language Programming

It may be desirable to write programs in which some X graphics is done by Prolog code using ProXL and other parts are done in C or another procedural language using Xlib. ProXL makes this possible by providing a way to translate between ProXL and Xlib resources from both C and Prolog code.

On the C side, there is a function to translate each kind of X resource into the corresponding ProXL object, and a macro to get the X resource from the ProXL object. From Prolog, there is a predicate to translate between X and ProXL displays, another to translate between X and ProXL screens, and a third to translate between XIDs and their ProXL counterparts.

ProXL does not permit ProXL GCs to be used in foreign code, nor will it allow a foreign GC to be used in ProXL. Therefore, none of these translations are available for GCs.

17 The ProXT Package

17.1 Technical Overview and Manual

17.1.1 Introduction

ProXT is a Prolog interface to the Motif widget set and the X Toolkit (Xt). Widgets are ready-made graphical components for building user-interfaces, for example, menus, dialog boxes, scroll bars and command buttons. The X Toolkit provides routines for creating and using such widgets. ProXT provides access from Prolog to all the widgets in Motif and to the Xt routines necessary for using the widgets.

ProXT permits the rapid development of user-interfaces with the Motif "look and feel". The interactive development environment of Prolog greatly shortens the edit-test-debug cycle. Individual files can be recompiled incrementally, avoiding the need to exit the application and re-link it. In addition, ProXT provides the programmer with a more logical view of the Motif widget set and reinforces its object-oriented nature.

Almost all of the predicates in ProXT are interfaced directly to functions from the underlying C interface. Also, there is a straightforward correspondence between the data structures used: for example ProXT expects a list where Xt expects an array. This makes it very easy to switch between writing Prolog code for ProXT and C code for Xt and Motif. In addition, ProXT can readily be used in conjunction with C code written for Motif and Xt.

ProXT is not available under Windows.

17.1.2 Using ProXT

ProXT is very easy to use if you already know Motif and Xt. If you do not already know Motif and Xt, experimentation with ProXT is an excellent way to learn. There is a wide array of books on the market covering X and Motif. O'Reilly & Associates publish *The Definitive Guides to the X Window System*, a series of books in the area of the X Window System. For a tutorial on how to use the Motif widget set refer to *The X Window System: Programming and Applications with Xt, OSF/Motif Edition* (Douglas Young, Prentice-Hall).

ProXT interfaces to Motif2.1 with the X11 Release 6 libraries. The user is expected to use the Motif and X Toolkit manuals when working with ProXT. This document concentrates on providing rules for applying that information to ProXT because the original manuals address the C interface. Section 17.5 [pxt-exp], page 934 lists the predicates in ProXT and the types of their arguments. Section 17.4 [pxt-wid], page 924 lists the widget resources and their types while Section 17.3 [pxt-typ], page 908 lists the data types and their Prolog representation.

17.1.3 Naming Conventions

ProXT preserves the original naming conventions used in the X Toolkit except that the first letter of each name is converted to lower case, since Prolog treats names beginning with an upper case letter as variables. Thus all Xt functions start with the prefix '`xt`', and all Motif functions start with '`xm`'. Motif resource names start with the prefix '`xmN`'.

There is a set of ProXT predicates that do not correspond directly to functions from the underlying C interface. Such predicates start with the prefix '`proxt`'.

17.1.4 Predicate Arguments

The order of the arguments in ProXT predicates is the same as that of the functions interfaced to them. The return value of a function, if it exists, always appears as the last argument of the predicate. Motif functions that return a Boolean value are mapped to ProXT predicates that succeed or fail depending on whether the Motif function returns `True` or `False`.

There is a general class of C functions that take an array and a count of the elements in the array as arguments. In ProXT, arrays are replaced by Prolog lists and the count argument is dropped.

The table in Section 17.5 [pxt-exp], page 934 contains all of the built-in predicates in ProXT with their arguments and types.

17.1.5 Type Matching

The table in Section 17.3 [pxt-typ], page 908 lists the types used in ProXT.

The rule of thumb is that simple types in C are mapped to simple types in Prolog.

In most cases C struct types are represented as handles. A handle is a term whose functor is the type name and whose argument is the address of the structure in memory. The address of a C structure would be of interest to programmers interfacing to foreign code.

Character strings are represented by atoms and arrays are represented by lists.

17.1.6 Widget Resources

The table in Section 17.4 [pxt-wid], page 924 shows the types of the widget resource values in ProXT.

In the ProXT documentation the terms *resource* and *attribute* are used interchangeably. A widget attribute is represented as a term whose functor is the resource name and whose argument is the value of the attribute. This defines the type **Attribute**.

17.1.7 Callbacks

In ProXT, callbacks are specified as Prolog goals. The number of arguments in a goal is the same as the number of arguments in the corresponding type of C callback functions. For example, the widget callback goals are of arity 3 with arguments +*Widget*, +*ClientData* and +*CallData*.

17.1.8 Using ProXT with ProXL

ProXL is a direct interface to the X Window System. You may want to use ProXL in your ProXT programs in order to do graphics or use some of the X Window System's more subtle features not supported directly by the X toolkit.

ProXT provides a library file, `library(xif)`, which allows ProXL predicates to be used smoothly in your ProXT programs. In order to use any of these predicates, you must load `library(xif)`. Quintus distributes the source code for '`xif.pl`' and users are encouraged to look at it for examples of mixed ProXL/ProXT programming.

If you need to use ProXT and ProXL together, make sure your code has:

```
:- use_module(library(proxt)).
:- use_module(library(proxl)).
:- use_module(library(xif)).
```

17.1.8.1 `xif_initialize/3`

```
xif_initialize(+Name, +Class, -Widget)
```

Call this predicate instead of `xtInitialize/3` to initialize Xt when you plan to use ProXL. The arguments and usage are the same as for `xtInitialize/3`.

`xif_initialize/3` sets ProXL's notion of default screen to the same screen ProXT is using. This will ensure that ProXL objects you create can be used with your widgets.

17.1.8.2 `widget_window/2`

```
widget_window(+Widget, -Window)
widget_window(-Widget, +Window)
```

Given a ProXT Widget, return the associated ProXL Window, or given a ProXL Window that is already associated with a widget, return that widget. Note that every Xt widget has

an associated window, so you can supply any widget and get its window, but the reverse is not true.

Once you have a ProXL window, you may use any ProXL drawing or any other operation on that window.

17.1.8.3 widget_to_screen/2

```
widget_to_screen(+Widget, -Screen)
```

Given a ProXT *Widget*, return the ProXL *Screen* on which Widget resides.

17.1.8.4 widget_to_display/2

```
widget_to_display(+Widget, -Display)
```

Given a ProXT *Widget*, return the ProXL *Display* on which Widget resides.

17.1.8.5 xif_main_loop/[0,1,2,3]

```
xif_main_loop
xif_main_loop(+ExitCond)
xif_main_loop(+ExitCond, +Context)
xif_main_loop(+Displayable, +ExitCond, +Context)
```

These handle both ProXT and ProXL events. ProXT and ProXL have their own event handling mechanisms. If you combine ProXL and ProXT callbacks, you must use one of `xif_main_loop/[0,1,2,3]` instead of `xtMainLoop/0`.

These predicates are direct analogues of the ProXL predicates `handle_events/[0,1,2,3]`, and are used in exactly the same way. See the ProXL documentation for more detailed information on the arguments, and how they tie in with ProXL callbacks.

If you do not use ProXL callbacks, you do not need these predicates; you can just call `xtMainLoop/0`. You may mix ProXT and ProXL callbacks freely.

17.2 Tutorial

17.2.1 Introduction

This tutorial provides an introduction to the elementary concepts of programming with ProXT. It incorporates some instruction on the basic ideas behind the X Toolkit (Xt) and the Motif widget set. However, this is not a comprehensive tutorial on Motif and Xt programming. For such refer to *The X Window System: Programming and Applications*

with Xt, OSF/Motif Edition (Douglas Young, Prentice-Hall). Programmers who are already familiar with Motif will find this tutorial useful in translating their knowledge to ProXT. In addition the original Motif and Xt naming conventions are preserved to a great extent, which is intended to facilitate the transition to ProXT.

17.2.2 The ProXT programming model

The ProXT programming model can be summarized best in the following program example, which creates a push button, and maps it on the screen.

```
create_button :-
        xtAppInitialize(App,'Demo',[],Shell),
        xmCreatePushButton(Shell,push_button,
                           [xmNwidth(100), xmNheight(100)],Button),
        xtManageChild(Button),
        xtRealizeWidget(Shell),
        xtAppMainLoop(App).
```

First, the Toolkit must be initialized. The call to `xtAppInitialize/4` returns an application context and a shell widget, which is the root widget of the application. The second argument is the application class, attributes can be specified in the third argument. Then the push button widget is created as a child of the shell widget with the call to `xmCreatePushButton/4`. In order to be visible on the screen a widget must be managed. This is accomplished with the call to `xtManageChild/1`. Note that the shell widget should not be managed. The next step is to realize every widget. This can be achieved by realizing the root widget. `xtRealizeWidget/1` propagates the realization recursively to all of the children. Then the program must enter the event loop by calling `xtAppMainLoop/1` where the server dispatches events for the widgets in this application context and certain actions that occur within the application.

17.2.3 The Motif Widget Set

Motif has a variety of widgets and gadgets, each designed to accomplish a specific set of tasks, either individually or in combination with others. Convenience functions create certain widgets or sets of widgets for a specific purpose. `xmCreatePushButton/4` in the program above is an example of such a procedure.

The widgets are grouped into several classes, depending on the function of the widget. Logically, a widget class consists of the procedures and data associated with all widgets belonging to that class. These procedures and data can be inherited by subclasses. Detailed information on the properties of each widget and the procedures associated with it can be found in books on Motif.

17.2.4 Widget Resources

The widgets are objects and they have resources, also called attributes. These can be set at creation time by passing the list of attributes as an argument to the predicate that creates the widget or in later time by calling `xtSetValues/2`. In the example above the third argument of `xmCreatePushButton/4` is the attribute list and the size of the push button is set initially to 100x100 pixels. If that argument was an empty list then no attributes would have been set. Each attribute is represented as a term whose functor is its resource name and whose argument is its value. In Section 17.4 [pxt-wid], page 924 all of the resources are listed with the types of their values.

There is a method to inspect the values of the attributes. This can be accomplished using the predicate `xtGetValues/2`, which is almost the same as `xtSetValues/2` except that the argument of the attribute term is a variable, which is to be bound as a result of the call with the value of the attribute. For example to inspect the size of the push button from the previous example call:

```
xtGetValues(Button, [xmNwidth(X), xmNheight(Y)]).
```

At last, a word of caution. Not all resources can be set at creation time or set/get at a later time. To check for that look at the Motif and X Toolkit manuals.

17.2.5 Event Handling

Handling of events is what enables the application to respond to input, i.e. to be alive. The X Toolkit and Motif provide means for procedures to be mapped to the occurrence of an event in the application. Actually, there are few different types of events within the X Toolkit paradigm.

17.2.5.1 Widget Callbacks

One type of events corresponds to changes in the widget state. They are known as widget callbacks and each widget has its own set of them. There is a logical connection between the type of the widgets and their callbacks. For example, the push button widget has a callback, which is activated when it is pressed.

There are few alternative ways of registering callback procedures. First, the callbacks are widget resources whose values are the corresponding procedures, therefore they can be set as such at creation. In addition, the X Toolkit provides a way for callbacks to be set explicitly. This can be done with the predicates `xtAddCallback/4` and `xtAddCallbacks/3`.

In `xtAddCallback/4` the first argument is the widget for which the callback is registered, the second is the callback name. The third argument is the name of the procedure to be called and the last argument is some data that the programmer wants to pass to the callback procedure. In `xtAddCallbacks/3` the last argument is a list of terms of type

`callback(`*`Predicate`*`,`*`ClientData`*`)` where *Predicate* is the name of the procedure to be called and *ClientData* is the data that the programmer wants to pass to the callback procedure. If the callback is set as a widget resource then the resource value is of the same term as used in the `xtAddCallbacks/3` list. For example,

```
xmCreatePushButton(Shell,push_button,
  [xmNactivateCallback(callback(pressed,'Hello Quintus!'))])
```

would add a callback to the push button at creation time.

The callback procedures must be of arity 3. The first argument is the widget for which the callback was registered. The second argument is reserved for data that the programmer may want pass to the procedure and the last argument is used by the Motif widgets to pass widget specific data to the application.

Following is an extension of the first example showing how to register a widget callback using `xtAddCallbacks/3`:

```
create_button :-
        xtAppInitialize(App,'Demo',[],Shell),
        xmCreatePushButton(Shell,push_button,
                           [xmNwidth(100), xmNheight(100)],Button),
        xtManageChild(Button),
        xtAddCallback(Button,xmNactivateCallback,pressed,
                      'Hello Quintus!'),
        xtRealizeWidget(Shell),
        xtAppMainLoop(App).

pressed(Widget, ClientData, CallData) :-
        write(Widget), nl,
        write(ClientData), nl,
        write(CallData), nl.
```

To remove callbacks use `xtRemoveCallback`, `xtRemoveCallbacks`, or `xtRemoveAllCallbacks`.

17.2.5.2 Translations

The translation mechanism provides the application developers with a powerful syntax for specifying sequences of events and means to map them to actions. Actually, a level of redirection is introduced at this level. The event sequences are mapped into strings, which are mapped to executable procedures further in the program. This is intended to allow the mapping of events to actions to be specified through the resource database.

To specify translations programmatically the programmer can use one of these predicates: `xtAugmentTranslations/2` or `xtOverrideTranslations/2`. Both of these predicates register a translation table with a particular widget. `xtAugmentTranslations/2` merges the

new translation table with the current one while `xtOverrideTranslations/2` not only does that but also replaces existing translations with entries from the new translation table whenever there is a conflict.

The translation table is a structure that is opaque to the Prolog programmer. One can use the predicate `xtParseTranslationTable/2` to parse the translations string and generate the translation table. The exact syntax of the translations string is explained in the X Toolkit documentation.

The predicate `xtAppAddActions/2` is used to map the action names to executable procedures. The first argument is the application context and the second argument is a list of terms of type **Action**. In ProXT the translation procedures have three arguments — the first argument is the widget where the event occurred, the second argument is the event that invoked the action and the third is a list of parameters specified in the action name.

Following is a reimplementation of the previous example using translations for event handling.

```
create_button :-
        xtAppInitialize(App,'Demo',[],Shell),
        xmCreateLabel(Shell,push_button,
                      [xmNwidth(100), xmNheight(100)],Button),
        xtManageChild(Button),
        xtAppAddActions(App, [action(foo,pressed)]),
        xtParseTranslationTable('<Btn1Down: foo("Hello Quintus")',
                                TranslationTable),
        xtAugmentTranslations(Button,TranslationTable),
        xtRealizeWidget(Shell),
        xtAppMainLoop(App).

pressed(Widget, Event, ParamList) :-
        write(Widget), nl,
        write(Event), nl,
        write(ParamList), nl.
```

Registering translations through the resource database is covered to the section dedicated on the resource database.

17.2.5.3 Accelerators

Accelerators are like translations but they map sequences of events in one widget to actions in another. This mechanism is needed when adding a keyboard interface in the application allowing, for example, a menu to pop up as a result of a key stroke.

Every widget has a **xmNAccelerators** resource, inherited from the Core widget class. The application must then call `xtInstallAccelerators` or `xtInstallAllAccelerators` to

specify which widget will be used as the source of the actions to be invoked, and which will be the source of the events that invoke them.

17.2.5.4 Event Handlers

The X Toolkit allows applications to catch low level X events occurring in a window owned by a widget. This can be accomplished with a call to `xtAddEventHandler/5`. This predicate is exactly like the corresponding X Toolkit function, therefore refer to the Xt manual for more information about it. Note that the event mask is represented as a list of event names in ProXT. There is also the `xtAddRawEventHandler/5` predicate, which is almost the same as `xtAddEventHandler/5` except that it does not affect the widget's event mask. To remove an event handler use `xtAddRawEventHandler/5` or `xtAddRawEventHandler/5` accordingly.

Following is the first example rewritten in order to utilize event handlers.

```
create_button :-
        xtAppInitialize(App,'Demo',[],Shell),
        xmCreatePushButton(Shell,push_button,
                           [xmNwidth(100), xmNheight(100)],Button),
        xtManageChild(Button),
        xtAddEventHandler(Button,[buttonPressMask],false,
                          pressed,'Hello Quintus!'),
        xtRealizeWidget(Shell),
        xtAppMainLoop(App).

pressed(Widget, ClientData, CallData) :-
        write(Widget), nl,
        write(ClientData), nl,
        write(CallData), nl.
```

17.2.5.5 Other Events Types

Many X applications require input from sources other than the X event queue. The X Toolkit provides a way for registering procedures to be invoked when there is input coming from a source such as UNIX file. The call to `xtAppAddInput/6` accomplishes that. The first argument in `xtAppAddInput/6` is the application context, the UNIX file descriptor for the input source is second, the third is the input condition mask and the next two arguments are the predicate and client data. An Id is returned in the final argument, which can be used to unregister the procedure.

Another useful facility in the X Toolkit allows applications to register callback procedures to be invoked when a specified interval has elapsed. The call to `xtAppAddTimeOut/5` accomplishes that. The time interval is measured in milliseconds and the registered procedure must be of arity 2, where the first argument is the client data and the second is the interval

id. Following is an extension to the first example, which sets a timer when the push button is activated. This in turn activates a procedure some 2 seconds later.

```
timer(_ClientData,_IntervalId) :-
       write('Button pressed approximately 2 seconds earlier...'),
       nl.

set_timer(Widget,Interval,_CallData) :-
       xtWidgetToApplicationContext(Widget, App),
       xtAppAddTimeOut(App,Interval,timer,_,_Id).

time :-
       xtAppInitialize(App,'Test',[],Shell),
       xmCreatePushButton(Shell,push_button,
                          [xmNactivateCallback([callback(set_timer,2000)]),
                           xmNwidth(100),
                           xmNheight(100)],
                          Button),
       xtManageChild(Button),
       xtRealizeWidget(Shell),
       xtAppMainLoop(App).
```

The X Toolkit includes another type of callback mechanism, known as a WorkProc, that provides a limited form of background processing. It allows the application to invoke a callback whenever there are no events pending. The application can register WorkProc procedures using the call to `xtAppAddWorkProc/4`.

17.2.5.6 Event Handling Loop

At some point the application must start handling events. The procedure for obtaining these events is `xtAppNextEvent/2` and the procedure for dispatching them to the widgets is `xtDispatchEvent/1`. The call to `xtAppMainLoop/1` starts an endless loop where these two procedures are called one after another. Of course, the programmer can always write her own event loop, specifying some exit conditions, etc. The following example is a slight modification of the first program, which exits the event loop when the button is pressed.

```
:- dynamic exit_loop/1.

exit_loop(no).

exit_callback(_Widget,_CLientData,_CallData) :-
        retract(exit_loop(no)),
        assert(exit_loop(yes)).

create_button :-
        xtAppInitialize(App,'Test',[],Shell),
        xmCreatePushButton(Shell,push_button,
                           [xmNwidth(100), xmNheight(100)],
                           Button),
        xtManageChild(Button),
        xtAddCallback(Button,xmNactivateCallback,exit_callback,_),
        xtRealizeWidget(Shell),
        main_loop(App).

main_loop(App) :-
        (   exit_loop(yes)
        ->  write('Exiting...'), nl
        ;   xtAppNextEvent(App,Event),
            xtDispatchEvent(Event),
            main_loop(App)
        ).
```

17.2.6 Using The Resource Database

As mentioned earlier, most widget resources can be specified through the resource database. According to some authors this is the preferred way of resource setting as opposed to hard wiring in the program. During the initialization of the application the resource manager loads the resource database. Most end-users take advantage of that mechanism in customizing their applications without the need to recompile or know anything about programming. There are few places where the resource information may reside but the usual place is the file '`.Xdefaults`' in the user's home directory. Also, the application may keep a file of the application defaults settings in the directory: '`/usr/lib/X11/app-defaults`', and the filename must match the application name.

More on the syntax of specifying resource values can be found in any book on the X Toolkit. Following is another reimplementation of the first example with the widget resources and translations set through the resource database.

```
button_pressed(_Widget,_Event,[Param]) :-
        write(Param), nl.

create_button :-
        xtAppInitialize(App,'Test',[],Shell),
        xmCreateLabel(Shell,button,[],Button),
        xtManageChild(Button),
        xtAppAddActions(App, [action(foo,button_pressed)]),
        xtRealizeWidget(Shell),
        xtAppMainLoop(App).
```

Entries in the resource file:

```
Test.button.translations: #augment <Btn1Down>:foo("Hello Quintus")
Test.button.labelString:           "Press Here"
Test.button.height:                100
Test.button.width:                 100
```

17.2.7 Interaction with Xlib

In many instances the application will need to use functions from the Xlib layer for graphics or pure flexibility. Most of the the Xlib procedures take as arguments the display, screen, window, or gc structures. Therefore proxt provides the predicates `xtDisplay`, `xtScreen`, `xtWindow`, `xCreateGC` and `xtWindowToWidget` to facilitate the mapping between Xt and Xlib. Also, the predicate `xLoadQueryFont` is provided to supply the font structure needed by Motif font predicates such as `xmFontCreateList`.

17.3 ProXT 3.5 Data Types

Action Represented as `action(`*`ActionName`*`,`*`Predicate`*`)`, where *ActionName* is the action name to be mapped to a Prolog goal whose functor is *Predicate*.

ActionsList
Represented as a list of items of type **Action**.

AnyTerm This can be any Prolog term

Attribute Represented as a term whose functor is the resource name and whose argument is the value of the attribute.

AttributeList
Represented as a list of items of type **Attribute**.

Boolean Represented as atom. Accepted values: `true` or `false`.

CallbackList
Represented as a list of items of type **CallbackTerm**.

CallbackProc
This must be a Prolog predicate.

CallbackTerm
Represented as `callback(`*`Predicate`*`,`*`ClientData`*`)`, where *Predicate* is the name of the callback goal and *ClientData* is the callback client data.

Calldata A term returned in the callback call data.

CalldataFields
A list of the fields and their values from the term returned in the callback call data.

Cardinal Represented as integer.

Char Represented as a single character atom.

ClipboardStatus
Represented as atom. Accepted values:

- `xmClipboardFail`
- `xmClipboardSuccess`
- `xmClipboardTruncate`
- `xmClipboardLocked`
- `xmClipboardBadFormat`
- `xmClipboardNoData`

Colormap Represented as integer.

CopyStatus
Represented as atom. Accepted values:

- `xmCOPY_FAILED`
- `xmCOPY_SUCCEEDED`
- `xmCOPY_TRUNCATED`

Cursor Represented as integer.

Dimension
Represented as integer.

Display Represented as `xtdisplay(`*`Address`*`)`.

EventFields
A list of the fields and their values from the **XEvent** structure.

EventProc
This must be a Prolog predicate.

InputProc This must be a Prolog predicate.

Integer Of course, this is an integer.

IntegerList
This is a list of integers.

KeySym Represented as a single character atom.

KeySymTable
Represented as a list of items of type **KeySym**.

Pixel Represented as integer.

Pixmap Represented as integer.

Position Represented as integer.

Screen Represented as `xtscreen(`*`Address`*`)`.

String This is an atom.

StringTable
Represented as a list of atoms.

Time Represented as integer.

TimerProc
This must be a Prolog predicate.

Visual Represented as integer.

Widget Represented as `widget(`*`Address`*`)`.

WidgetCallbackType
Represented as atom. Accepted values are the resource names for resources of type **CallbackProc**

WidgetClass
Represented as atom. Accepted values:

- `compositeWidgetClass`
- `constraintWidgetClass`
- `coreWidgetClass`
- `widgetClass`
- `objectClass`
- `rectObjClass`
- `shellWidgetClass`
- `overrideShellWidgetClass`
- `wmShellWidgetClass`
- `transientShellWidgetClass`
- `topLevelShellWidgetClass`
- `applicationShellWidgetClass`
- `vendorShellWidgetClass`
- `xmArrowButtonWidgetClass`
- `xmArrowButtonGadgetClass`
- `xmBulletinBoardWidgetClass`
- `xmCascadeButtonWidgetClass`
- `xmCascadeButtonGadgetClass`
- `xmCommandWidgetClass`
- `xmDialogShellWidgetClass`

- xmDropSiteManagerObjectClass
- xmDragIconObjectClass
- xmDropTransferObjectClass
- xmDrawingAreaWidgetClass
- xmDrawnButtonWidgetClass
- xmFileSelectionBoxWidgetClass
- xmFormWidgetClass
- xmFrameWidgetClass
- xmGadgetClass
- xmLabelWidgetClass
- xmLabelGadgetClass
- xmListWidgetClass
- xmMainWindowWidgetClass
- xmManagerWidgetClass
- xmMenuShellWidgetClass
- xmMessageBoxWidgetClass
- xmPanedWindowWidgetClass
- xmPrimitiveWidgetClass
- xmPushButtonWidgetClass
- xmPushButtonGadgetClass
- xmRowColumnWidgetClass
- xmScaleWidgetClass
- xmScrollBarWidgetClass
- xmScrolledWindowWidgetClass
- xmSelectionBoxWidgetClass
- xmSeparatorWidgetClass
- xmSeparatorGadgetClass
- xmTextWidgetClass
- xmTextFieldWidgetClass
- xmToggleButtonWidgetClass
- xmToggleButtonGadgetClass

WidgetList
Represented as a list of items of type **Widget**.

Window Represented as integer.

WorkProc This must be a Prolog predicate.

XAtom Represented as integer.

XAtomList
Represented as a list of items of type **XAtom**.

XEvent Represented as `xtevent(`*`Address`*`)`.

XEventMask
Represented as a list of atoms. Accepted values:

- `keyPressMask`
- `keyReleaseMask`
- `buttonPressMask`
- `buttonReleaseMask`
- `enterWindowMask`
- `leaveWindowMask`
- `pointerMotionMask`
- `pointerMotionHintMask`
- `button1MotionMask`
- `button2MotionMask`
- `button3MotionMask`
- `button4MotionMask`
- `button5MotionMask`
- `buttonMotionMask`
- `keymapStateMask`
- `exposureMask`
- `visibilityChangeMask`
- `structureNotifyMask`
- `resizeRedirectMask`
- `substructureNotifyMask`
- `substructureRedirectMask`
- `focusChangeMask`
- `propertyChangeMask`
- `colormapChangeMask`
- `ownerGrabButtonMask`

XFont Represented as `xtfontstruct(`*`Address`*`)`.

XFontSet Represented as `xtfontset(`*`Address`*`)`.

XGC Represented as `xtgc(`*`Address`*`)`.

XGCMask Represented as a list of atoms. Accepted values:

- `gcFunction`
- `gcPlaneMask`
- `gcForeground`
- `gcBackground`
- `gcLindWidth`
- `gcLineStyle`

- `gcCapStyle`
- `gcJoinStyle`
- `gcFillStyle`
- `gcFillRule`
- `gcTile`
- `gcStipple`
- `gcTileStipXOrigin`
- `gcFont`
- `gcSubwindowMode`
- `gcGraphicsExposures`
- `gcClipXOrigin`
- `gcClipYOrigin`
- `gcClipMask`
- `gcDashOffset`
- `gcDashList`
- `gcArcMode`

XGCValues
Represented as a list of terms whose functor is the name of a **XGCMask** field and whose argument is the field value.

XImage Represented as `xtimage(`*`Address`*`)`.

XRectangle
Represented as a term of the form `rectangle(`*`X`*`,`*`Y`*`,`*`Width`*`,`*`Height`*`)`, where *X*, *Y*, *Width* and *Height* are all integers.

XRectangleList
Represented as a list of items of type **XRectangle**.

XmAlignment
Represented as atom. Accepted values:

- `xmALIGNMENT_BEGINNING`
- `xmALIGNMENT_CENTER`
- `xmALIGNMENT_END`

XmAnimationStyle
Represented at atom. Accepted values:

- `xmDRAG_UNDER_NONE`
- `xmDRAG_UNDER_PIXMAP`
- `xmDRAG_UNDER_SHADOW_IN`
- `xmDRAG_UNDER_SHADOW_OUT`
- `xmDRAG_UNDER_HIGHLIGHT`

XmArrowDirection

Represented as atom. Accepted values:

- `xmARROW_UP`
- `xmARROW_DOWN`
- `xmARROW_LEFT`
- `xmARROW_RIGHT`

XmAttachment

Represented as atom. Accepted values:

- `xmATTACH_NONE`
- `xmATTACH_FORM`
- `xmATTACH_OPPOSITE_FORM`
- `xmATTACH_WIDGET`
- `xmATTACH_OPPOSITE_WIDGET`
- `xmATTACH_POSITION`
- `xmATTACH_SELF`

XmAudibleWarning

Represented as atom. Accepted values:

- `xmNONE`
- `xmBELL`

XmBlendModel

Represented as atom. Accepted values:

- `xmBLEND_ALL`
- `xmBLEND_STATE_SOURCE`
- `xmBLEND_JUST_SOURCE`
- `xmBLEND_NONE`

XmButtonType

Represented as atom. Accepted values:

- `xmPUSHBUTTON`
- `xmTOGGLEBUTTON`
- `xmRADIOBUTTON`
- `xmCASCADEBUTTON`
- `xmSEPARATOR`
- `xmDOUBLE_SEPARATOR`
- `xmTITLE`

XmButtonTypeTable

Represented as a list of items of type **XmButtonType**.

XmChildPlacement

Represented as atom. Accepted values:

- `xmPLACE_TOP`

- `xmPLACE_ABOVE_SELECTION`
- `xmPLACE_BELOW_SELECTION`

XmChildType

Represented as atom. Accepted values:

- `xmDIALOG_NONE`
- `xmDIALOG_APPLY_BUTTON`
- `xmDIALOG_CANCEL_BUTTON`
- `xmDIALOG_DEFAULT_BUTTON`
- `xmDIALOG_OK_BUTTON`
- `xmDIALOG_FILTER_LABEL`
- `xmDIALOG_FILTER_TEXT`
- `xmDIALOG_HELP_BUTTON`
- `xmDIALOG_HISTORY_LIST`
- `xmDIALOG_LIST`
- `xmDIALOG_LIST_LABEL`
- `xmDIALOG_MESSAGE_LABEL`
- `xmDIALOG_SELECTION_LABEL`
- `xmDIALOG_SYMBOL_LABEL`
- `xmDIALOG_TEXT`
- `xmDIALOG_SEPARATOR`
- `xmDIALOG_DIR_LIST`
- `xmDIALOG_DIR_LIST_LABEL`

XmChildVerticalAlignment

Represented as atom. Accepted values:

- `xmALIGNMENT_BASELINE_TOP`
- `xmALIGNMENT_CENTER`
- `xmALIGNMENT_BASELINE_BOTTOM`
- `xmALIGNMENT_WIDGET_TOP`
- `xmALIGNMENT_WIDGET_BOTTOM`

XmClipboardPendingList

Represented as a list of terms of the form `clipboardpending(`*`DataId`*`,`*`PrivateId`*`)`, where *DataId* and *PrivateId* are integers.

XmCommandWindowLocation

Represented as atom. Accepted values:

- `xmCOMMAND_ABOVE_WORKSPACE`
- `xmCOMMAND_BELOW_WORKSPACE`

XmDefaultButtonType

Represented as atom. Accepted values:

- xmDIALOG_NONE
- xmDIALOG_CANCEL_BUTTON
- xmDIALOG_OK_BUTTON
- xmDIALOG_HELP_BUTTON

XmDeleteResponse

Represented as atom. Accepted values:

- xmDESTROY
- xmUNMAP
- xmDO_NOTHING

XmDialogStyle

Represented as atom. Accepted values:

- xmDIALOG_MODELESS xmDIALOG_PRIMARY_APPLICATION_MODAL
- xmDIALOG_FULL_APPLICATION_MODAL
- xmDIALOG_SYSTEM_MODAL

XmDialogType

Represented as atom. Accepted values:

- xmDIALOG_TEMPLATE
- xmDIALOG_ERROR
- xmDIALOG_INFORMATION
- xmDIALOG_MESSAGE
- xmDIALOG_WARNING
- xmDIALOG_WORKING
- xmDIALOG_WORK_AREA
- xmDIALOG_PROMPT
- xmDIALOG_SELECTION
- xmDIALOG_COMMAND
- xmDIALOG_FILE_SELECTION

XmDragAttachment

Represented as atom. Accepted values:

- xmATTACH_NORTH_WEST
- xmATTACH_NORTH
- xmATTACH_NORTH_EAST
- xmATTACH_EAST
- xmATTACH_SOUTH_EAST
- xmATTACH_SOUTH
- xmATTACH_SOUTH_WEST
- xmATTACH_WEST
- xmATTACH_CENTER
- xmATTACH_HOT

XmDragDropOperations

Represented as atom. Accepted values:

- `xmDRAG_NOOP`
- `xmDRAG_MOVE`
- `xmDRAG_COPY`
- `xmDRAG_LINK`

XmDragProtocolStyle

Represented as atom. Accepted values:

- `xmDRAG_NONE`
- `xmDRAG_DROP_ONLY`
- `xmDRAG_PREFER_PREREGISTER`
- `xmDRAG_PREREGISTER`
- `xmDRAG_PREFER_DYNAMIC`
- `xmDRAG_DYNAMIC`
- `xmDRAG_PREFER_RECEIVER`

XmDropSiteActivity

Represented as atom. Accepted values:

- `xmDROP_SITE_ACTIVE`
- `xmDROP_SITE_INACTIVE`

XmDropSiteType

Represented as atom. Accepted values:

- `xmDROP_SITE_SIMPLE`
- `xmDROP_SITE_COMPOSITE`

XmDropTransfers

Represented as list of terms of the form `droptransfer(`*`Widget`*`,`*`Target`*`)`, where *Widget* is of type **Widget** and *Target* is of type **XAtom**.

XmEditMode

Represented as atom. Accepted values:

- `xmMULTI_LINE_EDIT`
- `xmSINGLE_LINE_EDIT`

XmFileTypeMask

Represented as atom. Accepted values:

- `xmFILE_DIRECTORY`
- `xmFILE_REGULAR`
- `xmFILE_ANY_TYPE`

XmFontContext

Represented as `xmfontcontext(`*`Address`*`)`.

XmFontList

Represented as `xmfontlist(`*`Address`*`)`.

XmFontListEntry
Represented as `xmfontlistentry(`*`Address`*`)`.

XmFontListTag
Represented as `char_ptr(`*`Address`*`)` or as the atom `xmFONTLIST_DEFAULT_TAG`

XmFontType
Represented as atom. Accepted values:

- `xmFONT_IS_FONT`
- `xmFONT_IS_FONTSET`

XmHighlightMode
Represented as atom. Accepted values:

- `xmHIGHLIGHT_NORMAL`
- `xmHIGHLIGHT_SELECTED`
- `xmHIGHLIGHT_SECONDARY_SELECTED`

XmIndicatorType
Represented as atom. Accepted values:

- `xmN_OF_MANY`
- `xmONE_OF_MANY`

XmKeyboardFocusPolicy
Represented as atom. Accepted values:

- `xmEXPLICIT`
- `xmPOINTER`

XmLabelType
Represented as atom. Accepted values:

- `xmPIXMAP`
- `xmSTRING`

XmListSizePolicy
Represented as atom. Accepted values:

- `xmVARIABLE`
- `xmCONSTANT`
- `xmRESIZE_IF_POSSIBLE`

XmMultiClick
Represented as atom. Accepted values:

- `xmMULTICLICK_DISCARD`
- `xmMULTICLICK_KEEP`

XmNavigationType
Represented as atom. Accepted values:

- `xmNONE`
- `xmTAB_GROUP`
- `xmSTICKY_TAB_GROUP`

- `xmEXCLUSIVE_TAB_GROUP`
- `xmDYNAMIC_DEFAULT_TAB_GROUP`

XmOrientation
Represented as atom. Accepted values:

- `xmNO_ORIENTATION`
- `xmVERTICAL`
- `xmHORIZONTAL`

XmPacking
Represented as atom. Accepted values:

- `xmNO_PACKING`
- `xmPACK_TIGHT`
- `xmPACK_COLUMN`
- `xmPACK_NONE`

XmPositionIndex
Represented as atom. Accepted values:

- `xmLAST_POSITION`
- `xmFIRST_POSITION`

XmProcessingDirection
Represented as atom. Accepted values:

- `xmMAX_ON_TOP`
- `xmMAX_ON_BOTTOM`
- `xmMAX_ON_LEFT`
- `xmMAX_ON_RIGHT`

XmRepTypeEntry
Represented as `xmreptypeentry(`*`Address`*`)`.

XmRepTypeId
Represented as an integer.

XmRepTypeList
Represented as `xmreptypelist(`*`Address`*`)`.

XmResizePolicy
Represented as atom. Accepted values:

- `xmRESIZE_NONE`
- `xmRESIZE_GROW`
- `xmRESIZE_ANY`

XmRowColumnType
Represented as atom. Accepted values:

- `xmWORK_AREA`
- `xmMENU_BAR`
- `xmMENU_PULLDOWN`

- xmMENU_POPUP
- xmMENU_OPTION

XmScrollBarDisplayPolicy
Represented as atom. Accepted values:

- xmSTATIC
- xmAS_NEEDED

XmScrollBarPlacement
Represented as atom. Accepted values:

- xmBOTTOM_RIGHT
- xmTOP_RIGHT
- xmBOTTOM_LEFT
- xmTOP_LEFT

XmScrollingPolicy
Represented as atom. Accepted values:

- xmAUTOMATIC
- xmAPPLICATION_DEFINED

XmSelectionArray
Represented as list of items of type **XmSelectionType**.

XmSelectionPolicy
Represented as atom. Accepted values:

- xmSINGLE_SELECT
- xmMULTIPLE_SELECT
- xmEXTENDED_SELECT
- xmBROWSE_SELECT

XmSelectionType
Represented as atom. Accepted values:

- xmSELECT_POSITION
- xmSELECT_WHITESPACE
- xmSELECT_WORD
- xmSELECT_LINE
- xmSELECT_ALL
- xmSELECT_PARAGRAPH

XmSeparatorType
Represented as atom. Accepted values:

- xmNO_LINE
- xmSINGLE_LINE
- xmDOUBLE_LINE
- xmSINGLE_DASHED_LINE
- xmDOUBLE_DASHED_LINE

- xmSHADOW_ETCHED_IN
- xmSHADOW_ETCHED_OUT
- xmSHADOW_ETCHED_IN_DASH
- xmSHADOW_ETCHED_OUT_DASH
- xmINVALID_SEPARATOR_TYPE

XmShadowType
Represented as atom. Accepted values:

- xmSHADOW_ETCHED_IN
- xmSHADOW_ETCHED_OUT
- xmSHADOW_IN
- xmSHADOW_OUT

XmStackMode
Represented as atom. Accepted values:

- xmABOVE
- xmBELOW

XmString Represented as xmstring(*Address*).

XmStringCharSet
Represented as xmstringcharset(*Address*) or as an atom with the following accepted values:

- xmFONTLIST_DEFAULT_TAG
- xmSTRING_DEFAULT_CHARSET
- xmSTRING_FALLBACK_CHARSET
- xmSTRING_ISO8859_1

XmStringCharSets
Represented as list of items of type **XmStringCharSet** xmstringcharset(*Address*) or atoms with values:

XmStringComponentType
Represented as atom. Accepted values:

- xmSTRING_COMPONENT_UNKNOWN
- xmSTRING_COMPONENT_CHARSET
- xmSTRING_COMPONENT_TEXT
- xmSTRING_COMPONENT_DIRECTION
- xmSTRING_COMPONENT_SEPARATOR
- xmSTRING_COMPONENT_END
- xmSTRING_COMPOUND_STRING
- xmSTRING_COMPONENT_USER_BEGIN
- xmSTRING_COMPONENT_USER_END

XmStringContext
Represented as `xmstringcontext(`*`Address`*`)`.

XmStringDirection
Represented as atom. Accepted values:

- `xmSTRING_DIRECTION_L_TO_R`
- `xmSTRING_DIRECTION_R_TO_L`
- `xmSTRING_DIRECTION_DEFAULT`

XmStringTable
Represented as a list of items of type **XmString**.

XmTearOffModel
Represented as atom. Accepted values:

- `xmTEAR_OFF_ENABLED`
- `xmTEAR_OFF_DISABLED`

XmTextDirection
Represented as atom. Accepted values:

- `xmTEXT_FORWARD`
- `xmTEXT_BACKWARD`

XmTextPosition
Represented as integer.

XmTextSource
Represented as `xmtextsource(`*`Address`*`)`.

XmTransferStatus
Represented as atom. Accepted values:

- `xmTRANSFER_FAILURE`
- `xmTRANSFER_SUCCESS`

XmTraversalDirection
Represented as atom. Accepted values:

- `xmTRAVERSE_CURRENT`
- `xmTRAVERSE_NEXT`
- `xmTRAVERSE_PREV`
- `xmTRAVERSE_HOME`
- `xmTRAVERSE_NEXT_TAB_GROUP`
- `xmTRAVERSE_PREV_TAB_GROUP`
- `xmTRAVERSE_UP`
- `xmTRAVERSE_DOWN`
- `xmTRAVERSE_LEFT`
- `xmTRAVERSE_RIGHT`

XmUnitType
Represented as atom. Accepted values:

- `xmPIXELS`
- `xm100TH_MILLIMETERS`
- `xm1000TH_INCHES`
- `xm100TH_POINTS`
- `xm100TH_FONT_UNITS`

XmUnpostBehavior
Represented as atom. Accepted values:

- `xmUNPOST`
- `xmUNPOST_AND_REPLAY`

XmValueWcs
Represented as `wchar_ptr(`*`Address`*`)`.

XmVerticalAlignment
Represented as atom. Accepted values:

- `xmALIGNMENT_BASELINE_TOP`
- `xmALIGNMENT_CENTER`
- `xmALIGNMENT_BASELINE_BOTTOM`
- `xmALIGNMENT_CONTENTS_TOP`
- `xmALIGNMENT_CONTENTS_BOTTOM`

XmVisibility
Represented as atom. Accepted values:

- `xmVISIBLILITY_UNOBSCURED`
- `xmVISIBLILITY_PARTIALLY_OBSCURED`
- `xmVISIBLILITY_FULLY_OBSCURED`

XmVisualPolicy
Represented as atom. Accepted values:

- `xmVARIABLE`
- `xmCONSTANT`

XtAccelerators
Represented as `xtaccelerators(`*`Address`*`)`.

XtAppContext
Represented as `app_context(`*`Address`*`)`.

XtCallbackStatus
Represented as atom. Accepted values:

- `xtCallbackNoList`
- `xtCallbackHasNone`
- `xtCallbackHasSome`

XtGrabKind
Represented as atom. Accepted values:

- `xtGrabNone`

- `xtGrabNonexclusive`
- `xtGrabExclusive`

XtInputCondMask
Represented as a list of atoms. Accepted values:

- `xtInputReadMask`
- `xtInputWriteMask`
- `xtInputExceptMask`

XtInputId Represented as integer.

XtInputMask
Represented as a list of atoms. Accepted values:

- `xtIMXEvent`
- `xtIMTimer`
- `xtIMAlternateInput`
- `xtIMAll`

XtIntervalId
Represented as integer.

XtPointer Represented as `xtpointer(`*`Address`*`)`.

XtTranslations
Represented as `xttranslations(`*`Address`*`)`.

XtWorkProcId
Represented as integer.

17.4 ProXT 3.5 Widget Resource Data Types

`xmNaccelerator`:	**String**
`xmNacceleratorText`:	**XmString**
`xmNaccelerators`:	**XtAccelerators**
`xmNactivateCallback`:	**CallbackTerm**
`xmNadjustLast`:	**Boolean**
`xmNadjustMargin`:	**Boolean**
`xmNalignment`:	**XmAlignment**
`xmNallowOverlap`:	**Boolean**
`xmNallowResize`:	**Boolean**
`xmNallowShellResize`:	**Boolean**
`xmNancestorSensitive`:	**Boolean**
`xmNanimationMask`:	**Pixmap**
`xmNanimationPixmap`:	**Pixmap**
`xmNanimationPixmapDepth`:	**Integer**
`xmNanimationStyle`:	**XmAnimationStyle**

xmNapplyCallback:	**CallbackTerm**
xmNapplyLabelString:	**XmString**
xmNargc:	**Integer**
xmNargv:	**StringTable**
xmNarmCallback:	**CallbackTerm**
xmNarmColor:	**Pixel**
xmNarmPixmap:	**Pixmap**
xmNarrowDirection:	**XmArrowDirection**
xmNattachment:	**XmAttachment**
xmNaudibleWarning:	**XmAudibleWarning**
xmNautoShowCursorPosition:	**Boolean**
xmNautoUnmanage:	**Boolean**
xmNautomaticSelection:	**Boolean**
xmNbackground:	**Pixel**
xmNbackgroundPixmap:	**Pixmap**
xmNbaseHeight:	**Integer**
xmNbaseWidth:	**Integer**
xmNblendModel:	**XmBlendModel**
xmNblinkRate:	**Integer**
xmNborderColor:	**Pixel**
xmNborderPixmap:	**Pixmap**
xmNborderWidth:	**Dimension**
xmNbottomAttachment:	**XmAttachment**
xmNbottomOffset:	**Integer**
xmNbottomPosition:	**Integer**
xmNbottomShadowColor:	**Pixel**
xmNbottomShadowPixmap:	**Pixmap**
xmNbottomWidget:	**Widget**
xmNbrowseSelectionCallback:	**CallbackTerm**
xmNbuttonAcceleratorText:	**XmStringTable**
xmNbuttonAccelerators:	**StringTable**
xmNbuttonCount:	**Integer**
xmNbuttonFontList:	**XmFontList**
xmNbuttonMnemonicCharSets:	**XmStringCharSets**
xmNbuttonMnemonics:	**KeySymTable**
xmNbuttonSet:	**Integer**
xmNbuttonType:	**XmButtonTypeTable**
xmNbuttons:	**XmStringTable**
xmNcancelButton:	**Widget**
xmNcancelCallback:	**CallbackTerm**
xmNcancelLabelString:	**XmString**
xmNcascadePixmap:	**Pixmap**
xmNcascadingCallback:	**CallbackTerm**
xmNchildHorizontalAlignment:	**XmAlignment**
xmNchildHorizontalSpacing:	**Dimension**
xmNchildPlacement:	**XmChildPlacement**
xmNchildType:	**XmChildType**

xmNchildVerticalAlignment:	**XmChildVerticalAlignment**
xmNchildren:	**WidgetList**
xmNclientData:	**Widget**
xmNclipWindow:	**Widget**
xmNcolormap:	**Colormap**
xmNcolumns:	**Integer**
xmNcommand:	**XmString**
xmNcommandChangedCallback:	**CallbackTerm**
xmNcommandEnteredCallback:	**CallbackTerm**
xmNcommandWindow:	**Widget**
xmNcommandWindowLocation:	**XmCommandWindowLocation**
xmNconvertProc:	**CallbackProc**
xmNcreatePopupChildProc:	**CallbackProc**
xmNcursorBackground:	**Pixel**
xmNcursorForeground:	**Pixel**
xmNcursorPosition:	**XmTextPosition**
xmNcursorPositionVisible:	**Boolean**
xmNdarkThreshold:	**Integer**
xmNdecimalPoints:	**Integer**
xmNdecrementCallback:	**CallbackTerm**
xmNdefaultActionCallback:	**CallbackTerm**
xmNdefaultButton:	**Widget**
xmNdefaultButtonShadowThickness:	**Dimension**
xmNdefaultButtonType:	**XmDefaultButtonType**
xmNdefaultCopyCursorIcon:	**Widget**
xmNdefaultFontList:	**XmFontList**
xmNdefaultInvalidCursorIcon:	**Widget**
xmNdefaultLinkCursorIcon:	**Widget**
xmNdefaultMoveCursorIcon:	**Widget**
xmNdefaultNoneCursorIcon:	**Widget**
xmNdefaultPosition:	**Boolean**
xmNdefaultSourceCursorIcon:	**Widget**
xmNdefaultValidCursorIcon:	**Widget**
xmNdeleteResponse:	**XmDeleteResponse**
xmNdepth:	**Integer**
xmNdestroyCallback:	**CallbackTerm**
xmNdialogStyle:	**XmDialogStyle**
xmNdialogTitle:	**XmString**
xmNdialogType:	**XmDialogType**
xmNdirListItemCount:	**Integer**
xmNdirListItems:	**XmStringTable**
xmNdirListLabelString:	**XmString**
xmNdirMask:	**XmString**
xmNdirSearchProc:	**CallbackProc**
xmNdirSpec:	**XmString**
xmNdirectory:	**XmString**
xmNdirectoryValid:	**Boolean**

xmNdisarmCallback:	**CallbackTerm**
xmNdoubleClickInterval:	**Integer**
xmNdragCallback:	**CallbackTerm**
xmNdragDropFinishCallback:	**CallbackTerm**
xmNdragInitiatorProtocolStyle:	**XmDragProtocolStyle**
xmNdragMotionCallback:	**CallbackTerm**
xmNdragOperations:	**XmDragDropOperations**
xmNdragProc:	**CallbackProc**
xmNdragReceiverProtocolStyle:	**XmDragProtocolStyle**
xmNdropFinishCallback:	**CallbackTerm**
xmNdropProc:	**CallbackProc**
xmNdropRectangles:	**XRectangleList**
xmNdropSiteActivity:	**XmDropSiteActivity**
xmNdropSiteEnterCallback:	**CallbackTerm**
xmNdropSiteLeaveCallback:	**CallbackTerm**
xmNdropSiteOperations:	**XmDragDropOperations**
xmNdropSiteType:	**XmDropSiteType**
xmNdropStartCallback:	**CallbackTerm**
xmNdropTransfers:	**XmDropTransfers**
xmNeditMode:	**XmEditMode**
xmNeditable:	**Boolean**
xmNentryAlignment:	**XmAlignment**
xmNentryBorder:	**Dimension**
xmNentryCallback:	**CallbackTerm**
xmNentryClass:	**WidgetClass**
xmNentryVerticalAlignment:	**XmVerticalAlignment**
xmNexportTargets:	**XAtomList**
xmNexposeCallback:	**CallbackTerm**
xmNextendedSelectionCallback:	**CallbackTerm**
xmNfileListItemCount:	**Integer**
xmNfileListItems:	**XmStringTable**
xmNfileListLabelString:	**XmString**
xmNfileSearchProc:	**CallbackProc**
xmNfileTypeMask:	**XmFileTypeMask**
xmNfillOnArm:	**Boolean**
xmNfillOnSelect:	**Boolean**
xmNfilterLabelString:	**XmString**
xmNfocusCallback:	**CallbackTerm**
xmNfont:	**XFont**
xmNfontList:	**XmFontList**
xmNforeground:	**Pixel**
xmNforegroundThreshold:	**Integer**
xmNfractionBase:	**Integer**
xmNgainPrimaryCallback:	**CallbackTerm**
xmNgeometry:	**String**
xmNheight:	**Dimension**
xmNheightInc:	**Integer**

xmNhelpCallback:	**CallbackTerm**
xmNhelpLabelString:	**XmString**
xmNhighlightColor:	**Pixel**
xmNhighlightOnEnter:	**Boolean**
xmNhighlightPixmap:	**Pixmap**
xmNhighlightThickness:	**Dimension**
xmNhistoryItemCount:	**Integer**
xmNhistoryItems:	**XmStringTable**
xmNhistoryMaxItems:	**Integer**
xmNhistoryVisibleItemCount:	**Integer**
xmNhorizontalFontUnit:	**Integer**
xmNhorizontalScrollBar:	**Widget**
xmNhorizontalSpacing:	**Dimension**
xmNhotX:	**Position**
xmNhotY:	**Position**
xmNiconMask:	**Pixmap**
xmNiconName:	**String**
xmNiconNameEncoding:	**XAtom**
xmNiconPixmap:	**Pixmap**
xmNiconWindow:	**Window**
xmNiconX:	**Integer**
xmNiconY:	**Integer**
xmNiconic:	**Boolean**
xmNimportTargets:	**XAtomList**
xmNincrement:	**Integer**
xmNincrementCallback:	**CallbackTerm**
xmNincremental:	**Boolean**
xmNindicatorOn:	**Boolean**
xmNindicatorSize:	**Dimension**
xmNindicatorType:	**XmIndicatorType**
xmNinitialDelay:	**Integer**
xmNinitialFocus:	**Widget**
xmNinitialResourcesPersistent:	**Boolean**
xmNinitialState:	**Integer**
xmNinput:	**Boolean**
xmNinputCallback:	**CallbackTerm**
xmNinputMethod:	**String**
xmNinsertPosition:	**CallbackProc**
xmNinvalidCursorForeground:	**Pixel**
xmNisAligned:	**Boolean**
xmNisHomogeneous:	**Boolean**
xmNitemCount:	**Integer**
xmNitems:	**XmStringTable**
xmNkeyboardFocusPolicy:	**XmKeyboardFocusPolicy**
xmNlabelFontList:	**XmFontList**
xmNlabelInsensitivePixmap:	**Pixmap**
xmNlabelPixmap:	**Pixmap**

xmNlabelString:	**XmString**
xmNlabelType:	**XmLabelType**
xmNleftAttachment:	**XmAttachment**
xmNleftOffset:	**Integer**
xmNleftPosition:	**Integer**
xmNleftWidget:	**Widget**
xmNlightThreshold:	**Integer**
xmNlistItemCount:	**Integer**
xmNlistItems:	**XmStringTable**
xmNlistLabelString:	**XmString**
xmNlistMarginHeight:	**Dimension**
xmNlistMarginWidth:	**Dimension**
xmNlistSizePolicy:	**XmListSizePolicy**
xmNlistSpacing:	**Dimension**
xmNlistUpdated:	**Boolean**
xmNlistVisibleItemCount:	**Integer**
xmNlosePrimaryCallback:	**CallbackTerm**
xmNlosingFocusCallback:	**CallbackTerm**
xmNmainWindowMarginHeight:	**Dimension**
xmNmainWindowMarginWidth:	**Dimension**
xmNmapCallback:	**CallbackTerm**
xmNmappedWhenManaged:	**Boolean**
xmNmappingDelay:	**Integer**
xmNmargin:	**Dimension**
xmNmarginBottom:	**Dimension**
xmNmarginHeight:	**Dimension**
xmNmarginLeft:	**Dimension**
xmNmarginRight:	**Dimension**
xmNmarginTop:	**Dimension**
xmNmarginWidth:	**Dimension**
xmNmask:	**Pixmap**
xmNmaxAspectX:	**Integer**
xmNmaxAspectY:	**Integer**
xmNmaxHeight:	**Integer**
xmNmaxLength:	**Integer**
xmNmaxWidth:	**Integer**
xmNmaximum:	**Integer**
xmNmenuAccelerator:	**String**
xmNmenuBar:	**Widget**
xmNmenuCursor:	**String**
xmNmenuHelpWidget:	**Widget**
xmNmenuHistory:	**Widget**
xmNmenuPost:	**String**
xmNmessageAlignment:	**XmAlignment**
xmNmessageString:	**XmString**
xmNmessageWindow:	**Widget**
xmNminAspectX:	**Integer**

xmNminAspectY:	**Integer**
xmNminHeight:	**Integer**
xmNminWidth:	**Integer**
xmNminimizeButtons:	**Boolean**
xmNminimum:	**Integer**
xmNmnemonic:	**KeySym**
xmNmnemonicCharSet:	**XmFontListTag**
xmNmodifyVerifyCallback:	**CallbackTerm**
xmNmodifyVerifyCallbackWcs:	**CallbackTerm**
xmNmotionVerifyCallback:	**CallbackTerm**
xmNmoveOpaque:	**Boolean**
xmNmultiClick:	**XmMultiClick**
xmNmultipleSelectionCallback:	**CallbackTerm**
xmNmustMatch:	**Boolean**
xmNmwmDecorations:	**Integer**
xmNmwmFunctions:	**Integer**
xmNmwmInputMode:	**Integer**
xmNmwmMenu:	**String**
xmNnavigationType:	**XmNavigationType**
xmNnoMatchCallback:	**CallbackTerm**
xmNnoMatchString:	**XmString**
xmNnoResize:	**Boolean**
xmNnoneCursorForeground:	**Pixel**
xmNnumChildren:	**Cardinal**
xmNnumColumns:	**Integer**
xmNnumDropRectangles:	**Cardinal**
xmNnumDropTransfers:	**Cardinal**
xmNnumExportTargets:	**Cardinal**
xmNnumImportTargets:	**Cardinal**
xmNoffsetX:	**Position**
xmNoffsetY:	**Position**
xmNokCallback:	**CallbackTerm**
xmNokLabelString:	**XmString**
xmNoperationChangedCallback:	**CallbackTerm**
xmNoperationCursorIcon:	**Widget**
xmNoptionLabel:	**XmString**
xmNoptionMnemonic:	**KeySym**
xmNorientation:	**XmOrientation**
xmNoverrideRedirect:	**Boolean**
xmNpacking:	**XmPacking**
xmNpageDecrementCallback:	**CallbackTerm**
xmNpageIncrement:	**Integer**
xmNpageIncrementCallback:	**CallbackTerm**
xmNpaneMaximum:	**Dimension**
xmNpaneMinimum:	**Dimension**
xmNpattern:	**XmString**
xmNpendingDelete:	**Boolean**

xmNpixmap:	**Pixmap**
xmNpopdownCallback:	**CallbackTerm**
xmNpopupCallback:	**CallbackTerm**
xmNpopupEnabled:	**Boolean**
xmNpositionIndex:	**XmPositionIndex**
xmNpostFromButton:	**Integer**
xmNpreeditType:	**String**
xmNprocessingDirection:	**XmProcessingDirection**
xmNpromptString:	**XmString**
xmNpushButtonEnabled:	**Boolean**
xmNqualifySearchDataProc:	**CallbackProc**
xmNradioAlwaysOne:	**Boolean**
xmNradioBehavior:	**Boolean**
xmNrecomputeSize:	**Boolean**
xmNrefigureMode:	**Boolean**
xmNrepeatDelay:	**Integer**
xmNresizable:	**Boolean**
xmNresizeCallback:	**CallbackTerm**
xmNresizeHeight:	**Boolean**
xmNresizePolicy:	**XmResizePolicy**
xmNresizeWidth:	**Boolean**
xmNrightAttachment:	**XmAttachment**
xmNrightOffset:	**Integer**
xmNrightPosition:	**Integer**
xmNrightWidget:	**Widget**
xmNrowColumnType:	**XmRowColumnType**
xmNrows:	**Integer**
xmNrubberPositioning:	**Boolean**
xmNsashHeight:	**Dimension**
xmNsashIndent:	**Position**
xmNsashShadowThickness:	**Dimension**
xmNsashWidth:	**Dimension**
xmNsaveUnder:	**Boolean**
xmNscaleHeight:	**Dimension**
xmNscaleMultiple:	**Integer**
xmNscaleWidth:	**Dimension**
xmNscreen:	**Screen**
xmNscrollBarDisplayPolicy:	**XmScrollBarDisplayPolicy**
xmNscrollBarPlacement:	**XmScrollBarPlacement**
xmNscrollHorizontal:	**Boolean**
xmNscrollLeftSide:	**Boolean**
xmNscrollTopSide:	**Boolean**
xmNscrollVertical:	**Boolean**
xmNscrolledWindowMarginHeight:	**Dimension**
xmNscrolledWindowMarginWidth:	**Dimension**
xmNscrollingPolicy:	**XmScrollingPolicy**
xmNselectColor:	**Pixel**

xmNselectInsensitivePixmap:	**Pixmap**
xmNselectPixmap:	**Pixmap**
xmNselectThreshold:	**Integer**
xmNselectedItemCount:	**Integer**
xmNselectedItems:	**XmStringTable**
xmNselectionArray:	**XmSelectionArray**
xmNselectionArrayCount:	**Integer**
xmNselectionLabelString:	**XmString**
xmNselectionPolicy:	**XmSelectionPolicy**
xmNsensitive:	**Boolean**
xmNseparatorOn:	**Boolean**
xmNseparatorType:	**XmSeparatorType**
xmNset:	**Boolean**
xmNshadow:	**XmShadowType**
xmNshadowThickness:	**Dimension**
xmNshadowType:	**XmShadowType**
xmNshellUnitType:	**XmUnitType**
xmNshowArrows:	**Boolean**
xmNshowAsDefault:	**Dimension**
xmNshowSeparator:	**Boolean**
xmNshowValue:	**Boolean**
xmNsimpleCallback:	**CallbackProc**
xmNsingleSelectionCallback:	**CallbackTerm**
xmNskipAdjust:	**Boolean**
xmNsliderSize:	**Integer**
xmNsource:	**XmTextSource**
xmNsourceCursorIcon:	**Widget**
xmNsourcePixmapIcon:	**Widget**
xmNspacing:	**Dimension**
xmNstateCursorIcon:	**Widget**
xmNstringDirection:	**XmStringDirection**
xmNsubMenuId:	**Widget**
xmNsymbolPixmap:	**Pixmap**
xmNtearOffMenuActivateCallback:	**CallbackTerm**
xmNtearOffMenuDeactivateCallback:	**CallbackTerm**
xmNtearOffModel:	**XmTearOffModel**
xmNtextAccelerators:	**XtAccelerators**
xmNtextColumns:	**Integer**
xmNtextFontList:	**XmFontList**
xmNtextString:	**XmString**
xmNtextTranslations:	**XtTranslations**
xmNtitle:	**String**
xmNtitleEncoding:	**XAtom**
xmNtitleString:	**XmString**
xmNtoBottomCallback:	**CallbackTerm**
xmNtoPositionCallback:	**CallbackTerm**
xmNtoTopCallback:	**CallbackTerm**

xmNtopAttachment:	**XmAttachment**
xmNtopCharacter:	**XmTextPosition**
xmNtopItemPosition:	**Integer**
xmNtopLevelEnterCallback:	**CallbackTerm**
xmNtopLevelLeaveCallback:	**CallbackTerm**
xmNtopOffset:	**Integer**
xmNtopPosition:	**Integer**
xmNtopShadowColor:	**Pixel**
xmNtopShadowPixmap:	**Pixmap**
xmNtopWidget:	**Widget**
xmNtransferProc:	**CallbackProc**
xmNtransferStatus:	**XmTransferStatus**
xmNtransient:	**Boolean**
xmNtransientFor:	**Widget**
xmNtranslations:	**XtTranslations**
xmNtraversalOn:	**Boolean**
xmNtraverseObscuredCallback:	**CallbackTerm**
xmNtroughColor:	**Pixel**
xmNunitType:	**XmUnitType**
xmNunmapCallback:	**CallbackTerm**
xmNunpostBehavior:	**XmUnpostBehavior**
xmNuseAsyncGeometry:	**Boolean**
xmNuserData:	**XtPointer**
xmNvalidCursorForeground:	**Pixel**
xmNvalue:	**Integer**
xmNvalue:	**Integer**
xmNvalue:	**String**
xmNvalue:	**String**
xmNvalueChangedCallback:	**CallbackTerm**
xmNvalueWcs:	**XmValueWcs**
xmNverifyBell:	**Boolean**
xmNverticalFontUnit:	**Integer**
xmNverticalScrollBar:	**Widget**
xmNverticalSpacing:	**Dimension**
xmNvisibleItemCount:	**Integer**
xmNvisibleWhenOff:	**Boolean**
xmNvisual:	**Visual**
xmNvisualPolicy:	**XmVisualPolicy**
xmNwaitForWm:	**Boolean**
xmNwhichButton:	**Integer**
xmNwidth:	**Dimension**
xmNwidthInc:	**Integer**
xmNwinGravity:	**Integer**
xmNwindowGroup:	**Window**
xmNwmTimeout:	**Integer**
xmNwordWrap:	**Boolean**
xmNworkWindow:	**Widget**

xmNx: **Position**
xmNy: **Position**

17.5 ProXT 3.5 Exported Predicates

17.5.1 Motif Predicates

xmActivateProtocol(+*Shell*,+*Property*,+*Protocol*)
Shell: **Widget**
Property: **XAtom**
Protocol: **XAtom**

xmActivateWMProtocol(+*Shell*,+*Protocol*)
Shell: **Widget**
Protocol: **XAtom**

xmAddProtocolCallback(+*Shell*,+*Property*,+*Protocol*,+*Callback*,+*ClientData*
)
Shell: **Widget**
Property: **XAtom**
Protocol: **XAtom**
Callback: **CallbackProc**
ClientData: **AnyTerm**

xmAddProtocols(+*Shell*,+*Property*,+*Protocols*)
Shell: **Widget**
Property: **XAtom**
Protocols: **XAtomList**

xmAddTabGroup(+*TabGroup*)
TabGroup: **Widget**

xmAddWMProtocolCallback(+*Shell*,+*Protocol*,+*Callback*,+*ClientData*)
Shell: **Widget**
Protocol: **XAtom**
Callback: **CallbackProc**
ClientData: **AnyTerm**

xmAddWMProtocols(+*Shell*,+*Protocols*)
Shell: **Widget**
Protocol: **XAtomList**

xmCascadeButtonGadgetHighlight(+*CascadeButtonGadget*, +*Highlight*)
CascadeButtonGadget: **Widget**
Highlight: **Boolean**

```
xmCascadeButtonHighlight(+CascadeButton, +Highlight)
CascadeButton:      Widget
Highlight:  Boolean

xmChangeColor(+Widget, +Background)
CascadeButton:      Widget
Background: Pixel

xmClipboardCancelCopy(+Display,+Window,+ItemId)
Display:    Display
Window:     Window
ItemId:     Integer

xmClipboardCopy(+Display,+Window,+ItemId,+FormatName,+Buffer,+Length,
          +PrivateId,+DataId, -ReturnValue)
Display:    Display
Window:     Window
ItemId:     Integer
FormatName: String
Buffer:     String
Length:     Integer
PrivateId:  Integer
DataId:     Integer
ReturnValue:        ClipboardStatus

xmClipboardCopyByName(+Display,+Window,+DataId,+Buffer,+Length,+PrivateId,
         -ReturnValue)
Display:    Display
Window:     Window
DataId:     Integer
Buffer:     String
Length:     Integer
PrivateId:  Integer
ReturnValue:        ClipboardStatus

xmClipboardEndCopy(+Display,+Window,+ItemId,-ReturnValue)
Display:    Display
Window:     Window
ItemId:     Integer
ReturnValue:        ClipboardStatus

xmClipboardEndRetrieve(+Display,+Window,-ReturnValue)
Display:    Display
Window:     Window
ReturnValue:        ClipboardStatus
```

```
xmClipboardInquireCount(+Display,+Window,-Count,-MaxFormatLen,
        -ReturnValue)
Display:     Display
Window:      Window
Count:       Integer
MaxFormatLen:        Integer
ReturnValue:         ClipboardStatus

xmClipboardInquireFormat(+Display,+Window,+Index,-Format,
        +BufferLen,-CopiedLen,-ReturnValue)
Display:     Display
Window:      Window
Index:       Integer
Format:      String
BufferLen:   Integer
CopiedLen:   Integer
ReturnValue:         ClipboardStatus

xmClipboardInquireLength(+Display,+Window,+Format,-Length,-ReturnValue
)
Display:     Display
Window:      Window
Format:      String
Length:      Integer
ReturnValue:         ClipboardStatus

xmClipboardInquirePendingItems(+Display,+Window,+Format,-ItemList,-
Count,
        -ReturnValue)
Display:     Display
Window:      Window
Format:      String
ItemList:    XmClipboardPendingList
Count:       Integer
ReturnValue:         ClipboardStatus

xmClipboardLock(+Display,+Window,-ReturnValue)
Display:     Display
Window:      Window
ReturnValue:         ClipboardStatus

xmClipboardRegisterFormat(+Display,+Format,+FormatLength,-ReturnValue)
Display:     Display
Format:      String
FormatLength:        Integer
ReturnValue:         ClipboardStatus
```

```
xmClipboardRetrieve(+Display,+Window,+Format,-Buffer,
         +Length,-NumBytes,-PrivateId,-ReturnValue)
Display:     Display
Window:      Window
Format:      String
Buffer:      String or XtPointer
Length:      Integer
NumBytes:    Integer
PrivateId:   Integer
ReturnValue:         ClipboardStatus

xmClipboardStartCopy(+Display,+Window,+ClipLabel,+TimeStamp,
         +Widget,+Callback,-ItemId,-ReturnValue)
Display:     Display
Window:      Window
ClipLabel:   XmString
TimeStamp:   Time
Widget:      Widget
Callback:    CallbackProc
ItemId:      Integer
ReturnValue:         ClipboardStatus

xmClipboardStartRetrieve(+Display,+Window,+TimeStamp)
Display:     Display
Window:      Window
TimeStamp:   Time

xmClipboardUndoCopy(+Display,+Window,-ReturnValue)
Display:     Display
Window:      Window
ReturnValue:         ClipboardStatus

xmClipboardUnlock(+Display,+Window,+RemoveAllLocks,-ReturnValue)
Display:     Display
Window:      Window
RemoveAllLocks:      Boolean
ReturnValue:         ClipboardStatus

xmClipboardWithdrawFormat(+Display,+Window,+DataId,-ReturnValue)
Display:     Display
Window:      Window
DataId:      Integer
ReturnValue:         ClipboardStatus

xmCommandAppendValue(+Widget, +Command)
Widget:      Widget
Command:     XmString
```

xmCommandError(+*Widget*, +*Error*)
Widget: **Widget**
Error: **XmString**

xmCommandGetChild(+*Widget*, +*ChildType*, -*Child*)
Widget: **Widget**
ChildType: **XmChildType**
Child: **Widget**

xmCommandSetValue(+*Widget*, +*Command*)
Widget: **Widget**
Command: **XmString**

xmConvertUnits(+*Widget*,+*Orientation*,+*FromUnitType*,+*FromValue*,
+*ToUnitType*,-*ToValue*)
Widget: **Widget**
Orientation: **XmOrientation**
FromUnitType: **XmUnitType**
FromValue: **Integer**
ToUnitType: **XmUnitType**
ToValue: **Integer**

xmCreateArrowButton(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateArrowButtonGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateBulletinBoard(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateBulletinBoardDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateCascadeButton(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateCascadeButtonGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateCommand(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateDialogShell(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateDragIcon(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateDrawingArea(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateDrawnButton(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateErrorDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateFileSelectionBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateFileSelectionDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateForm(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateFormDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateFrame(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateInformationDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateLabel(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateLabelGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateList(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateMainWindow(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateMenuBar(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateMenuShell(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateMessageBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateMessageDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateOptionMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePanedWindow(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePopupMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePromptDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePulldownMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePushButton(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreatePushButtonGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateQuestionDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateRadioBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateRowColumn(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateScale(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateScrollBar(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateScrolledList(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateScrolledText(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateScrolledWindow(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSelectionBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSelectionDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSeparator(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSeparatorGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimpleCheckBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimpleMenuBar(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimpleOptionMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimplePopupMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimplePulldownMenu(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateSimpleRadioBox(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateTemplateDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateText(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateTextField(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateToggleButton(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateToggleButtonGadget(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateWarningDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateWorkArea(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCreateWorkingDialog(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCvtCTToXmString(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmCvtXmStringToCT(+*Parent*,+*Name*,+*Attributes*,-*Widget*)
Parent: **Widget**
Name: **String**
Attributes: **AttributeList**
Widget: **Widget**

xmDeactivateProtocol(+*Shell*,+*Property*,+*Protocol*)
Shell: **Widget**
Property: **XAtom**
Protocol: **XAtom**

xmDeactivateWMProtocol(+*Shell*,+*Protocol*)
Shell: **Widget**
Protocol: **XAtom**

xmDestroyPixmap(+*Screen*, +*Pixmap*)
Screen: **Screen**
Pixmap: **Pixmap**

xmDragCancel(+*DragContext*)
DragContext: **Widget**

xmDragStart(+*Widget*,+*Event*,+*Attributes*,-*DragContext*)
Widget: **Widget**
Event: **XEvent**
Attributes: **AttributeList**
DragContext: **Widget**

xmDropSiteConfigureStackingOrder(+*Widget*,+*Sibling*,+*StackMode*)
Widget: **Widget**
Sibling: **Widget**
StackMode: **XmStackMode**

xmDropSiteEndUpdate(+*Widget*)
Widget: **Widget**

xmDropSiteQueryStackingOrder(+*Widget*,-*Parent*,-*Children*)
Widget: **Widget**
Parent: **Widget**
Children: **WidgetList**

xmDropSiteRegister(*[+Widget]*,+*Attributes*
Widget: **Widget**
Attributes: **AttributeList**

xmDropSiteRetrieve(+*Widget*,+*Attributes*)
Widget: **Widget**
Attributes: **AttributeList**

```
xmDropSiteStartUpdate(+Widget)
Widget:      Widget

xmDropSiteUnregister(+Widget)
Widget:      Widget

xmDropSiteUpdate(+Widget,+Attributes)
Widget:      Widget
Attributes:  AttributeList

xmDropTransferAdd(+DropTransfer,+Transfers)
DropTransfer:        Widget
Transfers:   XmDropTransfers

xmDropTransferStart(+Widget,+Attributes,-DropWidget)
Widget:      Widget
Attributes:  AttributeList
DropWidget:  Widget

xmFileSelectionBoxGetChild(+Widget, +ChildType, -Child)
Widget:      Widget
ChildType:   XmChildType
Child:       Widget

xmFileSelectionDoSearch(+Widget, +DirMask)
Widget:      Widget
DirMask:     XmString

xmFontListAdd(+OldList,+NewFont,+Charset,-NewList)
OldList:     XmFontList
NewFont:     XFont
Charset:     XmStringCharSet
NewList:     XmFontList

xmFontListAppendEntry(+OldList,+Entry,-NewList)
OldList:     XmFontList
Entry:       XmFontListEntry
NewList:     XmFontList

xmFontListCopy(+FontList,-NewFontList)
FontList:    XmFontList
NewFontList:         XmFontList

xmFontListCreate(+Font,+Charset,-FontList)
Font:        XFont
Charset:     XmStringCharSet
FontList:    XmFontList
```

xmFontListEntryCreate(+*Tag*,+*Type*,+*Font*,-*Entry*)
Tag: **XmFontListTag**
Type: **XmFontType**
Font: **XFont** or **XFontSet**
Entry: **XmFontListEntry**

xmFontListEntryFree(+*Entry*)
Entry: **XmFontListEntry**

xmFontListEntryGetFont(+*Entry*,-*Type*,-*Font*)
Entry: **XmFontListEntry**
Type: **XmFontType**
Font: **XFont** or **XFontSet**

xmFontListEntryGetTag(+*Entry*,-*Tag*)
Entry: **XmFontListEntry**
Tag: **XmFontListTag**

xmFontListEntryLoad(+*Display*,+*FontName*,+*Type*,+*Tag*,-*Entry*)
Display: **Display**
FontName: **String**
Type: **XmFontType**
Tag: **XmFontListTag**
Entry: **XmFontListEntry**

xmFontListFree(+*FontList*)
FontList: **XmFontList**

xmFontListFreeFontContext(+*Context*)
Context: **XmFontContext**

xmFontListGetNextFont(+*Context*,-*Charset*,-*Font*)
Context: **XmFontContext**
Charset: **XmStringCharSet**
Font: **XFont**

xmFontListInitFontContext(-*Context*,+*FontList*)
Context: **XmFontContext**
FontList: **XmFontList**

xmFontListNextEntry(+*Context*,-*Entry*)
Context: **XmFontContext**
Entry: **XmFontListEntry**

xmFontListRemoveEntry(+*OldList*,+*Entry*,-*NewList*)
OldList: **XmFontList**
Entry: **XmFontListEntry**
NewList: **XmFontList**

```
xmGetAtomName(+Display,+Atom,-AtomName)
Display:    Display
Atom:       XAtom
AtomName:   String

xmGetColorCalculation(-ColorProc)
ColorProc:  CallbackProc

xmGetColors(+Screen,+Colormap,+Background,
               -Foreground,-TopShadow,-BottomShadow,-Select)
Screen:     Screen
Colormap:   Colormap
Background: Pixel
Foreground: Pixel
TopShadow:  Pixel
BottomShadow:       Pixel
Select:     Pixel

xmGetDestination(+Display, -Widget)
Display:    Display
Widget:     Widget

xmGetDragContext(+RefWidget, +TimeStamp, -Widget)
RefWidget:  Widget
TimeStamp:  Time
Widget:     Widget

xmGetFocusWidget(+Widget, -FocusWidget)
Widget:     Widget
FocusWidget:        Widget

xmGetMenuCursor(+Display, -Cursor)
Display:    Display
Cursor:     Cursor

xmGetPixmap(+Screen,+ImageName,+ForeG,+BackG,-Pixmap)
Screen:     Screen
ImageName:  String
ForeG:      Pixel
BackG:      Pixel
Pixmap:     Pixmap
```

xmGetPixmapByDepth(+*Screen*,+*ImageName*,+*ForeG*,+*BackG*,+*Depth*,-*Pixmap*)
Screen: **Screen**
ImageName: **String**
ForeG: **Pixel**
BackG: **Pixel**
Depth: **Integer**
Pixmap: **Pixmap**

xmGetPostedFromWidget(+*Menu*,-*Widget*)
Menu: **Widget**
Widget: **Widget**

xmGetTabGroup(+*Widget*,-*TabGroup*)
Widget: **Widget**
TabGroup: **Widget**

xmGetTearOffControl(+*Menu*,-*Widget*)
Menu: **Widget**
Widget: **Widget**

xmGetVisibility(+*Widget*,-*Visibility*)
Widget: **Widget**
Visibility: **XmVisibility**

xmGetXmDisplay(+*Widget*,-*DisplayObject*)
Widget: **Widget**
DisplayObject: **Widget**

xmGetXmScreen(+*Widget*,-*ScreenObject*)
Widget: **Widget**
ScreenObject: **Widget**

xmInstallImage(+*Image*, +*Name*)
Image: **XImage**
Name: **String**

xmInternAtom(+*Display*,+*AtomName*,+*OnlyIfExists*,-*Atom*)
Display: **Display**
AtomName: **String**
OnlyIfExists: **Boolean**
Atom: **XAtom**

xmIsMotifWMRunning(+*Shell*)
Shell: **Widget**

xmIsTraversable(+*Widget*)
Widget: **Widget**

```
xmListAddItem(+Widget, +Item, +Position)
Widget:     Widget
Item:       XmString
Position:   Integer

xmListAddItemUnselected(+Widget, +Item, +Position)
Widget:     Widget
Item:       XmString
Position:   Integer

xmListAddItems(+Widget, +ItemList, +Position)
Widget:     Widget
ItemList:   XmStringTable
Position:   Integer

xmListAddItemsUnselected(+Widget, +ItemList, +Position)
Widget:     Widget
ItemList:   XmStringTable
Position:   Integer

xmListDeleteAllItems(+Widget)
Widget:     Widget

xmListDeleteItem(+Widget, +Item)
Widget:     Widget
Item:       XmString

xmListDeleteItems(+Widget, +ItemList)
Widget:     Widget
ItemList:   XmStringTable

xmListDeleteItemsPos(+Widget, +ItemCount, +Position)
Widget:     Widget
ItemCount:  Integer
Position:   Integer

xmListDeletePos(+Widget, +Position)
Widget:     Widget
Position:   Integer

xmListDeletePositions(+Widget, +PositionList)
Widget:     Widget
PositionList:       IntegerList

xmListDeselectAllItems(+Widget)
Widget:     Widget
```

`xmListDeselectItem(+`*Widget*`, +`*Item*`)`
Widget: **Widget**
Item: **XmString**

`xmListDeselectPos(+`*Widget*`, +`*Position*`)`
Widget: **Widget**
Position: **Integer**

`xmListGetKbdItemPos(+`*Widget*`, -`*Position*`)`
Widget: **Widget**
Position: **Integer**

`xmListGetMatchPos(+`*Widget*`, +`*Item*`, -`*PositionList*`)`
Widget: **Widget**
Item: **XmString**
PositionList: **IntegerList**

`xmListGetSelectedPos(+`*Widget*`, -`*PositionList*`)`
Widget: **Widget**
PositionList: **IntegerList**

`xmListItemExists(+`*Widget*`, +`*Item*`)`
Widget: **Widget**
Item: **XmString**

`xmListItemPos(+`*Widget*`, +`*Item*`, -`*Position*`)`
Widget: **Widget**
Item: **XmString**
Position: **Integer**

`xmListPosSelected(+`*Widget*`, +`*Position*`)`
Widget: **Widget**
Position: **Integer**

`xmListPosToBounds(+`*Widget*`, +`*Position*`, -`*X*`, -`*Y*`, -`*Width*`, -`*Height*`)`
Widget: **Widget**
Position: **Integer**
X: **Integer**
Y: **Integer**
Width: **Integer**
Height: **Integer**

`xmListReplaceItems(+`*Widget*`, +`*OldItems*`, +`*NewItems*`)`
Widget: **Widget**
OldItems: **XmStringTable**
NewItems: **XmStringTable**

```
xmListReplaceItemsPos(+Widget, +NewItems, +Position)
Widget:     Widget
NewItems:   XmStringTable
Position:   Integer

xmListReplaceItemsPosUnselected(+Widget, +NewItems, +Position)
Widget:     Widget
NewItems:   XmStringTable
Position:   Integer

xmListReplaceItemsUnselected(+Widget, +OldItems, +NewItems)
Widget:     Widget
OldItems:   XmStringTable
NewItems:   XmStringTable

xmListReplacePositions(+Widget, +PosList, +NewItems)
Widget:     Widget
PosList:    IntegerList
NewItems:   XmStringTable

xmListSelectItem(+Widget, +Item, +Notify)
Widget:     Widget
Item:       XmString
Notify:     Boolean

xmListSelectPos(+Widget, +Position, +Notify)
Widget:     Widget
Position:   Integer
Notify:     Boolean

xmListSetAddMode(+Widget, +State)
Widget:     Widget
State:      Boolean

xmListSetBottomItem(+Widget, +Item)
Widget:     Widget
Item:       XmString

xmListSetBottomPos(+Widget, +Position)
Widget:     Widget
Position:   Integer

xmListSetHorizPos(+Widget, +Position)
Widget:     Widget
Position:   Integer
```

```
xmListSetItem(+Widget, +Item)
Widget:     Widget
Item:       XmString

xmListSetKbdItemPos(+Widget, +Position)
Widget:     Widget
Position:   Integer

xmListSetPos(+Widget, +Position)
Widget:     Widget
Position:   Integer

xmListUpdateSelectedList(+Widget)
Widget:     Widget

xmListYToPos(+Widget, +Y, -Position)
Widget:     Widget
Y:  Position
Position:   Integer

xmMainWindowSep1(+MainWindow, -Separator1)
MainWindow: Widget
Separator1: Widget

xmMainWindowSep2(+MainWindow, -Separator2)
MainWindow: Widget
Separator2: Widget

xmMainWindowSep3(+MainWindow, -Separator3)
MainWindow: Widget
Separator3: Widget

xmMainWindowSetAreas(+MainWindow, +MenuBar, +CommandWindow,
        +HorizontalScrollbar, +VerticalScrollbar, +WorkRegion)
MainWindow: Widget
MenuBar:    Widget
CommandWindow:      Widget
HorizontalScrollbar:        Widget
VerticalScrollbar:  Widget
WorkRegion: Widget

xmMapSegmentEncoding(+FontListTag,-Encoding)
FontListTag:        XmFontListTag
Encoding:   String

xmMenuPosition(+Menu, +Event)
Menu:       Widget
Event:      XEvent
```

```
xmMessageBoxGetChild(+MessageBox, +ChildType, -Child)
MessageBox: Widget
ChildType:  XmChildType
Child:      Widget

xmOptionButtonGadget(+OptionMenu, -ButtonGadget)
OptionMenu: Widget
ButtonGadget:       Widget

xmOptionLabelGadget(+OptionMenu, -LabelGadget)
OptionMenu: Widget
LabelGadget:        Widget

xmProcessTraversal(+Widget,+Direction)
Widget:     Widget
Direction:  XmTraversalDirection

xmRegisterSegmentEncoding(+FontListTag,+Encoding,-NewTag)
FontListTag:        XmFontListTag
Encoding:   String
NewTag:     XmFontListTag

xmRemoveProtocolCallback(+Shell,+Property,+Protocol,+Callback,+ClientData
)
Shell:      Widget
Property:   XAtom
Protocol:   XAtom
Callback:   CallbackProc
ClientData: AnyTerm

xmRemoveProtocols(+Shell,+Property,+Protocols)
Shell:      Widget
Property:   XAtom
Protocols:  XAtomList

xmRemoveTabGroup(+TabGroup)
TabGroup:   Widget

xmRemoveWMProtocolCallback(+Shell,+Protocol,+Callback,+ClientData)
Shell:      Widget
Protocol:   XAtom
Callback:   CallbackProc
ClientData: AnyTerm

xmRemoveWMProtocols(+Shell,+Protocols)
Shell:      Widget
Protocols:  XAtomList
```

```
xmRepTypeAddReverse(+RepTypeId)
RepTypeId:   XmRepTypeId

xmRepTypeGetId(+RepType, -RepTypeId)
RepType:     String
RepTypeId:   XmRepTypeId

xmRepTypeGetNameList(+RepTypeId, UseUC, -NameList)
RepTypeId:   XmRepTypeId
UseUC:       Boolean
NameList:    StringTable

xmRepTypeGetRecord(+RepTypeId, -RepTypeRecord)
RepTypeId:   XmRepTypeId
RepTypeRecord:       XmRepTypeEntry

xmRepTypeGetRegistered(-RepTypeList)
RepTypeList:         XmRepTypeList

xmRepTypeInstallTearOffModelConverter

xmRepTypeRegister(+RepType, +ValueNames, -RepTypeId)
RepType:     String
ValueNames:  StringTable
RepTypeId:   XmRepTypeId

xmRepTypeValidValue(+RepTypeId, +TestValue, +EnableWarning)
RepTypeId:   XmRepTypeId
TestValue:   Integer
EnableWarning:       Widget

xmScaleGetValue(+Widget, -Value)
Widget:      Widget
Value:       Integer

xmScaleSetValue(+Widget, +Value)
Widget:      Widget
Value:       Integer

xmScrollBarGetValues(+Widget, -Value, -SliderSize, -Increment,
         -PageIncrement)
Widget:      Widget
Value:       Integer
SliderSize:  Integer
Increment:   Integer
PageIncrement:       Integer
```

```
xmScrollBarSetValues(+Widget, -Value, -SliderSize, -Increment,
         -PageIncrement,-Notify)
Widget:      Widget
Value:       Integer
SliderSize:  Integer
Increment:   Integer
PageIncrement:       Integer
Notify:      Boolean

xmScrollVisible(+ScrollWidget,+Widget,+LRMargin,+TopBtmMargin)
ScrollWidget:        Widget
Widget:      Widget
LRMargin:    Dimension
TopBtmMargin:        Dimension

xmScrolledWindowSetAreas(+Widget,+MenuBar,+HorizontalScrollbar,
         +VerticalScrollbar,+WorkRegion)
Widget:      Widget
HorizontalScrollbar:         Widget
VerticalScrollbar:   Widget
WorkRegion:  Widget

xmSelectionBoxGetChild(+Widget,+ChildType,-Child)
Widget:      Widget
ChildType:   XmChildType
Child:       Widget

xmSetColorCalculation(+ColorProc, -PrevProc)
ColorProc:   CallbackProc
PrevProc:    CallbackProc

xmSetFontUnit(+Display, +FontUnitValue)
Display:     Display
FontUnitValue:       Integer

xmSetFontUnits(+Display, +HValue, +VValue)
Display:     Display
HValue:      Integer
VValue:      Integer

xmSetMenuCursor(+Display, +Cursor)
Display:     Display
CursorId     Cursor
```

```
xmSetProtocolHooks(+Shell,+Property,+Protocol,+PreHook,
             +PreHookData,+PostHook,+PostHookData)
Shell:       Widget
Property:    XAtom
Protocol:    XAtom
PreHook:     CallbackProc
PreHookData:         AnyTerm
PostHook:    CallbackProc
PostHookData:        AnyTerm

xmSetWMProtocolHooks(+Shell,+Protocol,+PreHook,
             +PreHookData,+PostHook,+PostHookData)
Shell:       Widget
Protocol:    XAtom
PreHook:     CallbackProc
PreHookData:         AnyTerm
PostHook:    CallbackProc
PostHookData:        AnyTerm

xmStringBaseline(+Fontlist, +String, -Baseline)
Fontlist:    XmFontList
String:      XmString
Baseline:    Dimension

xmStringByteCompare(+String1, +String2)
String1:     XmString
String2:     XmString

xmStringCompare(+String1, +String2)
String1:     XmString
String2:     XmString

xmStringConcat(+String1, +String2, -ResultString)
String1:     XmString
String2:     XmString
ResultString:        XmString

xmStringCopy(+String, -ResultString)
String:      XmString
ResultString:        XmString

xmStringCreate(+Text, +Tag, -String)
Text:        String
Tag:         XmFontListTag
String:      XmString
```

```
xmStringCreateLocalized(+Text, -String)
Text:       String
String:     XmString

xmStringCreateLtoR(+Text, +Tag, -String)
Text:       String
Tag:        XmFontListTag
String:     XmString

xmStringCreateSimple(+Text, -String)
Text:       String
String:     XmString

xmStringDirectionCreate(+Direction, -String)
Direction:  XmStringDirection
String:     XmString

xmStringDraw(+Display,+Window,+FontList,+String,+GC,
         +X,+Y,+Width,+Alignment,+LayoutDirection,+Clip)
Display:    Display
Window:     Window
FontList:   XmFontList
String:     XmString
GC: XGC
X:  Position
Y:  Position
Width:      Dimension
Alignment:  XmAlignment
LayoutDirection:    XmStringDirection
Clip:       XRectangle

xmStringDrawImage(+Display,+Window,+FontList,+String,+GC,
         +X,+Y,+Width,+Alignment,+LayoutDirection,+Clip)
Display:    Display
Window:     Window
FontList:   XmFontList
String:     XmString
GC: XGC
X:  Position
Y:  Position
Width:      Dimension
Alignment:  XmAlignment
LayoutDirection:    XmStringDirection
Clip:       XRectangle
```

```
xmStringDrawUnderline(+Display,+Window,+FontList,+String,+GC,
        +X,+Y,+Width,+Alignment,+LayoutDirection,+Clip,+Underline)
Display:     Display
Window:      Window
FontList:    XmFontList
String:      XmString
GC:  XGC
X:   Position
Y:   Position
Width:       Dimension
Alignment:   XmAlignment
LayoutDirection:     XmStringDirection
Clip:        XRectangle
Underline:   XmString

xmStringEmpty(+String)
String:      XmString

xmStringExtent(+FontList, +String, +Width, +Height)
FontList:    XmFontList
String:      XmString
Width:       Dimension
Height:      Dimension

xmStringFree(+String)
String:      XmString

xmStringFreeContext(+Context)
Context:     XmStringContext

xmStringGetLtoR(+String, +Tag, -Text)
String:      XmString
Tag:         XmFontListTag
Text:        String

xmStringGetNextComponent(+Context, -Text, -Tag,
        -Direction, -UnknownTag, -UnknownLength, -UnknownValue, -
Result)
Context:     XmStringContext
Text:        String
Tag:         XmFontListTag
Direction:   XmStringDirection
UnknownTag:  XmStringComponentType
UnknownLength        Integer
UnknownValue:        String
Result:      XmStringComponentType
```

```
xmStringGetNextSegment(+Context, -Text, -Tag,
         -Direction, -Separator)
Context:     XmStringContext
Text:        String
Tag:         XmFontListTag
Direction:   XmStringDirection
Separator:   Boolean

xmStringHasSubstring(+String, +SubString)
String:      XmString
SubString:   XmString

xmStringHeight(+FontList, +String, -Height)
FontList:    XmFontList
String:      XmString
Height::     Dimension

xmStringInitContext(-Context, +String)
Context:     XmStringContext
String:      XmString

xmStringLength(+String, -Length)
String:      XmString
Length:      Integer

xmStringLineCount(+String, -Lines)
String:      XmString
Lines:       Integer

xmStringNConcat(+String1, +String2, -NumBytes, -ResultString)
String1:     XmString
String2:     XmString
NumBytes:    Integer
ResultString:        XmString

xmStringNCopy(+String,+NumBytes, -ResultString)
String:      XmString
NumBytes:    Integer
ResultString:        XmString

xmStringPeekNextComponent(+StringContext, -Component)
StringContext:       XmStringContext
Component:   XmStringComponentType
```

xmStringSegmentCreate(+*Text*, +*Tag*, +*Direction*, +*Separator*, -*String*)
Text: **String**
Tag: **XmFontListTag**
Direction: **XmStringDirection**
Separator: **Boolean**
String: **XmString**

xmStringSeparatorCreate(-*Separator*)
Separator: **XmString**

xmStringWidth(+*FontList*, +*String*, -*Width*)
FontList: **XmFontList**
String: **XmString**
Width: **Dimension**

xmTargetsAreCompatible(+*Display*,+*ExportTargets*,+*ImportTargets*)
Display: **Display**
ExportTargets: **XAtomList**
ImportTargets: **XAtomList**

xmTextClearSelection(+*Widget*, +*Time*)
Widget: **Widget**
Time: **Time**

xmTextCopy(+*Widget*,+*Time*)
Widget: **Widget**
Time: **Time**

xmTextCut(+*Widget*,+*Time*)
Widget: **Widget**
Time: **Time**

xmTextDisableRedisplay(+*Widget*)
Widget: **Widget**

xmTextEnableRedisplay(+*Widget*)
Widget: **Widget**

xmTextFieldClearSelection(+*Widget*,+*Time*)
Widget: **Widget**
Time: **Time**

xmTextFieldCopy(+*Widget*,+*Time*)
Widget: **Widget**
Time: **Time**

xmTextFieldCut(+*Widget*,+*Time*)
Widget: **Widget**
Time: **Time**

xmTextFieldGetBaseline(+*Widget*,-*Position*)
Widget: **Widget**
Position: **Integer**

xmTextFieldGetCursorPosition(+*Widget*,-*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldGetEditable(+*Widget*)
Widget: **Widget**

xmTextFieldGetInsertionPosition(+*Widget*,-*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldGetLastPosition(+*Widget*,-*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldGetMaxLength(+*Widget*,-*Length*)
Widget: **Widget**
Length: **Integer**

xmTextFieldGetSelection(+*Widget*,-*Selection*)
Widget: **Widget**
Selection: **String**

xmTextFieldGetSelectionPosition(+*Widget*,-*Left*,-*Right*)
Widget: **Widget**
Left: **XmTextPosition**
Right: **XmTextPosition**

xmTextFieldGetSelectionWcs(+*Widget*,-*Selection*)
Widget: **Widget**
Selection: **XmValueWcs**

xmTextFieldGetString(+*Widget*,-*Text*)
Widget: **Widget**
Text: **String**

xmTextFieldGetStringWcs(+*Widget*,-*Text*)
Widget: **Widget**
Text: **XmValueWcs**

xmTextFieldGetSubstring(+*Widget*,+*Start*,+*Length*,-*Text*,-*Status*)
Widget: **Widget**
Start: **XmTextPosition**
Length: **Integer**
Text: **String**
Status: **CopyStatus**

xmTextFieldGetSubstringWcs(+*Widget*,+*Start*,+*Length*,-*Text*,-*Status*)
Widget: **Widget**
Start: **XmTextPosition**
Length: **Integer**
Text: **XmValueWcs**
Status: **CopyStatus**

xmTextFieldInsert(+*Widget*,+*Position*,+*Value*)
Widget: **Widget**
Position: **XmTextPosition**
Value: **String**

xmTextFieldInsertWcs(+*Widget*,+*Position*,+*Value*)
Widget: **Widget**
Position: **XmTextPosition**
Value: **XmValueWcs**

xmTextFieldPaste(+*Widget*)
Widget: **Widget**

xmTextFieldPosToXY(+*Widget*,+*Position*,-*X*,-*Y*)
Widget: **Widget**
Position: **XmTextPosition**
X: **Position**
Y: **Position**

xmTextFieldRemove(+*Widget*)
Widget: **Widget**

xmTextFieldReplace(+*Widget*,+*FromPos*,+*ToPos*,+*Text*)
Widget: **Widget**
FromPos: **XmTextPosition**
ToPos: **XmTextPosition**
Text: **String**

xmTextFieldReplaceWcs(+*Widget*,+*FromPos*,+*ToPos*,+*Text*)
Widget: **Widget**
FromPos: **XmTextPosition**
ToPos: **XmTextPosition**
Text: **XmValueWcs**

xmTextFieldSetAddMode(+*Widget*,+*State*)
Widget: **Widget**
State: **Boolean**

xmTextFieldSetCursorPosition(+*Widget*,+*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldSetEditable(+*Widget*,+*Editable*)
Widget: **Widget**
Editable: **Boolean**

xmTextFieldSetHighlight(+*Widget*,+*Left*,+*Right*,+*Mode*)
Widget: **Widget**
Left: **XmTextPosition**
Right: **XmTextPosition**
Mode: **XmHighlightMode**

xmTextFieldSetInsertionPosition(+*Widget*,+*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldSetMaxLength(+*Widget*,+*MaxLength*)
Widget: **Widget**
MaxLength: **Integer**

xmTextFieldSetSelection(+*Widget*,+*First*,+*Last*,+*Time*)
Widget: **Widget**
First: **XmTextPosition**
Last: **XmTextPosition**
Time: **Time**

xmTextFieldSetString(+*Widget*,+*String*)
Widget: **Widget**
String: **String**

xmTextFieldSetStringWcs(+*Widget*,+*String*)
Widget: **Widget**
String: **XmValueWcs**

xmTextFieldShowPosition(+*Widget*,+*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextFieldXYToPos(+*Widget*,+*X*,+*Y*,-*Position*)
Widget: **Widget**
X: **Position**
Y: **Position**
Position: **XmTextPosition**

xmTextFindString(+*Widget*,+*Start*,+*String*,+*Direction*,-*Position*)
Widget: **Widget**
Start: **XmTextPosition**
String: **String**
Direction: **XmTextDirection**
Position: **XmTextPosition**

xmTextFindStringWcs(+*Widget*,+*Start*,+*String*,+*Direction*,-*Position*)
Widget: **Widget**
Start: **XmTextPosition**
String: **XmValueWcs**
Direction: **XmTextDirection**
Position: **XmTextPosition**

xmTextGetBaseline(+*Widget*,-*Position*)
Widget: **Widget**
Position: **Integer**

xmTextGetCursorPosition(+*Widget*,-*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextGetEditable(+*Widget*)
Widget: **Widget**

xmTextGetInsertionPosition(+*Widget*, -*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextGetLastPosition(+*Widget*, -*Position*)
Widget: **Widget**
LastPosition: **XmTextPosition**

xmTextGetMaxLength(+*Widget*, -*Length*)
Widget: **Widget**
Length: **Integer**

xmTextGetSelection(+*Widget*, -*Selection*)
Widget: **Widget**
Selection: **String**

xmTextGetSelectionPosition(+*Widget*, -*Left*, -*Right*, -*Success*)
Widget: **Widget**
Left: **XmTextPosition**
Right: **XmTextPosition**
Success: **Boolean**

xmTextGetSelectionWcs(+*Widget*,-*Selection*)
Widget: **Widget**
Selection: **XmValueWcs**

xmTextGetSource(+*Widget*,-*Source*)
Widget: **Widget**
Source: **XmTextSource**

xmTextGetString(+*Widget*,-*Text*)
Widget: **Widget**
Text: **String**

xmTextGetStringWcs(+*Widget*,-*Text*)
Widget: **Widget**
Text: **XmValueWcs**

xmTextGetSubstring(+*Widget*,+*Start*,+*Length*,-*Text*,-*Status*)
Widget: **Widget**
Start: **XmTextPosition**
Length: **Integer**
Text: **String**
Status: **CopyStatus**

xmTextGetSubstringWcs(+*Widget*,+*Start*,+*Length*,-*Text*,-*Status*)
Widget: **Widget**
Start: **XmTextPosition**
Length: **Integer**
Text: **XmValueWcs**
Status: **CopyStatus**

xmTextGetTopCharacter(+*Widget*, -*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextInsert(+*Widget*,+*Position*,+*Value*)
Widget: **Widget**
Position: **XmTextPosition**
Value: **String**

xmTextInsertWcs(+*Widget*,+*Position*,+*Value*)
Widget: **Widget**
Position: **XmTextPosition**
Value: **XmValueWcs**

xmTextPaste(+*Widget*)
Widget: **Widget**

xmTextPosToXY(+*Widget*, +*Position*, -*X*, -*Y*)
Widget: **Widget**
Position: **XmTextPosition**
X: **Position**
Y: **Position**

xmTextRemove(+*Widget*)
Widget: **Widget**

xmTextReplace(+*Widget*,+*FromPos*,+*ToPos*,+*Text*)
Widget: **Widget**
FromPos: **XmTextPosition**
ToPos: **XmTextPosition**
Text: **String**

xmTextReplaceWcs(+*Widget*,+*FromPos*,+*ToPos*,+*Text*)
Widget: **Widget**
FromPos: **XmTextPosition**
ToPos: **XmTextPosition**
Text: **XmValueWcs**

xmTextScroll(+*Widget*, +*Lines*)
Widget: **Widget**
Lines: **Integer**

xmTextSetAddMode(+*Widget*,+*State*)
Widget: **Widget**
State: **Boolean**

xmTextSetCursorPosition(+*Widget*,+*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextSetEditable(+*Widget*, +*Editable*)
Widget: **Widget**
Editable: **Boolean**

xmTextSetHighlight(+*Widget*,+*Left*,+*Right*,+*Mode*)
Widget: **Widget**
Left: **XmTextPosition**
Right: **XmTextPosition**
Mode: **XmHighlightMode**

xmTextSetInsertionPosition(+*Widget*, +*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextSetMaxLength(+*Widget*, +*MaxLength*)
Widget: **Widget**
MaxLength: **Integer**

xmTextSetSelection(+*Widget*, +*First*, +*Last*, +*Time*)
Widget: **Widget**
First: **XmTextPosition**
Last: **XmTextPosition**
Time: **Time**

xmTextSetSource(+*Widget*, +*Source*, +*TopChar*, +*CursorPos*)
Widget: **Widget**
Source: **XmTextSource**
TopChar: **XmTextPosition**
CursorPos: **XmTextPosition**

xmTextSetString(+*Widget*, +*String*)
Widget: **Widget**
String: **String**

xmTextSetStringWcs(+*Widget*, +*String*)
Widget: **Widget**
String: **XmValueWcs**

xmTextSetTopCharacter(+*Widget*, +*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextShowPosition(+*Widget*,+*Position*)
Widget: **Widget**
Position: **XmTextPosition**

xmTextXYToPos(+*Widget*, +*X*, +*Y*, -*Position*)
Widget: **Widget**
X: **Position**
Y: **Position**
Position: **XmTextPosition**

xmToggleButtonGadgetGetState(+*Gadget*)
Gadget: **Widget**

xmToggleButtonGadgetSetState(+*Gadget*,+*State*,+*Notify*)
Gadget: **Widget**
State: **Boolean**
Notify: **Boolean**

xmToggleButtonGetState(+*Widget*)
Widget: **Widget**

xmToggleButtonSetState(+*Widget*, +*State*, +*Notify*)
Widget: **Widget**
State: **Boolean**
Notify: **Boolean**

xmTrackingEvent(+*Widget*, +*Cursor*, +*ConfineTo*, -*Event*, -*Result*)
Widget: **Widget**
Cursor: **Cursor**
ConfineTo: **Boolean**
Event: **XEvent**
Result: **Widget**

xmTrackingLocate(+*Widget*, +*Cursor*, +*ConfineTo*, -*Event*, -*Result*)
Widget: **Widget**
Cursor: **Cursor**
ConfineTo: **Boolean**
Result: **Widget**

xmTranslateKey(+*Display*,+*KeyCode*,+*Modifiers*, -*ModifiersRtn*,-*KeySym*)
Widget: **Widget**
KeyCode: **Integer**
Modifiers: **Integer**
ModifiersRtn: **Integer**
KeySym: **KeySym**

xmUninstallImage(+*Image*)
Image: **XImage**

xmUpdateDisplay(+*Widget*)
Widget: **Widget**

xmWidgetGetBaselines(+*Widget*, -*BaseLines*, -*LineCount*)
Widget: **Widget**
BaseLines: **IntegerList**
LineCount: **Integer**

```
xmWidgetGetDisplayRect(+Widget, -Rectangle)
Widget:     Widget
Rectangle:  XRectangle
```

17.5.2 X Toolkit Predicates

```
xtAddActions(+Actions)
Actions:    ActionsList

xtAddCallback(+Widget,+WidgetCallback,+Callback,+ClientData)
Widget:     Widget
WidgetCallback:     WidgetCallbackType
Callback:   CallbackProc
ClientData: AnyTerm

xtAddCallbacks(+Widget,+WidgetCallback,+Callbacks)
Widget:     Widget
WidgetCallback:     WidgetCallbackType
Callbacks:  CallbackList

xtAddEventHandler(+Widget,+EventMask,+NonMaskable,+EventHandler,+ClientData
)
Widget:     Widget
EventMask:  XEventMask
NonMaskable:        Boolean
EventHandler:       EventProc
ClientData: AnyTerm

xtAddGrab(+Widget,+Exclusive,+SpringLoaded)
Widget:     Widget
Exclusive:  Boolean
SpringLoaded:       Boolean

xtAddInput(+Source,+Condition,+InputProc,+ClientData,-InputId)
Source:     Integer
Condition:  XtInputCondMask
InputProc:  InputProc
ClientData: AnyTerm
InputId:    XtInputId

xtAddRawEventHandler(+Widget,+EventMask,+NonMaskable,+EventHandler,
        +ClientData)
Widget:     Widget
EventMask:  XEventMask
NonMaskable:        Boolean
EventHandler:       EventProc
ClientData: AnyTerm
```

xtAddTimeOut(+*Interval*,+*TimerProc*,+*ClientData*,-*IntervalId*)
Interval: **Integer**
TimerProc: **TimerProc**
ClientData: **AnyTerm**
IntervalId: **XtIntervalId**

xtAddWorkProc(+*WorkProc*,+*ClientData*,-*WorkProcId*)
WorkProc: **WorkProc**
ClientData: **AnyTerm**
WorkProcId: **XtWorkProcId**

xtAppAddActions(+*AppContext*,+*Actions*)
AppContext: **XtAppContext**
Actions: **ActionsList**

xtAppAddInput(+*AppContext*,+*Source*,+*Condition*,+*InputProc*,+*ClientData*,-*InputId*)
AppContext: **XtAppContext**
Source: **Integer**
Condition: **XtInputCondMask**
InputProc: **InputProc**
ClientData: **AnyTerm**
InputId: **XtInputId**

xtAppAddTimeOut(+*AppContext*,+*Interval*,+*TimerProc*,+*ClientData*,-*IntervalId*)
AppContext: **XtAppContext**
Interval: **Integer**
TimerProc: **TimerProc**
ClientData: **AnyTerm**
IntervalId: **XtIntervalId**

xtAppAddWorkProc(+*AppContext*,+*WorkProc*,+*ClientData*,-*WorkProcId*)
AppContext: **XtAppContext**
WorkProc: **WorkProc**
ClientData: **AnyTerm**
WorkProcId: **XtWorkProcId**

xtAppCreateShell(+*Name*,+*Class*,+*WidgetClass*,+*Display*,
+*Attributes*,-*Widget*)
Name: **String**
Class: **String**
WidgetClass: **WidgetClass**
Display: **Display**
Attributes: **AttributeList**
Widget: **Widget**

```
xtAppGetSelectionTimeout(+AppContext, +Timeout)
AppContext: XtAppContext
Timeout:    Integer

xtAppInitialize(-AppContext, +Class, +Attributes, -Widget)
AppContext: XtAppContext
Class:      String
Attributes: AttributeList
Widget:     Widget

xtAppMainLoop(+AppContext)
AppContext: XtAppContext

xtAppNextEvent(+AppContext, +Event)
AppContext: XtAppContext
Event:      XEvent

xtAppPeekEvent(+AppContext, +Event)
AppContext: XtAppContext
Event:      XEvent

xtAppPending(+AppContext, -Mask)
AppContext: XtAppContext
Mask        XtInputMask

xtAppProcessEvent(+AppContext, +Mask)
AppContext: XtAppContext
Mask        XtInputMask

xtAppSetErrorHandler(+AppContext, +ErrorProc)
AppContext: XtAppContext
ErrorProc:  CallbackProc

xtAppSetSelectionTimeout(+AppContext, +Timeout)
AppContext: XtAppContext
Timeout:    Integer

xtAppSetWarningHandler(+AppContext, +WarnProc)
AppContext: XtAppContext
WarnProc:   CallbackProc

xtAugmentTranslations(+Widget, +TranslationTable)
Widget:     Widget
TranslationTable:   XtTranslations

xtBuildEventMask(+Widget,-EventMask)
Widget:     Widget
EventMask:  XEventMask
```

xtCallbackExclusive(+*Widget*,+*Shell*,+*DummyClientData*)
Widget: **Widget**
Shell: **Widget**
DummyClientData: **Integer**

xtCallbackNone(+*Widget*,+*Shell*,+*DummyClientData*)
Widget: **Widget**
Shell: **Widget**
DummyClientData: **Integer**

xtCallbackNonexclusive(+*Widget*,+*Shell*[+DummyClientData])
Widget: **Widget**
Shell: **Widget**
DummyClientData: **Integer**

xtCallbackPopdown(+*Widget*,+*Shell*,+*DummyClientData*)
Widget: **Widget**
Shell: **Widget**
DummyClientData: **Integer**

xtClass(+*Widget*,-*WidgetClass*)
Widget: **Widget**
WidgetClass: **WidgetClass**

xtCloseDisplay(+*Display*)
Display: **Display**

xtConvert(+*Widget*,+*FromType*,+*From*,+*FromLen*,+*ToType*,-*To*,-*ToLen*)
Widget: **Widget**
FromType: **String**
From: **String** or **Integer**
FromLen: **Integer**
ToType: **String**
To: **String** or **Integer**
ToLen: **Integer**

xtCreateApplicationContext(-*AppContext*)
AppContext

xtCreateApplicationShell(+*Name*,+*WidgetClass*,+*Attributes*,-*Widget*)
Name: **String**
WidgetClass: **WidgetClass**
Attributes: **AttributeList**
Widget: **Widget**

```
xtCreateManagedWidget(+Name,+WidgetClass,+Parent,+Attributes,-Widget)
Name:        String
WidgetClass:         WidgetClass
Parent:      Widget
Attributes:  AttributeList
Widget:      Widget

xtCreatePopupShell(+Name,+WidgetClass,+Parent,+Attributes,-Widget)
Name:        String
WidgetClass:         WidgetClass
Parent:      Widget
Attributes:  AttributeList
Widget:      Widget

xtCreateWidget(+Name,+WidgetClass,+Parent,+Attributes,-Widget)
Name:        String
WidgetClass:         WidgetClass
Parent:      Widget
Attributes:  AttributeList
Widget:      Widget

xtDestroyApplicationContext(+AppContext)
AppContext:  XtAppContext

xtDestroyWidget(+Widget)
Widget:      Widget

xtDispatchEvent(+Event)
Event:       XEvent

xtDisplay(+Widget,-Display)
Widget:      Widget
Display:     Display

xtDisplayInitialize(+AppContext,+Display,+Name,+Class)
AppContext:  XtAppContext
Display:     Display
Name:        String
Class:       String

xtDisplayOfObject(+Widget,-Display)
Widget:      Widget
Display:     Display

xtDisplayToApplicationContext(+Display, -AppContext)
Display:     Display
AppContext:  XtAppContext
```

xtFree(+*Object*)
Object: **AnyTerm**

xtGetSelectionTimeout(+*AppContext*, +*Timeout*)
Timeout: **Integer**

xtGetValues(+*Widget*,+*Attributes*)
Widget: **Widget**
Attributes: **AttributeList**

xtHasCallbacks(+*Widget*,+*WidgetCallback*,+*CallbackStatus*)
Widget: **Widget**
WidgetCallback: **WidgetCallbackType**
CallbackStatus: **XtCallbackStatus**

xtInitialize(+*Name*,+*Class*,-*Widget*)
Name: **String**
Class: **String**
Widget: **Widget**

xtInitializeWidgetClass(+*Widget*)
Widget: **Widget**

xtInstallAccelerators(+*Destination*,+*Source*)
Destination: **Widget**
Source: **Widget**

xtInstallAllAccelerators(+*Destination*,+*Source*)
Destination: **Widget**
Source: **Widget**

xtIsManaged(+*Widget*)
Widget: **Widget**

xtIsObject(+*Widget*)
Widget: **Widget**

xtIsRealized(+*Widget*)
Widget: **Widget**

xtIsSensitive(+*Widget*)
Widget: **Widget**

xtIsSubclass(+*Widget*, +*WidgetClass*)
Widget: **Widget**
WidgetClass: **WidgetClass**

xtMainLoop

```
xtManageChild(+Widget)
Widget:      Widget

xtManageChildren(+WidgetList)
WidgetList:  WidgetList

xtName(+Widget,-Name)
Widget:      Widget
Name:        String

xtNameToWidget(+Reference, +Name, -Widget)
Reference:   Widget
Name:        String
Widget:      Widget

xtNextEvent(+Event)
Event:       XEvent

xtOpenDisplay(+AppContext,+Name,AppName,+Class,-Display)
AppContext:  XtAppContext
Name:        String
AppName:     String
Class:       String
Display:     Display

xtOverrideTranslations(+Widget, +TranslationTable)
Widget:      Widget
TranslationTable:    XtTranslations

xtParent(+Child,-Parent)
Child:       Widget
Parent:      Widget

xtParseAcceleratorTable(+Accelerators, -AcceleratorTable)
Accelerators:        String
AcceleratorTable:    XtAccelerators

xtParseTranslationTable(+Translations, -TranslationTable)
Translations:        String
TranslationTable:    XtTranslations

xtPeekEvent(-Event)
Event:       XEvent

xtPending(-InputMask)
InputMask:   XEventMask
```

xtPopdown(+*Widget*)
Widget: **Widget**

xtPopup(+*Widget*,+*GrabKind*)
Widget: **Widget**
GrabKind: **XtGrabKind**

xtProcessEvent(+*InputMask*)
InputMask: **XEventMask**

xtRealizeWidget(+*Widget*)
Widget: **Widget**

xtRemoveAllCallbacks(+*Widget*,+*WidgetCallback*)
Widget: **Widget**
WidgetCallback: **WidgetCallbackType**

xtRemoveCallback(+*Widget*,+*WidgetCallback*,+*Callback*,+*ClientData*)
Widget: **Widget**
WidgetCallback: **WidgetCallbackType**
Callback: **CallbackProc**
ClientData: **AnyTerm**

xtRemoveCallbacks(+*Widget*,+*WidgetCallback*,+*Callbacks*)
Widget: **Widget**
WidgetCallback: **WidgetCallbackType**
Callbacks: **CallbackList**

xtRemoveEventHandler(+*Widget*,+*EventMask*,+*NonMaskable*,+*EventHandler*,
+*ClientData*)
Widget: **Widget**
EventMask: **XEventMask**
NonMaskable: **Boolean**
EventHandler: **EventProc**
ClientData: **AnyTerm**

xtRemoveGrab(+*Widget*)
Widget: **Widget**

xtRemoveInput(+*InputId*)
InputId: **XtInputId**

```
xtRemoveRawEventHandler(+Widget,+EventMask,+NonMaskable,+EventHandler,
        +ClientData)
Widget:      Widget
EventMask:   XEventMask
NonMaskable:         Boolean
EventHandler:        EventProc
ClientData: AnyTerm

xtRemoveTimeOut(+IntervalId)
IntervalId: XtIntervalId

xtRemoveWorkProc(+WorkProcId)
WorkProcId: XtWorkProcId

xtScreen(+Widget,-Screen)
Widget:      Widget
Screen:      Screen

xtScreenOfObject(+Widget,-Screen)
Widget:      Widget
Screen:      Screen

xtSetErrorHandler(+ErrorProc)
ErrorProc:   CallbackProc

xtSetKeyboardFocus(+Subtree,+Descendant)
Subtree:     Widget
Descendant: Widget

xtSetMappedWhenManaged(+Widget,+MappedWhenManaged)
Widget:      Widget
MappedWhenManaged:   Boolean

xtSetSelectionTimeout(+Timeout)
Timeout:     Integer

xtSetSensitive(+Widget, +Sensitive)
Widget:      Widget
Sensitive:   Boolean

xtSetValues(+Widget,+Attributes)
Widget:      Widget
Attributes: AttributeList

xtSetWarningHandler(+WarnProc)
WarnProc:    CallbackProc
```

xtSuperclass(+*Widget*, -*WidgetClass*)
Widget: **Widget**
WidgetClass: **WidgetClass**

xtToolkitInitialize

xtTranslateCoords(+*Widget*, +*X*, +*Y*, -*RootX*, -*RootY*)
Widget: **Widget**
X: **Position**
Y: **Position**
RootXReturn: **Position**
RootYReturn: **Position**

xtUninstallTranslations(+*Widget*)
Widget: **Widget**

xtUnmanageChild(+*Widget*)
Widget: **Widget**

xtUnmanageChildren(+*WidgetList*)
WidgetList: **WidgetList**

xtUnrealizeWidget(+*Widget*)
Widget: **Widget**

xtWidgetToApplicationContext(+*Widget*,-*AppContext*)
Widget: **Widget**
AppContext: **XtAppContext**

xtWindow(+*Widget*,-*Window*)
Widget: **Widget**
Window: **Window**

xtWindowOfObject(+*Widget*,-*Window*)
Widget: **Widget**
Window: **Window**

xtWindowToWidget(+*Display*, +*Window*, +*Widget*)
Display: **Display**
Window: **Window**
Widget: **Widget**

xCreateGC(+*Display*, +*Window*, +*Values*, -*GC*)
Display: **Display**
Window: **Window**
Values: **XGCValues**
GC: **XGC**

xFreeGC(+*Display*, +*GC*)
Display: **Display**
GC: **XGC**

xLoadQueryFont(+*Display*, +*FontName*, -*Font*)
Display: **Display**
FontName: **String**
Font: **XFont**

17.5.3 ProXT Specific Predicates

proxtGetCallbackEvent(+*CallData*,-*Event*)
CallData: **Calldata**
Event: **XEvent**

proxtGetCallbackReason(+*CallData*,-*Reason*)
CallData: **Calldata**
Reason: **String**

proxtGetCallbackFields(+*CallData*,-*CallDataList*)
CallData: **Calldata**
CallDataList: **CalldataFields**

proxtGetWidgetClass(+*Widget*,-*WidgetClass*)
Widget: **Widget**
WidgetClass: **WidgetClass**

proxtGetEventFields(+*Event*,-*Fields*)
Event: **XEvent**
Fields: **EventFields**

proxtSetCallbackFields(+*CallData*,-*CallDataList*)
CallData: **Calldata**
CallDataList: **CalldataFields**

17.6 Changes from ProXT 3.1

17.6.1 Highlights

There are a number of changes from the previous version of ProXT that was released with Quintus Prolog 3.1. These changes are summarized as follows:

- Predicates and attributes that used to take a **CharPtr** type now take a **String**

type, which is represented as a Prolog atom. This means that the predicates `proxtCharPtrToString/2` and `proxtStringToCharPtr/2` are obsolete.

- The predicate `proxtGetDefaultCharset/1` is obsolete. In Motif2.1, most functions that used to take a *Charset* parameter now take a *FontListTag* parameter and the atom `xmDEFAULT_FONTLIST_TAG` can be used as a *FonListTag*.
- All predicates that correspond to Boolean functions in Motif now succeed or fail according to whether the Motif function returns `true` or `false`. This means that some predicates that used to include a *Results* parameter have been replaced by predicates with that argument removed.
- The *Calldata* argument returned by widget callbacks is now a handle, rather than a list of field, that can be inspected (and modified) by three new proxt predicates: the callback reason can be extracted by the predicate `proxtGetCallbackReason/2`; a list of all fields is returned by `proxtGetCallbackFields/2` (the contents of this list varies depending on the type of callback); and finally a list of fields to be modified can be specified in `proxtSetCallbackFields/2`.
- ProXT will now backtrack through callback predicates if they return nondeterminately.
- The client data argument in the **Action** datatype has been dropped and similarly the client data argument to action callbacks has been dropped. Instead parameters specified in the translation table are now passed into the action callback as a list of items of type **String** in the third argument. The second argument to the action callback is now the Event.
- Attributes that used to be of type **CallbackList** are now of type **CallbackTerm**, that is, they take a single callback as value, not a list of callbacks.

  The format of datatype **XRectangle** has changed. This is now a term of the form `rectangle(`*`X`*`,`*`Y`*`,`*`Width`*`,`*`Height`*`)`.
- The format of datatype **XmClipboardPendingList** has changed. This is now a list of terms of the form `clipboardpending(`*`DataId`*`,`*`PrivateId`*`)`.
- The return value of `xtPending/1` has changed, returning an **XEventMask** type rather than a `mask/1` structure.
- The predicate `proxtGetDefaultGC/2` is obsolete. The predicate `xCreateGC/4` should be used to define a GC graphics context.
- The predicate `proxtFree/1` has been renamed `xtFree/1`.
- The values returned as datatype **ClipboardStatus** have been renamed from `clipboard*` to `xmClipboard*` in line with the change made in Motif2.1.
- Some values included in the datatype **XmDialogType** have changed.
- The predicate `xtOpenDisplay` now has an additional argument specifying the *AppName*.

One of the main changes to ProXT 3.5 concerns the ability to extend ProXT with new widget classes and resources. The file `demo(proxrt.pl)` demonstrates how a graph widget can be incorporated into ProXT.

17.6.2 Backward Compatibility

Definitions for the predicates that are now obsolete in ProXT 3.5 are provided in the file `library(proxtbc)`. Many ProXT 3.1 programs will be able to run simply by adding the following declaration in the program:

```
:- use_module(library(proxtbc)).
```

This module includes appropriate definitions for `proxtCharPtrToString/2`, `proxtStringToCharPtr/2` and `proxtGetDefaultCharset/1` to enable programs that use these in conjunction with '`xmString`' and '`xmText`' predicates to continue to work.

18 Prolog Reference Pages

18.1 Reading the Reference pages

18.1.1 Overview

The reference pages for Quintus Prolog built-in predicates conform to certain conventions concerning

- mode annotations
- predicate categories
- argument types

These are particularly important in utilizing the Synopsis and Arguments fields of each reference page. The **Synopsis** field consists of the goal template(s) with mode annotations and a brief description of the purpose of the predicate. For example, consider this excerpt from the reference page for `assert/[1,2]`:

Synopsis

`assert`*(+Clause)* `assert`*(+Clause, -Ref)*

Adds dynamic clause *Clause* to the Prolog database. Returns database reference in *Ref*.

The **Arguments** field lists, for each metavariable name in the template, its argument type, (e.g. *callable*), a brief description (sometimes omitted), and an indication ('`[MOD]`') if it does module name expansion. For example,

Arguments

Clause callable [MOD]
A valid Prolog clause.

Ref db_reference

For further information see Section 18.1.3 [mpg-ref-cat], page 987.

18.1.2 Mode Annotations

The mode annotations are useful to tell whether an argument is input or output or both. They also describe formally the instantiation pattern to the call that makes the call to the built-ins determinate.

The mode annotations in the above example are '`+`' and '`-`'. Following is a complete description of the mode annotations you will find in the reference pages:

'`+`' Input argument. This argument will be inspected by the predicate, and affects the behavior of the predicate, but will not be further instantiated by the predicate. An exception is raised if the argument isn't of the expected type. Note that the type class of an input arguments might include *var*.

'`-`' Determinate output argument. This argument is unified with the output value of the predicate. An output argument is only tested to be of the same type as the possible output value, if the type is simple (see Section 18.1.4.1 [mpg-ref-aty-sim], page 988), and such testing is helpful to the user. Given the input arguments, the value of a determinate output argument is uniquely defined.

'`*`' Nondeterminate output argument. This argument is unified with the output value of the predicate. An output argument is only tested to be of the same type as the possible output value, if the type is simple (see Section 18.1.4.1 [mpg-ref-aty-sim], page 988), and such testing is helpful to the user. The predicate might be resatisfiable, and might through backtracking generate more than one output value for this argument.

'`+-`' An input argument that determinately might be further instantiated by the predicate. Since it is an input argument, an exception will be raised if it isn't in the expected domain.

'`+*`' An input argument that might be further instantiated by the predicate. The predicate might be resatisfiable, and might through backtracking generate more than one instantiation pattern for this argument. Since it is an input argument, an exception will be raised if it isn't in the expected domain.

If the synopsis of a predicate has more than one mode declaration, the first (the topmost) that satisfies *both* modes and types (of a goal instance), is the one to be applied (to that goal instance).

All built-in predicates of arity zero are determinate (with the exception of `repeat/0`).

For *input* arguments, an exception *will* be raised if the argument isn't of the specified type.

For *output* arguments, an exception *might* be raised if the argument is *nonvar*, and not of the specified type. The generated *value* of the argument *will* be of the specified type.

18.1.3 Predicate Categories

This section describes the categories of predicates and how they are indicated in the reference pages for predicates of each given category. The names of categories **hookable**, **hook**, **extendible**, **declaration**, and **meta-logical** appear to the right of the title of the reference page. The annotation **development** is used for predicates that are not available in runtime systems.

- *hookable*: The behavior of the predicate can be customized/redefined by defining one or more hooks. The mode and type annotations of a hookable predicate might not be absolute, since hooks added by the user can change the behavior.
- *hook*: The predicate is user defined, and is called by a *hookable* builtin. A hook must be defined in module user. For a hook, the mode and type annotations should be seen as guide-lines to the user who wants to add his own hook; they describe how the predicate is used by the system.
- *extendible*: A dynamic, multifile predicate, to which new clauses can be added by the user. For such a predicate, the mode and type annotations should be seen as guide-lines to the user who wants to extend the predicate; they describe how the predicate is used by the system.
- *declaration*: You cannot call these directly but they can appear in files as '`:- declaration`' and give information to the compiler. The goal template is preceded by '`:-`' in the **Synopsis**.

Meta-predicates and *operators* are recognizable by the implicit conventions described below.

- *Meta-predicates* are predicates that need to assume some module. A list of built-in predicates that do module name expansion is provided in Section 8.13.16 [ref-modmne], page 282. The reference pages of these predicates indicate which arguments are in a module expansion position by marking them as [MOD] in the **Arguments** field. That is, the argument can be preceded by a module prefix (an *atom* followed by a colon). For example:

  ```
  assert(mod:a(1), Ref)
  ```

  If no module prefix is supplied, it will implicitly be set to the calling module. If the module prefix is a variable, an instantiation error will be raised. If it is not an atom a type error will be raised. So in any meta-predicate reference page the following exceptions are implicit:

Exceptions

`instantiation_error`
: A module prefix is written as a variable.

`type_error`
: A module prefix is not an atom.

- Whenever the name of a built-in predicate is defined as *operator*, the name is presented in the **Synopsis** as an operator, for example

```
:- initialization +Goal                    (A)
+Term1 @> +Term2                           (B)
```

  It is thus always possible to see if a name is an operator or not. The predicate can, of course, be written using the canonical representation, even when the name is an operator. Thus (A) and (B) can be written as (C) and (D), respectively:

```
:- initialization(+Goal)                   (C)
@>(+Term1,  +Term2)                        (D)
```

18.1.4 Argument Types

The argument section describes the type/domain of each argument. If it is a '+' argument, then the built-in always tests if the argument is the right type/domain. In some cases, types/domains mentioned in the Arguments section need not be the smallest set of all acceptable arguments.

18.1.4.1 Simple Types

The simple argument types are those for which type tests are provided. They are summarized in Section 18.2.23 [mpg-top-typ], page 1004.

In addition there is *stream_object*, a special type of term described in Section 8.7.7.1 [ref-iou-sfh-sob], page 226.

If an output argument is given the type *var*, it means that that argument is not used by the predicate in the given instantiation pattern.

18.1.4.2 Extended Types

Following is a list of argument types that are defined in terms of the simple argument types. This is a formal description of the types/domains used in the Arguments sections of the reference pages for the built-ins. The rules are given in BNF (Backus-Naur form).

term ::= (any Prolog term) | *var* | *nonvar*

list ::= `[]` | `[`*term*`|`*list*`]`

list of Type ::= `[]` | `[`*Type*`|`*list of Type*`]`

one of [Element|Rest] ::= *Element* | *one of Rest*

arity ::= {An *integer X* in the range 0..255}

char ::= {An *integer X* in the range 1..255}

chars ::= `[]` | `[`*char*`|`*chars*`]`

pair ::= *term*`-`*term*

simple_pred_spec ::= *atom*`/`*arity*

pred_spec ::= *simple_pred_spec* | *atom*`:`*pred_spec*

pred_spec_tree ::= `[]` | *pred_spec* | `[`*pred_spec_tree*`|`*pred_spec_tree*`]`

pred_spec_forest ::= `[]` | *pred_spec*
| `[`*pred_spec_forest*`|`*pred_spec_forest*`]`
| *pred_spec_forest*`,`*pred_spec_forest*

gen_pred_spec ::= *atom* | *atom*`:`*gen_pred_spec*
| *simple_pred_spec*

gen_pred_spec_tree ::= `[]` | *gen_pred_spec*
| `[`*gen_pred_spec_tree*`|`*gen_pred_spec_tree*`]`

gen_pred_spec_tree_var ::= *gen_pred_spec_tree* {in which all atoms also can be variables}

extern_spec ::= *atom*
| *compound* {all arguments being *extern_arg*}

extern_arg ::= `+`*interf_arg_type* | `-`*interf_arg_type*

foreign_spec ::= *atom*
| *compound* {all arguments being *foreign_arg*}

foreign_arg ::= `+`*interf_arg_type* | `-`*interf_arg_type* | `[-`*interf_arg_type*`]`

interf_arg_type ::= `integer` | `float` | `single` | `double` | `atom`
| `term` | `string` | `string(`*integer*`)` | `address` | `address(`*atom*`)`

file_spec ::= *atom* | `atom(file_spec)`

expr ::= {everything that is accepted as second argument to `is/2`; see the description of arithmetic expressions in Section 8.8.4 [ref-ari-aex], page 235.}

18.1.5 Exceptions

The exceptions field of the reference page consists of a list of exception type names, each followed by a brief description of the situation that causes that type of exception to be raised. The following example comes from the reference page for `assert/[1,2]`:

Exceptions

`instantiation_error`
: If *Head* (in *Clause*) or *M* is uninstantiated.

`type_error`
: If *Head* is not of type callable, or if *M* is not an atom, or if *Body* is not a valid clause body.

For *input* arguments, an exception *will* be raised if the argument isn't of the specified type.

For *output* arguments, an exception *might* be raised if the argument is *nonvar*, and not of the specified type. The generated *value* of the argument *will* be of the specified type.

18.2 Topical List of Prolog Built-ins

Following is a complete list of Quintus Prolog built-in predicates, arranged by topic. A predicate may be included in more than one list.

18.2.1 Arithmetic

`Y is X` Y is the value of arithmetic expression X

`X =:= Y` the results of evaluating terms X and Y as arithmetic expressions are equal.

`X =\= Y` the results of evaluating terms X and Y as arithmetic expressions are not equal.

`X < Y` the result of evaluating X as an arithmetic expression is less than the result of evaluating Y as an arithmetic expression.

`X >= Y` the result of evaluating X as an arithmetic expression is not less than the result of evaluating Y as an arithmetic expression.

`X > Y` the result of evaluating X as an arithmetic expression X is greater than the result of evaluating Y as an arithmetic expression.

`X =< Y` the result of evaluating X as an arithmetic expression is not greater than the result of evaluating Y as an arithmetic expression.

18.2.2 Character I/O

`at_end_of_file`
testing whether end of file is reached for current input stream

`at_end_of_file(S)`
testing whether end of file is reached for the input stream S

`at_end_of_line`
testing whether at end of line on current input stream

`at_end_of_line(S)`
testing whether at end of line on input stream S

`get(C)` C is the next non-blank character on the current input stream

`get(S,C)` C is the next non-blank character on input stream S

`get0(C)` C is the next character on the current input stream

`get0(S,C)`
C is the next character on input stream S

`nl` send a newline to the current output stream

`nl(S)` send a newline to stream S

`peek_char(C)`
looks ahead for next input character on the current input stream

`peek_char(S,C)`
looks ahead for next input character on the input stream S

`put(C)` write character C to the current output stream

`put(S,C)` write character C to stream S

`skip(C)` skip input on the current input stream until after character C

`skip(S,C)`
skip input on stream S until after character C

`skip_line`
skip the rest input characters of the current line (record) on the current input stream

`skip_line(S)`
skip the rest input characters of the current line (record) on the input stream S

`tab(N)` send N spaces to the current output stream

`tab(S,N)` send N spaces to stream S

`ttyget(C)`
the next non-blank character input from the terminal is C

`ttyget0(C)`
the next character read in from the terminal is C

`ttynl` display a new line on the terminal

`ttyput(C)`
the next character sent to the terminal is C

`ttyskip(C)`
skip over terminal input until after character C

`ttytab(N)`
send N spaces to the terminal

18.2.3 Control

`P,Q` prove P and Q

`P;Q` prove P or Q

`M:P` call P in module M

`P->Q;R` if P succeeds, prove Q; if not, prove R

`P->Q` if P succeeds, prove Q; if not, fail

`!` cut any choices taken in the current procedure

`\+ P` goal P is not provable

`X ^ P` there exists an X such that P is provable (used in `setof` and `bagof`)

`bagof(X,P,B)`
B is the bag of instances of X such that P is provable

`call(P)` prove (execute) P

`fail` fail (start backtracking)

`false` same as fail

`findall(T,G,L)`
L is the list of all solutions T for the goal G

`otherwise`
same as true

`repeat` succeed repeatedly on backtracking

`setof(X,P,S)`
S is the set of instances of X such that P is provable

`true` succeed

18.2.4 Database

`abolish(F)`
: abolish the predicate(s) specified by F

`abolish(F,N)`
: abolish the predicate named F of arity N

`assert(C)`
: clause C (for dynamic predicate) is added to database

`assert(C,R)`
: clause C is asserted; reference R is returned

`asserta(C)`
: clause C is asserted before existing clauses

`asserta(C,R)`
: clause C is asserted before existing clauses; reference R is returned

`assertz(C)`
: clause C is asserted after existing clauses

`assertz(C,R)`
: clause C is asserted after existing clauses; reference R is returned

`clause(P,Q)`
: there is a clause for a dynamic predicate with head P and body Q

`clause(P,Q,R)`
: there is a clause for a dynamic predicate with head P, body Q, and reference R

`current_key(N, K)`
: N is the name and K is the key of a recorded term

`dynamic(P)`
: declaration that predicates specified by P are dynamic

`erase(R)` erase the clause or record with reference R

`instance(R,T)`
: T is an instance of the clause or term referenced by R

`multifile_assertz(C)`
: add clause C to the end of a (possibly compiled) multifile procedure

`recorda(K,T,R)`
: make term T the first record under key K; reference R is returned

`recorded(K,T,R)`
: term T is recorded under key K with reference R

`recordz(K,T,R)`
: make term T the last record under key K; reference R is returned

`retract(C)`
: erase the first dynamic clause that matches C

`retractall(H)`
: erase every clause whose head matches *H*

18.2.5 Debugging

`add_spypoint(P)`
: adds a spypoint to a procedure or to a particular call to a procedure

`add_advice(G,P,A)`
: associates advice with a port of *P*rolog predicate model

`check_advice`
: enables advice checking for all predicates with advice

`check_advice(P)`
: enables advice checking for the specified predicates

`current_advice(G,P,A)`
: find out what advice exists

`current_spypoint(L)`
: find out what spypoints exist

`debug` : switch on debugging

`debugging`
: display debugging status information

`get_profile_results(B,N,L,T)`
: get the results of the last execution profile

`leash(M)` : set the debugger's leashing mode to *M*

`nocheck_advice`
: disables all advice-checking

`nocheck_advice(P)`
: disables advice-checking from specified predicates

`nodebug` : switch off debugging

`noprofile`
: switch off profiling

`nospy(P)` : remove spypoints from the procedure(s) specified by *P*

`nospyall` : remove all spypoints

`notrace` : switch off debugging (same as `nodebug/0`)

`profile` : switch on profiling

`profile(G)`
: switch on profiling and profile the execution of goal *G*

`remove_advice(G,P,A)`
: remove advice from a port/predicate

`remove_spypoint`
: removes a spypoint

`show_profile_results`
: show the results of the last execution profile by time

`show_profile_results(B)`
: show the results of the last execution profile by B

`show_profile_results(B,N)`
: show the results of the last execution profile by B, listing N predicates

`spy(P)` set spypoints on the procedure(s) specified by P

`trace` switch on debugging and start tracing immediately

`unknown_predicate_handler(G,M,N)`
: user-defined handle for unknown predicates.

18.2.6 Executables and QOF-Saving

`initialization(G)`
: G is to be run when program is started

`runtime_entry(S)`
: entry point for a runtime system

`save_modules(L,F)`
: save the modules specifed in L into file F

`save_predicates(L,F)`
: save the predicates specified in L into file F

`version` display system identification messages

`version(A)`
: add the atom A to the list of introductory messages

`volatile(P)`
: predicates specified by P are not to be included in saves.

`save_program(F)`
: save all Prolog data

18.2.7 Execution State

`abort` abort execution of the program; return to current break level

`break` start a new break-level to interpret commands from the user

`halt` exit from Prolog

`on_exception(E,P,H)`
: specify a handler H for any exception E arising in the execution of the goal P

`raise_exception(E)`
: raise an exception

18.2.8 Filename Manipulation

`absolute_file_name(R,A)`
: A is the absolute name of file R

`absolute_file_name(R,O,A)`
: expand relative filename R to absolute file name A using options specified in O

`file_search_path(F,D)`
: user-extendible; directory D is included in file search path F

`library_directory(D)`
: user-extendible; D is a library directory that will be searched

18.2.9 File and Stream Handling

`character_count(S,N)`
: N is the number of characters read/written on stream S

`close(F)` close file or stream F

`current_input(S)`
: S is the current input stream

`current_output(S)`
: S is the current output stream

`current_stream(F,M,S)`
: S is a stream open on file F in mode M

`fileerrors`
: enable reporting of file errors

`flush_output(S)`
: flush the output buffer for stream S

`line_count(S,N)`
: N is the number of lines read/written on stream S

`line_position(S,N)`
: N is the number of characters read/written on the current line of S

`nofileerrors`
: disable reporting of file errors

`open(F,M,S)`
: file F is opened in mode M returning stream S

`open(F,M,O,S)`
: creates a Prolog stream S by opening the file F in mode M with options O

`open_null_stream(S)`
: new output to stream S goes nowhere

`prompt(O,N)`
queries or changes the prompt string of the current input stream

`prompt(S,O,N)`
queries or changes the prompt string of the current input stream or an input stream S

`see(F)` make file F the current input stream

`seeing(N)`
the current input stream is named N

`seek(S,O,M,N)`
seek to an arbitrary byte position on the stream S

`seen` close the current input stream

`set_input(S)`
select S as the current input stream

`set_output(S)`
select S as the current output stream

`stream_code(S,C)`
Converts between Prolog and C representations of a stream

`stream_position(S,P)`
P is the current position of stream S

`stream_position(S,O,N)`
O is the old position of stream S; N is the new position

`tell(F)` make file F the current output stream

`telling(N)`
to file N

`told` close the current output stream

`ttyflush` transmit all outstanding terminal output

18.2.10 Foreign Interface

`assign(A,V)`
poke into memory

`extern(P)`
declares predicate P to be callable from foreign code

`foreign(F,P)`
user-defined; C function F is attached to predicate P

`foreign(F,L,P)`
user-defined; function F in language L is attached to P

`foreign_file(F,L)`
user-defined; file F defines foreign functions in list L

`load_foreign_files(`*F*`,`*L*`)`
load object files *F* using libraries *L*

`unix(`*T*`)` give access to system commands

18.2.11 Grammar Rules

Head `-->` *Body*
A possible form for *Head* is *Body*

`'C'(`*S1*`,`*T*`,`*S2*`)`
(in grammar rules) *S1* is connected to *S2* by the terminal *T*

`expand_term(`*T*`,`*X*`)`
term *T* expands to term *X* using term_expansion/2 or grammar rule expansion

`phrase(`*P*`,`*L*`)`
list *L* can be parsed as a phrase of type *P*

`phrase(`*P*`,`*L*`,`*R*`)`
R is what remains of list *L* after phrase *P* has been found

`term_expansion(`*T*`,`*X*`)`
hook called by `expand_term/2`

18.2.12 Help

`help` prints some help information

`help(`*Topic*`)`
indexed access to the on-line manual

`manual` access the top level of the on-line manual

`manual(`*X*`)`
access the specified manual section

`user_help`
user-defined; tells `help/1` what to do

18.2.13 Hook Predicates

`foreign/[2,3]`
Describes the interface between Prolog and the foreign *Routine*

`foreign_file/2`
Describes the foreign functions in *ObjectFile* to interface to.

`message_hook/3`
Overrides the call to `print_message_lines/3` in `print_message/2`.

`generate_message_hook/3`
: A way for the user to override the call to `'QU_messages':generate_message/3` in `print_message/2`.

`portray/1`
: A way for the user to over-ride the default behavior of `print/1`.

`query_hook/6`
: Provides a method of overriding Prolog's default keyboard based input requests.

`runtime_entry/1`
: This predicate is called upon start-up and exit of stand alone applications.

`term_expansion/2`
: hook called by `expand_term/2`

`unknown_predicate_handler/3`
: User definable hook to trap calls to unknown predicates

`user_help/0`
: A hook for users to add more information when `help/0` is called.

18.2.14 List Processing

T `=..` *L* the functor and arguments of term T comprise the list L

`append(`*A*`,`*B*`,`*C*`)`
: the list C is the concatenation of lists A and B

`keysort(`*L*`,`*S*`)`
: the list L sorted by key yields S

`length(`*L*`,`*N*`)`
: the length of list L is N

`sort(`*L*`,`*S*`)`
: sorting the list L into order yields S

18.2.15 Loading Programs

`[`*F*`]` same as `load_files(`*F*`)`

`compile(`*F*`)`
: compile procedures from file (or list of files) F into the database

`consult(`*F*`)`
: same as compile(F)

`reconsult(`*F*`)`
: same as compile(F)

`ensure_loaded(`*F*`)`
: load F if not already loaded

`load_files(F)`
: load the specified Prolog source and/or QO*F* files *F* into memory

`load_files(F,O)`
: load files according to options *O*

`meta_predicate(P)`
: declares predicates *P* that are dependent on the module from which they are called

`module(M,L)`
: declaration that module *M* exports predicates in *L*

`multifile(P)`
: declares that the clauses for *P* are in more than one file

`no_style_check(A)`
: switch off style checking of type *A*

`restore(F)`
: restart system and load file *F*

`style_check(A)`
: turn on style checking of type *A*

`use_module(F)`
: import the module-file(s) *F*, loading them if necessary

`use_module(F,I)`
: import the procedure(s) *I* from the module-file *F*

`use_module(M,F,I)`
: import *I* from module *M*, loading module-file *F* if necessary

`term_expansion(T,N)`
: user-defined; compile-time transformation of clauses

18.2.16 Memory

`garbage_collect`
: force an immediate garbage collection

`garbage_collect_atoms`
: garbage collect atom space

`gc` : enable garbage collection

`nogc` : disable garbage collection

`statistics`
: display various execution statistics

`statistics(K,V)`
: the execution statistic with key *K* has value *V*

`trimcore` : reduce free stack space to a minimum

18.2.17 Messages

`message_hook(M,S,L)`
: user defined; intercept the printing of a message

`'QU_messages':generate_message(M,SO,S)`
: determines the mapping from a message term into a sequence of lines of text to be printed

`generate_message_hook(M,SO,S)`
: user defined; intercept message before it is given to `'QU_messages':generate_message/3`

`print_message(S,M)`
: print a message M of severity S

`print_message_lines(S,P,L)`
: print the message lines L to stream S with prefix P

`'QU_messages':query_abbreviation(T,P)`
: specifies one letter abbreviations for responses to queries from the Prolog system

`query_hook(Q,A)`
: user-defined; intercept a system request for user input

18.2.18 Modules

`current_module(M)`
: M is the name of a current module

`current_module(M,F)`
: F is the name of the file in which M's module declaration appears

`meta_predicate(P)`
: declares predicates P that are dependent on the module from which they are called

`module(M)`
: makes M the type-in module

`module(M,L)`
: declaration that module M exports predicates in L

`save_modules(L,F)`
: save the modules specifed in L into file F

`use_module(F)`
: import the module-file(s) F, loading them if necessary

`use_module(F,I)`
: import the procedure(s) I from the module-file F

`use_module(M,F,I)`
: import I from module M, loading module-file F if necessary

18.2.19 Program State

`current_atom(A)`
: backtrack through all atoms

`current_module(M)`
: M is the name of a current module

`current_module(M,F)`
: F is the name of the file in which M's module declaration appears

`current_predicate(A,P)`
: A is the name of a predicate with most general goal P

`listing` list all dynamic procedures in the type-in module

`listing(P)`
: list the dynamic procedure(s) specified by P

`module(M)`
: make M the type-in module

`predicate_property(P,Prop)`
: *Prop* is a property of the loaded predicate P

`prolog_flag(F,V)`
: V is the current value of Prolog flag F

`prolog_flag(F,O,N)`
: O is the old value of Prolog flag F; N is the new value

`prolog_load_context(K,V)`
: find out the context of the current load

`source_file(F)`
: F is a source file that has been loaded into the database

`source_file(P,F)`
: P is a predicate defined in the loaded file F

`source_file(P,F,N)`
: Clause number N of predicate P came from file F

18.2.20 Term Comparison

`compare(C,X,Y)`
: C is the result of comparing terms X and Y

`X == Y` terms X and Y are strictly identical

`X \== Y` terms X and Y are not strictly identical

`X @< Y` term X precedes term Y in standard order for terms

`X @>= Y` term X follows or is identical to term Y in standard order for terms

`X @> Y` term X follows term Y in standard order for terms

`X @=< Y` term X precedes or is identical to term Y in standard order for terms

18.2.21 Term Handling

`X = Y` terms X and Y are unified

`arg(N,T,A)`
the Nth argument of term T is A

`atom_chars(A,L)`
A is the atom containing the characters in list L

`copy_term(T,C)`
C is a copy of T in which all variables have been replaced by new variables

`functor(T,F,N)`
the principal functor of term T has name F and arity N

`ground(T)`
term T is a nonvar, and all substructures are nonvar

`hash_term(T,H)`
H is a hash-value for term T

`name(A,L)`
the list of characters of atom or number A is L

`number_chars(N,L)`
N is the numeric representation of list of characters L

`numbervars(T,M,N)`
number the variables in term T from M to N-1

`subsumes_chk(G,S)`
true when G(eneral) subsumes S(pecific); S and G are terms.

18.2.22 Term I/O

`current_op(P,T,A)`
atom A is an operator of type T with precedence P

`display(T)`
write term T to the user output stream in prefix notation

`format(C,A)`
write arguments A according to control string C

`format(S,C,A)`
write arguments A on stream S according to control string C

`op(P,T,A)`
make atom A an operator of type T with precedence P

`portray(T)`
hook, which is called when allows user to `print(T)`.

`portray_clause(C)`
: write clause C to the current output stream

`print(T)` display the term T on the current output stream using `portray/1` or `write/1`

`print(S,T)`
: display the term T on stream S using `portray/1` or `write/2`

`read(T)` read term T from the current input stream

`read(S,T)`
: read term T from stream S

`read_term(O,T)`
: read term T according to options O

`read_term(S,O,T)`
: read T from stream S according to options O

`write(T)` write term T on the current output stream

`write(S,T)`
: write term T on stream S

`write_canonical(T)`
: write term T on the current output stream so that it can be read back by `read/[1,2]`

`write_canonical(S,T)`
: write term T on stream S so that it can be read back by `read/[1,2]`

`writeq(T)`
: write term T on the current output stream, quoting atoms where necessary

`writeq(S,T)`
: write term T on stream S, quoting atoms where necessary

`write_term(T,O)`
: writes T to current output with options O

`write_term(S,T,O)`
: writes T to S according to options O

18.2.23 Type Tests

`atom(T)` term T is an atom

`atomic(T)`
: term T is an atom, a number or a db_reference

`callable(T)`
: T is an atom or a compound term

`compound(T)`
: T is a compound term (a skeletal predicate specification; see Section 8.1.7 [ref-syn-spc], page 169)

`db_reference(`*D*`)`
: *D* is a db_reference

`float(`*N*`)`
: *N* is a floating-point number

`ground(`*T*`)`
: term *T* is a nonvar, and all substructures are nonvar

`integer(`*T*`)`
: term *T* is an integer

`nonvar(`*T*`)`
: term *T* is one of atom, db_reference, number, compound (that is, *T* is instantiated)

`number(`*N*`)`
: *N* is an integer or a float

`simple(`*T*`)`
: *T* is not a compound term; it is either atomic or a var

`var(`*T*`)`
: term *T* is a variable (that is, *T* is uninstantiated)

18.3 Built-in Predicates

The following reference pages, alphabetically arranged, describe the Quintus Prolog built-in predicates.

For a functional grouping of these predicates including brief descriptions, see Section 18.2 [mpg-top], page 990.

In many cases the heading of a reference page will contain not only the name and arity of the predicate, but also the name of a major category to which the predicate belongs. These categories are defined in Section 18.1.3 [mpg-ref-cat], page 987.

Further information about categories of predicates and arguments, mode annotations, and the conventions observed in the reference pages is found in Section 18.1 [mpg-ref], page 985.

18.3.1 !/0

Synopsis

!

Cut.

Description

When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the parent goal will fail (the parent goal is the one that matched the head of the clause containing the cut).

See Also

Section 2.5.2.1 [bas-eff-cut-ove], page 33

18.3.2 ;/2 — disjunction

Synopsis

+*P* ; +*Q*

Disjunction: Succeeds if *P* succeeds *or* *Q* succeeds.

Arguments

P *callable* [MOD]
Q *callable* [MOD]

Description

This is normally regarded as part of the syntax of the language, but it is like a built-in predicate in that you can say `call` (*P* ; *Q*). The character '|' (vertical bar) can be used as an alternative to ';'.

Note that ;/2 has a distinct behaviour if the first argument is a ->/2 term. See ;/2 - if-then-else.

See Also

Section 2.5.8 [bas-eff-cdi], page 49 and Section 8.2.3 [ref-sem-dis], page 182

18.3.3 ,/2

Synopsis

+*P , +*Q

Conjunction: Succeeds if P succeeds *and* then Q succeeds.

Arguments

P callable [MOD]
Q callable [MOD]

Description

This is not normally regarded as a built-in predicate, since it is part of the syntax of the language. However, it is like a built-in predicate in that you can say `call`((*P* , *Q*)) to execute P and then Q.

See Also

Section 8.2.3 [ref-sem-dis], page 182

18.3.4 ;/2 — if-then-else

Synopsis

+-*P* -> +**Q* ; +**R*

If P then Q else R, using first solution of *P* only.

Arguments

P callable [MOD]
Q callable [MOD]
R callable [MOD]

Description

The character '|' can be used as an alternative to ';', giving the form:

```
P -> Q | R
```

The '|' is transformed into a ';' when the goal is read.

First *P* is executed. If it succeeds, then *Q* is executed, and if *Q* fails, the whole conditional goal fails. If *P* fails, however, *R* is executed instead of *Q*.

The operator precedences of the ';' and '->' are both greater than 1000, so that they dominate commas.

Backtracking

If *P* succeeds and *Q* then fails, backtracking into *P* does not occur. *P* may not contain a cut. '->' acts like a cut except that its range is restricted to within the disjunction: it cuts away *R* and any choice points within *P*. '->' may be thought of as a "local cut".

See Also

Section 2.5.8 [bas-eff-cdi], page 49 and Section 8.2.3 [ref-sem-dis], page 182

18.3.5 `->/2`

Synopsis

`+`*P* `->` `+*`*Q*

"Local cut"

If *P* then *Q* else fail, using first solution of *P* only.

Arguments

P *callable* [MOD]
Q *callable* [MOD]

Description

When occurring other than as the first argument of a disjunction operator ('`|`' or '`;`'), this is equivalent to:

```
P -> Q | fail.
```

(For a definition of *`P -> Q | R`*, see Section 8.2.7 [ref-sem-con], page 186.)

'`->`' cuts away any choice points in the execution of *P*

Note that the operator precedence of '`->`' is greater than 1000, so it dominates commas. Thus, in:

```
f :- p, q -> r, s.
f.
```

'`->`' cuts away any choices in `p` or in `q`, but unlike cut (`!`) it does not cut away the alternative choice for `f`.

Exceptions

`context_error`

18.3.6 =/2

Synopsis

+-*Term1* = +-*Term2*

unifies *Term1* and *Term2*.

Arguments

Term1 *term*
Term2 *term*

Description

This is defined as if by the clause '`Z = Z.`'.

If `=/2` is not able to unify *Term1* and *Term2*, it will simply fail.

18.3.7 =../2

Synopsis

+-*Term* =.. +-*List*

Unifies *List* with a list whose head is the atom corresponding to the principal functor of *Term* and whose tail is a list of the arguments of *Term*.

Arguments

Term term
: any term

List list
: and not empty

Description

Pronounced "univ".

If *Term* is uninstantiated, then *List* must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number.

Exceptions

`type_error`
`domain_error`
: *Term* is uninstantiated and *List* is not a proper list. *Term* is uninstantiated and the head of *List* is not atomic.

`instantiation_error`
: *Term* and *List* are both uninstantiated

`representation_error`
: *Term* is uninstantiated and *List* is longer than 256.

See Also

`functor/3`, `arg/3`

Section 8.9.2 [ref-lte-act], page 239

18.3.8 </2, =:=/2, =</2, =\=/2, >/2, >=/2

Synopsis

+*Expr1* < +*Expr2*

Evaluates *Expr1* and *Expr2* as arithmetic expressions. The goal succeeds if the result of evaluating *Expr1* is strictly *less than* the result of evaluating *Expr2*.

+*Expr1* =:= +*Expr2*

Succeeds if the results of evaluating *Expr1* and *Expr2* *are equal*.

+*Expr1* =< +*Expr2*

Succeeds if the result of evaluating *Expr1* is *less than or equal* to the result of evaluating *Expr2*.

+*Expr1* =\= +*Expr2*

Succeeds if the results of evaluating *Expr1* and *Expr2* are *not equal*.

+*Expr1* > +*Expr2*

Succeeds if the result of evaluating *Expr1* is strictly *greater than* the result of evaluating *Expr2*.

+*Expr1* >= +*Expr2*

Succeeds if the result of evaluating *Expr1* is *greater than or equal to* the result of evaluating *Expr2*.

Arguments

Expr1 expr
: Arithmetic expression

Expr2 expr
: Arithmetic expression

Description

All of these predicates evaluate *Expr1* and *Expr2* as arithmetic expressions and compare the results.

The possible values for *Expr* are spelled out in detail in Section 8.8.4 [ref-ari-aex], page 235.

Exceptions

`instantiation_error`
`type_error`
`representation_error`
`domain_error`
Expr1 or *Expr2* does not evaluate to a number.

Examples

```
| ?- 23 + 2.2  <  23 - 2.2.

yes
| ?- X = 31, Y = 25, X + Y  <  X - Y

no

| ?- 1.0 + 1.0 =:= 2.

yes

| ?- "a" =:= 97.

yes

| ?- 42 =< 42.

yes

| ?- "b" =< "a".

no

| ?- 7 =\= 14/2.

no

| ?- 7 =\= 15/2.

yes

| ?- "g" > "g".

no
```

```
| ?- 4*2 > 15/2.

yes

| ?- 42 >= 42.

yes

| ?- "b" >= "a".

yes
```

Comments

Note that the symbol '=<' is used here rather than '<=', which is used in some other languages. One way to remember this is that the inequality symbols in Prolog are the ones that cannot be thought of as looking like arrows. The '<' or '>' always points at the '='.

See Also

Section 8.8 [ref-ari], page 233

18.3.9 \+/1

Synopsis

\+ +*P*

Fails if the goal *P* has a solution, and succeeds otherwise.

Arguments

P callable [MOD]

Description

This is not real negation ("*P* is false"), which is not possible in Prolog, but negation-by-failure meaning "*P* is not provable". *P* may not contain a cut. The goal `\+ P` behaves exactly like

```
( P -> fail ; true)
```

Exceptions

```
type_error
context_error
```

Tip

Remember that with prefix operators such as this one, it is necessary to be careful about spaces if the argument starts with a '('. For example:

```
| ?- \+ (P, Q).
```

is the \+/1 operator applied to the conjunction of *P* and *Q*, but

```
| ?- \+(P, Q).
```

would require a predicate \+/2 for its solution. The prefix operator can, however, be written as a functor of one argument; thus

```
| ?- \+((P,Q)).
```

is also correct.

See Also

`library(not)` — defines a safer form of negation as failure.

18.3.10 ==/2, \==/2

Synopsis

+*Term1* == +*Term2*

Succeeds if the terms currently instantiating *Term1* and *Term2* are literally *identical* (in particular, variables in equivalent positions in the two terms must be identical).

+*Term1* \== +*Term2*

Succeeds if the terms currently instantiating *Term1* and *Term2* are *not* literally identical.

Arguments

Term1 term
Term2 term

Description

Query (A) fails because Term1 and Term2 are distinct uninstantiated variables. However, query (B) succeeds because the first goal unifies the two variables:

```
| ?- Term1 == Term2.                                    (A)

no

| ?- Term1 = Term2, Term1 == Term2.                     (B)

yes
```

Query (C) succeeds because Term1 and Term2 are distinct uninstantiated variables. However, query (D) fails because the first goal unifies the two variables.

```
| ?- Term1 \== Term2.                                   (C)

yes

| ?- Term1 = Term2, Term1 \== Term2.                    (D)

no
```

See also

`compare/3`, `@</2`, `@=</2`, `@>/2`, `@>=/2` Section 8.9 [ref-lte], page 238

18.3.11 `@</2`, `@=</2`, `@>/2`, `@>=/2`

Synopsis

+*Term1* `@<` +*Term2*

Succeeds if term *Term1* is *before* term *Term2* in the standard order.

+*Term1* `@=<` +*Term2*

Succeeds if term *Term1* is *not after* term *Term2* in the standard order.

+*Term1* `@>` +*Term2*

Succeeds if term *Term1* is *after* term *Term2* in the standard order.

+*Term1* `@>=` +*Term2*

Succeeds if term *Term1* is *not before* term *Term2* in the standard order.

Arguments

Term1 term
Term2 term

Description

These predicates use a standard total order when comparing terms. The standard total order is:

variables `@<` database references `@<` numbers `@<` atoms `@<` compound terms

For further details see Section 8.9.7.2 [ref-lte-cte-sot], page 242.

Examples

```
 | ?- foo(1) @< foo(2).

yes
| ?- chicken @< egg.

yes
| ?- a @< "a".

yes

| ?- liberty @=< pride.

yes
|?- 1 @=< 1.0.

yes

| ?- fie(1,1) @> fie(1).

yes

| ?- 1 @>= 1.0.

no
| ?- 1.0 @>= 1.

yes
```

See Also

`compare/3`, `==/2`, `\==/2` Section 8.9 [ref-lte], page 238

18.3.12 -->/2

Synopsis

+Head-->+Body

A possible form for *Head* is *Body*. Used primarily in grammar rules.

Arguments

Head term
: Prolog term, or list of terms

Body term
: Prolog term, or list of terms

The formal description of grammar heads and grammar bodies is spelled out in Section 8.1.8.3 [ref-syn-syn-sen], page 172.

Description

Head and *Body* are translated into lists of Prolog terms by `expand_term/2`.

Exceptions

`context_error`
: Cannot call `-->/2` as a predicate

Examples

See examples in Section 8.16.3 [ref-gru-exa], page 300 and Section 8.16.4 [ref-gru-tra], page 301.

See Also

`expand_term/2`, `'C'/3`, `term_expansion/2`, `phrase/[2,3]` Section 8.16 [ref-gru], page 298

18.3.13 `^/2`

Synopsis

`+`*X* `^` `+`**P*

Equivalent to "there exists an *X* such that *P* is true", thus *X* is normally an unbound variable. The use of the explicit existential quantifier outside `setof/3` and `bagof/3` is superfluous.

Arguments

X term

P callable [MOD]

Description

Equivalent to simply calling *P*

Exceptions

As for `call/1`:

```
type_error
context_error
instantiation_error
```

Examples

Using `bagof/3` without and with the existential quantifier:

```
| ?- bagof(X, foo(X,Y), L).

X = _3342,
Y = 2,
L = [1,1] ;

X = _3342,
Y = 3,
L = [2] ;

no

| ?- bagof(X, Y^foo(X,Y), L).

X = _3342,
Y = _3361,
L = [1,1,2] ;

no
```

Section 8.15.2.1 [ref-all-cse-equ], page 297

See Also

`setof/3`, `bagof/3`

18.3.14 `abolish/[1,2]`

Synopsis

`abolish(+PredSpecTree)`

`abolish(+Name, +Arity)`

Removes procedures from the Prolog database.

Arguments

PredSpecTree pred_spec_tree [MOD]
: A procedure specification in the form *Name/Arity*, or a list of such procedure specifications.

Name atom [MOD]
: A string representing the name of a predicate.

Arity arity
: An integer giving the arity of the predicate.

Description

Removes all procedures specified. After this command is executed the current program functions as if the named procedures had never existed. That is, in addition to removing all the clauses for each specified procedure, `abolish` removes any properties that the procedure might have had, such as being dynamic or multifile.

You may abolish any of your own procedures, regardless of whether they are dynamic, static, compiled, interpreted, or foreign. You cannot abolish built-in procedures.

It is important to note that `retract/1`, `retractall/1`, and `erase/1` can only remove dynamic predicates. They cannot remove the predicates properties (such as being dynamic or multifile) from the system. `abolish[1,2]`, on the other hand, can remove both static and dynamic predicates. It removes the clauses of the predicates and its properties.

The procedures that are abolished do not become invisible to a currently running procedure. See Section 8.14.7 [ref-mdb-exa], page 292 for an example illustrating this point.

Space occupied by abolished procedures is reclaimed. The space occupied by the procedures is reclaimed.

Procedures must be defined in the source module before they can be abolished. An attempt to abolish a procedure that is imported into the source module will cause a permission error. Using a module prefix, M:, clauses in any module may be abolished.

Abolishing a foreign procedure destroys only the link between that Prolog procedure and the associated foreign code. The foreign code that was loaded remains in memory. This is necessary because Prolog cannot tell which subsequently-loaded foreign files may have links to the foreign code. The Prolog part of the foreign procedure is destroyed and reclaimed.

Specifying an undefined procedure is not an error.

`abolish/2` is an obsolete special case of `abolish/1` maintained to provide compatibility with DEC-10 Prolog, C Prolog, and earlier versions of Quintus Prolog.

Exceptions

`instantiation_error`
: if one of the arguments is not ground.

`type_error`
: if a *Name* is not an atom or an *Arity* not an integer.

`domain_error`
: if a *PredSpec* is not a valid procedure specification, or if an *Arity* is specified as an integer outside the range 0-255.

`permission_error`
: if a specified procedure is built-in or imported into the source module.

See Also

`dynamic/1`, `erase/1`, `retract/1`, `retractall/1`.

Section 8.14.3 [ref-mdb-dre], page 288

18.3.15 `abort/0`

Synopsis

`abort`

Abandons the current execution and returns to the beginning of the current break level.

Description

Fairly drastic predicate that is normally only used when some error condition has occurred and there is no way of carrying on, or when debugging.

Often used via the debugging option `a` or the `^C` interrupt option `a`.

If `abort` is called from initialization (see Section 9.1.6 [sap-srs-eci], page 349), then `QP_initialize()` returns. If user has not defined their own `main()`, this means that `QP_toplevel()` gets called and you get the default top level loop. The same holds true for `^C` followed by the `a` option.

> **Please note:** The goal specified by `save_program/2` is also run as an `initialization` when a saved file is restored.
>
> **Please note:** In the context of a runtime system, `QP_toplevel()` corresponds to calling `runtime_entry(start)`.

Tip

Does not close any files that you may have opened. When using `see/1` and `tell/1`, (rather than `open/3`, `set_input/1`, and `set_output/1`), close files yourself to avoid strange behavior after your program is aborted and restarted.

See Also

`halt/[0,1]`, `break/0`, `QP_toplevel()`, `QP_initialize()`, `runtime_entry/1`

Section 8.11.1 [ref-iex-int], page 250

18.3.16 `absolute_file_name/[2,3]`

Synopsis

`absolute_file_name`(+*RelFileSpec*, -*AbsFileName*)

`absolute_file_name`(+*RelFileSpec*, +*Options*, -*AbsFileName*)

`absolute_file_name`(+*RelFileSpec*, +*Options*, **AbsFileName*)

Unifies *AbsFileName* with the the absolute filename that corresponds to the relative file specification *RelFileSpec*.

Arguments

RelFileSpec file_spec
: A valid file specification.

AbsFileName atom
: The first absolute filename derived from *RelFileSpec* that satisfies the access modes given by *Options*.

Options list
: A list of zero or more of the following:

`ignore_underscores(`*Bool*`)`
: *Bool* =

`true`
: When constructing an absolute filename that matches the given access modes, two names are tried: First the absolute filename derived directly from *RelFileSpec*, and then the filename obtained by first deleting all underscores from *RelFileSpec*.

`false`
: (default) Suppresses any deletion of underscores.

`extensions(`*Ext*`)`
: Has no effect if *RelFileSpec* contains a file extension. *Ext* is an atom or a list of atoms, each atom representing an extension that should be tried when constructing the absolute filename. The extensions are tried in the order they appear in the list. Default value is *Ext* = `['']`, i.e. only the given *RelFileSpec* is tried, no extension is added. To specify `extensions('')` or `extensions([])` is equal to not giving any extensions option at all.

`file_type(`*Type*`)`
: Has no effect if *RelFileSpec* contains a file extension. Picks an adequate extension for the operating system currently running, which

means that programs using this option instead of `extensions(`*`Ext`*`)` will be more portable between operating systems. *Type* must be one of the following atoms:

`text` implies `extensions([''])`.

RelFileSpec is a file without any extension. (Default)

`prolog` implies `extensions(['pl',''])`.

RelFileSpec is a Prolog source file, maybe with a '`.pl`' extension.

`object` implies `extensions(['o',''])`

(substitute the actual object file extension for '`o`') *RelFileSpec* is a foreign language object file, maybe with an extension.

`executable`
implies `extensions(['so',''])`

(substitute the actual shared object file extension for '`so`') *RelFileSpec* is a foreign executable (shared object file), maybe with an extension.

`qof` implies `extensions(['qof',''])`.

RelFileSpec is a Prolog object code file, maybe with a '`.qof`' extension.

`directory`
implies `extensions([''])`.

RelFileSpec is a directory, not a regular file. Only when this option is present can `absolute_file_name/3` access directories without raising an exception.

`foreign` implies `extensions(['o',''])`

(substitute the actual object file extension for '`o`') Same as `object`. Included for backward compatibility. Might be removed from future releases.

`ignore_version(`*`Bool`*`)`
This switch has no effect on systems where files don't have version numbers. *Bool* =

`true` When looking for a file that obeys the specified access modes, the version numbers will be ignored. The

returned absolute filename will not contain any version number. To find a filename that includes a proper version number, use `absolute_file_name/3` with the returned file as input, and the option `ignore_version(false)`.

See description of `access` option.

`false` (default) Version numbers are significant.

`access(`*Mode*`)`

Mode must be one of the following:

`read` *AbsFileName* must be readable.

`write` If *AbsFileName* exists, it must be writable. If it doesn't exist, it must be possible to create.

`append` If *AbsFileName* exists, it must be writable. If it doesn't exist, it must be possible to create.

`exist` The file represented by *AbsFileName* must exist.

`none` (default) The file system is not accessed. The first absolute filename that is derived from *RelFileSpec* is returned. Note that if this option is specified, no existence exceptions can be raised.

list of access modes

A list of one or more of the above options. If *AbsFile* exists, it must obey every specified option in the list. This makes it possible to combine a read and write, or write and exist check, into one call to `absolute_file_name/3`.

Please note: The following only applies to file systems with version numbered files. If the user explicitly has specified a version number, only that version is considered. If no version number is supplied, the version number of *AbsFileName* is determined by:

`read` newest readable version.

`write` the file exists, the newest version plus one. If the file doesn't exist, a system dependent "youngest" version number.

`append` If the file exists, the newest version. If the file doesn't exist, a system dependent "youngest" version number.

`exist` newest version.

`none` A system dependent "youngest" version number. Note that this can be switched of with the `ignore_version` option.

list of modes
: newest version.

`file_errors(`*Val*`)`
: *Val* =

`error`
: (default) Raise an exception if a file derived from *RelFileSpec* has the wrong permissions, that is, can't be accessed at all, or doesn't satisfy the the access modes specified with the `access` option.

`fail`
: Fail if a file derived from *RelFileSpec* has the wrong permissions. Normally an exception is raised, which might not always be a desirable behavior, since files that do obey the access options might be found later on in the search. When this option is given, the search space is guaranteed to be exhausted. Note that the effect of this option is the same as having the Prolog flag `fileerrors` set to `off`.

`solutions(`*Val*`)`
: i'Val' =

`first`
: (default) As soon as a file derived from *RelFileSpec* is found, commit to that file. Makes `absolute_file_name/3` determinate.

`all`
: Return each file derived from *RelFileSpec* that is found. The files are returned through backtracking. This option is probably most useful in combination with the option `file_errors(fail)`.

Description

We can restrict our description to `absolute_file_name/3`, because `absolute_file_name/2` can be defined as:

```
absolute_file_name(RelFileSpec, AbsFileName) :-
   Options = [ ignore_underscores(true),
               file_type(prolog),
               access(exist),
               file_errors(fail) ],
   (  absolute_file_name(RelFileSpec,
                         Options,
                         AbsFileName)
   -> true
   ;  absolute_file_name(RelFileSpec,
                         [],
                         AbsFileName)
   ).
```

Note that the semantics of `absolute_file_name/2` has changed slightly since previous releases. The difference is that `absolute_file_name/2` now always succeeds and returns an absolute filename, also when the *RelFileSpec* is a compound term. For instance, if the relative filename is `library(strings)`, you get an absolute filename, even if the library file '`strings.pl`' doesn't exist. In previous releases, this would have raised an exception. If it's important that an error is raised when the file doesn't exist, `absolute_file_name/3` with option `access(exist)` can be used.

Four phase process: The functionality of `absolute_file_name/3` is most easily described as a four phase process, in which each phase gets an infile from the preceding phase, and constructs one or more outfiles to be consumed by the succeeding phases. The phases are:

1. Syntactic rewriting
2. Underscore deletion
3. Extension expansion
4. Access checking

Each of the three first phases modifies the infile and produces variants that will be fed into the succeeding phases. The functionality of all phases but the first are decided with the option list. The last phase checks if the generated file exists, and if not asks for a new variant from the preceding phases. If the file exists, but doesn't obey the access mode option, a permission exception is raised. If the file obeys the access mode option, `absolute_file_name/3` commits to that solution, unifies *AbsFileName* with the filename, and succeeds determinately. For a thorough description, see below.

Note that the relative filename *RelFileSpec* may also be of the form PathAlias(*FileSpec*), in which case the absolute filename of the file *FileSpec* in one of the directories designated by *PathAlias* is returned (see the description of each phase below, and Section 8.6.1 [ref-fdi-fsp], page 205).

Description of each phase

(Phase 1) This phase translates the relative file specification given by *RelFileSpec* into the corresponding absolute filename.

If *RelFileSpec* is a term with one argument, it is interpreted as PathAlias(*FileSpec*) and outfile becomes the file as given by `file_search_path/2`. If `file_search_path/2` has more than one solution, outfile is unified with the solutions in the order they are generated. If the succeeding phase fails, and there is no more solutions, an existence exception is raised.

If *RelFileSpec* = `''`, outfile is unified with the current working directory. If `absolute_file_name/3` is called from a goal in a file being loaded, the directory containing that file is considered current working directory. If the succeeding phase fails, an existence exception is raised.

If *RelFileSpec* is an atom, other than `''`, it's divided into components. A component are defined to be those characters:

1. Between the beginning of the filename and the end of the filename if there are no '`/`'s in the filename.
2. Between the beginning of the filename and the first '`/`'.
3. Between any two successive '`/`'-groups (where a '`/`'-group is defined to be a sequence of one or more '`/`'s with no non-'`/`' character interspersed.)
4. Between the last '`/`' and the end of the filename.

To give the absolute filename, the following rules are applied to each component of *RelFileSpec*:

1. The component '`~user`', if encountered as the first component of *RelFileSpec*, is replaced by the absolute name of the home directory of *user*. If *user* doesn't exist, a permission exception is raised.
2. The component '`~`', if encountered as the first component of *RelFileSpec*, is replaced by the absolute name of the home directory of the current user.
3. If neither (1) nor (2) applies, then *RelFileSpec* is prefixed with the current working directory. If `absolute_file_name/3` is called from a goal in a file being loaded, the directory containing that file is considered current working directory.
4. The component '`.`' is deleted.
5. The component '`..`' is deleted together with the directory name syntactically preceding it. For example, '`a/b/../c`' is rewritten as '`a/c`'.
6. Two or more consecutive '`/`'s are replaced with one '`/`'.

When these rules have been applied, the absolute filename is unified with outfile. If the succeeding phase fails, an existence exception is raised.

(Phase 2) See `ignore_underscores` option.

(Phase 3) See `extensions` and `file_type` options.

(Phase 4) See `access` option.

Exceptions

`domain_error`
: *Options* contains an undefined option.

`instantiation_error`
: Any of the *Options* arguments or *RelFileSpec* is not ground.

`type_error`
: In *Options* or in *RelFileSpec*.

`existence_error`
: *RelFileSpec* is syntactically valid but does not correspond to any file.

`permission_error`
: *RelFileSpec* names an existing file but the file does not obey the given access mode.

Comments

If an option is specified more than once the rightmost option takes precedence. This provides for a convenient way of adding default values by putting these defaults at the front of the list of options.

Note that the default behavior of `absolute_file_name/3`, that is when *Options* = `[]`, does not mimic the behavior of `absolute_file_name/2`.

If `absolute_file_name/3` succeeds, and the file access option was one of `{`read, write, append`}`, it is guaranteed that the file can be opened with `open/[3,4]`. If the access option was `exist`, the file does exist, but might be both read and write protected.

Note that `absolute_file_name/3` signals a permission error if a specified file refers to a directory (unless the option `access(none)` is given.)

`absolute_file_name/[2,3]` is sensitive to the `fileerrors` flag, which causes the predicate to fail rather than raising permission errors when reading files with wrong permission. This has the effect that the search space always is exhausted.

directives in the extensions list. This causes the so specified access mode to be used as default access mode from there on, and the subsequently generated file names will thus be tried for this access mode, and not the default set by the access option. This can be particularly useful when used in combination with the `fileerrors` flag mentioned above. `}`

If *RelFileSpec* contains a '`..`' component, the constructed absolute filename might be wrong. This occurs if the parent directory is not the same as the directory preceding '`..`' in the relative file name, which only can happen if a soft link is involved.

Examples

To check whether the file '`my_text`' exists in the current user directory, with one of the extensions '`text`' or '`txt`', and is both writable and readable:

```
absolute_file_name('~/my_text',
                   [extensions(['text','txt']),
                    access([read,write])],
                   File).
```

To check if the Prolog file '`same_functor`' exists in some library, and also check if it exists under the name '`samefunctor`':

```
absolute_file_name(library(same_functor),
                   [file_type(prolog), access(exist),
                    ignore_underscores(true)],
                   File).
```

See Also

`file_search_path/2`, `library_directory/2` `nofileerrors/0`, `fileerrors/0`, `prolog_flag/[2,3]`

`library(files)` `library(directory)`

18.3.17 `add_advice/3` *development*

Synopsis

`add_advice`(+*Goal*,+*Port*,+*Action*)

Associate an action with entry to a port of a procedure.

Arguments

Goal callable [MOD]
: a term to be unified against a calling goal.

Port one of `[call,exit,done,redo,fail]`
: an atom indicating the port at which to check advice.

Action callable [MOD]
: a goal to be called when advice is checked at the given port.

Description

`add_advice/3` associates an advice action (a goal to be called) with a port of the standard Prolog debugger model (see Section 6.1.2 [dbg-bas-pbx], page 113). Variable bindings made when *Goal* matches the incoming call carry across to the advice action, so incoming arguments can be verified or processed by advice checking. Any number of advice actions can be associated with a given *Goal*, *Port*, or *Goal-Port* combination. Putting advice on a procedure does not automatically turn on checking of advice, so advice can be built into a program and checked only when necessary. At each port, advice is checked before interaction with the Prolog debugger, so the advice action can be used to control the debugger. It is not currently possible to associate advice with Prolog system built-in predicates.

Advice added using a call to `add_advice/3` will be checked after all preexisting advice for that predicate and port.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: if an argument is not sufficiently instantiated.

`type_error`
: if *Goal* or *Action* is not a callable, or a module prefix is not an atom, or *Port* is not an atom.

`domain_error`
: if *Port* is not a valid port.

`permission_error`
: if a specified procedure is built-in.

Tips

Using advice can streamline debugging of deep recursions and other situations where a given call is made correctly many times but eventually goes amiss. Use of the Prolog debugger's spypoints is inconvenient because of the many calls before the error. If, for instance, it is known that a certain bad datum is present in the particular call producing the error, it is possible to use advice to set a spypoint only when that datum is seen:

```
:- add_advice(recurse(X,Y), call,
      (bad_data(X), spy recurse/2)).
```

When advice checking is enabled, this piece of advice will take effect only if the first argument passed to `recurse/2` is bad. When that is so, a spypoint will be placed on `recurse/2` and execution will continue at the call port of `recurse/2`. Since advice is checked before debugger interaction at the port, the debugger will immediately stop. There is no need to interact with the debugger for all the calls that have valid data.

Advice can also provide a simple and flexible profiling tool by associating a counter with various ports of each "interesting" predicate. The advice associated with each port and predicate might map the name, arity, module and port to a counter value held in a dynamic table. When advice checking is on and an advised predicate port is reached, the advice action simply increments the counter. The counter table can then be inspected to determine the number of times each predicate-port combination was reached.

Advice can also be used to associate "pre-conditions" and "post-conditions" to predicates. "pre-conditions" can be associated with the "call" port of a predicate and "post-conditions" can be associated with the "done" or "exit" port of a predicate. Checking for "pre" and "post" conditions will be done only when checking advice is turned on.

See Also

`remove_advice/3`, `current_advice/3`, `check_advice/[0,1]`, `nocheck_advice/[0,1]`

Section 6.4 [dbg-adv], page 141

18.3.18 `add_spypoint/1` ***development***

Synopsis

`add_spypoint`*(+SpySpec)*

sets a spypoint on the specified predicate or call.

Arguments

SpySpec compound
: a specification of an individual spypoint. Two forms of *spyspec* are allowed:

  `predicate(`*Pred*`)`
  : A spypoint on any call to *Pred*. *Pred* must be a skeletal predicate specification, and may be module qualified.

  `call(`*Caller*`,`*Clausenum*`,`*Callee*`,`*Callnum*`)`
  : A spypoint on the *Callnum* call to *Callee* in the body of the *Clausenum* clause of *Caller*. *Callee* and *Caller* must be skeletal predicate specifications. *Callnum* and *Clausenum* must be integers, and begin counting from 1. Note that *Callnum* specifies a *lexical* position, that is, the number of the occurrence of *Callee* counting from the beginning of the body of the clause, and ignoring any punctuation.

Description

`add_spypoint/1` is used to set spypoints on predicates or on specific calls to predicates while debugging.

`add_spypoint/1` does not turn on the debugger. You have to explicitly turn on the debugger with a call to `debug/0` or `trace/0`.

You can add spypoints to predicates or calls that do not exist. If they later get defined the spypoints get placed.

Turning off the debugger does not remove spypoints. Use `remove_spypoint/1` to remove these spypoints.

If you are using QUI, the more convenient way to add these spypoints is to use the QUI based source debugger to select a particular goal or predicate and to use the Spypoints menu.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: *SpySpec* is not sufficiently instantiated.

`type_error`
: *SpySpec* is not a compound term.

`domain_error`
: *SpySpec* is not a `predicate/1` or `call/4` term.

See Also

`current_spypoint/1`, `remove_spypoint/1`, `spy/1`, `nospy/1`, `debugging/0`,

Section 6.1.1 [dbg-bas-bas], page 113.

18.3.19 append/3

Synopsis

`append`*(+*List1, +*List2, +*List3)*

True when all three arguments are lists, and the members of *List3* are the members of *List1* followed by the members of *List2*.

Arguments

List1 term a list

List2 term a list

List3 term a list consisting of *List1* followed by *List2*

Description

Appends lists *List1* and *List2* to form *List3*:

```
| ?- append([a,b], [a,d], X).

X = [a,b,a,d]

| ?- append([a], [a], [a]).

no
| ?- append(2, [a], X).

no
```

Takes *List3* apart:

```
| ?- append(X, [e], [b,e,e]).

X = [b,e]

| ?- append([b|X], [e,r], [b,o,r,e,r]).

X = [o,r]

| ?- append(X, Y, [h,i]).

X = [],
Y = [h,i] ;

X = [h],
Y = [i] ;

X = [h,i],
Y = [] ;

no
```

Backtracking

Suppose L is bound to a proper list (see Section 12.2.2 [lib-lis-prl], page 529). That is, it has the form [*T1*,. . .,*Tn*] for some *n*. In that instance, the following things apply:

1. `append(L, X, Y)` has at most one solution, whatever *X* and *Y* are, and cannot backtrack at all.
2. `append(X, Y, L)` has at most *n+1* solutions, whatever *X* and *Y* are, and though it can backtrack over these it cannot run away without finding a solution.
3. `append(X, L, Y)`, however, can backtrack indefinitely if *X* and *Y* are variables.

Examples

The following examples are perfectly ordinary uses of `append/3`:

To enumerate adjacent pairs of elements from a list:

```
next_to(X, Y, /*in*/ List3) :-
        append(_, [X,Y|_], List3).
```

To check whether Word1 and Word2 are the same except for a single transposition. (`append/5` in `library(lists)` would be better for this task.)

```
one_transposition(Word1, Word2) :-
        append(Prefix, [X,Y|Suffix], Word1),
        append(Prefix, [Y,X|Suffix], Word2).

| ?- one_transposition("fred", X).
X = "rfed" ;
X = "ferd" ;
X = "frde" ;
no
```

Given a list of words and commas, to backtrack through the phrases delimited by commas:

```
comma_phrase(List3, Phrase) :-
        append(F, [','|Rest], List3),
        !,
        (   Phrase = F
        ;   comma_phrase(Rest, Phrase)
        ).
comma_phrase(List3, List3).

| ?- comma_phrase([this,is,',',um,',',an,
                        example], X).
X = [this,is] ;
X = [um] ;
X = [an,example] ;
no
```

See Also

`length/2` Section 12.2 [lib-lis], page 528 `library(lists)`

18.3.20 `arg/3` *meta-logical*

Synopsis

`arg`(+*ArgNum*, +*Term*, -*Arg*)

unifies *Arg* with the *ArgNum*th argument of term *Term*.

Arguments

ArgNum integer
: positive integer

Term nonvar
: compound term

Arg term

Description

The arguments are numbered from 1 upwards.

Exceptions

`instantiation_error`
: if *ArgNum* or *Term* is unbound.

`type_error`
: if *ArgNum* is not an integer.

Example

```
| ?- arg(2, foo(a,b,c), X).

X = b
```

See Also

`functor/3`, `=../2`

Section 8.9.2 [ref-lte-act], page 239, Section 12.3.3 [lib-tma-arg], page 551

18.3.21 `assert/[1,2]`

Synopsis

These predicates add a dynamic clause, *Clause*, to the Prolog database. They optionally return a database reference in *Ref*:

`assert(+Clause)`

`assert(+Clause, -Ref)`

Undefined whether *Clause* will precede or follow the clauses already in the database.

`asserta(+Clause)`

`asserta(+Clause, -Ref)`

Clause will precede all existing clauses in the database.

`assertz(+Clause)`

`assertz(+Clause, -Ref)`

Clause will follow all existing clauses in the database.

Arguments

Clause callable [MOD]
: A valid dynamic Prolog clause.

Ref db_reference
: a database reference, which uniquely identifies the newly asserted *Clause*.

Description

Clause must be of the form:

```
        Head
or      Head :- Body
or      M:Clause
```

where *Head* is of type callable and *Body* is a valid clause body. If specified, *M* must be an atom.

`assert(Head)` means assert the unit-clause *Head*. The exact same effect can be achieved by `assert((Head :- true))`.

If *Body* is uninstantiated it is taken to mean `call(Body)`. For example, (A) is equivalent to (B):

```
| ?- assert((p(X) :- X)).                    (A)
| ?- assert((p(X) :- call(X))).              (B)
```

Ref should be uninstantiated; a range exception is signalled if *Ref* does not unify with its return value. This exception is signalled after the assert has been completed.

The procedure for *Clause* must be dynamic or undefined. If it is undefined, it is set to be dynamic.

If you want to write a term of the form '`Head :- Body`' as the argument to `assert/1`, you must put it in parentheses, because the operator precedence of the `:-/2` functor is greater than 1000 (see Section 8.1.5.3 [ref-syn-ops-res], page 167). For example, (C) will cause a syntax error; instead you should type (D):

```
| ?- assert(foo:-bar).                       (C)

| ?- assert((foo:-bar)).                     (D)
```

When an assert takes place, the new clause is immediately seen by any subsequent call to the procedure. However, if there is a currently active call of the procedure at the time the clause is asserted, the new clause is not encountered on backtracking by that call. See Section 8.14.7 [ref-mdb-exa], page 292 for further explanation of what happens when currently running code is modified.

Exceptions

`instantiation_error`
: if *Head* (in *Clause*) or *M* is uninstantiated.

`type_error`
: if *Head* is not of type callable, or if *M* is not an atom, or if *Body* is not a valid clause body.

`permission_error`
: if the procedure corresponding to *Head* is built-in or has a static definition.

`context_error`
: if a cut appears in the if-part of an if-then-else.

`range_error`
: if *Ref* does not unify with the returned database reference.

See Also

`abolish/[1,2]`, `dynamic/1`, `erase/1`, `multifile_assertz/1` `retract/1`, `retractall/1`, `clause/[2,3]`.

Section 8.14.3 [ref-mdb-dre], page 288

18.3.22 `assign/2`

Synopsis

`assign(+LHS, +Expr)`

Evaluates *Expr* as an arithmetic expression, and stores the value in the memory location and format given by *LHS*.

Arguments

LHS compound
: One of the following terms

- `integer_8_at(L_Exp)`
- `integer_16_at(L_Exp)`
- `unsigned_8_at(L_Exp)`
- `unsigned_16_at(L_Exp)`
- `integer_at(L_Exp)`
- `address_at(L_Exp)`
- `single_at(L_Exp)`
- `double_at(L_Exp)`

L_Exp expr
: a valid arithmetic expression

Expr expr A valid arithmetic expression

Description

Can be used to poke data directly into memory. Evaluates *L_Exp* in *LHS* and *Expr* as arithmetic expressions. The functor of the first argument describes the type of data to be stored: `integer_8_at/1` will store a signed 8 bit integer, `single_at/1` will store a single precision floating point number, etc. For more structured ways of doing this, see the Structs and Objects packages.

The only proper addresses that should be assigned to are ones that are obtained through the foreign interface. `assign/2` is a very primitive built-in and users should have only very rare occasions to use it. To directly access and change data structures represented in foreign languages (like C) users should look at using the Structs and Objects packages.

Both arguments can be unbound at compile time. But it is more efficient if *LHS* is bound at compile time. Also note that attempting to overwrite improper locations of memory can

cause "Segmentation faults" or "Bus errors" and overwriting Prolog memory can result in undesirable behaviour long after the assignment is done.

Exceptions

`instantiation_error`
: *LHS* or *Expr* is not ground

`type_error`
: *Expr* is not a proper arithmetic expression or

L_Exp in *LHS* is not a proper arithmetic expressions or

Expr is of a different type than what is specified by *LHS*

`domain_error`
: *LHS* is not one of the above listed terms

Examples

foo.c

```
static int counter;
int * init_counter()
{
        counter = 0;
        return &counter;
}
```

foo.pl

```
foreign(init_counter, c, init_counter([-address])).

get_counter(Counter, Count) :-
        Count is integer_at(Counter).

incr_counter(Counter) :-
        assign(integer_at(Counter),
        integer_at(Counter)+1).

| ?- init_counter(C), incr_counter(C), get_counter(C, Count1),
     incr_counter(C), incr_counter(C), get_counter(C, Count2).

C = 1418304,
Count1 = 1,
Count2 = 3

| ?-
```

See Also

Section 8.8.4 [ref-ari-aex], page 235 `library(structs)`, `library(objects)`

18.3.23 `at_end_of_file/[0,1]`

Synopsis

`at_end_of_file`

`at_end_of_file(+`*`Stream`*`)`

Tests whether end of file has been reached for the current input stream or for the input stream *Stream*.

Arguments

Stream *stream_object* a valid Prolog input stream

Description

`at_end_of_file/[0,1]` checks if end of file has been reached for the specified input stream. An input stream reaches end of file when all characters except the file border code (-1 by default) of the stream have been read. It remains at end of file after the file border code has been read.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: Some operating system dependent error occurred in reading.

Comments

`at_end_of_file/[0,1]` peeks ahead for next input character if there is no character available on the buffer of the specified input stream.

Coding with `at_end_of_file/[0,1]` to check for end of file condition is more portable among different operating systems than checking end of file by the character code (for example, peek_char(-1)).

See Also

`at_end_of_line/[0,1]`.

18.3.24 `at_end_of_line/[0,1]`

Synopsis

`at_end_of_line`

`at_end_of_line(+`*Stream*`)`

Test whether end of line (record) has been reached for the current input stream or for the input stream *Stream*.

Arguments

Stream stream_object
: a valid Prolog input stream

Description

`at_end_of_line/[0,1]` succeeds when end of line (record) is reached for the specified input stream. An input stream reaches end of line when all the characters except the line border code of the current line have been read.

`at_end_of_line/[0,1]` is also true whenever `at_end_of_file/[0,1]` is true.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: Some operating system dependent error occurred in reading.

Comments

Coding with `at_end_of_line/[0,1]` to check for end of line is more portable among different operating systems than checking end of line by the input character code.

See Also

`at_end_of_file/[0,1]`, `skip_line/[0,1]`, `get0/[1,2]`, `set_input/1`

18.3.25 `atom/1` *meta-logical*

Synopsis

`atom(+Term)`

Term is currently instantiated to an atom.

Arguments

Term term

Examples

```
| ?- atom(pastor).

yes
| ?- atom(Term).

no
| ?- atom(1).

no
| ?- atom('Time').

yes
```

See Also

`atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1 simple/1`

Section 8.1.2.4 [ref-syn-trm-ato], page 160

18.3.26 `atom_chars/2`

Synopsis

`atom_chars(+`*`Atom`*`, -`*`Chars`*`)`

`atom_chars(-`*`Atom`*`, +`*`Chars`*`)`

Chars is the list of ASCII character codes comprising the printed representation of *Atom*.

Arguments

Chars chars
: the list of ASCII character codes comprising the printed representation of *Atom*.

Atom atom
: will be instantiated to an atom containing exactly those characters, even if the characters look like the printed representation of a number.

Description

Initially, either *Atom* must be instantiated to an atom, or *Chars* must be instantiated to a proper list of character codes (containing no variables).

Any atom that can be read or written by Prolog can be constructed or decomposed by `atom_chars/2`.

Comment

If you deal with chars values often, you may find it useful to load `library(printchars)`. Once this is done, a list of character codes will be written by `print/1` as double-quoted text.

Exceptions

`instantiation_error`
`type_error`
`representation_error`

See Also

`print/1`, `library(printchars)`

18.3.27 atomic/1 *meta-logical*

Synopsis

`atomic(+`*Term*`)`

Succeeds if *Term* is currently instantiated to either an atom number or a db_reference.

Arguments

Term term

Example

```
| ?- atomic(9).

yes
| ?- atomic(a).

yes
| ?- atomic("a").

no
| ?- assert(foo(1), Ref), atomic(Ref).

Ref = '$ref'(1195912,1)
```

See Also

`atom/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1` `simple/1`

18.3.28 `bagof/3`

Synopsis

`bagof(+Template, +*Generator, *Set)`

Like `setof/3` except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. This relaxation saves time and space in execution.

Arguments

Template term
Generator callable [MOD]
: a goal to be proved as if by `call/1`.

Set list of term
: non-empty set

Examples

See `findall/3` for examples that illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`.

Exceptions

As for `call/1`, and additionally:

`resource_error`
: *Template* contains too many free variables.

See Also

`findall/3`, `setof/3`, `^/2`

Section 8.15 [ref-all], page 295

18.3.29 `break/0` *development*

Synopsis

`break`

causes the current execution to be interrupted; enters next break level.

Description

The first time `break/0` is called, it displays the message

```
% Beginning break level 1

| ?-
```

The system is then ready to accept input as though it were at top level. If another call to `break/0` is encountered, it moves up to level 2, and so on. The break level is displayed in the editor mode line when you are running under the editor interface; otherwise it is displayed on a separate line before each top-level prompt, as follows:

```
[1]
| ?-
```

To close a break level and resume the execution that was suspended, type the end-of-file character applicable on your system (default `^D`). `break/0` then succeeds, and execution of the interrupted program is resumed. Alternatively, the suspended execution can be abandoned by interrupting with a `^C` and using the `q` option.

Changes can be made to a running program while in a break level. Any change made to a procedure will take effect the next time that procedure is called. See Section 8.4.3 [ref-lod-rpx], page 191, for details of what happens if a procedure that is currently being executed is redefined. When a break level is entered, the debugger is turned off (although leashing and spypoints are retained). When a break level is exited, the debugging state is restored to what it was before the break level was entered.

Often used via the debugging option `b`.

This predicate is not supported in runtime systems.

See Also

`abort/0`, `halt/[0,1]`, `QP_toplevel()`

18.3.30 C/3

Synopsis

'C'(+-*List1*, +-*Terminal*, +-*List2*)

In a grammar rule: *Terminal* connects*List1* and *List2*. It is defined by the clause 'C'([T|S], T, S).

Arguments

List1 term

List2 term

Terminal term

Description

Analyzes *List1* into head and tail, and creates the tail, *List2*.

'C'/3 is not normally of direct use to the user. If its arguments are not of the expected form, it simply fails.

Examples

```
| ?- 'C'([the, slithy, toves, did, grob], Head, Tail).

Head = the,
Tail = [slithy,toves,did,grob] ;

no
```

See examples in Section 8.16.4 [ref-gru-tra], page 301.

See Also

Section 8.16 [ref-gru], page 298

18.3.31 `call/1`

Synopsis

`call(+*P)`

Proves(executes) *P*.

Arguments

P callable [MOD]

Description

If *P* is instantiated to an atom or compound term, then the goal `call(P)` is executed exactly as if that term appeared textually in its place, except that any cut (‘`!`’) occurring in *P* only cuts alternatives in the execution of *P*.

Exceptions

`instantiation_error`
: *P* is not instantiated enough.

`type_error`
: A module prefix is not an atom, or a goal is not a callable.

`context_error`
: A cut occurred in the if-part of an if-then-else.

`existence_error`
: An undefined predicate was called.

18.3.32 `callable/1` *meta-logical*

Synopsis

`callable(+`*Term*`)`

Term is currently instantiated to a term that `call/1` would take as an argument and not give a type error (an atom or a compound term).

Arguments

Term term

Examples

```
| ?- callable(a).

yes
| ?- callable(a(1,2,3)).

yes
| ?- callable([1,2,3]).

yes
| ?- callable(1.1).

no
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `nonvar/1`, `simple/1`

18.3.33 `character_count/2`

Synopsis

`character_count(+`*`Stream`*`, -`*`Count`*`)`

Obtains the total number of characters either input from or output to the open stream *Stream* and unifies it with *Count*.

Arguments

Stream stream_object
: a valid open Prolog stream

Count integer
: the resulting character count of the stream

Description

A freshly opened stream has a character count of 0. When a character is input from or output to a non-tty Prolog stream, the character count of the Prolog stream is increased by one. Character count for a tty stream reflects the total character input from or output to the tty since the tty is opened to any stream. See Section 8.7.8.1 [ref-iou-sos-spt], page 230, for details on the use of this predicate on a stream that is directed to the user's terminal.

A `nl/[0,1]` operation also increases the character count of a stream by one unless the line border code (end_of_line option in `open/4`) is less than 0.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226).

See Also

`line_count/2`, `line_position/2`, `stream_position/[2,3]`. Section 8.7 [ref-iou], page 214

18.3.34 `check_advice/[0,1]` *development*

Synopsis

`check_advice`

`check_advice(+`*`PredSpecs`*`)`

Enable advice checking.

Arguments

PredSpecs gen_pred_spec_tree [MOD]
: A list of predicate specifications.

Description

`check_advice/1` is used to enable advice checking on all predicates specified in *PredSpecs*. `check_advice/0` enables advice checking on all predicates for which advice has been added. When advice checking is enabled for a predicate, and execution of that predicate reaches an advised port, the term carrying the current instantiation of the Prolog call is unified against the goal term of the advice. If the two unify, the action associated with the goal and port is executed then failed over. If there are multiple pieces of advice associated with the goal and port, they are sequentially called and failed over.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: if the argument is not ground.

`type_error`
: if a *Name* is not an atom or an *Arity* not an integer.

`domain_error`
: if a *PredSpec* is not a valid procedure specification, or if an *Arity* is specified as an integer outside the range 0-255.

`permission_error`
: if a specified procedure is built-in.

Tips

`check_advice/0` behaves as though implemented by

```
check_advice :-
        current_advice(Goal, Port, Action),
        functor(Goal, Name, Arity),
        check_advice(Name/Arity),
        fail.
check_advice.
```

See Also

`add_advice/3`, `remove_advice/3`, `current_advice/3`, `nocheck_advice/[0,1]`

Section 6.4 [dbg-adv], page 141

18.3.35 clause/[2,3]

Synopsis

`clause(+*Head*, *Body*)`

`clause(-*Head*, -*Body*, +*Ref*)`

`clause(+*Head*, *Body*, *Ref*)`

Searches the database for a clause whose head matches *Head* and whose body matches *Body*.

Arguments

Head callable [MOD]
: a term whose functor names a dynamic procedure.

Body callable
: compound term or `true`

Ref db_reference
: a database reference

Description

In the case of unit-clauses, *Body* is unified with `true`.

If a procedure consists entirely of unit-clauses then there is no point in calling `clause/2` on it. It is simpler and faster to call the procedure.

In `clause/3`, either *Head* or *Ref* must be instantiated. If *Ref* is instantiated, `(`*Head* `:-` *Body* `)` is unified with the clause identified by *Ref*. (If this clause is a unit-clause, *Body* is unified with `true`.)

If the predicate did not previously exist, then it is created as a dynamic predicate and `clause/2` fails. If *Ref* is not instantiated, `clause/3` behaves exactly like `clause/2` except that the database reference is returned.

By default, clauses are accessed with respect to the source module.

Backtracking

Can be used to backtrack through all the clauses matching a given *Head* and *Body*. It fails when there are no (or no further) matching clauses in the database.

Exceptions

`instantiation_error`
: Neither *Head* nor *Ref* is instantiated.

`type_error`
: *Head* is not of type callable

: *Ref* is not a syntactically valid database reference.

`permission_error`
: Procedure is static (not dynamic).

`existence_error`
: *Ref* is a well-formed database reference but does not correspond to an existing clause or record.

Comments

If `clause/[2,3]` is called on an undefined procedure it fails, but before failing it makes the procedure dynamic. This can be useful if you wish to prevent unknown procedure catching from happening on a call to that procedure.

It is not a limitation that *Head* is required to be instantiated in `clause(`*Head*`, `*Body*`)`, because if you want to backtrack through all clauses for all dynamic procedures this can be achieved by:

```
| ?- predicate_property(P,(dynamic)), clause(P,B).
```

If there are clauses with a given name and arity in several different modules, or if the module for some clauses is not known, the clauses can be accessed by first finding the module(s) by means of `current_predicate/2`. For example, if the procedure is `f/1`:

```
| ?- current_predicate(_,M:f(_)), clause(M:f(X),B).
```

`clause/3` will only access clauses that are defined in, or imported into, the source module, except that the source module can be overridden by explicitly naming the appropriate module. For example:

```
| ?- assert(foo:bar,R).

R = '$ref'(771292,1)

| ?- clause(H,B,'$ref'(771292,1)).

no
| ?- clause(foo:H,B,'$ref'(771292,1)).

H = bar,
B = true

| ?-
```

Accessing a clause using `clause/2` uses first argument indexing when possible, in just the same way that calling a procedure uses first argument indexing. See Section 2.5.3 [bas-eff-ind], page 36.

See Also

`instance/2`, `assert/[1,2]`, `dynamic/1`, `retract/1`

Section 8.14.3 [ref-mdb-dre], page 288

18.3.36 `close/1`

Synopsis

`close(+`*Stream*`)`

closes the stream corresponding to *Stream*.

Arguments

Stream stream_object
: stream or file specification

Description

If *Stream* is a stream object, then if the corresponding stream is open, it will be closed; otherwise, `close/1` succeeds immediately, taking no action.

If *Stream* is a file specification, the corresponding stream will be closed, provided that the file was opened by `see/1` or `tell/1`.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`permission_error`
: File not opened by `see/1` or `tell/1`.

`domain_error`
: *Stream* is neither a filename nor a stream.

Examples

In this example, '`foo`' will be closed:

```
see(foo),
...
close(foo)
```

However, in this example, a permission error will be raised and '`foo`' will not be closed:

```
open(foo, read, S),
...
close(foo)
```

Here, `close(S)` should have been used.

See Also

`see/1`, `tell/1`, `open/[3,4]`, `write_canonical/[1,2]`

18.3.37 compare/3

Synopsis

`compare(-Order, +Term1, +Term2)`

succeeds if the result of comparing terms *Term1* and *Term2* is *Order*

Arguments

Order one of `[<,=,>]`

'='	if *Term1* is identical to *Term2*,
'<'	if *Term1* is before *Term2* in the standard order,
'>'	if *Term1* is after *Term2* in the standard order.

Term1 term
Term2 term

Description

The standard total order is as follows. For further details see Section 8.9.7.2 [ref-lte-cte-sot], page 242.

variables `@<` database references `@<` numbers `@<` atoms `@<` compound terms

The goal (A) is equivalent to (B):

```
| ?- compare(=, Term1, Term2).                    (A)

|?- (Term1 == Term2).                             (B)
```

The following query succeeds, binding *R* to `<`, because 1 comes before 2 in the standard order.

```
 | ?- compare(R, 1, 2).

R = <
```

If *Order* is supplied, and is not one of `<`, `>`, or `=`, `compare/3` simply fails.

See Also

`@</2`, `@=</2`, `@>/2`, `@>=/2`, `QP_compare()` Section 8.9 [ref-lte], page 238

18.3.38 `compile/1`

Synopsis

`compile(+`*Files*`)`

Compiles the specified Prolog source file(s) into memory.

Arguments

Files file_spec or *list of file_spec* [MOD]
: a file specification or a list of file specifications; a '`.pl`' extensions may be omitted in file specifications.

Description

Reads Prolog clauses from the specified file or files and adds them to the Prolog database, after first deleting any previous versions of the predicates they define. Clauses for a single predicate must all be in the same file unless that predicate is declared to be `multifile`.

If there are any directives in the file being loaded, that is, any terms with principal functor `:-/1` or `?-/1`, then these are executed as they are encountered.

When `compile/1` is called from an embedded command in a file being compiled by `qpc`, the specified files are compiled from source into QOF.

If desired, clauses and directives can be transformed as they are loaded. This is done by providing a definition for `term_expansion/2` (see `load_files/[1,2]`).

When `compile/1` is called in a runtime system, all predicates are loaded as dynamic predicates. The reason for this is that the compiler is not available in runtime systems.

This predicate is defined as if by:

```
compile(Files) :-
        load(Files, [load_type(compile)]).
```

For further details on loading files, see Section 8.4 [ref-lod], page 189.

Exceptions

Same as for `load_files/[1,2]`

See Also

`multifile/1`, `dynamic/1`, `no_style_check/1`, `style_check/1`, `nofileerrors/0`, `fileerrors/0`, `source_file/1`, `term_expansion/2`, `prolog_load_context/2`, `load_files/[1,2]`, `ensure_loaded/1`, `use_module/[1,2,3]`, `volatile/1`, `initialization/1`.

18.3.39 compound/1 *meta-logical*

Synopsis

`compound(+Term)`

Term is currently instantiated to a compound term.

Arguments

Term term

Examples

```
| ?- compound(9).

no
| ?- compound(a(1,2,3)).

yes
| ?- compound("a").

yes
| ?- compound([1,2]).

yes
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.40 `consult/1`

Synopsis

`consult(+Files)`

Same as `compile/1`

Arguments

Files file_spec or *list of file_spec* [MOD]

See Also

`compile/1`, `load_files/[1,2]`.

18.3.41 copy_term/2 *meta-logical*

Synopsis

`copy_term(+`*Term*`, -`*Copy*`)`

Makes a copy of `+`*Term* in which all variables have been replaced by new variables that occur nowhere outside the newly created term.

Arguments

Term term
Copy term

Description

- This is precisely the effect that would have been obtained from the definition below, although the system implementation of `copy_term/2` is more efficient.

```
copy_term(Term, Copy) :-
   recorda(copy, copy(Term), DBref),
   instance(DBref, copy(Temp)),
   erase(DBref),
   Copy = Temp.
```

- When you call `clause/[2,3]` or `instance/2`, you get a new copy of the term stored in the database, in precisely the same sense that `copy_term/2` gives you a new copy.
- Used in writing interpreters for logic-based languages.

Example

- A naive way to attempt to find out whether one term is a copy of another:

```
identical_but_for_variables(X, Y) :-
   \+ \+ (
      numbervars(X, 0, N),
      numbervars(Y, 0, N),
      X = Y
   ).
```

  This solution is sometimes sufficient, but will not work if the two terms have any variables in common.
- If you want the test to succeed even when the two terms do have some variables in common, you need to copy one of them; for example,

```
identical_but_for_variables(X, Y) :-
   \+ \+ (
      copy_term(X, Z),
      numbervars(Z, 0, N),
      numbervars(Y, 0, N),
      Z = Y
   ).
```

See Also

atomic/1, float/1, integer/1, nonvar/1, number/1, var/1, simple/1, compound/1, callable/1, ground/1, simple/1, db_reference/1.

18.3.42 current_advice/3 *development*

Synopsis

`current_advice(*Goal, *Port, *Action)`

Provides a means for checking what advice is present.

Arguments

Goal callable [MOD]
: any term.

Port one of `[call,exit,done,redo,fail]`
Action callable [MOD]
: any term.

Description

Unifies *Goal* with the goal term, *Port* with the port, and *Action* with the action term of currently existing user-defined advice. None of the three arguments need to be instantiated.

This predicate is not supported in runtime systems.

Tips

To backtrack through all the advice that exists for a predicate `mypred/2`, you can use the goal

```
| ?- current_advice(mypred(_,_), Port, Action).
```

If you are only interested in the advice for `mypred/2` on the call port, use

```
| ?- current_advice(mypred(_,_), call, Action).
```

To determine what predicates you told to call `format/2`, use

```
| ?- current_advice(Goal, Port, format(_,_)).
```

See Also

`add_advice/3`, `remove_advice/3`, `check_advice/[0,1]`, `nocheck_advice/[0,1]`

18.3.43 `current_atom/1` *meta-logical*

Synopsis

`current_atom(+`*Atom*`)`

`current_atom(*`*Atom*`)`

Atom is a currently existing atom.

Arguments

Atom atom

Backtracking

If *Atom* is uninstantiated, `current_atom/1` can be used to enumerate all known atoms. The order in which atoms are bound to *Atom* on backtracking corresponds to the times of their creation.

Comments

Note that the predicate `atom/1` is recommended for determining whether a term is an atom, as `current_atom/1` will succeed if *Atom* is uninstantiated as well.

See Also

`atom/1`

18.3.44 `current_input/1`

Synopsis

`current_input(-`*Stream*`)`

unifies *Stream* with the current input stream.

Arguments

Stream stream_object

See Also

`open/[3,4]`, `see/1`, `seeing/1`

Section 8.7.7.5 [ref-iou-sfh-cis], page 228

18.3.45 `current_key/2`

Synopsis

`current_key(*KeyName, *KeyTerm)`

Succeeds when *KeyName* is the name of *KeyTerm*, and *KeyTerm* is a recorded key.

Arguments

KeyName atomic
: either of:
  - *KeyTerm*, if *KeyTerm* is an atom or an integer; or
  - the principal functor of *KeyTerm*, if *KeyTerm* is a compound term.

KeyTerm nonvar
: is an integer, atom, or compound term, which is the key for a currently recorded term.

Description

If *KeyName* is not an atom, an integer, or an unbound variable, `current_key/2` fails. If *KeyTerm* is not a current key, `current_key/2` simply fails.

See Also

`recorda/3`, `recorded/3`, `recordz/3`,

18.3.46 current_module/[1,2]

Synopsis

`current_module(+ModuleName)`

`current_module(*ModuleName)`

Queries whether a module is "current" or backtracks through all of the current modules.

`current_module(+ModuleName, -AbsFile)`

`current_module(-ModuleName, +AbsFile)`

`current_module(*ModuleName, *AbsFile)`

Associates modules with their module-files.

Arguments

ModuleName *atom*
AbsFile *atom*
absolute filename

Description

A loaded module becomes "current" as soon as some predicate is defined in it, and a module can never lose the property of being current.

It is possible for a current module to have no associated file, in which case `current_module/1` will succeed on it but `current_module/2` will fail. This arises for the special module `user` and for dynamically-created modules (see Section 8.13.9 [ref-mod-dmo], page 276).

If its arguments are not correct, or if *Module* has no associated file, `current_module/2` simply fails.

Backtracking

`current_module/1` backtracks through all of the current modules. The following command will print out all current modules:

```
| ?- current_module(Module), writeq(Module), nl, fail.
```

`current_module/2` backtracks through all of the current modules and their associated files.

Exceptions

`type_error`

See Also

`module/1`, `module/2`

18.3.47 `current_output/1`

Synopsis

`current_output(-`*`Stream`*`)`

Unifies *Stream* with the current output stream.

Arguments

Stream stream_object

See Also

`open/[3,4]`, `tell/1`, `telling/1`

18.3.48 `current_op/3`

Synopsis

`current_op(+`*`Precedence`*`, +`*`Type`*`, +`*`Name`*`)`

`current_op(*`*`Precedence`*`, *`*`Type`*`, *`*`Name`*`)`

Succeeds when the atom *Name* is currently an operator of type *Type* and precedence *Precedence*.

Arguments

Precedence integer
: if instantiated, must be an integer in the range 1 to 1200.

Type one of `[xfx, xfy, yfx, fx, xf, yf]`
: if instantiated.

Name atom
: atom or a list of atoms if instantiated.

Description

None of the arguments need be instantiated at the time of the call; that is, this predicate can be used to find the precedence or type of an operator or to backtrack through all operators.

To add or remove an operator, use `op/3`.

Exceptions

`type_error`
: *Name* not an atom or *Type* not an atom or *Precedence* not an integer

`domain_error`
: *Precedence* not between 1-1200, or *Type* not one of listed atoms

See Also

`op/3`

Section 8.1.5 [ref-syn-ops], page 165.

18.3.49 current_predicate/2

Synopsis

`current_predicate(-`*`Name`*`, +`*`Term`*`)`

`current_predicate(*`*`Name`*`, *`*`Term`*`)`

Unifies *Name* with the name of a user-defined predicate, and *Term* with the most general term corresponding to that predicate.

Arguments

Name atom
Term callable [MOD]

Description

If you have loaded the predicates `foo/1` and `foo/3` into Prolog, `current_predicate/2` would return the following:

```
 | ?- current_predicate(foo, T).

T = foo(_116) ;

T = foo(_116,_117,_118) ;

no
```

Backtracking

- The following goal can be used to backtrack through every predicate in your program.

  ```
  | ?- current_predicate(Name, Module:Term).
  ```

- If a module is specified, `current_predicate/2` only succeeds for those predicates that are *defined* in the module. It fails for those predicates that are imported into a module.

  ```
  | ?- current_predicate(_, m:P).
  ```

  will backtrack through all predicates `P` that are defined in module `m`. To backtrack through all predicates imported by a module use `predicate_property/2` (see Section 8.10.1 [ref-lps-ove], page 245).

Tip

To find out whether a predicate is built-in, use `predicate_property/2`.

```
% Is there a callable predicate named gc?

| ?- current_predicate(gc, Term).

no
| ?- predicate_property(gc, Prop)

Prop = built_in
```

See Also

`predicate_property/2`

18.3.50 `current_spypoint/1` *development*

Synopsis

`current_spypoint(*Spyspec)`

Determines if there is currently a spypoint on a particular predicate or call, or enumerates all current spypoints.

Arguments

Spyspec compound
: can be any Prolog term. Prolog will try to unify it to terms of the form:

  `predicate(Pred)`
  : A spypoint on any call to *Pred*. *Pred* will be a skeletal predicate specification, and may be module qualified.

  `call(Caller,Clausenum,Callee,Callnum)`
  : A spypoint on the *Callnum* call to *Callee* in the body of the *Clausenum* clause of *Caller*. *Callee* and *Callnum* will be skeletal predicate specifications Section 6.1.4.2 [dbg-bas-tra-spy], page 117. *Callnum* and *Clausenum* will be integers, and begin counting from 1. Note that *Callnum* specifies a *lexical* position, that is, the number of the occurrence of *Callee* counting from the beginning of the body of the clause, and ignoring any punctuation.

Description

This predicate is not supported in runtime systems.

Backtracking

Can generate all current spypoints on backtracking.

See Also

`add_spypoint/1`, `remove_spypoint/1`, `spy/1`, `nospy/1`, `debugging/0`

Section 6.1.1 [dbg-bas-bas], page 113.

18.3.51 `current_stream/3`

Synopsis

`current_stream(-`*`AbsFile`*`, -`*`Mode`*`, +`*`Stream`*`)`

`current_stream(*`*`AbsFile`*`, *`*`Mode`*`, *`*`Stream`*`)`

Stream is a stream, which is currently open on file *AbsFile* in mode *Mode*.

Arguments

AbsFile atom
: absolute filename.

Mode one of `[read, write, append]`
Stream stream_object
: a term, which will be unifed with an open stream.

Description

- None of the arguments need be initially instantiated.
- Ignores the three special streams for the standard input, output, and error channels.

Backtracking

Can be used to backtrack through all open streams.

See Also

`open/[3,4]`, `see/1`, `tell/1`

Section 8.7.7.7 [ref-iou-sfh-bos], page 229

18.3.52 `db_reference/1` *meta-logical*

Synopsis

`db_reference(+`*Term*`)`

Term is a *db_reference*.

Arguments

Term *term*

See Also

`recorda/3`, `assert/2`, `atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.53 `debug/0` *development*

Synopsis

`debug`

Turns on the debugger in debug mode.

Description

`debug/0` turns the debugger on and sets it to debug mode. Turning the debugger on in debug mode means that it will stop at the next spypoint encountered in the current execution.

The effect of this predicate can also be achieved by typing the letter `d` after a `^c` interrupt (see Section 8.11.1 [ref-iex-int], page 250).

If you are runnning Prolog with QUI then `debug/0` will cause the debugger window to be popped open.

This predicate is not supported in runtime systems.

See Also

`spy/1`,`add_spypoint/1`,`trace/0`, `nodebug/0`

18.3.54 `debugging/0` *development*

Synopsis

`debugging`

Prints out current debugging state

Description

`debugging/0` displays information on the terminal about the current debugging state. It shows

- The top-level state of the debugger, which is one of

  `debug` The debugger is on but will not show anything or stop for user interaction until a spypoint is reached.

  `trace` The debugger is on and will show everything. As soon as you type a goal, you will start seeing a debugging trace. After printing each trace message, the debugger may or may not stop for user interaction: this depends on the type of leashing in force (see below).

  `zip` The debugger is on but will not show anything or stop for user interaction until a spypoint is reached. The debugger does not even keep any information of the execution of the goal till the spypoint is reached and hence you will not be able to see the ancestors of the goal when you reach the spypoint.

  `off` The debugger is off.

  The top-level state can be controlled by the predicates `debug/0`, `nodebug/0`, `trace/0`, `notrace/0` and `prolog_flag/3`.
- The type of leashing in force. When the debugger prints a message saying that it is passing through a particular port (one of Call, Exit, Head, Done, Redo, or Fail) of a particular procedure, it stops for user interaction only if that port is leashed. The predicate `leash/1` can be used to select which of the seven ports you want to be leashed.
- All the current spypoints. Spypoints are controlled by the predicates `spy/1`, `nospy/1`, `add_spypoint/1`, `remove_spypoint/1` and `nospyall/0`.

This predicate is not supported in runtime systems.

18.3.55 `discontiguous/1` *declaration*

Synopsis

`:- discontiguous +PredSpec`

Declares the clauses of the predicates defined by *PredSpecs* to be discontiguous in the source file (suppresses compile-time warnings).

Arguments

PredSpec gen_pred_spec_tree
: a skeletal predicate specification

Exceptions

`type_error`
`context_error`
: "declaration appeared in query"

See Also

Section 8.14.2 [ref-mdb-dsp], page 287.

18.3.56 `display/1`

Synopsis

`display(+`*Term*`)`

Displays *Term* on the standard output stream .

Arguments

Term term

Description

Ignores operator declarations and shows all compound terms in standard prefix form.

Tips

`display/1` is a good way of finding out how Prolog parses a term with several operators. `display(Term)` is equivalent to

```
write_term(Term, [quoted(false),ignore_ops(true)])
```

Output is not terminated by a full-stop; therefore, if you want the term to be acceptable as input to `read/[1,2]`, you must send the terminating full-stop to the output stream yourself. `display/1` does not put quotes around atoms and functors.

Example

```
| ?- display(a+b).
+(a,b)
yes

| ?- read(X), display(X), nl.
|: a + b * c.
+(a,*(b,c))

X = a+b*c

| ?-
```

See Also

`write/[1,2]`, `write_term/[2,3]`

18.3.57 `dynamic/1` *declaration*

Synopsis

`:-dynamic` *+PredSpecs*

Declares the predicates in *PredSpecs* to be dynamic.

Arguments

PredSpecs pred_spec_forest
: A single predicate specification of the form *Name/Arity*, or a sequence of predicate specifications separated by commas. *Name* must be an atom and *Arity* an integer in the range 0..255. [MOD]

Description

Exceptions

`type_error`
`context_error`
: If the declaration contradicts previous declaration or clauses for the same predicate in the file; or

  cannot call `dynamic/1` as a goal.

`permission_error`
: Cannot redefine built-in predicate, `dynamic/1`.

Comments

To declare a grammar rule *gram/n* dynamic, the arity of *PredSpec* must be *n*+2.

See Also

Section 8.14.2 [ref-mdb-dsp], page 287.

18.3.58 `ensure_loaded/1`

Synopsis

`ensure_loaded(+`*`Files`*`)`

Load the specified Prolog source and/or QOF file(s) into memory, if not already loaded and up to date.

Arguments

Files file_spec or *list of file_spec* [MOD]
: a file specification or a list of file specifications; a '`.pl`' or '`.qof`' extension may be omitted in a file specification.

Description

Loads each of the specified files except for files that have previously been loaded and that have not been changed since they were last loaded.

In the case of non-module-files, a file is not considered to have been previously loaded if it was loaded into a different module. For restrictions on non-module QOF-files, and how they can be loaded, see `load_files/[1,2]`. In this case the file is loaded again and a warning message is printed to let you know that two copies of the file have been loaded (into two different modules). If you *want* two copies of the file, you can avoid the warning message by changing the `ensure_loaded/1` command to a `compile/1` command. If you do not want multiple copies of the file, make the file a module-file.

When `ensure_loaded/1` is called from an embedded command in a file being compiled by `qpc`, each specified file is compiled from source into QOF unless there is already a QOF file that is more recent than the source.

When `ensure_loaded/1` is called in a runtime system, all predicates will be loaded as dynamic predicates and therefore this code will run slower. The reason for this is that the compiler is not available in runtime systems.

This predicate is defined as:

```
ensure_loaded(Files) :-
        load_files(Files, [if(changed),
                   load_type(latest)]).
```

For further details on loading files, see Section 8.4 [ref-lod], page 189.

Exceptions

`instantiation_error`
: *M* or *Files* is not ground.

`type_error`
: In *M* or in *Files*.

`existence_error`
: A specified file does not exist. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

See Also

`compile/1`, `load_files/[1,2]`.

18.3.59 `erase/1`

Synopsis

`erase(+Ref)`

Erases from the database the dynamic clause or recorded term referenced by *Ref*.

Arguments

Ref db_reference

Description

Erases from the database the dynamic clause or recorded term referenced by *Ref*. (Recorded terms are described in Section 8.14.8 [ref-mdb-idb], page 294.)

Ref must be a database reference to an existing clause or recorded term.

`erase/1` is not sensitive to the source module; that is, it can erase a clause even if that clause is neither defined in nor imported into the source module.

Exceptions

`instantiation_error`
: If *Ref* is not instantiated.

`type_error`
: If *Ref* is not a database reference.

`existence_error`
: if *Ref* is not a database reference to an existing clause or recorded term.

See Also

`abolish/[1,2]`, `assert/2`, `dynamic/1`, `retract/1`, `retractall/1`.

18.3.60 `expand_term/2` *hookable*

Synopsis

`expand_term(+`*Term1*`, -`*Term2*`)`

Transforms grammar rules into Prolog clauses before they are compiled. Normally called by the compiler, but can be called directly. The transform can be customized by defining the hook `term_expansion/2`.

Arguments

Term1 *term*
Term2 *term*

Description

Usually called by the built-in Load Predicates and not directly by user programs.

Normally used to translate grammar rules, written with `-->/2`, into ordinary Prolog clauses, written with `:-/2`. If *Term1* is a grammar rule, then *Term2* is the corresponding clause. Otherwise *Term2* is simply *Term1* unchanged.

If *Term1* is not of the proper form, or if *Term2* does not unify with its clausal form, `expand_term/2` simply fails.

Calls `term_expansion/2`.

Exceptions

Prints messages for exceptions raised by `term_expansion/2`.

Examples

See examples in Section 8.16.4 [ref-gru-tra], page 301.

See Also

`term_expansion/2`, `phrase/[2,3]`, `'C'/3`, `-->/2` Section 8.16 [ref-gru], page 298

18.3.61 `extern/1` *declaration*

Synopsis

`:-extern(+`*ExternSpec*`)`

Declares a Prolog predicate to be callable from functions written in other languages.

Arguments

ExternSpec extern_spec [MOD]
: a term of the form *Name*(*Argspec*, *Argspec*, ...)

  Name — the name of the Prolog predicate

  Argspec — an argument specification for each argument of the predicate. Each should be one of the following where *T* is a foreign type name.

```
+integer    +float    +single   +double
-integer    -float    -single   -double

+atom       +term     +string
-atom       -term     -string

+string     +address(T)
-string     -address(T)
```

Description

`extern/1` is used to make Prolog predicates callable from functions written in other languages. `extern/1` must appear as a compile-time declaration; furthermore, it may not appear in files loaded into runtime systems. The user has to declare as callable each Prolog predicate that is to be called from foreign functions. Any Prolog predicate can be declared to be callable from foreign functions, including system built-ins and predicates that do not currently have definitions. Predicates must be declared callable before they can actually be called from a function written in a foreign language.

Arguments are passed to the foreign interface as specified in *ExternSpec*:

'+' indicates that an argument is to be passed to Prolog from a foreign function.

'-' indicates that an argument is to be passed from Prolog to the foreign function.

Unlike the interface enabling Prolog to call functions written in other languages, when foreign functions call Prolog there are no return values or corresponding designators in *ExternSpec*.

When a Prolog predicate is declared to be callable, an interface predicate is created in the current module. The arity of the interface predicate is the same as that of the Prolog predicate. The name of the interface predicate is that of the Prolog predicate with an underscore prepended. The interface predicate is made available to the user as a hook to the "callability" of the Prolog predicate; for instance, the callability of the predicate can be saved by putting the interface predicate in a QOF file via `save_predicates/2`, then reloaded like any other predicate. The interface predicate can also be abolished like any other predicate; this also has the effect of making the previously callable Prolog predicate no longer available to foreign functions. A call to any interface predicate simply fails.

For more details about passing arguments from the foreign interface, see the chapter on the foreign language interface.

Exceptions

`instantiation_error`
: *ExternSpec* is uninstantiated

  Some *Argspec* in *ExternSpec* is uninstantiated or is a term that is insufficiently instantiated

`type_error`
: *ExternSpec* is instantiated but is not a callable term

  Some *Argspec* in *ExternSpec* is not a callable term

`domain_error`
: Some *Argspec* in *ExternSpec* is not one of the forms listed above

Examples

It can be quite useful to make the system built-in `call/1` available to foreign functions. Combined with term manipulation in C, doing so provides an evaluator for arbitrary Prolog queries. This can be done by loading a Prolog file containing the declaration

```
:- extern(call(+term)).
```

Prolog's `call/1` is then available to C via a function like

```
call_prolog(term)
QP_term_ref term;
{
    QP_pred_ref call;
    call = QP_predicate("call",1,"user");
    QP_query(call, term);
}
```

For the sake of brevity, this example does not check return values for failure or errors. Doing so is generally recommended. Of course, as is the case in Prolog, it is faster to call a Prolog predicate directly.

18.3.62 `fail/0`

Synopsis

`fail`

Always fails.

18.3.63 `false/0`

Synopsis

`false`

Same as `fail/0`.

18.3.64 `file_search_path/2` *extendable*

Synopsis

```
:- multifile file_search_path/2.

file_search_path(*PathAlias, *DirSpec)
```

Defines a symbolic name for a directory or a path. Used by predicates taking *file_spec* as input argument.

Arguments

PathAlias atom
: A string that represents the path given by *DirSpec*.

DirSpec file_spec
: Either a string giving the path to a file or directory, or *PathAlias(DirSpec)*, where *PathAlias* is defined by another `file_search_path/2` rule.

Description

`file_search_path/2` is a dynamic, multifile predicate. It resides in module `user`.

The `file_search_path` mechanism provides an extensible way of specifying a sequence of directories to search to locate a file. For instance, if a filename is given as a structure term, `library(basics)`. The principle functor of the term, `library`, is taken to be another `file_search_path/2` definition of the form

```
file_search_path(library, LibPath)
```

and file `basics` is assumed to be relative to the path given by *LibPath*. *LibPath* may also be another structure term, in which case another `file_search_path/2` fact gives its definition. The search continues until the path is resolved to an atom.

There may also be several definitions for the same *PathAlias*. Certain predicates, such as `load_files/[1,2]` and `absolute_file_name/[2,3]`, search all these definitions until the path resolves to an existing file.

There are several system defined search paths, such as `quintus`, `runtime`, `library`, `system`, `helpsys`. These are initialized at system startup, and used by some of the system predicates, but they may be redefined by the user. Furthermore, the user may create extra `file_search_path`s to define certain paths, and these may be used exactly as the predefined system paths. See Section 8.6 [ref-fdi], page 205 for more detail.

Examples

```
| ?- assert(file_search_path(home, '/usr/joe_bob')).

yes
| ?- assert(file_search_path(review, home('movie/review'))).

yes
| ?- compile(review(blob)).
% compiling /usr/joe_bob/movie/review/blob.pl
```

See Also

`absolute_file_name/[2,3]`, `assert/[1,2]`, `dynamic/1`, `library_directory/1`, `listing/1`, `load_files/[1,2]`,

Section 8.6 [ref-fdi], page 205.

18.3.65 `fileerrors/0`

Synopsis

`fileerrors`

Cancels the effect of `nofileerrors/0`.

Description

Sets the `fileerrors` flag to its default state, `on`, in which an exception is raised by `see/1`, `tell/1`, and `open/3` if the specified file cannot be opened.

The `fileerrors` flag is only disabled by an explicit call to `nofileerrors/0`, or via `prolog_flag/[2,3]`, which can also be used to obtain the current value of the `fileerrors` flag. See Section 8.10.1 [ref-lps-ove], page 245, for more information on the `fileerrors` flag.

See Also

`nofileerrors/0`, `prolog_flag/[2,3]`

18.3.66 `findall/3`

Synopsis

`findall(+Template, +*Generator, -List)`

Collects in *List* all the instances of *Template* for which the goal *Generator* succeeds. A special case of `bagof/3`, where all free variables in the generator are taken to be existentially quantified.

Arguments

Template term
Generator callable [MOD]
: a goal to be proved as if by `call/1`.

List list of terms

Description

A special case of `bagof/3`, where all free variables in the generator are taken to be existentially quantified, as if by means of the '`^`' operator.

Because `findall/3` avoids the relatively expensive variable analysis done by `bagof/3`, using `findall/3` where appropriate rather than `bagof/3` can be considerably more efficient.

Examples

To illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`:

```
| ?- [user].
| foo(1,2).
| foo(1,2).
| foo(2,3).
|
% user compiled in module user, 0.100 sec 352 bytes

yes
| ?- bagof(X, foo(X,Y), L).

X = _3342,
Y = 2,
L = [1,1] ;

X = _3342,
Y = 3,
L = [2] ;

no

| ?- bagof(X, Y^foo(X,Y), L).

X = _3342,
Y = _3361,
L = [1,1,2] ;

no

| ?- findall(X, foo(X,Y), L).

X = _3342,
Y = _3384,
L = [1,1,2] ;

no

| ?- setof(X, foo(X,Y), L).

X = _3342,
Y = 2,
L = [1] ;

X = _3342,
Y = 3,
L = [2] ;

no
```

Exceptions

As for `call/1`.

See Also

`setof/3`, `bagof/3`, `^/2`

Section 8.15 [ref-all], page 295

18.3.67 `float/1` *meta-logical*

Synopsis

`float(+`*Term*`)`

Term is currently instantiated to a float.

Arguments

Term term

Example

```
| ?- float(Term1).

no
| ?- float(5.2).

yes
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.68 `flush_output/1`

Synopsis

`flush_output(+`*`Stream`*`)`

Forces the buffered output of the stream *Stream* to be sent to the associated device.

Arguments

Stream stream_object
: a valid Prolog stream

Description

Sends the current buffered output of an output stream *Stream* to the actual output device, which is usually a disk or a tty device. `flush_output/1` fails if *Stream* does not permit flushing or the bottom layer flushing function of *Stream* is not properly defined.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`permission_error`
: An error occurred in flushing out the buffered output.

Comments

If the host operating system, such as UNIX, buffers an output file stream, the output may be written to the disk some time after `flush_output/1` succeeds.

See Also

`nl/[0,1]`, `QP_flush()`, `QU_flush_output()`, `ttyflush/0`

18.3.69 `foreign/[2,3]` ***hook***

Synopsis

```
:- discontiguous foreign/2, foreign/3.

foreign(*Routine, *ForeignSpec)

foreign(*Routine, *Language, *ForeignSpec)
```

Describes the interface between Prolog and the foreign *Routine*. The user has to define a `foreign/3` or the fact for every foreign function that is to be called from Prolog. Used by `load_foreign_files/2` and `load_foreign_executable/2`.

Arguments

Routine atom
: An atom that names a foreign code *Routine*

Language atom
: An atom that names the *Language* in which *Routine* is written. It must be one of `c`, `pascal` or `fortran`.

ForeignSpec foreign_spec
: A term of the form *PredName*(*Argspec*, *Argspec*, . . .) where:

  PredName the name of the Prolog predicate

  Argspec an argument specification (for each argument of the predicate) One of the following:

```
 +integer    +float    +single   +double
 -integer    -float    -single   -double
[-integer]  [-float]  [-single] [-double]

 +atom       +term     +string
 -atom       -term     -string
[-atom]     [-term]   [-string]

+string(N)    +address(T)
-string(N)    -address(T)
[-string(N)]  [-address(T)]
```

  where N is a positive integer and T is a foreign type name.

Description

`foreign/2` is a special case of `foreign/3` where *Language* is C. `foreign/2` is for backward compatibility.

The user has to define a `foreign/3` fact for every foreign function that is to be called from Prolog. Note that *Routine* does not have to be the same as *PredicateName*. Arguments are passed to the foreign function as specified in *ForeignSpecs*

`+`*type*	specifies that an argument is to be passed to the foreign function.
`-`*type*	specifies that an argument is to be received from the foreign function.
[-type]	argument is used to obtain the return value of a foreign function call. At most one "return value" argument can be specified.

For more details about the passing arguments through the foreign interface, see Section 10.4.2.1 [fli-ffp-ppc-api], page 415.

The `foreign/3` facts are used only in the context of a `load_foreign_files/2` command and can be removed once the foreign files are loaded.

If you have `foreign/3` facts in different files, Prolog will warn you that `foreign/3` has been previously defined in another file. This is generally not a problem if you are using the module system.

`load_foreign_files/2` will only look for `foreign/3` facts defined in its source module.

Exceptions

Errors in the specification of `foreign/3` will only be detected at `load_foreign_files/2` time. Otherwise defining a `foreign/3` fact is just like defining any other Prolog fact.

Tips

A good practice in loading several foreign files is to insert the call to `load_foreign_files/2` into the file that defines `foreign/3` as an embedded command. For example,

```
foreign(f11, c, f11(+integer)).
foreign(f12, c, f12(+atom, -atom)).
foreign_file('f1', [f11, f12]).
:- load_foreign_files('f1', []),
   abolish([foreign/3,foreign_file/2]).
```

Examples

`solve()` is a C function that takes three integer coeffecients of a quadratic equation and returns the two solutions. We assume that the solutions are not imaginary numbers.

a.pl

```
foreign(solve, c, solve(+integer, +integer, +integer,
                        -double, -double)).
foreign_file('a', [solve]).
:- load_foreign_files(['a'], ['-lm']),
   abolish([foreign/3, foreign_file/2]).
```

a.c

```
void solve(a, b, c, f1, f2)
    long int a, b, c;
    double *f1;
    double *f2;
    {
        *f1 = (-b + sqrt(b*b - 4*a*c)) / 2 * a;
        *f2 = (-b - sqrt(b*b - 4*a*c)) / 2 * a;
    }
```

See Also

`load_foreign_files/2`, `foreign_file/2`, `extern/1` Section 10.3 [fli-p2f], page 375

18.3.70 `foreign_file/2` *hook*

Synopsis

```
:- discontiguous foreign_file/2.

foreign_file(+ObjectFile, +ForeignFunctions)
```

Describes the foreign functions in *ObjectFile* to interface to. The user has to define a `foreign_file/2` fact for every object file that is to be loaded into Prolog.

Arguments

ObjectFile file_spec
: The foreign object file

ForeignFunctions list of atom
: A list of foreign function symbols that will be obtained from *ObjectFile*.

Description

The user has to define a `foreign_file/2` fact for every object file that is to be loaded into Prolog. The *ForeignFunctions* gives the list of foreign symbols that are to be found in the given object file. When a foreign file is loaded using `load_foreign_files/2`, Prolog looks for a `foreign_file/2` fact for that object file and finds the address of each symbol listed in the `foreign_file/2` fact. Prolog also expects a `foreign/3` definition for each symbol in the second argument of the `foreign_file/2` fact.

For more details about the foreign interface, see Section 10.3.2.2 [fli-p2f-uso-ffi], page 380.

The `foreign_file/2` facts are used only in the context of a `load_foreign_files/2` command and can be removed once the foreign files are loaded.

If you have `foreign_file/2` facts in different files, Prolog will warn you that `foreign_file/2` has been previously defined in another file.

`load_foreign_files/2` will only look for `foreign_file/2` facts defined in its source module.

Exceptions

Errors in the specification of `foreign_file/2` will only be detected when `load_foreign_files/2` is called. Otherwise defining a `foreign_file/2` fact is just like defining any other Prolog fact.

Examples

See example under `foreign/[2,3]`.

Tips

See Tip under `foreign/[2,3]`

See Also

`load_foreign_files/2`, `foreign/[2,3]`

18.3.71 format/[2,3]

Synopsis

`format(+Control, +Arguments)`

`format(+Stream, +Control, +Arguments)`

Interprets the *Arguments* according to the *Control* string and prints the result on the current or specified output stream.

Arguments

Stream stream_object
Control list of char or *atom*
: either an atom or a string, which can contain control sequences of the form '~<*n*><*c*>'

  <*c*> a format control option

  <*n*>

  is its optional argument.
  : <*n*> must be a non-negative integer.

  Any characters that are not part of a control sequence are written to the current output stream.

Arguments term
: list of arguments, which will be interpreted and possibly printed by format control options. If there is only one argument then this argument need not be enclosed in a list.

Description

Please note: In the case where there is only one argument and that argument is a list, then that argument must be enclosed in a list.

If <*n*> can be specified, then it can be the character '*'. In this case <*n*> will be taken as the next argument from *Arguments*.

The following control options cause formatted printing of the next element from *Arguments* to the current output stream. The argument must be of the type specified, or `format/1` will raise a consistency error.

`~<n>a` argument is an atom, which is printed without quoting. The maximum number of characters printed is <*n*>. If <*n*> is omitted the entire atom is printed.

```
| ?- format('~a', foo).
foo
```

`~<n>c` argument is a numeric ASCII code (0 =< code =< 127), which is printed <*n*> times. If <*n*> is omitted, it defaults to 1.

```
| ?- format('~2c', 97).
aa
```

`~<n>e` argument is a floating-point number, which is printed in exponential notation with precision <*n*>. The form of output is (in left-to-right order):

- an optional '`-`',
- a digit,
- a '`.`' if <*n*> is greater than 0,
- <*n*> digits,
- an '`e`',
- a '`+`' or a '`-`', and
- two or more digits.

If <*n*> is omitted, it defaults to 6.

```
| ?- format('~3e', 1.33333).
1.333e+00
```

See Section 8.8.1 [ref-ari-ove], page 233 for detailed information on precision.

Notes:

1. '`~<n>e`' coerces integers to floats
2. If n is greater than 60, only 60 digits will be printed.

`~<n>E` same as `~`<*n*>`e`, except '`E`' is used for exponentiation instead of '`e`'.

```
| ?- format('~3E', 1.33333).
1.333E+00
```

`~<n>f` argument is a floating-point number, which is printed in non-exponential format, with <*n*> digits to the right of the decimal point. If <*n*> is omitted, it defaults to 6. If <*n*> is equal to 0, no decimal point is printed.

```
| ?- format('~3f', 1.33333).
1.333
```

Notes:

1. '`~<n>f`' coerces integers to floats
2. If n is greater than 60, only 60 digits will be printed.

See the section on floating-point arithmetic for detailed information on precision.

`~<n>g` argument is a floating-point number, which is printed in either `~`<*n*>`e` or `~`<*n*>`f` form, whichever gives the best precision in minimal space, with the exception that no trailing zeroes are printed unless one is necessary immediately after the

decimal point to make the resultant number syntactically valid. At most <*n*> significant digits are printed. If <*n*> is omitted, it defaults to 6.

```
| ?- format('~g', 1000000000.0).
1e+09

| ?- format('~20g', 1000000000.0).
1000000000
```

See the section on floating-point arithmetic for detailed information on precision.

`~<n>G` same as `~<n>g`, except '`E`' is used for exponentiation instead of '`e`'.

```
| ?- format('~G', 1000000.0).
1E+06
```

`~<n>d` argument is an integer, which is printed as a signed decimal number, shifted right <*n*> decimal places. If <*n*> is omitted, it defaults to 0. If <*n*> is 0, the decimal point is not printed.

```
| ?- format('~d', 29).
29

| ?- format('~1d', 29).
2.9
```

`~<n>D` same as `~<n>d`, except that commas are inserted to separate groups of three digits to the left of the decimal point.

```
| ?- format('~1D', 29876).
2,987.6
```

`~<n>r` argument is an integer, which is printed in radix <*n*> (where 2 =< n =< 36) using the digits 0-9 and the letters `a-z`. If <*n*> is omitted, it defaults to 8.

```
| ?- format('~2r', 13).
1101

| ?- format('~r', 13).
15

| ?- format('~16r', 13).
d
```

`~<n>R` same as `~<n>r`, except it uses the digits 0-9 and the letters `A-Z` instead of `a-z`.

```
| ?- format('~16R', 13).
D
```

`~<n>s` argument is a string (list of numeric ASCII codes), from which at most the first <*n*> codes are printed as ASCII characters. If <*n*> is zero or if <*n*> is omitted, it defaults to the length of the string. If the string is shorter than <*n*> then all the ASCII codes that make up the string are printed.

```
| ?- format('~s', ["string"]).
string

| ?- format('~3s', ["string"]).
str

| ?- format('~a', "string").
! Consistency error: a and
  [115,116,114,105,110,103] are inconsistent
! the argument for the format control
  option "a" must be of type "atom".
! goal:  format('~a',
  [115,116,114,105,110,103])
```

The following control options can take an argument of any type:

`~i` argument is ignored.

```
| ?- format('~i', 10).
```

`~k` argument is passed to `write_canonical/[1,2]`.

```
| ?- format('~k', 'A'+'B').
+('A','B')
```

`~p` argument is passed to `print/[1,2]`.

```
| ?- asserta((portray(X+Y) :-
          write(X), write(' plus '),
          write(Y))).

| ?- format('~p', 'A'+'B').
A plus B
```

`~q` argument is passed to `writeq/[1,2]`.

```
| ?- format('~q', 'A'+'B').
'A'+'B'
```

`~w` argument is passed to `write/[1,2]`.

```
| ?- format('~w', 'A'+'B').
A+B
```

The following control options do not have a corresponding argument:

`~~` prints one '~'.

```
| ?- format('~~', []).
~
```

`~<n>n` prints <*n*> newline characters. If <*n*> is omitted, it defaults to 1.

```
| ?- format('begin~nend', []).
begin
end
```

`~N` prints nothing if at the beginning of a line, otherwise prints one newline character.

```
| ?- format('~Nbegin~N~Nend', []).
begin
end
```

The following control options manipulate column boundaries (tab positions). These column boundaries only apply to the line currently being written. A column boundary is initially assumed to be in line position 0.

`~<n>|` sets a column boundary at line position <*n*> and moves the cursor to that line position. If <*n*> is omitted, a column boundary is set at the current line position. See extended example below (also see Section 8.7.6.4 [ref-iou-cou-fou], page 223).

`~<n>+` sets a column boundary at <*n*> positions past the previous column boundary and moves the cursor to that line position. If <*n*> is omitted, it defaults to 8. See extended example below.

`~<n>t` When fewer characters are written between two column boundaries than the width of the column, the space remaining in the column is divided equally amongst all the '`~t`'s, if any, in the column, and each `~t` fills its allotted space with characters of ASCII code <*n*>. If <*n*> is omitted, it defaults to ASCII 32 (space). <*n*> can also be of the form '<*c*>, where <*c*> is the fill character. See extended example below.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`consistency_error`
: wrong number of arguments

`domain_error`
: wrong format option type

Examples

1. The following is an extended example of the use of `format/[2,3]` and the character escaping facility.

```
| ?- prolog_flag(character_escapes, _, on).

yes
| ?- compile(user).
| toc(Rel) :-
    format('Table of Contents ~t ~a~72|~*n', [i,3]),
    format('~tTable of Contents~t~72|~*n', 2),
    format("1. Documentation supplement for ~s~1f \c
      ~'.t ~d~72|~*n", ["Quintus Prolog Release ",Rel,2,2]),
    format("~t~*+~w Definition of the term \"loaded\" \c
      ~'.t ~d~72|~n", [3,1-1,2]),
    format("~t~*+~w Finding all solutions ~'.t ~d~72|~n", [3,1-2,3]),
    format("~t~*+~w Searching for a file in a library \c
      ~'.t ~d~72|~n", [3,1-3,4]),
    format("~t~*+~w New Built-in Predicates ~'.t ~d~72|~n", [3,1-
4,5]),
    format("~t~*+~w write_canonical (?Term) ~'.t ~d~72|~n", [7,1-4-
1,5]),
    format("~*+.~n~*+.~n~*+.~n", [20,20,20]),
    format("~t~*+~w File Specifications ~'.t ~d~72|~n", [3,1-7,17]),
    format("~t~*+~w multifile(+PredSpec) ~'.t ~d~72|~n", [7,1-7-
1,18]).
| ^D
% user compiled, 20.783 sec 4888 bytes

yes
| ?- toc(1.5).
                          Table of Contents

1. Documentation supplement for Quintus Prolog Re-
lease 1.5 ........... 2

   1-1 Defini-
tion of the term "loaded" ............................... 2
   1-2 Finding all solu-
tions ..................................... 3
   1-3 Searching for a file in a li-
brary ............................ 4
   1-4 New Built-in Predi-
cates ..................................... 5
       1-4-
1 write_canonical (?Term) ................................ 5
                    .
                    .
                    .
   1-7 File Specifica-
tions ..................................... 17
       1-7-1 multi-
file(+PredSpec) ................................ 18

yes
```

2. Misc. examples:

```
| ?- X=12, format('X =:= ~2d', X).     % These three
| ?- X=12, format("X=:= ~2d", X).      %  have the
| ?- X=12, format('X =:= ~*d', [2,X]). %  same results

| ?- format('~s', ["string"]).     % These two have
| ?- format('string', []).         % the same results

| ?- X=12, Y= 123, format('X = ~d, Y = ~d', [X,Y]).
```

See Also

`write_canonical/[1,2]`, `print/[1,2]`, `write/[1,2]` Section 8.7 [ref-iou], page 214

18.3.72 `functor/3` *meta-logical*

Synopsis

```
functor(+Term, -Name, -Arity)
```

```
functor(-Term, +Name, +Arity)
```

Succeeds if the principal functor of term *Term* has name *Name* and arity *Arity*.

Arguments

Term term
Name atom
Arity arity

Description

There are two ways of using this predicate:

1. If *Term* is initially instantiated, then
   - if *Term* is a compound term, *Name* and *Arity* are unified with the name and arity of its principal functor.
   - if *Term* is an atom or number, *Name* is unified with *Term*, and *Arity* is unified with 0.
2. If *Term* is initially uninstantiated, *Name* and *Arity* must both be instantiated, and
   - if *Arity* is an integer in the range 1..255, then *Name* must be an atom, and *Term* becomes instantiated to the most general term having the specified *Name* and *Arity*; that is, a term with distinct variables for all of its arguments.
   - if *Arity* is 0, then *Name* must be an atom or a number, and it is unified with *Term*.

Examples

```
| ?- functor(foo(a,b), N, A).

N = foo,
A = 2

| ?- functor(X, foo, 2).

X = foo(_1,_2)
```

Please note: `_1` and `_2` are anonymous variables. The term `foo(_1,_2)` is the "most general" term that has name foo and arity 2.

```
| ?- functor(X, 2, 0).

X = 2
```

Exceptions

`instantiation_error`
: *Term* and either *Name* or *Arity* are uninstantiated.

`type_error`
: *Name* is not atomic when *Arity* is 0, or *Arity* is not an integer.

`representation_error`
: *Term* is uninstantiated and *Arity* is an integer > 255.

See Also

`arg/3`, `name/2`, `=../2`

Section 8.9.2 [ref-lte-act], page 239

18.3.73 `garbage_collect/0`

Synopsis

`garbage_collect`

Explicitly invokes the garbage collector.

Description

This predicate invokes the garbage collector to reclaim data structures in the heap that are no longer accessible to the computation.

No expansion of the heap is done, even if `gc_margin` kilobytes cannot be reclaimed (see Section 8.10.4.1 [ref-lps-flg-cha], page 246). This means that calls to this predicate are effective only when the heap contains a significant amount of garbage.

The cut may be used in conjunction with `garbage_collect/0` to allow code that works in cycles and builds up large data structures to run for more cycles without running out of memory. The cut removes any alternatives that may be pending, thus potentially freeing up garbage that could not otherwise be collected.

Example

In the code fragment:

```
cycle(X) :- big_goal(X, X1), cycle(X1).
```

if `cycle/1` is to run for a long time, and if `big_goal/2` generates a lot of garbage, then rewrite the code like this:

```
cycle(X) :- big_goal(X, X1), !, garbage_collect, cycle(X1).
```

Tip

Use of the '`!, garbage_collect`' idiom is only desirable when you notice that your code does frequent garbage collections. It will allow the garbage collector to collect garbage more effectively, and the cycle will run without demanding increasing amounts of memory.

See Also

`gc/0`, `prolog_flag(gc_margin,_,_)`, `nogc/0`, `statistics/2`

18.3.74 `garbage_collect_atoms/0`

Synopsis

`garbage_collect_atoms`

Invokes the atom garbage collector.

Description

This predicate invokes the atom garbage collector to discard atoms that are no longer accessible to the computation, reclaiming their space.

Tips

A program can use the `atoms` keyword to `statistics/2` to determine if a call to `garbage_collect_atoms/0` would be appropriate.

See Also

`garbage_collect/0`, `statistics/2`

18.3.75 gc/0

Synopsis

gc

Enables the garbage collector.

Description

As if defined by:

```
gc :- prolog_flag(gc, _, on).
```

The garbage collection is enabled by default.

gc needs to be called only if the user has disabled te garbage collector by calling nogc or prolog_flag(gc,_,off).

18.3.76 'QU_messages':generate_message/3 *extendable*

Synopsis

```
:- multifile 'QU_messages':generate_message/3.

generate_message(+MessageTerm, -S0, -S)
```

For a given *MessageTerm*, generates a list composed of *Control-Arg* pairs and the atom `nl`. This can be translated into a nested list of *Control-Arg* pairs, which can be used as input to `print_message_lines/3`.

Arguments

MessageTerm term
: May be any term.

S0 list of pair
: the resulting list of *Control-Arg* pairs.

S list of pair
: the remaining list.

Description

Clauses for `generate_message/3` underly all messages from Prolog. They may be examined and altered. They are found in `messages(language('QU_messages'))`, which by default is `qplib('embed/english/QU_messages.pl')`.

The purpose of this predicate is to allow you to totally redefine the content of Prolog's messages. In particular, it is possible to translate all the messages from English into some other language.

This predicate should *not* be modified if all you want to do is modify or add a few messages: `user:generate_message_hook/3` is provided for that purpose.

The Prolog system uses the built-in predicate `print_message/2` to print all its messages. When `print_message/2` is called, it calls

```
user:generate_message_hook(Message,L,[])
```

to generate the message. If that fails,

```
'QU_messages':generate_message(Message,L,[])
```

is called instead.

If `generate_message/3` succeeds, L is assumed to have been bound to a list whose elements are either *Control-Args* pairs or the atom `nl`. Each *Control-Arg* pair should be such that the call

```
format(user_error, Control, Args)
```

is valid. The atom `nl` is used for breaking the message into lines. Using the format specification '`~n`' (new-line) is discouraged, since the routine that actually prints the message (see `user:message_hook/3` and `print_message_lines/3`) may need to have control over newlines.

`'QU_messages':generate_message/3` is not included by default in runtime systems, since end-users of application programs should probably not be seeing any messages from the Prolog system. If a runtime system does require the messages facility its source code should include a goal such as:

```
:- ensure_loaded(library(
        language('QU_messages'))).
```

If there is a call to `print_message/2` when `'QU_messages':generate_message/3` is undefined, or if `generate_message/3` fails for some reason, the message term itself is printed. Here is an example of what happens when `generate_message/3` fails.

```
| ?- print_message(error,unexpected_error(37)).
! unexpected_error(37)
```

`generate_message/3` failed because the message term was not recognized. In the following example `print_message/2` is being called by the default exception handler.

```
| ?- write(A,B).
! Instantiation error in argument 1 of write/2
! goal:  write(_2107,_2108)

| ?- abolish('QU_messages':generate_message/3).
...
| ?- write(A,B).
! instantiation_error(write(_2187,_2188),1)
```

Note that a call to `'QU_messages':generate_message/3` simply fails if the predicate is undefined; an `existence_error` is never signalled.

Examples

The following example shows how the output of `generate_message/3` is translated and passed to `print_message_lines/3`.

```
gen_message_and_print_lines(Msg, Stream, Prefix) :-
        generate_message(Msg, L, []),
        lines(L, Lines, []),
        print_message_lines(Stream, Prefix, Lines)

lines([]) --> [].
lines([H|T]) --> line(H), [nl], lines(T).

line([]) --> [].
line([Control-Args|T]) --> [Control-Args], line(T).
```

Errors

When `print_message/2` calls `'QU_messages':generate_message/3` it handles any exceptions that arise by printing out an error message. It then writes out the original message.

See Also

`print_message/2`, `message_hook/3`, `format/[2,3]`, `print_message_lines/3`, `user:generate_message_hook/3`, `QU_messages':query_abbreviation/2`

Section 8.20 [ref-msg], page 325

18.3.77 generate_message_hook/3 *hook*

Synopsis

```
:- multifile generate_message_hook/3.

generate_message_hook(+MessageTerm, -S0, -S)
```

A way for the user to override the call to `'QU_messages':generate_message/3` in `print_message/2`.

Arguments

MessageTerm term
: May be any term.

S0 list of pair
: the resulting list of *Control-Args* pairs.

S list of pair
: the remaining list.

Description

For a given *MessageTerm*, generates the list of *Control-Args* pairs required for `print_message_lines/3` to format the message for display.

This is the same as `'QU_messages':generate_message/3` except that it is a hook. It is intended to be used when you want to override particular messages from the Prolog system, or when you want to add some messages. If you are using your own exception classes (see `raise_exception`) it may be useful to provide generate_message_hook clauses for those exceptions so that the `print_message/2` (and thus the default exception handler that calls `print_message/2`) can print them out nicely.

The Prolog system uses the built-in predicate `print_message/2` to print all its messages. When `print_message/2` is called, it calls

```
user:generate_message_hook(Message,L,[])
```

to generate the message. If that fails,

```
'QU_messages':generate_message(Message,L,[])
```

is called instead.

If `generate_message_hook/3` succeeds, L is assumed to have been bound to a list whose elements are either *Control-Args* pairs or the atom `nl`. Each *Control-Args* pair should be such that the call

```
format(user_error, Control, Args)
```

is valid. The atom `nl` is used for breaking the message into lines. Using the format specification '`~n`' (new-line) is discouraged, since the routine that actually prints the message (see `user:message_hook/3` and `print_message_lines/3`) may need to have control over newlines.

It is recommended that you declare this predicate multifile when you define clauses for it so that different packages that define clauses for it can be used together.

Examples

- When a package is put in a module, it can still supply clauses like this:

```
:- multifile user:generate_message_hook/3.
user:generate_message_hook(hello_world) -->
        ['hello world'-[],nl].
```

  Note that the terminating `nl` is required.

Tips

See also:

`'QU_messages':generate_message/3`, `print_message/2`, `message_hook/3`, `format/[2,3]`, `print_message_lines/3`

18.3.78 `get/[1,2]`

Synopsis

`get(-Char)`

`get(+Stream, -Char)`

unifies *Char* with the ASCII code of the next non-layout character from Stream or the current input stream.

Arguments

Char char integer; legal ASCII code

Stream stream_object
valid Prolog input stream

Description

Layout characters are all outside the inclusive range 33..126; this includes space, tab, line-feed, delete, and all control characters.

If there are no more non-layout characters in the stream, *Char* is unified with -1.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
Trying to read beyond end of *Stream*

Comment

If the stream is tty, trying to read beyond the end of the stream results in resetting the input stream and trying to read the next character. By using the `eof_action` option of `open/[3,4]`, it is possible to specify that it should not be an error to run off the end of a stream.

See Also

`get0/[1,2]`, `ttyget/1`, `prompt/[2,3]`, `open/[3,4]`

18.3.79 `get0/[1,2]`

Synopsis

`get0(-Char)`

`get0(+Stream, -Char)`

Same as `get/[1,2]` except that *Char* includes layout characters.

Arguments

Char char

Stream stream_object

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: Trying to read beyond end of *Stream*

See Also

`get/[1,2]`, `ttyget0/1`, `prompt/[2,3]`

18.3.80 `get_profile_results/4` *development*

Synopsis

`get_profile_results(+By, +Num,-Results,-Total)`

Returns the results of the last profiled execution.

Arguments

By atom must be one of the atoms:

1. `by_time`
2. `by_choice_points`
3. `by_calls`
4. `by_redos`

Num integer
specifies the maximum length of the *Results* list

Results list of term
the results list

Total integer

Description

Returns profiling information accumulated from the last call to `profile/1`. The *By* argument specifies the *display mode*, which determines how the list is sorted and what the *Total* argument returns. The *Num* argument determines the maximum length of the *Results* list. This list is always sorted in descending order so that the top *Num* predicates are included in the list.

Results is a list of terms of the form `proc(Name,Ncalls,Nchpts,Nredos,Time,Callers)` where *Name,Ncalls,Nchpts,Nredos,Time* give call, choice point, redo counts and the execution time spent in milliseconds, and *Callers* is a list of `calledby(Time,Calls,Name,ClauseNo,CallNo)` terms, where *Time* in this case is the percentage of time attributed to this caller, *Calls* is the number of calls made from this caller and *Name, ClauseNo, CallNo* locate precisely the actual caller.

If the display mode is `by_time` then *Total* is the total execution time in milliseconds. If the display mode is `by_calls`, `by_choice_points` or `by_redos` then *Total* returns the total number of calls, choice points or redos respectively.

This predicate is not supported in runtime systems.

Example

```
| ?- get_profile_results(by_time,3,List,Total).

List = [proc(user:setof/3,227,0,0,1980,
             [calledby(61,152,user:satisfy/1,6,1),
              calledby(20,27,user:satisfy/1,7,1),
              calledby(18,48,user:seto/3,1,1)]),
        proc(user:satisfy/1,35738,36782,14112,260,
             [calledby(69,13857,user:satisfy/1,1,2),
              calledby(15,12137,user:satisfy/1,2,1)]),
        proc(user:write/1,2814,0,0,240,
             [calledby(33,481,user:reply/1,3,1),
              calledby(25,608,user:replies/1,3,1),
              calledby(16,562,user:out/1,2,1),
              calledby(8,203,user:reply/1,2,5),
              calledby(8,34,user:replies/1,2,3)])],
Total = 6040

[profile]
```

See Also

profile/[0,1,2,3], show_profile_results/[0,1,2]

18.3.81 ground/1 *meta-logical*

Synopsis

`ground(+`*Term*`)`

Term is currently instantiated to a term that is completely bound (has no uninstantiated variables in it).

Arguments

Term term

Examples

```
| ?- ground(9).

yes
| ?- ground(major(tom)).

yes
| ?- ground(a(1,Term,3)).

no
| ?- ground("a").

yes
| ?- ground([1,foo(Term)]).

no
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.82 `halt/[0,1]`

Synopsis

`halt`

`halt(+`*`ExitCode`*`)`

Causes an exit from Prolog.

Arguments

ExitCode integer
: an exit status code

Description

causes an exit from Prolog

`halt/0` exits with a "success" exit status (0).

`halt/1` exits with the exit status given by its *ExitCode* argument.

Exceptions

`instantiation_error`
`type_error`
: *N* is not an integer.

See Also

`abort/0`, `break/0`

Section 8.11.1 [ref-iex-int], page 250

18.3.83 hash_term/2

Synopsis

`hash_term(+Term, -HashValue)`

Provides an efficient way to calculate an *integer* hash value for the ground term *Term*.

Arguments

Term term
HashValue term
is an integer or variable

Description

If the first argument passed to `hash_term/2` is ground, an integer hash value corresponding to that term is calculated and returned in the second argument. If the first argument is not ground, a new variable is returned in the second argument.

For example:

```
 | ?- hash_term(foo(name,2,module), H).

H = 1391

| ?- hash_term(foo(X), H).

X = _4734,
H = _4755

| ?-
```

Tips

`hash_term/2` is provided primarily as a tool for the construction of sophisticated Prolog clause access schemes. Its intended use is to generate hash values for ground terms that will be used with first argument clause indexing, yielding compact and efficient multi-argument or deep argument indexing.

`hash_term/2` is most easily used when a known pattern of access to a predicate is desired and both arguments of the call and arguments of the predicate are known to be ground.

In the following simple but typical example, `hash_term/2` calls are used together with Prolog's database manipulation predicates (`assert/1` and `clause/2`) to calculate and add an additional argument to the clauses actually stored in the Prolog database:

```
add_pred_info(Name, Arity, Module, Info) :-
        hash_term([Name,Arity,Module], Hash),
        assert(info(Hash,Name,Arity,Module,Info)).

get_pred_info(Name, Arity, Module, Info) :-
        hash_term([Name,Arity,Module], Hash),
        clause(info(Hash,Name,Arity,Module,Info), _).
```

This example assumes that the name, arity and module to be stored in the Prolog database are ground when `add_pred_info/4` is called, and that they are also ground when `get_pred_info/4` is called. The predicate that is actually asserted, `info/5`, has an additional argument calculated by `hash_term/2`; `info/5` would not normally be called directly. A predicate using `hash_term/2` to delete the stored information would also be straightforward.

If the first argument passed to `hash_term/2` is not ground, `hash_term/2` returns a variable. Thus, if `add_pred_info/4` is called with the name, arity or module not ground, the `info/5` information will be asserted with a variable as its first argument, so it will not be indexed. If `get_pred_info/4` is called with the name, arity or module not ground, `info/5` will simply be searched sequentially. Prolog's normal semantics will be retained, although access will be considerably less efficient.

It is possible to use `hash_term/2` in more complex indexing schemes as well by checking instantiation when adding, accessing, and deleting clauses; however, it is up to the user to ensure appropriate instantiation patterns in calls. The tradeoff between run-time argument checking and reduced indexing effectiveness depends on the degree of discrimination otherwise afforded by normal first argument indexing. The efficiency gained by fast multi-argument indexing can often more than make up for such additional run-time costs.

It is also possible to use such indexing techniques on compiled predicates using term expansion. Note that calculated hash values are not dependent on transitory information like atom numbers or internal pointers. Hash values are consistent across saving and restoring or multiple invocations of an application.

Calculation of hash values is very fast, and indices constructed using the techniques sketched above are also very compact, as the only additional cost is for storing the additional (hash value) argument. When a solution to a complex indexing problem can be constructed using `hash_term/2` it will probably be preferable to solutions using other techniques.

18.3.84 `help/[0,1]` *hookable,development*

Synopsis

`help`

Gives basic information, such as how to start using the help system and how to exit from Prolog.

`help(+`*`Topic`*`)`

Displays help available on *Topic*.

Arguments

Topic atom

Description

`help(`*`Topic`*`)` is the basic help command. It attempts to accept any argument you give it as a topic for which help may exist in the manual. The argument is converted into a character string, and all the index entries that start with that string are combined into a menu, which gives you a choice of entry points into the manual hierarchy.

It is not necessary to type the whole of the word that is the topic you want information about. However, the fewer characters you type, the larger the menu is likely to be, because more index entries will contain with that character sequence.

A hook is provided so that users can add to or replace this information: `help/0` first calls `user_help/0`, and if that succeeds `help/0` does nothing else. Only if the call to `user_help/0` fails is the standard information displayed.

With the emacs interface see Section 4.2 [ema-emi], page 88

With QUI see Section 3.2 [qui-mai], page 55

This predicate is not supported in runtime systems.

See Also

`user_help/0`, `manual/[0,1]`

18.3.85 `initialization/1` *declaration*

Synopsis

`:- initialization` *Goal*

Declares that *Goal* is to be run when the file in which the declaration appears is loaded into a running system, or when a stand-alone program or runtime system that contains the file is started up.

Arguments

Goal callable [MOD]
A valid goal.

Description

Defined as built-in prefix operator, so a simplified syntax can be used when using `initialization/1` as a directive. See examples.

Callable at any point during compilation of a file. That is, it can be used as a directive, or as part of a goal called at compile-time. The initialization goal will be run as soon as the loading of the file is completed. That is at the end of the load, and notably after all other directives appearing in the file have been run.

`qpc` and `save_program/[1,2]` save initialization goals in the QOF file, so that they will run when the qof file is loaded.

Goal is associated with the file loaded, and with a module, if applicable. When a file, or module, is going to be reloaded, all goals earlier installed by that file or in that module, are removed. This is done before the actual load, thus allowing a new initialization *Goal* to be specified, without creating duplicates.

Exceptions

`instantiation_error`
The argument *Goal* is not instantiated

Examples

To understand the examples fully, read the reference page on `volatile/1` first.

A common case is when the Prolog process at start up should connect itself to an external database. It should also make the connection when the file with the code for the connection is loaded for the first time.

```
:- volatile db_connection/1.
:- initialization my_init.

my_init :-
  ( clause(db_connection(_), _) ->
    true
  ; set_up_connection(Connection),
    assert(db_connection(Connection))
  ).
```

In the above example, `set_up_connection/1` is user defined. We do not declare `db_connection/1` as dynamic in the file, since such a declaration would implicitly delete all clauses of the predicate when the file is reloaded.

It might not always be desirable to have the connection set up the first time the file is loaded, but only when a system is started up (for instance during the debugging of a database application.) This can be achieved with the following code (note that we use the property that a dynamic declaration reinitiliazes/resets the declared predicate):

```
:- dynamic connect/0.
:- volatile db_connection/1.
:- initialization my_init.

my_init :-
  ( connect ->
    set_up_connection(Connection),
    assert(db_connection(Connection))
  ; assert(connect)
  ).
```

See Also

`volatile/1`, `load_files/1`, `compile/1`

See Section 8.5 [ref-sls], page 192

18.3.86 instance/2

Synopsis

`instance(+Ref, -Term)`

Unifies *Term* with the most general instance of the dynamic clause or recorded term indicated by the database reference *Ref*.

Arguments

Ref db_reference
Term term

Description

Ref must be instantiated to a database reference to an existing clause or recorded term. `instance/2` is not sensitive to the source module and can be used to access any clause, regardless of its module.

Exceptions

`instantiation_error`
: if *Ref* is not instantiated

`type_error`
: if *Ref* is not a syntactically valid database reference

`existence_error`
: if *Ref* is a syntactically valid database reference but does not refer to an existing clause or recorded term.

Comments

`instance/2` ignores the module of a clause. Because of this, accessing a clause with via `instance/2` is different from accessing it via `clause/3` with a given *Ref*.

If the reference is to a unit-clause *C*, then *Term* is unified with '`C :- true`'.

Examples

```
| ?- assert(foo:bar,R).

R = '$ref'(771292,1)

| ?- instance('$ref'(771292,1),T).

T = (bar:-true)

| ?- clause(H,B,'$ref'(771292,1)).

no
| ?- clause(foo:H,B,'$ref'(771292,1)).

H = bar,
B = true

| ?-
```

See Also

`clause/3`, `asserta/2`, `assertz/2`

Section 8.14.1 [ref-mdb-bas], page 286

18.3.87 integer/1 *meta-logical*

Synopsis

`integer(+Term)`

Term is an integer.

Arguments

Term term

Examples

```
| ?- integer(5).

yes
| ?- integer(5.0).

no
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.88 `is/2`

Synopsis

-*Term* `is` +*Expression*

Evaluates *Expression* as an arithmetic expression (see Section 8.8.4 [ref-ari-aex], page 235), and unifies the resulting number with *Term*.

Arguments

Expression expr
: an expression made up of:

  functors representing arithmetic operations
  numbers

  variables bound to numbers or arithmetic expressions

Term number
: a number

Description

The possible values for *Expression* are spelled out in detail in Section 8.8.4 [ref-ari-aex], page 235.

Character codes like '`"a"`' are arithmetic expressions.

Exceptions

`instantiation_error`
`type_error`
: *Expression* is not an arithmetic expression

`domain_error`
: Attempting to divide by zero.

`representation_error`
: overflow during arithmetic evaluation

Examples

```
| ?- X is 2 * 3 + 4.

X = 10

| ?- Y = 32.1, X is Y * Y.

Y = 3.21E+01,
X = 1.03041E+03

| ?- Arity is 3 * 8, X is 4 + Arity + (3 * Arity * Arity).

Arity = 24,
X = 1756

| ?- X is 6/0.
! Domain error in argument 2 of is/2
! non-zero number expected, but 6/0 found
! goal:  _3211 is 6/0

| ?- X is 16' 7fffffff + 3.
! Syntax error
! between lines 64 and 65
! X is 0
! <<here>>
! 7 fffffff+3

| ?- X is "a".

X = 97

| ?- X is 4 * 5, Y is X * 4.

X = 20,
Y = 80
```

See example under `assign/2` to see use of `is/2` to peek at random memory addresses.

Comments

If a variable in an arithmetic expression is bound to another arithmetic expression (as opposed to a number) at runtime then the cost of evaluating that expression is much greater. It is approximately equal to the cost of `call/1` of an arithmetic goal.

See Also

`assign/2`, `</2`, `=:=/2`, `=</2`, `=\=/2`, `>/2`, `>=/2` Section 8.8 [ref-ari], page 233

18.3.89 keysort/2

Synopsis

`keysort(+List1, -List2)`

Sorts the elements of the list *List1* into ascending standard order (see Section 8.9.7.2 [ref-lte-cte-sot], page 242 with respect to the key of the pair structure.

Arguments

List1 list of pair
List2 list of pair

Description

The list *List1* must consist of terms of the form *Key-Value*. Multiple occurrences of any term are not removed.

(The time taken to do this is at worst order (N log N) where N is the length of the list.)

Note that the elements of *List1* are sorted *only* according to the value of *Key*, *not* according to the value of *Value*.

`keysort` is stable in the sense that the relative position of elements with the same key maintained.

Examples

```
| ?- keysort([3-a,1-b,2-c,1-a,1-b], X).

X = [1-b,1-a,1-b,2-c,3-a]

|?- keysort([2-1, 1-2], [1-2, 2-1]).

yes
```

Exceptions

`instantiation_error`
: If *List1* is not properly instantiated

`type_error`
: If *List1* is not a list of key-value pair.

See Also

`library(samsort)`

18.3.90 `leash/1` *development*

Synopsis

`leash(+`*Mode*`)`

Starts leashing on the ports given by *Mode*.

Arguments

Mode one of `[all]` or *one of* `[call,exit,redo,fail,done,head,exception]`
either the atom `all`, or a list of the ports to be leashed.

Description

- The leashing mode only applies to procedures that do not have spypoints on them, and it determines which ports of such procedures are leashed. By default, all seven ports are leashed. On arrival at a leashed port, the debugger will stop to allow you to look at the execution state and decide what to do next. At unleashed ports, the goal is displayed but program execution does not stop to allow user interaction.
- If you are using QUI, a more convenient way to set leashing is by using the "Leashing. . ." item in the "Options" menu. This brings up a dialog, which allows you to choose which ports to leash.
- In DEC-10 Prolog, a different form of argument was used for `leash/1`. This form, in which the argument is an integer from 0 to 127, is also supported by Quintus Prolog, but is not recommended, since the new form is clearer.
- This predicate is not supported in runtime systems.

Examples

```
| ?- leash([]).
```

turns off all leashing; now when you creep you will get an exhaustive trace but no opportunity to interact with the debugger. You can get back to the debugger to interact with it by pressing `^c t`. The command

```
| ?- leash([call,redo]).
```

leashes on the Call and Redo ports. When creeping, the debugger will now stop at every Call and Redo port to allow you to interact.

Exceptions

`instantiation_error`
: *Mode* is not ground

`domain_error`
: *Mode* is not a valid leash specification

See Also

Section 6.1.1 [dbg-bas-bas], page 113

18.3.91 `length/2`

Synopsis

`length(-List, +Integer)`

`length(*List, *Integer)`

Integer is the length of *List*. If *List* is instantiated to a proper *list*, the predicate is determinate, also when *Integer* is *var*.

Arguments

List list a list

Integer integer
non-negative integer

Description

If *List* is a list of indefinite length (that is, either a variable or of the form `[...|X]`) and if *Integer* is bound to an integer, then *List* is made to be a list of length *Integer* with unique variables used to "pad" the list. If *List* cannot be made into a list of length *Integer*, the call fails.

```
| ?-  List = [a,b|X], length(List, 4).

List = [a,b,_3473,_3475],
X = [_3473,_3475] ;

| ?-
```

If *Integer* is unbound, then it is unified with all possible lengths for the list *List*.

If *List* is bound, and is not a list, `length/2` simply fails.

Backtracking

If both *List* and *Integer* are variables, the system will backtrack, generating lists of increasing length whose elements are anonymous variables.

Exceptions

`type_error`
: *Integer* integer

Examples

```
| ?- length([1,2], 2).

yes
| ?- length([1,2], 0).

no
| ?- length([1,2], X).

X = 2 ;

no
```

18.3.92 library_directory/1 *extendable*

Synopsis

`:- multifile library_directory/1. library_directory(*DirSpec*)`

Defines a library directory. Used by predicates taking *file_spec* as input argument.

Arguments

DirSpec file_spec
: Either an atom giving the path to a file, or *PathAlias(DirSpec)*, where PathAlias is defined by a `file_search_path` rule (see the reference page for `file_search_path/2`).

Description

The dynamic, multifile `library_directory/1` facts reside in module user. They define directories to search when a file specification `library(File)` is expanded to the full path.

There are a set of predefined `library_directory/1` facts, but users may also define their own libraries simply by asserting the appropriate `library_directory/1` facts into module user. To locate a library file, the `library_directory/1` facts are tried one by one in the same sequence they appear in the Prolog database.

The `file_search_path` mechanism is an extension of the `library_directory` scheme. See `file_search_path/2` and Section 8.6 [ref-fdi], page 205.

Examples

```
| ?- assert(library_directory('/usr/joe_bob/prolog/libs')).

yes
| ?- ensure_loaded(library(flying)).
% loading file /usr/joe_bob/prolog/libs/flying.qof
...
```

See Also

`absolute_file_name/[2,3]`, `assert/[1,2]`, `dynamic/1`, `file_search_path/2`, `listing/1`, `load_files/[1,2]`

Section 8.6 [ref-fdi], page 205.

18.3.93 `line_count/2`

Synopsis

`line_count(+Stream, -N)`

Unifies *N* with the total number of lines either read or written on the open stream *Stream*.

Arguments

Stream stream_object
N integer

Description

A freshly opened stream has a line count of 1. See Section 8.7.8.1 [ref-iou-sos-spt], page 230, for details on the use of this predicate on a stream that is directed to the user's terminal.

Exception

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226)

See Also

`character_count/2`, `line_position/2`, `stream_position/3` Section 8.7 [ref-iou], page 214

18.3.94 `line_position/2`

Synopsis

`line_position(+`*`Stream`*`, -`*`N`*`)`

Unifies *N* with the total number of characters either read or written on the current line of *Stream*.

Arguments

Stream stream_object
: specifies an open stream

N integer
: current line position

Description

A fresh line has a line position of 0. See Section 8.7.8.1 [ref-iou-sos-spt], page 230, for details on the use of this predicate on a stream that is directed to the user's terminal.

Exception

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226)

See Also

`character_count/2`, `line_count/2`, `stream_position/3` Section 8.7 [ref-iou], page 214

18.3.95 `listing/[0,1]`

Synopsis

`listing`

`listing(+`*`PredSpecs`*`)`

Prints the clauses of all the dynamic procedures currently in the Prolog database, or of *PredSpecs*, to the current output stream, using `portray_clause/1`.

Arguments

PredSpecs gen_pred_spec_tree_var [MOD]
: a predicate specification, or a list of predicate specifications or atoms

Description

If *Predicate* is an atom, then `listing/1` lists the dynamic procedures for all predicates of that name, as for `listing/0`

If *PredSpecs* is a predicate specification of the form *Name*/*Arity*, only the clauses for the specified predicate are listed.

PredSpecs can be a list of predicate specifications and/or atoms; for example,

```
| ?- listing([concatenate/3, reverse, go/0]).
```

Examples

You could list the entire program to a file using the command

```
| ?- tell(file), listing, told.
```

Note that `listing/[0,1]` does not work on compiled procedures.

`listing/1` is dependent on the source module. As a special case,

```
| ?- listing(mod:_).
```

will list all the dynamic predicates in module mod. However, `listing/0` is not dependent on the source module; it refers instead to the type-in module.

Variables may be included in predicate specifications given to `listing/1`. For example, you can list clauses for `f` in any current module with:

```
| ?- listing(_:f).
```

Comments

Under the Emacs interface, there is a facility for finding the source code definition for a specified compiled or dynamic procedure and reading it into an edit buffer. This is likely to be more helpful than `listing/1` in most cases. See Section 4.2.2 [ema-emi-key], page 89 for more information.

18.3.96 `load_files/[1,2]`

Synopsis

`load_files(+Files)`

`load_files(+Files, +Options)`

`[+File|+Files]`

`[]`

Load the specified Prolog source and/or QOF files into memory. Subsumes all other load predicates.

Arguments

Files file_spec or *list of file_spec* [MOD]
: a file specification or a list of file specifications; a '`.pl`' or '`.qof`' extension may be omitted in a file specification.

Options list
: a list of zero or more options of the form:

`if(X)`

`X=true` (default) always load

`X=changed`
: load file if it is not already loaded or if it has been changed since it was last loaded

`when(X)`

`X=run_time`
: (default) The file does not define any predicates that will be called during compilation of other files.

`X=compile_time`
: the file *only* defines predicates that wil be called during compilation of other files; it does not define any predicates that will be called when the application is running.

`X=both` the file defines some predicates that will be needed during compilation and some that will be needed during execution.

`load_type(X)`

`X=compile` compile Prolog source code

`X=qof` load QOF code

`X=latest` (default) load QOF or compile source, whichever is newer. The latest option is effective only if Files are sepcified without extensions.

`must_be_module(X)`

`X=true` the files are required to be module-files

`X=false` (default) the files need not be module-files

`imports(X)`

`X=all` (default) if the file is a module-file, all exported predicates are imported

`X=List` list of predicates to be imported

Note that if the option `imports` is present, the option `must_be_module(true)` is enforced.

`all_dynamic(X)`

`X=true` load all predicates as dynamic

`X=false` (default) load predicates as static unless they are declared dynamic

Note that the `all_dynamic option` has no effect when a QOF file is loaded. Thus it is not normally useful to use `all_dynamic(true)` in conjunction with `load_type(latest)`, since the file will be loaded in dynamic mode only if the source file is more recent than the QOF file.

`silent(X)`

`X=true` loading information is printed as silent messages (see Section 8.20 [ref-msg], page 325 for details).

`X=false` (default) loading information is printed as informational message.

Description

`load_files/2` is the most general predicate for loading Prolog files. Special cases of it are provided by the following predicates:

```
load_files(Files) :-
        load_files(Files, []).
[].
[File|Files] :-
        load_files([File|Files]).
compile(Files) :-
        load_files(Files, [load_type(compile)]).
consult(Files) :-     /*consult equivalent to
                        compile now*/
        compile(Files).
ensure_loaded(Files) :-
        load_files(Files, [if(changed)]).
use_module(Files) :-
        load_files(Files, [if(changed),
                      must_be_module(true)]).
use_module(File, Imports) :-
        load_files(File,  [if(changed),
                      must_be_module(true),
                      imports(Imports)]).
```

`load_files/[1,2]` reads Prolog clauses, in source or in compiled (QOF) form, and adds them to the Prolog database, after first deleting any previous versions of the predicates they define. Clauses for a single predicate must all be in the same file unless that predicate is declared to be `multifile`.

If the file contains directives, that is, terms with principal functor `:-/1` or `?-/1`, then these are executed as they are encountered.

Clauses and directives can be transformed as they are read from source files (not from QOF), by providing a definition for `term_expansion/2`. This is true in both the development system and QPC, but in order for this to work properly in QPC, your definition of `term_expansion/2` (and everything it calls) must be loaded into QPC. This is accomplished with the `when` option to `load_files/2`, or the '`-i`' option to QPC.

A non-module source file can be loaded into any module by `load_files/[2,3]`, but the module of the predicates in a QOF-file is fixed at the time it is created (by QPC, `save_predicates/2` or `save_program/[1,2]`). It is thus not possible to qof save a predicate from say module `foo`, and reloaded it into module `bar`, or QPC the non-module-file '`f1.pl`' into '`f1.qof`', and then load '`f1.qof`' into module `mod` (QPC assumes module `user` when non-module files are compiled separately). To avoid mistakes, `load_files/[1,2]` loads the corresponding source file, if such exists, whenever a non-module-file is loaded into module other than `user`. If no corresponding source file exists, the QOF file is loaded; care should be taken in this case.

Initialization goals specified with `initialization/1` are executed after the load.

When `load_files/[1,2]` is called from an embedded command in a file being compiled by QPC, the `load_type` and `if` options are ignored. The specfied files are compiled from

source to QOF, if the source is newer than the corresponding QOF file. If the option `when(compile_time)` is given, the file is instead compiled into QPC memory, and no QOF is generated (see above).

When `load_files/[1,2]` is called in a runtime system, the `all_dynamic` option will be automatically set to true because the compiler is not available in runtime systems. This means that the loaded code will run slower.

Exceptions

`instantiation_error`
: *M*, *Files*, or *Options* is not ground.

`type_error`
: In *M*, in *Files*, or in *Options*.

`domain_error`
: Illegal option in *Options*.

`existence_error`
: A specified file does not exist. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

See Also

`compile/1`, `consult/1`, `dynamic/1`, `ensure_loaded/1`, `fileerrors/0`, `multifile/1`, `no_style_check/1`, `nofileerrors/0`, `prolog_load_context/2`, `source_file/[1,2]`, `style_check/1`, `term_expansion/2`, `use_module/[1,2,3]`, `initialization/1`, `volatile/1`.

Section 8.4 [ref-lod], page 189

18.3.97 `load_foreign_executable/1` *hookable*

Synopsis

`load_foreign_executable(+Executable)`

Load the foreign executable (shared object file) *Executable* into Prolog. Relies on the hook predicates `foreign_file/2` and `foreign/[2,3]`.

Arguments

Executable file_spec [MOD]
The shared object file to be loaded.

Description

`load_foreign_executable/1` takes a shared object file and loads it into Prolog. If the file contains dependencies on other shared objects/libraries, then these are loaded automatically. For details on how these are loaded see Section 10.3.12 [fli-p2f-lfe], page 401.

The extension can be omitted from the filename given in the *Executable* argument.

Uses the `foreign/3` and `foreign_file/2` facts defined by the user to make the connection between a Prolog procedure and the foreign function. When loading the shared object file, it looks for a `foreign_file/2` fact for that file and for each symbol in the `foreign_file/2` fact it looks for a `foreign/3` fact that gives the name of the Prolog procedure associated with the foreign symbol and the argument specification.

Looks for `foreign/3` and `foreign_file/2` facts defined in its source module only.

Before calling this predicate, generate the shared object file from object files (and libraries); see Section 10.3.2 [fli-p2f-uso], page 376.

Exceptions

Errors in the specification of `foreign/3` will all be reported when `load_foreign_executable/1` is called.

Examples

See example under `foreign/[2,3]`

See Also

`foreign_file/2`, `foreign/[2,3]`, `load_foreign_files/2`

Section 10.3 [fli-p2f], page 375

18.3.98 `load_foreign_files/2` *hookable*

Synopsis

`load_foreign_files(+ObjectFiles, +Libraries)`

Loads foreign object files into Prolog. Relies on the hook predicates `foreign_file/2` and `foreign/[2,3]`.

Arguments

ObjectFiles list of file_spec [MOD]
: A list of foreign object files to be loaded.

Libraries list of atom
: A list of shared libraries that need to be searched while loading *ObjectFiles*.

Description

`load_foreign_files/2` takes a list of object files and a list of shared libraries, links them and then loads the result into Prolog. The linking is done using the system linker. For details on the call to the linker, see Section 10.3.13 [fli-p2f-lff], page 401.

The extension can be omitted from the filenames given in the *ObjectFiles* argument.

Uses the `foreign/3` and `foreign_file/2` facts defined by the user to make the connection between a Prolog procedure and the foreign function. When loading each object file, it looks for a `foreign_file/2` fact for the object file and for each symbol in the `foreign_file/2` fact it looks for a `foreign/3` fact that gives the name of the Prolog procedure associated with the foreign symbol and the argument specification.

Looks for `foreign/3` and `foreign_file/2` facts defined in its source module only.

Looks at the environment variable `TMPDIR` for the directory to store all the temporary files created during the linking and loading process. The default directory is '`/tmp`'.

Before calling this predicate, generate the object files for the foreign functions using the foreign language compiler. The object files should contain position independent code; see Section 10.3.2 [fli-p2f-uso], page 376.

Exceptions

Errors in the specification of `foreign/3` will all be reported when `load_foreign_files/2` is called.

Examples

See example under `foreign/[2,3]`

See Also

`foreign_file/2`, `foreign/[2,3]`, `load_foreign_executable/1`

Section 10.3 [fli-p2f], page 375

18.3.99 `manual/[0,1]` *development*

Synopsis

`manual`

Displays a menu of the top layer of the manual hierarchy.

`manual(+`*Term*`)`

Goes directly to the point in the manual represented by *Term*.

Arguments

Term term
: Either a reference of form *word-. . .-word* (e.g. `int-dir`), or a topic.

Description

The menu brought up by `manual/0` gives you a choice among the major parts of the manual (see values of *part* above). Whichever you select, you will then be shown a menu of chapter titles. When you select a chapter you will see a menu of the section titles of that chapter, and so on.

`manual/1` can also be used as a synonym for `help/1`. If the argument to `manual/1` is not a reference (above), then `help/1` is called with that argument, so that the following behave alike:

```
| ?- manual(Topic).
| ?- help(Topic).
```

This predicate is not supported in runtime systems.

Examples

To view the text in Section 1.3 [int-dir], page 11 type:

```
| ?- manual(int-dir).
```

To look up character escaping, type:

```
| ?- manual('character escaping').
```

This brings up a menu, which contains reference terms enclosed in curly braces. Simply copy the one you want at the prompt.

See Also

`help/1`

Section 8.17 [ref-olh], page 304

18.3.100 `message_hook/3` *hook*

Synopsis

```
:- multifile message_hook/3.

message_hook(+MessageTerm, +Severity, +Lines)
```

Overrides the call to `print_message_lines/3` in `print_message/2`. A way for the user to intercept the *Message* of type *Severity*, whose translations is *Lines*, before it is actually printed.

Arguments

MessageTerm term
: any term

Severity one of `[informational,warning,error,help,silent]`

Lines list of pair
: is of the form `[Line1, Line2, ...]`, where each *Linei* is of the form `[Control_1-Args_1,Control_2-Args_2, ...]`.

Description

After a message is parsed, but before the message is written, `print_message/2` calls

```
user:message_hook(+MsgTerm,+Severity,+Lines)
```

If the call to `user:message_hook/3` succeeds, `print_message` succeeds without further processing. Otherwise the built-in message display is used. It is often useful to have a message hook that performs some action and then fails, allowing other message hooks to run, and eventually allowing the message to be printed as usual. See Section 8.20.3.3 [ref-msg-umf-ipm], page 331 for an example.

Exceptions

An exception raised by this predicate causes an error message to be printed and then the original message is printed using the default message text and formatting. Since the user defines `message_hook/3`, they can write code that might raise exceptions.

Examples

The following is the default, built-in message portrayal predicate:

```
message_hook(MessageTerm,Severity,Lines):-
  ( Severity == silent ->
    true
    /* Don't translate or print silent messages */
  ; severity_prefix(Severity,Prefix,Stream) ->
    print_message_lines(Stream,Prefix,Lines)
  ; raise_exception(domain_error(
        print_message(Severity,MessageTerm),1,
        one_of([help,error,warning,
                informational,silent]),
        Severity)).

severity_prefix(silent,         '', user_error).
severity_prefix(help,           '', user_error).
severity_prefix(error,       '! ',  user_error).
severity_prefix(warning,     '* ',  user_error).
severity_prefix(informational,'% ', user_error).
```

The reasoning behind the assignment of streams is that all unsolicited output should go to `user_error`.

Tips

See Also

`print_message/2`, `generate_message/3`, `print_message_lines/3`

Section 8.20 [ref-msg], page 325

18.3.101 `meta_predicate/1` *declaration*

Synopsis

`:- meta_predicate +`*MetaSpec*

Provides for module name expansion of arguments in calls to the predicate given by *MetaSpec*. All `meta_predicate/1` declarations must be at the beginning of a module.

Arguments

MetaSpec callable
: Goal template or list of goal templates, of the form:

  `functor(`*Arg1*`, `*Arg2*`,...)`

  Each *Argn* is one of:

  ‘`:`’ requires module name expansion

  integer `>=0`
  : same as ‘`:`’

  ‘`+`’, ‘`-`’, ‘`?`’, ‘`*`’
  : ignored

Description

All `meta_predicate` declarations must be at the beginning of a module, immediately after the module declaration, because the `meta_predicate` declarations need to be known at the time other modules are loaded if those modules use the meta-predicates.

The reason for allowing a non-negative integer as an alternative to ‘`:`’ is that this may be used in the future to supply additional information to the cross-referencer (`library(xref)`) and to the Prolog compiler. An integer *n* is intended to mean that that that argument is a term, which will be supplied *n* additional arguments.

Represents DEC-10 Prolog-style “mode” declaration. Provides for module name expansion of arguments in *MetaSpec*.

Exceptions

`context_error`
: Declaration appears in query.

Caveat

When a `meta_predicate` declaration is added, removed or changed, the file containing it, as well as all the modules that import the predicate given by *MetaSpec*, must be reloaded.

Examples

Consider a sort routine, `mysort/3`, to which the name of the comparison predicate is passed as an argument:

```
mysort(+CompareProc, +InputList, -OutputList)
```

If *CompareProc* is module sensitive, an appropriate `meta_predicate` declaration for `mysort/3` is:

```
:- meta_predicate mysort(:, +, -).
```

This means that whenever a goal `mysort(A, B, C)` appears in a clause, it will be transformed at load time into `mysort(M:A, B, C)`, where *M* is the source module. The transformation will happen unless

1. *A* is of the form *MetaSpec*.
2. *A* is a variable and the same variable appears in the head of the clause in a module-name-expansion position.

Many examples in library, e.g. `library(samsort)`.

See Also

`module/2`

Section 8.13.17 [ref-mod-met], page 284

18.3.102 `mode/1` *declaration*

Synopsis

`:- mode (+Mode)`

Currently a dummy declaration.

Arguments

Mode term [MOD]

Description

So that DEC-10 Prolog programs can be read in

18.3.103 `module/1`

Synopsis

`module(+`*`ModuleName`*`)`

Changes the type-in module (see Section 8.13.8 [ref-mod-tyi], page 276) to *ModuleName*. Thus subsequent top-level goals use *ModuleName* as their source module.

Arguments

ModuleName *atom*
: the name of a module

Description

If *ModuleName* is not a current module, a warning message is printed, but the type-in module is changed nonetheless.

ModuleName does not become a current module until predicates are loaded into it.

Calling `module/1` from a command embedded in a file that is being loaded does not affect the loading of clauses from that file. It only affects subsequent goals that are typed at top level.

Exceptions

`instantiation_error`

See also

`module/2`, `current_module/[1,2]`

18.3.104 `module/2` *declaration*

Synopsis

`:- module(+ModuleName, +PublicPred).`

Declares the file in which the declaration appears to be a module-file named *ModuleName*, with public predicates *PublicPred*. Must appear as the first term in the file.

Arguments

ModuleName atom
: an atom

PublicPred list of simple_pred_spec
: List of predicate specifications of the form *Name/Arity*.

Description

The definition of a module is not limited to a single file, because a module-file may contain commands to load other files. If '`myfile`', a module-file for *ModuleName*, contains an embedded command to load '`yourfile`' and if '`yourfile`' is not itself a module-file, then all the predicates in '`yourfile`' are loaded into module *ModuleName*.

If the export list is not properly specified, there will be a warning or error message at compile time.

Exceptions

At compile time:

`context_error`
: Declaration appears other than as the first term in a file being loaded.

`instantiation_error`
: *ModuleName* not instantiated.

`type_error`
: *PredSpecList* is not a list of simple_pred_spec

See Also

`module/1` Section 8.13 [ref-mod], page 271

18.3.105 `multifile/1` *declaration*

Synopsis

`:- multifile` *+PredSpecs*

Allows the clauses for the specified predicates to be in more than one file.

Arguments

PredSpecs pred_spec_forest [MOD]
: A single predicate specification of the form *Name/Arity*, or a sequence of predicate specifications separated by commas. *Name* must be an atom and *Arity* an integer in the range 0..255.

Description

A built-in prefix operator, so that declarations can be written as e.g.

```
:- multifile a/1, b/3.
```

By default, all clauses for a predicate are expected to come from just one file. This assists with reloading and debugging of code. Declaring a predicate `multifile` means that its clauses can be spread across several different files. This is independent of whether or not the predicate is declared `dynamic`.

Should precede all the clauses for the specified predicates in the file.

There should be a `multifile` declaration for a predicate P in every file that contains clauses for P. This restriction is not currently enforced in the Development System: for compatibility with earlier releases it suffices to have a `multifile` declaration in the first file loaded that contains clauses for P. However, a warning is noted if the `multifile` declaration is omitted in subsequent files. The `multifile` declarations *must* be included in every file when `qpc` is being used to compile files separately.

If a `multifile` predicate is dynamic, there should be a `dynamic` declaration in every file containing clauses for the predicate. Again, this is not enforced in the Development System, for backwards compatibility, but warnings are printed if the `dynamic` declarations are omitted. The `dynamic` declarations may not be omitted when `qpc` is being used to compile files separately.

When a file containing clauses for a `multifile` predicate (P) is reloaded, the clauses for P that previously came from that file are removed. Then the new clauses for P (which may be the same as the old ones) are added to the end of the definition of the `multifile` predicate.

An exception to this is when the file concerned is the pseudo-file `user`, meaning that clauses are being entered from the terminal; in this case the clauses are always added to the end of the predicate without removing any previously defined clauses.

If a `multifile` declaration is found for a predicate that has already been defined in another file (without a `multifile` declaration), then this is considered to be a redefinition of that predicate. Normally this will result in a multiple-definition style-check warning (see `style_check/1`).

The predicate `source_file/2` can be used to find all the files containing clauses for a `multifile` predicate.

`multifile` predicates can be extended at run-time using `multifile_assertz/1`.

`multifile/1` cannot be called as a built-in predicate. It can only be used as a declaration to the compiler in a Prolog source file.

Exceptions

`instantiation_error`
: *PredSpecs* not ground.

`type_error`
: Either name or arity in PredSpec has the wrong type

`domain_error`
: *Arity* not in the range 0..255.

`context_error`
: If the declaration contradicts previous declaration or clauses for the same predicate in the file.

See Also

`multifile_assertz/1`, `source_file/[1,2]`, `compile/1`, `load_files/[1,2]`, `dynamic/1`.

18.3.106 `multifile_assertz/1`

Synopsis

`multifile_assertz(+`*`Clause`*`)`

Adds a compiled clause to the database. The clause will be added at the end of all existing clauses in the database.

Arguments

Clause [MOD]
: *callable* A valid Prolog clause.

Description

If a predicate is multifile (compiled, interpreted-static or dynamic) `multifile_assertz/1` can be used to add a clause, *Clause*, to the end of the predicate. In a runtime system (see Section 9.1 [sap-srs], page 337), it is an error to `multifile_assertz` a compiled clause because the compiler is not available.

If predicate is undefined at the time of the `multifile_assertz`, it is set to be compiled (in the Development System) or dynamic (in a Runtime System). In either case the predicate is also set to be multifile.

[Note that in runtime systems `compile(`*`File`*`)` actually loads the file as `all_dynamic`.]

Except for the case of a multifile dynamic predicate, the effect of `multifile_assertz` if used on a predicate that is currently running will not be well-defined. The new clause may or may not be seen on backtracking. If you want the proper semantics, use `assertz` instead.

Exceptions

Same as `assert/1`.

See Also

`abolish/[1,2]`, `assertz/1`, `dynamic/1`, `multifile/1`, Section 9.1 [sap-srs], page 337

Section 9.2 [sap-rge], page 355

18.3.107 name/2

Synopsis

`name(+Constant, -Chars)`

`name(-Constant, +Chars)`

Chars is the list consisting of the ASCII character codes comprising the printed representation of *Constant*.

Arguments

Constant atomic
Chars chars

Description

Initially, either *Constant* must be instantiated to a number or an atom, or *Chars* must be instantiated to a proper list of character codes (containing no variables).

If *Constant* is initially instantiated to an atom or number, *Chars* will be unified with the list of character codes that make up its printed representation.

If *Constant* is uninstantiated and *Chars* is initially instantiated to a list of characters that corresponds to the correct syntax of a number (either integer or float), *Constant* will be bound to that number; otherwise *Constant* will be instantiated to an atom containing exactly those characters.

Examples

```
| ?- name(foo, L).

L = [102,111,111]

| ?- name('Foo', L).

L = [70,111,111]

| ?- name(431, L).

L = [52,51,49]
```

```
| ?- name(X, [102,111,111]).

X = foo

| ?- name(X, [52,51,49]).

X = 431

| ?- name(X, "15.0e+12").

X = 1.5e+13
```

There are atoms that can be read and written by Prolog, and that can be converted to chars by `name/2`, but that it can *not* construct. One example of this is the atom `0`:

```
| ?- X = '0', atom(X), name(X, L).

X = '0',
L = [48]

| ?- name(X, [48]), atom(X).

no

| ?- name(X, [48]), integer(X).

X = 0
```

This anomaly is present in DEC-10 Prolog and C-Prolog. `name/2` is retained for compatibility with them. New programs should mainly use `atom_chars/2` (see Section 8.9.4 [ref-lte-c2t], page 240) or `number_chars/2` (see Section 8.9.4 [ref-lte-c2t], page 240) as appropriate.

Exceptions

`instantiation_error`
: If Constant and Chars are both uninstantiated

`type_error`
: If Constant is not a constant

`domain_error`
: Chars is not a list of ASCII codes

See Also

Section 8.9.4 [ref-lte-c2t], page 240

18.3.108 `nl/[0,1]`

Synopsis

`nl`

`nl(+`*`Stream`*`)`

Terminates the current output record on the current output stream or on *Stream*. See Section 10.5.3.3 [fli-ios-sst-fmt], page 438.

Arguments

Stream stream_object
: a valid Prolog stream

Description

Wraps the current output record (line) and writes out the record. How the record is wrapped up depends on the format of the output stream.

- For a default text stream `nl/[0,1]` will output a ⟨LFD⟩ (ASCII code 10). Windows translates this to the sequence ⟨RET⟩⟨LFD⟩ (ASCII codes 13, 10).
- For a binary stream, the record is simply written out.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: Some operating system dependent error occurred in writing.

`permission_error`
: There is an error in the bottom layer of write function of the stream.

Comments

How the wrapped record is written depends upon the bottom layer write function associated with the output stream. The default tty stream displays the wrapped output record to the terminal immediately for each `nl/[0,1]` operation, while the default text stream under UNIX and Windows does not send the wrapped record (line) to the disk unless the buffer for the output stream is full.

See Also

`put/[1,2]` `flush_output/1`.

18.3.109 `no_style_check/1`

Synopsis

`no_style_check(+`*Type*`)`

Turns off the specified *Type* of compile-time style checking.

Arguments

Type is one of the following atoms:

`all` Turn off all style checking.

`single_var`
: Turn off checking for clauses containing a single instance of a named variable, where variables that start with a '`_`' are not considered named.

`discontiguous`
: Turn off checking for procedures whose clauses are not all adjacent to one another in the file.

`multiple` Turn off checking for multiple definitions of the same procedure in different files.

Description

The normal use of this predicate is as an embedded command in a file that has not been written to follow the recommended style conventions (see Section 2.2.5 [bas-lod-sty], page 24). For example, you could put

```
:- no_style_check(discontiguous).
```

at the beginning of a file and

```
:- style_check(discontiguous).
```

at the end of the file.

Exceptions

`instantiation_error`
: *Type* is not bound.

`type_error`
: *Type* is not an atom.

`domain_error`
: *Type* is not a valid type of style checking.

See Also

`style_check/1`.

18.3.110 `nocheck_advice/[0,1]` *development*

Synopsis

`nocheck_advice`

`nocheck_advice(+`*`PredSpecs`*`)`

Disable advice checking on all predicates given by *PredSpecs*.

Arguments

PredSpecs gen_pred_spec_tree [MOD]
: A list of predicate specifications.

Description

`nocheck_advice/1` is used to disable advice checking on all predicates specified in *PredSpecs*. `nocheck_advice/0` disables advice checking on all predicates for which advice checking is currently enabled. When advice checking is disabled for a predicate, and execution of that predicate reaches an advised port, execution will proceed as though the port wasn't advised.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: if the argument is not ground.

`type_error`
: if a *Name* is not an atom or an *Arity* not an integer.

`domain_error`
: if a *PredSpec* is not a valid procedure specification, or if an *Arity* is specified as an integer outside the range 0-255.

`permission_error`
: if a specified procedure is built-in.

Tips

`nocheck_advice/0` behaves as though implemented by

```
nocheck_advice :-
        current_advice(Goal, Port, Action),
        functor(Goal, Name, Arity),
        nocheck_advice(Name/Arity),
        fail.
nocheck_advice.
```

See Also

add_advice/3, remove_advice/3, current_advice/3, check_advice/[0,1]

18.3.111 `nodebug/0` *development*

Synopsis

`nodebug`

Turns the debugger off. Equivalent to `notrace/0`.

Description

Does *not* remove any spypoints. Spypoints will remain where they were set, although they will have no effect while the debugger is off. When the debugger is turned on again, the spypoints will again take effect.

To remove all spypoints, use `nospyall/0`

This predicate is not supported in runtime systems.

18.3.112 `nofileerrors/0`

Synopsis

`nofileerrors`

Disables the `fileerrors` flag

Description

The built-in predicates that open files simply fail, instead of raising an exception if the specified file cannot be opened.

The `fileerrors` flag is only enabled by an explicit call to `fileerrors/0`, or via `prolog_flag/[2,3]`, which can also be used to obtain the current value of the `fileerrors` flag. See Section 8.10.1 [ref-lps-ove], page 245, for more information on the `fileerrors` flag.

Tips

`nofileerrors` is a drastic predicate, since it affects the use of all predicates that open files. You might be unintentionally changing the behaviour of calls to `open/3` from other parts of the system (written by other people or from libraries). A better way to detect and ignore file errors is to wrap specific calls to `open/3` with `on_exception/3` and ignore the types of errors you want to ignore.

See Also

`fileerrors/0`, `prolog_flag/[2,3]`

18.3.113 `nogc/0`

Synopsis

`nogc`

Disables the garbage collector.

Description

As if defined by:

```
nogc :- prolog_flag(gc, _, off).
```

See Also

`prolog_flag/[2,3]`

Section 8.12.3 [ref-mgc-egc], page 261

18.3.114 nonvar/1 *meta-logical*

Synopsis

`nonvar(+`*Term*`)`

Term is currently instantiated. This is the opposite of `var/1`.

Arguments

Term term

Example

```
| ?- nonvar([X,Y]).

X = _288
Y = _303
| ?- nonvar(X).

no
```

See Also

`atom/1`, `atomic/1`, `number/1`, `var/1`, `compound/1`, `callable/1`, `simple/1`

18.3.115 `noprofile/0` *development*

Synopsis

`noprofile`

Turns off the profiler.

Description

Turns off the profiler and removes data structures containing profiling information associated with any predicates that have been profiled.

This predicate is not supported in runtime systems.

See Also

`profile/[0,1,2,3]`

18.3.116 `nospy/1` *development*

Synopsis

`nospy(+`*PredSpecs*`)`

Removes spypoints on all the predicates represented by *PredSpecs*.

Arguments

PredSpecs gen_pred_spec_tree
: Single predicate specification of form *Name* or *Name/Arity*, or a list of such.

Description

To remove all spypoints, use `nospyall/0`

If `nospy/1` is given any invalid argument it prints a warning.

Note that since `nospy` is a built-in operator, the parentheses, which usually surround the arguments to a predicate are not necessary (although they can be used if desired).

```
| ?- nospy test/1.
% spypoint removed from test/1

yes
| ?-
```

This predicate is not supported in runtime systems.

See Also

`spy/1`, `nospyall/0`, `debug/0`, `add_spypoint/1`, `remove_spypoint/1`

18.3.117 `nospyall/0` *development*

Synopsis

`nospyall`

Removes all spypoints.

Description

The only way to remove all spypoints at once, since turning off debugging with `nodebug/0` does *not* remove spypoints; they remain in place and are reactivated if the debugger is turned back on using `trace/0` or `debug/0`.

This predicate is not supported in runtime systems.

See Also

`nospy/1`

18.3.118 `notrace/0` *development*

Synopsis

`notrace`

Turns the debugger off. Equivalent to `nodebug/0`

Description

This predicate is not supported in runtime systems.

18.3.119 `number/1` *meta-logical*

Synopsis

`number(+`*Term*`)`

Term is currently instantiated to either an integer or a float.

Arguments

Term term

Examples

```
| ?- number(5.2).

yes
| ?- number(5).

yes
```

See Also

`atom/1`, `atomic/1`, `var/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.120 `number_chars/2`

Synopsis

`number_chars(+`*Number*`, -`*Chars*`)`

`number_chars(-`*Number*`, +`*Chars*`)`

Chars is the list consisting of the ASCII character codes comprising the printed representation of *Number*.

Arguments

Number number
Chars chars

Description

Chars is the list of ASCII character codes comprising the printed representation of *Number*.

Initially, either *Number* must be instantiated to a number, or *Chars* must be instantiated to a proper list of character codes (containing no variables).

If *Number* is initially instantiated to a number, *Chars* will be unified with the list of character codes that make up its printed representation.

If *Number* is uninstantiated and *Chars* is initially instantiated to a list of characters that corresponds to the correct syntax of a number (either integer or float), *Number* will be bound to that number; otherwise `number_chars/2` will simply fail.

Exceptions

`instantiation_error`
: *Number* and *Chars* are both uninstantiated

`domain_error`
: *Chars* is not a list of ASCII codes

`type_error`
: *Number* is not a number or *Chars* is not a list

`representation_error`
: *Chars* is a list corresponding to a number that can't be represented

Examples

```
| ?- number_chars(foo, L).

no

| ?- number_chars(431, L).

L = [52,51,49]

| ?- number_chars(X, [102,111,111]).

no

| ?- number_chars(X, [52,51,49]).

X = 431

| ?- number_chars(X, "15.0e+12").

X = 1.5e+13
```

See Also

atom_chars/2

18.3.121 numbervars/3 *meta-logical*

Synopsis

`numbervars(+-`*Term*`, +`*FirstVar*`, -`*LastVar*`)`

instantiates each of the variables in *Term* to a term of the form `'$VAR'(`*N*`)`.

Arguments

Term term
FirstVar integer
LastVar integer

Description

FirstVar is used as the value of *N* for the first variable in *Term* (starting from the left). The second distinct variable in *Term* is given a value of *N* satisfying "*N* is *FirstVar*+1"; the third distinct variable gets the value *FirstVar*+2, and so on. The last variable in *Term* has the value *LastVar*-1.

Notice that in the example below, `display/1` is used rather than `write/1`. This is because `write/1` treats terms of the form `'$VAR'(`*N*`)` specially; it writes '`A`' if *N*=0, '`B`' if *N*=1, ...'`Z`' if *N*=25, '`A1`' if *N*=26, etc. That is why, if you type the goal in the example below, the variable bindings will also be printed out as follows:

```
Term = foo(W,W,X),
A = W,
B = X
```

Exceptions

`instantiation_error`
: *Number* and *Chars* are both instantiated

`type_error`
: *Number* is not a number or *Char* is not a list

Example

```
| ?- Term = foo(A, A, B), numbervars(Term, 22, _),
     display(Term).
foo($VAR(22),$VAR(22),$VAR(23))
```

See Also

`write_term/1`, `write_canonical/1`

18.3.122 `on_exception/3`

Synopsis

`on_exception(-`*`Exception`*`, +*`*`ProtectedGoal`*`, +*`*`Handler`*`)`

Specify an exception handler for *ProtectedGoal*, and call *ProtectedGoal*.

Arguments

Exception term
: Any term.

ProtectedGoal callable [MOD]
: A goal.

Handler callable [MOD]
: A goal.

Description

ProtectedGoal is executed. This will behave just as if *ProtectedGoal* had been written without the `on_exception/3` wrapper. If *ProtectedGoal* is determinate, then `on_exception/3` will also be determinate. *ProtectedGoal* can also be nondeterminate. As a general rule, code is easier to read when *ProtectedGoal* is a simple goal, however a conjunction of goals (*Goal1*,. . . *GoalN*) or any other form that `call/1` accepts is allowed.

If an exception is raised while *ProtectedGoal* is running, Prolog will *abandon ProtectedGoal* entirely. Any bindings made by *ProtectedGoal* will be undone, just as if it had failed. Side effects, such as data-base changes and input/output, are not undone, just as they are not undone when a goal fails. After undoing the bindings, Prolog then tries to unify an object called an *exception term* with the *Exception* argument. If this unification succeeds, *Handler* will be executed as if you had written

```
Exception=the actual exception term,
Handler
```

If this unification fails, Prolog will keep looking for a handler. It will always find a handler at the top level, which prints out a message corresponding to the exception.

In applications lacking a top level (C calling Prolog, where `QP_toplevel()` has not been called) exceptions are indicated by the return status `QP_ERROR`. For more details refer to Section 10.4 [fli-ffp], page 413.

Exceptions

Same as `call/1`.

Tip

More efficient code is generated when *ProtectedGoal* is a simple goal. In other cases, such as where *ProtectedGoal* is a conjunction of goals (*Goal1*,. . . *GoalN*), the compiler treats this as if it were `call((`*`Goal1`*`,..., `*`GoalN`*`))`. This potential inefficiency does not apply to *Handler*.

Examples

Fail on exception:

```
:-meta_predicate fail_on_exception/1.
fail_on_exception(C):-
    on_exception(E,C,print_exception_then_fail(C,E)).

print_exception_then_fail(C,E):-
    format('Exception occured while calling ~q:~n',
                                              [C]),
    print_message(warning,E),
    fail.
```

See Also

`raise_exception/1`, `print_message/2`.

Section 8.19.3 [ref-ere-hex], page 312.

18.3.123 `op/3`

Synopsis

`op(+Precedence, +Type, +Name)`

declares *Name* to be an operator of the stated *Type* and *Precedence*.

Arguments

Precedence integer
: integer in the range 1-1200

Type one of `[xfx,xfy,yfx,fx,fy,xf,yf]`

Name atom
: atom or a list of atoms.

Description

Operators are a notational convenience to read and write Prolog terms. You can define new operators using `op/3`.

The *Precedence* of an operator is used to disambiguate the way terms are parsed. The general rule is that the operator with the highest precedence is the principal functor.

The *Type* of an operator decides the position of an operator and its associativity. In the atom that represents the type the character '`f`' represents the position of the operator. For example, a type '`fx`' says that the operator is a prefix operator. The character '`y`' indicates that the operator is associative in that direction. For example, an operator of type '`xfy`' is a right-associative, infix operator.

To cancel the operator properties of *Name* (if any) set *Precedence* to 0.

For more details, see Section 8.1.5 [ref-syn-ops], page 165

Exceptions

`instantiation_error`
: *Precedence*, *Type* or *Name* is a variable

`type_error`
: *Precedence* is not an integer or *Type* is not an integer or *Name* is not an atom

`domain_error`
: *Precedence* is not in the range 1-1200

See Also

`current_op/3`

Section 8.1.5 [ref-syn-ops], page 165

18.3.124 `open/[3,4]`

Synopsis

`open(+`*FileSpec*`, +`*Mode*`, -`*Stream*`)`

`open(+`*FileSpec*`, +`*Mode*`, +`*Options*`, -`*Stream*`)`

Creates a Prolog stream by opening the file *FileSpec* in mode *Mode* with options *Options*.

Arguments

FileSpec file_spec
: a file specification (see Section 8.6 [ref-fdi], page 205).

Mode one of `[read,write,append]`
: an atom specifying the open mode of the target file. One of:

  `read` open *FileSpec* for input.

  `write` open *FileSpec* for output. A new file is created if *FileSpec* does not exist. If the file already exists, then it is set to empty and its previous contents are lost.

  `append` opens *FileSpec* for output. If *FileSpec* already exists, adds output to the end of it. If not, a new file is created.

Options list
: a list of zero or more of the following.

  `text` Specifies that the file stream is a text stream. This sets the line border code to ⟨LFD⟩, the file border code to -1, and turns on trimming. This is the default.

  `binary` Specifies that the file stream is a binary stream. This sets the line border code to none, the file border code to -1, the format to `variable`, and turns off trimming.

  `record(`*Size*`)`
  : *Size* is an integer value to specify the maximum record (line) size in the file. This also sets the internal buffer size to be used for input/output options on the stream to *Size*. If *Size* is 0, the opened stream operates in non-buffered mode. The value of *Size* should be greater than or equal to 0.

    Under UNIX, the default is 256 for tty streams and 8192 for other stream.

`end_of_line(`*`EolCode`*`)`

EolCode is an integer value to specify the line (record) border code for the stream. *EolCode* is

-1 Indicates there is no line border code.

Charcode ASCII code for EOL character. Default = ⟨LFD⟩ (ASCII code for ⟨LFD⟩).

If an output predicate writes out the character whose code is the line border code of the stream, the Prolog system terminates the output record according to the format of the stream.

`end_of_file(`*`EofCode`*`)`

EofCode is an integer value to specify the file border code for an input stream.

-2 Indicates there is no file border code for the stream. Reading at the end of file is same as reading past end of file.

The file border code is the value to be returned to an input predicate when an input stream reaches the end of file. The default file border code is -1.

`eof_action(`*`Action`*`)`

Specifies what to do for reading past end of file. This option has no effect on an output stream. *Action* is one of the following.

`error` It's an error to read past end of file. This is the default for text binary streams.

`eof_code` Return file border code as set in `end_of_file` option for reading past end of file.

`reset` Reset the stream and make an attempt to read for input past end of file. This is the default for tty stream.

`overflow(`*`OvAction`*`)`

Specifies what to do when output overflows the current record size. This option has no effect on an input stream. *OvAction* is one of the following.

`error` It's an error.

`truncate` Discard the overflow characters.

`flush` Write out the overflow partial record (line). No characters are discarded. This is the default under UNIX and Windows.

`seek(SeekOption)`

Request seeking method that will be performed on the file. *SeekOption* is defined as follows:

`error` It's an error to issue a seeking command on the stream. This is the default for a tty stream.

`previous` The seeking request will be made only to a previous input/output position. `stream_position/3` is the only predicate that can be used to seek on the stream. This is the default for both text and binary streams.

`byte` Seeking to an arbitrary byte position on the stream. This option also permits `seek(previous)`. Both `stream_position/3` and `seek/4` work on the stream.

`record` Seeking to the beginning of an arbitrary record in the file stream. This option is not available under UNIX or Windows.

`flush(`*`FlushType`*`)`
Request flushing method for an output stream. This option has no effect on an input stream. It can be one of the following.

`error` It's an error to try to flush an output stream.

`flush` Write out all the characters buffered. This is the default under UNIX and Windows.

`trim` Turns on the trimming on the file stream. Trimming means that trailing blanks are deleted in input records. The default is no trimming. See `format` below.

`system(`*`SysAttrs`*`)`
This option is provided to allow extensions.

SysAttrs
must be an atom and is passed to the `QU_open()` function, which can be redefined by the user. The default version of `QU_open()` will report an error, causing a `permission_error` to be raised, if system(*SysAttrs*) is specified.

`format(`*`Format`*`)`
Specifies
the file format (see Section 10.5.3.3 [fli-ios-sst-fmt], page 438). For Prolog running under UNIX and Windows, the default format is `format(delimited(lf))` for text stream, `format(variable)` for binary stream, and `format(delimited(tty))` for tty file. Users will not normally need to use the `format(`*`Format`*`)` options directly. *Format* is one of:

`variable` Each record in the file has its own length. There are no delimiter characters between records. The Prolog system removes the trailing blank characters for each input record it reads if the `trim` option is set.

`delimited(lf)`
: For an application program's point of view, a single ⟨LFD⟩ (ASCII code 10) terminates each record in the file. Under Windows, however, what's actually stored in the file is the sequence ⟨RET⟩⟨LFD⟩.

`delimited(tty)`
: *FileSpec* is a terminal device, a pseudo-terminal device, or a terminal emulator. The Prolog input/output system treats this format like `QP_DELIM_LF` as far as record termination is concerned.

If one of these delimiters is specified, the Prolog system removes the delimiter characters at the end of record for input. The line border code (specified by `end_of_line` option) is returned instead as the character code at the end of the record. Prolog system also puts delimiter characters at the end of record when a record is written out.

When no format has been specified, the format is decided as follows: if there is no line border code and trimming is off, then `format(variable)` is used; otherwise `format(delimited(lf))` is used.

Stream *stream_object* the resulting opened Prolog stream.

Description

`open/3` is equivalent to `open/4` with *Options*=`[]`.

`open/4` is designed to work on various file systems under different operating systems.

Stream is used as an argument to Prolog input and output predicates.

Stream can also be converted to the corresponding foreign representation through `stream_code/2` and used in foreign code to perform input/output operations.

Exceptions

`domain_error`
: *Mode* is not one of `read`, `write` or `append`. *Options* has an undefined option or an element in *Options* is out of the domain of the option.

`instantiation_error`
: *FileSpec* or *Mode* is not instantiated. *Options* argument is not ground.

`type_error`

FileSpec or *Mode* is not an atom type. *Options* is not a list type or an element in *Options* is not a correct type for open options.

`existence_error`
: The specified *FileSpec* does not exist.

`permission_error`
: Can not open *FileSpec* with specified *Mode* and *Options*.

`resource_error`
: There are too many files opened.

Comments

If an option is specified more than once the rightmost option takes precedence.

Prolog streams are in general classified as tty streams, text streams, or binary streams. A Prolog stream is a tty stream if the format of the stream is set to `format(delimited(tty))`, or if no format is specified and *FileSpec* refers to a terminal (decided by the function `isatty(3)`). Prolog provides a special service to print prompts for a tty input stream. A text stream corresponds to a text file. The Prolog system removes the control characters of the text stream. A binary stream is a stream of bytes; the *Prolog system* returns the actual characters stored in the file. Specifying `binary` or `text` along with `trim` and `end_of_line` options will result in a hybrid of binary and text streams.

Defaults are provided for *Options* in `QU_stream_param()` function. This description is based on those input/output defaults.

Format is seldom set by the user. It is only useful in case the user has redefined `QU_open()`.

Examples

1. Opening a stream that behaves like a C standard I/O stream without maintaining correct line count and line position.

```
open(FileSpec, Mode, [binary, seek(byte),
        eof_action(eof_code)], Stream).
```

2. Opening a non-buffered stream

```
open(FileSpec, Mode, [record(0)], Stream).
```

3. On UNIX systems, if *FileSpec* is '`/dev/tty`', it means that the file is the default tty for the Prolog system. Terminal is used interactively.

See Also

`open_null_stream/1`, `close/1`, `QP_prepare_stream/[3,4]` `QP_fopen()`, `QP_fdopen()`, `QU_open()`

Section 10.2.3.3 [fli-emb-how-iou], page 374

18.3.125 `open_null_stream/1`

Synopsis

`open_null_stream(-`*`Stream`*`)`

opens an output stream that is not connected to any file and unifies its stream object with *Stream*.

Arguments

Stream stream_object

Description

Characters or terms that are sent to this stream are thrown away. This predicate is useful because various pieces of local state are kept for null streams: the predicates `character_count/2`, `line_count/2`, and `line_position/2` can be used on these streams (see Section 8.7.8 [ref-iou-sos], page 230).

If *Stream* is fully instantiated at the time of the call to `open_null_stream/1`, the call simply fails.

See Also

`character_count/2`, `line_count/2`, `line_position/2`

18.3.126 `otherwise/0`

Synopsis

`otherwise`

Always succeeds (same as `true/0`).

Description

`otherwise/0` is useful for laying out conditionals (see Section 8.2.7 [ref-sem-con], page 186) in a readable way.

Examples

```
( test1 ->
    goal1
| test2 ->
    goal2
| otherwise ->
    goal3
)
```

18.3.127 `peek_char/[1,2]`

Synopsis

`peek_char(-`*`Char`*`)`

`peek_char(+`*`Stream`*`, -`*`Char`*`)`

looks ahead for next input character on the current input stream or on the input stream *Stream*.

Arguments

Stream stream_object
: a valid Prolog stream.

Char char
: the resulting next input character available on the stream.

Description

`peek_char/[1,2]` looks ahead of the next input character of the specified input stream and unifies the character with *Char*. The peeked character is still available for subsequent input on the stream.

Example

<<NEEDS EXAMPLE>>

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: This is an attempt to read past end of file, or some operating system dependent error occurred in reading.

Comments

It is safe to call `peek_char/[1,2]` several times without actually inputting any character. For example:

```
| ?- peek_char(X), peek_char(X), get0(X).
|: a

X = 97
```

See Also

`get0/[1,2]`, `get/[1,2]`, `open/[3,4]`

Section 8.7 [ref-iou], page 214

18.3.128 phrase/[2,3]

Synopsis

`phrase(+*PhraseType*, +*List*)`

`phrase(+*PhraseType*, +*List*, *Rest*)`

Used in conjunction with a grammar to parse or generate strings.

Arguments

PhraseType callable
: non-variable, name of a phrase type. [MOD]

List list
: a list of symbols — tokens or character codes.

Rest list
: a tail of *List*; what remains of *List* after *PhraseType* has been found.

Description

Runs through the grammar rules checking whether there is a path by which *PhraseType* can be rewritten as *List*.

If *List* is bound, this goal corresponds to using the grammar for parsing. If *List* is unbound, this goal corresponds to using the grammar for generation.

`phrase/2` succeeds when the list *List* is a phrase of type *PhraseType* (according to the current grammar rules), where *PhraseType* is either a non-terminal or, more generally, a grammar rule body. This predicate is a convenient way to start execution of grammar rules.

`phrase/3` succeeds when the portion of *List* between the start of *List* and the start of *Rest* is a phrase of type *PhraseType* (according to the current grammar rules), where *PhraseType* is either a non-terminal or, more generally, a grammar rule body.

`phrase/3` allows variables to occur as non-terminals in grammar rule bodies, just as `call/1` allows variables to occur as goals in clause bodies.

Exceptions

`instantiation_error`
: *PhraseType* is not bound.

`type_error`
: *PhraseType* is not callable.

Examples

See example in Section 8.16.3 [ref-gru-exa], page 300.

See also

`-->/2`, `'C'/3`, `expand_term/2`, `term_expansion/2` Section 8.16 [ref-gru], page 298

18.3.129 `portray/1` *hook*

Synopsis

```
:- multifile portray/1.

portray(+Term)
```

A way for the user to over-ride the default behavior of `print/1`.

Arguments

Term term

Description

Note that `print/1` always calls `portray` in module `user`. Therefore, to be visible to `print/1`, `portray` must either be defined in or imported into module `user`. See the reference page for `print/1` for information on what happens if `portray/1` fails.

If you would like lists of character codes printed by `print/1` using double-quote notation, you should include `library(printchars)` (see Section 12.13 [lib-abs], page 641) as part of your version of `portray/1`.

If `portray/1` is defined, it is called from:

1. `print/1` (default is `write/1`)
2. to print the variable bindings after a question has succeeded (default is `writeq/1`. see '`QU_messages`' for full details)
3. to print a goal during debugging (default is `writeq/1`. See '`QU_messages`' for full details)

Tips

See Also

`print/1` Section 8.7.4.5 [ref-iou-tou-por], page 219

18.3.130 portray_clause/1

Synopsis

`portray_clause(+Clause)`

Writes *Clause* to the current output stream. Used by `listing/[0,1]`.

Arguments

Clause term

Description

The operation used by `listing/0` and `listing/1`. *Clause* is written to the current output stream in exactly the format in which `listing/1` would have written it, including a terminating full-stop.

If you want to print a clause, this is almost certainly the command you want. By design, none of the other term output commands puts a full-stop after the written term. If you are writing a file of facts to be loaded by the Load Predicates, use `portray_clause/1`, which attempts to ensure that the clauses it writes out can be read in again as clauses.

The output format used by `portray_clause/1` and `listing/1` has been carefully designed to be clear. We recommend that you use a similar style. In particular, never put a semicolon (disjunction symbol) at the end of a line in Prolog.

Exceptions

Always succeeds without error.

Example

```
| ?- portray_clause((X:- a -> b ; c)).
A :-
        (   a ->
            b
        ;   c
        ).

X = _3185
```

```
| ?- portray_clause((X:- a -> (b -> c ; d ; e); f)).
A :-
        (   a ->
            (   b ->
                c
            ;   d
            ;   e
            )
        ;   f
        ).

X = _3295

| ?- portray_clause((a:-b)).
a :-
        b.

yes

| ?- portray_clause((a:-b,c)).
a :-
        b,
        c.

yes

| ?- portray_clause((a:-(b,!,c))).
a :-
        b,
        !,
        c.

yes
```

See Also

`listing/[0,1]`, `read/[1,2]`

Section 8.7.4.6 [ref-iou-tou-pcl], page 220

18.3.131 `predicate_property/2`

Synopsis

`predicate_property(*Callable, *PredProperty)`

Unifies *PredProperty* with a predicate property of an existing predicate, and *Callable* with the most general term that corresponds to that predicate.

Arguments

Callable callable [MOD]
: the skeletal specification (see Section 8.1.7 [ref-syn-spc], page 169) of a loaded predicate

PredProperty term
: the various properties associated with *Callable*. Each loaded predicate will have one or more of the properties:

Property	Comments
`compiled`	can have the `multifile` property
`interpreted`	can have either the `dynamic` or `multifile` property or both
`built_in`	
`multifile`	
`dynamic`	
`exported`	on the public predicate list of its source module
`imported_from(Module)`	imported into the source module from *Module*)
`foreign`	
`meta_predicate(Term)`	*Term* was specified in a meta_predicate declaration. Thus *Term* consists of the principal functor name followed by mode declarations for its arguments. For example: `mysort(:, +, -)` See Section 8.13.17 [ref-mod-met], page 284 for further information.
`volatile`	not to be saved in QOF files
`locked`	not visible in the debugger due to use of '`-h`' option to `qpc`

`has_advice`
: advice has been added for the predicate

`checking_advice`
: advice checking is enabled for the predicate

Description

- If *Callable* is instantiated then `predicate_property/2` successively unifies *PredProperty* with the various properties associated with *Callable*.
- If *PredProperty* is bound to a valid predicate property, then `predicate_property/2` successively unifies *Callable* with the skeletal specifications of all loaded predicates having *PredProperty*.
- If *Callable* is not a loaded predicate or *PredProperty* is not a valid predicate property, the call fails.
- If both arguments are unbound, then `predicate_property/2` can be used to backtrack through all currently defined predicates and their corresponding properties.

Examples

- Predicates acquire properties when they are defined:

```
| ?- [user].
| :- dynamic p/1.
| p(a).
| end_of_file.
% user compiled 0.117 sec 296 bytes

yes
| ?- predicate_property(p(_), Property).

Property = (dynamic) ;

Property = interpreted ;

no
```

- To backtrack through all the predicates `P` imported into module `m` from any module:

```
| ?- predicate_property(m:P, imported_from(_)).
```

- To backtrack through all the predicates `P` imported into module `m1` from module `m2`:

```
| ?- predicate_property(m1:P, imported_from(m2)).
```

- To backtrack through all the predicates `P` exported by module `m`:

```
| ?- predicate_property(m:P, exported).
```

- A variable can also be used in place of a module atom to find the names of modules having a predicate and property association:

```
| ?- predicate_property(M:f, imported_from(m1)).
```

will return all modules `M` that import `f/0` from `m1`.

Please note: All dynamic predicates are currently interpreted.

See Also

`fileerrors/0`, `nofileerrors/0`, `gc/0`, `compile/1`, `module/[1,2]`, `foreign/[2,3]`, `meta_predicate/1`, `volatile/1`, `add_advice/3`, `check_advice/[0,1]`, `current_predicate/2`

Section 8.13.14.2 [ref-mod-ilm-vis], page 281

18.3.132 print/1 *hookable*

Synopsis

`print(+`*Term*`)`

`print(+`*Stream*`, +`*Term*`)`

Writes *Term* to the current output stream, or *Stream*. Can be redefined with the hook `portray/1`.

Arguments

Stream stream_object
Term term

Description

By default, the effect of this predicate is the same as that of `write/1`, but you can change its effect by providing clauses for the predicate `portray/1`.

If *Term* is a variable, then it is printed using `write(`*Term*`)`.

Otherwise the user-definable procedure `portray(`*Term*`)` is called. If this succeeds, then it is assumed that *Term* has been printed and `print/1` exits (succeeds). Note that `print/1` always calls `portray/1` in module `user`. Therefore, to be visible to `print/1`, `portray/1` must either be defined in or imported into module `user`.

If the call to `portray/1` fails, and if *Term* is a compound term, then `write/1` is used to write the principal functor of *Term* and `print/1` is called recursively on its arguments. If *Term* is atomic, it is written using `write/1`.

When `print/1` has to print a list, say [*Term1*,*Term2*,. . . ,*TermN*], it passes the whole list to `portray/1`. As usual, if `portray/1` succeeds, it is assumed to have printed the entire list, and `print/1` does nothing further with this term. Otherwise `print/1` writes the list using bracket notation, calling `print/1` on each element of the list in turn.

Since [*Term1*,*Term2*,. . . ,*TermN*] is simply a different way of writing .(*Term1*,[*Term2*,. . . ,*TermN*]), one might expect `print/1` to be called recursively on the two arguments *Term1* and [*Term2*,. . . ,*TermN*], giving `portray/1` a second chance at [*Term2*,. . . ,*TermN*]. This does *not* happen; lists are a special case in which `print/1` is called separately for each of *Term1*,*Term2*,. . . *TermN*.

If you would like lists of character codes printed by `print/1` using double-quote notation, you should include `library(printchars)` (see Section 12.13 [lib-abs], page 641) as part of your version of `portray/1`.

Exceptions

Succeeds without error, except for any errors that may occur in the execution of `portray/1`.

See Also

`portray/1`, `library(printchars)`

18.3.133 print_message/2 *hookable*

Synopsis

`print_message(+`*Severity*`, +`*MessageTerm*`)`

Print a *Message* of a given *Severity*. The behavior can be customized using the two hooks `generate_message_hook/3` and `message_hook/3`.

Arguments

Severity atom
: Unless the default system portrayal is overidden with `message_hook/3`, *Severity* must be one of

Value	Prefix
`informational`	'% '
`warning`	'* '
`error`	'! '
`help`	no prefix
`silent`	no prefix

MessageTerm term
: any term

Description

Messages are parsed according to the definite clause grammars in `qplib('QU_messages')`, which defines `'QU_messages':generate_message/3`. If `generate_message(`*MessageTerm*`,L,[])` is true, the message is printed according to the transformation *L*; otherwise, the message is considered to be undefined.

An unhandled exception message *E* calls `print_message(error, `*E*`)` before returning to the top level. The convention is that an error message is the result of an unhandled exception. Thus, an error message should only be printed if `raise_exception/1` does not find a handler and unwind to the top-level.

All messages from the system are printed using this predicate. Means of intercepting these messages before they are printed are provided.

`print_message/2` always prints to `user_error`. Messages can be redirected to other streams using `message_hook/3` and `print_message_lines/3`

"Silent" messages do not get translated or printed. They do not go through `generate_message/3` or `generate_message_hook/3` but they can be intercepted with `message_hook/3`.

See Also

`message_hook/3`, `generate_message/3`, `print_message_lines/3`, `generate_message_hook/3`

Section 8.20.2 [ref-msg-tbm], page 327

18.3.134 print_message_lines/3

Synopsis

`print_message_lines(+`*Stream*`, +`*Prefix*`, +`*Lines*`)`

Print the *Lines* to *Stream*, preceding each line with *Prefix*. Note that `print_message_lines/3` only succeeds if *Lines* is a *list of pair*.

Arguments

Stream stream_object
: Any valid output stream.

Prefix term
: Any term.

Lines list
: is of the form `[`*Line1*`, `*Line2*`, ...]`, where each *Linei* is of the form `[`*Control_1-Args_1*`,`*Control_2-Args_2*`, ...]`.

Description

This command is intended to be used in conjunction with `message_hook/3`. After a message is intercepted using `message_hook/3`, this command is used to print the lines. If the hook has not been defined, the arguments are those provided by the system.

`print_message_lines/3` is a simple failure driven loop over the *Lines* data structure, implemented as:

```
:-use_module(library(basics),[member/2]).

print_message_lines(Stream,Prefix,Lines):-
    member(Line,Lines),
    format(Stream,'~N~w',[Prefix]),
    (   member(C-A,Line),
        format(Stream,C,A)
    ;   nl(Stream)
    ),
    fail.
print_message_lines(_,_,_).
```

Exceptions

Any exception that `format/3` might raise.

Examples

A typical use of this would be when using the user defined predicate, `message_hook/3` to redirect output. For example:

```
message_hook(_,_,Lines):-
    my_stream(MyStream),
    print_message_lines(MyStream,'',Lines).
```

See Also:

`message_hook/3`, `print_message/2`, `generate_message/3`, `query_hook/6`

18.3.135 `profile/[0,1,2,3]` *development*

Synopsis

`profile`

`profile(+`*`Goal`*`)`

`profile(+`*`Goal`*`,+`*`Interval`*`)`

`profile(+`*`Goal`*`,+`*`Interval`*`,+`*`Sigtype`*`)`

Turns on the profiler and profiles the execution of *Goal*.

Arguments

Goal callable [MOD]
: The goal to profile.

Interval integer
: The hit interval in microseconds (default 10000).

Sigtype one of `[with_sigprof,with_sigalrm,with_sigvtalrm]`
: Which signal to use (default `SIGPROF`).

Description

`profile/0` turns the profiler on and resets all counts accumlated from previous runs of the profiler. Counts of the number of call, choice points created and redos tried are maintained by the profiler.

`profile/1` performs this same action and then proceeds to execute *Goal*, in addition monitoring timing data. Once execution of the goal has completed, the profiling results can be displayed with `show_profile_results/[0,1,2]`.

Note that the profiler cannot be used when in debugging mode.

This predicate is not supported in runtime systems.

See Also

`noprofile/0`, `show_profile_results/[0,1,2]`

18.3.136 `prolog_flag/[2,3]`

Synopsis

`prolog_flag(*FlagName, *Value)`

FlagName is a flag, which currently is set to *Value*.

`prolog_flag(+FlagName, -OldValue, +NewValue)`

Unifies the current value of *FlagName* with *OldValue* and then sets the value of the flag to *NewValue*.

Arguments

FlagName atom
Value atomic
OldValue atomic
NewValue atomic

Currently, the supported *FlagName*s and *Value*s for both `prolog_flag/2` and `prolog_flag/3` are:

FlagNames **Values**

`character_escapes`
: `on` or `off`

`debugging`
: `trace`, `debug`, `zip`, or `off`

`fileerrors`
: `on` or `off`

`gc` `on` or `off`

`gc_margin`
: non-negative integer

in thousands of bytes

`gc_trace` `on` or `off`

`unknown` `error` or `fail`

`syntax_errors`
: (see Section 8.19.4.10 [ref-ere-err-syn], page 322)

`single_var`
: `on` or `off`

`discontiguous`
: `on` or `off`

`multiple` `on` or `off`

Values available only to `prolog_flag/2` (query-only) are:

FlagNames

Values

`add_ons` an atom containing the list of add-on products that are statically linked into the Prolog system. If no add-ons are part of the system, the empty atom `''` is returned.

`host_type`
: the host type, which is generally a hardware-operating system combination. This `prolog_flag` is used to create the `system file_search_path/2` facts (see Section 8.6.1.4 [ref-fdi-fsp-pre], page 210 and Section 8.6.1.5 [ref-fdi-fsp-sys], page 213).

`quintus_directory`
: the absolute name of the Quintus directory. The Quintus directory is the root of the entire Quintus installation hierarchy.

`runtime_directory`
: the absolute name of the directory where all the Prolog executables reside.

`version` the version of the Prolog being run.

`system_type`
: `development` or `runtime`.

Description

To inspect the value of a flag without changing it, use `prolog_flag/2` or the following idiom, where FlagName is bound to one of the valid flags above.

```
| ?- prolog_flag(FlagName, Value, Value).
```

Use `prolog_flag/2` to query and `prolog_flag/3` to set values.

`prolog_flag/3` can be used to save flag values so that one can return a flag to its previous state. For example:

```
...
prolog_flag(debugging,Old,on), % Save in Old and set
...
prolog_flag(debugging,_,Old),  % Restore from Old
...
```

The read-only `prolog_flag/2` flags `add_ons`, `host_type`, `quintus_directory`, and `runtime_directory` represent information set by the `qsetpath` program. For more detail on the `qsetpath` and `qgetpath` utilities, see and .

Backtracking

`prolog_flag/2` enumerates all valid flagnames of a given current value, or all pairs of flags and their current values. It is not a way to find out non-current values for a flag.

Exceptions

`instantiation_error`
: In `prolog_flag/3`, *FlagName* unbound, *or*
NewValue unbound and not identical to *OldValue*.

`type_error`
: *FlagName* is not an atom.

`domain_error`
: In `prolog_flag/3`, *FlagName* bound to an atom that does not represent a supported flag, *or*
NewValue bound to atom that does not represent a valid value for *FlagName*.

See Also

`gc/0`, `nogc/0`, `style_check/1`, `no_style_check/1`, `unknown/2`, `fileerrors/0`, `nofileerrors/0`

Section 8.10 [ref-lps], page 244

18.3.137 `prolog_load_context/2`

Synopsis

`prolog_load_context(+`*`Key`*`, -`*`Value`*`)`

`prolog_load_context(*`*`Key`*`, *`*`Value`*`)`

Finds out the context of the current load.

Arguments

Key atom

Value atom

Description

You can call `prolog_load_context/2` from an embedded command or by `term_expansion/2` to find out the context of the current load. If called outside the context of a load, it simply fails.

Key	Value
`module`	the module you are compiling into
`file`	absolute filename of the file being compiled
`stream`	the stream you are compiling from
`directory`	directory of the file on which the stream is open
`term_position`	a stream position object referring to the position of the clause just read

Backtracking

This predicate is meant to be used in the mode (‘`+`’, ‘`-`’), but it is also possible to backtrack through it.

See Also

`load_files/[1,2]`

Section 8.10 [ref-lps], page 244

18.3.138 prompt/[2,3]

Synopsis

`prompt(-OldPrompt, +NewPrompt)`

`prompt(+Stream, -OldPrompt, +NewPrompt)`

Queries or changes the prompt string of the current input stream or an input stream *Stream*.

Arguments

Stream stream_object
: a valid Prolog input stream.

OldPrompt atom
: the old prompt atom of the stream.

NewPrompt atom
: the new prompt atom of the stream.

Description

A *prompt atom* is a sequence of characters that indicates the Prolog system is waiting for input when a "Read" or "Get" predicate is called. If an input stream connected to a terminal is waiting for input at the beginning of a line (at line position 0), the prompt atom of the stream will be printed through an output stream associated with the same terminal.

Prolog sets the prompt of the input stream to '`|:`'. This is the prompt that can be changed by invoking `prompt/[2,3]`. Unlike state changes such as those implemented as prolog flags, the scope of a prompt change is a goal typed at the toplevel. Therefore, the change is in force only until returning to the toplevel (prompt = '`| ?- `').

To *query* the current prompt atom of a stream, *OldPrompt* and *NewPrompt* should be the same unbound variable. For example:

```
prompt(X, X).  prompt(user_input, X, X).
```

To *set* the prompt of a stream, *NewPrompt* should be an instantiated atom. `prompt/2` queries or changes the prompt on the current Prolog input stream.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`instantiation_error`
: *NewPrompt* argument is not instantiated

`type_error`
: *NewPrompt* is not an atom type

Comments

The "Load" predicates change the prompt of the input stream during the time operations are performed: If a built-in loading predicate is performed on the module `user` (such as `compile(user)`, etc.), the prompt string of the standard Prolog input stream, `user_input` (`user`) is set to '`| `'. This prompt is not affected by `prompt/[2,3]`.

`prompt/3` succeeds for any valid input stream. If the input stream is not a tty format stream, the Prolog system does not print out the prompt string when it is waiting input from the stream.

Normally prompts only appear on `user_error` when the system is waiting for input on `user_input`. These prompts are suppressed when `user_input` is not connected to a terminal, unless the '`+tty`' option to `prolog(1)` was specified. (See Section 10.5.4 [fli-ios-tty], page 444.)

For prompts to be used on streams other than `user_input` or `user_error`, the C function `QP_add_tty()` must be used.

See Also

`QP_add_tty()`, `read/[1,2]`, `read_term/[2,3]`, `get/[1,2]`, `get0/[1,2]` Section 10.5 [fli-ios], page 433

18.3.139 `public/1` *declaration*

Synopsis

`:- public` *+Term*

Dummy declaration for backwards compatibility.

Arguments

Term term [MOD]

18.3.140 put/[1,2]

Synopsis

`put(+Char)`

`put(+Stream, +Char)`

Evaluates the integer expression *Char*, and writes the lower 8-bits to the current output stream or to *Stream*.

Arguments

Stream stream_object
: a valid Prolog output stream

Char expr
: a legal character code or an integer expression. A useful form of integer expression for this argument is a single character following '`0'`', such as `0'a`, `0'b`, etc.

Description

`put/[1,2]` writes out the least significant 8 bits of the evaluated *Char* to the specified output stream unless *Char* is evaluated to be the line border code of the stream. The character written out is usually stored in the buffer of the stream. If the buffer overflows, it is written out to the disk. If the evaluated *Char* is the same as the line border code of the output stream, the operation works like `nl/[0,1]`. The default line border code for a text stream and a tty stream is the linefeed character (ASCII code 10).

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`instantiation_error`
: *Char* is not instantiated.

`type_error`
: *Char* is not an integer type.

`permission_error`
: There is an error in the bottom layer of write function of the stream.

See Also

`nl/[0,1]`, `skip_line/[0,1]`, `open/[3,4]`

Section 8.7 [ref-iou], page 214

18.3.141 `query_abbreviation/3` *extendable*

Synopsis

```
:- multifile 'QU_messages':query_abbreviation/3.

query_abbreviation(+Tag, -Prompt, -Pairs)
```

A way to specify one letter abbreviations for responses to queries from the Prolog System.

Arguments

Tag atom This indicates which query type.

Prompt atom
An atom indicating appropriate abbreviations.

Pairs list of pair
A list of word-abbreviation pairs.

Description

Prolog only asks for keyboard input in a few different ways. These are enumerated in the clauses for `query_abbreviation/3` in the module `messages(language('QU_messages'))`. These clauses specify valid abbreviations for a given key word. For example,

```
query_abbreviation(yes_or_no,'(y/n)',[yes-"yY",no-"nN"]).
```

a French translator might decide that the letters '`O`' and '`o`' are reasonable abreviations for '`oui`' (yes), and therefore write

```
query_abbreviation(yes_or_no,'(o/n)',[yes-"oO",no-"nN"]).
```

For an example of how this is used with `query_hook/6`, see the reference page for `query_hook_example`.

See Also:

Section 8.20.3.4 [ref-msg-umf-int], page 331, and the reference page for `query_hook/6`

18.3.142 query_hook/6 *hook*

Synopsis

```
:- multifile query_hook/6.

query_hook(+QueryClass, +Prompt, +PromptLines, +Help, +HelpLines, -Answer)
```

Provides a method of overriding Prolog's default keyboard based input requests. The query hook is used by the Quintus User Interface.

Arguments

QueryClass term
: determines the allowed values for the atom *Answer*.

  If *QueryClass* **is:** — *Answer* **must be:**

  `yes_or_no(Question)` — `yes` or `no`.

  `toplevel` — `yes` or `no`

  `yes_no_proceed` — `yes`, `no`, or `proceed`.

Prompt list of pair
: A message term.

PromptLines list of pair
: The message generated from the *Prompt* message term.

Help term
: A message term.

HelpLines list of pair
: The message generated from the *Help* message term.

Answer term
: see *QueryClass*

Description

This provides a way of overriding Prolog's default method of interaction. If this predicate fails, Prolog's default method of interaction is invoked.

The default method first prints out the prompt, then if the response from the user is not one of the allowed values, the help message is printed.

It is useful to compare this predicate to `message_hook/3`, since this explains how you might use the *Prompt*, *PromptLines*, *Help*, *HelpLines*.

Exceptions

An exception raised by this predicate causes an error message to be printed and then the default method of interation is invoked. In other words, exceptions are treated as failures.

Examples

If Prolog is looking for a yes-no response to one question '`Done?`', as in the toplevel, this request for input can be captured

```
query_hook(toplevel,_,_,_,_,Answer):-
    my_yes_no('Done?',Answer).
```

where `my_yes_no/2` binds Answer to either `yes` or `no`.

Here is roughly how the default method works. Notice the interaction with `query_abbreviation/3`.

```
query_hook(QueryClass,_,PromptLines,_,HelpLines,Answer):-
    'QU_messages':query_abbreviation(QueryClass,
                                     AbbreviationPrompt,
                                     Pairs),
    repeat,
        (   print_message_lines(user_output,'',PromptLines),
            (   AbbreviationPrompt == ''
            ->  write(Stream,' ')
            ;   format(Stream,' ~w ',[AbbreviationPrompt])
            ),
            flush_output(Stream),
            get0(C),
            member(Answer-Abrv,Pairs),
            member(C,Abrv),
            !
        ;   print_message_lines(Stream,'',HelpLines),
            fail
        ).
```

Tips

See Also

`query_abbreviation/3`, `message_hook/3`, `print_message_lines/3`

Section 8.20 [ref-msg], page 325

18.3.143 `raise_exception/1`

Synopsis

`raise_exception(+`*`Exception`*`)`

Raise an exception (that might be intercepted by `on_exception/3`).

Arguments

Exception nonvar
: Any term.

Description

A call to `raise_exception/1` can *never* backtrack, fail or succeed. Rather, `raise_exception/1` searches for an ancestor of the current goal, *ProtectedGoal*, which is of the form:

```
on_exception(E,ProtectedGoal,Handler)
```

The first argument, *E*, unifies with *Exception*. It then executes the *Handler* instead of the *ProtectedGoal*. It will always find a handler at the top level, which prints out a message corresponding to the exception. See Section 8.20.2 [ref-msg-tbm], page 327 for a discussion on how exceptions are printed.

Exceptions

`instantiation_error`
: when *Exception* is unbound. When a built-in predicate detects an error situation, it causes an exception to be raised.

See Also

`on_exception/3`.

Section 8.19 [ref-ere], page 310

18.3.144 `read/[1,2]`

Synopsis

`read(-`*`Term`*`)`

`read(+`*`Stream`*`, -`*`Term`*`)`

Reads the next term from the current input stream or *Stream* and unifies it with *Term*.

Arguments

Stream stream_object
: a valid Prolog stream, which is open for input

Term term
: the term to be read

Description

Term must be followed by a full-stop. The full-stop is removed from the input stream and is not a part of the term that is read.

For the syntax of Prolog terms see Section 8.1.8 [ref-syn-syn], page 171.

The term is read with respect to current operator declarations. See Section 8.1.5 [ref-syn-ops], page 165, for a discussion of operators.

Does not finish until the full-stop is encountered. Thus, if you type at top level

```
| ?- read(X)
```

you will keep getting prompts (first '`|: `', and five spaces thereafter) every time you type ⟨RET⟩, but nothing else will happen, whatever you type, until you type a full-stop.

When a syntax error is encountered, an error message is printed and then `read/1` tries again, starting immediately after the full-stop that terminated the erroneous "term". That is, `read/1` does not fail on a syntax error, but perseveres until it eventually manages to read a term.

If the end of the current input stream has been reached, then `read(X)` will cause *X* to be unified with the atom `end_of_file`

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`syntax_error`

See Also

`read_term/[2,3]`, `prompt/[2,3]`

Section 8.1.8 [ref-syn-syn], page 171

18.3.145 `read_term/[2,3]`

Synopsis

`read_term+`*`Options, -Term`*`)]`

`read_term+`*`Stream, +-Options, -Term`*`)]`

Read a term from the current input stream or from *Stream*, optionally returning extra information about the term.

Arguments

Stream stream_object
: A valid Prolog stream, which is open for input

Term term
: the term that is read

Options list of term
: a list of zero or more of the following:

`syntax_errors(`*`Val`*`)`
: *Val* must be bound to one of the following, indicating what should be done when a syntax error is found:

`quiet`	nothing is printed, and `read_term/[2,3]` fails
`dec10`	a syntax error message is printed, and `read_term/[2,3]` tries to read the next term (this is compatible with DEC-10 Prolog and previous versions of Quintus Prolog)
`fail`	a syntax error message is printed, and `read_term/[2,3]` fails
error	an exception is raised.

The default value if this option is not specified is the current value of the `syntax_errors` prolog flag. The default value for this flag is `dec10`. See `prolog_flag/2` for more information on these flags.

`variable_names(`*`Names`*`)`
: On completion, *Names* is bound to a list of *Name*=*Var* pairs, where each *Name* is an atom indicating the spelling of the name of a variable in the term just read, and *Var* is the corresponding variable. Note that anonymous variables, written as '`_`', are not included in this list.

`singletons(`*Singletons*`)`
: On completion, *Singletons* is bound to a list of *Name*=*Var* pairs, one for each variable only appearing once in the term. Anonymous variables are not included on this list.

`term_position(`*Position*`)`
: On completion, *Position* is the position of the start of the actual term, as might be returned by `stream_position/2`. Any white space and comments before the actual term are not reflected by the position. To find the position of the end of the term, you need only call `stream_position/2`; it will give you the position of the first character after the period ending the term.

`subterm_positions(`*PositionTerm*`)`
: On completion, *PositionTerm* is bound to a position term that describes the position of the term just read and all of its subterms. A position term is of one of the forms listed below. In all these forms, *Start* and *End* are the character positions of first character of the term and the character following the last character of the term, respectively. Similarly *FStart* and *FEnd* specify the start and end of the principle functor of the term. Note that the positions are *character* positions, not position terms as returned by `stream_position/2`.

  Start-End
  : The term corresponding to this position term is either atomic or a variable. *Start* and *End* are the character positions of the first character of the term and the character following the last character of the term, respectively.

  `list_position(`*Start*`,`*End*`,`*Elts*`,`*Tail*`)`
  : The term corresponding to this position term is a list, which was written using bracket notation (for example, `[a,list]`). *Elts* is a list of position terms for each proper element of the list. Tail is the position of the tail of the list (the part following the '`|`'), or the atom `none` if the list has no tail part.

  `string_position(`*Start*`,`*End*`)`
  : The term corresponding to this position term is a list of character codes written as a quoted string (for example, `"a string"`). The positions specified include the quote characters.

  `brace_term_position(`*Start*`,`*End*`,`*Arg*`)`
  : The term corresponding to this position term is of the form `{`*X*`}`. Arg is a position term describing the argument of this term.

`term_position(`*Start*`,`*End*`,`*FStart*`,`*FEnd*`,`*Args*`)`
: The term corresponding to this position term is a compound term not specifically mentioned above. This includes terms written with operators. Args is a list of position terms, one for each argument of the term.

Exceptions

`syntax_error`
: A syntax error is found

`permission_error`
: The input stream cannot be read

`domain_error`
: An illegal option or an invalid stream is specified

`instantiation_error`
: Either *Stream* or *Options*, or one of the elements of the option list, or the argument of the `syntax_errors` option is unbound

`type_error`
: The argument to the `syntax_errors` option is not an atom

Examples

```
| ?- read_term([variable_names(L)], T).
|: append([U|X],Y,[U|Z]) :- append(X,Y,Z).

L = ['U'=_1988,'X'=_2003,'Y'=_2020,'Z'=_2046],
T = (append([_1988|_2003],_2020,[_1988|_2046]):-
         append(_2003,_2020,_2046))

| ?- read_term([subterm_positions(P)], T).
|: foo+bar+baz.

P = term_position(1642,1653,1649,1650,
        [term_position(1642,1649,1645,1646,
             [1642-1645,1646-1649]),
         1650-1653]),
T = foo+bar+baz
```

See Also

`read/[1,2]`, `prompt/[2,3]` `prolog_flag/[2,3]`

Section 8.7 [ref-iou], page 214

18.3.146 `reconsult/1`

Synopsis

`reconsult(+Files)`

Same as `compile/1`

Arguments

Files file_spec or *list of file_spec* [MOD]

18.3.147 `recorda/3`

Synopsis

`recorda(+Key, +Term, -Ref)`

records the *Term* in the internal database as the first item for the key *Key*; a database reference to the newly-recorded term is returned in *Ref*.

Arguments

Key atomic
Term term
Ref db_reference

Description

If *Key* is a compound term, only its principal functor is significant. That is, `foo(1)` represents the same key as `foo(n)`.

Exceptions

`instantiation_error`
: *Key* is not instantiated

`range_error`
: *Ref* is not a database reference or an unbound variable

See Also

`recorded/3`, `recordz/3`, `current_key/2`

Section 8.14.1 [ref-mdb-bas], page 286

18.3.148 `recorded/3`

Synopsis

`recorded(-Key, -Term, +Ref)`

`recorded(+Key, *Term, *Ref)`

searches the internal database for a term recorded under the key *Key* that unifies with *Term*, and whose database reference unifies with *Ref*.

Arguments

Key atomic
Term term
Ref db_reference

Description

If *Ref* is instantiated, then *Key* and *Term* are unified with the key and term associated with *Ref*. Otherwise, If *Key* is a compound term, only its principal functor is significant. That is, `foo(1)` represents the same key as `foo(n)`.

A call to `recorded/3` of the form ('`-`', '`?`', '`+`') will succeed if the expected relation holds. *Key* need not be instantiated.

Backtracking

Can be used to backtrack through all the matching terms recorded under the specified key. Therefore, if you want to match only a single term you should use a cut to prevent backtracking. Alternatively, use the `library(not)` predicate `once/1`.

Exceptions

`type_error`
: *Ref* is not a database reference, or *Key* is a float

`instantiation_error`

See Also

`record/3`, `recorda/3`, `current_key/3`

18.3.149 `recordz/3`

Synopsis

`recordz(+Key, +Term, -Ref)`

Records the term *Term* in the internal database as the last item for the key *Key*; a database reference to the newly-recorded term is returned in *Ref*.

Arguments

Key atomic
Term term
Ref db_reference

Exceptions

`instantiation_error`
: *Key* is not instantiated

`range_error`
: *Ref* is not a db_reference or an unbound variable

See Also:

`recorded/3`, `recorda/3`, `current_key/2`

18.3.150 `remove_advice/3` *development*

Synopsis

`remove_advice(+`*`Goal`*`,+*`*Port`*`,+*`*Action`*`)`

remove the association of an action with entry to a port of a procedure. `remove_advice/3` will only succeed when *Port* is *var* or one of `{call`, `exit`, `done`, `redo`, `fail}`, and *Action* is *var* or *callable*.

Arguments

Goal callable [MOD]
: a term to be unified against a calling goal of existing advice.

Port term
: any term.

Action term [MOD]
: any term.

Description

`remove_advice/3` removes the association of an advice action with a goal and port, undoing the effect of `add_advice/3`.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: if an argument is not sufficiently instantiated.

`type_error`
: if *Goal* or *Action* is not a callable, or a module prefix is not an atom, or *Port* is not an atom.

`domain_error`
: if *Port* is not a valid port.

`permission_error`
: if a specified procedure is built-in.

See Also

`add_advice/3`, `current_advice/3`, `check_advice/[0,1]`, `nocheck_advice/[0,1]`

18.3.151 `remove_spypoint/1` *development*

Synopsis

`remove_spypoint(+`*`Spyspec`*`)`

removes a spypoint from the specified predicate or call.

Arguments

Spyspec compound
: a specification of an individual spypoint. Two forms of spyspec are allowed:

  `predicate(`*`Pred`*`)`
  : A spypoint on any call to *Pred*. *Pred* must be a skeletal predicate specification, and may be module qualified.

  `call(`*`Caller`*`,`*`Clausenum`*`,`*`Callee`*`,`*`Callnum`*`)`
  : A spypoint on the *Callnum* call to *Callee* in the body of the *Clausenum* clause of *Caller*. *Callee* and *Callnum* must be skeletal predicate specifications. *Callnum* and *Clausenum* must be integers, and begin counting from 1. Note that *Callnum* specifies a *lexical* position, that is, the number of the occurrence of *Callee* counting from the beginning of the body of the clause, and ignoring any punctuation.

Description

This predicate is not supported in runtime systems.

See Also

`current_spypoint/1`, `add_spypoint/1`, `spy/1`, `nospy/1`, `debugging/0`

Section 6.1.1 [dbg-bas-bas], page 113

18.3.152 `repeat/0`

Synopsis

`repeat`

Succeeds immediately when called and whenever reentered by backtracking.

Description

Generally used to simulate the looping constructs found in traditional procedural languages. The general form of a repeat loop is as follows:

```
repeat,
   action1,
   action2,
   ...,
   actionn,
   test,
!,
 ... rest of clause body ...
```

The effect of this is to execute *action1* through *actionn* in sequence. The test is then executed. If it succeeds, the loop is (effectively) terminated by the cut (`!`) (which cuts away any alternatives in the clause, including the one created by `repeat/0`). A failure of the test will cause backtracking that will be caught by `repeat/0`, which will succeed again and re-execute the *actions*.

The easiest way to understand the effect of `repeat/0` is to think of failures as "bouncing" back off them causing re-execution of the later goals.

Repeat loops are not often needed; usually recursive procedure calls will lead to code that is easier to understand as well as more efficient. There are certain circumstances, however, in which `repeat/0` will lead to greater efficiency. An important property of Quintus Prolog is that all run-time data is stored in stacks so that any storage that has been allocated during a proof of a goal is recovered immediately on backtracking through that goal. Thus, in the above example, any space allocated by any of the *actions* is very efficiently reclaimed. When an iterative construct is implemented using recursion, storage reclamation will only be done by the garbage collector.

Tips

In the most common use of repeat loops, each of the calls succeeds determinately. It can be confusing if calls sometimes fail, so that backtracking starts before the test is reached, or if calls are nondeterminate, so that backtracking does not always go right back to `repeat/0`.

Note that the repeat loop can only be useful if one or more of the *actions* involves a side-effect — either a change to the data base (such as an assertion) or an I/O operation. Otherwise you would do the same thing each time around the loop (which would never terminate).

Examples

`repeat/0` could have been written in Prolog as follows:

```
repeat.
repeat :- repeat.
```

18.3.153 `restore/1`

Synopsis

`restore(+`*`FileSpec`*`)`

Restores a saved-state.

Arguments

FileSpec file_spec
The name of a QOF file.

Description

`restore(`*`file`*`)` terminates the currently running executable and restarts it with the command line arguments '`+L` *`file old args`*' where *old args* are the arguments specified when the executable was started. *file* is normally a file previously created by a call to save_program, but it can be any QOF file. The '`+L`' option causes *file* to be loaded into the executable as it starts up.

If *file* was created by `save_program/[1,2]`, then it includes information about operator declarations, debugging and advice information, Prolog flags, and `file_search_path` and `library_directory` tables, as well as the Prolog code that was saved. Thus restoring *file* will create the same Prolog state and database that existed at the time the `save_program` was done (assuming that the same executable that was used for the `save_program` is used for the `restore`).

It is also possible to give any QOF file to `restore/1`. In this case, the running executable is reinitialized, and then the QOF file is reloaded into the system. As such QOF files store no state information, the state is the same as in a freshly started Prolog system.

It is not normally useful to use `restore/1` in a runtime system. In a runtime system, command-line arguments are not interpreted by the system, so the re-started runtime system will just begin again at `runtime_entry(start)` and will not load the specified *file* automatically. An application could, if the programmer so chose, pick up the arguments with `unix(argv(L))`, and then take some appropriate action. For example:

```
runtime_entry(start) :-
    unix(argv(['+L',File|_])),
    !,
    load_files(File),
    start_after_restore.
runtime_entry(start) :-
    normal_start.
```

See Section 8.3.1 [ref-pro-arg], page 186 and Section 20.1.1 [too-too-prolog], page 1476 for a description of the '`+L`' option.

Exceptions

`instantiation_error`
: *FileSpec* is not bound.

`type_error`
: *FileSpec* is not a valid file specification.

`domain_error`
: *FileSpec* is not a QOF file.

`permission_error`
: *FileSpec* is not readable

Windows Caveat

Under Windows, it is not possible to replace a running executable with another. Under Windows, `restore/1` will instead start a new sub-process and then terminate the running process. For more details see the Microsoft documentation for `execv()`.

In a Windows command prompt window, the command interpreter does not wait when a process executes an `execv()` library call. Thus after `restore/1`, the program gives the appearance of running in the background.

See Also

`load_files/[1,2]`, `save_modules/2`, `save_predicates/2`, `save_program/[1,2]`

Section 8.5 [ref-sls], page 192

18.3.154 `retract/1`

Synopsis

`retract(+*Clause)`

Removes the first occurrence of dynamic clause *Clause* from module *M*.

Arguments

Clause callable [MOD]
A valid Prolog clause.

Description

`retract/1` erases the first clause in the database that matches *Clause*. *Clause* is retracted in module *M* if specified. Otherwise, *Clause* is retracted in calling module.

`retract/1` is nondeterminate. If control backtracks into the call to `retract/1`, successive clauses matching *Clause* are erased. If and when no clauses match, the call to `retract/1` fails.

Clause must be of one of the forms:

- *Head*
- *Head* `:-` *Body*
- *Module*`:`*Clause*

where *Head* is of type callable and the principal functor of *Head* is the name of a dynamic procedure. If specified, *Module* must be an atom.

`retract(`*Head*`)` means retract the unit-clause *Head*. The exact same effect can be achieved by `retract((`*Head* `:- true))`.

Body may be uninstantiated, in which case it will match any body. In the case of a unit-clause it will be bound to `true`. Thus, for example,

```
| ?- retract((foo(X) :- Body)), fail.
```

is guaranteed to retract all the clauses for `foo/1`, including any unit-clauses, providing of course that `foo/1` is dynamic.

Since `retract/1` is nondeterminate it is important if you only want to retract a single clause to use a cut to eliminate the alternatives generated. See Section 8.14.5.1 [ref-mdb-rcd-efu], page 290 for more information on the use of cuts with `retract/1`.

`retract/1` searches for the clause to remove in the same way that `clause/2` does. (And, like `clause/2`, it uses first argument indexing to speed up this search when possible.) Therefore it is redundant to call `clause/2` immediately before calling `retract/1` on the clause it returns. That is, the call to `clause/2` in the following program fragment can be removed without changing its effect.

```
... clause(H,B), retract((H:-B)), ...
```

The space occupied by a clause that is retracted is reclaimed. The reclamation does not necessarily happen immediately, but is not delayed until backtracking past the call to `retract/1`, as in some implementations.

WARNING: `retract/1` is a nondeterminate procedure. Thus, we can use

```
| ?- retract((foo(X) :- Body)), fail.
```

to retract all clauses for `foo/1`. However, when `retract/1` is used determinately; for example, to retract a single clause, it is crucial that you cut away unintended chice points to avoid "freezing" the retracted *Clause*, disabling tail recursion optimization, or runaway retraction on the unexpected failure of a subsequent goal. See Section 8.14.5.1 [ref-mdb-rcd-efu], page 290 for further discussion.

Exceptions

Same as `assert/1`.

See Also:

`abolish/[1,2]`, `assert/1`, `dynamic/1`, `erase/1`, `retractall/1`.

18.3.155 `retractall/1`

Synopsis

`retractall(+`*Head*`)`

Removes every clause in module *M* whose head matches *Head*.

Arguments

Head callable [MOD]
: Head of a Prolog clause.

Description

Head must be instantiated to a term that looks like a call to a dynamic procedure. For example, to retract all the clauses of `foo/3`, you would write

```
| ?- retractall(foo(_,_,_)).
```

Head may be preceded by a *M*: prefix, in which case the clauses are retracted from module *M* instead of the calling module.

`retractall/1` could be defined (less efficiently) as

```
retractall(Head) :-
        clause(Head, _, Ref),
        erase(Ref),
        fail ; true.
```

or

```
retractall(Head) :-
        retract((Head :- _Body)),
        fail ; true.
```

`retractall/1` is useful for erasing all the clauses of a dynamic procedure without forgetting that it is dynamic; `abolish/1` will not only erase all the clauses, but will also forget absolutely everything about the procedure. `retractall/1` only erases the clauses. This is important if the procedure is called later on.

Since `retractall/1` erases *all* the dynamic clauses whose heads match *Head*, it has no choices to make, and is determinate. If there are no such clauses, it succeeds trivially. None of the variables in *Head* will be instantiated by this command. For example,

```
| ?- listing(baz/2).

baz(a,1).
baz(b,2).
baz(a,3).
baz(b,4).

yes
| ?- retractall(baz(a, X)).

X = _798

| ?- listing(baz/2).

baz(b,2).
baz(b,4).

yes
```

The space previously occupied by a retracted clause is reclaimed. This reclamation is not necessarily immediate, but it is not delayed until backtracking past the call of `retractall/1`, as in some implementations.

Exceptions

`instantiation_error`
: if *Head* or *Module* is uninstantiated.

`type_error`
: if *Head* is not of type callable.

`permission_error`
: if the procedure corresponding to *Head* is built-in or has a static definition.

See Also

`abolish/[1,2]`, `assert/1`, `dynamic/1`, `erase/1`, `retract/1`

Section 8.14.1 [ref-mdb-bas], page 286

18.3.156 runtime_entry/1 *hook*

Synopsis

```
:- multifile runtime_entry/1.

runtime_entry(+Event)
```

This predicate is called upon start-up and exit of stand alone applications.

Arguments

Event one of `[start,abort]`

Description

In a default runtime system, the program starts by executing the goal,

```
runtime_entry(start)
```

When that goal terminates, either by succeeding or by failing, the runtime system terminates.

Similarly, it is possible to specify what is to be done on an abort. An abort happens when a call is made either to the built-in predicate `abort/0` or to the C routine `QP_action(QP_ABORT)`. (By default, a call of `QP_action(QP_ABORT)` happens when a user types `^c` — see Section 9.2.4 [sap-rge-iha], page 358). At this point, the current computation is abandoned and the program is restarted with the goal

```
runtime_entry(abort)
```

Effectively this replaces the original call to `runtime_entry(start)`, so that when this call succeeds or fails, the runtime system terminates.

Users of the module system should ensure that the predicate `runtime_entry/1` is defined in the module `user`, that is, not inside any user-defined module.

See Also

`QP_toplevel()`

Section 10.2.3.1 [fli-emb-how-mai], page 372

18.3.157 `save_modules/2`

Synopsis

`save_modules(+Modules, +File)`

Saves all predicates in *Modules* in QOF format to *File*.

Arguments

Modules atom or *list of atom*
: An atom representing a current module, or a list of such atoms representing a list of modules.

File file_spec
: An atom representing a filename

Description

`save_modules/2` saves the current definitions of all predicates in a module, or list of modules, in QOF format into a file. The modules imported by the saved modules are recorded as dependencies in the QOF file. The foreign files loaded into that module are also recorded as foreign dependencies in the QOF file. The QOF file produced can be loaded into a development system (using `load_files/1`) or it can be linked using `qld`.

When multiple modules are saved into a file, loading that file will import only the first of those modules into the module in which the load occurred.

Exceptions

`instantiation_error`
: *Modules* or *File* is not bound.

`type_error`
: *Modules* is not a valid list of module names, or a single module name, or *File* is not a valid file specification

`permission_error`
: *File* is not writable

`existence_error`
: A given module is not a current module.

See Also:

`load_files/1`, `save_predicates/2`, `save_program/1`, `volatile/1`

18.3.158 `save_predicates/2`

Synopsis

`save_predicates(+PredSpecs, +File)`

Saves the predicates specified by the *PredSpecs* in QOF format to *File*.

Arguments

PredSpecs pred_spec_tree [MOD]
: A list of predicate specifications.

File file_spec
: An atom representing a filename

Description

`save_predicates/2` saves the current definitions of all the predicates specified by the list of predicate specifications in QOF format into a file. The `exported` and `meta_predicate` properties of the predicates are *not* stored in the QOF file. The module of the predicates saved in the QOF file is fixed, so it is not possible to save a predicate from any module `foo`, and reload it into module `bar`. Likewise, the module dependencies or foreign file dependencies of these predicates are *not* saved into the QOF file. A typical use of this would be to take a snapshot of a table of dynamic facts. The QOF file that is written out can be loaded into a development system (using `load_files/1`) or it can be linked with other QOF files using `qld`.

Exceptions

`instantiation_error`
: the list of predicate specifications or the filename is not ground.

`type_error`
`domain_error`
: in the list of predicate specifications or in the filename.

`permission_error`
: the file is not writable, or a predicate is built-in.

`existence_error`
: A predicate is undefined.

See Also:

`load_files/1`, `save_modules/2`, `save_program/1`, `volatile/1`

18.3.159 save_program/[1,2]

Synopsis

save_program(+*File*)

save_program(+*File*, +*Goal*)

Saves the state of the current execution in QOF format to *File*. A goal, *Goal*, to be called upon execution/restoring of the saved state, may be specified.

Arguments

File file_spec
: An atom representing a filename.

Goal callable [MOD]
: A goal.

Description

save_program/[1,2] creates a QOF representation of all predicates in all modules existing in the system. However, it does not save the user's pre-linked code. It also saves such states of the system as operator definitions, prolog_flags, debugging and advice state, and initializations. Object files dynamically loaded into the system are saved in the qof file as object dependencies.

The resulting file is executable, and can be started up as a command, or can be restored using restore/1.

save_program/[1,2] saves module import/export information, which gets reinstated when *File* is loaded. No new module-importation will be done when *File* is loaded, because it is assumed that it was done before save_program/[1,2] was called. Thus if your program consists of one or more modules, and you save it with save_program/[1,2], loading the resulting *File* into some new module will not import any of your predicates into that module. If you want to save out a module such that it will be imported automatically into any module from which it is loaded, then use save_modules/2.

Exceptions

instantiation_error
: *File* or *Goal* is not bound.

`type_error`
: *File* is not a valid file specification, or *Goal* is not a valid goal.

`permission_error`
: *File* is not writable.

See Also

`load_files/[1,2]`, `restore/1`, `save_modules/2`, `save_predicates/2`, `volatile/1`

Section 8.5 [ref-sls], page 192

18.3.160 `see/1`

Synopsis

`see(+`*`FileOrStream`*`)`

Makes file *FileOrStream* the current input stream.

Arguments

FileOrStream file_spec or *stream_object*
: File specification or stream object.

Description

If there is an open input stream associated with *FileOrStream*, and that stream was opened by `see/1`, then it is made the current input stream;

Otherwise, the specified file is opened for input and made the current input stream. If it is not possible to open the file, `see/1` raises an exception.

Different file names (that is, names that do not unify) represent different streams (even if they correspond to the same file). Therefore, assuming '`food`' and '`./food`' represent the same file, the following sequence will open two streams, both connected to the same file.

```
see(food)
...
see('./food')
```

It is important to remember to close streams when you have finished with them. Use `seen/0` or `close/1`.

Exceptions

`instantiation_error`
: *FileOrStream* is not instantiated enough.

`existence_error`
: *FileOrStream* not currently open for input, and `fileerrors` flag is `on`.

`domain_error`
: *FileOrStream* is neither a filename nor a stream.

See Also

`seen/0`, `close/1`, `abort/0`, `seeing/1`

Section 8.7.7.4 [ref-iou-sfh-opn], page 227

18.3.161 `seeing/1`

Synopsis

`seeing(-FileOrStream)`

Unifies *FileOrStream* with the current input stream or file.

Arguments

FileOrStream *file_spec* or *stream_object*

Description

Exactly the same as `current_input(FileOrStream)`, except that *FileOrStream* will be unified with a filename if the current input stream was opened by `see/1` (Section 8.7.7.4 [ref-iou-sfh-opn], page 227).

Can be used to verify that `FileNameOrStream` is still the current input stream as follows:

```
/* nonvar(FileNameOrStream), */
see(FileNameOrStream),
...
seeing(FileNameOrStream)
```

If the current input stream has not been changed (or if changed, then restored), the above sequence will succeed for all file names and all stream objects opened by `open/[3,4]`. However, it will fail for all stream objects opened by `see/1` (since only filename access to streams opened by `see/1` is supported). This includes the stream object `user_input` (since the standard input stream is assumed to be opened by `see/1`, and so `seeing/1` would return `user` in this case).

If *FileOrStream* is instantiated to a value that is not the identifier of the current input stream, `seeing`(*FileOrStream*) simply fails.

Can be followed by `see/1` to ensure that a section of code leaves the current input unchanged:

```
/* var(OldFileNameOrStream), */
seeing(OldFileNameOrStream),
...
see(OldFileNameOrStream)
```

The above is analogous to its stream-object-based counterpart,

```
/* var(OldStream), */
current_input(OldStream),
...
set_input(OldStream)
```

Both of these sequences will always succeed regardless of whether the current input stream was opened by `see/1` or `open/3` (Section 8.7.7.4 [ref-iou-sfh-opn], page 227).

See Also

`see/1`, `open/[3,4]`, `current_input/1`

18.3.162 `seek/4`

Synopsis

`seek(+Stream, +Offset, +Method, -NewLocation)`

Seeks to an arbitrary byte position in *Stream*.

Arguments

Stream stream_object
: a valid Prolog stream

Offset integer
: the offset in bytes to seek relative to *Method* specified.

Method one of `[bof,current,eof]`
: specifies where to start seeking. It is one of the following.

`bof`	Seek from beginning of the file stream.
`current`	Seek from current position of the file stream.
`eof`	Seek from end of the file stream.

NewLocation integer
: The byte offset from beginning of the file after seeking operation.

Description

Sets the current position of the file stream *Stream* to a new position according to *Offset* and *Method*. If *Method* is

`bof`	the new position is set to *Offset* bytes from beginning of the file stream.
`current`	the new position is *Offset* bytes plus the current position of *Stream*.
`eof`	the new position is *Offset* bytes, a negative integer, plus the size of the file.

If *Offset* is 0, `seek/4` returns the current position from the beginning of *Stream* and sets the position to the same location.

If *Stream* is an output stream permitting flushing output, the characters in the buffer of the stream are flushed before seek is performed. If the output stream *Stream* does not permit flushing output and there are characters remaining in the buffer, then a permission error is raised.

If *Stream* is an input stream, the characters in the input buffer of the stream are discarded before seek is performed. The input buffer is empty when the `seek/4` call returns.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`domain_error`
: *Method* is not one of `bof`, `current` or `eof`.

Offset is a negative value and *Method* is `bof`.

Offset is a positive value and *Method* is `eof`.

`instantiation_error`
: *Offset* or *Method* is not instantiated.

`type_error`
: *Stream* is not a stream object.

Offset is not an integer type.

Method is not an atom type.

`permission_error`
: *Stream* names an open stream but the stream is not open with `seek(byte)` permission.

An error occurred while seeking in the file stream.

Flushing attempted but not permitted.

See Also

`stream_position/[2,3]`, `open/4`, `character_count/2`, `line_count/2`, `line_position/2`.

Section 8.7 [ref-iou], page 214

18.3.163 `seen/0`

Synopsis

`seen`

Closes the current input stream.

Description

Current input stream is set to be `user_input`; that is, the user's terminal.

Always succeeds

See Also

`close/1`

18.3.164 `set_input/1`

Synopsis

`set_input(+`*`Stream`*`)`

makes *Stream* the current input stream.

Arguments

Stream stream_object
: a valid input stream

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`instantiation_error`
`type_error`

See Also

`read/1`, `get/1`

18.3.165 `set_output/1`

Synopsis

`set_output(+`*`Stream`*`)`

makes *Stream* the current output stream.

Arguments

Stream stream_object
: a valid output stream

Description

Subsequent output predicates such as `write/1` and `put/1` will use this stream.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

```
instantiation_error
type_error
```

See Also:

`write/1`, `put/[1,2]`

18.3.166 `setof/3`

Synopsis

`setof(+`*`Template`*`, +*`*`Generator`*`, *`*`Set`*`)`

Returns the set *Set* of all instances of *Template* such that *Generator* is provable.

Arguments

Template term
Generator callable [MOD]
: a goal to be proved as if by `call/1`.

Set list of term
: non-empty set

Description

Set is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 8.9.7 [ref-lte-cte], page 242). If there are no instances of *Template* such that *Generator* is satisfied, then `setof/3` simply fails.

Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case *Set* will only provide an imperfect representation of what is in reality an infinite set.

If *Generator* is instantiated, but contains uninstantiated variables that do not also appear in *Template*, then `setof/3` can succeed nondeterminately, generating alternative values for *Set* corresponding to different instantiations of the free variables of *Generator*. (It is to allow for such usage that *Set* is constrained to be non-empty.)

If Generator is of the form *A*`^`*B* then all the variables in *A* are treated as being existentially quantified.

Examples

See `findall/3` for examples that illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`.

Exceptions

As for `call/1`, and additionally:

`resource_error`
: *Template* contains too many free variables.

See Also

`bagof/3`, `^/2`

Section 8.15 [ref-all], page 295

18.3.167 `show_profile_results/[0,1,2]` *development*

Synopsis

`show_profile_results`

`show_profile_results(+`*By*`)`

`show_profile_results(+`*By*`,+`*Num*`)`

Displays the results of the last profiled execution.

Arguments

By one of `[by_time,by_choice_points,by_calls,by_redos]`
Num integer

Description

Displays profiling information accumulated from the last call to `profile/1`. The *By* argument specifies the *display mode*, which determines how the list is sorted and what the percentage figure included in the output refers to. The *Num* argument determines the maximum number of predicates displayed. This list is always sorted in descending order so that the top *Num* predicates are displayed for a give display mode.

The output lists the predicate name, number of calls, choice points and redos for the predicate, then the time in milliseconds and a percentage figure that depends on the display mode. For example, if the display mode is `by_calls` then this is the percentage of the total calls during profiling made to this predicate.

Then the callers are listed, showing for each caller the predicate name, clause number and call number within that clause of the call, followed by the number of calls made from here and the percentage of time spent in the predicate attributed to this caller.

`show_profile_results/1` displays up to a maximum of 10 predicates.

`show_profile_results/0` displays up to a maximum of 10 predicates using the `by_time` display mode.

This predicate is not supported in runtime systems.

Example

```
| ?- show_profile_results(by_time, 3).
Proc              Calls ChPts Redos Time  %
                                 Caller(proc,cl#,cll#,%)
user:setof/3      227   0     0     2.04  34.0
                                 user:satisfy/1,6,1 152 61.0
                                 user:seto/3,1,1 48 20.0
                                 user:satisfy/1,7,1 27 17.0
user:satisfy/1    35738 36782 14112 0.32  5.3
                                 user:satisfy/1,1,2 13857 43.0
                                 user:satisfy/1,2,1 12137 31.0
                                 user:satisfy/1,1,1 7315 18.0
                                 user:satisfy/1,3,1 1155 6.0
user:inv_map_l/5 4732  4732  3115  0.20  3.3
                                 user:inv_map_l/5,2,1 3115 60.0
                                 user:inv_map/4,5,1 1617 40.0
```

See Also

`profile/[0,1,2,3]`, `get_profile_results/4`, `noprofile/0`

18.3.168 simple/1 *meta-logical*

Synopsis

simple(+*Term*)

Term is currently instantiated to either an atom, a number, a database or a variable.

Arguments

Term term

Examples

```
| ?- simple(9).

yes
| ?- simple(_X).

_X = _2487
| ?- simple("a").

no
```

See Also

atom/1, number/1, var/1, compound/1, callable/1, nonvar/1

18.3.169 `skip/[1,2]`

Synopsis

`skip(+Char)`

`skip(+Stream, +Char)`

Skips over characters from the current input stream, or *Stream*, through the first character whith an ASCII code that match the lower 8-bits of the value of the integer expression *Char*.

Arguments

Stream stream_object
Char expr an integer expression.

Description

Char may be an integer expression. The most useful form of integer expression in this context is the zero-quote notation, for example, `0'a`, which evaluates to 97, the ASCII code for the letter '`a`', so that

```
| ?- skip(0'a).
```

skips over (ignores) all input until the next occurrence of the letter '`a`'.

If *Char* does appear, `skip/1` will consume *Char*, so that `get0/1` will read the following character.

To skip to the end of the current input stream:

```
| ?- repeat, get0(-1), !.
```

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`instantiation_error`
`type_error`
`domain_error`
Char evaluates to a float, or an integer outside the range 0..25.

`existence_error`
: If *Char* does not appear on the current input stream, an error message is given for reading beyond the end of the stream, and the computation is aborted.

The portion of the input following *Char* is not a valid Prolog term.

See Also

`tab/1`

18.3.170 `skip_line/[0,1]`

Synopsis

`skip_line`

`skip_line(+`*`Stream`*`)`

Skip the remaining input characters on the current line on the current input stream, or on *Stream*.

Arguments

Stream stream_object
: a valid Prolog input stream

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`
: Trying to read beyond end of *Stream*.

Comments

Coding with `skip_line/[0,1]` and `at_end_of_line/[0,1]` to handle line input is more portable among different operating systems than checking end of line by the input character code.

See Also

`get0/[1,2]`, `at_end_of_line/[0,1]`, `at_end_of_file/[0,1]`.

18.3.171 sort/2

Synopsis

`sort(+List1, -List2)`

Sorts the elements of the list *List1* into the ascending standard order, and removes any multiple occurrences of an element. The resulting sorted list is unified with the list *List2*.

Arguments

List1 list of term
List2 list of term

Examples

```
| ?- sort([a,X,1,a(x),a,a(X)], L).
L = [X,1,a,a(X),a(x)]
```

(The time taken to do this is at worst order ($N \log N$) where N is the length of the list.)

Exceptions

`instatiation error`
: *List1* is not properly instantiated

`type_error`
: *List1* is not a proper list

See Also

`keysort/2` Section 8.9.7.3 [ref-lte-cte-sor], page 243

18.3.172 source_file/[1,2,3]

Synopsis

```
source_file(+AbsFile)

source_file(*AbsFile)

source_file(*Pred, *AbsFile)

source_file(*Pred, *ClauseNumber, *AbsFile)
```

AbsFile is the absolute name of a loaded file, and *ClauseNumber* is the number of a clause for *Pred* in that file.

Arguments

Pred callable [MOD]
: selected predicate specification.

ClauseNumber integer
: integer representing clause number

AbsFile atom
: absolute filename

Description

Loaded files include compiled, QOF loaded and pre-linked files.

If *AbsFile* is not the name of a loaded file, then `source_file(AbsFile)` simply fails. If *AbsFile* is bound to an illegal filename, `source_file/1` fails.

If *Pred* is not a loaded predicate, then `source_file/2` simply fails. If bound to an illegal predicate specification, it fails. *Pred* is assumed to refer to the source module. Thus, to find *any* predicates defined in a given file, use the form:

```
source_file(M:P, File)
```

`source_file/3` is true if clause number *ClauseNumber* of predicate *Pred* comes from file *AbsFile*. `source_file/3` is useful for handling multifile predicates, but it works for predicates defined completely in one file, as well.

Any combination of bound and unbound arguments is possible, and `source_file/3` will generate the others.

Backtracking

If *AbsFile* is unbound, it is successively unified with the absolute names of all currently loaded files. Files loaded through the foreign function interface are not reported by `source_file/1`.

If *Pred* is instantiated to the skeletal specification of a loaded predicate, then *AbsFile* will be successively unified with the absolute names of the files in which the *Pred* was defined.

If *AbsFile* is instantiated to the absolute name of a loaded file, then *Pred* will be successively unified with the skeletal specifications of all predicates defined in *AbsFile*.

See Also

`absolute_file_name/[2,3]`, `multifile/1`

18.3.173 spy/1 *development*

Synopsis

`spy` *+PredSpecs*

Sets spypoints on all the predicates represented by *PredSpecs*

Arguments

PredSpecs gen_pred_spec_tree
: Single predicate specification of form: *Name*, or *Name/Arity*, or a list of such. [MOD]

Description

Turns debugger on in debug mode, so that it will stop as soon as it reaches a spypoint. Turning off the debugger does not remove spypoints. Use `nospy/1` or `nospyall/0`) to explicitly remove them.

Note that since `spy` is a built-in operator, the parentheses, which usually surround the arguments to a predicate, are not necessary (although they can be used if desired).

If you use the predicate specification form *Name* but there are no clauses for *Name* (of any arity), then a warning message will be displayed and no spypoint will be set.

```
| ?- spy test.
% The debugger will first leap -- showing spypoints (debug)
* There are no predicates with the name test in module user

yes
[debug]
```

To place a spypoint on a currently undefined procedure, use the full form *Name/Arity*; you will still get a warning message, but the spypoint will be set .

```
| ?- spy test/1.
* You have no clauses for user:test/1
% Spypoint placed on user:test/1

yes
[debug]
| ?-
```

If `spy/1` is given any invalid argument it prints a warning.

This predicate is not supported in runtime systems.

Exceptions

`instantiation_error`
: if the argument is not ground.

`type_error`
: if a *Name* is not an atom or an *Arity* not an integer.

`domain_error`
: if a *PredSpec* is not a valid procedure specification, or if an *Arity* is specified as an integer outside the range 0-255.

`permission_error`
: if a specified procedure is built-in or imported into the source module.

See Also

`nospy/1`, `nospyall/0`, `debug/0`, `current_spypoint/1`, `add_spypoint/1`, `remove_spypoint/1`

18.3.174 `statistics/[0,2]`

Synopsis

`statistics`

Displays statistics relating to memory usage and execution time.

`statistics(+`*`Keyword`*`, -`*`List`*`)`

`statistics(*`*`Keyword`*`, *`*`List`*`)`

Obtains individual statistics.

Arguments

Keyword atom
: keyword such as `runtime`

List list of integer
: list of statistics (see following table)

Times are given in milliseconds and sizes are given in bytes.

`runtime` [cpu time used by Prolog, cpu time since last call to `statistics/[0,2]`]

`system_time`
: [cpu time used by the operating system, cpu time used by the system since the last call to `statistics/[0,2]`]

`real_time`
: [wall clock time since 00:00 GMT 1/1/1970, wall clock time since the last call to `statistics/[0,2]`]

`memory` [total virtual memory in use, total virtual memory free]

`stacks` [total global stack memory, total local stack memory]

`program` [program space, 0]

`global_stack`
: [global stack in use, global stack free]

`local_stack`
: [local stack in use, local stack free]

`trail` [size of trail, 0]

`garbage_collection`
: [number of GCs, freed bytes, time spent]

`stack_shifts`
: [number of global stack area shifts, number of local stack area shifts, time spent shifting]

`atoms`
: [number of atoms, atom space in use, atom space free]

`atom_garbage_collection`
: [number of AGCs, freed bytes, time spent]

`core`
: (same as memory)

`heap`
: (same as program)

Description

`statistics/0` displays various statistics relating to memory usage, runtime and garbage collection, including information about which areas of memory have overflowed and how much time has been spent expanding them.

Garbage collection statistics are initialized to zero when a Prolog session starts (this includes sessions started from saved-states created by `save_program/[1,2]`, and includes re-starts caused when `restore/1` is used). The statistics increase until the session is over.

`statistics/2` is usually used with *Keyword* instantiated to a keyword such as `runtime` and *List* unbound. The predicate then binds *List* to a list of statistics related to the keyword. It can be used in programs that depend on current runtime statistical information for their control strategy, and in programs that choose to format and write out their own statistical summaries.

If keyword is `garbage_collection` the list returned contains three elements:

- the number of garbage collections performed since the beginning of the Prolog session.
- the number of bytes of heap space freed by those garbage collections.
- the number of milliseconds spent performing those garbage collections.

Examples

The output from `statistics/0` looks like this:

```
memory (total)     377000 bytes: 350636 in use,  26364 free
   program space   219572 bytes
      atom space    (2804 atoms)  61024 in use,  43104 free
   global space     65532 bytes:   9088 in use,  56444 free
      global stack   6984 bytes
      trail                        16 bytes
      system                     2088 bytes
   local stack      65532 bytes:    356 in use,  65176 free
      local stack                  332 bytes
      system                        24 bytes

 0.000 sec. for 0 global and 0 local space shifts
 0.000 sec. for 0 garbage collections
                  which collected 0 bytes
 0.000 sec. for 0 atom garbage collections
                  which collected 0 bytes
 0.233 sec. runtime
```

To report information on the runtime of a predicate `p/0`, add the following to your program:

```
:- statistics(runtime, [T0|_]),
   p,
   statistics(runtime, [T1|_]),
   T is T1 - T0,
   format('p/0 took ~3d sec.~n', [T]).
```

See Also

Section 8.12.1.2 [ref-mgc-ove-sta], page 257

18.3.175 `stream_code/2`

Synopsis

`stream_code(-`*`Stream`*`, +`*`CStream`*`)`

`stream_code(+`*`Stream`*`, -`*`CStream`*`)`

Converts between Prolog representation, *Stream*, and C representation, *CStream*, of a stream.

Arguments

Stream stream_object
: a variable or a valid Prolog stream

CStream integer
: a variable or a valid C stream

Description

`stream_code/2` is used when there are input/output related operations performed on the same stream in both Prolog code and foreign code.

Stream argument is a valid type if it is *user*, *user_input*, *user_output*, *user_error*, a variable, or a value obtained through `open/[3,4]` or previous `stream_code/2` call. Such a valid *Stream* value can be used as the stream argument to any of the Prolog built-in I/O predicates taking a stream argument.

CStream argument is a valid type if it is a variable, a value obtained through previous `stream_code/2` call, a value obtained through `QP_fopen()`, `QP_fdopen()`, or a value of pointer to `QP_stream` structure obtained through foreign function call. Such a valid *CStream* value can be used as the stream argument to any of the QP foreign function taking stream as an argument.

Exceptions

`instantiation_error`
: *Stream* argument or *CStream* argument is not ground.

`type_error`
: *Stream* or *CStream* is not a stream type or *CStream* is not an integer type.

`existence_error`
: *Stream* is syntactically valid but does not name an open stream or *CStream* is of integer type but does not name a pointer to a stream.

Comments

The most frequent use of `stream_code/2` is to get a stream value to be used in Prolog code for a stream created in foreign code. This is necessary when a desired stream can not be obtained through `open/[3,4]`; e.g. a stream referring to a socket or an encrypted file.

See Also:

`open/[3,4]`, `QP_fopen()`, `QP_fdopen()`

18.3.176 stream_position/[2,3]

Synopsis

`stream_position(+`*`Stream`*`, -`*`Current`*`)`

True when *Current* represents the current position of *Stream*.

`stream_position(+`*`Stream`*`, -`*`Current`*`, +`*`New`*`)`

Unifies the current position of the read/write pointer for *Stream* with *Current*, then sets the position to *New*.

Arguments

Stream stream_object
: an open stream

Current term
: stream position object representing the current position of *Stream*.

New term stream position object

Caution

A stream position object is represented by a special Prolog term. The only safe way of obtaining such an object is via `stream_position/3` or `stream_position/2`. You should not try to construct, change, or rely on the form of this object. It may vary under different operating systems and/or change in subsequent versions of Quintus Prolog. On some systems, a stream position object currently has the form:

```
'$stream_position'(CharCount, LineCount, LinePos, Magic1, Magic2)
```

Description

Character count, line count and line position determine the position of the pointer in the stream. Such information is found by using `character_count/2`, `line_count/2` and `line_position/2`.

`stream_position/2` may be used on any stream at all: streams that are connected to disk files, streams that are connected to sockets, streams that are connected to the terminal (including the standard streams `user_input`, `user_output`, `user_error`), and even streams defined using `QP_make_stream()`.

`stream_position/3` may only be used on streams that are connected to disk files.

Standard term comparison of two stream position objects for the same stream will work as one expects. When *SP1* and *SP2* refer to positions in the same stream, *`SP1 @< SP2`* if and only if *SP1* is before *SP2* in the stream.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`type_error`
: *Current*, *New* are not valid stream position objects.

`instantiation_error`

Example

To find the current position of a stream without changing it, one can ask

```
| ?- stream_position(Stream, Current).
```

See Also

`seek/4`, `open/[3,4]`, `character_count/2`, `line_count/2`, `line_position/2`. Section 8.7 [ref-iou], page 214

18.3.177 `style_check/1`

Synopsis

`style_check(+`*Type*`)`

Turns on the specified *Type* of compile-time style checking.

Arguments

Type atom
: is one of the following atoms:

  `all` Turn on all style checking.

  `single_var`
  : Turn on checking for clauses containing a single instance of a named variable, where variables that start with a '`_`' are not considered named.

  `discontiguous`
  : Turn on checking for procedures whose clauses are not all adjacent to one another in the file.

  `multiple` Turn on checking for multiple definitions of the same procedure in different files.

Description

Since all style checking is on by default, this predicate is only used to put back style checking after it has been turned off by `no_style_check/1`.

Exceptions

`instantiation_error` *Type* is not bound.

`type_error` *Type* is not an atom.

`domain_error` *Type* is not a valid type of style checking.

See Also

`no_style_check/1`.

18.3.178 subsumes_chk/2 *meta-logical*

Synopsis

`subsumes_chk(+General, +Specific)`

True when *General* subsumes *Specific*; that is, when *Specific* is an instance of *General*.

Arguments

General term
Specific term

Description

In previous releases, `subsumes_chk/2` was available as a library predicate. In release 2.5, it was made part of the system because it was found to be useful in applications such as writing theorem provers. The built-in predicate behaves identically to the original version of `subsumes_chk/2` but is much more efficient.

Term subsumption is a sort of one-way unification. Term S and T unify if they have a common instance, and unification in Prolog instantiates both terms to that common instance. S subsumes T if T is already an instance of S. For our purposes, T is an *instance* of S if there is a substitution that leaves T unchanged and makes S identical to T.

Comments

There are two other related predicates defined in `library(subsumes)`, `subsumes/2` and `variant/2`. These predicates are defined in terms of `subsumes_chk/2`, and they are still available in that library package.

See Also

`library(subsumes)`, `library(occurs)`

18.3.179 `tab/[1,2]`

Synopsis

`tab(+`*Integer*`)`

`tab(+`*Stream*`, +`*Integer*`)`

Writes *Integer* spaces to the current output stream, or *Stream*.

Arguments

Stream stream_object
Integer expr
: an integer expression.

Description

If *Integer* evaluates to a negative integer, `tab/1` simply succeeds without doing anything.

If the current output device is the user's terminal, the spaces are not necessarily printed immediately; see `ttyflush/0`.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`instantiation_error`
`type_error`
: *N* is not an integer.

`permission_error`
: There is an error in the bottom layer of write function of the stream.

See Also

`ttyflush/0`

18.3.180 `tell/1`

Synopsis

`tell(+`*`FileOrStream`*`)`

makes *FileOrStream* the current output stream.

Arguments

FileOrStream file_spec or *stream_object*
: file specification or stream object.

Description

If there is an open output stream currently associated with the filename, and that stream was opened by `tell/1`, then it is made the current output stream;

Otherwise, the specified file is opened for output and made the current output stream. If the file does not exist, it is created. If it is not possible to open the file (because of protections, for example), `tell/1` raises an exception.

Different file names (names that do not unify) represent different streams (even if they correspond to the same file). Therefore, assuming '`food`' and '`./food`' represent the same file, the following sequence will open two streams, both connected to the same file:

```
tell(food)
...
tell('./food')
```

It is important to remember to close streams when you have finished with them. Use `told/0` (Section 8.7.7.8 [ref-iou-sfh-cst], page 229) or `close/1` (Section 8.7.7.8 [ref-iou-sfh-cst], page 229).

Exceptions

`instantiation_error`
: *FileOrStream* is not instantiated enough.

`domain_error`
: *FileOrStream* is neither a filename nor a stream.

`permission_error`
: *FileOrStream* exists but cannot be opened.

`existence_error`
: *FileOrStream* not currently open for input, and `fileerrors` flag is `on`.

See Also

`told/0`, `telling/1`

18.3.181 `telling/1`

Synopsis

`telling(-`*`FileOrStream`*`)`

Unifies *FileOrStream* with the current output stream.

Arguments

FileOrStream file_spec or *stream_object*

Description

Exactly the same as `current_output(`*`FileOrStream`*`)` Section 8.7.7.6 [ref-iou-sfh-cos], page 228), except that *FileOrStream* will be unified with a filename if the current output stream was opened by `tell/1`.

If *FileOrStream* is not a filename or stream object corresponding to an open output stream, `telling`(*FileOrStream*) simply fails.

A common usage of `telling/1` is

```
tell('Some File Name')
...
telling('Some File Name')
```

Should succeed if the current input stream was not changed (or if changed, restored). It succeeds for any filename (including `user`) and any stream object opened by `open/3` (Section 8.7.7.4 [ref-iou-sfh-opn], page 227), but fails for `user_output` and any stream object opened by `tell/1` (Section 8.7.7.4 [ref-iou-sfh-opn], page 227). Passing file names to `tell/1` is the only DEC-10 Prolog usage of `telling/1`, so Quintus Prolog is compatible with this usage.

Examples

WARNING: The following sequence will fail if the current output stream was opened by `tell/1`.

```
telling(File),
...
set_output(File),
```

The only sequences that are guaranteed to succeed are

```
telling(FileOrStream),
...
tell(FileOrStream)
```

and

```
current_output(Stream),
...
set_output(Stream)
```

See Also

`tell/1`, `open/[3,4]`, `current_output/1`

18.3.182 term_expansion/2 *hook*

Synopsis

```
:- multifile term_expansion/2.

term_expansion(+Term1, -Term2)
```

The user may override the standard transformations to be done by `expand_term/2` by defining clauses for `term_expansion/2`.

Arguments

Term1 term
Term2 term

Description

`expand_term/2` calls `term_expansion/2` first; if it succeeds, the standard grammar rule expansion is not tried.

`expand_term/2` always calls `term_expansion/2` in module `user`. Therefore, to be visible to `expand_term/2`, `term_expansion/2` must either be defined in or imported into module `user`. Alternatively, you may define it in any module by using module prefixing; refer to Section 8.13.6 [ref-mod-vis], page 274.

This hook predicate may now return a list of terms rather than a single term. Each of the terms in the list is then treated as a separate clause.

Tip

See Also

`expand_term/2`, `-->/2`, `phrase/[2,3]`, `'C'`, `prolog_load_context/2`. Section 8.16 [ref-gru], page 298

18.3.183 `told/0`

Synopsis

`told`

Closes the current output stream.

Description

Current output stream is set to be `user_output`; that is, the user's terminal.

Always succeeds without error.

18.3.184 `trace/0` *development*

Synopsis

`trace`

Turns the debugger on and starts it creeping; that is, it sets the debugger to trace mode.

Description

The debugger will start showing goals as soon as the first call is reached, and it will stop to allow you to interact as soon as it reaches a leashed port (see `leash/1`, Section 18.3.90 [mpg-ref-leash], page 1157). Setting the debugger to trace mode means that every time you type a query, the debugger will start by creeping.

The effect of this predicate can also be achieved by typing the letter `t` after a `^c` interrupt (see Section 8.11.1 [ref-iex-int], page 250).

This predicate is not supported in runtime systems.

See Also

`debug/0`, `notrace/0`

18.3.185 `trimcore/0`

Synopsis

`trimcore`

Force reclamation of memory in all of Prolog's data areas.

Description

Reduces the free space in all the data areas as much as possible, and gives the space no longer needed back to the operating system.

Exceptions

Automatically called after each directive at the top level.

See Also

Section 8.12.1.1 [ref-mgc-ove-rsp], page 257

18.3.186 `true/0`

Synopsis

`true`

Always succeeds. This could have been trivially defined in Prolog by the single clause:

```
true.
```

18.3.187 ttyflush/0, ttyget/1, ttyget0/1, ttynl/0, ttyput/1, ttyskip/1, ttytab/1

Synopsis

`ttyflush` Equivalent to `flush_output(user)`.

`ttyget(-`*`Char`*`)` Equivalent to `get(user, `*`Char`*`)`.

`ttyget0(-`*`Char`*`)` Equivalent to `get0(user, `*`Char`*`)`.

`ttynl` Equivalent to `nl(user)`.

`ttyput(+`*`Char`*`)` Equivalent to `put(user, `*`Char`*`)`.

`ttyskip(+`*`Char`*`)` Equivalent to `skip(user, `*`Char`*`)`.

`ttytab(+`*`Integer`*`)` Equivalent to `tab(user, `*`Integer`*`)`.

Arguments

Char char

Integer expr

Description

For compatibility with DEC-10 Character I/O a set of predicates are provided, which are similar to the primary ones except that they always use the standard input and output streams, which normally refer to the user's terminal rather than to the current input stream or current output stream. They are easily recognizable as they all begin with "tty".

Given stream-based input/output, these predicates are actually redundant. For example, you could write `get0(user, C)` instead of `ttyget0(C)`.

18.3.188 unix/1

Synopsis

`unix(shell(+Command))` Spawns a command interpreter and executes *Command*. Note that, despite the name, `unix/1` works on both UNIX and Windows.

`unix(system(+Command))` Spawns a shell process and executes *Command*.

`unix(system(+Command, -Status))` Spawns a shell process and executes *Command*. The exit status of the executed command is returned in *Status*.

`unix(cd(+Path))` Changes working directory to *Path*.

`unix(argv(-ArgList))` Returns in *ArgList* the list of commandline arguments as Prolog objects.

`unix(args(-ArgList))` Returns in *ArgList* the list of commandline arguments as a list of atoms.

Arguments

Command term
: atom corresponding to a command (or null)

Status integer
: exit status of the command executed

Path atom
: atom corresponding to a legal directory (or null)

ArgList list of term
: list of arguments used to start up current session.

Description

`unix(cd)` changes the working directory of Prolog (and of Emacs if running under the editor interface) to your home directory. Note that the ⟨ESC⟩ *x* `cd` command under Emacs has the same effect as this, except that Emacs also provides filename completion.

If the return status of *Command* is 0, `unix(system(`*Command*`))` succeeds, otherwise it fails.

`unix(system(`*Command*`, `*Status*`))` returns the status of the executed command, similar to the function `system(3)`. The low-order 8 bits of the *Status* is the value returned by the system call `wait(2V)` and the next 8-bits higher up in the *Status* has the shell exit status

if the shell was not interrupted by a signal. An exit status of 127 indicates that the shell could not be executed.

To start up an interactive shell, type `unix(shell)`.

If *ArgList* is instantiated to a term that does not unify with the result returned, `unix(argv(`*ArgList*`))` or `unix(args(`*ArgList*`))` will simply fail.

Exceptions

`instantiation_error`
: Argument to `unix/1` is not sufficiently instantiated.

`domain_error`
: Argument to `unix/1` is invalid.

`type_error`
: *Path* is not an atom.

`existence_error`
: *Path* is a nonexistent directory.

Examples

To list the QOF files in the current working directory:

```
| ?- unix(shell('ls -l *.qof')).
-rw-rw-r--  1 joe   9152 Oct 20  1990 table.qof
-rw-rw-r--  1 joe    576 Oct 25  1990 test.qof

yes
```

Alternatively, enter a command interpreter, execute commands, and type `exit` to return to prolog:

```
| ?- unix(shell).
% ls -l *.qof
-rw-rw-r--  1 joe   9152 Oct 20  1990 table.qof
-rw-rw-r--  1 joe    576 Oct 25  1990 test.qof
% exit

yes
| ?-
```

If Prolog was invoked using the command (A), the command line arguments can be retrieved as in (B):

```
% prolog screaming yellow + yellow.pl                    (A)
    .
    .
    .

| ?- unix(argv(ArgList)).                                (B)

ArgList = [screaming, yellow]
```

See Also

`QP_initialize()`, `QP_toplevel()`, `system/1` — from `library(strings)`

Section 8.18 [ref-aos], page 307

18.3.189 `unknown/2`

Synopsis

`unknown(-OldAction, +NewAction)`

Unifies *OldAction* with the current action on unknown procedures, and then sets the current action to *NewAction*.

Arguments

OldAction one of `[error,fail]`
NewAction one of `[error,fail]`

Description

This action determines what happens when an undefined predicate is called:

`error` Undefined procedures will raise an exception

`fail` Undefined procedures will simply fail

The default action is `error`. `trace` is accepted as a synonym for `error` for backword compatibility. Note that

```
| ?- unknown(Action, Action).
```

just returns *Action* without changing it.

Procedures that are known to be dynamic just fail when there are no clauses for them. Their behavior is not affected by `unknown/2`. For more information on dynamic procedures, see Section 8.14.2 [ref-mdb-dsp], page 287.

See Also

`unknown_predicate_handler/3`

18.3.190 `unknown_predicate_handler/3` *hook*

Synopsis

```
:- multifile unknown_predicate_handler/3.

unknown_predicate_handler(+Goal, +Module, -NewGoal)
```

User definable hook to trap calls to unknown predicates

Arguments

Goal callable
: Any Prolog term

Module atom
: Any atom that is a current module

NewGoal callable
: Any callable Prolog term

Description

When Prolog comes across a call to an unknown predicate and the `unknown` flag is set to `error`, Prolog makes a call to `unknown_predicate_handler/3` in module `user` with the first two arguments bound. *Goal* is bound to the call to the undefined predicate and *Module* is the module in which that predicate is supposed to be defined. If the call to `unknown_predicate_handler/3` succeeds then Prolog replaces the call to the undefined predicate with the call to *NewGoal*. By default *NewGoal* is called in module `user`. This can be overridden by making *NewGoal* have the form *Module:SomeGoal*.

Examples

The following clause gives the same behaviour as setting `unknown(_,fail)`.

```
unknown_predicate_handler(_, _, fail).
```

The following clause causes calls to undefined predicates whose names begin with '`xyz_`' in module `m` to be trapped to `my_handler/1` in module `n`. Predicates with names not beginning with this character sequence are not affected.

```
unknown_predicate_handler(G, m, n:my_handler(G)) :-
    functor(G,N,_),
    atom_chars(N,Chars),
    append("xyz_" _, Chars).
```

Tips

See Also

unknown/2, prolog_flag/3

18.3.191 use_module/[1,2,3]

Synopsis

`use_module(+Files)`

Loads the module-file(s) *Files*, if not already loaded and up-to-date imports all exported predicates.

`use_module(+File, +Imports)`

Loads module-file *File*, if not already loaded and up-to-date imports according to *Imports*.

`use_module(+Module, -File, +Imports)`

Module is already loaded and up-to-date. Imports according to *Imports*.

`use_module(-Module, +File, +Imports)`

Module has not been loaded, or is out-of-date. Loads *Module* from *File* and imports according to *Imports*.

Arguments

File file_spec or *list of file_spec* [MOD]
: Any legal file specification. Only `use_module/1` accepts a *list* of file specifications. A '`.pl`' or '`.qof`' extension may be omitted in a file specification.

Imports list of simple_pred_spec or *atom*
: Either a list of predicate specifications in the *Name/Arity* form to import into the calling module, or the atom `all`, meaning all predicates exported by the module are to be imported.

Module atom
: The module name in *Files*, or a variable, in which case the module name is returned.

Description

Loads each specified file except the previously loaded files that have not been changed since last loaded. All files must be module-files, and all the public predicates of the modules are imported into the calling module (or module *M* if specified).

`use_module/2` imports only the predicates in *Imports* when loading *Files*.

`use_module/3` allows *Module* to be imported into another module without requiring that its source file (*File*) be known, as long as the *Module* already exists in the system. This predicate is particularly useful when the module in question has been linked with the Development Kernel as described in Section 9.1 [sap-srs], page 337.

Generally, `use_module/3` is similar to `use_module/[1,2]`, except that if *Module* is already in the system, *Module*, or predicates from *Module*, are simply imported into the calling module, and *File* is not loaded again. If *Module* does not already exist in the system, *File* is loaded, and `use_module/3` behaves like `use_module/2`, except that *Module* is unified, after the file has been loaded, with the actual name of the module in *File*. If *Module* is a variable, *File* must exist, and the module name in *File* is returned.

When `use_module/3` is called from an embedded command in a file being compiled with `qpc`, and *File* is unbound, an `initialization/1` fact is generated, so that the actual execution of the `use_module/3` command is delayed until the QOF file is loaded. This means that the module given must exist when the QOF file is loaded, but not when it is created.

As *File* is not checked if *Module* already exists in the system, and *File* can even be left unnamed in that case, for example,

```
:- use_module(mod1, _, all).
```

In other words, the filename may be an unbound variable as long as *Module* is already in the system.

Special case of `load_files/2` and is defined as

```
use_module(Files) :-
    load_files(Files, [if(changed),
                       load_type(latest),
                       must_be_module(true)]).

use_module(File, Imports) :-
    load_files(File, [if(changed),
                      load_type(latest),
                      must_be_module(true),
                      imports(Imports)]).
```

`use_module/1` is similar to `ensure_loaded/1` except that all files must be module-files.

An attempt to import a predicate may fail or require intervention by the user because a predicate with the same name and arity has already been defined in, or imported into, the loading module (or module *M* if specified). Details of what happens in the event of such a *name clash* are given in Section 8.13.13 [ref-mod-ncl], page 279.

After loading the module-file, the source module will attempt to import all the predicates in *Imports*. *Imports* must be a list of predicate specifications in *Name/Arity* form. If any of the predicates in *Imports* are not public predicates, an error message is printed, but the predicates are imported nonetheless. This lack of strictness is for convenience; if you forget

to declare a predicate to be public, you can supply the necessary declaration and reload its module, without having to reload the module that has imported the predicate.

While `use_module/1` may be more convenient at the top level, `use_module/2` is recommended in files because it helps document the interface between modules by making the list of imported predicates explicit.

For consistency, `use_module/2` has also been extended so that the *Imports* may be specified as the term `all`, in which case it behaves the same as `use_module/1`, importing the entire module into the caller.

For further details on loading files, see Section 8.4 [ref-lod], page 189. On file specifications, see Section 8.6 [ref-fdi], page 205.

Exceptions

`instantiation_error`
: *M*, *Files*, or *Imports* is not ground.

`type_error`
: One of the arguments is the wrong type.

`existence_error`
: A specified file does not exist. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

See Also

`compile/1`, `ensure_loaded/1`, `initialization/1`, `load_files/[1,2]`, `volatile/1`,

Section 8.4 [ref-lod], page 189, Section 8.6 [ref-fdi], page 205

18.3.192 `user_help/0` *hook*

Synopsis

```
:- multifile user_help/0.

user_help
```

A hook for users to add more information when `help/0` is called.

Description

Useful when you want a standard way to tell users something, like how to run a demo.

`help/0` always calls `user_help/0` in module `user`. Therefore, to be visible to `help/0`, `user_help/0` must either be defined in or imported into module `user`.

Example

The common test harness for many Quintus test suites includes the clause:

```
user_help :-
  suite_type(_,Type), suite_host(Host),
  write('You have loaded the Quintus test suite for '),
  write(Type), write(' on '), write(Host), nl, nl,
  write('You can invoke the suite as follows:'), nl, nl,
  write(' ?- quiet.               % run suite, concise output'), nl,
  write(' ?- verbose.             % run suite, verbose output'), nl,
  write(' ?- quiet(+Pred).        % run suite, concise output for'), nl,
  write('                         % tests of predicate Pred'), nl,
  write(' ?- quiet(+Pred, +N).    % similar to above, but runs the'),nl,
  write('                         %  test specified by N'), nl,
  write(' ?- verbose(+Pred).      % run suite, verbose output for'), nl,
  write('                         %  tests of predicate Pred'), nl,
  write('                         %  (Pred a name, NOT name/arity!'),nl,
  write(' ?- verbose(+Pred, +N).  % similar to above, but runs the'),nl,
  write('                         %  test specified by N'), nl,
  write(' ?- help.                % to get this message'), nl.
```

So if you compile the Prolog, C, Pascal or FORTRAN suites, you have a consistent help message telling you how to run the suites:

```
<run prolog>

<compile /ptg/suite/plsuite.pl>

| ?- help.
```

If you have loaded the Quintus test suite for Prolog on Sun ??? you can invoke the suite as follows:

```
?- quiet.               % run suite, concise output
?- verbose.             % run suite, verbose output
?- quiet(+Pred).        % run suite, concise output for
                        % tests of predicate Pred
?- quiet(+Pred, +N).    % similar to above, but runs the
                        %  test specified by N
?- verbose(+Pred).      % run suite, verbose output for
                        %  tests of predicate Pred
                        %  (Pred a name, NOT name/arity!
?- verbose(+Pred, +N).  % similar to above, but runs the
                        %  test specified by N
?- help.                % to get this message
```

See Also

`help/1` Section 8.17 [ref-olh], page 304

18.3.193 var/1 *meta-logical*

Synopsis

`var(+Term)`

Term is currently uninstantiated ('`var`' is short for variable).

Arguments

Term term

Description

An uninstantiated variable is one that has not been bound to anything, except possibly another uninstantiated variable. Note that a compound term with some arguments that are uninstantiated is not itself considered to be uninstantiated.

Examples

```
| ?- var(foo(X,Y)).

no
| ?- var([X,Y]).

no
| ?- var(X).

X = _3437 ;

no
| ?- Term = foo(X,Y), var(Term).

no
```

See Also

`atom/1`, `atomic/1`, `number/1`, `compound/1`, `callable/1`, `nonvar/1`, `simple/1`

18.3.194 `version/[0,1]`

Synopsis

`version`

`version(+`*`Atom`*`)`

Display system identification messages

Add the atom *A* to the list of introductory messages.

Arguments

Atom atom

18.3.195 `vms/[1,2]`

Synopsis

`vms(+-`*`Command`*`)`

`vms(+-`*`Command`*`, +`*`Flag`*`)`

Issue a VMS specific system command (only available on VMS platforms).

Arguments

Command term
Flag one of `[n,i]`

18.3.196 `volatile/1` *declaration*

Synopsis

`:- volatile` *+PredSpecs*

Declares *PredSpecs* to be volatile. Volatile predicates are not saved in QOF files by Prolog '`save`' predicates.

Arguments

PredSpecs pred_spec_forest [MOD]
: A single predicate specification of the form *Name/Arity*, or a sequence of predicate specifications separated by commas. *Name* must be an atom and *Arity* an integer in the range 0..255.

Description

A built-in prefix operator, so that declarations can be written as e.g.

```
:- volatile a/1, b/3.
```

callable both at compile-time and run-time. In both cases the predicate specified will, with immediate effect, be declared as volatile.

When used as a compile-time directive, the volatile declaration of a predicate must appear before all clauses of that predicate. The predicate is reinitialized.

When used as a callable goal, the only effect on the predicate is that it is set to be volatile.

Exceptions

`instantiation_error`
: If *PredSpec* is not ground.

`type_error`
: If *PredSpec* is not a proper predicate specification.

`permission_error`
: *PredSpec* names a non-volatile predicate that is already defined (This exception is only raised when volatile is used as a compile-time directive.)

Comments

Whether *PredSpec* is volatile can be checked with `predicate_property/2`.

The properties, as well as the predicate, can be deleted with `abolish/1`.

PredSpec clauses are saved by `qpc`.

Examples

see examples under `initialization/1`.

See Also

`initialization/1`, `save_program/[1,2]`, `save_modules/2`, `save_predicates/2`

Section 8.5.6.2 [ref-sls-igs-vol], page 203

18.3.197 `write/[1,2]`

Synopsis

`write(+`*Term*`)`

`write(+`*Stream*`, +`*Term*`)`

Writes *Term* to the current output stream or *Stream*.

Arguments

Stream stream_object
: a valid output stream

Term term
: the term to be written

Description

Equivalent to `write_term/[2,3]` with these options:

```
[quoted(false),ignore_ops(false),numbervars(true)]
```

If *Term* is uninstantiated, it is written as an anonymous variable (an underscore followed by a non-negative integer).

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226), plus:

`existence_error`

Example

```
| ?- write('a b').
a b
```

See Also

`read[1,2]`, `writeq/[1,2]`, `write_canonical/[1,2]` or `write_term/[2,3]`.

18.3.198 `write_canonical/[1,2]`

Synopsis

`write_canonical(+`*Term*`)`

`write_canonical(+`*Stream*`, +`*Term*`)`

Writes *Term* to the current or specified output stream in standard syntax.

Arguments

Stream stream_object
: a valid Prolog stream, which is open for output

Term term
: the term to be written

Description

Equivalent to `write_term/[2,3]` with the options:

```
[quoted(true),ignore_ops(true),numbervars(false),char_escapes(false)]
```

This predicate is provided so that *Term*, if written to a file, can be read back by `read/[1,2]` regardless of special characters in *Term* or prevailing operator declarations.

Does not terminate its output with a full-stop, which is required by `read/[1,2]`.

In general, one can only read (using `read/[1,2]`) a term written by `write_canonical/1` if the value of the `character_escapes` flag is the same when the term is read as when it was written.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226).

Examples

The following sequence will succeed:

```
...
open(FileName, write, StreamOut),
write_canonical(StreamOut, Term),
write(StreamOut, '.'),
nl(StreamOut),
close(StreamOut),
...
open(FileName, read, StreamIn),
read(StreamIn, Term),
close(StreamIn),
...
```

To contrast `write/[1,2]` and `write_canonical/[1,2]`:

```
| ?- write({'A' + '$VAR'(0) + [a]}).

{A+A+[a]}

| ?- write_canonical({'A' + '$VAR'(0) + [a]}).

{}(+(+('A','$VAR'(0)),.(a,[])))
```

See Also

`write_term/[2,3]`, `write/[1,2]`, `writeq/[1,2]`, `read/[1,2]`

Section 8.7.4.2 [ref-iou-tou-cha], page 218

18.3.199 write_term/[2,3]

Synopsis

`write_term(+`*Term*`, +`*Options*`)`

`write_term(+`*Stream*`, +`*Term*`, +`*Options*`)`

Writes *Term* to the current output stream or to *Stream* in a format given by the options.

Arguments

Stream stream_object
: a valid Prolog stream, which is open for output

Term term
: the term to be written

Options list of term
: a list of zero or more of the following, where *Bool* must be `true` or `false` (`false` is the default).

  `quoted(`*Bool*`)`
  : Should atoms and functors be quoted as necessary to make them acceptable as input to read?

  `ignore_ops(`*Bool*`)`
  : Ignore current operator declarations? If *Bool* is true, compound terms are always written in the form: *predicate name* `(`*arg1*`,...,`*argn*`)`.

  `portrayed(`*Bool*`)`
  : Call user:`portray/1` for each subterm. By default the behavior of `write_term/[2,3]` is controlled by *Options*, but you can change its effect by providing clauses for the predicate `portray/1`.

  `character_escapes(`*Bool*`)`
  : Use character escapes. *Bool* must be `true` or `false` (the default depends on the value of the `character_escapes` flag as set by `prolog_flag/3`). If *Bool* is `true` then `write_term/[2,3]` tries to write layout characters (except ASCII 9 and ASCII 32) in the form '`\`*lower-case-letter*', if possible; otherwise, `write_term/[2,3]` writes the '`\^`*control-char*' form. If *Bool* is `false` then it writes the actual character, without using an escape sequence.

  `numbervars(`*Bool*`)`
  : Should terms like `'$VAR'(N)` be treated specially? If *Bool* is true, `write_term/[2,3]` writes '`A`' if N=0, '`B`' if N=1, ...'`Z`' if

N=25, 'A1' if N=26, etc. Terms of this form are generated by numbervars/3.

max_depth(*N*)
: Depth limit on printing. *N* is any integer; 0 means no limit and approximately 33 million is the default.

Description

write_term/[2,3] is the most general of the write family of predicates. write_term/[2,3] subsumes all predicates in the family, with the exception of portray_clause/1. That is, all write predicates can be written as calls to write_term/[2,3].

Exceptions

domain_error
: *Options* contains an undefined option.

instantiation_error
: Any of the *Options* arguments or *Stream* is not ground.

type_error
: In *Stream* or in *Options*.

existence_error
: *Stream* is syntactically valid but does not name an open stream.

permission_error
: *Stream* names an open stream but the stream is not open for output.

Comments

If an option is specified more than once the rightmost option takes precedence. This provides for a convenient way of adding default values by putting these defaults at the front of the list of options. For example, the predicate my_write_term/2 defined as

```
my_write_term(Term, Options) :-
   write_term(Term, [quoted(true),
                     numbervars(true)|Options]).
```

is equivalent to write_term/2 except that two of the defaults are different.

Examples

How certain options affect the output of write_term/2:

```
| ?- write_term('a b', [quoted(true)]).
'a b'

| ?- write_term(a+b, [ignore_ops(true)]).
+(a,b)

| ?- write_term(f('$VAR'(2)),
                 [numbervars(true)].)
f(C)

| ?- write_term(f('$VAR'('C')),
                 [numbervars(true)]).

f(C)
```

If your intention is to name variables such as that generated by `read_term/2` with the variable_names option then this can be done by defining a predicate like:

```
var_to_names([]) :- !.
var_to_names([=(Name,Var)|RestofPairs]) :-
     ( var(Var) ->
          Var = '$VAR'(Name)
     ; true
     ),
     var_to_names(RestofPairs).

| ?- read_term([variable_names(Names)], X),
     var_to_names(Names),
     write_term(X, [numbervars(true)]),
     nl,
     fail.
|: a(X, Y).
a(X, Y).

no
```

See Also

`write/[1,2]`, `writeq/[1,2]`, `write_canonical/[1,2]`, `display/1`, `print/1`, `portray_clause/1`

Section 8.7.4.2 [ref-iou-tou-cha], page 218

18.3.200 `writeq/[1,2]`

Synopsis

`writeq(+`*Term*`)`

`writeq(+`*Stream*`, +`*Term*`)`

Writes the term *Term* to *Stream* or the current output stream.

Arguments

Stream stream_object
: a valid Prolog stream, which is open for output

Term term
: the term to be written

Description

Equivalent to `write_term/[2,3]` with the options:

```
[quoted(true),ignore_ops(false),numbervars(true)]
```

Does not terminate its output with a full-stop. Therefore, if you want this term to be input to `read/[1,2]`, you must explicitly write a full-stop to mark the end of the term.

Comments

Depending upon whether character escaping is on or off, `writeq/[1,2]` and `write_canonical/[1,2]` behave differently when writing quoted atoms. If character escaping is on:

- The characters with ASCII codes 9 (horizontal tab), 32 (space), and 33 through 126 (non-layout characters) are written as themselves.
- The characters with ASCII codes 8, 10, 11, 12, 13, 27, and 127 are written in their '`\`*`lowercase letter`*' form (see above for the corresponding letter).
- The character with ASCII code 39 (single quote) is written as two consecutive single quotes.
- The character with ASCII code 92 (back slash) is written as two consecutive back slashes.
- All other characters are written in their '`\^`*`control char`*' form.

If character escaping is off:

- The character with ASCII code 39 (single quote) is written as two consecutive single quotes.
- All other characters are written as themselves.

Exceptions

Stream errors (see Section 8.7.7.2 [ref-iou-sfh-est], page 226)

See Also

`write_term/[2,3]`, `write/[1,2]`, `write_canonical/[1,2]`

Section 8.1.4 [ref-syn-ces], page 163 for information about character escaping.

19 C Reference Pages

19.1 Return Values and Errors

Quintus Prolog C functions return the following:

integer one of

`QP_SUCCESS` (0)

`QP_FAILURE` (-1)

`QP_ERROR` (-2)

boolean true (1), false (0)

pointer a pointer or null (0)

In some error situations, a global variable, `QP_errno` may also be set to give more information about the error condition.

Those I/O related predicates that take a Prolog stream argument do not set the error code in `QP_errno`. Instead, they set the errno field of the stream.

There are two `QP_` functions provided to help diagnose error conditions from error numbers:

`QP_perror()`
: prints out a user message, together with a short error message describing the last error encountered that set `QP_errno`. This function is similar to the system function `perror(3)`.

`QP_error_message()`
: returns a pointer to the diagnostic message corresponding to a specified error number.

In addition, the error number is sometimes reported in the message field of exception terms, as in

```
existence_error(Goal, _,_,_,errno(error number))
```

See Section 18.1.1 [mpg-ref-ove], page 985 for a description of the conventions observed in the Reference Pages for Prolog predicates. C function Reference Pages differ primarily in the synopsis.

19.2 Topical List of C Functions

Following is a complete list of Quintus-supplied C functions. They fall into two categories: built-in and user-redefinable. The user-redefinable functions are used in embedding Prolog sub-programs (see Section 10.2 [fli-emb], page 365). Quintus provides default definitions for these functions, and for most purposes it is unneccessary to even know about them. However, it is possible to redefine any of these to make Quintus Prolog programs behave appropriately when embedded in C code. Changing these default definitions means replacing them with your own code, not adding clauses.

By convention these functions are named to be recognizable as C functions, and as belonging to one or the other of these categories: Regular C builtins are prefixed with `QP_` and user-redefinable ones with `QU_`.

19.2.1 C Errors

`QP_perror()`
: prints an error message based on a QP error number

`QP_error_message()`
: gets the corresponding error message from a QP error number

`QU_error_message()`
: as `QP_error_message()`, but user-redefinable

19.2.2 Character I/O

`QP_char_count()`
: obtains the character count for a Prolog stream

`QP_fgetc()`
: gets a character from a Prolog input stream

`QP_fgets()`
: gets a string from a Prolog input stream

`QP_fpeekc()`
: looks a character ahead from a Prolog input stream

`QP_fprintf()`
: prints formatted output on a Prolog output stream

`QP_fputc()`
: puts a character on a Prolog output stream

`QP_fputs()`
: puts a character string on a Prolog output stream

`QP_fskipln()`
: skips the current input record of a Prolog input stream

`QP_getc()`
gets a character from a Prolog input stream

`QP_getchar()`
gets a character from the Prolog current input stream

`QP_newln()`
terminates an output record for a Prolog output stream

`QP_newline()`
terminates an output record for the Prolog current output stream

`QP_peekc()`
looks a character ahead from a Prolog input stream

`QP_peekchar()`
looks a character ahead from the Prolog current input stream

`QP_putc()`
puts a character on a Prolog output stream

`QP_putchar()`
puts a character on the Prolog current output stream

`QP_puts()`
puts a character string on the Prolog current output stream

`QP_skipline()`
skips the current input record of the Prolog current input stream

`QP_skipln()`
skips the current input record of a Prolog input stream

`QP_tab()` puts the specified character the number of times specified on a Prolog output stream

`QP_tabto()`
puts the specified character up to the specified line position on a Prolog output stream

`QP_ungetc()`
"unget"s the previous read character from a Prolog input input stream

19.2.3 Exceptions

`QP_exception_term()`
fetches the Prolog term representing the most recently raised exception

19.2.4 Files and Streams

`QP_add_tty()`
registers a created Prolog stream as a tty stream group

`QP_clearerr()`
clears the previous error on a Prolog stream

`QP_close()`
closes a Prolog stream

`QP_eof()` tests for the end of file on an input stream

`QP_eoln()`
tests for the end of record on an input stream

`QP_fclose()`
closes a Prolog stream

`QP_fdopen()`
creates a text stream or a binary stream from a file descriptor

`QP_ferror()`
tests error condition for a Prolog stream

`QP_flush()`
flushes output on a Prolog output stream

`QP_fnewln()`
terminates an output record for a Prolog output stream

`QP_fopen()`
opens a text file or a binary file as a Prolog stream

`QP_getpos()`
gets the current position for a Prolog stream

`QP_line_count()`
obtains the line count for a Prolog stream

`QP_line_position()`
obtains the line position for a Prolog stream

`QP_prepare_stream()`
initializes internal fields of a `QP_stream` structure

`QP_register_stream()`
registers a created Prolog stream

`QP_rewind()`
repositions a Prolog stream back to the beginning

`QP_seek()`
seeks to a random position in a Prolog stream

`QP_setinput()`
sets the Prolog current input stream

`QP_setoutput()`
sets the Prolog current output stream

`QP_setpos()`
positions a Prolog stream back to a previous read/written position

`QU_fdopen()`
creates streams opened by `QP_fdopen()`

`QU_initio()`
creates three Prolog initial streams

`QU_open()`
creates streams opened by `open/[3,4]`

`QU_stream_param()`
sets up default field values in a `QP_stream` stream structure

19.2.5 Foreign Interface

`QP_atom_from_padded_string()`
returns the Prolog atom corresponding to a blank-padded string; used with FORTRAN & Pascal

`QP_atom_from_string()`
returns the Prolog atom corresponding to a null-terminated string

`QP_close_query()`
closes a Prolog query opened from C by `QP_open_query()`

`QP_cons_functor()`
creates a Prolog compound term from C

`QP_cons_list()`
creates a Prolog list from C

`QP_cut_query()`
terminates a nondeterminate Prolog query opened from C

`QP_exception_term()`
returns the Prolog term to C corresponding to the most recent Prolog error

`QP_get_arg()`
fetches a specified argument of a compound term in a Prolog term reference

`QP_get_atom()`
fetches an atom from a Prolog term reference

`QP_get_float()`
fetches a floating point number from a Prolog term reference

`QP_get_functor()`
fetches the name and arity of a term in a Prolog term reference

`QP_get_head()`
fetches the head of a list in a Prolog term reference

`QP_get_integer()`
fetches an integer in a Prolog term reference

`QP_get_list()`
fetches the head and tail of a list in a Prolog term reference

`QP_get_tail()`
: fetches the tail of a list in a Prolog term reference

`QP_next_solution()`
: gets the next solution, if any, to an open Prolog query

`QP_open_query()`
: opens a Prolog query from C

`QP_padded_string_from_atom()`
: returns the blank-padded string corresponding to a Prolog atom; used with FORTRAN & Pascal

`QP_pred()`
: fetches an identifier for a Prolog predicate

`QP_predicate()`
: fetches an identifier a Prolog predicate

`QP_put_atom()`
: assigns an atom to a Prolog term reference

`QP_put_float()`
: assigns a floating point number to a Prolog term reference

`QP_put_functor()`
: assigns a new compound term to a Prolog term reference

`QP_put_integer()`
: assigns a Prolog integer to a Prolog term reference

`QP_put_list()`
: assigns a new list to a Prolog term reference

`QP_put_term()`
: assigns a Prolog term reference to another Prolog term reference

`QP_put_variable()`
: assigns a Prolog variable to a Prolog term reference

`QP_query()`
: makes a determinate query to a Prolog predicate

`QP_string_from_atom()`
: returns a null-terminated string corresponding to a Prolog atom

19.2.6 Input Services

`QP_add_input()`
: registers a function to be called when input occurs on a file descriptor

`QP_add_output()`
: registers a function to be called when output occurs on a file descriptor

`QP_add_exception()`
: registers a function to be called when an exception condition occurs on a file descriptor

`QP_add_timer()`
: arranges for a function to be called after a period of time

`QP_add_absolute_timer()`
: arranges for a function to be called at a given time

`QP_remove_input()`
: removes any input callbacks registered on a file descriptor

`QP_remove_output()`
: removes any output callbacks registered on a file descriptor

`QP_remove_exception()`
: removes any exception callbacks registered on a file descriptor

`QP_remove_timer()`
: removes a timer callback

`QP_select()`
: waits until I/O is ready on any of a set of file descriptors, or a timeout period occurs

`QP_wait_input()`
: waits until input is ready on a file descriptor or a timeout period occurs

19.2.7 main()

`QP_initialize()`
: initializes Prolog default

`QP_toplevel()`
: in a development system, calls the Prolog "read-prove" loop; in a runtime system, calls `runtime_entry/1`.

19.2.8 Memory Management

`QP_register_atom()`
: prevents an atom from being discarded by atom garbage collection even if not referenced by Prolog code

`QP_trimcore()`
: asks Prolog to purge all memory not in use

`QP_unregister_atom()`
: enables an atom to be discarded during atom garbage collection if not referenced by Prolog code

`QU_alloc_init_mem()`
: user-redefinable function to allocate memory for Prolog

`QU_alloc_mem()`
: user-redefinable function to allocate memory for Prolog

`QU_free_mem()`
: user-redefinable function to free memory from Prolog

19.2.9 Signal Handling

`QP_action()`
: requests certain kinds of Prolog action

19.2.10 Terms in C

`QP_compare()`
: compares two terms using Prolog's standard term order

`QP_new_term_ref()`
: returns a reference, which can be used to hold a Prolog term in C

`QP_unify()`
: unifies two Prolog terms

19.2.11 Term I/O

`QP_fread()`
: reads several items of data from a Prolog input stream

`QP_fwrite()`
: writes several items of data on a Prolog output stream

`QP_printf()`
: prints formatted output on the Prolog current output stream

`QP_vfprintf()`
: prints formatted output of a varargs argument list on a Prolog output stream

19.2.12 Type Tests

`QP_is_atom()`
: tests whether a Prolog term reference contains an atom

`QP_is_atomic()`
: tests whether a Prolog term reference contains an atomic term

`QP_is_compound()`
: tests whether a Prolog term reference contains a compound term

`QP_is_float()`
: tests whether a Prolog term reference contains a floating point number

`QP_is_integer()`
: tests whether a Prolog term reference contains a Prolog integer

`QP_is_list()`
: tests whether a Prolog term reference contains a list

`QP_is_number()`
: tests whether a Prolog term reference contains an integer or a floating point number

`QP_is_variable()`
: tests whether a Prolog term reference contains a Prolog variable

`QP_term_type()`
: returns the type of the term in a Prolog term reference

19.3 C Functions

The following reference pages, alphabetically arranged, describe the Quintus Prolog built-in C functions.

For a functional grouping of these functions including brief descriptions, see Section 19.2 [cfu-top], page 1346.

For information about return values and errors, see Section 19.1 [cfu-rve], page 1345

19.3.1 QP_action()

Synopsis

```
#include <quintus/quintus.h>

int QP_action(action)
int action;
```

Called to request certain actions of Prolog.

Arguments

`action` is one of:

`QP_ABORT` *Abort to the current break level

`QP_REALLY_ABORT`
*Abort to top level

`QP_STOP` Stop (suspend) process

`QP_IGNORE`
Do nothing

`QP_EXIT` Exit Prolog immediately

`QP_MENU` Present action menu

`QP_TRACE` Turn on trace mode

`QP_DEBUG` Turn on debugging

Description

This function allows the user to make Prolog abort, exit, suspend execution, turn on debugging, or prompt for the desired action.

Calls to `QP_action()` from an interrupt handler must be viewed as *requests*. They are requests that will definitely be honored, but not always at the time of the call to `QP_action()`. If Prolog is in a critical region the action might be delayed to when it has exitted the critical region.

Return Value

`QP_ERROR`

`QP_SUCCESS`

Errors

For systems that do not have a toplevel, the actions marked with an asterisk (*) will have no effect other than to make `QP_action()` return `QP_ERROR`.

Examples

For a full discussion of `QP_action()` and examples of its use, see Section 8.11.2 [ref-iex-iha], page 251

19.3.2 QP_add_*()

Synopsis

```
#include <quintus/quintus.h>

int QP_add_input(id,fn,data,flush_fn, flush_data)
int  id;
void (*fn)();
char *data;
void (*flush_fn)();
char *flush_data;

int QP_add_output(id,fn,data,flush_fn,flush_data)
int  id;
void (*fn)();
char *data;
void (*flush_fn)();
char *flush_data;

int QP_add_exception(id,fn,data,flush_fn,flush_data)
int  id;
void (*fn)();
char *data;
void (*flush_fn)();
char *flush_data;

int QP_add_timer(msecs,fn,data)
int  msecs;
void (*fn)();
char *data;

int QP_add_absolute_timer(timeo,fn,data)
struct timeval *timeo;
void (*fn)();
char *data;
```

These C functions register callback functions to be called on input/output or timing events.

Description

`QP_add_input()` arranges for a function to be called when input becomes available on the file descriptor *id*. The callback function *fn* is called with two arguments: the file descriptor *id* and the specified call data *data*.

Before the function is called, the callback is disabled so that the function will not be inadvertently reentered while it is running. The callback will be enabled automatically after the callback function returns.

If the flush function *flush_fn* is not `NULL` then it is called whenever Prolog needs to wait for input. This is useful when you communicate with another process using bidirectional buffered connections, where you must flush the output before you wait for input, lest your process waits for a response to a message that is still buffered in your output queue.

`QP_add_output()` is like `QP_add_input()` except that the callback function is called if output is ready on file descriptor *id*. `QP_add_exception()` is like `QP_add_input()` except that the callback function is called if an exception condition occurs on file descriptor *id*.

`QP_add_timer()` arranges for a function to be called in *msecs* milliseconds time with two arguments: the actual time waited and the specified call data *data*. This timer does not repeat automatically; if you want a repeating timer, you should call `QP_add_timer()` within the callback function explicitly.

`QP_add_absolute_timer()` is like `QP_add_timer()` except that an absolute time is specified by the timeval structure *timeo*; see `gettimeofday(2)`.

Return Values

timerid
: Returned by `QP_add_timer()` and `QP_add_absolute_time()` if successful.

`QP_SUCCESS`
: Returned by other functions.

`QP_ERROR`
: Returned by all functions if an error occurs.

Tips

Often your code will maintain a buffer associated with an input connection. If this is the case, then your flush function must check for this buffered input, and as long as it finds some, it should repeatedly call your callback function directly. If you don't do this, then your callback function may not be called, even though you have pending input, since the operating system isn't aware of your buffer.

See Also

`QP_wait_input()`, `QP_select()`, `QP_remove_*()`

19.3.3 `QP_add_tty()`

Synopsis

```
#include <quintus/quintus.h>

int QP_add_tty(stream, tty_id)
QP_stream *stream;
char      *tty_id;
```

Register a created Prolog stream to a tty stream group.

Arguments

stream	a pointer to a valid stream structure
tty_id	an identification string for a tty group

Description

This function is used to register a stream to a tty group. All the streams in a tty group share a single stream position (see the reference pages for `line_count/2`, `line_position/2` and `character_count/2`). When input is requested on one of the streams and the shared line position is 0, a prompt is output on one of the output streams.

See Also

Section 10.5.4 [fli-ios-tty], page 444

19.3.4 QP_atom_from_string(), QP_atom_from_padded_string()

Synopsis

```
#include <quintus/quintus.h>

QP_atom QP_atom_from_string(string)
char *string;
```

Returns the canonical representation of the atom whose printed representation is the (null-terminated) string *string*.

```
QP_atom QP_atom_from_padded_string(p_atom, p_string, p_length)
QP_atom *p_atom;
char    *p_string;
int     *p_length;
```

Computes the canonical representation of the atom whose printed representation is the (blank-padded) string *p_string* in a character array of length *p_length*.

Description

`QP_atom_from_string()` returns the canonical representation of the atom whose printed representation is *string*. *string* must be a valid null-terminated string. The string is copied and internalised by Prolog and the foreign function can reuse the string and its space.

`QP_atom_from_padded_string()` is useful for Pascal and FORTRAN and can be used with any language that has a C-compatible calling convention for passing integers and pointers (on the user's platform). e.g. some Pascal and FORTRAN compilers running under UNIX.

p_string is a pointer to a character array and *p_length* is a pointer to an integer specifying the length of the array. `QP_atom_from_padded_string()` sets the atom referenced by *p_atom* to the canonical representation of the atom whose printed representation is the string (less any trailing blanks) in the character array. It returns the length of the resulting atom (not the character array's length) as the function value.

Examples

`rev_atom()` is a C function that takes an atom and returns an atom whose string representation is the reverse of the string representation of the atom passed in.

foo.pl

```
foreign(rev_atom, c, rev_atom(+atom, [-atom])).
```

foo.c

```
QP_atom rev_atom(atom)
QP_atom atom;
{
    char *string[MAX_ATOM_LEN];

    strcpy(string, QP_string_from_atom(atom));
    reverse(string); /* reverses string in place */
    return QP_atom_from_string(string);
}
```

Giving:

```
| ?- rev_atom(draw, X).

X = ward

yes
| ?-
```

See Also

`QP_string_from_atom()`, `QP_padded_string_from_atom()`

Section 10.3.7 [fli-p2f-atm], page 389

19.3.5 `QP_char_count()`

Synopsis

```
#include <quintus/quintus.h>

int QP_char_count(stream)
QP_stream *stream;
```

Obtains the current character count for the Prolog stream *stream*. `QP_char_count()` is a macro.

Arguments

stream a pointer to a valid Prolog stream structure

See Also

`QP_get_pos()`, `QP_line_count()`, `QP_line_position()`, `character_count/2`, `QP_getpos()`, `QP_setpos()`, `QP_seek()`, `stream_position/[2,3]`

Section 10.5 [fli-ios], page 433

19.3.6 `QP_clearerr()`

Synopsis

```
#include <quintus/quintus.h>

void QP_clearerr(stream)
QP_stream *stream;
```

Resets the error indication and EOF indication to zero on the named stream.

`QP_clearerr()` is similar to the library function `clearerr(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

Section 10.5 [fli-ios], page 433

19.3.7 `QP_close_query()`

Synopsis

```
#include <quintus/quintus.h>

int QP_close_query(qid)
QP_qid qid;
```

Equivalent to the Prolog call:

```
!, fail.
```

Description

The cut renders the current computation determinate, removing the possibility of future backtracking. The following call to `fail/0` then initiates backtracking to the first parent goal with outstanding alternatives. In doing so it pops the Prolog heap to its state when the parent goal succeeded, in effect throwing away any terms created since that parent goal.

In the context of calling Prolog from foreign languages, terminating a query using `QP_close_query()` generally means throwing away the last solution that was calculated, unless that solution has been copied into a more permanent place. (Of course, any previous solutions must also be assumed to have been overwritten by subsequent solutions unless copied elsewhere!) The converse of this behavior is that closing a query using `QP_close_query()` automatically frees up the Prolog memory that holds the last solution.

Return Values

`QP_SUCCESS`
: Query was closed successfully

`QP_ERROR`
: either the query could not be closed or an exception was signalled from Prolog but not caught

See Also

`QP_cut_query()`, `QP_open_query()`, `QP_query()`, `QP_next_solution()`, `QP_pred()`, `QP_predicate()`, Section 10.4 [fli-ffp], page 413

19.3.8 QP_compare()

Synopsis

```
#include <quintus/quintus.h>

int QP_compare(term1, term2)
QP_term_ref term1;
QP_term_ref term2;
```

Compares the two terms referenced by *term1* and *term2*. Both arguments are *term1* before *term2*

Description

The comparison uses the standard total ordering of Prolog terms (also used by the built-in Prolog predicate `compare/3`).

In Standard Order:

Ret Value

term1 before *term2*
: -1

term1 same as *term2*
: 0

term1 after *term2*
: 1

Examples

`c_compare(term1, term2)` is an equivalent C version of the Prolog builtin `compare/3`:

foo.pl

```
foreign(c_compare, c, c_compare(+term, +term, +term)).
```

foo.c

```
#include <quintus/quintus.h>

int c_compare(t1, t2, t3)
QP_term_ref t1, t2;
{
   int res;
   QP_term_ref l_than = QP_new_term_ref();
   QP_term_ref equal  = QP_new_term_ref();
   QP_term_ref g_than = QP_new_term_ref();

   QP_put_atom(l_than, QP_atom_from_string("<"));
   QP_put_atom(equal,  QP_atom_from_string("="));
   QP_put_atom(g_than, QP_atom_from_string(">"));

   res = QP_compare(t2, t3);
   if ( res < 0) {
      return QP_unify(t1, l_than);
   } else if (res == 0) {
      return QP_unify(t1, equal);
   } else if (res > 0) {
      return QP_unify(t1, g_than);
   }
}
```

See Also:

`QP_unify()`, `compare/3`

19.3.9 QP_cons_*()

Synopsis

```
#include <quintus/quintus.h>

void QP_cons_list(term, head, tail)
QP_term_ref term;
QP_term_ref head;
QP_term_ref tail;

void QP_cons_functor(term, name, arity, arg1, ... ,arg_arity)
QP_term_ref   term;
QP_atom       name;
int           arity;
QP_term_ref   arg1, ... , arg_arity;
```

Description

These are C functions that can be used to create new Prolog terms from C.

`QP_cons_list()` assigns to *term* a reference to a list whose head is the term referred to by *head* and whose tail is the term referred to by *tail*.

`QP_cons_functor()` assigns to *term* a reference to a compound term whose functor is the atom represented by *name* and whose arity is the integer *arity*. The arguments of the compound term are terms referred to by *arg1*, *arg2*, etc. The call to this function should make sure that the number of arguments passed is equal to the arity of the compound term.

Note that the following are equivalent:

QP_cons_list(term, head, tail)

dot = QP_atom_from_string("."); QP_cons_functor(term, dot, 2, head, tail)

However, the former is likely to be more efficient.

Examples

`float_to_chars()` is a C function that converts a floating point number to a list of characters. Note the use of `QP_put_integer()`.

foo.pl

```
foreign(flt_to_chars, flt_to_chars(+float, -term)).
```

foo.c

```
#include <quintus/quintus.h>

void flt_to_chars(flt, chars)
double flt;
QP_term_ref chars;
{
   char buffer[28], *p;
   int len;
   QP_term_ref term_char = QP_new_term_ref();

   QP_put_nil(chars);
   sprintf( buffer , "%.17e" , flt );

   /* move to end of buffer */
   for (p=buffer, len=0; *p; p++, len++);

   while ( len-- )  {
      QP_put_integer(term_char, *--p);
      QP_cons_list(chars, term_char, chars);
   }
}
```

See Also:

`QP_term_type()`, `QP_get_*()`, `QP_new_term_ref()`

19.3.10 QP_cut_query()

Synopsis

```
#include <quintus/quintus.h>

int QP_cut_query(qid)
QP_qid qid;
```

Equivalent to just calling '!' in Prolog.

Description

The computation is rendered determinate, but as it is not failed over the Prolog heap is not popped. Thus when terminating a query using QP_cut_query() more space may be retained, but so is the last solution.

Return Values

QP_SUCCESS
QP_ERROR either something is wrong with the QP_qid or Prolog has not been initialized

See Also

QP_query(), QP_close_query(), QP_next_solution(), QP_open_query(), QP_pred(), QP_predicate()

Section 10.4 [fli-ffp], page 413

19.3.11 QP_error_message()

Synopsis

```
#include <quintus/quintus.h>

int QP_error_message(errno, is_qp_error, error_string)
int    errno;
int   *is_qp_error;
char **error_string;
```

Arguments

errno the error number in question

is_qp_error
set to:

0	if it *is not* an error number created by Quintus.
1	if it *is* an error number created by Quintus.

error_string
the text of the error message.

Description

This function supplies the text corresponding to the error number *errno*. The output parameter *is_qp_error* can be used to determine if the error number was one created by Quintus or is a system error number.

Typically, the error number of interest is the one in the global variable `QP_errno`. This variable is discussed in the man pages for `QP_perror()`.

Return Value

`QP_SUCCESS`

Examples

The following fragment takes the status value returned by some function that returns C calling Prolog style status values and prints out the corresponding error.

```
#include <quintus/quintus.h>
    ...
int is_qp_error;
char *error_message;
    ...
status = some_function();
switch(status) {
case QP_ERROR:
    (void) QP_error_message(QP_errno, &is_qp_error,
        &error_message);
    if(is_qp_error)
        (void) QP_fputs("prolog error: ", QP_stderr);
    else (void) QP_fputs("UNIX error: ", QP_stderr);
    (void) QP_fputs(error_message, QP_stderr);
    (void) QP_fnewln(QP_stderr);
    ...
```

See Also

`QP_perror()`

19.3.12 QP_exception_term()

Synopsis

```
#include <quintus/quintus.h>

int QP_exception(term)
QP_term_ref term;
```

A function that users can call when their call to Prolog signals an error. If `QP_query()` returns `QP_ERROR` then users can call `QP_exception_term()` to get at the exception term signalled.

Description

If C calls Prolog and the Prolog goal raises an exception, `QP_query()` (or `QP_next_solution()`) returns the value `QP_ERROR`. If the user wants to get at the exception term that has been raised, they can call the function `QP_exception_term()`. `QP_exception_term()` takes a `QP_term_ref` as argument and returns a Prolog term.

Example

foo.pl

```
:- extern(error).

error :- raise_exception(error_term(from_prolog)).
```

foo.c

```
        QP_pred_ref pred;

        if ((pred = QP_predicate("error",0,"user")) !=
            (QP_pred_ref) QP_ERROR) {

            if (QP_query(pred) == QP_ERROR) {
                QP_term_ref err_term = QP_new_term_ref();

                QP_exception_term(err_term);
            }
        }
```

Once you get `err_term`, you can use functions such as the `QP_get_*()` family to take apart the error term or to print it.

See Also

`QP_query()`, `QP_next_solution()`, `raise_exception/3`

Section 8.19 [ref-ere], page 310

19.3.13 `QP_fclose()`

Synopsis

```
#include <quintus/quintus.h>

int QP_fclose(stream)
QP_stream *stream;
```

Writes out any buffered data for the named stream, and closes the named stream.

`QP_fclose()` is similar to the library function `fclose(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fopen()`

Section 10.5 [fli-ios], page 433

19.3.14 QP_fdopen()

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QP_fdopen(fildes, type)
int   fildes;
char *type;
```

Associates a stream with the file descriptor files. File descriptors are obtained from system calls like `open(2V)`, `dup(2)`, `creat(2)`, or `pipe(2)`, which open files but do not return streams. Streams are necessary input for many of the system functions. The type of the stream must agree with the mode of the open file.

`QP_fdopen()` is similar to the library function `fdopen(3V)`, however the return values differ and the normal return value is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Ablso

`QP_fclose()`, `QP_fopen()`, `QP_prepare_stream/[3,4]`

Section 10.5 [fli-ios], page 433

19.3.15 QP_ferror()

Synopsis

```
#include <quintus/quintus.h>

int QP_ferror(stream)
QP_stream *stream;
```

Returns non-zero when an error has occurred reading from or writing to the named stream, otherwise zero.

`QP_ferror()` is similar to the library function `ferror(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fopen()`

Section 10.5 [fli-ios], page 433

19.3.16 QP_fgetc()

Synopsis

```
#include <quintus/quintus.h>

int QP_fgetc(stream)
QP_stream *stream;
```

Behaves like `QP_getc()`, but is a function rather than a macro.

`QP_fgetc()` is similar to library function `fgetc(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_ferror()`, `QP_fopen()`, `QP_fread()`, `QP_putc()`, `QP_ungetc()`

Section 10.5 [fli-ios], page 433

19.3.17 `QP_fgets()`

Synopsis

```
#include <quintus/quintus.h>

char *QP_fgets(s, n, stream)
char      *s;
int        n;
QP_stream *stream;
```

Reads characters from the stream into the array pointed to by *s*, until *n*-1 characters are read, a NEWLINE character is read and transferred to *s*, or an EOF condition is encountered. The string is then terminated with a NULL character.

`QP_fgets()` is similar to the library function `fgets(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

Section 10.5 [fli-ios], page 433

19.3.18 QP_flush()

Synopsis

```
#include <quintus/quintus.h>

int QP_flush(stream)
QP_stream *stream;
```

Forces the buffered output of the stream *stream* to be sent to the associated device.

Arguments

stream pointer to a valid stream structure.

Description

Calls the bottom layer flushing function of *Stream* to write out the current buffered output of the stream. The output is usually written out to a disk or a tty device.

Return Value

`QP_SUCCESS`
: The function succeeds

`QP_ERROR`
: There is an error in the function call, the error number is stored in both `QP_errno` and *stream->errno*.

Errors

`QP_E_PERMISSION`
: *stream* is not an output stream or it does not permit flushing.

`QP_E_CANT_WRITE`
: Unknown error in the bottom layer of flush function of *stream*

: Errors from host operating system

See Also

`flush_output/1`.

19.3.19 `QP_fnewln()`

Synopsis

```
#include <quintus/quintus.h>

int QP_fnewln(stream)
QP_stream *stream;
```

Terminates an output record for a Prolog output stream.

Arguments

stream pointer to a valid stream structure

See Also

`QP_newline()`, `QP_newln()`, `nl/[0,1]`

Section 10.5 [fli-ios], page 433

19.3.20 QP_fopen()

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QP_fopen(filename, type)
unsigned char *filename;
char          *type;
```

Opens the file named by filename and associates a stream with it.

`QP_fopen()` is similar to the library function `fopen(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fopen()`, `QP_fdopen()`, `QP_prepare_stream/[3,4]`

Section 10.5 [fli-ios], page 433

19.3.21 QP_fpeekc()

Synopsis

```
#include <quintus/quintus.h>

int QP_fpeekc(stream)
QP_stream *stream;
```

Look ahead for the next character to be read in from a Prolog input stream.

Arguments

stream pointer to a valid stream structure

Return Value

Character code or QP_ERROR.

See Also

QP_peekc(), QP_peekchar(), peek_char/[1,2]

19.3.22 QP_fprintf()

Synopsis

```
#include <quintus/quintus.h>

int QP_fprintf(stream, format [ , arg ]...)
QP_stream *stream;
char      *format;
```

Places output onto the Prolog output stream *stream*.

`QP_fprintf()` is similar to the library function `fprintf(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR`
: Otherwise

See Also

`QP_printf()`, `QP_putc()`

Section 10.5 [fli-ios], page 433

19.3.23 QP_fputc()

Synopsis

```
#include <quintus/quintus.h>

int QP_fputc(c, stream)
int        c;
QP_stream *stream;
```

Behaves like `QP_putc()`, but is a function rather than a macro.

`QP_fputc()` is similar to the library function `fputc(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fclose()`, `QP_ferror()`, `QP_fopen()`, `QP_fread()`, `QP_getc()`, `QP_printf()`, `QP_puts()`

Section 10.5 [fli-ios], page 433

19.3.24 QP_fputs()

Synopsis

```
#include <quintus/quintus.h>

int QP_fputs(s, stream)
unsigned char *s;
QP_stream     *stream;
```

Writes the NULL-terminated string pointed to by *s* to the named output stream.

`QP_fputs()` is similar to the library function `fputs(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_ferror()`, `QP_fopen()`, `QP_fread()`, `QP_printf()`, `QP_putc()`

Section 10.5 [fli-ios], page 433

19.3.25 QP_fread()

Synopsis

```
#include <quintus/quintus.h>

int QP_fread (ptr, size, nitems, stream)
char      *ptr;
int        size;
int        nitems;
QP_stream *stream;
```

Reads, into a block pointed to by *ptr*, *nitems* items of data from the named input stream *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a NULL byte) of length *size*.

`QP_fread()` is similar to the library function `fread(3S)`, however the return values differ slightly and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

the number of items read
: Returned if the function succeeds

`QP_ERROR`
: Otherwise

See Also

`QP_fopen()`, `QP_getc()`, `QP_gets()`, `QP_putc()`, `QP_puts()`, `QP_printf()`

Section 10.5 [fli-ios], page 433

19.3.26 QP_fskipln()

Synopsis

```
#include <quintus/quintus.h>

int QP_fskipln(stream)
QP_stream *stream;
```

Skip the current input record of a Prolog input stream

Return Value

QP_SUCCESS, or QP_ERROR

Errors

When QP_ERROR is returned, QP_errno contains an error code.

See Also

skip_line/[0,1]

Section 10.5 [fli-ios], page 433

19.3.27 `QP_fwrite()`

Synopsis

```
#include <quintus/quintus.h>

int QP_fwrite(ptr, size, nitems, stream)
char      *ptr;
int        size;
int        nitems;
QP_stream *stream;
```

Writes at most *nitems* items of data from the block pointed to by *ptr* to the named Prolog output stream. `QP_fwrite()` stops writing when it has written *nitems* of date or if an error condition is encountered on *stream*.

`QP_fwrite()` is similar to the library function `fwrite()`.

Return Value

the number of items written
: Returned if the function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fopen()`, `QP_getc()`, `QP_gets()`, `QP_putc()`, `QP_puts()`, `QP_printf()`

Section 10.5 [fli-ios], page 433

19.3.28 QP_get_*()

Synopsis

```
#include <quintus/quintus.h>

int QP_get_atom(term, atom)
QP_term_ref      term;
QP_atom         *atom;

int QP_get_integer(term, integer)
QP_term_ref      term;
long int        *integer;

int QP_get_float(term, float)
QP_term_ref      term;
double          *float;

int QP_get_functor(term, name, arity)
QP_term_ref      term;
QP_atom         *name;
int             *arity;

int QP_get_arg(argnum, term, arg)
int              argnum;
QP_term_ref      term;
QP_term_ref      arg;

int QP_get_list(term, head, tail)
QP_term_ref      term;
QP_term_ref      head;
QP_term_ref      tail;

int QP_get_head(term, head)
QP_term_ref      term;
QP_term_ref      head;

int QP_get_tail(term, tail)
QP_term_ref      term;
QP_term_ref      tail;

int QP_get_nil(term)
QP_term_ref      term;

int QP_get_db_reference(term, ref)
QP_term_ref      term;
QP_db_reference *ref;
```

These C functions can be used to test and access Prolog terms passed to C through the foreign interface.

Description

If *term* refers to an atom then `QP_get_atom()` assigns to **atom* the unsigned integer representing that atom and returns 1. Else `QP_get_atom()` returns 0. To get at the string corresponding to the atom, use `QP_string_from_atom()`.

If *term* refers to a Prolog integer then `QP_get_integer()` assigns that integer to **integer* and returns 1. Else `QP_get_integer()` returns 0.

If *term* refers to a floating point number then `QP_get_float()` assigns that number to **float* and returns 1. Else `QP_get_float()` returns 0.

If *term* refers to a compound term then `QP_get_functor()` assigns to **name* the unsigned integer representing the name of the functor, assigns to **arity* the arity of the functor and returns 1. If *term* refers to an atom, then `QP_get_functor()` assigns to **name* that atom, assigns 0 to **arity* and returns 1. If *term* does not refer to a compound term or an atom then `QP_get_functor()` returns 0. Note that a list is a compound term with functor `.` and arity 2.

If *term* refers to a compound term and *argnum* is between 1 and the arity of the compound term then `QP_get_arg()` assigns to *arg* a reference to the *argnum* argument of the compound term and returns 1. If *term* does not refer to a compound term `QP_get_arg()` returns 0. Note that `QP_get_arg()` is similar to the Prolog builtin `arg/3` with its first and second arguments bound and its third argument unbound. `QP_get_arg()` differs from the other `QP_get` functions in that it does not have *term* as its first argument. This is to make it consistent with `arg/3`.

If *term* refers to a list then `QP_get_list()` assigns to *head* a reference to the head of that list, assigns to *tail* a reference to the tail of the list and returns 1. If *term* does not refer to a list then `QP_get_list()` returns 0.

If *term* refers to a list then `QP_get_head()` assigns to *head* a reference to the head of that list and returns 1. If *term* does not refer to a list then `QP_get_head()` returns 0.

If *term* refers to a list then `QP_get_tail()` assigns to *tail* a reference to the tail of that list and returns 1. If *term* does not refer to a list then `QP_get_tail()` returns 0.

If *term* refers to the atom `[]` then `QP_get_nil()` returns 1. Else it returns 0.

If *term* refers to a db_reference (e.g. returned by `asserta/3` or `recorda/3`) then `QP_get_db_reference()` assigns to **ref* that reference and returns 1. If *term* does not refer to a db_reference then `QP_get_db_reference()` returns 0.

Examples

`write_term()` is a C function that writes out a Prolog term passed to it.

foo.pl

```
foreign(write_term, c, write_term(+term)).
```

foo.c

```
#include <quintus/quintus.h>

void write_term(term)
QP_term_ref term;
{
    QP_atom a;
    long int i;
    double d;
    switch (QP_term_type(term)) {
    case QP_VARIABLE:
        QP_printf("_");
        break;
    case QP_INTEGER:
        QP_get_integer(term, &i);
        QP_printf("%d", i);
        break;
    case QP_FLOAT:
        QP_get_float(term, &d);
        QP_printf("%f", d);
        break;
    case QP_ATOM:
        QP_get_atom(term, &a);
        QP_printf("%s", QP_string_from_atom(a));
        break;
    case QP_DB_REFERENCE:
        QP_printf("'$ref'()");
        break;
    case QP_COMPOUND:
        if (QP_is_list(term)) {
            write_list(term);
        } else {
            write_compound(term);
        }
        break;
    }
}
```

foo.c

```
void write_list(term)
QP_term_ref term;
{
    QP_term_ref head = QP_new_term_ref();
    QP_term_ref tail = QP_new_term_ref();
    QP_atom a;

    QP_printf("[");
    QP_get_list(term, head, tail);
    write_term(head);
    while (QP_is_list(tail)) {
        QP_printf(",");
        QP_get_list(tail, head, tail);
        write_term(head);
    }
    if (QP_get_nil(tail)) {
        QP_printf("]");
    } else {
        QP_printf("|");
        write_term(tail);
        QP_printf("]");
    }
}

void write_compound(term)
QP_term_ref term;
{
    int i, arity;
    QP_atom name;
    QP_term_ref arg = QP_new_term_ref();

    QP_get_functor(term, &name, &arity);
    QP_printf("%s(", QP_string_from_atom(name));
    for (i = 1; i < arity; i++) {
        QP_get_arg(i, term, arg);
        write_term(arg);
        QP_printf(",");
    }
    QP_get_arg(i, term, arg);
    write_term(arg);
    QP_printf(")");
}
```

See Also

`QP_term_type()`, `QP_put_*()`, `QP_new_term_ref()`

19.3.29 QP_getchar()

Synopsis

```
#include <quintus/quintus.h>

int QP_getchar()
```

Defined as `QP_getc(QP_curin)`.

`QP_getchar()` is similar to the library function `getchar(3V)`, however it operates on the Prolog current input stream rather than the standard input stream `stdin`. Like `getchar(3V)`, `QP_getchar()` is a macro.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_ferror()`, `QP_fopen()`, `QP_fread()`, `QP_gets()`, `QP_putc()`, `QP_ungetc()`

Section 10.5 [fli-ios], page 433

19.3.30 QP_getpos()

Synopsis

```
#include <quintus/quintus.h>

int QP_getpos(stream, pos)
QP_stream   *stream;
QP_position *pos;
```

Get the current position for a Prolog stream

Arguments

stream	pointer to a valid stream structure
pos	pointer to a `QP_position` structure

Description

Upon successful return of this function call, the char_count, line_count, line_position and magic member of the `QP_position` structure pointed to by *pos* have the valid values indicating the current character count, line count, line position and system-dependent position. The *pos* can be passed as the second argument to `QP_setpos()` for seeking back to the current position later.

Return Value

Always returns `QP_SUCCESS`.

See Also

`QP_setpos()`, `QP_seek()`, `stream_position/[2,3]`, `QP_char_count()`, `QP_line_count()`, `QP_line_position()`

Section 10.5 [fli-ios], page 433

19.3.31 QP_initialize()

Synopsis

```
#include <quintus/quintus.h>

int QP_initialize(argc, argv)
int    argc;
char **argv;
```

Initializes Prolog's memory management, I/O, symbol table, etc.

Arguments

argc	number of command line arguments (or 0)
argv	list of command line arguments (or NULL)

Description

You can ignore `QP_initialize()` if you aren't redefining `main()`.

Must be called before any other QP_ functions if you are redefining `main()` (in which case your top-level view of Prolog is via `QP_predicate()` and `QP_query()`).

Summary of functionality: Initializes memory, I/O; sets up command line arguments; initializes file search paths, file tables and symbol tables; do initializations and start up hooks associated with a statically linked component in qof files; do any necessary restores, and any initlization and start up hooks associated with the restored files.

`QP_initialize()` also sets up signal handlers so that users can interrupt the execution of a start-up goal or initialization with a `^c`. If users chose the `a` option after a `^c` (or if the builtin `abort/0` is called) when initializations are run, then `QP_initialize()` returns. In a default system, (where `main()` hasnt been redefined) this means that `QP_toplevel()` gets called. `QP_toplevel()` executes the toplevel read-prove loop in a development system. In a runtime system, it results in `runtime_entry(start)` being called.

argc and *argv* are necessary for Prolog to execute the builtin `unix/1` (e.g. `unix(argv(_))` etc.) properly, as well as for restoring saved states.

Can be safely called any number of times.

> **Please note:** The first call to `QP_initialize()` with non-null arguments will determine the command line arguments as seen by Prolog.

Return Value

`QP_SUCCESS`
: Prolog was successfully initialized.

`QP_FAILURE`
: otherwise

Examples

In Quintus Prolog the default implementation of `main()` looks like this:

```
#include <quintus/quintus.h>

main(argc, argv)
    int argc;
    char **argv;
    {
        int status;

        status = QP_initialize(argc, argv);

        if (status == QP_SUCCESS) QP_toplevel();
    }
```

The user can choose not to have the default `main()` and the default toplevel loop. Here is an example of how the user can call a Prolog predicate with their own `main()`.

hello.pl

```
:- extern(hi(+atom)).

hi(X) :-
    format('Hello world from ~a to Prolog~n',[X]).
```

main.c

```
#include <quintus/quintus.h>

main(argc, argv)
    int argc;
    char **argv;
    {
        int status;
        QP_pred_ref pred;

        status = QP_initialize(argc, argv);

        if (status == QP_SUCCESS) {
           pred = QP_predicate("hi", 1, "user");

           if (pred != QP_BAD_PREDREF) {
                status = QP_query(pred,
                         QP_atom_from_string("C"));

                if (status == QP_FAILURE) {
                    printf("hi/1 failed\n");
                    exit(1);
                } else if (status == QP_ERROR) {
                  printf("hi/1 raised exception\n");
                    /* Use QP_exception_term to get
                       the error term signaled */
                    exit(1);
                }
           } else {
                printf("hi/1 doesn't exist or ");
                printf("doesn't have an extern ");
                printf("declaration\n");
                exit(1);
           }
        } else {
           printf("QP_initialize didn't succeed\n");
           exit(1);
        }
    }
```

Steps to produce the executable:

1. Compile '`hello.pl`' using *`qpc -c hello.pl`*
2. Compile '`main.c`' using *`cc -c main.c`*
3. Link the two using *`qld -Dd hello.qof main.o -o qtest`*
4. Run `qtest`. The output should be: '`Hello from C to Prolog`'

See Also:

`runtime_entry/1`, `unix/1`, `QP_predicate()`, `QP_query()`, `QP_toplevel()`

Section 10.2 [fli-emb], page 365

19.3.32 QP_is_*()

Synopsis

```
#include <quintus/quintus.h>

int QP_is_variable(term)
QP_term_ref term;

int QP_is_atom(term)
QP_term_ref term;

int QP_is_integer(term)
QP_term_ref term;

int QP_is_float(term)
QP_term_ref term;

int QP_is_compound(term)
QP_term_ref term;

int QP_is_list(term)
QP_term_ref term;

int QP_is_db_reference(term)
QP_term_ref term;

int QP_is_atomic(term)
QP_term_ref term;

int QP_is_number(term)
QP_term_ref term;
```

These C functions and macros can be used to test the type of the Prolog terms passed to C through the foreign interface.

Description

`QP_is_variable()`
: (macro) returns a nonzero value if its argument is a Prolog variable; zero otherwise.

`QP_is_atom()`
: (macro) returns a nonzero value if its argument is an atom; zero otherwise.

`QP_is_integer()`
: (macro) returns a nonzero value if its argument is a Prolog integer; zero otherwise.

`QP_is_float()`
: (macro) returns a nonzero value if its argument is a float; zero otherwise.

`QP_is_compound()`
: (macro) returns a nonzero value if its argument is a compound Prolog term; zero otherwise.

`QP_is_list()`
: (function) returns a nonzero value if its argument is a Prolog list; zero otherwise.

`QP_is_db_reference()`
: (macro) returns a nonzero value if its argument is a database reference; zero otherwise.

`QP_is_atomic()`
: (function) returns a nonzero value if its argument is an atomic Prolog object; zero otherwise.

`QP_is_number()`
: (function) returns a nonzero value if its argument is a Prolog integer or float; zero otherwise.

Examples

`print_type()` is a C function that prints the type of the Prolog term passed to it.

foo.pl

```
foreign(print_type, c, print_type(+term)).
```

foo.c

```
#include <quintus/quintus.h>

void print_type(term)
QP_term_ref term;
{
    if (QP_is_atom(term)) {
        QP_printf("Term is an atom\n");
    } else if (QP_is_integer(term)) {
        QP_printf("Term is an integer\n");
    } else if (QP_is_float(term)) {
        QP_printf("Term is a float\n");
    } else if (QP_is_variable(term)) {
        QP_printf("Term is a variable\n");
    } else if (QP_is_db_reference(term)) {
        QP_printf("Term is a database reference\n");
    } else if (QP_is_compound(term)) {
        if (QP_is_list(term)) {
            QP_printf("Term is a list\n");
        } else {
            QP_printf("Term is a compound term\n");
        }
    } else {
        QP_printf("Unrecognized term\n");
    }
}
```

See Also

`QP_get_*()`, `QP_put_*()`, `QP_new_term_ref()`

19.3.33 `QP_line_count()`

Synopsis

```
#include <quintus/quintus.h>

int QP_line_count(stream)
QP_stream *stream;
```

Obtains the line count for a Prolog stream. `QP_line_count()` is a macro.

Arguments

stream pointer to a valid stream structure

See Also

`QP_getpos()`, `QP_setpos()`, `QP_seek()`, `stream_position/[2,3]` `QP_char_count()`, `QP_line_position()`

Section 10.5 [fli-ios], page 433

19.3.34 `QP_line_position()`

Synopsis

```
#include <quintus/quintus.h>

int QP_line_position(stream)
QP_stream *stream;
```

Obtains the line position for a Prolog stream. `QP_line_position()` is a macro.

Arguments

stream pointer to a valid stream structure

See Also

`QP_getpos()`, `QP_setpos()`, `QP_seek()`, `stream_position/[2,3]` `QP_char_count()`, `QP_line_count()`,

Section 10.5 [fli-ios], page 433

19.3.35 `QP_malloc()`, `QP_free()`

Synopsis

```
#include <quintus/quintus.h>

void *QP_malloc(size)
int  size;
```

A replacement for the C function `malloc()`.

```
void QP_free(mem)
void *mem;
```

A replacement for the C function `free()`.

Description

These function provide memory allocation and deallocation via Prolog's embeddable memory allocation layer, instead of directly via the C library. Using the embeddable memory allocation layer tends to keep memory fragmentation down.

See Also

Section 10.2 [fli-emb], page 365

19.3.36 QP_new_term_ref()

Synopsis

```
#include <quintus/quintus.h>

QP_term_ref QP_new_term_ref()
```

Description

`QP_new_term_ref()` returns an initialized `QP_term_ref`. Every `QP_term_ref` declared has to be initialized with a call to this function. A `QP_term_ref` can be considered as a reference to a Prolog term. Calling this function initialises that reference to a location where terms can be stored. The actual term that the reference points to is initialized to `[]`.

Example

`create_term()` is a simple C function that returns different types of Prolog terms depending on its first argument.

foo.pl

```
foreign(create_term, create_term(+integer,[-term])).
```

foo.c

```
#include <quintus/quintus.h>

QP_term_ref create_term(kind);
long int kind;
{
    QP_term_ref new = QP_new_term_ref();

    switch (kind) {
      case 1:
        QP_put_integer(new, 23);
        break;
      case 2:
        QP_put_atom(new, QP_atom_from_string("Ayn"));
        break;
      case 3:
        QP_put_float(new, 1.1);
        break;
      default:
        QP_put_nil(new);
        break;
    }
    return new;
}
```

See Also:

`QP_put_*()`, `QP_term_type()`, `QP_get_*()`

19.3.37 `QP_newline()`

Synopsis

```
#include <quintus/quintus.h>

int QP_newline()
```

Terminates the current output record for the Prolog current output stream. `QP_newline()` is a macro.

Arguments

stream pointer to a valid stream structure

Description

Calling `QP_newline()` is equivalent to call `QP_newln(QP_stdout)`.

Return Value

a line border character or `QP_ERROR`

See Also

`QP_newln()`, `QP_error_message()`, `nl/[0,1]`

Section 10.5 [fli-ios], page 433

19.3.38 QP_newln()

Synopsis

```
#include <quintus/quintus.h>

int QP_newln(stream)
QP_stream *stream;
```

Terminates the current output record for a specified Prolog output stream. `QP_newln()` is a macro.

Arguments

stream pointer to a valid stream structure

Return Value

a line border character or `QP_ERROR`

See Also

`QP_newline()`, `QP_error_message()`, `nl/[0,1]`

Section 10.5 [fli-ios], page 433

19.3.39 QP_next_solution()

Synopsis

```
#include <quintus/quintus.h>

int QP_next_solution(qid)
QP_qid qid;
```

Returns the next solution (if any) from an open nondeterminate Prolog query.

Description

Solutions are computed on demand, and multiple solutions are returned in the normal Prolog order. `QP_next_solution()` is passed the `QP_qid` returned by `QP_open_query()` when the nondeterminate query was opened. No additional input or output parameters are passed: after a call to `QP_open_query()`, Prolog manages inputs itself, and has been told where storage for outputs has been reserved.

Each time `QP_next_solution()` computes a new solution it writes it on the output storage for the foreign function to use as it likes. Each new solution overwrites the old memory, destroying the previous solution, so it is important that the foreign function copies solutions elsewhere if it wants to accumulate them.

Comment

An important restriction: only the innermost, i.e. the most recent, open query can be asked to compute a solution. A new nondeterminate query can be made at any point whether or not other queries are open; however, while the new query remains open only it will be able to return solutions. Of course, determinate queries can be made at any time.

Return Values

`QP_SUCCESS`
: Solution found

`QP_FAILURE`
: No solution found

`QP_ERROR`

See Also

`QP_cut_query()`, `QP_close_query()`, `QP_query()`, `QP_open_query()`, `QP_pred()`, `QP_predicate()`,

Section 10.4 [fli-ffp], page 413

19.3.40 QP_open_query()

Synopsis

```
#include <quintus/quintus.h>

QP_qid QP_open_query(pred_ref, arg1,...,arg255)
QP_pred_ref pred_ref;
```

Initiates a nondeterminate Prolog query.

Description

The first argument passed to `QP_query()` is a reference to the Prolog predicate to be called. `QP_query()` accepts between 0 and 255 arguments after its first argument. Any arguments after the first represent parameters to be passed to and from the Prolog predicate. For the types of arguments that may be passed between C and Prolog predicates, see Section 10.4 [fli-ffp], page 413.

The arguments passed to `QP_open_query()` are identical to those that would be passed to `QP_query()`; however, `QP_open_query()` does not compute a solution to the query. Its effect is to prepare Prolog for the computation of solutions to the query, which must be initiated using `QP_next_solution()`. For consistency checking, `QP_open_query()` returns a `QP_qid`, which represents the Prolog query. The type definition for `QP_qid` is found in the file '`<quintus/quintus.h>`'.

The `QP_qid` returned by a call to `QP_open_query()` must be passed to each call to `QP_next_solution()` for that query, as well as to `QP_cut_query()` or `QP_close_query()` when terminating the query.

When requesting solutions from an open nondeterminate query, input and output parameters are *not* passed. The effect of `QP_open_query()` is to pass inputs to Prolog, which subsequently maintains them. It also tells Prolog where storage for outputs has been reserved. This storage will be written to when solutions are returned.

Return Value

`QP_qid` query was opened successfully

`QP_ERROR` an error occurred when attempting to open the query and the query was automatically terminated

See Also

`QP_cut_query()`, `QP_close_query()`, `QP_query()`, `QP_next_solution()`, `QP_pred()`, `QP_predicate()`,

Section 10.4 [fli-ffp], page 413

19.3.41 `QP_peekc()`

Synopsis

```
#include <quintus/quintus.h>

int QP_peekc(stream)
QP_stream *stream;
```

Look a character ahead from a specified Prolog input stream. `QP_peekc()` is a macro.

Arguments

stream pointer to a valid stream structure

Return Value

Character code or `QP_ERROR`.

See Also

`QP_peekchar()`, `QP_fpeekc()`, `peek_char/[1,2]`

Section 10.5 [fli-ios], page 433

19.3.42 QP_peekchar()

Synopsis

```
#include <quintus/quintus.h>

int QP_peekchar()
```

Look a character ahead from the Prolog current input stream. `QP_peekchar()` is a macro.

Description

`QP_peekchar()` is equvalent to `QP_peekc(QP_stdin)`.

Return Values

Character code or `QP_ERROR`.

See Also

`QP_peekc()`, `QP_fpeekc()`, `peek_char/[2,3]`

Section 10.5 [fli-ios], page 433

19.3.43 `QP_perror()`

Synopsis

```
#include <quintus/quintus.h>

void QP_perror(s)
char *s;
```

Description

`QP_perror()` produces a short error message on the stream `user_error` describing the last error encountered. If *s* is not a NULL pointer and does not point to a null string, the string it points to is printed, followed by a colon, followed by a space, followed by the message and a NEWLINE. If *s* is a NULL pointer or points to a null string, just the message is printed, followed by a NEWLINE. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *QP_errno* (see Section 19.1 [cfu-rve], page 1345), which is set when errors occur but not cleared when non-erroneous calls are made.

This function is modeled on the system function `perror(3)`.

See Also

`QP_error_message()`

19.3.44 QP_pred()

Synopsis

```
#include <quintus/quintus.h>

QP_pred_ref QP_pred(name_atom, arity, module_atom)
QP_atom name_atom;
int     arity;
QP_atom module_atom;
```

Looks up a callable Prolog predicate.

`QP_pred()` is faster, but less convenient, than `QP_predicate()`

Description

Differences from `QP_predicate()`: Name and module arguments passed as Prolog atoms. These may have been returned to C from Prolog, or may have been built in the foreign language using `QP_atom_from_string()`.

The name passed is *not* the name of the Prolog predicate to be called, but rather the name of the interface predicate constructed when the Prolog predicate was made callable from foreign code (see Section 10.4.2 [fli-ffp-ppc], page 414).

Much of the cost of `QP_predicate()` is from having to look up Prolog atoms for its name and module arguments. By avoiding doing this unnecessarily, what `QP_pred()` gives up in convenience is returned in performance.

Return Value

`QP_pred_ref`
: a valid predicate reference

`QP_ERROR` if the predicate called hasn't been declared callable or doesn't exist

See Also

`QP_predicate()`, `QP_query()`, `QP_open_query()`, `QP_next_solution()`, `QP_cut_query()`, `QP_close_query()`, Section 10.4 [fli-ffp], page 413

19.3.45 QP_predicate()

Synopsis

```
#include <quintus/quintus.h>

QP_pred_ref QP_predicate(name_string, arity, module_string)
char *name_string;
int   arity;
char *module_string;
```

Description

Before a Prolog predicate can be called from a foreign language it must be looked up. The C functions `QP_predicate()` and `QP_pred()` perform this function. The lookup step could have been folded into the functions that make the query, but if a predicate was to be called many times the redundant, if hidden, predicate lookup would be a source of unnecessary overhead. Instead, `QP_predicate()` or `QP_pred()` can be called just once per predicate. The result can then be stored in a variable and used as necessary.

Both `QP_predicate()` and `QP_pred()` return a `QP_pred_ref()`, which represents a Prolog predicate.

`QP_predicate()` is the most convenient way of looking up a callable Prolog predicate. It is simply passed the name and module of the predicate to be called as strings, the arity as an integer, and returns a `QP_pred_ref()`, which is used to make the actual call to Prolog.

`QP_predicate()` can only be used to look up predicates that have been declared callable from foreign code. If some other predicate, or a predicate that does not exist, is looked up, `QP_ERROR` is returned. This protects you from attempting to call a predicate that isn't yet ready to be called.

Return Value

`QP_pred_ref`
: a valid predicate reference

`QP_ERROR` if the predicate called hasn't been declared callable or doesn't exist

See Also

`QP_pred()`, `QP_query()`, `QP_open_query()`, `QP_next_solution()`, `QP_cut_query()`, `QP_close_query()`

Section 10.4 [fli-ffp], page 413

19.3.46 QP_prepare_stream()

Synopsis

```
#include <quintus/quintus.h>

void QP_prepare_stream(stream, buffer)
QP_stream     *stream;
unsigned char *buffer;
```

Initialize internal fields of a `QP_stream` structure.

Arguments

stream	pointer to a valid stream structure
buffer	pointer to a buffer

Description

`QP_prepare_stream()` should be called after other fields in `QP_stream` are properly set up.

The first parameter is a pointer to `QP_stream` and the second parameter is the address of the input/output buffer for the stream. Here *&stream->qpinfo* is used to get the corresponding `QP_stream` pointer from *stream* although a casting operation of `QP_stream` **stream* will have the same effect.

Example

```
QP_prepare_stream(&stream->qpinfo, stream->buffer);
```

See Also

`QP_fdopen()`, `QP_fopen()`, `open/[3,4]`,

Section 10.5.5.6 [fli-ios-cps-ire], page 450

19.3.47 QP_printf()

Synopsis

```
#include <quintus/quintus.h>

int QP_printf(format [ , arg ] ... )
char *format;
```

Places output onto the current Prolog output stream.

`QP_printf()` is similar to the library function `printf(3V)`, however the return values differ it puts its output on the current Prolog output stream (`QP_curout`) rather than a `stdio` stream.

Return Value

the number characters written
: Returned if the function succeeds

`QP_ERROR` Otherwise

See Also

`QP_putc()`

Section 10.5 [fli-ios], page 433

19.3.48 QP_put_*()

Synopsis

These C functions can be used to create new Prolog terms from C.

```
#include <quintus/quintus.h>

void QP_put_variable(term)
QP_term_ref     term;

void QP_put_atom(term, atom)
QP_term_ref     term;
QP_atom         atom;

void QP_put_integer(term, integer)
QP_term_ref     term;
long int        integer;

void QP_put_float(term, float)
QP_term_ref     term;
double          float;

void QP_put_functor(term, name, arity)
QP_term_ref     term;
QP_atom         name;
int             arity;

void QP_put_list(term)
QP_term_ref     term;

void QP_put_nil(term)
QP_term_ref     term;

void QP_put_term(term1, term2)
QP_term_ref     term1;
QP_term_ref     term2;

void QP_put_db_reference(term, ref)
QP_term_ref     term1;
QP_db_reference ref;
```

Description

`QP_put_variable()`
: assigns to *term* a reference to a new unbound Prolog variable.

`QP_put_atom()`
: assigns to *term* a reference to the atom represented by *atom*. *atom* is assumed to be the canonical representation of a Prolog atom, either obtained from Prolog or returned by `QP_atom_from_string()`.

`QP_put_integer()`
: assigns to *term* a reference to *integer* tagged as a Prolog term.

`QP_put_float()`
: assigns to *term* a reference to the floating point number *float* tagged as a Prolog term.

`QP_put_functor()`
: assigns to *term* a reference to a new compound term whose functor is the atom represented by *name* and whose arity is *arity*. All the args of the compound term are unbound. This is similar to the Prolog builtin `functor/3` with its first argument unbound and its second and third argument bound.

`QP_put_list()`
: assigns to *term* a reference to a new list whose head and tail are both unbound.

`QP_put_nil()`
: assigns to *term* a reference to the atom `[]`.

`QP_put_term()`
: assigns to *term1* a reference to the term that *term2* references. Any reference to another term that *term1* contained is lost.

`QP_put_db_reference()`
: assigns to *term* a reference to the Prolog db_reference represented by *ref*. *ref* must have been a reference obtained through `QP_get_db_reference()`. Any reference to another term that *term1* contained is lost.

Examples

`flt_to_chars()` is a C function that converts a floating point number to a list of characters. Note the use of `QP_put_integer()`.

foo.pl

```
foreign(flt_to_chars, flt_to_chars(+float, -term)).
```

foo.c

```
#include <quintus/quintus.h>

void flt_to_chars(flt, chars)
double flt;
QP_term_ref chars;
{
   char buffer[28], *p;
   int len;
   QP_term_ref term_char = QP_new_term_ref();

   QP_put_nil(chars);
   sprintf( buffer , "%.17e" , flt );

   /* move to end of buffer */
   for (p=buffer, len=0; *p; p++, len++);

   while ( len-- )  {
      QP_put_integer(term_char, *--p);
      QP_cons_list(chars, term_char, chars);
   }
}
```

See Also

`QP_term_type()`, `QP_get_*()`, `QP_new_term_ref()`

19.3.49 QP_puts()

Synopsis

```
#include <quintus/quintus.h>

int QP_puts(s)
unsigned char *s;
```

Writes the NULL-terminated string pointed to by *s*, followed by a NEWLINE character, to the Prolog current output stream `QP_curout`.

`QP_puts()` is similar to the library function `puts(3S)`, however it operates on the Prolog current output stream rather than the standard output stream `stdout`.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_ferror()`, `QP_fopen()`, `QP_fread()`, `QP_printf()`, `QP_putc()`

Section 10.5 [fli-ios], page 433

19.3.50 `QP_query()`

Synopsis

```
#include <quintus/quintus.h>

int QP_query(pred_ref, arg1,...,arg255)
QP_pred_ref pred_ref;
```

Make a determinate query in a single C function call.

Description

The first argument passed to `QP_query()` is a reference to the Prolog predicate to be called. `QP_query()` accepts between 0 and 255 arguments after its first argument. Any arguments after the first represent parameters to be passed to and from the Prolog predicate. For the types of arguments that may be passed between C and Prolog predicates, see Section 10.4 [fli-ffp], page 413.

The foreign language interface will interpret arguments passed to the Prolog predicate according to the call specification given when the predicate was made callable. Hence, it is important that the arguments to be passed to and from the Prolog predicate should correspond with that call specification. In certain cases (passing Prolog atoms in canonical form) it is possible to detect inconsistencies between data supplied to `QP_query()` and the call specification, but for the most part this is impossible. Calls that are inconsistent with their call specifications will produce undefined results.

Only when the return value is `QP_SUCCESS` are the values in variables passed as outputs from Prolog valid. Otherwise, their contents are undefined.

Return Value

`QP_SUCCESS`
: query was made and a solution to the query was computed

`QP_FAILURE`
: query was made but no solution could be found

`QP_ERROR`
: either the query could not be made, or that an exception was signaled from Prolog but not caught.

See Also

`QP_cut_query()`, `QP_close_query()`, `QP_next_solution()`, `QP_open_query()`, `QP_pred()`, `QP_predicate()`, Section 10.4 [fli-ffp], page 413

19.3.51 QP_register_atom(), QP_unregister_atom()

Synopsis

```
#include <quintus/quintus.h>

int QP_register_atom(atom)
QP_atom atom;

int QP_unregister_atom(atom)
QP_atom atom;
```

Description

garbage_collect_atoms/0 is able to locate all atoms accessible from Prolog code, but cannot trace atoms that are only accessible from foreign code.

QP_register_atom() registers an atom as referenced from foreign code so that if garbage_collect_atoms/0 is called, the atom will not be reclaimed. QP_unregister_atom() unregisters the atom, so that if no other code (Prolog or foreign) refers to it, then it is a candidate for atom garbage collection.

These functions use a reference counting mechanism to keep track of atoms that have been registered. As a result, it is safe to combine different libraries that register and unregister atoms multiple times; the atoms will not be reclaimed until everyone has unregistered them.

Return Value

the current reference count of the atom or QP_ERROR if an error occurs.

Tips

Atoms do not normally need to be registered when calling foreign code. The only situation where this is needed is when the atom is being stored in a global or static data structure before returning to Prolog code, to be accessed subsequently by later calls to the foreign code.

See Also

garbage_collect_atoms/0

19.3.52 `QP_register_stream()`

Synopsis

```
#include <quintus/quintus.h>

int QP_register_stream(stream)
QP_stream *stream;
```

Register a created Prolog stream.

Arguments

stream pointer to a valid stream structure

Description

A customized Prolog stream must be registered via a call to `QP_register_stream()` before it can be accessed in Prolog code.

See Also

`stream_code/2`

Section 10.5.5.6 [fli-ios-cps-ire], page 450

19.3.53 QP_remove_*()

Synopsis

```
#include <quintus/quintus.h>

int QP_remove_input(id)
int  id;

int QP_remove_output(id)
int  id;

int QP_remove_exception(id)
int  id;

int QP_remove_timer(timerid)
int  timerid;
```

These C functions remove registered input/output or timing callback functions.

Description

`QP_remove_input()` removes any input callback functions registrations for the file descriptor *id*. Similarly, `QP_remove_output()` and `QP_remove_exception()` remove output and exception callbacks respectively.

`QP_remove_timer()` removes timer callback identified by *timerid*, which is the value returned by `QP_add_timer()` or `QP_add_absolute_timer()`.

Return Values

`QP_SUCCESS`
: If the callback is removed

`QP_ERROR` Otherwise

See Also

`QP_add_*()`, `QP_select()`, `QP_wait_input()`

19.3.54 QP_rewind()

Synopsis

```
#include <quintus/quintus.h>

int QP_rewind(stream)
QP_stream *stream;
```

`QP_rewind()` is similar to the library function `rewind(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fopen()`, `QP_ungetc()`

Section 10.5 [fli-ios], page 433

19.3.55 QP_seek()

Synopsis

```
#include <quintus/quintus.h>

int QP_seek(stream, offset, whence)
QP_stream *stream;
long int   offset;
int        whence;
```

Seeks to an arbitrary byte position on the *stream*.

Arguments

stream pointer to a valid stream structure.

offset the offset in bytes to seek relative to *whence* specified.

whence specifies where to start seeking. It is one of the following.

`QP_BEGINNING`
: seek from beginning of the file stream. The new position of the file stream is set to *offset* bytes.

`QP_CURRENT`
: seek from current position of the file stream. The new position of the file stream is set to its current location plus *offset*.

`QP_END` seek from end of the file stream. The new position of the file stream is set to the size of the file plus *offset*.

Description

The new position in bytes from the beginning of the file stream is stored in magic field of *stream*. It is *stream->magic.byteno* under UNIX.

If *stream* is an output stream permitting flushing output, the characters in the buffer of the stream is flushed through `QP_flush()` before seek is performed. If the *stream* does not permit flushing output and there are characters remaining in the output buffer, it is an error to seek. If *stream* is an input stream, the characters in the input buffer of the stream are discarded before seek is performed. The input buffer is empty when `QP_seek()` returns.

Return Value

`QP_SUCCESS`
: The function succeeds

`QP_ERROR` There is an error in function call, the error number is stored in both `QP_errno` and *stream->errno*.

Errors

`QP_E_INVAL`
whence is not one of `QP_BEGINNING`, `QP_CURRENT`, or `QP_END`.

`QP_E_CANT_SEEK`
Unknown error in the bottom layer of seek function of *stream*

Errors from `QP_flush()`

Errors from host operating system

Tips

`QP_seek(`*stream*`, 0L, QP_CURRENT)` sets the current position to the magic field of *stream*. It does not change the position of *stream*, but the side effect of flushing output and clearing buffer also takes place.

Comments

The seek type in *stream* must permits seeking by bytes, i.e. the seek_type field in *stream* is `QP_SEEK_BYTE`. So *stream* is created by defining a private stream and setting seek_type field to `QP_SEEK_BYTE`, opening a Prolog stream with seek(byte) option in `open/4`, or opening a binary stream through `QP_fopen()` or `QP_fdopen()`.

Examples

Get the current byte offset from beginning of the file stream.

```
if (QP_seek(stream, 0L, QP_CURRENT) != QP_SUCCESS)
        QP_perror(stream->errno, "QP_seek");
else
        location = stream->magic.byteno;
```

See Also

`QP_getpos()`, `QP_setpos()`, `QP_rewind()`, `QP_flush()`.

Section 10.5 [fli-ios], page 433

19.3.56 QP_select()

Synopsis

```
#include <quintus/quintus.h>

int QP_select(wid,read_fds,write_fds,except_fds,timeo)
int    wid;
fd_set *read_fds;
fd_set *write_fds;
fd_set *except_fds;
struct timeval *timeo;
```

Wait until I/O is ready on a file descriptor or until a timeout occurs

Description

This is a more general version of `QP_wait_input()`, which is compatible with the system call `select(2)`. It waits for any of *read_fds* to be ready for reading, or *write_fds* to be ready for writing, or *except_fds* to have an exceptional command pending, or for the timeout period *timeo* to elapse, whichever comes first. Callbacks on other descriptors are handled while waiting. However, no callbacks on any of the desciptors specified in *read_fds*, *write_fds*, and *except_fds* are called, rather `QP_select()` returns immediately.

Return Values

the number of descriptors in the bit mask, `QP_SUCCESS` if a timeout occurred or `QP_ERROR` if an error occurred.

Windows Caveats

Under Windows, there is a special `SOCKET` data type, which is different from file descriptors. The arguments to `QP_select()` are sets of such sockets, not file descriptors, exactly like the WinSock `select()` function.

`QP_select()` is not interruptible by `^C`. For this reason, calling `QP_select()` with infinite timeout is probably a bad idea. If called with infinite timeout and if there are no open sockets, then `QP_select()` will return immediately, indicating a timeout.

With a finite timeout value but no open sockets, `QP_select()` will use the Win32 `Sleep()` function to perform a (non-interruptible) sleep.

See Also

`QP_wait_input()`, `QP_add_*()`

19.3.57 `QP_setinput()`

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QP_setinput(stream)
QP_stream *stream;
```

Set the Prolog current input stream to a specified value *stream*.

Arguments

stream pointer to a valid input stream

Description

This function sets an input stream *stream* to be the current Prolog input stream and returns the previous current Prolog input stream.

See Also

`set_input/1`, `QP_setoutput()`

Section 10.5 [fli-ios], page 433

19.3.58 `QP_setoutput()`

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QP_setoutput(stream)
QP_stream *stream;
```

Set the Prolog current output stream to a specified value *stream*.

Arguments

stream pointer to a valid Prolog output stream

Description

This function sets the current Prolog outpout stream to *stream* and returns the previous current Prolog output stream before the operation.

See Also

`set_ouptut/1`, `QP_setinput()`

Section 10.5 [fli-ios], page 433

19.3.59 `QP_setpos()`

Synopsis

```
#include <quintus/quintus.h>

int QP_setpos(stream, pos)
QP_stream   *stream;
QP_position *pos;
```

Reset a specified Prolog stream back to a previous read/written position.

Arguments

stream	pointer to a valid stream structure
pos	pointer to a `QP_position` structure.

Description

Upon successful return of this function call, the *stream* is repositioned to the value specified in the `magic` member pointed to by *pos*. The character, line and line position counts of *stream* are also reset to the values specified in `char_count`, `line_count` and `line_position` members in *pos*.

The specified *stream* must have the permission to seek back to a previous read/written position. Typically, the value of *pos* is obtained through a previous `QP_getpos()` call.

See Also

`QP_getpos()`, `QP_seek()`, `stream_position/[2,3]` `QP_char_count()`, `QP_line_count()`, `QP_line_position()`

Section 10.5 [fli-ios], page 433

19.3.60 `QP_skipline()`

Synopsis

```
#include <quintus/quintus.h>

int QP_skipline()
```

Skip the current input record of the current Prolog input stream. `QP_skipline()` is a macro.

Description

`QP_skipline()` is equivalent to `QP_skipln(QP_curin)`.

See Also

`QP_skipln()`

Section 10.5 [fli-ios], page 433

19.3.61 `QP_skipln()`

Synopsis

```
#include <quintus/quintus.h>

int QP_skipln(stream)
QP_stream *stream;
```

Skip the current input record of a Prolog input stream. `QP_skipln()` is a macro.

Arguments

stream pointer to a valid Prolog input stream

See Also

`QP_skipline()`, `QP_fskipln()`

Section 10.5 [fli-ios], page 433

19.3.62 QP_string_from_atom(), QP_padded_string_from_atom()

Synopsis

```
#include <quintus/quintus.h>

char *QP_string_from_atom(atom)
QP_atom atom;

int QP_padded_string_from_atom(p_atom, p_string, p_length)
QP_atom *p_atom;
char    *p_string;
int     *p_length;
```

Description

`QP_string_from_atom()` returns a pointer to a string representing *atom*. This string should not be overwritten by the foreign function.

`QP_padded_string_from_atom()` is useful for Pascal and FORTRAN and can be used for any language that has a C-compatible calling convention for passing integers and pointers (on the users platform). This is true for many Pascal and FORTRAN compilers running under UNIX.

p_atom and *p_length* can be seen as integers passed by reference. Fills in the character array of length *length* with the string representation of *atom*. The string is truncated or blank-padded to *length*. The length of the atom (not *length*) is returned as the function value. In the above description *atom* refers to the argument passed by reference corresponding to the declared argument *p_atom* and similarly for *p_string* and *p_length*.

Examples

`rev_atom()` is a C function that takes an atom and returns an atom whose string representation is the reverse of the string representation of the atom passed in.

foo.pl

```
foreign(rev_atom, c, rev_atom(+atom, [-atom])).
```

foo.c

```
QP_atom rev_atom(atom)
QP_atom atom;
{
    char * string[MAX_ATOM_LEN];

    strcpy(string, QP_string_from_atom(atom));
    reverse(string); /* reverses string in place */
    return QP_atom_from_string(string);
}
```

```
| ?- rev_atom(draw, X).

X = ward

yes
| ?-
```

See Also

`QP_atom_from_string()`, `QP_atom_from_padded_string()`

Section 10.3.7 [fli-p2f-atm], page 389

19.3.63 QP_tab()

Synopsis

```
#include <quintus/quintus.h>

int QP_tab(stream, count, c)
QP_stream *stream;
int        count;
int        c;
```

Output *count* number of the character *c* on Prolog output stream *stream*.

Arguments

stream	pointer to a valid Prolog output stream
count	how many characters of *c* to be output
c	character to be written out

Examples

`QP_tab(QP_curout, 5, ' ')` puts 5 blank characters to the current output stream.

See Also

`QP_tabto()`

19.3.64 QP_tabto()

Synopsis

```
#include <quintus/quintus.h>

int QP_tabto(stream, line_pos, c)
QP_stream *stream;
int        line_pos;
int        c;
```

Pad the character *c* up to the specified line position *line_pos* on Prolog output stream *stream*.

Arguments

stream	pointer to a valid Prolog output stream
line_pos	line position to be padded to.
c	character used for padding

See Also

QP_tab()

19.3.65 QP_term_type()

Synopsis

```
#include <quintus/quintus.h>

int QP_term_type(term)
QP_term_ref term;
```

Tests the type of Prolog terms in C.

Description

`QP_term_type()` returns the type of a Prolog term that is passed to it as argument. The returned value is one of the following constants defined in the file '`<quintus/quintus.h>`': `QP_VARIABLE`, `QP_INTEGER`, `QP_ATOM`, `QP_FLOAT` or `QP_COMPOUND`.

Examples

`print_type()` is a C function that prints the type of the Prolog term passed to it.

foo.pl

```
foreign(print_type, c, print_type(+term)).
```

foo.c

```
#include <quintus/quintus.h>

void print_type(term)
QP_term_ref term;
{
    switch (QP_term_type(term)) {
    case QP_VARIABLE:
        QP_printf("Term is a variable\n");
        break;
    case QP_INTEGER:
        QP_printf("Term is an integer\n");
        break;
    case QP_FLOAT:
        QP_printf("Term is a float\n");
        break;
    case QP_ATOM:
        QP_printf("Term is an atom\n");
        break;
    case QP_COMPOUND:
        if (QP_is_list(term)) {
            QP_printf("Term is a list\n");
        } else {
            QP_printf("Term is a compound term\n");
        }
        break;
    }
}
```

See Also

`QP_is_*()`, `QP_get_*()`, `QP_put_*()`, `QP_new_term_ref()`

19.3.66 QP_toplevel()

Synopsis

```
#include <quintus/quintus.h>

int QP_toplevel()
```

Description

Invokes Prolog's default top level read-prove loop.

For runtime systems, `QP_toplevel()` immediately transfers control to the definition of the Prolog predicate `runtime_entry/1`.

`QP_toplevel()` takes no arguments. `QP_query()` and related predicates should be used for calling specific Prolog predicates.

One of the effects of calling this function is that the default signal handling for Prolog is enabled. Upon return, the old signal handlers are restored.

The built-in predicate `break/0` calls this function. Nested calls to this function are equivalent to calling the Prolog predicate `break/0`.

This function returns when an end-of-file character is read.

Return Value

`QP_SUCCESS`
`QP_FAILURE`
`QP_ERROR`

See Also

`QP_initialize()`, `break/0`.

19.3.67 `QP_trimcore()`

Synopsis

```
#include <quintus/quintus.h>

int QP_trimcore()
```

`QP_trimcore()` is the C equivalent of Prolog's `trimcore/0`.

Description

`trimcore/0` is usually called by Prolog when you return to top level after each query. But if you have an embedded application without Prolog's top level or a runtime system then you can call `QP_trimcore()` explicitly to ask Prolog to consolidate all its free memory and free as much as possible back to the operating system.

Like `trimcore/0` it should be used judiciously, as overuse can result in unnecessary time being spent in memory expansion and contraction. However, it can be used when Prolog is to be dormant for a period, or as much free memory as possible is desired.

See Also

`trimcore/0`

Section 10.2 [fli-emb], page 365

19.3.68 QP_ungetc()

Synopsis

```
#include <quintus/quintus.h>

int QP_ungetc(c, stream)
int        c;
QP_stream *stream;
```

Pushes the character *c* back onto Prolog input stream *stream*.

`QP_ungetc()` is similar to the library function `ungetc(3S)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_fseek()`, `QP_getc()`

Section 10.5 [fli-ios], page 433

19.3.69 QP_unify()

Synopsis

```
#include <quintus/quintus.h>

int QP_unify(term1, term2)
QP_term_ref term1;
QP_term_ref term2;
```

Unify two Prolog terms.

Description

`QP_unify()` unifies the two terms referenced by *term1* and *term2*. Both should be initialized references. If the unification succeeds, the function returns `QP_SUCCESS`, otherwise it returns `QP_FAILURE`. If the unification results in any bindings then the bindings are trailed. If Prolog backtracks over the foreign function that called `QP_unify()` then the bindings are undone.

Examples

`c_unify(term1, term2)` is equivalent to the usual Prolog builtin `=/2`, but it returns a third argument, which is an integer indicating success or failure.

foo.pl

```
foreign(c_unify, c, c_unify(+term, +term, [-integer])).
```

foo.c

```
#include <quintus/quintus.h>

long int c_unify(t1, t2);
QP_term_ref t1, t2;
{
    return QP_unify(t1, t2);
}
```

See Also

`=/2`

19.3.70 QP_vfprintf()

Synopsis

```
#include <quintus/quintus.h>

int QP_vfprintf(stream, format, ap)
QP_stream *stream;
char      *format;
va_list    ap;
```

Places output onto the Prolog output stream *stream*.

`QP_vfprintf()` is also similar to the library function `vfprintf(3V)`, however the return values differ and *stream* is a Prolog stream rather than a `stdio` stream. It also resembles `QP_fprintf()` except that rather than being called with a variable number of arguments, it is called with an argument list as defined by `varargs(3)`.

Return Value

`QP_SUCCESS`
: If function succeeds

`QP_ERROR` Otherwise

See Also

`QP_printf()`, `QP_fprintf()`

Section 10.5 [fli-ios], page 433

19.3.71 QP_wait_input()

Synopsis

```
#include <quintus/quintus.h>

int QP_wait_input(id,timeout)
int id;
int *timeout;
```

Wait until I/O is ready on a file descriptor or until a timeout occurs

Description

`QP_wait_input()` waits until input is ready on file descriptor *id* or until *timeout* milliseconds pass. If timeout is `QP_NO_TIMEOUT`, it waits indefinitely for input to arrive on *id*. While waiting, `QP_wait_input()` makes sure that registered callbacks are called when input is ready on other file descriptors. However, no callback will be called when input is ready on *id* even if one is registered, rather `QP_wait_input()` returns immediately.

Return Values

`QP_SUCCESS`
: if input arrived on *id*

`QP_FAILURE`
: if a timeout occurred

`QP_ERROR` if an error occurs

See Also

`QP_select()`, `QP_add_*()`

19.3.72 QU_alloc_mem(), QU_alloc_init_mem(), QU_free_mem()

Synopsis

```
char *QU_alloc_mem(size, alignment, actualsize)
unsigned int  size;
unsigned int  alignment;
unsigned int *actualsize;
```

The primitive function that Prolog calls to get memory

```
char *QU_alloc_init_mem(size, alignment, actualsize)
unsigned int  size;
unsigned int  alignment;
unsigned int *actualsize;
```

Called when Prolog needs memory for the first time

```
int QU_free_mem(mem, size)
char          *mem;
unsigned int  size;
```

The primitive function called when Prolog wants to free memory.

Description

These are the primitive functions on which the all of Prolog's sophisticated memory management is built. If Prolog is to be embedded into an application that would like to provide its own memory management routines then the user can redefine these functions and link it with the Prolog system.

`QU_alloc_mem()` must allocate a piece of memory that has at least *size* bytes aligned at *alignment* in it and return a pointer to it. The memory returned itself need not be aligned at *alignment*. The *alignment* argument is guaranteed to be a power of 2. The actual size of the piece of memory returned should be stored in **actualsize*. Prolog uses all the memory given to it; there is no memory wasted when *actualsize* is greater than *size*. `QU_alloc_mem()` should return 0 if it cannot allocate any more memory.

`QU_alloc_init_mem()` is a special case of `QU_alloc_mem()`. It can do whatever initialization that this layer of memory management wants to do.

`QU_free_mem()` is called with a pointer to the memory that is to be freed and the size of the memory to be freed. If `QU_free_mem()` was not able to free this piece of memory then this function should return 0. In this case Prolog will continue using the memory as if it was not freed.

The default definitions for these functions look at the environment variables `PROLOGINITSIZE`, `PROLOGINCSIZE`, `PROLOGKEEPSIZE` and `PROLOGMAXSIZE`. These environment variables are useful to customize the default memory manager. If users redefine this layer of memory management they can choose to ignore these environment variables.

Examples

Here is a simple example of the embeddable layer of memory management based on `malloc(3)` and `free(3)`. This example is far from ideal because you might be over-allocating memory to ensure the required size of aligned memory, but demonstrates the capability. The C file '`mem.c`' defines `QU_alloc_mem()`, `QU_alloc_init_mem()` and `QU_free_mem()`.

mem.c

```
unsigned int IncSize = 0x100000;        /* 1M */
unsigned int InitSize = 0x100000;       /* 1M */
unsigned int MaxSize = 0x1000000;       /* 16 M */
unsigned int KeepSize = 0x100000;       /* 1M */
unsigned int MemTotal;

char * QU_alloc_mem(size, align, actualsize)
    unsigned int size;  /* in bytes */
    unsigned int align;         /* power of 2 */
    unsigned int *actualsize;
    {
        char *mem, *malloc();

        size = size + align;
        if (size <= IncSize) size = IncSize;
        if ((size + MemTotal) > MaxSize) return 0;
        mem = malloc(size);
        *actualsize = size;
        MemTotal += (mem == 0 ? 0 : size);
        return mem;
    }

char * QU_alloc_init_mem(size, align, actualsize)
    unsigned int size;  /* in bytes */
    unsigned int align;         /* power of 2 */
    unsigned int *actualsize;
    {
        char *mem, *str, *malloc(), *getenv();

        str = getenv("PROLOGINCSIZE");
        if (str) sscanf(str, "%u", &IncSize);
        str = getenv("PROLOGINITSIZE");
        if (str) sscanf(str, "%u", &InitSize);
        str = getenv("PROLOGMAXSIZE");
        if (str) sscanf(str, "%u", &MaxSize);
        str = getenv("PROLOGKEEPSIZE");
        if (str) sscanf(str, "%u", &KeepSize);

        MemTotal = 0;
        return QU_alloc_mem(size, align, actualsize);
    }

int QU_free_mem(mem, size)
    char * mem;
    unsigned int size;
    {
        if ((MemTotal - size) < KeepSize) return 0;
        free(mem);
        MemTotal = MemTotal - size;
        return 1;
    }
```

To build a Prolog development system based on the functions defined in ‘`mem.c`’:

```
% cc -c mem.c
% qld -D mem.o -o prolog_on_my_mm_fns
```

See Also

Section 10.2 [fli-emb], page 365

19.3.73 `QU_fdopen()` user-redefinable

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QU_fdopen(stream_option, system_option, er-
ror_number, file_des)
QP_stream *stream_option;
char      *system_option;
int       *error_number;
int        file_des;
```

The embedding function for creating a stream opened through `open/[3,4]` or `QP_fopen()`. Creates a stream and returns the `QP_stream` pointer for that stream.

Description

`QU_fdopen()` is similar to `QU_open()` except that the file stream is already opened and the opened file descriptor is passed through the parameter *file_des*.

Examples

See the example in the manual page for `QU_open()`.

See Also

`QU_open()`, Section 10.5 [fli-ios], page 433

19.3.74 `QU_free_mem()` user-redefinable

Described in reference page for `QU_alloc_mem()`.

19.3.75 `QU_initio()` user-redefinable

Synopsis

```
#include <quintus/quintus.h>

int QU_initio(user_input, user_output, user_error, act_astty, error_num)
QP_stream **user_input;
QP_stream **user_output;
QP_stream **user_error;
int         act_astty;
int        *error_num;
```

Initializes Prolog input/output system. Returns `QP_SUCCESS` upon success and `QP_ERROR` upon failure.

Description

The three Prolog initial stream are created in `QU_initio()`. The Prolog standard input stream is returned through *user_input*, the standard output stream is returned through *user_output*, and the standard error stream is returned through *user_error*. The created streams are accessed in the Prolog system as `user_input` (`QP_stdin`), `user_output` (`QP_stdout`), and `user_error` (`QP_stderr`).

If *act_astty* is non-zero, the Prolog system requests `QU_initio()` to initialize the three initial streams as tty streams even if they are not really connected to a tty. One example of such a request is that Prolog is running under remote shell.

The parameter *error_num* stores the error code if `QU_initio()` returns `QP_ERROR`. The error code can be any of the host operating system error numbers, QP error numbers or a user-defined error number.

Tip

The process required to create these three initial streams is similar to that of implementing a customized Prolog stream. (see Section 10.5.5 [fli-ios-cps], page 445). However, these three initial streams should not be registered. Calling `QP_register_stream()` to register any of the three streams created by `QU_initio()` may cause an error when Prolog starts up.

Examples

The following is the source code for an implementation of `QU_initio()` function in C language.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <quintus/quintus.h>

#define TTY_BUFSIZ              128
#define MAX_FIFO_BUFSIZ         4096

extern  QP_stream    *QU_fdopen();

/*
 *  This I/O initialization function only handles three possible
 *  types of file, a tty file , a pipe and an ordinary file
 */
int QU_initio(user_input, user_output, user_error, act_astty,
                        error_num)
    QP_stream   **user_input, **user_output, **user_error;
    int         act_astty, *error_num;
    {
        int             fd, is_tty;
        struct  stat    statbuf;
        QP_stream       option, *streams[3], *prompt_stream;
        extern char     *ttyname();

        for (fd=2; fd >= 0 ; --fd) {
            is_tty = isatty(fd);
            QU_stream_param((is_tty) ? "/dev/tty" : "",
                        (fd) ? QP_WRITE : QP_READ, &option);
            if (is_tty || act_astty) {
                /* make sure other parameters are right  */
                option.format     = QP_DELIM_TTY;
                option.max_reclen = TTY_BUFSIZ;
                option.seek_type  = QP_SEEK_ERROR;
                if (fd == 0)
                    option.peof_act = QP_PASTEOF_RESET;
            } else  {
                if (fstat(fd, &statbuf) < 0)
                    return QP_ERROR;
                if ((statbuf.st_mode & S_IFIFO) == S_IFIFO) {
                    option.max_reclen = MAX_FIFO_BUFSIZ;
                    option.seek_type  = QP_SEEK_ERROR;
                } else
                    option.max_reclen = statbuf.st_blksize;
            }
```

```
            option.mode = (fd) ? QP_WRITE : QP_READ;
            if ((streams[fd]=QU_fdopen(&option,"",error_num,fd))
                                ==QP_NULL_STREAM)
                return QP_ERROR;
            if (is_tty) {
                char        *tty_id;
                if (! (tty_id = ttyname(fd)) )
                    tty_id = "/PROLOG DEFAULT TTYS";
                (void) QP_add_tty(streams[fd], tty_id);
            } else if (act_astty)
                (void) QP_add_tty(streams[fd],
                                  "/PROLOG INITAIL STREAMS");
        }
        (streams[0])->filename="USER$INPUT";
        *user_input  = streams[0];
        (streams[1])->filename="USER$OUTPUT";
        *user_output = streams[1];
        (streams[2])->filename="USER$ERROR";
        *user_error  = streams[2];
        if ((streams[0])->format == QP_DELIM_TTY
                && (streams[1])->format != QP_DELIM_TTY
                && (streams[2])->format != QP_DELIM_TTY) {
            char *tty_id;
            /* create an output stream for prompt */
            QU_stream_param(isatty(0) ? "/dev/tty" : "", QP_WRITE,
                                            &option);
            option.format = QP_DELIM_TTY;
            option.max_reclen = TTY_BUFSIZ;
            option.seek_type = QP_SEEK_ERROR;
            if ((prompt_stream = QU_fdopen(&option, "", error_num,
                               0)) == QP_NULL_STREAM)
                return QP_ERROR;
            (void) QP_register_stream(prompt_stream);
            if (! (tty_id = ttyname(0)) )
                tty_id = "/PROLOG DEFAULT TTYS";
            (void) QP_add_tty(prompt_stream, tty_id);
        }
        return QP_SUCCESS;
    }
```

19.3.76 `QU_open()` user-redefinable

Synopsis

```
#include <quintus/quintus.h>

QP_stream *QU_open(stream_option, system_option, error_number)
QP_stream *stream_option;
char      *system_option;
int       *error_number;
```

The embedding function for creating a stream opened through `open/[3,4]` or `QP_fopen()`.

Creates a stream and returns the `QP_stream` pointer for that stream.

Description

stream_option specifies the options for the stream to be created.

system_option specifies the system-dependent stream option specified in the option of `system(`*option*`)` in `open/4`. If there is no system-dependent option, the `system_option` field is the empty string, `""`.

The functionality of `QU_open()` is illustrated in these steps `open/4` uses to create a Prolog stream:

1. Set up the default stream options through `QU_stream_param()`.
2. Changes the stream options obtained in the previous step based on the options specified in `open/4`.
3. Calls `QU_open()` with the stream options resulted in the previous two steps to create the stream.
4. Register the stream created by `QU_open()` through `QP_register_stream()`.
5. Converts the stream pointer returned by `QU_open()` to the Prolog representation of the stream through `stream_code/2`.

The process required to implement `QU_open()` is similar to implementing a customized Prolog stream. (see Section 10.5.5 [fli-ios-cps], page 445). There are only a few differences.

1. The stream options are specified as a parameter in a call to `QU_open()`, so `QU_open()` does not need to call `QU_stream_param()`.
2. `QU_open()` may need to parse *system_option* string to recognize the option specified in that string. Notice that the core Prolog system itself does not make use of any information specified in *system_option*. It simply takes the string specified in `open/4` and passes it to `QU_open()` without any alteration.

3. `QU_open()` does not need to register the stream it creates. The caller of `QU_open()` will register it.

Implementation of `QU_fdopen()` is similar to `QU_open()` except that the file stream is already opened and the opened file descriptor is passed through the parameter *file_des*.

Return Value

If any error occurs during the creation of the stream, `QP_NULL_STREAM` is returned and the error code for the error condition is set in the third parameter, *error_number*. The error code could be any of the host operating system error numbers, QP error numbers or a user-defined error number. `QU_fdopen()` requires an addition parameter, *file_des*, which is the opened file descriptor for the stream to be created.

Examples

The following is the source code for an implementation of `QU_open()` function and `QU_fdopen()` function in C.

```
#include <fcntl.h>
#include <errno.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>

#ifndef L_SET
#define L_SET   0
#endif
#ifndef L_INCR
#define L_INCR  1
#endif
#ifndef L_XTND
#define L_XTND  2
#endif

#include <quintus/quintus.h>

extern  char *ttyname();
```

```
QP_stream *
QU_fdopen(option, sys_option, error_num, file_des)
    QP_stream   *option;
    char        *sys_option;
    int         *error_num, file_des;
    {
        QP_stream       *stream;
        extern long     lseek();
        extern QP_stream *QU_tty_open(), *QU_text_open(),
                         *QU_raw_open();

        if (sys_option && *sys_option != '\0') {
            *error_num = QP_E_SYS_OPTION;
            return QP_NULL_STREAM;
        }
        *error_num = 0;
        if (option->format == QP_FMT_UNKNOWN) {
            if (option->line_border==QP_NOLB &&
                option->trim==0)
                option->format = QP_VAR_LEN;
                                /* binary file */
            else
                option->format = QP_DELIM_LF;
        }
```

```
switch (option->mode) {
case QP_READ:
case QP_WRITE:
    option->magic.byteno = 0;
    if (option->seek_type != QP_SEEK_ERROR) {
        struct stat     statbuf;

        if (fstat(file_des, &statbuf) == 0 &&
           (statbuf.st_mode & S_IFMT) == S_IFREG)
            if ((option->magic.byteno =
                 lseek(file_des,0L,L_INCR))==-1) {
                *error_num = errno;
                return QP_NULL_STREAM;
            }
    }
    break;
case QP_APPEND:
    if ((option->magic.byteno =
                 lseek(file_des,0L,L_XTND)) == -1)
    {
         *error_num = errno;
        return QP_NULL_STREAM;
    }
    break;
default:
    *error_num = QP_E_BAD_MODE;
    return QP_NULL_STREAM;
}
```

```
    switch (option->format) {
    case QP_DELIM_TTY:
        stream = QU_tty_open(option, error_num,
                             file_des);
        break;
    case QP_DELIM_LF:
        stream = QU_text_open(option, error_num,
                              file_des);
        break;
    case QP_VAR_LEN:
        stream = QU_raw_open(option, error_num,
                             file_des);
        break;
    default:
        *error_num = QP_E_BAD_FORMAT;
        return QP_NULL_STREAM;
    }
    return stream;
}
```

```
QP_stream *
QU_open(option, sys_option, error_num)
    QP_stream   *option;
    char        *sys_option;  /* not useful in this
                                 version */
    int         *error_num;
    {
        QP_stream       *stream;
        int             fd;
        char            *filename;
        struct stat     statbuf;

        *error_num = 0;
        if (! (filename = option->filename)) {
            *error_num = QP_E_FILENAME;
            return QP_NULL_STREAM;
        }
        switch (option->mode) {
        case QP_READ:
            fd = open(filename, O_RDONLY, 0000);
            break;
        case QP_WRITE:
            fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC,
                      0666);
            break;
        case QP_APPEND:
            fd = open(filename, O_WRONLY|O_CREAT, 0666);
            break;
        default:
            *error_num = QP_E_BAD_MODE;
            return QP_NULL_STREAM;
        }
```

```
        if (fstat(fd, &statbuf) == 0) {
            if ((statbuf.st_mode & S_IFMT) == S_IFDIR) {
                (void) close(fd);
                *error_num = QP_E_DIRECTORY;
                return QP_NULL_STREAM;
            }
        }
        if (fd < 0) {
            *error_num = errno;
            return QP_NULL_STREAM;
        }
        if (option->format == QP_DELIM_TTY &&
                              !(isatty(fd)) ) {
            *error_num = QP_E_BAD_FORMAT;
            (void) close(fd);
            return QP_NULL_STREAM;
        }
        if ((stream=QU_fdopen(option, sys_option,
                              error_num, fd))
                         == QP_NULL_STREAM) {
            (void) close(fd);
            return QP_NULL_STREAM;
        }
        if (stream->format == QP_DELIM_TTY) {
            char        *tty_id;
            if (! (tty_id = ttyname(fd)) )
                tty_id = "/PROLOG DEFAULT TTYS";
            (void) QP_add_tty(stream, tty_id);
        }
        return stream;
    }
```

The following is the source code for the implementation of `QU_tty_open()` in C. `QU_tty_open()` is called in the `QU_fdopen()` source code listed above. (`QU_text_open()` and `QU_raw_open()` are also called in `QU_fdopen()`. The source code for these two functions is not listed here, but they are shipped with Quintus Prolog.)

```
#include <fcntl.h>
#include <errno.h>

#include <quintus/quintus.h>

extern  char *QP_malloc();

#define Min_Buffer_Size         4

struct TtyStream
    {
        QP_stream qpinfo;
        int fd;
        unsigned char buffer[Min_Buffer_Size];
    };

#define CoerceTtyStream(x) ((struct TtyStream *)(x))

static int
tty_read(stream, bufptr, sizeptr)
    QP_stream           *stream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        int n;
        extern int errno;
        register struct TtyStream
                 *u = CoerceTtyStream(stream);

        n = read(u->fd, (char*)u->buffer,
                        (int)u->qpinfo.max_reclen);
        if (n > 0) {
            *bufptr  = u->buffer;
            *sizeptr = n;
            if (u->buffer[n-1] == '\n')
                return QP_FULL;
            else
                return QP_PART;
        } else if (n == 0) {
            *sizeptr = 0;
            return QP_EOF;
        } else {
            u->qpinfo.errno = errno;
            return QP_ERROR;
        }
    }
```

```
static int
tty_write(stream, bufptr, sizeptr)
    QP_stream           *stream;
    unsigned char       **bufptr;
    size_t              *sizeptr;
    {
        struct TtyStream  *u = CoerceTtyStream(stream);
        int                n, len=(int) *sizeptr;
        char              *buf = (char *) *bufptr;

        if (len==0) {   /* be sure to set *sizeptr
                           and *bufptr */
            *sizeptr = u->qpinfo.max_reclen;
            *bufptr = u->buffer;
            return QP_SUCCESS;
        }
        while ((n = write(u->fd, buf, len)) > 0 && n
                < len) {
            buf += n;
            len -= n;
        }
        if (n >= 0) {
            *sizeptr = u->qpinfo.max_reclen;
            *bufptr = u->buffer;
            return QP_SUCCESS;
        } else {
            u->qpinfo.errno = errno;
            return QP_ERROR;
        }
    }

static int
tty_close(stream)
    QP_stream   *stream;
    {
        struct TtyStream *u = CoerceTtyStream(stream);
        int fd = u->fd;

        QP_free(stream);
        if (close(fd) < 0)
            return QP_ERROR;
        return QP_SUCCESS;
    }
```

```
QP_stream *
QU_tty_open(option, error_num, fd)
    register        QP_stream       *option;
    int         *error_num, fd;
    {
        struct  TtyStream       *stream;

        if (option->seek_type != QP_SEEK_ERROR) {
            *error_num = QP_E_SEEK_TYPE;
            return QP_NULL_STREAM;
        }
        stream = (struct  TtyStream *)
                  QP_malloc(sizeof(*stream) +
            ((option->max_reclen <= Min_Buffer_Size) ? 0
                : option->max_reclen - Min_Buffer_Size) );
        stream->qpinfo = *option;
        QP_prepare_stream(&stream->qpinfo, stream->buffer);
        stream->fd = fd;
        stream->qpinfo.close = tty_close;
        if (option->mode != QP_READ)  {
            stream->qpinfo.write =
            stream->qpinfo.flush = tty_write;
        } else
            stream->qpinfo.read  = tty_read;
        return (QP_stream *) stream;
    }
```

See Also

`open/4`, `QU_fdopen()`

Section 10.5 [fli-ios], page 433

19.3.77 QU_stream_param() user-redefinable

Synopsis

```
#include <quintus/quintus.h>

void QU_stream_param(filename, mode, format, option)
char          *filename;
int            mode;
unsigned char  format;
QP_stream     *option;
```

Embedding function, which sets up default values of user-accessible fields in a Prolog stream structure.

Arguments

filename name of the file to be opened. If the stream does not have a filename, the *filename* is the empty string, `""`. If the *filename* is '`/dev/tty`', the caller requests `QU_stream_param()` to set the field values as a tty stream.

mode

- `QP_READ` for an input stream.
- `QP_WRITE` for an output stream.
- `QP_APPEND` for an output stream opened in append mode.

format format of the stream, which could have one of the following values:

- `QP_DELIM_LF`
-
-
-
-
- `QP_VAR_LEN`
- `QP_DELIM_TTY`
- `QP_FMT_UNKNOWN`

option pointer to a `QP_stream` structure whose fields are to be set up.

Description

The format is used in assisting `QU_stream_param()` to determine the best values for the other member fields of the `QP_stream` structure.

All of the fields listed below of the `QP_stream` structure, described in Section 10.5.3 [fli-ios-sst], page 437, are set by `QU_stream_param()`. Under UNIX, they are given the indicated values:

field	value
`filename`	*filename* argument to `QU_stream_param()`
`mode`	*mode* argument to `QU_stream_param()`
`format`	*format* argument to `QU_stream_param()`
`max_reclen`	8192 bytes for a file, 256 bytes for a tty
`line_border`	`QP_LF`
`file_border`	`QP_EOF`
`peof_act`	`QP_PASTEOF_ERROR`
`prompt`	`""` (empty string)
`trim`	0
`seek_type`	`QP_SEEK_PREVIOUS` for a file, `QP_SEEK_ERROR` for a tty
`flush_type`	`QP_FLUSH_FLUSH`
`overflow`	`QP_OV_FLUSH`
`errno`	0
`magic`	(byteno) 0
`read`	`bad_read` — a function that returns `QP_ERROR`
`write`	`bad_write` — a function that returns `QP_ERROR`
`flush`	`bad_flush` — a function that returns `QP_ERROR`
`seek`	`bad_seek` — a function that returns `QP_ERROR`
`close`	`bad_close` — a function that returns `QP_ERROR`

20 Command Reference Pages

20.1 Command Line Utilities

The reference pages for the Quintus supplied command line utilities follow, in alphabetical order.

- `prolog(1)`
- `qcon(1)`
- `qgetpath(1)`
- `qld(1)`
- `qnm(1)`
- `qpc(1)`
- `qplm(1)`
- `qsetpath(1)`
- `qui(1)`

Once these files are installed in `library('q3.5')`, you can access them on-line by typing, for example,

```
% man qpc
```

Refer to Section 1.3 [int-dir], page 11 for the location of these reference pages in the Quintus directory hierarchy.

20.1.1 `prolog` — Quintus Prolog Development System

Synopsis

```
prolog [ +f ] [ +l file ] [ +L file ]
     [ +p [path-name] ] [ +P [path-name] ] [+tty ]
     [ +z user's-arguments | + [emacs-arguments] ]

qpwin [ +f ] [ +l file ] [ +L file ]
     [ +p [path-name] ] [ +P [path-name] ] [+tty ]
     [ +z user's-arguments | + [emacs-arguments] ]
```

Description

`prolog` is the command to invoke the Quintus Prolog Development System. The prompt '`| ?-`' indicates that the execution of Quintus Prolog Development System is in top-level mode. In this mode, Prolog queries may be issued and executed interactively. A program written in the Prolog programming language can be compiled during the execution of `prolog` as additional information for subsequent execution. The Prolog command `halt/[0,1]` is used to exit from prolog; under the GNU Emacs editor, exit is `^x^c`.

Under Windows, `prolog.exe` is a console-based program that can run in a command prompt window, whereas `qpwin.exe` runs in its own window and directs the Prolog standard streams to that window. `qpwin.exe` is a "windowed" executable; see Section 9.1.3.1 [sap-srs-qld-iin], page 344.

Compiled programs can be saved into a file as a saved-state. The file can be restored either through Quintus Prolog built-in restore command or issued as a command to the command interpreter. The UNIX command `head -1` *`saved-state`* displays how the *saved-state* file is restored in the latter case. The *saved-state* file can also be passed to `qld(1)` to be linked into an executable program.

`prolog` depends on preset paths to locate the license files, Prolog libraries, system-dependent foreign object files, and certain executables. There are three main paths, *quintus-directory*, *runtime-directory* and *host-type*, which are set during the installation of the Quintus Prolog Development System. The paths can be checked with `qgetpath(1)`, and changed with `qsetpath(1)`. Upon startup, `prolog` checks the files in '*`quintus-directory`*`/license`*`quintus-version`*' for authorization of execution. Under UNIX, `prolog` locates its libraries under the directory '*`quintus-directory`*`/generic/qplib`*`quintus-version`*' where *quintus-version* is the Quintus Prolog release number built into the executables. Under Windows, `prolog` locates its libraries under the directory '*`quintus-directory`*`/src`'. `prolog` searches the system dependent foreign object files in directories based on *host-type*.

Quintus Prolog Development System can be linked with other Quintus products, and user's application programs. For more detail, see `qld(1)`, `qpc(1)`, and the add-on products QUI, ProXT, and ProXL, which are shipped with Quintus Prolog.

Options

Any argument that does not match options described in this section and does not start with a '`+`' is regarded as a user's argument. The user's arguments may be obtained using the prolog command `unix(argv(ListOfArgs))`. If a user's argument needs to begin with a '`+`', it should be issued as '`++`' instead or given after the '`+z`' option. Double pluses will be translated into a single plus, so the user's code will not see the '`++`'. Arguments beginning with '`+`' are reserved for prolog, and an unrecognized argument starting with a '`+`' is treated as an error.

'`+` *emacs-arguments*'
: Invoke Quintus Prolog with the Emacs interface. Any arguments following '`+`' are taken as arguments to the Emacs editor. If the environment variable `QUINTUS_EDITOR_PATH` is set then that Emacs invoked. Otherwise, by default, GNU emacs is assumed to be in the path as `emacs`.

'`+f`'
: Fast start. The initialization file '`prolog.ini`' will not be read upon startup.

'`+l` *file*'
: Load the specified file upon startup. *file* can be a Prolog or a QOF file, and it may be specified either as a string (e.g. '`file`', '`~/prolog/file.pl`') or as a file search path specification (e.g. '`library(file)`', '`home(language(file))`'). Note, however, that the latter needs to be quoted to escape the shell interpretation of the parentheses. Giving the extension is not necessary; if both source ('`.pl`') and QOF ('`.qof`') files exist, the more recent of the two will be loaded.

'`+L` *file*'
: Similar to '`+l`' but the user's environment variable `PATH` will be searched for the specified *file*.

'`+p [`*path-name*`]`'
: Print the Prolog file search path definitions that begin with the string *path-name* (e.g. '`library`' if '`+p lib`' is specified); *path-name* is optional, and if not given, all file search path definitions are printed; Prolog exits after producing the required output to `stdout`.

'`+P [`*path-name*`]`'
: Similar to '`+p`', but the absolutized versions of the file search path definitions are printed.

'`+tty`'
: Force the three standard stream associated with a Prolog process to act as tty streams. A tty stream is usually line buffered and handles prompt automatically.

'`+z` *user's-arguments*'
: Any arguments following '`+z`' are taken as user's arguments. User's arguments can then be obtained through `unix(argv(ListOfArgs))`.

Environment

}

PAGER A program to use for interactively delivering the help-system's output to the screen. The default is either `more` or `pg` depending on the host operating system.

PATH Directories to search for the executables and saved-states.

QUINTUS_LISP_PATH
Absolute filename for the Emacs-Lisp directories. The default is: '*quintus-directory*/editor*quintus-version*/gnu'.

QUINTUS_EDITOR_PATH
Absolute filename of the GNU Emacs executable. By default the command `emacs` is looked for in your path.

SHELL Default shell interpreter to be used for Prolog commands `unix(shell)` and `unix(shell(`*command*`))`.

Files

'*file*.pl' Prolog source file

'*file*.qof'
Quintus Object File (QOF) files

'prolog.ini'
Quintus Prolog initialization file, looked up in the home directory

'/tmp/qp*'
Temporary files for loading foreign object files and for Emacs editor interface

'*quintus-directory*/license*quintus-version*'
Location of license files

See Also

`qgetpath(1)`, `qld(1)`, `qpc(1)`, `qsetpath(1)`, `unix/1`, `QP_initialize()`

Section 8.3 [ref-pro], page 186

20.1.2 `qcon` — QOF consolidator

Synopsis

```
qcon [ -wx ] -o output-filename
```

Description

`qcon` consolidates the specified QOF file filename into a machine specific object file. There is no default name for the output file. Therefore, the '`-o`' option must be specified. `qcon` is normally called from `qld(1)` and is not intended to be called by the user directly.

Options

'`-w`' Normally, `qcon` issues warning messages for undefined procedures. This option instructs `qcon` to suppress such messages.

'`-x`' This option instructs `qcon` to issue warnings regarding procedures that are either not called or not defined. '`-x`' overrides the '`-w`' option.

'`-o` *`output-filename`*'
This option is used to specify a name for the object file.

See Also

`qld(1)`, `qnm(1)`, `qpc(1)`, `qsetpath(1)`

20.1.3 qgetpath — Get parameters of Quintus utilities and runtime applications

Synopsis

```
qgetpath [ -abhqr ] filename ...
```

Description

`qgetpath` displays preset parameters defined in the Quintus Prolog executable files, printing the result to the standard output. The executable files specified must be Quintus Prolog utilities, such as `qpc` and `qld`, runtime applications built using the Quintus Prolog Runtime Generator, `prolog`(1), or executables generated with `qld`(1).

There are five paths, *add-ons string*, *runtime-directory*, *quintus directory*, *host type* and *banner message*. *add-ons string* identifies the Quintus add-on products that are included in the executable. *runtime-directory* and *quintus-directory* are used in the executable to find certain files relative to those paths. *host type* identifies the platform of the executable. *banner message* is the banner displayed upon the start-up of the executable. The paths except *banner message* can be obtained through prolog command `prolog_flag(`*`path flag`*`, Variable)` where *path flag* is `add_ons` for *add-ons string* , `runtime_directory` for *runtime-directory*, `quintus_directory` for *quintus-directory*, and `host_type` for *host type*.

Options

'`-a`' Display the *add-ons string* in the specified executables.

'`-r`' Display the *runtime-directory* in the specified executables.

'`-q`' Display the *quintus-directory* in the specified executables.

'`-h`' Display the *host type* in the specified executables.

'`-b`' Display the *banner message* in the specified executables

See Also

`prolog(1)`, `qsetpath(1)`

20.1.4 `qld` — QOF link editor

Synopsis

```
qld [-cCdDEghkNqrRsSvwWxY] [ -o output ] [ -llibrary ]
    [ -L library-directory ] [ -a quintus-product ]
    [ -f path-name:path-spec ]
    [ -F path-name:path-spec ]
    [ -p path-name ]
    [ -P path-name ]
    filename ... [ -LD ld-options ]
```

Description

`qld` links the specified QOF files together with the Kernel QOF file. This results in a single QOF file, which is then consolidated into a machine object file. Finally, the native compiler/linker is invoked to link this object file with the Kernel object file and produce an executable image. The default Kernel QOF file is '*runtime-directory*`/qprte.qof`'. Under UNIX, the default Kernel Object file is '*runtime-directory*`/qprte.o`'; under Windows it consists of '*runtime-directory*`/qprte.lib`' and '*runtime-directory*`/qpeng.lib`' (see `qsetpath(1)` and `qgetpath(1)`).

If any of the specified files depends on a foreign file, then that file will be included in the call to the C compiler or linker if the '`-d`' option is specified. A QOF file depends on a foreign file if its source contains an embedded `load_foreign_files/2` or `load_foreign_executable/1` command for that file (the '`-D`' option of `qnm(1)` shows the dependencies of a QOF file; see `qnm(1)`).

A filename in the command line could be either a QOF file with '`.qof`' suffix, an object file, a shared object file (UNIX only), an import library (Windows only), or an archive file. If the filename specified is a machine object file, it will be passed as an argument to the C compiler or linker. If the command line file does not exist, the same file with a '`.qof`' suffix is tried.

File names may be specified either as regular paths (e.g. '`file.qof`', '`~/home/file.o`') or as Prolog file search paths, such as '`library(file)`', '`home(system(file.o))`', etc. Note that the file search path specifications may need to be quoted to escape the shell's interpretation of the parentheses.

The intermediate QOF and object files are deleted when `qld` exits (unless the '`-k`' options is specified; see below). By default, these files are stored in the directory '`/tmp`'. The environment variable `TMPDIR` may be set to specify another directory to be used instead for temporary files. If `TMPDIR` is set to a non-existent directory or to a directory to which the user does not have read and write permissions then the default value of `TMPDIR` is used for temporary files.

Options

'`-a` *quintus-product*'
: Specifies that the libraries for a particular Quintus product that is sold separately are to be used. These Quintus products are normally installed in the *quintus-directory*. List this directory to find the valid directory names for these products. This option is equivalent to one or more '`-L`' switches. Note that the libraries shipped with Quintus Prolog (`qui`, `proxt`, and `proxl`) are automatically available in the system, and, therefore, require no '`-a`' flag.

'`-c`'
: If this option is specified, `qld` terminates after producing a machine object file. It does not call the C compiler or linker to produce an executable image. If an output filename is not specified with the '`-o`' option, the file is named '`a.o`' under UNIX and '`a.obj`' under Windows. No foreign files, e.g. foreign dependencies, are included in the output file. The resulting object file may be passed to `qld` again on a different command (with '`-N`'), or it may be passed directly to the linker.

'`-d`'
: This flag is always used when `qpc(1)` calls `qld`. It causes all QOF files on which any of the specified files depends to be linked in as well, and any machine object files on which any of the linked QOF files depends to be passed to the C compiler or linker. A QOF file depends on another QOF file if the source for the first contains an embedded command to load the source of the second. A QOF file depends on an object file if its source contains an embedded `load_foreign_files/2` or `load_foreign_executable/1` command of the object file.

  '`-f` *path-name*`:`*path-spec*'Similar to the '`-L`' option, but *path-name*:*path-spec* defines a general file search path, which instructs `qld` to look for a file in directory *path-spec* whenever a file specification of the form *path-name*`(`*file-spec*`)` is encountered in QOF file dependencies or on the command line. The *path-name* and the directory, *path-spec*, are separated by a colon ('`:`'), and, therefore, *path-name* cannot contain a colon. If *path-spec* is given in the file search path form (as in '`library:mylib(library)`'), then the argument must be quoted to escape the shell's interpretation of the parentheses. *path-spec* may be '`.`' or null, in which case '`.`' is assumed.

  There may be a list of path definitions (i.e. '`-f`' or '`-F`' options) for the same *path-name*. `qld` searches the list, just like `prolog` and `qpc`, whenever it needs to expand a file search path specification. The '`-f`' options appends (like assertz in `prolog` and `qpc`) the new path to the end of the list of paths for *path-name*, while '`-F`' prepends (like asserta in `prolog` and `qpc`).

'`-g`'
: This option is not used by `qld`, but is intended for the linker. If some of the specified object files or object dependency files are compiled with the debug flag, this option should be specified to preserve the debugging information in the executable.

'`-h`'
: Hides or "locks" the predicates in the file so that they are not visible to the debugger. Such predicates will have predicate property "locked" when they are linked or loaded into a Prolog system.

'`-k`' With this option, the intermediate files are not deleted.

'`-l`*library*'
This option is not used by `qld`, but is intended for the linker, which is called to link with the specified library to product the executable.

'`-o` *output*'
The default output file names may be overridden using this option. For executable files, the default name is '`a.out`' under UNIX or '`a.exe`' under Windows. With the '`-r`' or '`-R`' options, the default name is '`a.qof`'. With the '`-c`' or '`-C`' options, the default name is '`a.o`' under UNIX or '`a.obj`' under Windows.

'`-p` *path-name*'
The files search path definitions for *path-name* are printed. If *path-name* is '`*`' then all file search path definitions are printed.

'`-q`' Like the '`-r`' option but also adds a Kernel QOF file '`qprte.qof`' or '`qprel.qof`'. This option should very rarely be necessary.

'`-r`'

If this option is specified, `qld` terminates after linking together all the specified QOF files to make a new QOF file. No Kernel QOF file is not linked in. If an output filename is not specified with the '`-o`' option, the file is named '`a.qof`'.

If '`-d`' is not used in conjunction with this option, then it is recommended that the output file, as specified by the '`-o`' option, be in the current working directory. This way, any dependencies of the output QOF file on other files will be correct. Otherwise, the dependencies would only be correct when absolute, as opposed to relative, paths had been specified in the sources. This matters only if this file is to be used in a future call to `qld` with the '`-d`' option specified.

'`-s`' This option is not used by `qld`, but is intended for the linker, which is called to actually generate the executable. The '`-s`' option instructs the linker to strip the executable. **Please note:** that once the executable is stripped then the dynamic foreign interface including the Prolog builtins `load_foreign_files/2` and `load_foreign_executables/1` cannot work.

'`-v`' When this option is specified, `qld` echoes its activities, including calls to subcomponents and '`ld.sh`'

'`-w`' This option suppresses warnings regarding undefined procedures.

'`-x`' When this option is specified, `qld` gives warnings about the predicates that are not called, as well as those that are undefined. Also, a warning message will be printed if either of the user-definable predicates (`portray/1` and `user_error_handler/4`) is undefined. (Such warnings are normally suppressed for these predicates.) The '`-x`' option overrides the '`-w`' option.

'`-C`' UNIX only. Same as '`-c`', except that the object dependencies of QOF files are also linked into the created object file. The object file so produced can be directly passed to the C compiler, linker, or `qld -N` to generate the executable.

'`-D`' If this option is used, `qld` links the specified files with the Development Kernel rather than the Runtime Kernel. The Development Kernel QOF file, '*runtime-directory*`/qprel.qof`', is linked with the specified QOF file, and the Development Kernel object file(s). Under UNIX, the Development Kernel Object file is '*runtime-directory*`/qprel.o`'; under Windows it consists of '*runtime-directory*`/qprel.lib`' and '*runtime-directory*`/qpeng.lib`'. The '`-D`' option may not be used in conjunction with the '`-E`' option.

'`-E`' This option tells `qld` to link the specified files with the Extended Runtime Kernel rather than the Runtime Kernel (see Section 9.1 [sap-srs], page 337). The Extended Runtime Kernel QOF file, '*runtime-directory*`/qprex.qof`', is linked with the specified QOF file, and the default Kernel object file. Under UNIX, the default Kernel Object file is '*runtime-directory*`/qprte.o`'; under Windows it consists of '*runtime-directory*`/qprte.lib`' and '*runtime-directory*`/qpeng.lib`' The '`-E`' option may not be used in conjuction with the '`-D`' option.

'`-F` *path-name*`:`*path-spec*'
Similar to '`-f`', but the path is added at the front of the list of paths for *path-name*. Note that '`-F library:`*library-directory*' is identical to '`-L` *library-directory*'.

'`-L` *library-directory*'
File specifications of the form '`library(`*Filespec*`)`' are searched for in the library search paths when that file is linked. The initial search paths are the same as in the Development System (see `prolog(1)`). Additional directories may be prepended to the list of library search paths with this option. Note that the command line is parsed from left to right. Also note that the '`-L`' must be followed by a space; otherwise, `qld` assumes that the option specifies a library directory for the linker.

Library directories may also be specified with the '`-f`' and '`-F`' options. Note that *library-directory* may be a path to a directory (e.g. '`dir`', '`~/dir`') or a file search path specification of the form '`mylib(library)`'. In the latter case, the *path-name* '`mylib`' must be defined by a '`-f`' or '`-F`' option.

'`-L`*directory*'
UNIX only. Same as the '`-L`' option to the C compiler and linker, and specifies a directory in which the linker looks for library files. This option is simply passed to the linker.

'`-LD`' All remaining options are simply passed to the linker.

'`-N`' Don't link in any Kernel files (e.g. '`qprel.qof`'/'`qprel.o`', '`qprex.qof`'/'`qprte.o`' or '`qprte.qof`'/'`qprte.o`') from the runtime directory. This option is only useful for producing an executable image from a machine object file that was created from QOF files using `qld` (see '`-C`').

'`-P` *path-name*'
Similar to '`-p`', but the absolutized versions of the file search paths are also printed.

'`-R`' Similar to '`-r`', but it does not include the resulting dependencies. This flag is mainly useful for clearing the dependencies from the QOF file (for example, when they get absolutized by `save_program/[1,2]` or `save_modules/2`).

'`-S`'

Use archive files instead of shared object files. Where a qof file contains a dependency on a shared object file, if an archive file exists with the same name but with an archive file extension, then this is substituted for the shared object file in the call to the linker.

Note that if the shared object file that is being substituted contains dependencies to other shared libraries then these have to be included in the `qld` command line. Running `ldd(1)` on a shared object file will indicate whether it has such dependencies.

'`-W`' Windows only. Pass the argument '`-subsystem:windows`', instead of '`-subsystem:console`', to the linker so that a windowed executable is built, rather than a console-based one. The window properties of executables built with '`-W`' can be controlled by the environment variable `CONSOLE`; see below. Other properties of the Windows component of such executables built with '`-W`' can be controlled with *resource files*.

`qld` recognizes files with the '`.res`' extension as resource files and treats them like object files, passing them to the linker. Resource files are generated from '`.rc`' files using the `rc` program supplied with Microsoft Visual C++. They contain various information relating to the Windows component of an application, such as the program name, icon and key bindings. The file `messages(system('qpwin.res'))` contains such data for the Quintus Prolog window. Thus, the `qpwin.exe` executable can be generated with the command:

```
C:\> qld -WDdo qpwin.exe messages(system(qpwin.res))
```

The Quintus Prolog resource source file is in '*quintus-directory*`\src\embed\qpwin.rc`' and this references the icon file '*quintus-directory*`\src\embed\qp.ico`', which `rc` also incorporates into '`qpwin.res`'.

'`-Y`' Windows only. Pass the option '`/dll`' to the linker. See Section 9.2.6 [sap-rge-dll], page 361 for details.

Environment

`TMPDIR` Directory for creating temporary files. The default is '`/usr/tmp`'.

`CONSOLE`

Windows only. Controls the window properties executables build with '`-W`'. The value should be a comma separated list of:

'`sl:`*INT*' history buffer size (default 200)

'`cols:`*INT*'
number of columns (default 80)

‘`rows:`*INT*’
number of rows (default 24)

‘`x:`*INT*’ X position in pixels

‘`y:`*INT*’ Y position in pixels

For example, setting `CONSOLE` to `sl:400,rows:32` before starting `qpwin` yields a window with 32 rows and a history buffer of size 400.

Files

‘`file.qof`’
Quintus Object File (QOF) files

‘`$TMPDIR/qp*.{qof,o,obj}`’
intermediate QOF and object files

‘*runtime-directory*`/qcon`’
the QOF consolidator

‘*runtime-directory*`/qprel.{o,lib}`’
Development Kernel object code

‘*runtime-directory*`/qprte,{o,lib}`’
Runtime Kernel object code

‘*runtime-directory*`/qpeng.lib`’
Windows only. Common Kernel object code, used with the two previous items

‘*runtime-directory*`/qprel.qof`’
Development Kernel QOF code

‘*runtime-directory*`/qprte.qof`’
Runtime Kernel QOF code

‘*runtime-directory*`/qprex.qof`’
Extended Runtime Kernel QOF code

‘*runtime-directory*`/ld.sh`’
Front End script to the UNIX linker

See Also

`cc(1)`, `ld(1)`, `qcon(1)`, `qgetpath(1)`, `qnm(1)`, `qpc(1)`, `qsetpath(1)`

20.1.5 `qnm` — print QOF file information

Synopsis

```
qnm [-PADFMU] [ [-m module] [-n name] [-a arity]]
    [-p proc#] [-o] filename ...
```

Description

`qnm` prints information about each QOF file named in the argument list. The command line options specify the information required. The default behavior, if no command line option is given, is identical to the behavior of the '`-P`' option. Except where noted, only the first valid command line option is accepted per invocation of `qnm`; subsequent options are ignored. Each filename must be a QOF file.

Options

'`-P`' Print information about all procedures that are defined or called in each QOF file.

Information for a procedure is preceded by a decimal number that is unique for that QOF file. This is followed by letter codes indicating the procedure state and properties. The letter codes are:

`U`	undefined
`S`	static
`D`	dynamic
`M`	multifile
`F`	foreign
`V`	volatile
`L`	locked

If the procedure is neither foreign nor multifile, the second letter is omitted.

The state and property information is followed by the module, name and arity of the procedure, in the format *module:name/arity*. Procedure information is sorted alphabetically by procedure name. Internal procedure names and modules that have been made anonymous are printed as '`$anon`'.

'`-A`' Print all atoms referenced in each QOF file. Each atom is preceded by a number that is unique for that QOF file. Atoms are sorted alphabetically. Internal atoms that have been made anonymous are printed as '`$anon`'.

'`-D`' Print the names of all QOF or object files and library directories on which each QOF file depends.

'`-F`' Print the number of source files contained in the QOF file.

'`-M`' Print information about modules contained in the QOF file. This information consists of the name of each module, its export list, and its meta-predicates.

'`-U`' Print information about all procedures that are called but not defined in each QOF file. Information about undefined procedures is in the same format as when using the '`-P`' flag.

'`-m` *module*' Print information about procedures in module *module*. This option may be used in conjunction with the '`-n`' and '`-a`' options.

'`-n` *name*' Print information about procedures named *name*. This option may be used in conjunction with the '`-m`' and '`-a`' options.

'`-a` *arity*' Print information about procedures with arity *arity*. This option may be used in conjunction with the '`-m`' and '`-n`' options.

'`-p` *proc#*' Print information about the procedure numbered *proc#*. Procedure numbers are as given by the '`-P`' option.

'`-o`' Prepend to each output line the name of its QOF file. The '`-o`' flag may be used in addition to any valid combination of options.

See Also

`nm(1)`, `qpc(1)`, `qld(1)`

20.1.6 qpc — Quintus Prolog compiler

Synopsis

```
qpc [-cvhDHMN] [-o output] [-i initialization-file]
     [ -L library-directory ] [ -a quintus-product ]
     [ -f path-name:path-spec ]
     [ -F path-name:path-spec ]
     [ -p path-name ]
     filename ...  [ -QLD qld-options ]
```

Description

`qpc` compiles the specified Prolog files into QOF (Quintus Object Format) files. It then invokes `qld(1)` to link them together and produce an executable image (unless the '`-c`' option is given). The QOF files are not deleted after processing terminates.

Each filename must be the name of a valid Prolog source file or a QOF file. Either absolute or relative filenames may be specified. If filename does not name an existing file, and if it does not already have an extension, then '`.pl`' and '`.qof`' extensions are sought in that order.

File names may be specified as simple paths (e.g. '`file.pl`', '`~/library/file`') or as file search paths of the form '`library(file)`', '`mylib(language(file))`', etc. In the latter case, the path specification must be quoted to escape the shell's interpretation of the parentheses.

Unless the '`-o`' option is given, the name of the output file is the name of the input file with the trailing '`.pl`', if any, replaced by '`.qof`'. If the input filename does not have a '`.pl`' extension, then a '`.qof`' extension is appended. The argument to '`-o`' may also be specified the file search path form (see above).

Source files specified on the `qpc` command line are always recompiled even if the corresponding QOF files are up to date (unless the '`-M`' switch is specified). QOF files, on the other hand, are only recompiled if they are out of date compared to the corresponding source files.

All the "dependencies" of a file, that is all the files named in embedded load commands in that file (or in its source if it is a QOF file), are checked to ensure that they are up-to-date, and they are recompiled if necessary. This checking and recompiling of dependencies can be disabled using the '`-N`' option.

A filename of '`-`' can be used to specify that Prolog source code is to be read from the standard input. The corresponding QOF file will be called '`a.qof`'.

Command line options may alter the above behavior as indicated below. Unrecognized options and their arguments, if any, are passed to `qld(1)`. Furthermore, the arguments following a '`-QLD`' option are not processed by `qpc` but are passed to `qld(1)`. Note also that the command line is parsed from left to right. This will affect how the file search path or library directory definitions are added if '`-f`', '`-F`', '`-L`', or '`-a`' options are used.

Options

'`-a` *quintus-product*'
: Specifies that the libraries for a particular Quintus product that is sold separately are to be used. These Quintus products are normally installed in the *quintus-directory*. List this directory to find the valid directory names for these products. This option is equivalent to one or more '`-L`' switches. Note that the libraries shipped with Quintus Prolog (`qui`, `proxt`, and `proxl`) are automatically available in the system, and, therefore, require no '`-a`' flag.

'`-c`'
: The input files are simply compiled into QOF format, and no further processing takes place.

'`-f` *path-name*`:`*path-spec*'
: Similar to the '`-L`' option, but *path-name*:*path-spec* defines a general file search, which instructs `qpc` to look for a file in directory *path-spec* whenever a file specification of the form *path-name*(*file-spec*) is encountered (in embedded load commands, in QOF file dependencies, or on the command line). The *path-name* and the directory, *path-spec*, are separated by '`:`', and, therefore, *path-name* cannot contain a colon. If *path-spec* is given in the file search path form (as in '`library:mylib(library)`'), then the argument must be quoted to escape the shell's interpretation of the parentheses. *path-spec* may be '`.`' or null, in which case '`.`' is assumed.

  There may be a list of path definitions (i.e. '`-f`' or '`-F`' options) for the same *path-name*. `qpc` searches the list, just like Prolog, whenever it needs to expand a file search path specification. The '`-f`' options appends (like assertz in Prolog) the new path to the end of the list of paths for *path-name*, while '`-F`' prepends (like asserta in Prolog).

  File search paths may also be defined using asserts in the Prolog source being compiled or in initializaiton files (see '`-i`'). The '`-f`', '`-F`', '`-L`', and '`-a`' options given on the `qpc` command line, and file_search_path and library_directory definitions asserted in source files or initialization files, are passed on to `qld(1)`.

'`-h`'
: Hides or "locks" the predicates in the file so that they are not visible to the debugger. Such predicates will have predicate_property "locked" when they are linked or loaded into a Prolog system.

'`-i` *initialization-file*'
: Specifies an initialization file. The initialization file may be a source ('`.pl`') or QOF ('`.qof`') file. Currently, source files cannot load foreign code; in other words, they cannot contain calls to `load_foreign_files/2` or `load_foreign_executable/1`. The definitions in the initialization file apply during the com-

pilation of all files specified to the right of the '`-i`' switch on the command line. The definitions in the initialization file apply only during compile time. Therefore, no QOF file is generated from an initialization file and its content is not included in any of the generated QOF files. The initialization file may be specified in the file search path form (eg. '`-i "library(basics)"`').

'`-o` *output*'
: Specifies a name for the output file. If used with the '`-c`' option, the qof file will be produced into the given file. In this case, there may be several '`-o`' options for each qof file. If the '`-o`' names a directory, all qof files will be placed in the given directory. If the '`-c`' option is not used, the '`-o`' is passed onto `qld(1)`.

'`-p` *path-name*'
: This option is just passed to `qld(1)` along with its argument, asking `qld(1)` to print out the file search definition for *path-name*.

'`-v`'
: When this option is specified, `qpc` echoes its activities, including the call to `qld(1)`, to the standard output. This flag is also passed on to `qld(1)`.

'`-D`'
: This option is just passed to the linker `qld(1)` indicating that the Quintus Development System should be linked in.

'`-F` *path-name*`:`*path-spec*'
: Similar to '`-f`', but the path is added at the front of the list of paths for *path-name*. Note that '`-F library:library-directory`' is identical to '`-L library-directory`'.

'`-H`'
: Like '`-h`' but in this case the hiding (locking) is done also to any files that are compiled because of embedded load commands in the file.

'`-L` *library-directory*'
: File specifications of the form '`library(`*Filespec*`)`' encountered in embedded load commands are searched for in the library search paths. The initial search paths are the same as in the Development System (see `prolog(1)`). Additional directories may be prepended to the list of library search paths with this option. Note that the command line is parsed from left to right. Also note that the '`-L`' must be followed by a space; otherwise, `qpc` assumes that the option specifies a library directory for `qld`.

  Library directories may also be specified with the '`-f`' and '`-F`' options. library-directory may be a path to a directory (e.g. '`dir`', '`~/dir`') or a file search path specification of the form '`mylib(library)`'. In the latter case, the path-name '`mylib`' must be defined either in the Prolog source code being compiled or by '`-f`' or '`-F`' options.

'`-M`'
: Specifies that files on the command line are not to be compiled if their corresponding QOF files are more recent than they are.

'`-N`'
: Specifies that files specified in embedded load commands are not to be compiled. (By default they would be compiled unless their QOF files are already up-to-date.)

'`-QLD`'
: All remaining options are simply passed to `qld(1)`.

Environment

`TMPDIR` Directory for creating temporary files. The default is '`/usr/tmp`'

Files

'`a.out`' Executable output file

'`a.qof`' Output QOF file if filename is '`-`'

'`file.pl`' Prolog source file

'`file.qof`'
Quintus QOF file

'`$TMPDIR/qp*`'
Compiler temporary files

'*`runtime-directory`*`/qld`'
QOF link editor

'*`runtime-directory`*`/qcon`'
QOF consolidator

See Also

`prolog(1)`, `qcon(1)`, `qgetpath(1)`, `qld(1)`, `qnm(1)`

20.1.7 `qplm` — Quintus Prolog license manager

Synopsis

```
qplm -i SiteName

qplm -a Product Users [ Expiration ] Code

qplm -d User Product

qplm -p
```

Description

`qplm` initializes and maintains the license files for Quintus products.

A code is supplied for each Quintus product that is based on the name of the site or company name, product name, number of users and optionally an expiration date.

Users are distinguished as either occasional users, if they have used the product less than 5 times, or else regular users. When determining whether the number of users is within that allowed by the license, only regular users are counted.

Expiration dates are specified with the format YY-MM-DD.

Options

'`-i` *`SiteName`*'
: Initializes the license files for *SiteName*, where *SiteName* is a number of arguments comprising the site or company name. This command must be executed before any products are added.

'`-a` *`Product Users Expiration Code`*'
: Adds a products to the license file. The product is typically of the form *name*/*arch*/*version*, *Users* is the number of users allowed to use the product. *Expiration* is an optional argument specifying when the license will expire. The final *Code* argument is a 16 character code that is based on the *SiteName*, *Product*, *Users* and *Expiration*.

'`-d` *`User Product`*'
: Deletes *User* from the list of users who use *Product*. When a user no longer uses *Product*, he or she can be removed from the license file with this option.

'`-p`'
: Print the information in the license file. This prints the site name followed by all of the products licensed. This also prints the list of users using each product. For occasional users, the number of times they have used the product is also shown.

Example

To initialize the license file:

```
% qplm -i Hallatrow Designers Inc.
```

To add a 2 user license for Prolog that expires on 17 May 1994,

```
% qplm -a prolog/hppa/3.5 2 94-05-17 thiscodewontwork]
```

Files

‘`license.qof`’
: Contains the site name and product codes

‘`users.qof`’
: Records the users using the products

The license files are maintained in ‘*quintus-directory*`/license`*version*’. The ‘`users.qof`’ file must be writable by all users, therefore if the *quintus-directory* is stored on a read-only file system then the license subdirectory should be made into a symbolic link to a writable directory.

See Also

`prolog(1)`

20.1.8 `qsetpath` — Set parameters of Quintus utilities and runtime applications

Synopsis

```
qsetpath [ -aadd-ons ] [ -rruntime-directory ]
    [ -qquintus-directory ] [ -hhost-type ]
          [ -bbanner-message ] filename ...
```

Description

`qsetpath` sets parameters for the executable files. The executable files specified must be Quintus Prolog utilities, such as `qpc` and `qld`, runtime applications built using the Quintus Prolog Runtime Generator, `prolog(1)`, or executables generated with `qld(1)`. There are five settable paths, *add-ons string*, *runtime-directory*, *quintus-directory*, *host-type* and *banner message*. *add-ons string* identifies the Quintus add-on products that are included in the executable. *runtime-directory* and *quintus-directory* are used in the executable to find certain files relative to those paths. *host-type* identifies the platform of the installation. *banner message* is the banner displayed upon the start-up of the executable. The paths except *banner message* can be obtained through prolog command `prolog_flag(`*`path flag`*`, `*`Variable`*`)` where *path flag* is `add_ons` for *add-ons string*, `runtime_directory` for *runtime-directory*, `quintus_directory` for *quintus-directory*, and `host_type` for *host-type*.

Options

'`-a`*`add-ons`*'
: Set the add-ons identification string of the specified executables to *add-ons*.

'`-r`*`runtime-directory`*'
: Set the *runtime-directory* of the specified executables to *runtime-directory*. The specified *runtime-directory* must be an absolute filename.

'`-q`*`quintus-directory`*'
: Set the *quintus-directory* of the specified executables to *quintus-directory*. The specified *quintus-directory* must be an absolute filename.

'`-h/host-type`'
: Set the *host-type* of the specified executables to *host-type*. There must be a '`/`' preceding *host-type*. A *host-type* should be in the form of *machine-type* or *machine-type-OS-version*,

'`-b`*`banner-message`*'
: Set the displayed banner message for the specified executables to *banner-message*.

Errors

Setting banner message has partial or no effects on certain executables.

Argument specification must following the option immediately. There is no white space allowed between an option and its argument.

See Also:

`prolog(1)`, `qgetpath(1)`, `qld(1)`

20.1.9 `qui` — Quintus User Interface

Synopsis

qui [+f] [+l file] [+L file]

```
[ +p [path-name] ] [ +P [path-name] ]
      [ X-window arguments ]
      [ +z users arguments ]
```

Description

The Quintus User Interface (QUI) is a Motif-based window interface to the Quintus Prolog Development System. It includes a query interpreter window with history menu, a source linked debugger, a help window for accessing the on-line manuals and an edit window as well as an interface to the GNU Emacs editor.

Options

Any argument that does not match options described in this section and does not start with a '`+`' is regarded as a user's argument. The user's arguments may be obtained using the prolog command unix(argv(ListOfArgs)). If a user's argument needs to begin with a '`+`', it should be issued as '`++`' instead or given after the '`+z`' option. Double pluses will be translated into a single plus, so the user's code will not see the '`++`'. Arguments beginning with '`+`' are reserved for Prolog, and an unrecognized argument starting with a '`+`' is treated as an error.

'`+f`' Fast start. The initialization file '`prolog.ini`' will not be read upon startup.

'`+l` *file*' Load the specified file upon startup. *file* can be a Prolog or a QOF file, and it may be specified either as a string (e.g. '`file`', '`~/prolog/file.pl`') or as a file search path specification (e.g. '`library(file)`', '`home(language(file))`'). Note, however, that the latter needs to be quoted to escape the shell interpretation of the parentheses. Giving the extension is not necessary; if both source ('`.pl`') and QOF ('`.qof`') files exist, the more recent of the two will be loaded.

'`+L` *file*' Similar to '`+l`' but the user's environment variable `PATH` will be searched for the specified *file*.

'`+p [`*path-name*`]`'
Print the Prolog file search path definitions that begin with the string *path-name* (e.g. `library` if '`+p lib`' is specified); *path-name* is optional, and if not given, all file search path definitions are printed; QUI exits after producing the required output to `stdout`.

‘`+P [path-name]`’
: Similar to ‘`+p`’, but the absolutized versions of the file search path definitions are printed.

X-Window arguments
: Any arguments recognized as standard X options are passed to the X-Window system. Examples of these include ‘`-display displayname`’, ‘`-fg color`’, ‘`-bg color`’.

‘`+z user's-arguments`’
: Any arguments following ‘`+z`’ are taken as user's arguments. User's arguments can then be obtained through `unix(argv(ListOfArgs))`.

Environment

`QUINTUS_EDITOR_PATH`
: Name of the GNU Emacs command. If set, this is invoked as the editor rather than the standard text editor built in to QUI.

`QUINTUS_LISP_PATH`
: Absolute filename for the Emacs-Lisp directories. The default is: ‘*quintus-directory*`/editor3.5/gnu`’.

See Also

`prolog(1)`

Section 3.1 [qui-qui], page 53

Predicate Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Keystroke Index

H

L

M

N

P

Q

R

S

T

W

X

Z

Book Index

[

^

_

\

\

|

~

0

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z