Supplied by

Artificial Intelligence Limited

European Distributor of Quintus Prolog

Intelligence House
58-78 Merton Road
Watford
Hertfordshire WD1 7BY
England
Telephone Watford (0923) 47707
Telex 933883

Quintus Prolog User's Guide

Quintus Computer Systems, Inc.
2345 Yale St, Palo Alto, CA 94306

Written by Patti Kahn, with contributions from David Bowen,
Lawrence Byrd, Bill Kornfeld, Vince Pecora and Jennifer Zimmerman.

Table of Contents

INTRODUCTION

Quintus Prolog is a complete applications development system designed to streamline and facilitate program development. To maximize programmer speed and efficiency, Quintus Prolog provides

- a text editor interface

- a comprehensive debugger

- an on-line documentation and help facility

- a high degree of compatibility with DEC10 Prolog and Prolog-20

These features combine to produce a system that fully supports a programmer's needs during application development.

This User's Guide explains how to use Quintus Prolog to run Prolog programs. The guide includes information on entering and exiting Prolog, using the Emacs/Prolog editor interface, loading and running programs, debugging programs, and designing your programs for maximum readability, reliability, and efficiency. The appendices include information on porting programs written in DEC10 Prolog or Prolog-20, a summary of Emacs commands, a list of built-in predicates, and a list of built-in operators.

This User's Guide is written for people with a working knowledge of Prolog. For an introduction to Prolog, refer to Programming in Prolog (Clocksin and Mellish, Springer-Verlag, 1981). For more detailed information on Quintus Prolog, including information on syntax, built-in predicates, and notes on the Emacs interface, refer to the Quintus Prolog Reference Manual.

SYMBOLS AND CONVENTIONS

Throughout this manual, certain symbols and conventions are used.

- Control characters are represented in text by the notation Control **x**, where **x** represents the character you type while holding down the Control key. In examples, control characters are represented by the notation ^x.

- For many Emacs functions, you type a control character followed by another character. If the second character is preceded by the word Control, as in Control **x** Control v, continue to hold down the Control key as you type. If the second character is not preceded by the word Control, as in Control **x** o, you must release the Control key before you type the character. If you don't, the system won't recognize what you type.

- Certain functions in Emacs require that you press the Escape key and then type another character. These keys are represented by the notation Escape x in text and <esc>x in examples, where x represents the character you type after pressing the Escape key and releasing it. (On some keyboards, the Escape key is referred to as the Alt key.) Some terminals have a "meta" key which can be used as an alternative to Escape.

- The symbol <ret> is used to represent a user-typed carriage return in examples. This key is usually marked Return.

- Boldface type is used to indicate user responses in examples that show interactions between a user and Prolog. For instance, in the example below, the system prompt is shown in regular type, and the user response is shown in boldface.

  ```
  | ?- consult('file1.pl').
  ```

- Underlined words in examples indicate the type of information that should appear in a particular position. For instance, in the example below, the term filename would be replaced by an actual file name.

  ```
  | ?- consult(filename).
  ```

- Since the main Unix prompt varies from system to system, the symbol <Unix prompt> is used to represent the main Unix prompt in examples.

- The Unix end-of-file character is represented as Control d, which is the default end-of-file character at most Unix sites. The end-of-file character for your site may be different.

SECTION 1

GETTING STARTED

Quintus Prolog offers a flexible development environment that can accommodate the user's varying needs at different stages of the development process. Prolog can be accessed within an editor interface, a feature which greatly simplifies the editing and debugging phase, or Prolog can be accessed independently. This section describes how to access Prolog with and without the editor interface, what you'll see once you've entered Prolog, and how to use the on-line help system.

USING PROLOG WITH THE EMACS EDITOR

The Emacs/Prolog interface is designed to enable you to create a Prolog program and then to move back and forth easily between the file containing that program 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 Prolog source program appears in a "text window" on the top half of the screen, and the Prolog environment appears in the "Prolog window" on the bottom half of the screen.

In the Prolog window, Prolog programs can be run; and with Emacs, previously entered Prolog commands can be edited and resubmitted. In the text window, single procedures, groups of procedures, and entire programs can be edited and quickly reloaded without suspending the Prolog process. Additionally, any number of Prolog source files can be loaded into Prolog at once; and then, if required, Emacs can be used to locate a specific procedure in any one of those files.

Entering Prolog and Emacs

To enter Prolog and the Emacs editing environment simultaneously, type **prolog +** at the main Unix prompt. Notice that there must be a space between the word **prolog** and the **+** sign.

```
<Unix prompt>prolog +
```

The system displays a message telling you it is loading the editor. It then divides the screen into two windows, as shown below.

```
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|---------------------------------------------------------------|
| Buffer: ScratchPad.pl  File: ScratchPad.pl    (Prolog)  Top   |
|---------------------------------------------------------------|
|                                                               |
| Quintus Prolog Release 1.0                                    |
| Copyright (C) 1985, Quintus Computer Systems, Inc.            |
|                                                               |
| | ?-                                                          |
|                                                               |
|                                                               |
|                                                               |
|---------------------------------------------------------------|
| Quintus Prolog                                                |
|---------------------------------------------------------------|
```

The upper window is the text window, and the lower window is the Prolog environment. A mode line appears at the bottom of each window to identify the window's contents. You can move back and forth between the windows by using the Control x o command, as described on page 19.

The cursor is positioned to the right of the Prolog prompt | ?- in the Prolog window, indicating that you are at the top level of the Prolog system. Prolog commands may be executed from this point as described in the PROLOG PROMPT section below.

Alternatively, you can enter the Emacs/Prolog environment and retrieve your file in one step by typing prolog + <u>filename</u>, where <u>filename</u> is the name of the file you want to retrieve. For example, to enter the Emacs/Prolog environment and retrieve the file **myfile**, you would type

```
<Unix prompt>prolog + myfile
```

Exiting Emacs

You can exit from the editor in one 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 below. If you want to temporarily halt your session, return to the main Unix level, and then resume your session, you should choose the second exit option discussed below.

To exit from the editor irreversibly, type **Escape Control c**. If you have files that have been modified but not saved, Emacs displays the following message at the bottom of the screen:

```
Modified buffers exist, do you really want to exit?
```

If you want to store the information you've modified in a file, use the Control x Control s command (see page 12). If you want to exit without saving the modified information, type y, and you will be returned to the main Unix prompt.

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

```
Your Prolog process is still running, do you want it killed? (y/n)
```

To end the Prolog session, type y, and you will be returned to the main Unix prompt. If you don't want to end the Prolog session, type n to abort the exit. Prolog will continue running.

To exit from the editor by suspending the current editor session, type Control x Control c. If you type Control x Control c, you will see the message Stopped printed at the bottom of the screen and you will be returned to the main Unix prompt. However, the Emacs/Prolog job is only suspended and may be resumed by you at any time. To resume your Emacs/Prolog session, type fg (for foreground) at the main Unix prompt.

NOTE: In Release 1.0, Control x Control c only works under the C shell on Unix. Consult the Unix documentation for more information on the C shell.

USING PROLOG WITHOUT THE EMACS EDITOR

If you are not using the Emacs editor, you may enter Prolog independently and subsequently load the file(s) containing your program.

Entering Prolog

To enter Prolog without the Emacs interface, type prolog at the main Unix prompt:

```
<Unix prompt>prolog
```

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

```
Quintus Prolog
Copyright (C) 1985, Quintus Computer Systems, Inc.

| ?-
```

The main Prolog prompt | ?- indicates that you are at the top level of the Prolog system. At this point, you can direct the system to read in the contents of a previously-created file containing a Prolog program, as described in Section 3, page 25.

Exiting Prolog

To exit from Prolog when you are not using the Emacs interface, type Control **d** at the main Prolog prompt.

```
| ?- ^d
```

Prolog displays the message shown below and then returns you to the Unix environment.

```
[ End of Prolog execution ]
<Unix prompt>
```

NOTE: As mentioned earlier, Control d is the standard Unix end-of-file character. If Control d doesn't work at your site, type your system's end-of-file character or type halt. to exit.

THE TOP-LEVEL PROLOG PROMPT

The prompt | **?-** indicates that you are at the top level of the Prolog system. Prolog is now waiting for a command or a question. For example, you could type the **write** command as shown below; the system would execute the command and then return to the top level Prolog prompt.

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

Similarly, you could pose an arithmetical question.

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

X = 4<ret>

| ?-
```

When your Prolog programs have been loaded (see Section 3, page 23), questions and commands referring to your programs can also be given.

How to Return to the Top Level Prolog Prompt

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

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

Type a (for abort). The system then returns an execution aborted message and the top level Prolog prompt.

USING THE ON-LINE HELP SYSTEM

Quintus Prolog provides an on-line help system, which contains on-line versions of this User's Guide and the Quintus Prolog Reference Manual. You can obtain access to the on-line help system from within Prolog, whether or not you are using the Emacs interface.

Requesting General Information

If you are running Prolog under Emacs, the simplest way to ask for general information is to type **help.** at the main Prolog prompt.

```
| ?- help.
```

When you type help., the system displays some basic information about running Prolog and mentions that you can type help(Topic). to get information about a particular topic in which you are interested. For example, you might type

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

if you are interested in learning about using the debugger. In response to this you will get a menu indicating all the parts of the User's Guide and the Reference Manual which are concerned with debugging. Note that you can abbreviate topics; if you type

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

you will get a menu referring to all topics beginning with the characters "deb". Thus, the more you abbreviate, the larger the menu you are likely to get.

Once you have a menu, you can select a topic. If you are running under Emacs, the following line appears at the bottom of the screen to explain how to select a topic:

```
<SPACE> to advance cursor, <RETURN> to select item, q to Quit, ? for Help
```

To advance the cursor to the next item on the menu, press the spacebar. To select a topic, position the cursor next to that topic and press the Return key. To exit from the Help facility, type q. And for additional help, type a question mark (?).

When you select a topic from the menu, the sytem displays either another menu or text about the topic you selected. If the system displays text, you can page forward through the text by pressing the spacebar and backward by pressing the backspace key.

If you are not running under Emacs, type manual. at the main Prolog prompt to gain access to the on-line help system. (Alternatively, you can type manual(user). or manual(ref). at the main Prolog prompt, as described below.) The system displays the following menu:

```
     Quintus Prolog On-line Documentation System

   1 - The User's Guide for Quintus Prolog          {manual(user)}
   2 - The Reference Manual                         {manual(ref)}
```

To see a menu of topics for the User's Guide, type manual(user). at the main Prolog prompt. To see a menu of topics for the Reference Manual, type manual(ref). at the main Prolog prompt. The system displays a list of major topics, or chapters, in the indicated manual. To the right of each topic is the code you type to view that topic. For example, the first topic in the reference manual is listed as

```
   1 - Introduction                 {manual(ref-1)}
```

To select this topic, you would type the following at the main Prolog prompt:

```
| ?- manual(ref-1).
```

Note that you omit the curly brackets when you select the topic.

In response, the system displays the appropriate section of the manual on the screen.

NOTE: In the on-line help system, sections of the User's Guide are prefixed by the word user, as in manual(user-4-5); sections of the Reference Manual are prefixed by the word ref, as in manual(ref-6-7).

Occasionally, you will see cross references in the text. Cross references look like this in the on-line manual:

```
see also {manual(ref-5-1-1)}
```

If you are running Prolog under Emacs and you want to look at the section being cross-referenced, type x. The cursor will move to the line on the current screen containing the cross-reference. Press the Return key, and the system will display the section being cross-referenced.

If you are not running Prolog under Emacs and you want to look at the section being cross-referenced, type manual followed by the section you want to look at. For example, to look at the section being cross-referenced above, you would type the following:

```
| ?- manual(ref-5-1-1).
```

Requesting Information About a Specific Topic

As mentioned above, if you type help(Topic)., the system displays a menu of all the sections in the on-line help system that contain information about that topic. You can then select the entry you want to look at from that list. However, if you already know what entry you want to look at, and you want to go directly to that entry, you can instead type manual(Topic). at the main Prolog prompt. The system then displays information about that topic.

Using **manual**(Topic). can be particularly helpful if you want to obtain information about built-in predicates, as the reference manual contains an entry for each built-in predicate. For example, to request information about the built-in predicate trace, you would type **manual(trace).**, as shown below:

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

The difference between **manual**(Topic) and **help**(Topic) is just like the difference between looking up something in a table of contents and looking it up in an index. The **help** command is better when you are looking for general information; the **manual** command is better when you want to go directly to the portion of the manual that contains information about a particular topic. With both commands, you can abbreviate Topic; but the menus you get when you abbreviate are different for the two commands.

SECTION 2

USING THE EMACS EDITOR

This section explains how to use Emacs to create and edit a file. If you are already familiar with Emacs, you may want to skip this section. A summary listing of the Emacs functions and key bindings can be found in Appendix C. Information on entering and exiting Emacs can be found in Section 1.

NOTE: If you are not using the Emacs editor interface, skip to Section 3.

CREATING A FILE

To create a new file, type Control x followed by Control v. The following prompt appears at the bottom of the screen:

```
: * visit-file
```

Select a file name that complies with Unix file-naming conventions; end the file name with a .pl extension. The .pl extension is essential for work with Prolog. Type the file name and press the Return key.

NOTE: Any time the cursor is positioned at the bottom of the screen and the system is waiting for your input, you can cancel the command that moved the cursor to the bottom of the screen by typing Control g.

After you type the file name, the system displays a line like the following at the bottom of the screen to let you know that a new file has been created:

New file: /<u>yourdirectory</u>/<u>filename</u>

The system also moves the cursor to the upper window so you can begin editing the file. Note that while you are editing, you are actually working with a buffer rather than the file itself. When you finish editing, you save the contents of the buffer, as described below, in the file whose name you specified.

RETRIEVING AN EXISTING FILE

To retrieve an existing file, type Control x followed by Control v. The following prompt appears at the bottom of the screen:

```
: * visit-file
```

Type the name of the file you want to retrieve and press the Return key.

: * visit-file <u>filename</u>

The system then copies the file into a buffer and displays it in the upper window of the screen so you can begin editing it.

If you cannot remember the name of the file you want to retrieve, type a question mark (?) in response to the : * visit-file prompt. Emacs then displays a list of your files. Type Control x Control v again, and when the cursor moves to the bottom of the screen, type the name of the file you want to retrieve.

NOTE: As a shortcut, you can type the first few letters of the file name and then press the spacebar. When you press the spacebar, Emacs searches through your file directory and attempts to find a file name beginning with the characters you've typed. If Emacs can match what you've typed with a file name, it completes the file name when you press the spacebar and displays the file in the text window. If Emacs cannot match the file name, or if it finds more than one file name that begins with the characters you've typed, it beeps. If that happens, type another letter or two and then press the spacebar again.

SAVING YOUR WORK

If you want to save the contents of the buffer being edited in the upper window, you can do so by positioning the cursor in that window and then typing Control x Control s (for save-file). This will save the text in the file whose name appears in the mode line at the bottom of the text window.

Occasionally you may want to save the text in a file under a different name. You can do this by using the command Control x Control w (for write named file) instead of Control x Control s. When you type Control x Control w, you will see the prompt

```
: write-named-file
```

at the bottom of the screen. Type the name of the file you want to write to and press the Return key.

MOVING THE CURSOR IN AN EMACS FILE

Emacs is a full-screen editor; so to make changes in a file, you first move the cursor to the letter or word you want to change. The following table shows the editing function keys that move the cursor within a file. (Recall that the symbol ^ represents the Control key, and that the symbol <esc> represents the Escape key.)

Function	Key
Move right (forward) a character	^f
Move right (forward) a word	<esc>f
Move right (to end of line)	^e
Move left (backward) a character	^b
Move left (backward) a word	<esc>b
Move left (at beginning of line)	^a
Move down a line (to next line)	^n
Move up a line (to previous line)	^p
Move to the end of the file	<esc> >
Move to the beginning of the file	<esc> <
Move forward a screen	^v
Move backward a screen	<esc>v

If you are using a Sun workstation, you can also use the Mouse to move the cursor around and to scroll the text.

- To move the cursor, move the Mouse until the arrow points to the position you want to move the cursor to, and then click the middle button.

- To scroll text upwards, move the Mouse until the arrow points to the line you would like to be at the top of the window, and then click the right-hand button.

- To scroll text downwards, move the Mouse until the arrow points to the line you would like to be at the bottom of the window, and then click the left-hand button.

INSERTING TEXT

To insert text in a new file, simply type the text the way you want it to appear. To insert text at the end of an existing file, move the cursor to the end of the file and type your text.

To insert a letter or a word into an existing line of text, move the cursor to the letter that should come after the letter or word you want to add. Then type the text you want to add.

To insert a new line of text into an existing file, move the cursor to the beginning of the line that should follow the new line. To insert a blank line, type Control o; then type your text. If you want to insert several lines of text, type a carriage return at the end of the first line you type, and then type carriage returns at the end of each successive line you type until you finish.

DELETING TEXT

To delete text, move the cursor to the text to be deleted. Then use the appropriate editing function key to delete the text, as shown below.

Function	Key
Delete character to the left	Del
Delete word to the left	<esc>Del
Delete character to the right	^d
Delete word to the right	<esc>d
Delete (kill) all characters from cursor to end of line	^k

To delete a block of text, you first mark one end of the block by moving the cursor there and typing Control @ (Control Shift 2 on many terminals). Alternatively, you can set a mark by pressing the Control key and then pressing the spacebar. After you set a mark, Emacs displays the following message at the bottom of the screen:

```
Mark set
```

Move the cursor to the end of the text you want to delete. Then type Control w (for wipe out). The block of text that was between the mark and the cursor is removed from the screen and temporarily put in a buffer.

NOTE: If you delete a block of text and then decide you don't want to delete it after all, you can put it back where it was by typing Control y (for yank back text).

You can set a marker at either the beginning or end of a block of text by typing Control @. If you set a marker at the beginning of a block of text, you mark the end of the block by simply moving the cursor there, as described above. Conversely, if you set a marker at the end of a block of text, you mark the beginning of the block by moving the cursor there.

Once you have marked a block of text, you can verify the block by typing Control x Control x. Control x Control x moves the cursor from its current location to the other end of the block of text. You can toggle the cursor between the beginning and end of the blocked text by repeating the Control x Control x command.

COPYING TEXT

To copy a line of text, delete the line, as described above, by positioning the cursor at the beginning of the line and typing Control k. Type Control y to restore the text to its original position. Then move the cursor to the location in the file where you want to copy the text, and type Control y again. The system then copies the text into the new location. You can copy the same line as many times as you like by moving the cursor to the location where you want the text and typing Control **y**.

NOTE: Control y always yanks back the last thing you deleted, so be careful not to delete anything while you are copying text.

To copy a block of text, mark the beginning of the block as described above in the section on DELETING TEXT. Move the cursor to the end of the block you want to copy. Then type Escape **w**. The text between the mark and the cursor remains on your screen, but it is also put into a buffer. Move the cursor to the point in your file where you want to copy the text. Then type Control y (for yank text back). The system then copies the text to the new location in your file.

MOVING TEXT

To move a line of text, position the cursor at the beginning of the line you want to move and then type Control k twice. The first time you type Control k, the text is simultaneously removed from your screen and put into a buffer; however, a blank line is left on the screen where the text used to be. The second time you type Control k, the blank line is removed.

After you remove the text from the screen, you can empty the buffer containing the text -- that is, move its contents back onto the screen -- at any place you choose. To move the deleted text from the buffer back onto the screen, position the cursor where you want to move the text and then type Control **y** (for yank text back). Emacs then moves the text to the new location in your file.

To move a block of text, mark the block and delete it, as described above in the section on DELETING TEXT. Move the cursor to the point in your file where you want the text, and then type Control **y** (for yank text back). Emacs then moves the block of text to the cursor's current location.

NOTE: Control **y** always yanks back the last thing you deleted, so be careful not to delete anything while you are moving text.

FINDING TEXT

To locate text in your file, position the cursor at the beginning of the region you want to search and type Control s (for search). Emacs displays the following message at the bottom of the screen:

```
I-search:
```

I-search stands for incremental search. An incremental search begins as soon as you type the first letter of the word or character string you want to locate. When you type the first letter of the word, Emacs displays that letter at the bottom of the screen and searches through your file, beginning at the cursor's current position, for the first occurrence of that letter. When Emacs locates that letter, it moves the cursor to the letter immediately following it.

For example, suppose you are searching for the word _file_. When you type f, Emacs moves the cursor to the next occurrence in your file of the letter f. If that happens to be in the word _file_, you can halt the search by pressing the Escape key. However, that will usually not be the case, and you will want to continue the search by typing the next letter in the string you are trying to locate.

Each time you type another letter, Emacs adds that letter to the character string displayed at the bottom of the screen. In addition, it moves the cursor to the next occurrence in your file of the character string that is displayed at the bottom of the screen.

For example, suppose you positioned the cursor at the top of the file and then typed Control **s** followed by **f**. The following line would be displayed at the bottom of the screen.

```
I-search: f
```

In addition, the cursor would move to the first occurrence of the letter f in your file. If you then typed i, the line at the bottom of the screen would look like this:

```
I-search: fi
```

The cursor would then move to the first occurrence of the letters fi. You would continue spelling out the word you wanted to locate until Emacs located the word and moved the cursor to it.

Once you have located the character string you are looking for, press the Escape key to halt the search function. You can then make any necessary editing changes. To cancel the search function and move the cursor back to the point where it was when you started the search, type Control **g**.

Correcting Typing Errors

If you type a character string that does not match any character string in your file, Emacs displays the following message at the bottom of the screen:

```
Failing I-search:
```

If you receive this message and want to halt the search function, type Control g. However, if you made a typing error and want to continue the search, you can delete the last character you typed by pressing the Delete key. When you press the Delete key, the last character typed disappears from the character string shown at the bottom of the screen, and the cursor moves backward a character. You can then type the correct character and continue the search.

Searching Backward Through a File

To search backward through your file, type Control r (for reverse), and then type the character string you want to find. Emacs will search for the character string, as described above, except that it will search backward, rather than forward, from the cursor's current position.

REPLACING TEXT

You can replace an existing word or character string with another word in two ways: you can either replace every occurrence of the word by issuing a single command, or you can direct Emacs to ask you which occurrences of the word you want to replace.

Replacing All Occurrences of a Word

To replace all occurrences of an existing word or character string, move the cursor to the beginning of the file. Then type Escape r. Emacs moves the cursor to the bottom of the screen and displays the prompt

```
Old string:
```

Type the word or character string you want to replace, and press the Return key. Emacs then asks you to type the new string. Type the new word you want to use and press the Return key. Emacs then replaces all occurrences of the old word with the new word and displays a message telling you how many occurrences it replaced.

The succession of prompts looks like this, except that in a real session, each line would overwrite the preceding line at the bottom of the screen:

```
<esc>r
Old string: file<ret>
New string: program<ret>
Replaced 3 occurrences
```

NOTE: Use this command with caution. If you are not sure that you want to replace all occurrences of a word, use the selective replace command instead.

Replacing Selected Occurrences of a Word

To replace only selected occurrences of a word or character string, move the cursor to the beginning of the file or the beginning of the region where you want to replace text. Then type Escape q. Emacs moves the cursor to the bottom of the screen and displays the prompt

```
Old string:
```

Type the word or character string you want to replace, and press the Return key. Emacs then asks you to specify the new character string to be used. Type the new word and press the Return key. Emacs then displays the message

```
Query-Replace mode
```

and moves the cursor to the first occurrence of the string. Press the Return key again. Emacs displays the following prompt at the bottom of the screen:

```
Options: ' ' ','=>change; 'n'=>don't; '.'=>change, quit; '^G'=>quit
```

The succession of prompts looks like this, except that in a real session, each line would overwrite the preceding line at the bottom of the screen:

```
<esc>q
Old string: contains<ret>
New string: refers_to<ret>
Query-Replace mode <ret>
Options: ' ' ','=>change; 'n'=don't; '.'=>change, quit; '^G'=>quit
```

To replace the first occurrence of the specified word, press the spacebar. The spacebar is represented in the prompt by the first option, which is a pair of single quotes surrounding a space (' '). When you press the spacebar, Emacs changes the existing word to the new word and then moves the cursor to the next occurrence of the word in your file.

To change the word and not move the cursor to the next location of the word, type a comma (,). Emacs then replaces the word. To move to the next occurrence of the word, press the spacebar.

If you don't want to replace this occurrence of the word, type n. Emacs does not replace the word; instead, it moves the cursor to the next occurrence of the word.

To change the word and then halt the Replace function, type a period (.). Emacs then replaces the word where the cursor is currently located, halts the Replace function, and then displays a message at the bottom of the screen that tells you how many occurrences of the word were replaced.

To exit from the Replace function at any time, type Control g.

NOTE: After you perform the first replacement, the prompt line disappears from the bottom of the screen. However, unless you exit from the Replace facility, you can continue to replace words by using the commands that were shown in the prompt. To redisplay the prompt line, type a question mark (?).

EDITING SEVERAL FILES SIMULTANEOUSLY

From time to time, you might find it necessary or desirable to edit several files at once. This section describes how to load several files into Emacs so you can edit them simultaneously, how to split the screen into multiple windows so you can see several files at once, and how to move back and forth among the windows on the screen.

Retrieving Additional Files

To retrieve another file so you can edit it without leaving the file you're currently editing, type Control x Control v. The system moves the cursor to the bottom of the screen and displays the following prompt:

```
: * visit-file
```

Type the name of the file you want to edit. (If you want to create a new file, you can do so by typing a new file name.) Emacs then displays the contents of the file in a buffer in the upper window. The file you were originally editing is no longer displayed in the upper window, but it is still within the editor and you can return to it at any time (see Exchanging the Buffer Being Displayed in a Window below.)

You can use the Control **x** Control v command to retrieve as many files as you wish.

Moving the Cursor from Window to Window

To move the cursor from one window to another, type Control x o (for other window). The system then moves the cursor to the next window on the screen.

Exchanging the Buffer Being Displayed in a Window

If you want to exchange the buffer that is being displayed in a window for a buffer containing another file that has already been loaded into Emacs, type Control **x b** (for change buffer). The system then displays the following message at the bottom of the screen:

Buffer: [<RETURN> for buffer buffername]

where buffername represents the buffer that was previously displayed in that window. (Recall that when you load a file into Emacs, that file is copied into a buffer that has the same name as the file itself.)

To request that buffer buffername be displayed in that window, press the Return key. To request that a different buffer be displayed in that window, type the buffer name (the name will be displayed at the bottom of the screen as you type it); then press the Return key. For example, if you want to exchange the buffer containing the current file for a buffer containing a file called File1, you would type File1 and press the Return key. The system would then exchange File1 for the file that was being displayed in the window.

NOTE: If you want to look at a file that you have not yet put into a buffer, you must type Control x Control v to retrieve the file. You can only use the Control **x b** command on files that you have already read into Emacs.

If you are not sure which buffer you want to display, type a question mark. Emacs then displays a list of the names of the buffers you can currently see. To display the contents of a buffer, type the name of the buffer, as described above.

Enlarging the Current Window

When your screen is split into two windows, you can enlarge either one of the windows if you wish. To enlarge a window, move the cursor to that window by typing **Control x o**. Then type **Escape e**. The window that the cursor is in expands to fill most of the screen; the other window is reduced to just one line.

To return the windows to their original sizes, type **Escape e** again.

If you want to enlarge a window and leave more than one line in the other window, you can tell Emacs how many lines to leave in the smaller window. To do that, move the cursor to the window you want to enlarge, type **Control u** followed by the number of lines you want to leave in the smaller window, and then type **Escape e**. For example, to enlarge one window and leave 5 lines in the other window, you would move the cursor to the window to be enlarged and then type

```
^u5 <esc>e
```

Dividing a Window

If you want to divide a window into two windows, move the cursor to the window and type **Control x 2**. Emacs then divides the window in half and displays the file that was displayed in the original window in both the new windows. You can then view different portions of the same file at the same time. If you want to look at two files simultaneously, replace the file in one of the windows with another file, as described above.

Once you have divided a window in half, you can divide it in half again by typing **Control x 2**.

NOTE: This command works only on the text window; it is not possible to divide the Prolog window in half.

Deleting a Text Window

To delete a text window, position the cursor in that window and type **Control x d**. The system then deletes that window. Note that one text window must always be present; however, you can control the size of that window by using the **Escape e** command discussed above.

Returning to One Editor Window

To return the screen to a two-window display (with one text window on the top and Prolog on the bottom), move the cursor to the text window you want to keep. Then type Control **x** 1. Your screen then returns to the standard Emacs/Prolog two-window display.

As before, even though the other buffers are no longer displayed on the screen, they are still within the editor; you can gain access to them by typing Control **x** b.

Obtaining a List of Files Currently Being Edited

If you are working with multiple windows and multiple files, you can sometimes lose track of the files you are working on. To obtain a list of the files that you are currently editing, type Control **x** Control **b** (for buffer). Emacs then displays a list like the following:

```
Size  Type  Buffer           Mode        File
   0   Scr  Buffer list      Normal
 374  File  file1            Prolog      /ufs/clark/file1
 502  File  file2            Prolog      /ufs/clark/file2
```

The complete names of the files are listed under the heading File; the names you call the files by while you're editing them are listed under the heading Buffer.

As explained above, you can request that the buffer currently being displayed in a window be replaced with another buffer. To do that, you need to know the name of the buffer that should replace the current buffer. You can get that name by referring to the buffer list, as shown above.

To return to the screen that was being displayed when you requested the list of buffers, press the spacebar.

USING THE EMACS HELP FUNCTION

The Emacs commands described above are the basic commands you will use most often. Emacs has a number of additional commands that are not described here because they are not necessary for basic editing. However, Emacs has an on-line help facility that lists information about most Emacs commands. To obtain information about an Emacs command, type Escape ? (that is, Escape question mark). Emacs responds with the message

```
: * apropos:
```

Type the name of the function you want to obtain information about. For example, if you want to obtain information about deletion commands, type the word delete. Emacs would then display a list of all the commands whose names included the word delete.

To obtain information about a specific command, type **Escape x** followed by the phrase **describe-command** and the command name. For example, to obtain information about the delete-next-word command, you would type **Escape x**. Emacs would respond by displaying the prompt -> as shown below. At the prompt ->, you would type the name of the command you wanted information about.

```
<esc>x
->describe-command delete-next-word
```

SECTION 3

LOADING PROGRAMS INTO PROLOG

This section explains how to load programs into Prolog. Included are a description of the Quintus Prolog interpreter and compiler, an explanation of how to load programs into Prolog through both the interpreter and the compiler, how to load programs through Emacs, and how to load programs without Emacs. The section also describes the features of Quintus Prolog that automatically check your program for correctness of syntax and adherence to style conventions.

LOADING A FILE INTO PROLOG

You can load a program into Prolog by using either the interpreter or the compiler. Usually, you will use the interpreter. Interpreted code loads more quickly than compiled code and is easier to debug. Compiled code runs more quickly than interpreted code; but since it is harder to debug, you will usually compile programs only after they are thoroughly debugged. You can freely mix interpreted and compiled code, so you might often find it convenient to compile portions of your programs that have been debugged and interpret portions that you are still working on.

To use the interpreter, you use the consult predicate (see page 25); to compile programs, you use the compile predicate (see page 29). However, if you are using the Emacs/Prolog interface, you will often find it easier to load programs through Emacs, as described below. By using Emacs, you can invoke the interpreter or compiler from your Emacs text window and load as much or as little of your program as you like.

LOADING PROGRAMS THROUGH THE EMACS INTERFACE

To use the interpreter to load a program from Emacs into Prolog, go into the Emacs/Prolog interface by typing prolog + at the main Unix prompt. When the Emacs/Prolog screen appears, type Control x Control v 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; see page 3). After you enter the Emacs/Prolog environment, move the cursor to the window containing your file.

At this point, you have three options: you can load the entire buffer containing your file, you can load a designated portion of the program in your file, or you can load a single procedure.

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 make the program work better. You can make your changes and then reload just the changed portions, as described below, 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.

To load the entire buffer, type **Escape** i (for interpret). Prolog then displays the following prompt line at the bottom of the screen:

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

Type **b** (for buffer).

To load a portion of your program, mark the region you want to load. To do that, move the cursor to the beginning of the first line you want to load. Then type Control **@** (that is, Control Shift 2 on many terminals). Prolog 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 **Escape i**. When Prolog displays the following line at the bottom of the screen, type r for region.

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

NOTE: When you mark the region to be loaded by the **Escape i r** command, be sure to include all the clauses for any procedures you are loading.

To load a single procedure, move the cursor to any portion of any line within the procedure. Then type **Escape i**. When Prolog displays the following line at the bottom of the screen, type **p** (for procedure).

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

NOTE: You can load an individual command of the form :- <u>command</u> into Prolog by positioning the cursor on the line that contains the command and then typing Escape i followed by **p**.

The **Escape i p** facility requires that you use certain syntactic and structural conventions which are described in Section 6, page 73. If you are not sure whether your procedures adhere to these conventions, you should use the **Escape i r** facility instead, making sure your marked region encompasses these procedures in their entirety.

After you indicate how much of your program to load, the cursor moves to the Prolog window at the bottom of the screen, and Prolog displays a message that tells you it is loading the program. When it finishes, Prolog displays a message to let you know that the procedures have been successfully loaded. For example, if your program consisted of the procedures parts_of, assembly, and inventory, Prolog would display the message

```
/* Procedure parts_of/2 consulted */
/* Procedure assembly/2 consulted */
/* Procedure inventory/2 consulted */
```

After it lists the procedures it has loaded, Prolog displays the message "yes" followed by the main Prolog prompt, as shown below.

```
yes
| ?-
```

At this point, you can begin to run your program (see Section 4, page 35.)

NOTE: When you load procedures into Prolog, Prolog first removes any previous versions of those procedures from its database.

Moving Between the Text and Prolog Windows

As mentioned earlier, you can easily move back and forth between the text and Prolog windows, make changes to your program, and then reload the modified portion of the program, as described above.

Recall that to move from the text window into Prolog, or vice versa, you type Control **x** followed by **o**. The system then moves the cursor into the other window on the screen, and you can immediately begin working in that environment.

LOADING PROGRAMS WITHOUT EMACS

If you have created a Prolog program and stored it in a file, you load that file into Prolog through the interpreter by directing Prolog to consult the file.

When you consult a file, Prolog adds the procedures defined in the file to the Prolog database, after first deleting any previous version of those procedures from the database.

You can consult a file in two ways: you can type the name of the file enclosed in square brackets at the main Prolog prompt, as shown in the example below;

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

or you can type consult followed by the name of the file containing the program at the main Prolog prompt, as shown in the example below.

```
| ?- consult('myfile.pl').
```

Note that in either case, the command must be followed by a period. The single quotes are necessary because the file name contains a non-alphanumeric character (.). Usually, you can omit the .pl suffix. For example, you could simply type

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

or

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

Prolog would then look for a file called myfile.pl; if none existed, it would look for one called myfile.

When Prolog finishes consulting a file, it displays a message which shows the name of the file that was consulted and the length of time it took to consult the file.

```
[myfile.pl consulted (2.354 sec 2346 bytes)]
| ?-
```

As shown above, the main Prolog prompt reappears after the system finishes consulting a file. At this point, you can begin using your Prolog program, as described on page 35.

If you want to consult several files at once, enclose the list of file names within square brackets, and separate the file names by commas. For example, to consult three files called file1, file2, and file3, you would type

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

Alternatively, you could type

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

USING IMBEDDED CONSULT COMMANDS

Any Prolog file can contain imbedded commands which cause Prolog to consult subsidiary source files whenever it consults the primary source file. To direct Prolog to consult a subsidiary source file, you include the :- consult(<u>filename</u>). command in your primary source file. For example, to direct Prolog to consult a subsidiary file called myfile, you would include the following line of text in your primary source file:

```
:- consult(myfile).
```

NOTE: The :- symbol is placed at the beginning of the line just as it appears in the example above.

SYNTAX ERROR MESSAGES

When a file is read into Prolog, Prolog automatically checks the clauses in the file for correctness of syntax. If a clause contains a syntax error, Prolog tells you that a syntax error has been found and displays the clause that contains the error.

For example, suppose you accidentally omitted a closing parenthesis in a clause, as shown below.

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

When you directed Prolog to load the file containing that clause, Prolog would load all the clauses that preceded the clause containing the error. When it reached that clause, it would display the message

```
** Syntax error:  **
member(X,[a,b,c,d]
** here **
```

to let you know that the syntax of the clause was incorrect. Prolog would then ignore this clause and continue loading the rest of the file into the database.

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 which cannot be read in to Prolog, style warnings simply indicate typing mistakes or program construction that doesn't follow Quintus Prolog style conventions. The style conventions for Quintus Prolog are listed below. By adhering 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 consult and compile predicates do not allow the definition of a procedure to be spread across more than one file. If a procedure is defined in more than one file, and all the files in which the procedure is defined are consulted or compiled, each successive definition of the procedure will wipe out any clauses for the procedure which were previously defined.

2. Make all clauses for a given procedure contiguous in the source file. This does not mean that you shouldn't leave blank space or put comments between clauses if you want to, but simply that clauses for one procedure should not be interspersed with clauses from another procedure.

3. If a variable appears only once in a clause, write that variable as either the single character "_", 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 clauses is consulted or compiled. For example, if condition 1 is violated, Prolog displays a message like the one shown below before it consults or compiles a procedure that has been defined in another file that has already been consulted or compiled:

```
The procedure foo/2, previously defined in
/ufs/george/file1, is being redefined by /ufs/george/file2.
Do you really want to redefine it? (Y, N, P, or ?)
```

If you type **y**, the definition in the file being currently consulted replaces the existing definition for the predicate. If you type n, the existing definition remains intact, and the definition in the file being currently consulted or compiled is ignored. If you type p, the definition in the file being currently consulted or compiled replaces the existing definition; and furthermore, if yet another definition of that procedure is encountered in another file, that definition will automatically replace the existing definition and no warning message will be displayed. This option is particularly useful if you have changed the name of a file, since it suppresses the warnings you would otherwise get for every procedure in the file.

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

```
[Warning: Clauses for foo/1 are not together in the source file]
```

This indicates that in between some pair of clauses defining procedure foo/1, 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 clause 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).
```

If style convention 3 is violated, you will get a message of the form:

```
[Warning: 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 or reconsult. 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 immediately respond 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.

The following predicates can be typed at the main Prolog prompt to turn the style warning facility on and off:

style_check(X)

X = all	turns on all style checking (this is the default)
X = single_var	turns on checking for single variable occurrences
X = discontiguous	turns on checking for discontiguous clauses for procedures
X = multiple	turns on style checking for multiple definition of same procedures (in different files)

no_style_check(X)

X = all	turns off all style checking
X = single_var	turns off checking for single variable occurrences
X = discontiguous	turns off checking for discontiguous clauses for procedures
X = multiple	turns off style checking for multiple definition of same procedures (in different files)

COMPILING PROGRAMS THROUGH THE EMACS INTERFACE

When your program is completely debugged and is in its final form, you may want to compile it so it will run faster. You invoke the compiler from Emacs almost the same way as you do the interpreter. The only difference is that you type **Escape k** rather than Escape i.

If you want to compile an entire file, position the cursor anywhere within the file. Then type **Escape k** (for kompile). When Prolog displays the following line at the bottom of the screen, type b (for buffer).

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

To compile a region, mark the region (see page 14). Then type Escape k r.

To compile a single procedure, position the cursor anywhere within the procedure. Then type **Escape k p.**

After you indicate how much of the program to compile, Prolog compiles the program, lists the procedures it has compiled, and then displays the main Prolog prompt.

NOTE: When you compile procedures, Prolog first removes any previous versions of those procedures from the database.

COMPILING PROGRAMS WITHOUT EMACS

To compile a program from within the Prolog environment, type compile followed by the name of the file containing the program at the main Prolog prompt, as shown in the example below.

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

Remember that the command must be followed by a period.

Prolog compiles the file and then displays a message like the following:

```
[myfile.pl compiled (4.918 sec 2346 bytes)]

yes
| ?-
```

At this point, you can begin using your Prolog program, as described on page 35.

If you want to compile several files at once, enclose the list of file names within square brackets, and separate the file names by commas. For example, to compile three files called file1, file2, and file3, you would type

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

NOTE: In any Prolog source files, you can imbed compile commands which direct Prolog to automatically compile subsidiary source files whenenver the primary source file is compiled. To do that, include the :-compile(filename). command in your primary source file. For example, to direct Prolog to automatically compile a subsidiary file called myfile, you would include the following line in your primary source file:

```
:- compile(myfile).
```

DEFINING PROCEDURES DIRECTLY

If you want to define Prolog procedures directly without first typing them in a file, you can easily do so. Note, however, that this method of defining procedures is usually only used if you want to enter just a few clauses. If you are not using the editor interface, Prolog does not provide any editing capability, so this method is not recommended for entering procedures of any size.

To enter a procedure directly into the database through the interpreter, type either [user]. or consult(user). at the main Prolog prompt.

```
| ?- [user].
```

or

```
| ?- consult(user).
```

Note that in either case, the command must be followed by a period. Type all the clauses for the procedures you want to enter. When you are finished, type Control **x** Control **d** to return to the main Prolog prompt. (If you are not using the Emacs/Prolog interface, type Control **d** instead.) The system displays a message telling you it has consulted your clauses and telling you how long it took to consult the clauses. A sample session might look like this:

```
| ?- consult(user).
parts_of(transmission,gears).
parts_of(transmission,housing).
parts_of(transmission,shaft).
^x^d
[user consulted (0.100 sec 270 bytes)]

yes
```

```
| ?-
```

If you are using Prolog without the Emacs interface and you type consult(user)., the system displays a | prompt at the beginning of each line that follows the line on which you typed consult(user). When you finish typing in your procedures and you type Control d, the main Prolog prompt | ?- returns.

To enter a procedure directly into the database through the compiler, type compile(user). at the main Prolog prompt.

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

As noted above, you must type a period after the command. Type the clauses for the procedures you want to enter. When you finish, type Control x Control d to return to the main Prolog prompt. (If you are not using the Emacs/Prolog interface, type Control d instead.)

NOTE: When you define a procedure directly, the procedure replaces any procedures of the same name that were previously defined in the database.

SAVING A PROGRAM STATE

Once a program has been consulted or compiled, its facts and rules are resident in the Prolog database. It is possible to save the current state of the database in its consulted or compiled form. This allows you to restore the current database at a later time without having to reconsult or recompile a Prolog source file.

To save a program state in a file, type the following at the main Prolog prompt: save(<u>filename</u>). For example, to save a program state in a file called myprog, you would type

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

To restore, or retrieve, a saved program state from the Unix level, type the name of the file containing the saved state at the main Unix prompt. For example, to restore the program state stored in the file myprog, you could type myprog at the main Unix prompt, as shown below.

```
<Unix prompt>myprog
```

Restoring a Saved Program State at the Unix Level

To restore a saved program state at the Unix level, type the name of the file containing the saved state at the main Unix prompt. For example, to restore the program state stored in the file myprog, you could type myprog at the main Unix prompt, as shown below.

```
<Unix prompt>myprog
```

Restoring a Saved Program State and Emacs

If you want to restore a program state under the Emacs interface, type the name of the file containing the program state followed by a plus sign (+) at the main Unix prompt. For example, to restore a program state stored in the file myprog, you would type **myprog +** at the main Unix prompt, as shown below.

```
<Unix prompt>myprog +
```

The system would then divide the screen into two windows with a text window in the upper window and the program state myprog in the lower window.

Restoring a Saved Program State and Its Source File

If you want to restore a program state and load its source file into Emacs, type the name of the file containing the program state, a plus sign (+), and the name of the source file at the main Unix prompt. For example, if you had a source file called mysource, and if you had stored its program state in a file called myprog, you could load the source file and restore the program state by typing the following at the main Unix prompt:

```
<Unix prompt>myprog + mysource
```

The system would then split the screen into two windows; the file mysource would be loaded through Emacs into the upper window, and the program state stored in myprog would be loaded into Prolog in the lower window.

Restoring a Saved Program State from within Prolog

To retrieve a saved program state from within Prolog, type restore(<u>filename</u>). at the main Prolog prompt. So to restore the state stored in the file myprog, you would type

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

When you restore a saved program, the program will be loaded into Prolog in exactly the same state it was in when you saved it in the file. For example, if you saved a program by giving Prolog the command

```
| ?- save(myprog), write('myprog restored').
```

the message **myprog restored** would be displayed on the screen when you directed Prolog to restore that file.

SAVING A PROLOG SESSION

If you are running Prolog under the Emacs interface, you can save your current Prolog session in a file so you can review or print it.

The Prolog window is automatically associated with the file prolog.log in your home directory when you start Prolog by typing **prolog +**. To save your Prolog session, which is contained in an edit buffer, type **Control x Control s** in the Prolog window. The system then saves your session in the file prolog.log.

NOTE: Saving a Prolog session is different than saving a Prolog program state. A Prolog session saved in prolog.log can be displayed or printed, but it cannot be restored or started up again. In contrast, a saved state can be restored, and you can resume your Prolog session at the point where you left off.

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 that by using the initialization file **prolog.ini**.

To direct the system to automatically load the files containing your customized features, put commands in prolog.ini to consult those files (see page 26.) Since **prolog.ini** is consulted whenever you start up Prolog, any commands contained in the file will be automatically executed.

NOTE: **prolog.ini** is resident in your home directory.

SECTION 4

RUNNING PROGRAMS

This section describes how to run a program once you've loaded it into Prolog. The section also contains a discussion of certain features of Quintus Prolog that you will find it helpful to know about when you run your programs.

ASKING QUESTIONS

Suppose your program contained the following clauses:

```
parts_of(transmission,gears).
parts_of(transmission,housing).
parts_of(transmission,shaft).
```

You could ask the question "Are gears part of the transmission?" by typing

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

Prolog would attempt to satisfy this goal; and if it succeeded, it would display the word yes and then redisplay the main Prolog prompt, as shown below.

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

yes
| ?-
```

If Prolog had not found clauses to satisfy the goal, it would display the word no and then redisplay the main Prolog prompt.

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

no
| ?-
```

NOTE: Like clauses and commands, questions must end with a period.

As with Prolog clauses, you can use variables in place of constants in your questions. For example, to ask the question "What are the parts of the transmission?" you could type

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

X = gears
```

Prolog displays the first valid value it finds for X, as shown above.

If the first value is sufficient — that is, if you don't want Prolog to continue searching for other answers — type a carriage return. Prolog then returns you to the main Prolog prompt.

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

X = gears <ret>

| ?-
```

If you want Prolog to continue searching for valid answers, type a semicolon and a carriage return. (Recall that in Prolog, a semicolon means or. So by typing a semicolon, you direct Prolog to search for an alternative answer.) For example, to see if the transmission contains any parts other than gears, you would type a semicolon after Prolog displayed the first answer.

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

X = gears ;

X = housing
```

You can request as many additional answers as you like by typing a semicolon each time Prolog displays an answer. Prolog will continue to search for additional answers until it reaches the last valid answer in your program. If you type a semicolon and there are no more valid answers, Prolog simply displays the word no and returns to the main Prolog prompt, as shown below.

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

X = gears ;

X = housing ;

X = shaft ;

no
| ?-
```

REPEATING A QUESTION (USING EMACS)

Often during your Prolog sessions you might find it useful to submit a question, edit it slightly, and then resubmit it. For example, if you make a typing error in a question, you would want to correct the error and resubmit the question.

Everything you type during a Prolog session goes into a buffer, so it is easy to retrieve and copy lines you've already typed. To copy the last question you typed so you can edit it, type Control x Control e at the main Prolog prompt, | ?-. Prolog then redisplays the question.

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

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

no

| ?- ^x^e parts_of(transmissiom,X).
```

When you type Control x Control e, the last input string you typed is displayed on the screen.

If you want to redisplay an earlier input string, move the cursor to the line you want to copy and then type Control x Control e.

DISPLAYING PREVIOUS INPUT USING EMACS

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 (see page 13). Similarly, you can scroll forward to your current step in your Prolog session by using Emacs scrolling commands.

LOCATING PROCEDURES USING EMACS

The Emacs/Prolog interface also 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 particularly helpful to be able to locate a procedure directly without having to search through several files.

You can direct Prolog to locate a procedure in one of two ways. If you have already typed the procedure as part of a question you submitted to Prolog, move the cursor to the line containing the procedure. Then type Control x . (that is, Control x period). Prolog will then locate the procedure and display the portion of the file containing the procedure in the text window at the top of the screen.

Alternatively, you can type Control x . (that is, Control x period) at the main Prolog prompt. The cursor moves to the bottom of the screen, and the system displays the message

```
name/arity =
```

Type the name of the predicate you want to locate followed by a slash and the arity of the predicate. (Recall that the arity is the number of arguments the predicate has). For example, to locate the predicate employee(smith,harold), you would type employee/2, as shown below.

```
name/arity = employee/2
```

NOTE: You can type the predicate name without typing the arity, and the system will still locate the predicate. If the predicate was defined for more than one arity, the system will simply locate one of the definitions of the predicate.

INTERRUPTING THE EXECUTION OF A PROGRAM

You can interrupt the execution of a Prolog program at any time by typing Control c. For example, if you submit a question to Prolog and then decide you want to stop (abort) the question, you would type Control c, and Prolog would 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 type the letter that corresponds to the option you want to select.

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

```
Interrupt Options:

    c    continue - do nothing
    t    trace    - debugger will start creeping
    d    debug    - debugger will start leaping
    a    abort    - cause a Prolog abort
    e    exit     - irreversible exit from Prolog
    h    help     - this list

Prolog interruption (h for help)?
```

To select an option, type the letter that corresponds to that option and press the Return key. For example, to stop the execution of the current question, type **a** followed by a carriage return.

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 page 43). Typing d turns on the debug option of the debugger (see page 43). Typing **a** causes the current question to be aborted and the main Prolog prompt to be redisplayed. Typing **e** ends your Prolog session. (However, if you are using Prolog through the Emacs interface, you must still type Escape Control c to exit to the Unix level.)

ERROR MESSAGES FOR BUILT-IN PREDICATES

If your program uses a built-in predicate with arguments that are not appropriate for that predicate, the system may display an error message. After the message is displayed, one of three things happens, depending upon the severity of the error. The predicate succeeds, it fails, or it causes the program to be aborted. An example of each type of error message is shown below. See the Quintus Prolog Reference Manual for more detailed information.

```
[ Error 321: op(-1,xfx,foo) - illegal operator precedence ]
```

This message would be displayed if your program called the goal op(-1,xfx,foo).; however, the goal would still succeed.

```
[ Error 302: arithmetic expression contains a variable: _97 ]
```

If your program contained the goal **X is Y**, the goal would fail, and the error message shown above would be displayed.

```
[ Error 109: cannot create saved state (UNIX error foo) ]
[ Execution aborted ]
```

If the goal **save(foo).** was called and the file foo was protected, the system would display the error message shown and abort the program.

UNDEFINED PROCEDURES

By default, if Prolog encounters a call to an undefined procedure, it displays an error message, turns on the debugger, and begins tracing. (See page 43 for more information about tracing.)

Once an error message has been displayed, you can take one of several actions. You can turn the debugger off and continue the execution of the program by typing **n**. If you do this, the undefined procedure will fail. Alternatively, you can abort the execution of the program entirely by typing **a**. If you think the procedure was called by mistake, type g to find out where it was called from, and then correct the procedure that called it. If the procedure should have been defined, find or write its definition and then consult or compile it; then resubmit your original question or command. If the procedure should be undefined, make it a dynamic procedure (see page 41), or else define a clause for it with the body "fail."

NOTE: If you would prefer that calls to undefined procedures would simply fail (rather than activating the debugger), use the built-in predicate **unknown** to change the default action for calls to undefined procedures. (See the Quintus Prolog Reference Manual for more information.)

RESTARTING PROLOG (UNDER EMACS)

If you are running Prolog under the Emacs interface, you may sometimes find it convenient to halt the Prolog session that is currently running without halting the Emacs session, and to restart Prolog by calling a Prolog saved state. To halt the current Prolog session, type Control c followed by e.

To restart a Prolog session under Emacs, type Escape **x**. When the cursor moves to the bottom of the screen, type restart-prolog, as shown below.

```
-> restart-prolog
```

The system then displays the following line at the bottom of the screen:

```
Prolog save state: [<RETURN> for previous save state]
```

To restart the Prolog program you just halted, press the Return key. To run a different saved state, type the name of the file that contains that program. In either case, the system responds by restarting Prolog in the lower window.

EXECUTING UNIX COMMANDS FROM WITHIN PROLOG

For convenience, Quintus Prolog provides two commands that enable you to execute Unix functions from within the Prolog environment. These commands enable you to gain access to the main level of Unix and to change your directory.

To gain access to the main level of Unix from within Prolog, type unix(shell). at the main Prolog prompt, as shown below:

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

This command puts you at the main Unix shell, and you can execute any commands you would normally execute at that level. To return to the main Prolog prompt from this point, type Control x Control d. (If you are using Prolog without the Emacs interface, type Control d.)

Alternatively, you can gain access to the Unix shell and execute a command all at once. To do that, type the following at the main Prolog prompt:

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

where Atom represents the command you want to execute. For example, to obtain a listing of the files in your directory, you would type

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

To change your default directory, type unix(cd(Atom)). at the main Prolog prompt, as shown

```
| ?- unix(cd(Atom)).
```

where Atom represents the directory you want to change to. For example, to change to a directory named /ufs/albert, you would type

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

The directory name /ufs/albert/ is surrounded by single quotes because it contains non-alphanumeric characters.

The system then displays a message at the bottom of the screen showing you the directory it has changed to:

```
have now changed to directory /ufs/albert
```

It then redisplays the main Prolog prompt.

To change back to your default directory, you can either type unix(cd(directoryname)). followed by the name of your directory, as described above, or you can simply type **unix(cd).** at the main Prolog prompt. As before, the system displays a message at the bottom of the screen showing you the directory it has changed to and then redisplays the main Prolog prompt.

NOTE: Alternatively, you can change your directory by typing **Escape x**; when the cursor moves to the bottom of the screen, type cd. This method can be easier to use if your file has a long name because with this method, you can use Emacs file-name completion. See page 12 for more information on file-name completion.

CREATING 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 using consult or compile.

If a predicate is first defined by being consulted or compiled, it is static by default. Sometimes, however, it is necessary to add, remove, or inspect 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 only by specifically declaring it to be so, as described below, or by using one of the **assert** predicates (see Appendix D for a list of the assert predicates. For more information, refer to the Quintus Prolog Reference Manual).

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

:- dynamic name/arity

So, for example, to make the predicates exchange_rate/3, spouse_of/2, and gravitational_constant/1 dynamic, you would type

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

Any number of predicate names may be specified, as long as they are separated by commas. Note that the :- symbol must precede the line with the dynamic declaration, as shown above.

NOTE: The line that declares a predicate to be dynamic must occur before the definition of the predicate itself in the file.

Alternatively, you can declare a previously undefined predicate to be dynamic by using one of the assert predicates listed in Appendix D.

NOTE: If you want to use one of the assert predicates to define a new clause for an existing predicate, you must first make the predicate for that clause dynamic. That is, you must add the dynamic declaration to the definition of the procedure and consult or compile it again. If you don't, you will receive an error message like the one shown in the following example:

```
| ?- assert((f(x):-g)).
[ Error 330: attempting to assert clause for static procedure: f(x):-g]
```

PROMPTS

If you are using Prolog under the Emacs interface, the only Prolog prompt you will see is the main Prolog prompt, | ?- . However, if you are using Prolog without the Emacs interface, you will see other Prolog prompts, as described below.

The prompt |: is displayed instead of the | ?- prompt if you type a command or if your program contains a command that requires input from the terminal. For information on changing the |: prompt, refer to the definition of **prompt** in the Quintus Prolog Reference Manual.

The prompt | is a modified version of the Prolog top level prompt. If you are typing a question at the top level of Prolog, and your input is longer than one line, the | prompt is displayed on all lines after the first one. The | prompt is another modified version of the Prolog top level prompt which is displayed at the beginning of lines when you type [user] or compile(user) to enter procedures directly into Prolog.

SECTION 5

DEBUGGING YOUR PROGRAMS

This section explains the debugging facilities that are available in Quintus Prolog. The debugging facilities provide you with information concerning the control flow of your program so you can find problem areas.

With the debugger, you can exhaustively trace each step of your program; or, if you want less detail, you can set spypoints on selected procedures and monitor the execution of those procedures only. A wide range of control and information options available during debugging enable you to further tailor the debugging process to meet your needs.

BACKGROUND INFORMATION

The debugger can trace program execution at four key points:

-- procedure entry, when a solution to a goal is first looked for
-- procedure exit, when a solution to a goal is found
-- procedure re-entry, when an alternative solution is looked for, and a procedure is reentered during backtracking
-- procedure failure, when no (more) solutions are to be found for a goal, and the procedure is exited.

As the debugger proceeds through each step of the execution of a procedure, it displays a message letting you know which of these four points it is currently passing through. These entry and exit points are called ports, and they are referred to by the debugger as follows:

Port	Description
Call	Look for a solution to the goal
Exit	Exit because a solution has been found
Redo	Look for alternative solutions (backtrack)
Fail	Exit because there are no (more) solutions

TURNING THE DEBUGGER ON

To turn the debugger on, type either trace or debug at the main Prolog prompt. If you type trace, Prolog will start by showing the step-by-step execution of the procedure you specify. If you type debug, you set the debugger up to start showing selected portions of the execution of the procedures you specify. The option you select sets the "top-level" debugging mode, but you can select more or less detailed debugging any time the debugger stops and prompts you for interaction.

Once the debugger is on, a message is continually displayed to show which option (trace or debug) you selected. Under Emacs, the debugger state is displayed on the Prolog mode line at the bottom of the screen:

```
------------------------------------------------------------

¦ ?-


------------------------------------------------------------
Quintus Prolog                                    trace
------------------------------------------------------------
```

When you are not under Emacs, the debugger state is shown before each top-level prompt, as shown below:

```
------------------------------------------------------------

[trace]
¦ ?-

------------------------------------------------------------
```

As shown in the example below, you can turn the debugger on by typing trace or **debug** at the main Prolog prompt.

```
------------------------------------------------------------
¦ ?- trace.
[The debugger will first creep -- showing everything (trace)]

yes
¦ ?-
------------------------------------------------------------
```

or:

```
------------------------------------------------------------
¦ ?- debug.
[The debugger will first leap -- showing spypoints (debug)]

yes
¦ ?-
------------------------------------------------------------
```

TURNING THE DEBUGGER OFF

To turn the debugger off, type either nodebug or notrace. These predicates have exactly the same effect.

```
---------------------------------------------------------------
| ?- nodebug.
[The debugger is now switched off]

yes
| ?-
---------------------------------------------------------------
```

or:

```
---------------------------------------------------------------
| ?- notrace.
[The debugger is now switched off]

yes
| ?-
---------------------------------------------------------------
```

TRACING EVERY STEP IN THE EXECUTION OF A PROCEDURE

Once you type trace. to turn on the debugger, you can begin tracing the execution of a procedure by simply typing the procedure at the main Prolog prompt. For example, to trace the execution of a procedure works_for(X,Y), you would type that procedure at the main Prolog prompt, as shown below:

```
| ?- works_for(X,Y).
```

The system would then begin to execute the procedure in the normal way (variables would be instantiated, backtracking would occur when necessary, etc.). But because the debugger is activated, the system also shows when each procedure is called and whether or not it is successfully executed.

A sample trace is shown below for the procedure works_for(X,Y). The procedure is defined as follows:

```
works_for(X,Y) :- works_directly_for(X,Y).
works_for(X,Z) :- works_directly_for(X,Y), works_for(Y,Z).

works_directly_for(john,mary).
works_directly_for(mary,george).
```

```
   -----------------------------------------------------------------

| ?- works_for(X,Y).
   (1) 0 Call: works_for(_93,_108) ? <ret>
   (2) 1 Call: works_directly_for(_93,_108) ? <ret>
   (2) 1 Exit: works_directly_for(john,mary)
   (1) 0 Exit: works_for(john,mary)

X = john,
Y = mary;

   (1) 0 Redo: works_for(john,mary) ? <ret>
   (2) 1 Redo: works_directly_for(john,mary) ? <ret>
   (2) 1 Exit: works_directly_for(mary,george)
   (1) 0 Exit: works_for(mary,george)

X = mary,
Y = george ;
   (1) 0 Redo: works_for(mary,george) ? <ret>
   (2) 1 Redo: works_directly_for(mary,george) ? <ret>
   (2) 1 Fail: works_directly_for(_93,_108)
   (3) 1 Call: works_directly_for(_93,_228) ? <ret>
   (3) 1 Exit: works_directly_for(john,mary)
   (4) 1 Call: works_for(mary,_108) ? <ret>
   (5) 2 Call: works_directly_for(mary,_108) ? <ret>
   (5) 2 Exit: works_directly_for(mary,george)
   (4) 1 Exit: works_for(mary,george)
   (1) 0 Exit: works_for(john,george)

X = john,
Y = george <ret>

| ?-
-----------------------------------------------------------------
```

As shown above, when you type a question, the system displays a message telling you that it is calling the procedure you've specified.

```
| ?- (1) 0 Call: works_for(_107,_122) ?
```

To direct the debugger to execute the call to the specified procedure and to show the program step that immediately follows that call, press the Return key. The system then executes the call to that procedure and displays the next program step. You can direct the system to continue displaying each successive step in the execution of the program by simply pressing the Return key each time the system displays a procedure call followed by a question mark.

If you want to see the final result of the question but you do not want the debugger to continue displaying each step of the execution, type n in response to the question mark. If you want to cancel the trace, type a (for abort) in response to the question mark.

As the debugger displays the program's execution, check for instances where procedures behave differently than you expect. Check especially for procedure calls that fail and variables that aren't instantiated as you expect them to be.

DEBUGGING ONLY SELECTED PROCEDURES

To debug only selected procedures in a program, type **debug** at the main Prolog prompt. Then set a spypoint on each of the procedures you want to debug. A spypoint is a type of marker. By setting a spypoint on a procedure, you direct the debugger to skip directly to that procedure and display debugging information for that procedure only.

Setting Spypoints

To set a spypoint, type spy followed by the name and arity of the predicate you want to examine, as shown in the example below. Note that parentheses are not necessary (although they can be used if desired).

```
---------------------------------------------------------------
| ?- spy test/1.
[The debugger will first leap -- showing spypoints (debug)]
[Spypoint placed on test/1]

yes
| ?-
---------------------------------------------------------------
```

NOTE: you can set a spypoint any time the debugger is on, whether it is in trace or **debug** mode.

To set several spypoints at once, type a list of the names and arities of the predicates you want to examine, as shown in the example below.

```
---------------------------------------------------------------
| ?- spy [transform/2, transform/3].
[Spypoint placed on transform/2]
[Spypoint placed on transform/3]

yes
| ?-
---------------------------------------------------------------
```

To set spypoints on ALL existing procedures of a given name, type spy followed by the name of the predicate you want to examine, as shown in the example below. This is a quick way of placing spypoints on procedures, particularly if you cannot remember their arities.

```
-------------------------------------------------------------
| ?- spy transform.
[Spypoint placed on transform/2]
[Spypoint placed on transform/3]

yes
| ?-
-------------------------------------------------------------
```

Debugging a Procedure on Which Spypoints Have Been Set

Once you have set spypoints, type the procedure you want to debug at the main Prolog prompt. The system then begins showing you program execution at the spypoints you've set. You can switch to more detailed debugging by pressing the Return key any time the system prompts you with a question mark.

The example shown below uses the same procedure that was used in the example on page 45. However, this time, a spypoint is placed on the procedure works_directly_for/2; so the call to that procedure is the first one displayed when you type the goal works_for(X,Y). Note also that in response to the question mark prompt, the letter l is typed instead of a carriage return. The l stands for "leap", which means "leap" to the next spypoint. Leaping is discussed in more detail in the next section.

```
-----------------------------------------------------------
| ?- spy works_directly_for/2.
[Spypoint placed on works_directly_for/2]

| ?- works_for(X,Y).
** (2) 1 Call: works_directly_for(_41,_56) ? l
** (2) 1 Exit: works_directly_for(john,mary) ? l

X = john,
Y = mary <ret>

| ?-
-----------------------------------------------------------
```

Removing Spypoints

To remove a spypoint, type nospy followed by the name and arity of the predicate you want to remove the spypoint from. As with spy, you can type nospy followed only by a predicate name if you want to remove spypoints from all the predicates of that name.

```
-----------------------------------------------------------
| ?- nospy translate/3.
[Warning: You have no clauses for translate/3]
[Spypoint removed from translate/3]

yes
| ?- nospy transform.
[Spypoint removed from transform/2]
[Spypoint removed from transform/3]

yes
| ?- nospy [append/3,test/1].
[Spypoint removed from append/3]
[Spypoint removed from test/1]

yes
| ?-
-----------------------------------------------------------
```

To remove all the spypoints at once, type nospyall.

```
-----------------------------------------------------------
| ?- nospyall.
[All spypoints removed]

yes
| ?-
-----------------------------------------------------------
```

REQUESTING MORE OR LESS DETAILED DEBUGGING

By default, the debugger stops at every port and displays the procedure being currently called followed by a question mark. The question mark is a debugger prompt which asks you to to specify the next action the debugger should take. Whenever the question mark is displayed, you can specify the degree of detail with which debugging should proceed. In other words, even if you selected the **debug** option originally, you can direct the debugger to begin debugging a procedure in detail any time the question mark is displayed. Conversely, if you originally selected the **trace** option, you can direct the debugger to begin displaying spypoints only.

To direct the debugger to proceed with detailed debugging or to begin detailed debugging, press the Return key in response to the question mark. Alternatively, you can type c, which stands for "creep;" "creeping" through a program is the same as tracing it, or displaying it step by step.

To direct the debugger to show the program's execution at the next spypoint, type l, which stands for "leap to the next spypoint." If you type l each time the debugger prompts you with a question mark, the debugger will display the program's execution at spypoints only.

To direct the debugger to skip over the execution of a procedure, type s for "skip." The debugger will not display anything during the execution of the current procedure; it will begin displaying debugging information again at the final Exit or Fail port of the current procedure. If any spypoints have been set on procedures called by the current procedure, they will be ignored.

NOTE: In addition to the options discussed above, you can select several other options when the debugger stops and prompts you. These options are discussed on page 63.

REQUESTING MORE OR LESS FREQUENT PROMPTING

By default, the debugger stops at all four ports (Call, Exit, Redo, and Fail) and prompts you to specify the next action that should be taken. The ports at which the debugger stops are referred to as "leashed" ports.

If you want to change the leashed ports -- that is, you want to request that the debugger stop and prompt you at only some of the ports rather than all of them -- you can easily do so by using the leash predicate. (See the Quintus Prolog Reference Manual for more information.)

NOTE: Leashing does not affect procedures on which spypoints have been set; the debugger always stops at all ports of procedures on which spypoints have been set.

OBTAINING INFORMATION ABOUT THE DEBUGGER AND SPYPOINTS

At any time, you can obtain a description of the state of the debugger and the spypoints that have been set by typing **debugging**. This also shows which debugging ports are leashed (that is, which ports the debugger will stop at) and the default action that will be taken if the system encounters an undefined procedure.

```
------------------------------------------------------------
| ?- debugging.
The debugger will first leap -- showing spypoints (debug)
Using leashing stopping at [call,exit,redo,fail] ports
Undefined procedures will trap to the debugger ('trace' option)
Spypoints:
    spypoint(test/1).
    spypoint(append/3).

yes
| ?-
------------------------------------------------------------
```

DEBUGGING EXAMPLE

Shown below is a sample program that contains a number of bugs. The program is briefly described, and then an annotated debugging session is shown to demonstrate in detail how the debugger is used to find the bugs.

The program is designed to find a day when people from different branch offices of a company can meet together in a certain city (in this case, Seattle). The program contains some basic information about which cities airlines fly between, flight times, whether or not flights are nonstop, etc. The program is based on two rules:

1) All the people who are attending the meeting have to travel on the same day to the city in which the meeting is being held.

2) The only way (in this program) to travel anywhere is to fly there.

The predicates contained in the program are as follows:

could_meet(List,Place,Day). The people listed in the List can all meet in a certain place on a certain day.

travel_ok(Day,City1,City2). It is possible to travel on the specified day from City1 to City2.

one_step(Day,City1,City2,direct(Flight)). There is a nonstop flight on the specified day from City1 to City2. (Flight represents the flight number.)

two_step(Day,City1,City2,change(Flight1,Flight2)). There is a flight with one stopover on the specified day from City1 to City2. (Flight1 and Flight2 represent flight numbers.)

connects(Flight1,Flight2). It is possible to fly on Flight1 and connect to Flight2.

meet_on(Day). It is possible for the meeting to be held on the specified day.

Following are the program and the database of flight information it uses. Note that this program is a relatively simple one, and that a more sophisticated program (which would allow for different modes of transportation, etc.) could easily be developed.

```
/* -------------------------------------------------------------------
      Database of flight information
   ------------------------------------------------------------------- */

flight( monday,  u100, los_angeles,   san_francisco ).
flight( monday,  u101, san_francisco, seattle ).
flight( tuesday, u102, los_angeles,   seattle ).


flight_time( u100, 0800, 0900 ).
flight_time( u101, 1000, 1200 ).
flight_time( u102, 1100, 1400 ).

/* -------------------------------------------------------------------
       Program for use by corporate planning officer to decide if a
       meeting can be arranged at one of their offices between people
       from a number of different offices. The principal predicate is:

         could_meet(Offices,MeetingOffice,Day)

             is true if people from the list of Offices could meet at
             the MeetingOffice on the day Day.
   ------------------------------------------------------------------- */

could_meet([],_,_).
could_meet([X|Rest],Place,Day) :-
       travel_ok(Day,X,Place),
       could_meet(Rest,Place,Day).

travel_ok(Day,From,To) :- one_step(Day,From,To,_).
travel_ok(Day,From,To) :- two_step(Day,From,To,_).

one_step(Day,From,To,direct(Flight)) :- flight(Day,Flight,From,To).

two_step(Day,From,To,change(Flight1,Flight2)) :-
       flight(Day,Flight1,From,Stopover),
       flight(Day,Flight2,StopOver,To),
       connects(Flight1,Flight2).

connects(Flight1,Flight2) :-
       flight_time(Flight1,_,End1),
       flight_time(Flight2,Start2,_),
       Start2 < End1.

/* -------------------------------------------------------------------
       User interface for a company with offices in san_francisco,
       los_angeles and seattle wishing to arrange meetings in seattle.
   ------------------------------------------------------------------- */

meet_on(Day) :-
       could_meet([san_francisco,los_angeles,seattle],Day).
```

The program contains four bugs: the variable Stopover, which is used in the procedure two_step, is spelled two ways; the call to could_meet in the rule for meet_on is missing an argument; the variable Start2 should be greater, not less, than End1; and the predicate travel_ok is missing a rule (it cannot cover the case of a person in Seattle who would not need to fly to attend a meeting in Seattle.)

A sample debugging session for this program is shown on the next few pages.

```
--------------------------------------------------------------------------------

Quintus Prolog Release 1.0 (Sun)
Copyright (C) 1985, Quintus Computer Systems, Inc.

| ?- [flight0].

[Warning: Singleton variables, clause 1 of two_step/4: Stopover, StopOver]
[flight0.pl consulted (1.200 sec 328 bytes)]

yes
| ?- [Consulting procedure...]
[ Procedure two_step/4 consulted. ]

yes
| ?- meet_on(When).

[Warning: The procedure could_meet/2 is undefined]
[However, could_meet/3 is defined]
   (1) 1 Fail: could_meet([san_francisco,los_angeles,seattle],_50) ? g

[Ancestors:]
   (-) 0 : meet_on(_50)

   (1) 1 Fail: could_meet([san_francisco,los_angeles,seattle],_50) ? a
[ Execution aborted ]


| ?- [Consulting procedure...]
[ Procedure meet_on/1 consulted. ]

yes
| ?- meet_on(When).

no
```

Using System Warnings

The program is in a file called flight0. When it is first consulted, the style checker catches a spelling error in the predicate **two_step**; (Stopover is spelled two ways). The spelling of StopOver is corrected, and that procedure is reloaded (using the Emacs command **Escape i p**.)

When the goal meet_on(When). is typed, another error is found: the predicate could_meet/2 is undefined. To locate the error, **g** is typed to find the procedure that called could_meet. The **g** option can be helpful in pinpointing how a particular goal is called; it is discussed further on page 65.

Next, **a** is typed to abort the execution of the procedure, and the missing argument (seattle) is added. The procedure is then reconsulted (through the Emacs interface), and the goal meet_on(When) is tried again; the goal fails again.

```
| ?- trace.
[The debugger will first creep -- showing everything (trace)]

yes
| ?- meet_on(When).
   (1) 0 Call: meet_on(_41) ? <ret>
   (2) 1 Call: could_meet([san_francisco,los_angeles,seattle],seattle,_41)?
<ret>
   (3) 2 Call: travel_ok(_41,san_francisco,seattle) ? <ret>
   (4) 3 Call: one_step(_41,san_francisco,seattle,_227) ? <ret>
   (5) 4 Call: flight(_41,_261,san_francisco,seattle) ? <ret>
   (5) 4 Exit: flight(monday,u101,san_francisco,seattle) ? <ret>
   (4) 3 Exit: one_step(monday,san_francisco,seattle,direct(u101)) ? <ret>
   (3) 2 Exit: travel_ok(monday,san_francisco,seattle) ? <ret>
   (6) 2 Call: could_meet([los_angeles,seattle],seattle,monday) ? <ret>
   (7) 3 Call: travel_ok(monday,los_angeles,seattle) ? <ret>
   (8) 4 Call: one_step(monday,los_angeles,seattle,_408) ? <ret>
   (9) 5 Call: flight(monday,_442,los_angeles,seattle) ? <ret>
   (9) 5 Fail: flight(monday,_442,los_angeles,seattle) ? <ret>
   (8) 4 Fail: one_step(monday,los_angeles,seattle,_408) ? <ret>
   (10) 4 Call: two_step(monday,los_angeles,seattle,_408) ? <ret>
   (11) 5 Call: flight(monday,_442,los_angeles,_454) ? <ret>
   (11) 5 Exit: flight(monday,u100,los_angeles,san_francisco) ? <ret>
   (12) 5 Call: flight(monday,_443,san_francisco,seattle) ? <ret>
   (12) 5 Exit: flight(monday,u101,san_francisco,seattle) ? <ret>
   (13) 5 Call: connects(u100,u101) ? <ret>
   (14) 6 Call: flight_time(u100,_586,_587) ? <ret>
   (14) 6 Exit: flight_time(u100,800,900) ? <ret>
   (15) 6 Call: flight_time(u101,_596,_597) ? <ret>
   (15) 6 Exit: flight_time(u101,1000,1200) ? <ret>
   (16) 6 Call (system): 1000<900 ? <ret>
   (16) 6 Fail (system): 1000<900 ? a
[ Execution aborted ]
```

Exhaustively Tracing the Execution of a Program

A detailed trace is begun to follow the execution of the program. Each time the system displays a procedure call followed by a question mark, the Return key is pressed to direct the system to display the next step in the execution of the program.

An error is found when the program checks to see if flight u100 could connect to flight u101. In order for the flights to connect, flight u101 must depart after flight u100 arrives; that is, the departure time (in hours) of flight u101 must be greater than the arrival time of flight u100. In the current program, the clause checks to see if the departure time of flight u101 is less than, rather than greater than, the arrival time of flight u100.

Once the error is found, a is typed to abort execution of the program.

Before correcting this error, we will look at another way of tracking down this error.

```
| ?- meet_on(When).
   (1) 0 Call: meet_on(_41) ? <ret>
   (2) 1 Call: could_meet([san_francisco,los_angeles,seattle],seattle,_41)?
<ret>
   (3) 2 Call: travel_ok(_41,san_francisco,seattle) ? s
 > (3) 2 Exit: travel_ok(monday,san_francisco,seattle) ? <ret>
   (6) 2 Call: could_meet([los_angeles,seattle],seattle,monday) ? <ret>
   (7) 3 Call: travel_ok(monday,los_angeles,seattle) ? s
 > (7) 3 Fail: travel_ok(monday,los_angeles,seattle) ? r
[Debugger: retry goal]

   (7) 3 Call: travel_ok(monday,los_angeles,seattle) ? <ret>
   (17) 4 Call: one_step(monday,los_angeles,seattle,_408) ? s
 > (17) 4 Fail: one_step(monday,los_angeles,seattle,_408) ? <ret>
   (19) 4 Call: two_step(monday,los_angeles,seattle,_408) ? <ret>
   (20) 5 Call: flight(monday,_442,los_angeles,_454) ? <ret>
   (20) 5 Exit: flight(monday,u100,los_angeles,san_francisco) ? <ret>
   (21) 5 Call: flight(monday,_443,san_francisco,seattle) ? <ret>
   (21) 5 Exit: flight(monday,u101,san_francisco,seattle) ? <ret>
   (22) 5 Call: connects(u100,u101) ? s
 > (22) 5 Fail: connects(u100,u101) ? r
[Debugger: retry goal]

   (22) 5 Call: connects(u100,u101) ? <ret>
   (26) 6 Call: flight_time(u100,_586,_587) ? <ret>
   (26) 6 Exit: flight_time(u100,800,900) ? <ret>
   (27) 6 Call: flight_time(u101,_596,_597) ? <ret>
   (27) 6 Exit: flight_time(u101,1000,1200) ? <ret>
   (28) 6 Call (system): 1000<900 ? a
[ Execution aborted ]

| ?- [Consulting procedure...]
[ Procedure connects/2 consulted. ]
yes

| ?- meet_on(When).
no
```

Using the Skip and Retry Options

This trace follows the execution of the same procedure as the one shown on the preceding page; however, here a slightly different and more efficient method of tracing was used. Rather than tracing every step in the execution of the program, a skip was done at selected points. If the procedure succeeded, we moved on to the next step in the trace; however, if the procedure failed, we did a retry of the procedure and then exhaustively traced the execution of the procedure. This method of debugging can help save time because it can help you quickly pinpoint the procedures that are failing; you can then trace those procedures in detail rather than exhaustively tracing all the procedures in the execution of the program.

Note that the procedure call to one_step failed (see the line beginning with > (17)); this failure is legitimate, as there was no direct flight from Los Angeles to Seattle on Monday. The symbol > at the beginning of the line simply signifies the completion of a skip command.

After the error is found, a is typed to abort the execution of the program. The error is then corrected (the last clause of connects(Flight1,Flight2) is changed to Start2 > End1.), and the procedure connects is reloaded through the Emacs interface. However, the program still doesn't work.

```
| ?- spy [travel_ok,connects].
[Spypoint placed on travel_ok/3]
[Spypoint placed on connects/2]

yes
| ?- debug.
[The debugger will first leap -- showing spypoints (debug)]

yes
| ?- debugging.
The debugger will first leap -- showing spypoints (debug)
Using leashing stopping at [call,exit,redo,fail] ports
Undefined procedures will trap to the debugger ('trace' option)
Spypoints:
    spypoint(travel_ok/3).
    spypoint(connects/2).

yes
| ?- meet_on(When).
** (3) 2 Call: travel_ok(_41,san_francisco,seattle) ? l
** (3) 2 Exit: travel_ok(monday,san_francisco,seattle) ? l
** (7) 3 Call: travel_ok(monday,los_angeles,seattle) ? l
** (13) 5 Call: connects(u100,u101) ? <ret>
   (14) 6 Call: flight_time(u100,_586,_587) ? <ret>
   (14) 6 Exit: flight_time(u100,800,900) ? <ret>
   (15) 6 Call: flight_time(u101,_596,_597) ? <ret>
   (15) 6 Exit: flight_time(u101,1000,1200) ? <ret>
   (16) 6 Call (system): 1000>900 ? <ret>
   (16) 6 Exit (system): 1000>900 ? <ret>
** (13) 5 Exit: connects(u100,u101) ? -
[Spypoint removed from connects/2]
   (13) 5 Exit: connects(u100,u101) ? l
** (7) 3 Exit: travel_ok(monday,los_angeles,seattle) ? l
** (18) 4 Call: travel_ok(monday,seattle,seattle) ? l
** (18) 4 Fail: travel_ok(monday,seattle,seattle) ? a
[ Execution aborted ]


| ?- [Consulting procedure...]
[ Procedure travel_ok/3 consulted. ]

yes
| ?- meet_on(When).
** (3) 2 Call: travel_ok(_41,san_francisco,seattle) ? l
** (3) 2 Exit: travel_ok(monday,san_francisco,seattle) ? l
** (7) 3 Call: travel_ok(monday,los_angeles,seattle) ? l
** (7) 3 Exit: travel_ok(monday,los_angeles,seattle) ? l
** (18) 4 Call: travel_ok(monday,seattle,seattle) ? l
** (18) 4 Exit: travel_ok(monday,seattle,seattle) ? l

When = monday
```

Using Spypoints

A spypoint is placed on connects to see if that procedure executes properly now that it has been corrected. Because the program is still failing, we can surmise that one of the people can't attend the meeting for some reason. Therefore, a spypoint is also placed on travel_ok. By placing a spypoint on travel_ok, we hope to find out who can't attend the meeting; once we know that, we can trace the procedure in detail to find out why that person can't attend.

The debugger is then set to debug mode since we no longer want to trace each procedure call, and debugging. is typed to obtain a list of the spypoints that have been set. The goal is retyped; and when prompted for interaction the first three times, we direct the debugger to leap (l) to the next spypoint.

At the procedure connects, we begin a detailed trace again. The procedure executes successfully, so we remove the spypoint from connects by typing a minus (-) when the debugger prompts for interaction. We continue the debugging process and find the last bug in the program: the procedure travel_ok, as it is currently defined, requires that people travel from one city to another. Therefore, it cannot be valid for people who don't need to travel because they live in the city where the meeting is to be held. A clause is added to the program to take care of that condition:

```
travel_ok(_,City,City).
```

With the addition of this clause, it becomes possible for the goal travel_ok to be satisfied for people who live in the city where the meeting is to be held.

The procedure travel_ok is reconsulted, and the original goal is retyped; and this time, it succeeds.

FORMAT OF DEBUGGING MESSAGES

As shown in the debugging examples earlier, the system displays a debugging message on your terminal as it passes through each port of a procedure.

NOTE: All debugging messages are output to the terminal regardless of where other Prolog output is going. This allows you to debug a program while information is being read or written to files.

A sample debugging message and an explanation of its symbols are shown below.

```
** (23) 6 Call: 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.
*>	—	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 number is incremented by one every time a call is made to this procedure. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see page 67) are performed.

6

This number represents the number of direct ancestors this goal has. This is referred to as the "current depth." The depth increases as procedures are called and decreases when procedures return. There may be many goals at the same depth, which is why the unique invocation number is also provided. The depth may be shown as C, which indicates that this is a spypoint on a compiled procedure and that no depth or ancestor information is available. To display the ancestors for this goal, use the g debugging option. See page 65 for more information.

Call

This shows the current port: Call, Exit, Redo or Fail.

foo(hello,there,_123)

The goal is then printed so that its current instantiation state can be seen.

?

The final ? is a prompt indicating that you should type in one of the option codes (see next section). If this port is not leashed, there is no prompt and the debugger continues to the next port.

NOTE: The debugger does not show the execution of built-in predicates when they are called at the top level of Prolog. However, the execution of built-in predicates is shown when the predicates are called from within a program. Also, 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**, **otherwise**, **false**, **fail**, **!**, **;**, and **->**.

OPTIONS AVAILABLE DURING DEBUGGING

This section describes the options that you can select in response to the ? prompt, which the debugger displays at every leashed port. The options are one letter mnemonics, some of which can be optionally followed by a decimal integer. If you include blanks, they are ignored.

To see a list of the debugging options, type h (for help) in response to the ? prompt. When you type h (followed by Return), the following list of options is displayed:

```
------------------------------------------------------------------------
Debugging options:

 <cr>    creep      p       print        r       retry          command
  c      creep      w       write        r <i>   retry i    b    break
  l      leap       d       display      f       fail       a    abort
  s      skip       g       ancestors    f <i>   fail i     h    help
  n      nodebug    g <n>   n ancestors  +       spy this   ?    help
  x      backup     < <n>   set depth    -       nospy this =    debugging
                    .       find defn

------------------------------------------------------------------------
```

These options provide a number of different functions which fall into the following classes:

Basic control	— The basic ways of continuing with the execution
Printing	— Showing the goal, or its ancestors, in various ways
Advanced control	-- Affecting control flow and changing spypoints
Environment	— Executing commands, breaking and aborting
Help	— Showing the debugger state and listing options

Each of the options is described below.

Basic Control Options

<cr> (just the return key)

This is the same as the c (creep) option but is reduced to a single keystroke for convenience.

c creep

Causes the debugger to single-step to the very next port and display the goal. Then, if the port is leashed, you are prompted for further interaction. Otherwise, the debugger continues creeping and showing goals until it reaches a leashed port. If leashing has been turned off on all four ports, the debugger will not stop at the next port but will instead display a complete trace of all goals.

l leap

Causes the debugger to resume running your program without stopping until the next spypoint is reached. Spypoints are not affected by leashing, so you will always be prompted for interaction when a spypoint is reached. 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 an evenly spread set of pertinent procedures, and then following the control flow through these by leaping from one to the next.

s skip

At a Call or Redo port, this skips over the entire execution of the procedure. That is, you will not see anything until control comes back to this procedure (at either the Exit port or the Fail port). At an Exit or Fail port, this is equivalent to the c (creep) option. Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution of the procedure. If you skip, then no message at all will appear until control returns. This includes calls to procedures with spypoints set; they will be masked out during the skip.

n nodebug

Causes the debugger to be turned off 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 top-level state (trace or debug). This option does NOT change the top-level debugger state (that is, it does NOT turn the debugger off; to turn the debugger off, you must type nobdebug at the main Prolog prompt).

Printing Options

p print

Reprints the current goal using print and the current debugger print depth limit, which determines how much is printed. The depth limit can be changed with the < option (see below).

w write

Writes the current goal on the terminal using write. This may be useful if your "pretty print" routine (portray) is not doing what you want. Write has no depth limit.

d display

Displays the current goal on the terminal using display. This shows the goal in prefix notation. Display has no depth limit.

g ancestors

Prints the list of ancestors to the current goal (that is, all goals that are hierarchically above the current goal in the calling sequence). This uses the ancestors built-in predicate. Each ancestor goal is printed using print with the current debugger depth limit. Goals shown in the ancestor list are always accessible to invocations for the r (retry) option unless they are marked "(-)". Ancestors marked "(-)" were invoked before the debugger was turned on and have not had debugging information retained for them.

g <n> n ancestors

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

The debugger print depth limit is set 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 initial limit is 10.

Advanced Control Options

x backup

Can be used at the Fail and Redo ports. When used at the Call and Exit ports, causes the debugger to begin creeping.

Control is transferred back to the Call port located at the most recent choice point. The transfer is summarized by listing the ports of the shortest fail path from the original box to the target box. The shortest path will consist of a string of Fail ports ascending the search tree followed by a string of Redo ports descending the search tree to the target box.

This allows you to quickly back up to the most recent choice point without having to see (or stop at) irrelevant ports. This control mechanism does not stop at nor indicate spypoints. If the shortest fail path contains any compiled procedures, the trace messages will be erroneous; however, the control transfer is always completed correctly.

r retry

Can be used at any of the four ports (although at the Call port it has no effect). Control is transferred back to the Call port. This 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 except that procedures that have been modified by assert, retract, etc. will not be changed back to their original state. 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 at the time of the original call. A message [Debugger: retry goal] is output 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. Since the invocation specified may no longer be accessible, the result of this option will be either to return control to the invocation specified, or to the first actually available invocation before this point.

f fail

This is similar to Retry except that it transfers control to the Fail port of the current box. This puts your execution in a position where 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, or to the first actually available invocation before that point.

+ spy this

Places a spypoint on the procedure being currently shown.

- nospy this

Removes any spypoints from the procedure being currently shown.

. find definition

Find the place in the file where the predicate being called is defined.

Environment Options

@ command

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 break.

b break

Calls the built-in predicate break, thus putting you at a new break level with the execution so far being suspended. A break level is like interpreter top-level except that when you end the break (by typing Control x Control d), the suspended execution will resume and you will be reprompted at the port which you left. The new execution is separate from the suspended one, and invocation numbers will start again from 1. During the break, the debugger will be in the current top-level state (nodebug, debug, or trace) unless this is changed. Changes to the debugger, to leashing, or to spypoints will remain in effect after the break has finished.

a abort

Causes an abort of the current execution. All the execution states built so far are destroyed, and execution restarts at the top level of the interpreter.

Help Options

h help

Displays the table of options given above.

? help

Equivalent to the **h** option.

= debugging

Shows the current state of the debugger, the spypoints that have been set, and so forth.

DEBUGGING COMPILED PROCEDURES

The Prolog debugger is normally used to debug interpreted procedures. However, an interpreted program being debugged may have calls to procedures which are compiled. The debugger will show the Calls to these procedures from interpreted clauses and the corresponding Fails or Exits. However, the debugger does not show any of the internal execution for such procedure calls.

Spypoints can be placed on both interpreted and compiled procedures. Spypoints on compiled procedures allow these procedures to be debugged even when they are called from other compiled procedures. The debugger only retains depth and ancestor information when spypoints are called from interpreted clauses. Spypoints on compiled procedures which have been called from other compiled procedures do not retain this information (their depth is shown as C). This means that the **g** option cannot show any ancestors, and the **r** and **f** options are more restricted for such calls.

SECTION 6

PROGRAMMING STYLE

This section gives a number of tips on how to organize your programs for increased clarity and efficiency. See page 27 for additional style suggestions that can help you catch simple programming mistakes more easily.

PROGRAM LAYOUT

Because of its straightforward structure, Prolog is inherently more readable than languages like Pascal and C. You can take full advantage of Prolog's readability by following the style conventions described below when you write your programs.

Normal Layout

Recommended layout conventions are:

1. Insert blank lines between procedures but not between clauses for the same procedure.

2. Start the head of a clause at the left margin; indent goals a few spaces, and write each goal on its own line.

3. Write a comment immediately above each procedure to detail any assumptions that are made about the procedure's arguments and to explain what the procedure is supposed to do. In the example below, each procedure is preceded by a comment line in which a '+', '-' or '?' precedes each argument to indicate whether the variable representing the argument is assumed to be an input, an output, or either when the procedure is called.

4. To put a comment on a line of code, type % at the beginning of the line; do not use the /* */ comment notation on lines of code.

5. When necessary, write single line (%) comments to the right of goals in the body of the program.

6. Try to give variables meaningful names. A convention illustrated in qsort below is to have a series of variable names such as R0, R1, ..., R where R0 is initially given, R1 is derived from R0, R2 is derived from R1, and so on up to R, which is the final result.

The following example demonstrates the recommended layout conventions.

```
/* qsort(+List,-SortedList) sorts a list of numbers using Quicksort */
```

```
qsort(List,SortedList) :- qsort(List,[],SortedList).


/* qsort(+List,+R0,-R).  Sorts the List of numbers and concatenates
   the result with R0 to give R.  This works by partitioning the
   List into two sublists consisting of those elements of the list
   less than the first element and those which are greater.  Then
   the two sublists are sorted by the recursive calls.  */

qsort([X|List],R0,R) :-
   partition(List,X,Below,Above),
   qsort(Above,R0,R1),
   qsort(Below,[X|R1],R).
qsort([],R,R).


/* partition(+List,+X,-Below,-Above) returns in Below all those
   elements of List which are =< X, and in Above all those
   elements which are > X.  */

partition([X|List],Y,[X|Below],Above) :-
   X =< Y, !,                    % NB numbers only (else use @=<)
   partition(List,Y,Below,Above).
partition([X|List],Y,Below,[X|Above]) :-
   X > Y,
   partition(List,Y,Below,Above).
partition([],_,[],[]).
```

Disjunction and Conditionals

It is usually best to avoid using disjunctions in your programs, if possible. However, if you find that you must use disjunctions, use a consistent layout scheme to improve the readability of your program. For example,

```
bank_open(Day,Time) :-
   weekday(Day),                 % The bank is open on weekdays
   \+ bank_holiday(Day),         % except bank holidays
   1000 =< Time,                 % from 10 a.m.
   (  Time =< 1500,              % until 3 p.m.
      \+ friday(Day)             % Monday through Thursday
   |  Time =< 1800,              % or 6 p.m.
      friday(Day)                % on Fridays
   ).
```

All the goals in the disjunction are indented, and the disjunction symbol | is prominently placed directly below the opening parenthesis and above the closing one. It is best to put the | symbol at the beginning of the line containing the disjunction; if it is put at the end of the line preceding the disjunction, it is easy to overlook. The indentation style shown in the example above helps make the scope of the disjunction explicit.

Conditionals that are part of a disjunction are written similarly. The difference is that the test part of the conditional is started right after the opening parenthesis or disjunction symbol which precedes it. This is shown in the following example, which computes or tests the type of a character.

```
type_of_character(Ch,Type) :-
    (Ch >= "a", Ch =< "z" ->      % if "a" =< Ch =< "z"
        Type = lowercase          % then unify Type and 'lowercase'
    |Ch >= "A", Ch =< "Z" ->      % else if "A" =< Ch =< "Z"
        Type = uppercase          % then unify Type and 'uppercase'
    |Ch >= "0", Ch =< "9" ->      % else if "0" =< Ch =< "9"
        Type = digit              % then unify Type and 'digit'
    |otherwise ->                 % else
        Type = other              % unify Type and 'other'
    ).
```

The built-in predicate otherwise is equivalent to true; that is, it simply succeeds without doing anything. otherwise is provided solely for laying out conditionals in this way.

NOTE: The standard disjunction symbol is the vertical bar (|), but the semicolon (;) can be used in place of the vertical bar, if desired. The vertical bar should be used whenever possible because it makes your program layout clearer and easier to read. Note, however, that historically, the semicolon has been used as the disjunction symbol; so even when you use the vertical bar, the system automatically translates the symbol into a semicolon. Therefore, any printouts of your programs will show semicolons, even if you used vertical bars.

Emacs-Related Restrictions

If you want to use the Emacs commands for consulting or compiling a single procedure (<esc> i followed by p, or <esc> k followed by p), your program must conform to certain layout restrictions, as described below. These layout restrictions also apply to the Control x . command, which enables you to find the procedure definition for a particular predicate.

The restrictions are

1. Group Prolog clauses of the same name and arity together.

2. Start the heads of all Prolog clauses in column 1; start any continuing lines for those clauses in some column other than column 1.

3. If a comment continues onto another line, start the continuation of the comment in some column other than column 1.

4. Do not create clause definitions that use operators for 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".

WHERE TO USE "CUT"

One of the harder 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.

What you should do is consider each procedure in your program independently, and decide whether or not it should be able to succeed more than once. In most applications, most of the procedures should only succeed once; that is, they should be determinate. Having decided that a procedure should be determinate, you need to verify that, in fact, it is.

Making Procedures Determinate

Consider the following procedure which calculates the factorial of a number

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

You can find the factorial of 5 by typing

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

X = 120
```

However, if you backtrack into this procedure by typing a semicolon at this point, you get 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 which match the goal fac(0,F), and thus fac is not determinate. There are two 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 procedure is determinate. Thus, it can throw away the information it would otherwise need in order to backtrack to this point to try the second clause.

2. Robust solution: rewrite the second clause as

```
        fac(N,X) :-
                N > 0,
                M is N-1,
                fac(M,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 an infinite loop. In the first solution, :- fac(-1,X) would not terminate.

There is no reason why you can't adopt both of these solutions, and that is the best thing to do in most cases. The space-efficiency point is more important than it may at first seem; if fac is called from another determinate procedure, and if the cut is omitted, Prolog cannot detect the fact that fac is determinate. Therefore, it will not be able to detect the fact that the calling procedure is determinate, and space will be wasted for the calling procedure as well as for fac itself. This argument applies again if the calling procedure is itself called by a determinate procedure, and so on, so that the cost of an omitted cut can be very high in certain circumstances.

You can help make programs more readable by placing cuts as early as possible in clauses. For example, in the procedure

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

it is reasonable to suppose that "b" is a test which determines which clause of p applies; "a" may or may not be a test, but "c" and "d" are probably not supposed to fail under any circumstances. If in fact "a" is the test and "b" is not supposed to fail, then it would be much clearer to move the cut before the call to "b."

Terminating a Backtracking Loop

Cut is also commonly used like this:

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

where candidate_solution generates possible answers on backtracking, and you want to stop generating candidates as soon as you find one which satisfies test_solution. 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 is another example of terminating a generate-and-test loop. In general, you always want to have a cut after a repeat so that the backtracking loop is clearly terminated. If the cut were omitted in this case, then Prolog might backtrack and try to read another term after the end of the file had been reached.

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

INDEXING

In Quintus Prolog, compiled procedures (except those which are declared to be dynamic) have first argument indexing. This means that when a call is made to a procedure with an instantiated first argument, a hash-table is used to gain fast access to only those clauses which have a first argument with the same principal functor as the one in the procedure. (If the first argument is atomic, only clauses with a matching first argument are accessed.)

Keeping this feature in mind when you write your programs can help speed the execution of your programs. Some hints for program structuring that will best use the indexing facility are given below.

Data Tables

The major advantage of the indexing feature is that it provides fast access to tables of data. For example, if you have a table of employee records, you might represent it 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).  ...
```

If you also wanted fast access to the data via department, you could organize the data a little differently. You could index the employee records by some unique identifier, such as employee number, and create additional tables to facilitate access of 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_name(LastName,Id)

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

% department(Department,Id)

department(sales,1000000).
 ...
```

Improved Determinacy Detection

The other advantage of indexing is that it often makes it possible for the system to detect determinacy, even if cuts are not included in the program. For example, consider the following well-known procedure which joins two lists together:

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

If this procedure is called with an instantiated first argument, Quintus Prolog will recognize that only one of the two clauses can apply and thus that the procedure is determinate. Thus there is no need to put a cut in the first clause. By not adding a cut, you maintain greater flexibility: the procedure can now be used in a non-determinate fashion as well, for example by

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

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

TAIL RECURSION OPTIMIZATION

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

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

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

This procedure 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 tail recursion optimization is used, and effectively what happens is that the stack space being used for the current procedure call is reclaimed before the recursive call is made. This means that this procedure uses only a constant amount of space, no matter how deep the recursion.

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

APPENDIX A

WAYS OF INVOKING PROLOG FROM UNIX

The following is a list of ways you can start Prolog from the main Unix prompt.

prolog	Starts Prolog without the editor interface.
prolog +	Starts Prolog with the editor interface.
prolog + filename	Starts Prolog with the editor interface. Initializes the upper window to contain the file filename.
filename	Runs the Quintus Prolog saved state stored in filename.
filename +	Runs the Quintus Prolog saved state stored in filename under the editor interface.
filename1 + filename2	Runs the Quintus Prolog saved state stored in filename1 under the editor interface. Also displays filename2 in the editor window.

APPENDIX B

PORTING PROGRAMS TO QUINTUS PROLOG

Quintus Prolog is very similar to Prolog-20, DEC10 Prolog and C-Prolog. Porting programs from any of these systems to Quintus Prolog is straightforward: a program is provided which examines programs written for these other systems and identifies possible porting problems. For an explanation of how to use this "compatibility checker" see page 84 below. The rest of this appendix outlines the various areas where porting problems may arise.

FLOATING POINT

Quintus Prolog, unlike DEC10 Prolog, provides floating point numbers. The main problem this poses for porting is that the symbol '/' is used in Quintus Prolog for floating point division rather than integer division. The symbol for integer division (which truncates fractional parts of the answer) is '//'. This is consistent with C-Prolog. If you are moving a program from Prolog-20 to Quintus Prolog, the checker will spot all occurrences of '/' in expressions to be evaluated (such as in the right hand argument of is/2) so that they can be changed.

Another possible porting problem is the additional use of '.' in floating point numbers. If '.' is declared to be an infix operator, then 1.2 represents a list cell in Prolog-20 and a floating point number in Quintus Prolog.

THE COMPILER/INTERPRETER INTERFACE

Like Prolog-20, Quintus Prolog provides both an interpreter and a compiler. There are a number of important differences however. Those relevant to porting are:

1. Compiled and interpreted code can be freely intermixed, with no need for public declarations. However, public declarations and mode declarations are ignored by Quintus Prolog and need not be removed.

2. consult has been changed to behave like reconsult in Prolog-20. That is, the procedures defined in a consult operation replace (rather than extend) any previous definitions for those procedures. This makes consult much more like compile so that alternating between consulting and compiling a piece of code is simpler. It also means that spreading the definition of a procedure across more than one file will no longer work; a style warning will occur if you try to do this.

3. In Prolog-20 you can use the assert, retract, and clause predicates

on interpreted code but not on compiled code. In Quintus Prolog, you can use these procedures only on DYNAMIC procedures. Regardless of whether a procedure is compiled or consulted, it may be made dynamic by preceding it with a declaration of the form

```
:- dynamic Name/Arity.
```

You do not need to have a dynamic declaration for a procedure which is only used by assert, retract or clause. However, it is recommended that you have one if the procedure is to be called since you may get an "unknown procedure" trap if the procedure is called before it is known to be dynamic.

STYLE CHECKING

By default, Quintus Prolog gives certain warning messages as it compiles or consults a program (see page 27). When developing new programs, this has been found to be extremely useful in catching editing mistakes, but the warnings can be a nuisance when porting programs which have not been written with the supported style conventions in mind. If this is the case, you may just want to turn them all off with:

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

However, it is recommended that you at least keep the checking for multiple definitions of the same procedure, since this catches the case where the definition of a procedure was spread over more than one file (and this will not work under Quintus Prolog). To get this multiple definition checking call:

```
| ?- style_check(multiple).
```

It can also be worth modifying the program to conform to all the style conventions: longstanding bugs in existing programs have been found by using the style warnings.

MISCELLANEOUS

The form [X,..Y] is not accepted as an alternative notation for [X|Y]. A syntax error message will be displayed if this format is used.

The alternative bracket notations of %(and %) which were allowed as alternatives to { and } are not allowed.

System operator precedences cannot be altered. An error message will be given if this is attempted.

The end-of-file character was 26 (^Z) in Prolog-20. Quintus Prolog returns -1 when an attempt is made to read a character at end-of-file. Also, the character 31 was used for new-line in Prolog-20. In Quintus Prolog it is 10, the normal ASCII linefeed. The checker will give you "suspicious number" warnings whenever it encounters a 26 or a 31 in your program. Of course, it has no way of knowing when a 26 is actually being used to stand for end-of-file, nor when a 31 is actually being used for new-line. It simply warns you about every occurrence of these integers and you should check to see if they really need to be changed.

BUILT-IN PREDICATES

A small number of built-in predicates have been left out of the system:

'LC'/0	obsolete
'NOLC'/0	obsolete
current_functor/2	not useful
log/0	obsolete
nolog/0	obsolete
rename/2	obsolete
revive/2	obsolete
reconsult/1	obsolete

The checker will warn you about any calls to these procedures. If you do call one of them, you will get an error message. One other obsolete predicate is incore/1 which is now defined to be exactly the same as call/1.

The following built-in predicates succeed but currently have no effect as there is no garbage collector for constructed terms (the space is reclaimed only on backtracking).

gc/0
nogc/0
gcguide/3

There are some new built-in predicates:

number(X)	succeed if X is an integer or float (also in C-Prolog)
float(X)	succeed if X is a float
dynamic/1	(not a predicate; used in declarations - see above)
false/0	(same as fail)
otherwise/0	(same as true)
notrace/0	(same as nodebug)
help	print helpful information
help(T)	give help on topic T
user_help	user defined; tells help what to do
manual	access top-level of on-line manual
manual(X)	access specified manual section
style_check(X)	turn on specified style checking
no_style_check(X)	turn off specified style checking
save_program(F)	like save, but execution state is not saved

foreign(F,P)	(user defined) C function F is attached to predicate P
foreign_file(F,L)	(user defined) file F defines C functions in list L
load_foreign_files(L1,L2)	load object files from list L1 using libraries L2
open(F,M,S)	file F is opened in mode M returning stream S
open_null_stream(S)	create output stream S which goes nowhere
set_input(S)	set S to be the current input stream
set_output(S)	set S to be the current output stream
current_input(S)	S is the current input stream
current_output(S)	S is the current output stream
flush_output(S)	flush output buffer for stream S
character_count(S,N)	N is number of characters read/written on stream S
line_count(S,N)	N is number of lines read/written on stream S
line_position(S,N)	N is number of characters read/written on current line of S
get(S,C)	C is the next non-blank character input on stream S
get0(S,C)	C is the next character input on stream S
skip(S,C)	skip input on stream S until character C is found
put(S,C)	output character C on stream S
nl(S)	output a newline on stream S
tab(S,N)	output N spaces on stream S
read(S,T)	read term T from stream S
write(S,T)	write term T on stream S
writeq(S,T)	write T on S, quoting atoms where necessary
print(S,T)	portray or else write term T on stream S
term_expansion(T,N)	(user defined) tells expand_term what to do (also in C-Prolog)
unix(T)	gives access to Unix facilities

The checker warns you about uses of these procedures; in the event of a name clash with a procedure in your program, this facilitates finding all the occurrences so that you can change the name. If you try to define clauses for a built-in predicate in Quintus Prolog, you will get an error message.

RUNNING THE COMPATIBILITY CHECKER

The checker is most conveniently run under the editor interface so that you can peruse its output at your leisure rather than trying to read it as it is written. Assuming that the quintus directory has been placed in /usr/local in your computer, proceed as follows:

```
| ?- compile('/usr/local/quintus/checker').
yes
| ?- check(file1).

(warnings output here)
```

```
[ file1 checked ]
yes
| ?- check(file2).

(etc)
```

The argument to check may be a list of files. If a file has a '.pl' suffix, this may be omitted.

It may be useful to direct output from the checker to a file. This can easily be done by

```
| ?- tell(output-file), check(file), told.
```

APPENDIX C

EMACS SUMMARY

The following key bindings are in effect when running Prolog through the Emacs interface. To obtain a complete listing of key bindings, type <esc> x describe-bindings <ret>.

IMPORTANT KEYS

^g	Cancel any unfinished command - and ring bell
^l	Redisplay the screen (if it got messed up somehow)
^x ^s	Save current file
<esc> ^c	Exit irreversibly from Emacs and Prolog (Save files first!)
^x ^c	Temporarily suspend the Emacs job; type "fg" to get it back
^c	Send a Control-C interrupt to the Prolog process
^u	Enter following digits as an argument to the next function
^x ^u	Undo previous Emacs commands (within reason)

CURSOR MOVEMENT

^b	Move the cursor backward one character
^f	Move the cursor forward one character
^p	Move the cursor to the previous line
^n	Move the cursor to the next line
^a	Move the cursor to the beginning of the line
^e	Move the cursor to the end of the line
<esc> b	Move the cursor backward one word
<esc> f	Move the cursor forward one word
<esc> ,	Move the cursor to the top of the screen
<esc> .	Move the cursor to the bottom of the screen
<esc> <	Move the cursor to the beginning of the buffer
<esc> >	Move the cursor to the end of the buffer

INSERTING AND DELETING

^d	Delete the character under the cursor
<del>	Delete the character to the left of the cursor
<esc> d	Delete the word in front of the cursor (can be yanked back)
^k	Delete from the cursor to the end of line (can be yanked back)
^o	Open a new line before the current line
^q	(Quote) enter the very next keystroke exactly as itself
^t	Transpose the two characters to the left of the cursor
^@	Set mark at the current cursor position
^x ^x	Swap positions of cursor and mark; repeat to undo
^w	Delete region between cursor and mark (can be yanked back)
<esc> w	Copy region to the kill buffer (can be yanked back)
^y	Yank back most recently deleted text

GLOBALLY REPLACING

<esc> q	Globally replace one string with another, querying each time
<esc> r	Globally Replace one string with another, without querying

MOVING UP AND DOWN THE BUFFER

^s	Forward incremental search
^r	Reverse incremental search
^v	Move one page forward
<esc> v	Move one page backward
^z	Scroll the screen one line up
<esc> z	Scroll the screen one line down

WINDOW HANDLING

^x 1	Delete all the other windows (except the Prolog window)
^x 2	Split the current window in half
^x d	Delete the current window
^x n	Move the cursor down to the next window
^x o	Move the cursor to the other window (same as ^X n)
^x p	Move the cursor up to the previous window

FILE AND BUFFER HANDLING

^x ^v	Visit the file that is prompted for, in its own new buffer
^x ^i	Insert the contents of the file that is prompted for
^x ^r	Replace buffer's contents with the file that is prompted for
^x ^s	Save buffer to current file
^x ^w	Write buffer to the file that is prompted for
^x b	Switch the current window to showing a different buffer
^x ^b	List all the current buffers

PROLOG-RELATED COMMANDS

<esc> i	Interpret a Prolog procedure, region, or buffer
<esc> k	Compile a Prolog procedure, region, or buffer
<esc> e	Enlarge or shrink the Prolog window; repeat to undo
^x .	Find the source code definition of a Prolog procedure
^c	Send a Control c interrupt to the Prolog process
<esc> x restart-prolog	Start up a new Prolog saved state

KEYS EFFECTIVE ONLY IN THE PROLOG WINDOW

^x ^d	Send an end-of-file to the Prolog process
^x ^e	Capture the previous line of Prolog input for execution

APPENDIX D

BUILT-IN PREDICATES

Following is a complete list of Quintus Prolog built-in predicates.

abolish(F,N) abolish the procedure named F arity N
abort abort execution of the program; return to toplevel
ancestors(L) the list of interpreted ancestors of the current clause is L
arg(N,T,A) the Nth argument of term T is A
assert(C) clause C (dynamic predicate) is added to database
assert(C,R) clause C (dynamic predicate) is added to database: reference R
asserta(C) clause C (dynamic predicate) is added first in database.
asserta(C,R) clause C (dynamic) is added first in database: reference R
assertz(C) clause C (dynamic predicate) is added last in database
assertz(C,R) clause C (dynamic) is added last in database: reference R
atom(T) term T is an atom
atomic(T) term T is an atom or number
bagof(X,P,B) the bag of instances of X such that P is provable is B
break break at the next procedure call
'C'(S1,T,S2) (grammar rules) S1 is connected by the terminal T to S2
call(P) prove (execute) P
character_count(S,N) N is number of chars read/written on stream S
clause(P,Q) there is a clause for dynamic predicate, head P, body Q
clause(P,Q,R) clause for dynamic predicate, head P, body Q, ref R
close(F) close file F
compare(C,X,Y) C is the result of comparing terms X and Y
compile(F) add compiled procedures from file F to the database
consult(F) add interpreted procedures from file F to the database
current_atom(A) A is a currently available atom (nondeterminate)
current_input(S) S is the current input stream
current_op(P,T,A) atom A is an operator type T precedence P
current_output(S) S is the current output stream
current_predicate(A,P) A is name of a predicate, m. g. goal P
current_stream(F,M,S) S is a stream open on file F in Mode M
debug switch on debugging
debugging output debugging status information
depth(D) the current interpreted invocation depth is D
display(T) write term T (prefix notation) to the user stream
erase(R) erase the clause or record, reference R
expand_term(T,X) term T is a shorthand which expands to term X
fail backtrack immediately
false (same as fail)
fileerrors enable reporting of file errors
float(N) N is a floating point number
flush_output(S) flush output buffer for stream S
foreign(F,P) user defined; C function F is attached to predicate P
foreign_file(F,L) user defined; file F defines C functions in list L
functor(T,F,N) the principal functor of term T has name F, arity N
gc enable garbage collection (currently has no effect)
gcguide(F,O,N) change garbage collection parameter F from O to N
get(C) C is the next non-blank character input on the current input
get(S,C) C is the next non-blank character input on stream S

get0(C) C is the next character input on the current input
get0(S,C) C is the next character input on stream S
halt exit from Prolog
help display a help message
help(T) give help on topic T
incore(P) (same as call)
instance(R,T) an instance of the clause or term referenced by R is T
integer(T) term T is an integer
Y is X Y is the value of arithmetic expression X
keysort(L,S) the list L sorted by key yields S
leash(M) set the debugger's leashing mode to M
length(L,N) the length of list L is N
line_count(S,N) N is number of lines read/written on stream S
line_position(S,N) N is number of chars read/written on current line of S
listing list all interpreted procedures
listing(P) list the interpreted procedure(s) specified by P
load_foreign_files(F,L) load object files from list F using libraries L
manual access top-level of on-line manual
manual(X) access specified manual section
maxdepth(D) limit invocation depth (interpreted code only) to D
name(A,L) the list of characters of atom or number A is L
nl output a new line to current output
nl(S) output a newline on stream S
nodebug switch off debugging
nofileerrors disable reporting of file errors
nogc disable garbage collection (currently has no effect)
nonvar(T) term T is a non-variable
nospy(P) remove spy-points from the procedure(s) specified by P
nospyall remove all spypoints
no_style_check(A) turn off style checking of type A
notrace switch off debugging (same as nodebug)
number(N) N is a number
numbervars(T,M,N) number the variables in term T from M to N-1
op(P,T,A) make atom A an operator of type T precedence P
open(F,M,S) file F is opened in mode M returning stream S
open_null_stream(S) output stream S goes nowhere
otherwise (same as true)
phrase(P,L) list L can be parsed as a phrase of type P
portray(T) user defined; tells print what to do
print(T) portray or else write the term T on the current output
print(S,T) portray or else write term T on stream S
prompt(A,B) change the prompt from A to B
put(C) output character C on current output
put(S,C) output character C on stream S
read(T) read term T from current input
read(S,T) read term T from stream S
recorda(K,T,R) make term T the first record under key K, reference R
recorded(K,T,R) term T is recorded under key K, reference R
recordz(K,T,R) make term T the last record under key K, reference R
repeat succeed repeatedly
restore(S) restore the state saved in file S
retract(C) erase the first interpreted clause of form C
save(F) save the current state of Prolog in file F
save(F,R) as save(F) but R is 0 first time, 1 after a `restore'

save_program(F) save the current static state of Prolog in file F
see(F) make file F the current input stream
seeing(F) the current input stream is named F
seen close the current input stream
set_input(S) set S to be the current input stream
set_output(S) set S to be the current output stream
setof(X,P,S) the set of instances of X such that P is provable is S
skip(C) skip input on current input stream until after character C
skip(S,C) skip input on stream S until character C found
sort(L,S) the list L sorted into order yields S
spy(P) set spy-points on the procedure(s) specified by P
statistics output various execution statistics
statistics(K,V) the execution statistic key K has value V
style_check(A) turn on style checking of type A
subgoal_of(G) an interpreted ancestor goal of the current clause is G
tab(N) output N spaces to current output
tab(S,N) output N spaces on stream S
tell(F) make file F the current output stream
telling(F) the current output stream is named F
term_expansion(T,N) user defined; tells expand_term what to do
told close the current output stream
trace switch on debugging and start tracing immediately
trimcore reduce free stack space to a minimum
true succeed
ttyflush transmit all outstanding terminal output
ttyget(C) the next non-blank character input from the terminal is C
ttyget0(C) the next character input from the terminal is C
ttynl output a new line on the terminal
ttyput(C) the next character output to the terminal is C
ttyskip(C) skip over terminal input until after character C
ttytab(N) output N spaces to the terminal
unix(T) gives access to Unix facilities
unknown(A,B) change action on unknown procedures from A to B
user_help user defined; tells help what to do
var(T) term T is a variable
version displays system identification messages
version(A) adds the atom A to the list of introductory messages
write(T) write the term T on the current output
write(S,T) write term T on stream S
writeq(T) write the term T, quoting names where necessary
writeq(S,T) write T on S, quoting atoms where necessary
! 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
X < Y as integer values, X is less than Y
X =< Y as integer values, X is less than or equal to Y
X > Y as integer values, X is greater than Y
X >= Y as integer values, X is greater than or equal to Y
X = Y terms X and Y are equal (i.e. unified)
T =.. L the functor and arguments of term T comprise the list L
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
X @=< Y term X precedes or is identical to term Y

X @> Y term X follows term Y
X @>= Y term X follows or is identical to term Y

APPENDIX E

BUILT-IN OPERATORS

```
:-op( 1200, xfx, [ :-, --> ])
:-op( 1200,  fx, [ :-, ?- ])
:-op( 1150,  fx, [ mode, public, dynamic ])
:-op( 1100, xfy, [ ; ])
:-op( 1050, xfy, [ -> ])
:-op( 1000, xfy, [ ',' ])
:-op(  900,  fy, [ \+, spy, nospy ])
:-op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                              =:=, =\=, <, >, =<, >= ])
:-op(  500, yfx, [ +, -, /\, \/ ])
:-op(  500,  fx, [ +, - ])
:-op(  400, yfx, [ *, /, <<, >> ])
:-op(  300, xfx, [ mod ])
:-op(  200, xfy, [ ^ ])
```

INDEX

READER'S EVALUATION FORM

We are very much interested in your impressions of our documentation and in any suggestions you might have for its improvement. We would be grateful if you would take a few minutes to answer the questions below and mail this page back to us.

1. Can you find the information you need quickly and easily? Is there any information you need that you can't find?

2. Is the text clear and understandable? Please cite any paragraphs and pages that are difficult to understand.

3. Are the documents logically organized? What changes, if any, do you suggest?

4. Are there enough examples, and are the examples helpful?

5. Are there any inconsistencies between the documentation and the software?

6. How can the documents be improved? Please cite specific examples.

(optional) Name ______________________ Company ____________________

Please return to Peter Davies, Prolog Technical Support, Artificial Intelligence Ltd., Intelligence House, 58-78 Merton Road, Watford, Hertfordshire, WD1 7BY.