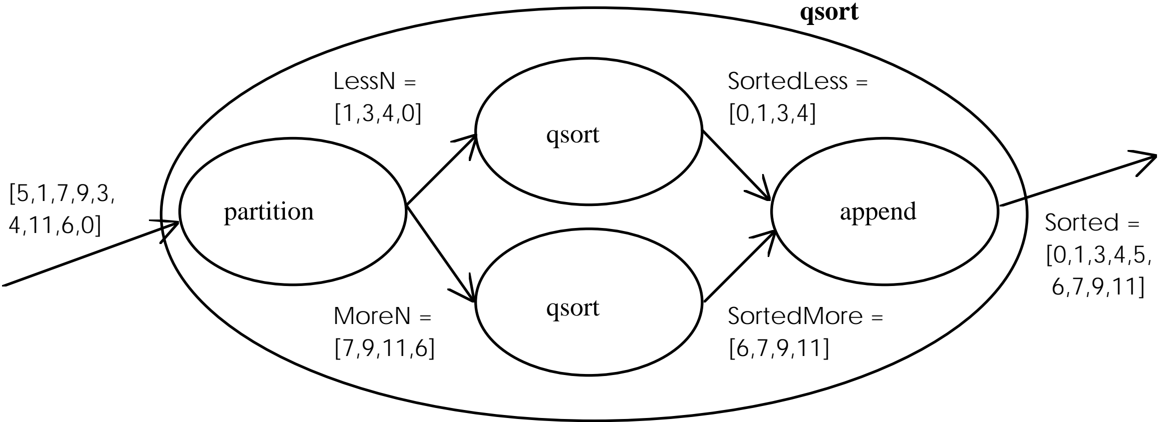
HANDS ON
PARLOG FOR WINDOWS 1.0

A User's Guide



Parallel Logic Programming Ltd.

Hands On Parlog for Windows 1.0

Tom Conlon
Steve Gregory

© PLP Ltd. 1995

January 1995

Parallel Logic Programming Ltd.
PO Box 49
Twickenham
TW2 5PH
United Kingdom

Telephone: (01454) 201652

© 1995 Parallel Logic Programming Ltd.

Contents

1 Introducing Parlog for Windows ... 1
1.1 Scope of this Guide ... 1
1.2 System overview ... 1
1.3 If you are upgrading ... 2
1.4 Installing Parlog for Windows ... 2
1.5 Running Parlog for Windows ... 3
2 Entering queries ... 5
2.1 A simple query ... 5
2.2 Displaying variable bindings ... 5
2.3 Failure and errors ... 6
2.4 Interrupting Parlog for Windows ... 7
2.5 Correcting and recalling queries ... 7
3 Entering and running programs ... 8
3.1 Entering the program ... 8
3.2 Saving the program ... 8
3.3 Loading the program ... 8
3.4 Running the program ... 9
3.5 Lazy and incremental compilation ... 9
3.6 Opening program files ... 10
3.7 Closing program windows ... 10
3.8 Direct loading and saving ... 10
4 Program groups ... 11
4.1 Creating a program group ... 11
4.2 Saving a program group ... 12
4.3 Opening a program group ... 12
5 Dynamic display of query variables ... 13
5.1 Incremental view ... 14
5.2 Film view ... 14
5.3 Snapshot view ... 16
5.4 Combining dynamic display formats ... 16
5.5 Pragmatics of dynamic display ... 17
6 Debugging tools ... 18
6.1 Process monitoring and channel monitoring ... 18
6.2 Running the Primes Sieve example ... 19
6.3 Query tracing ... 19
6.4 Filming a call ... 23
6.5 Process spying ... 24
6.6 Selective process spying ... 25
6.7 Clearing process spypoints ... 25
6.8 Channel spying ... 26
6.9 Call contexts and selective channel spying ... 28
6.9.1 "First-context" channel spying ... 28
6.9.2 Selective channel spying ... 29
6.9.3 The Quicksort example ... 31
6.10 Clearing channel spypoints ... 32
6.11 Summary of channel spying ... 33
6.12 Pragmatics of debugging ... 34

6.12.1 The process tracing model 34
6.12.2 Configuring the trace model 35
6.12.3 Traceable and regular code 36
6.12.4 Lazy and eager recompilation 37
7 Incremental keyboard input 38
7.1 Programming interactive input 38
7.2 Incremental interactive input 38
7.3 Asynchronous interactive input 39
7.4 Stream character input 39
7.5 The in_stream/1 primitive 40
7.6 Multiple input streams 41
8 The database interface 43
8.1 Entering a database procedure 43
8.2 Commands for database relations 43
8.3 Interrogating database relations 44
9 Miscellaneous features 45
9.1 Comments in programs 45
9.2 Queries in programs 45
9.3 Setting compiler directives 45
9.4 Windows 46
9.5 Configuring Parlog for Windows 46
9.6 Porting Parlog for Windows programs to other systems 47
10 Syntax and semantics 48
10.1 Syntax of conjunctions 48
10.2 Syntax of programs 48
10.3 Examples 49
10.4 Operators 50
10.5 BNF syntax definition 51
10.6 Operational semantics 52
10.6.1 Matching and guard execution 52
10.6.2 Concurrency in Parlog for Windows 53
10.6.3 Fair and-parallelism 54
10.6.4 Fair or-parallelism 54
10.6.5 Busy waiting 55
10.6.6 Viewing the scheduler's behaviour 55
10.6.7 Tuning the scheduler's behaviour 55
10.6.8 Execution speed and how to increase it 56
10.6.9 Minimizing evaluation memory use 56
10.6.10 Minimizing code size 56
11 Error messages 58
11.1 Program structure errors 58
11.1.1 Query errors 58
11.1.2 Procedure errors 58
11.1.3 Directive errors 59
11.2 Compiler errors 59
11.3 Run-time errors 59
12 Menus 62
12.1 The File menu 62
12.1.1 The New... option 62

12.1.2 The Open... option 62
12.1.3 The New Group... option 62
12.1.4 The Open Group... option 62
12.1.5 The Save All option 63
12.1.6 The Close All option 63
12.1.7 The Exit option 63
12.2 The Edit menu 63
12.3 The Search menu 63
12.3.1 The Find option 63
12.4 The Run menu 64
12.4.1 The Load All option 64
12.4.2 The Quit option 64
12.4.3 The Tidy Windows option 64
12.5 The Window menu 64
12.5.1 The Cascade option 64
12.5.2 The Tile option 65
12.5.3 The Arrange Icons option 65
12.5.4 The lower section 65
13 Primitives 66
13.1 Arithmetic primitives 66
X^ is Expression? 66
E1? =:= E2? 66
E1? =\= E2? 66
E1? < E2? 66
E1? > E2? 67
E1? =< E2? 67
E1? >= E2? 67
13.2 Unification related primitives 67
Term1? = Term2? 67
Term1? \= Term2? 67
Term1? == Term2? 68
same(Term1?,Term2?) 68
Term1? <= Term2? 68
var(Term?) 68
nonvar(Term?) 69
data(Term?) 69
ground(Term?) 69
13.3 Type checking primitives 69
atom(Term?) 69
integer(Term?) 69
float(Term?) 69
number(Term?) 69
atomic(Term?) 69
13.4 Metalevel primitives 70
arg(N?,Term?,Arg^) 70
functor(Term?,Functor?,Arity?) 70
Term? =.. List? 70
name(Atomic?,String?) 71
cat(Atoms?,Atom^) 71

varsin(Term?,Vars^) 71
toground(Term?,Gterm^,Varnames^) 71
tohollow(Gterm?,Term^,Varnames?) 72
13.5 Control primitives 72
true 72
fail 72
not Conj? 72
call(Conj?) 72
call(Conj?,Status^,Control?) 73
13.6 Database primitives 73
set(Solnlist^,Term?,Dbcall?) 73
subset(Solnlist?,Term?,Dbcall?). 74
13.7 Input and output primitives 74
read(Term^) 75
read(Channel?,Term^) 75
getkey(N^) 75
get0(N^) 75
get0(Channel?,N^) 75
get(N^) 76
get(Channel?,N^) 76
skip(N?) 76
skip(Channel?,N?) 76
gread(Channel?,Term^) 76
key(N^) 76
in_stream(Stream^) 77
in_streams(Streams?) 77
write(Term?) 78
write(Channel?,Term?) 78
writeq(Term?) 78
writeq(Channel?,Term?) 78
display(Term?) 78
display(Channel?,Term?) 78
nl 78
nl(Channel?) 78
put(N?) 79
put(Channel?,N?) 79
tab(N?) 79
tab(Channel?,N?) 79
incwrite(Channel?,Term?) 79
13.8 File, window, and dialog handling primitives 79
open(File?) 80
create(File?) 80
crwind(Name?,R?,C?,Depth?,Width?) 80
cuwind(Name?) 80
close(Name?) 80
dialog(Name?,Message?,Options?,Selection^) 80
title(Title?) 81
13.9 Program handling primitives 81
load(Channel?) 81

listing 81
listing(Relations?) 82
save(Channel?) 82
save(Channel?,Relations?) 82
kill(Relations?) 82
reinitialize 82
defined(Relation?) 82
13.10 Compilation primitives 82
compile 83
fastcode 83
13.11 Debugging primitives 83
trace 84
notrace 84
debug 84
nodebug 84
spy Relation? 84
nospy Relation? 84
nospyall 84
debug_options(Suspend?,Reduce?,Succeed?) 85
window_debug 85
nowindow_debug 85
windows(S?,R?,C?,Depth?,Width?) 85
13.12 Directive primitives 85
optimize Value? 86
optimize(Relation?,Value?) 86
depth Value? 86
depth(Relation?,Value?) 86
13.13 Miscellaneous primitives 87
free(B^,L^,R^,H^,T^,P^) 87
remember(Id?,Term?) 87
recall(Id?,Term^) 87
op(Precedence?,Type?,Name?) 87
current_op(Precedence?,Type?,Name?,Ops^) 87
ticks(Time^) 88
halt 88
Index of primitives 89

1 Introducing Parlog for Windows

Parlog for Windows is an implementation of the Parlog programming language running under Windows 3.1 (or later). It is completely faithful to the standard Parlog syntax and semantics used in Tom Conlon's book *Programming in PARLOG*. Every program from that book should run unchanged under Parlog for Windows; in fact, source code for the book's final chapter "case study" programs is included on the Parlog for Windows distribution disk.

1.1 Scope of this Guide

This Guide is both an introduction to the Parlog for Windows system and a reference manual for the language. If you know Parlog then this document should be all you need to get you programming in the language under Windows. If you are new to Parlog then you should use this Guide in conjunction with one of the books recommended below.

The Guide has two parts. The first ten sections describe the Parlog for Windows system, including how to get it running, how the system is structured, and in general how to develop and run Parlog programs. These sections are tutorial in style and we suggest that you read them through in sequence, trying the examples out in "hands-on" fashion as you go. Following Section 9 comes more formal material which you will want to consult mainly on a reference basis. These sections chiefly define the syntax and semantics of the language and document the available primitives. The primitives are described in groups arranged by function. There is an alphabetical index to them at the back of the Guide.

This Guide will *not* teach you how to program in Parlog. For this we advise that you read either the tutorial book:

> *Programming in PARLOG*
> by Tom Conlon
> Addison-Wesley 1989

or else the more advanced text:

> *Parallel Logic Programming in PARLOG:*
> *The Language and its Implementation*
> by Steve Gregory
> Addison-Wesley 1987

1.2 System overview

Parlog for Windows is a new member of PLP's family of Parlog systems. PC-Parlog, for the MSDOS environment, and MacParlog, for the Macintosh™, were first released in 1989 — the first concurrent logic programming systems to become commercially available on any microcomputer.

Parlog for Windows has a wide range of implementation features:

- Parallel and sequential clause search operators.

- Parallel and sequential conjunction operators.
- "Deep" guards with full or-parallelism.
- Control metacalls.
- Nearly 100 primitives, including all standard Parlog primitives.
- Lazy and incremental compilation.
- Advanced concurrent debugger with channel and process spying.
- High-level query facilities with "dynamic view" for query variables.
- Database interface.
- Stream keyboard input capability.
- Easy-to-use programming environment.

These features and others are explained in the rest of this Guide.

1.3 If you are upgrading

If you have owned the latest release version of Parlog for MSDOS (PC-Parlog 2.0) it may be helpful to identify how this new system differs from that one.

As a 32-bit application, Parlog for Windows frees you from the memory limitations associated with MSDOS applications such as PC-Parlog: there is effectively no limit on the size of Parlog programs that can be run under Parlog for Windows. The other main enhancement provided by Parlog for Windows is a more friendly programming environment, making use of fully scrollable windows and menus.

There are a few new primitives, partly to exploit the new facilities provided by the Windows environment. These are `cat/2`, `dialog/4`, `recall/2`, `remember/2`, `same/2`, `title/1`, `toground/3`, `tohollow/3`, and `varsin/2`.

Care has been taken to ensure upward compatibility between PC-Parlog and Parlog for Windows. The DOS-specific primitives `edit/0`, `edit/1`, and `os/0` have been removed, while `key/0` and `space/4` have been replaced by `key/1` and `free/6`, respectively. With those exceptions, Parlog programs developed under PC-Parlog should compile and run without difficulty under Parlog for Windows.

1.4 Installing Parlog for Windows

The Parlog for Windows package comprises a single 3½-inch disk together with a manual *Hands on Parlog for Windows 1.0*, which you're reading now. We recommend that you copy the disk in the usual fashion: put the master away in a safe place and work only with the copy. That way you have a backup against accidents. The Parlog for Windows disk is not copy-protected — we trust you not to abuse our licensing agreement.

To use Parlog for Windows you need an IBM PC compatible machine running

Windows 3.1, with at least 3Mb of available XMS memory.

To start, insert the Parlog for Windows diskette and copy all of its contents to a suitable directory on your hard disk. The diskette contains the following files and directories:

- Five files which constitute the Parlog for Windows system: 'PARLOG.EXE', 'PARLOG.OVL', 'PARLOG.EXP', 'PARLOG.INI', 'WINMEM32.DLL'.

- A directory named 'EXAMPLES', containing Parlog code which you can study. Some of these example programs will be used in what follows.

- A file named 'README'. This is a text file which has been created since this Guide was printed. It will contain updated information about the system. Read it quickly now just to see if there is anything really important: the details can wait until later.

1.5 Running Parlog for Windows

One way to get Parlog for Windows running is to use the Windows File Manager to select the file 'PARLOG.EXE' and use the '`File/Run`' menu option, or just double-click on the file. Alternatively, you can create a program item for 'PARLOG.EXE', using the Program Manager's '`File/New`' option.

When Parlog for Windows starts, it will begin by displaying a graphical banner which remains on the screen while the system is loaded. When loading is complete, the banner is replaced by two text windows: the *main* window (named '`Parlog for Windows`') enclosing the *console* window '`Console`' (Figure 1.1). The console window contains a textual banner, followed by the supervisor prompt.

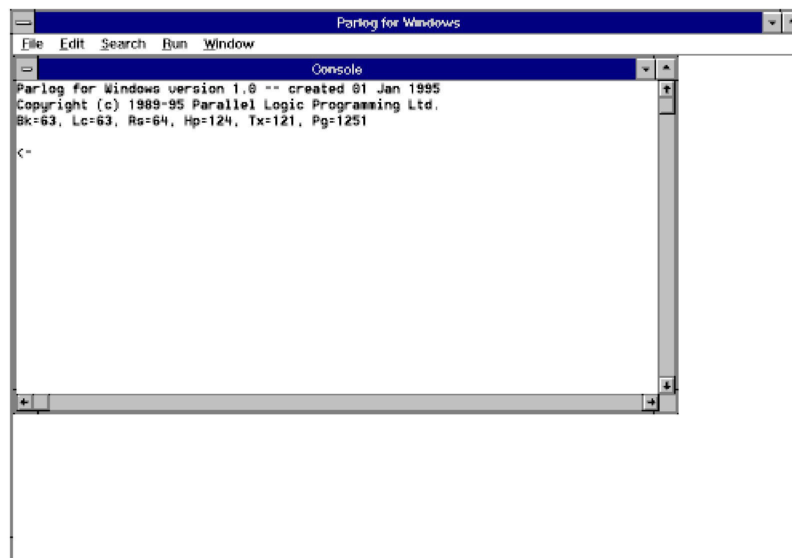


Figure 1.1

The console window acts rather like a standard DOS window: commands and Parlog queries can be typed in at the supervisor prompt. Unlike a DOS window, you can scroll back over text that has disappeared from view, cut and paste text, and reenter previous commands with a single keystroke. The main window provides a set of menus for loading and saving program files, selcting windows, searching, editing, and so on. The menu options, and the

commands available, are described exhaustively in Section 12. First, in the next section, we give a tutorial introduction to the use of the system.

2 Entering queries

2.1 A simple query

As a simple first problem for Parlog for Windows to solve, try entering a query containing a few calls to the `is/2` primitive. Type the following query into the console window:

```
X is Y+2, Y is 3+4.
```

(Note that it is essential to type the period, and then `Return`, at the end of the query.)

Parlog for Windows simply evaluates the query and reports its result, in this case `succeeded`, as shown in Figure 2.1.

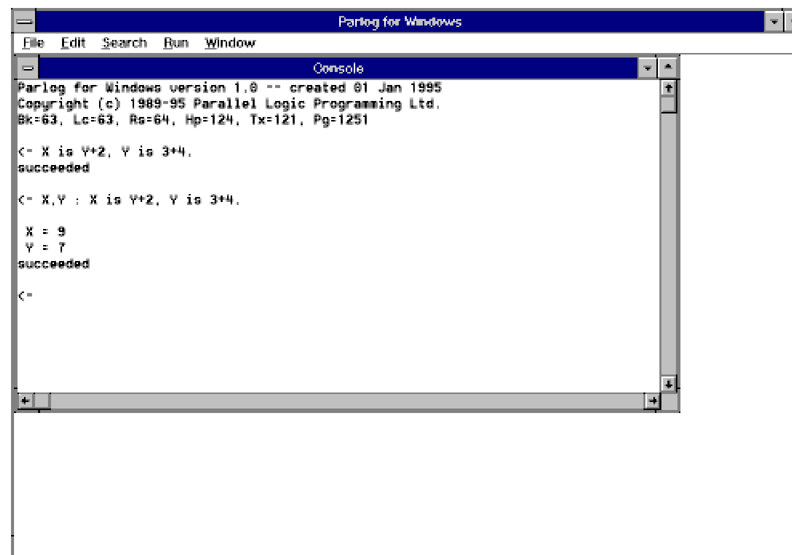


Figure 2.1

This is not very interesting: we would probably be more interested in seeing the bindings that are computed for the query's variables. One way to do this is by including calls to output primitives within your query, for example:

```
X is Y+2, Y is 3+4 & write(answer(X,Y)) & nl.
```

The call to `write/1` appearing in this query will display the term `answer(9,7)` on the screen. The call to `nl/0` will ensure that the result, `succeeded`, will be written to the start of the next line. The primitives `write/1` and `nl/0`, along with many other primitives which can be used for output, are described in Section 13.7.

Notice the mix of parallel with sequential conjunction operators in the above query: Parlog for Windows follows the standard Parlog convention whereby the parallel operator "binds more tightly" than the sequential operator, so that the calls to `write/1` and `nl/0` cannot be made until the evaluation of the concurrent `is/2` calls is complete.

2.2 Displaying variable bindings

Parlog for Windows provides a simpler way to view the bindings of variables, without

explicit output calls: just type the variable names at the beginning of the query, separated by ',' and followed by ':', for example:

```
X,Y : X is Y+2, Y is 3+4.
```

The result of this query is shown in Figure 2.1.

If you want to see bindings for *all* variables in a query, it is not necessary to explicitly list their names: you can type the word `all` in place of the variable list. For example, the above query is equivalent to:

```
all : X is Y+2, Y is 3+4.
```

There are alternative, or additional, ways of displaying variable bindings which are useful for more complex queries; these will be explained later.

2.3 Failure and errors

A query may not succeed like the one shown above. It may fail if the arguments of a call are incorrect *or* if you call a relation that is not defined. An example of a failing query is:

```
2 is 3+4.
```

Another possible outcome of a query is a run-time error. If an error occurs at any time during the evaluation of a query, the evaluation is aborted immediately and an error message displayed: this includes a numeric error code, an explanatory message, and the call that is at fault. For example, an attempt to divide by zero produces a run-time error:

```
X is 1/0.
```

Run-time error messages are described fully in Section 11.3.

Figure 2.2 shows the effect of both of the above queries.

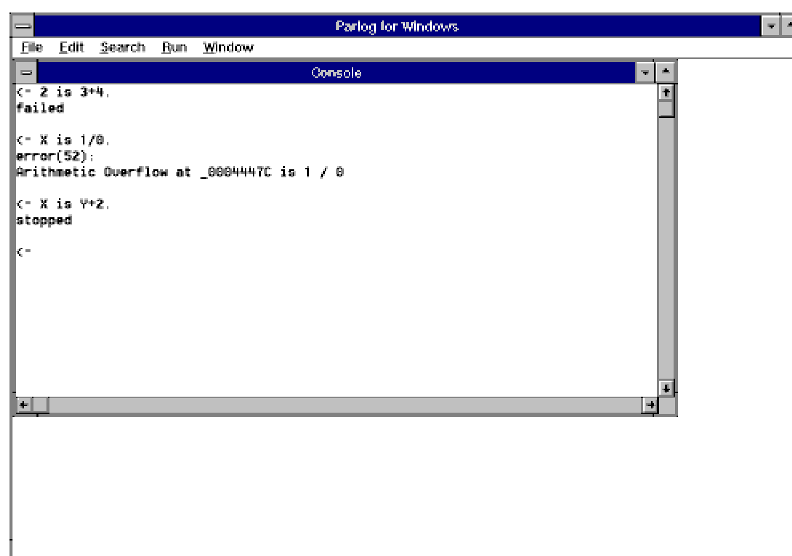


Figure 2.2

2.4 Interrupting Parlog for Windows

When a Parlog evaluation cannot terminate because some process is left waiting for something which will never happen, we have the state of affairs known as *deadlock*. It is only too easy to illustrate how this state can be reached. Enter the query:

```
X is Y+2.
```

and your computer will seem to go into a trance. Deadlock arises here because the `is/2` process suspends waiting for the expression `Y+2` to become ground, which in this case means waiting forever. When you tire of waiting, you can break the deadlock by hitting the `Ctrl-break` key. This will cause the query to be aborted immediately and a `stopped` message displayed, as shown in Figure 2.2.

Of course, deadlock is just one of many reasons why a query may fail to terminate. There are many kinds of program bug that could cause a query to run indefinitely without producing useful results. In all such cases, you will need to use the `Ctrl-break` key (or, equivalently, the '`Run/Quit`' menu option) to regain control of your machine.

2.5 Correcting and recalling queries

If you make a mistake while entering a query, it can easily be corrected at any time before you type `Return`: just use the mouse or cursor keys to position the cursor, and the `Backspace` or `Delete` keys to delete characters. Arbitrary text, from the console window or elsewhere, can be pasted in and edited while forming a query.

It is even possible to recall the last (or any previous) query, avoiding the need to type it in again. Simply move the cursor to the appropriate line in the console window and type `Return`: that line will then be copied to the bottom of the console window and executed automatically. If desired, the line can even be edited before typing `Return`.

3 Entering and running programs

Now it's time to try entering and running a program. As an example we shall enter the procedure for the `integers/3` relation which is given in Chapter 4 of *Programming in PARLOG*. This is provided on the distribution disk.

3.1 Entering the program

First, create a new program window by using the `New...` command from the `File` menu (which we shall abbreviate as '`File/New...`'). A directory dialog box will appear, listing (initially) all files with a '.PAR' extension in the current directory. Use the dialog to enter the name of a file that does not already exist, say 'TEST.PAR': this file will be associated with the new program window. If you select the name of an existing file, a warning message will appear: if you choose to proceed, the existing file will be replaced.

After selecting a filename, a new empty window will appear on the screen with the same name as the file. Now type the `integers/3` procedure into the window, using the usual Windows editing facilities to correct any mistakes.

In general, you can create any number of program windows, and your program can be divided among them as you wish. The only restriction is that a given procedure must not be split between windows.

3.2 Saving the program

While developing a program, you will want to save it on disk at regular intervals. To do this, simply select the '`File/Save All`' menu option. This will immediately save the contents of your program window in its associated disk file. In general, this option saves *all* open program windows that have been edited since they were last opened or saved, into their respective disk files.

3.3 Loading the program

When you are satisfied with the contents of the 'TEST.PAR' window, select the '`Run/Load All`' menu option. This will check the syntax and structure of the source program, and store it in memory in an internal form. In general, this option loads *all* program windows that have been opened or edited since they were last loaded. The program windows are automatically saved into their respective disk files before they are loaded.

If a procedure already exists with the same relation name and arity as one being `loaded`, the new procedure will silently replace the old one, with no warning message.

If your program contains syntax errors, the loading will be aborted and an error message will appear in the console window. If an error is found in the structure of the program, a message will again be displayed in the console window but the loading will continue. (See Section 11 for an explanation of error messages.) In either case, the '`Run/Load All`' option will remain enabled: you should then correct the errors in the window and select '`Run/Load All`' again to load the program.

The internal form of a program preserves all aspects of the source program except its layout and comments. If you want to view the program currently stored, type the command

`listing` in the console window.

3.4 Running the program

Now to see if it works. Enter a query into the console window, such as:

```
X : integers(1,10,X).
```

The solution to the query should now appear on the screen, as shown in Figure 3.1. Congratulations — you have now successfully run your first Parlog for Windows program!

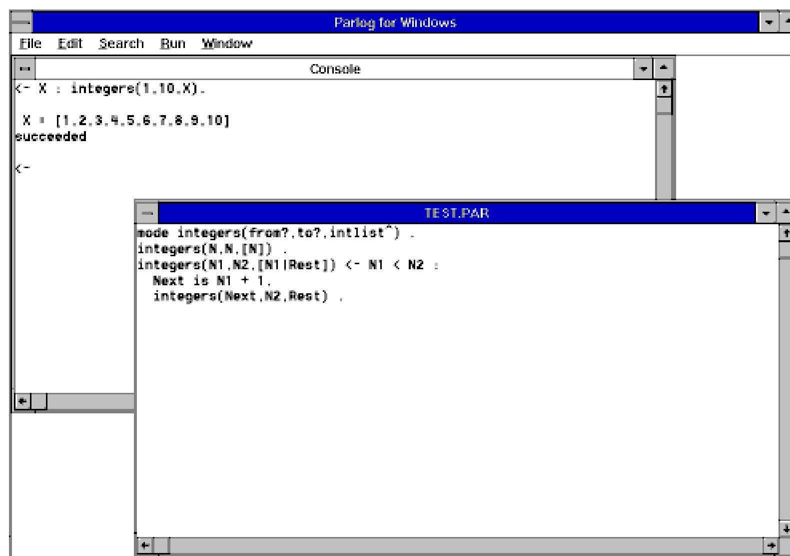


Figure 3.1

3.5 Lazy and incremental compilation

You will have noticed that the query is not run immediately: there is a slight delay while Parlog for Windows compiles the `integers/3` procedure to object code. While this is happening, Parlog for Windows displays messages (in a status box) to tell you what relations it is compiling.

Parlog for Windows's compilation is both lazy and incremental. The compilation is *incremental* in that procedures are automatically compiled if they have just been defined, or have been changed in any way; no time is wasted in recompiling previously compiled procedures. Incremental compilation means that you do not have to worry about handling object code; Parlog for Windows takes care of the compilation automatically.

Lazy compilation means that procedures that need compiling are not compiled until the last minute. That is, procedures are compiled only when they are actually called, even if this is deep inside an evaluation. This facility can be very helpful in the development of a large program: you can load the entire program and test parts of it; only procedures that are actually used will be compiled.

Lazy compilation is useful but you are not forced to make use of it. Instead, you can compile a program *eagerly*: just enter the query:

```
compile.
```

This will force the immediate compilation of all procedures currently in memory except those which have already been compiled.

3.6 Opening program files

The 'File/Open...' menu option is used to open Parlog programs stored in disk files. Select the 'File/Open...' menu option now. A directory dialog box will appear, listing (initially) all files with a '.PAR' extension in the current directory. Now use the dialog to enter the name of an existing file, such as one of the example programs supplied on the distribution disk. The file will be read into a new program window with the same name as the file. If you select the name of a non-existent file, an error message will appear in the console window.

3.7 Closing program windows

A program window may be closed at any time in the usual way: by selecting the `Close` option of the window's control menu. If a window is closed in this way, it is first saved into its associated disk file; the internal form of the window's procedures will remain in memory.

The 'File/Close All' menu option saves all open program windows into their respective disk files (there is no need to select 'File/Close All' first), closes the windows, and deletes the internal form currently stored in memory. Try this option, and then type `listing` in the console window: this should confirm that no program is stored.

3.8 Direct loading and saving

It is possible to load a Parlog program directly from a disk file, bypassing program windows, by using the `load/1` primitive described in Section 13.9. For example:

```
load(integers)
```

will load into memory the contents of the 'INTEGERS.PAR' file in the current directory (a '.PAR' extension is automatically added if one is not specified). You may explicitly specify a disk drive and/or directory, for example:

```
load('c:\parlog\examples\integers')
```

The `load/1` primitive can be especially useful in conjunction with queries in programs (see Section 9.2): one program file can automatically load others without user intervention.

Conversely, a program currently stored in internal form can be saved into a disk file by the `save/1` or `save/2` primitive described in Section 13.7.

4 Program groups

It is usually helpful to divide a Parlog program into many small windows (files) rather than one large window. This helps to separate logically independent parts of the program. Another advantage of this approach is that, when you edit the program, only the window containing the change needs to be reloaded and recompiled: the smaller this window, the faster this process will be.

It would be very tedious to have to open a large number of program files individually, by the methods described in Section 3. To overcome this problem, Parlog for Windows provides the concept of *program groups*: related program files can be grouped together, so that they can subsequently opened in a single action.

4.1 Creating a program group

A program group can be created, or opened, only if there are no program windows currently open. If there are open windows, close them by selecting the 'File/Close All' menu option. Now the two menu options 'File/New Group...' and 'File/Open Group...' will be enabled.

Select 'File/New Group...' now. A directory dialog box will appear, listing all files with a '.GRP' extension in the current directory. Use the dialog to enter the name of a file that does not already exist, say 'TEST.GRP'. If you select the name of an existing file, a warning message will appear: if you choose to proceed, the existing file will be replaced.

After selecting a filename, nothing will seem to happen except that the group filename will be added to the title of the main window. However, every program file that is subsequently created or opened will become part of the named group; program windows that are closed (using their control menu) will be removed from the group.

Try opening several program files now: for example, the programs 'INTEGERS.PAR', 'VARS.PAR', and 'SQUARES.PAR' supplied on the distribution disk. These program files have now become part of the 'TEST.GRP' group. The result should appear as in Figure 4.1.

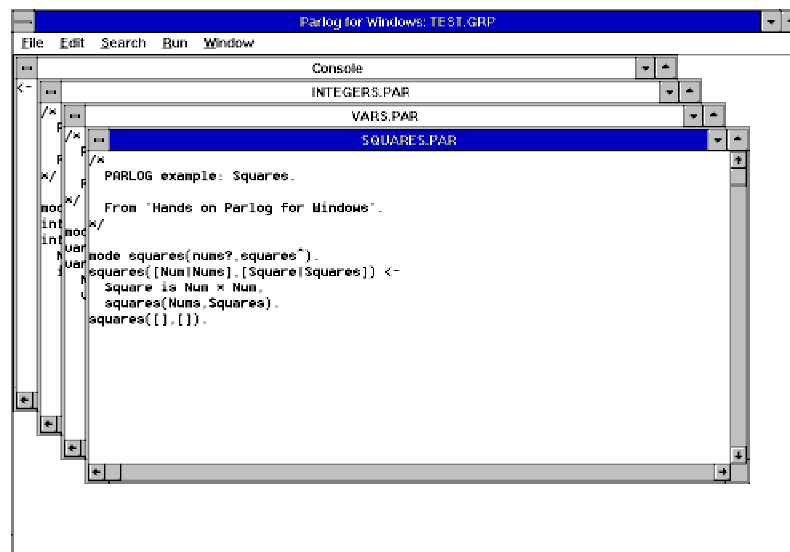


Figure 4.1

4.2 Saving a program group

When a group is open (indicated by the group name in the main window title), it will be saved by the commands that are used to save program windows. If you select 'File/Save All' (or 'File/Close All' or 'Run/Load All', which automatically save all windows), the group file 'TEST.GRP' is updated with the names of all currently open program windows. Selecting 'File/Close All' will close the group, as well as all program windows, removing the group name from the title of the main window. After closing the group, you can open program windows that are not part of a group, create a new group, or open an existing group.

4.3 Opening a program group

The 'File/Open Group...' menu option is used to open a program group previously stored in a disk file. Select the 'File/Open Group...' menu option now. A directory dialog box will appear, listing all files with a '.GRP' extension in the current directory. Now use the dialog to enter the name of an existing group file, such as the 'TEST.GRP' file used as our example. The group filename will be added to the main window title, and all of the program files in the group will appear in program windows on the screen.

5 Dynamic display of query variables

An `integers/3` process is a typical Parlog process in that it is *incremental* in its production of data. The variable `X` in the query:

```
X : integers(1,10,X).
```

actually receives its value as a succession of bindings:

```
X     =     [1|X1]
      =     [1,2|X2]
      =     [1,2,3|X3]
            .................
      =     [1,2,3,4,5,6,7,8,9,10]
```

in the course of the query evaluation. The list data is computed as a "stream" of integers: this incremental behaviour is essential to Parlog's usefulness as a concurrent language. It means that potentially some other process could be running concurrently with the `integers/3` process, and this other process could be doing productive work with the individual integers as they arrive on the stream.

However, the incremental behaviour is not visible from a query like the one above, because the '`:`' operator in the query asks Parlog for Windows to display the bindings computed for its variables only *after* the query evaluation has terminated.

The standard way to animate the display of stream data is to define a "suspendable write" procedure, such as `swrite_list/1`, described in Chapter 5 of *Programming in PARLOG*. Try this now: create a new program window and type in the `swrite_list/1` procedure:

```
mode swrite_list(list?).
swrite_list([H|T]) <-
     data(H) &
     write(H) &
     swrite_list(T).
swrite_list([]).
```

Now with a query such as:

```
integers(1,10,X), swrite_list(X).
```

you can watch a dynamic display on the screen. The integers will be written by the `swrite_list/1` process concurrently with their generation by the `integers/3` process.

This approach works, but Parlog for Windows offers something more convenient. The capability to display dynamically bindings made to query variables is built into the query system. There are three "view formats" for this dynamic display:

- Incremental view.
- Film view.
- Snapshot view.

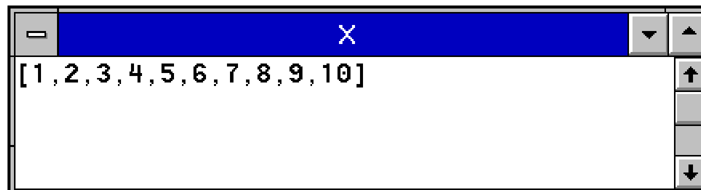
As we shall see next, these facilities are powerful and they make it largely unnecessary to define anything like `swrite_list/1` for yourself.

5.1 Incremental view

To try the first of the dynamic display formats, *incremental* format, enter the query:

```
X :: integers(1,10,X).
```

That is, type the variable(s) that you want displayed, followed by the '`::`' operator (two colons), followed by the query conjunction. Parlog for Windows will now *automatically* display the data produced for `X` incrementally as it is generated by the `integers/3` process. It will output this data into a specially created "view window" which is named `X` after the variable itself:



When the query has terminated, whether successfully or not, the view window will continue to be displayed so that you can examine the contents of the view window(s) at your leisure.

In general, the incremental view option generates one display window for each variable listed before the '`::`' in your query. As a more interesting example, define another procedure which can compute the squares of a given list of numbers:

```
mode squares(nums?,squares^).
squares([Num|Nums],[Square|Squares]) <-
     Square is Num*Num,
     squares(Nums,Squares).
squares([],[]).
```

You don't have to enter it from scratch. If you feel lazy, load it from the 'SQUARES.PAR' file in the 'EXAMPLES' directory on the Parlog for Windows distribution disk. Now enter the query:

```
all :: integers(1,10,X), squares(X,XSquares).
```

(As we saw in Section 2.2, the word `all` is shorthand for the list of all variables in the query.) You should soon see something like Figure 5.1: the list `X` generated by the `integers/3` process and the list `XSquares` generated by the `squares/2` process are displayed incrementally in separate windows.

5.2 Film view

Now to try the second of the dynamic display formats, *film* format. Enter the query:

```
X ::: integers(1,10,X).
```

This is the same as our previous query except that the operator '`:::`' (three colons) is used in place of '`::`'.

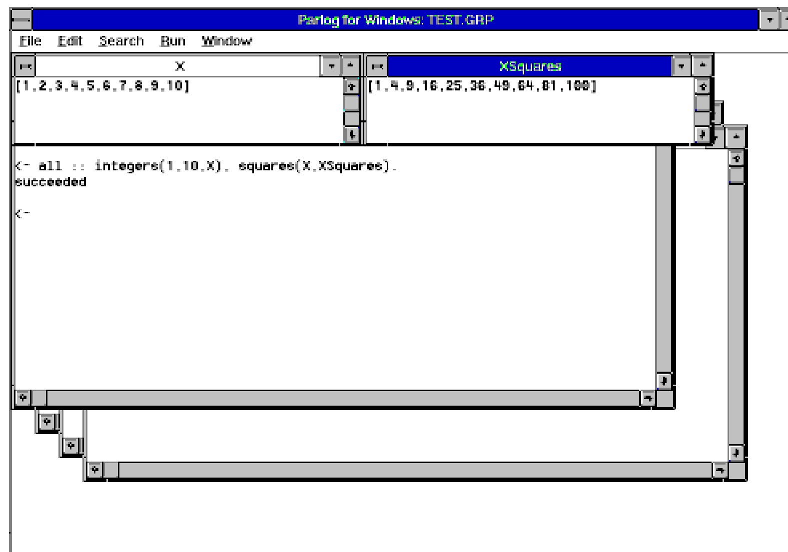
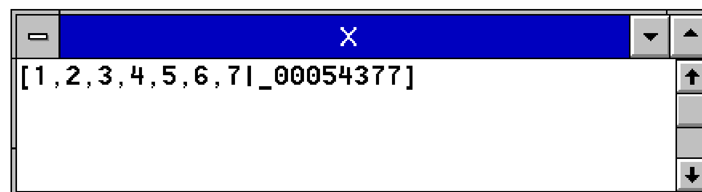


Figure 5.1

At a certain point in mid-evaluation you should see something like this:



The film format of display creates a view window for each displayed variable, as with the incremental format. The difference is that, with a film display, the *entire* binding for the variable is always shown, whether or not this binding contains free variables. A free variable is indicated by a symbol such as `_00054377`, representing an internal address. A film window is continuously updated throughout the evaluation so that the progressive nature of the binding is made visible. The effect can be thought of as a kind of "filming" of the term as it gradually acquires its ultimate value.

This form of dynamic output is more suited to displaying some kinds of term, particularly those containing variables which will never become instantiated. As an extreme example, try a query involving a call to the following `variables/3` procedure (alternatively, load the procedure from the file 'VARS.PAR' on the distribution disk):

```
mode variables(from?,to?,varlist^).
variables(N,N,[V]).
variables(N1,N2,[V|Rest]) <- N1 < N2 :
        Next is N1 + 1,
        variables(Next,N2,Rest).
```

The call `variables(1,10,X)` has a solution for `X` which is a list of ten unbound variables. If we ask for this to be output incrementally, for example:

```
X :: variables(1,10,X).
```

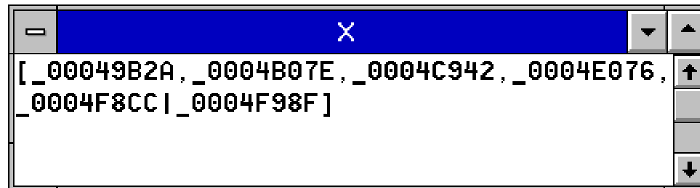
the `X` window will never display more than the opening bracket '`[`' of the list to which `X` is bound. This is because the incremental output facility can only write ground terms and

would suspend, waiting in vain for the first variable in the list to be instantiated; it will finally give up when the `variables/3` process terminates.

Now try the film format instead:

```
X ::: variables(1,10,X).
```

This will produce a more interesting display: the `X` window might look like the following at some point during the query evaluation:



and will display the entire list of ten variables by the time that the query terminates.

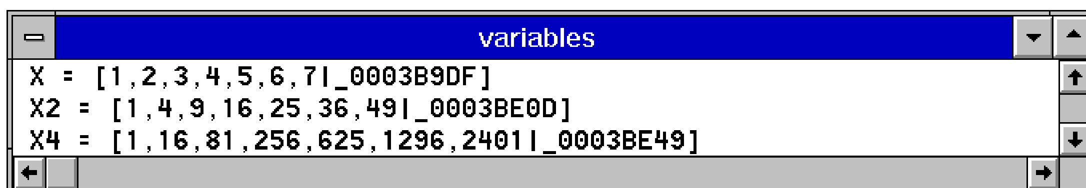
5.3 Snapshot view

The snapshot format is really just an economical version of the film format. No matter how many variables are contained in the query only one window is created, which is named `variables`. Into this window are written all the required variables using the continuously updated "film" style, one variable per line. The `variables` window is wider than the default width of incremental and film output windows, and can be scrolled horizontally.

Assuming that you still have the `integers/3` and `squares/2` definitions intact (otherwise, you can load them from files 'INTEGERS.PAR' and 'SQUARES.PAR'), try it out with a query using the '`::::`' operator (four colons!) such as:

```
all :::: integers(1,10,X),squares(X,X2),squares(X2,X4).
```

At one point in mid-evaluation the display looks like this:



Use the snapshot format whenever you want to save screen space and do not mind that only part of the terms might be visible. Another advantage of snapshot format is that the `variables` window is positioned near the bottom of the screen and the upper part of the screen is left clear for other purposes. Other windows created dynamically — either for dynamic display or for tracing — will not overlap the `variables` window unless this is unavoidable.

5.4 Combining dynamic display formats

The four forms of variable output are not mutually exclusive. You may specify any combination of them, in the general form:

Terminal : *Incremental* :: *Film* ::: *Snapshot* :::: *Conjunction*.

where each of *Terminal*, *Incremental*, *Film*, and *Snapshot* is a list of variable names, or the word `all`. Note that the order is important.

The only restriction is that, if the same variable appears in both the *Incremental* list and the *Film* list, only incremental output is provided for that variable.
Examples of valid queries include:

```
X : X :: integers(1,20,X).
X :: Y ::: integers(1,20,X), variables(1,20,Y).
P :: P :::: primes(P).
```

5.5 Pragmatics of dynamic display

How does the dynamic display feature work? Essentially, your query is pre-processed by the addition of concurrent calls to special system-defined procedures which perform the output. The query supervisor ensures that the system-generated calls terminate as soon as your own calls terminate, so they will never cause deadlock.

The special procedure used for the incremental view format is actually available to your own programs. This is the relation `incwrite/2` which is a Parlog for Windows primitive, documented in Section 13.7. Its definition is quite similar to `swrite_list/1` as shown above, but it is more complex because it has to be able to handle any type of term (not just flat lists).

Terms written with `incwrite/2` are displayed in the same format as used by the `display` primitive. That is, atoms and functors are quoted if necessary but operator declarations are ignored. However, in the case of terms which are written by the film and snapshot procedures, current operator declarations are observed, so the format used is the same as the `writeq` primitive.

A disadvantage of the dynamic display facility is that a query which makes use of it will incur an overhead of extra execution time and storage space, because of the evaluation of the extra output calls. Also, you should not try to infer too much about the behaviour of your processes when the incremental output feature is used. For example, it is possible to be misled by the time-lag which inevitably exists between the production of data by a call and the appearance of that data in an incremental view window.

6 Debugging tools

An important aspect of programming in any language is concerned with analysing program behaviour and, in particular, identifying and fixing bugs. Parlog for Windows provides a powerful set of debugging tools to help you with these tasks.

6.1 Process monitoring and channel monitoring

The evaluation of a typical Parlog program can be understood as a process network: the network comprises nodes representing individual computations or *processes* connected by arcs representing the shared variables or *channels*. For example, the query:

```
integers(1,10,X), squares(X,XSquares).
```

creates the simple process network shown in Figure 6.1.

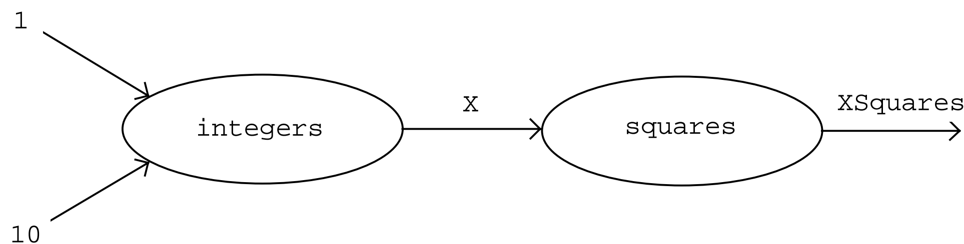


Figure 6.1

Here, there are two processes named `integers` and `squares` and the two are connected by a single communication channel named `X`. The processes each reduce, recursively in this example, to other processes. The communication channel carries the stream of data representing the list `[1,2,3,...,10]` from the first process to the second.

The fact that programs can be understood in terms of processes and channels leads to two complementary approaches to studying program execution. A *process monitoring* approach observes the behaviour of processes, especially the reduction of processes to sub-processes. A *channel monitoring* approach observes the flow of data along the channels which connect processes. Our debugging might be either channel-oriented or process-oriented, depending on the kind of information which we want.

Parlog for Windows provides support for both approaches. Process monitoring is provided by the ability to perform *query tracing* and, more generally, by the ability to set *process spypoints* on selected relations. Channel monitoring is supported by the facility to display query variables dynamically, as described in Section 5, and more generally by the ability to set *channel spypoints* on selected program variables. Figure 6.2 illustrates this for the query mentioned above. We could set a process spypoint on the `integers/3` or `squares/2` relations — this would be process monitoring. Alternatively, or additionally, we may decide to view dynamically the query variables `X` or `XSquares` — this would be channel monitoring.

In the following sections we illustrate Parlog for Windows's debugging tools with the Primes Sieve example which is in the file 'PRIMES.PAR' in the 'EXAMPLES' directory on the Parlog for Windows distribution disk.

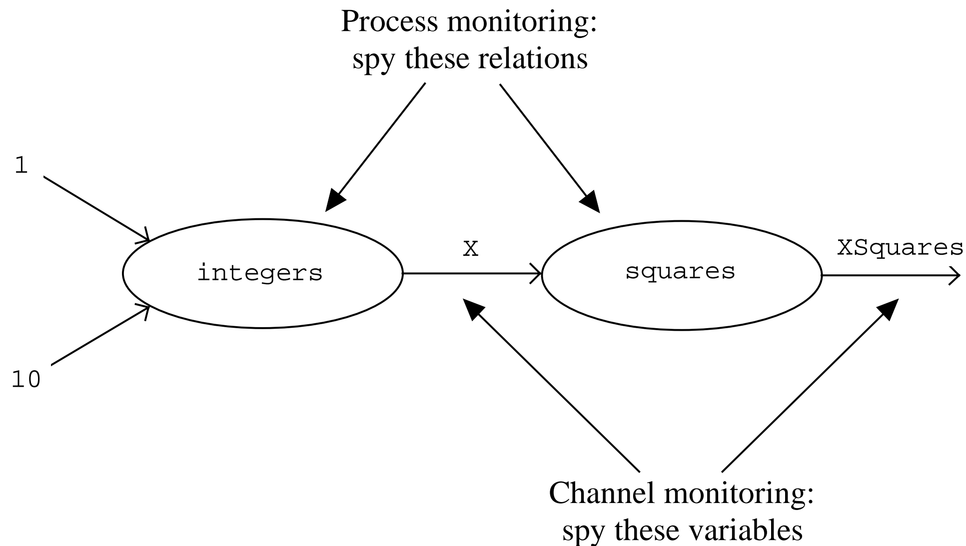


Figure 6.2

6.2 Running the Primes Sieve example

Begin by resetting Parlog for Windows to scratch state. The easiest way to do this is by using the 'File/Close All' menu option. Now use the 'File/Open...' menu option to read in the file 'EXAMPLES\PRIMES.PAR'. The program contains a classic definition for a relation primes/1:

```
mode primes(^).
primes(Primes) <-
     integers_from(2,Ints),
     sift(Ints,Primes).
```

This procedure, together with its subsidiary procedures, specifies a parallel version of the well-known Sieve of Eratosthenes algorithm for generating prime numbers.

First, check that the program does actually work. Try a query like this one:

```
P :: primes(P).
```

After compiling the program, Parlog for Windows will begin to display a list of prime numbers in an incremental view window named P.

Notice how helpful is the dynamic display feature here. The solution to the query is an *infinite* list of prime numbers, so had we used the "terminal" display of variable bindings (using the ':' operator) the wait would have been a long one! Of course, since the evaluation is non-terminating you will have to interrupt Parlog for Windows from the keyboard in order to regain control of the system. The easiest way to do this is by selecting the 'Run/Quit' menu option; the view window for P will remain visible after the interruption.

6.3 Query tracing

A form of process monitoring which is very often useful is *query tracing*. To trace the

Primes Sieve program, just enter the query:

```
trace.
```

to switch on *trace mode*, and then run the `primes` query as before. The tracer will now supervise the evaluation of the query. After creating a view window for `P` (assuming that you have again requested incremental output for `P`), a trace window will be opened, displaying a `call` event. Immediately below this trace window is a *trace dialog* with five buttons, labelled `enter`, `skip`, `unleash`, `film`, and `quit`. This is shown in Figure 6.3.

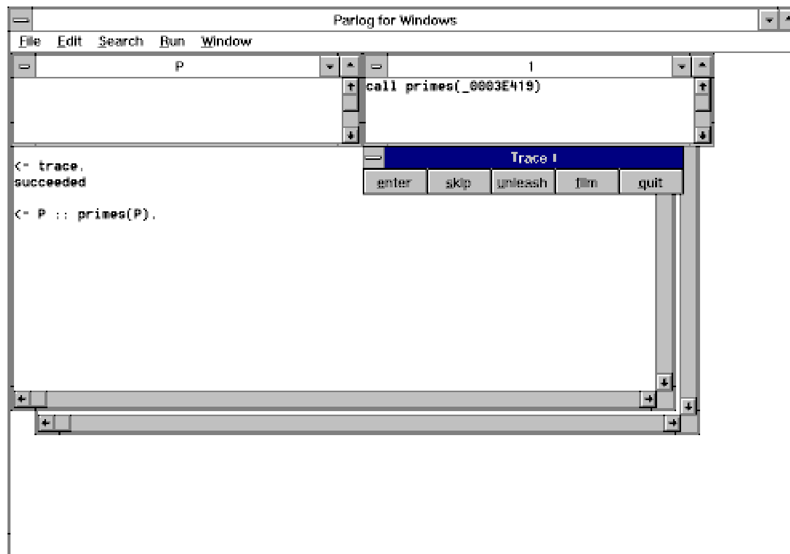


Figure 6.3

The Parlog for Windows tracer normally creates a window for each process and it names them `0,1,2,_` in turn. Here the trace window is named `1` (not `0`, because there is one view window present). The message `call primes(_0003E419)` in the trace window is reporting the first event in the execution of the query: a call has been made to the `primes/1` procedure. Notice that the tracer refers not to a variable's source name but to its internal address.

Each trace dialog is positioned next to the trace window to which it refers; its title is '`Trace` *N*', where *N* is the name of the corresponding trace window. Trace dialogs are modeless: while they are on display you can not only move them around the screen but also select other windows (such as trace windows), move them, resize them, and examine their contents.* As an alternative to moving trace windows, you may prefer to use the `windows/5` primitive to globally change their positions and/or sizes; see Section 13.11. When you are ready to continue with the computation, you must click on one of the buttons on the trace dialog.

From the definition of the `primes/1` procedure, we can see that a `primes/1` process reduces to two sub-processes: an `integers_from/2` process and a `sift/2` process.

Select `enter` from the trace dialog. This will cause Parlog for Windows to report that

---

* **Unfortunately, in the current version of Parlog for Windows, the trace dialog will disappear behind the main Parlog window if you click on any other window while it is on display**. To retrieve the dialog: minimize the main Parlog window, drag the trace dialog to a clear part of the screen, e.g., the bottom, and then restore the Parlog window.

the `primes/1` process has reduced. (At this point, you might notice that the `primes/1` procedure is automatically recompiled. The recompilation is necessary because the tracer requires a special form of object code; this is described in detail later.)

A second trace window will now be opened and a new dialog will appear below it to offer you the same set of options for tracing the `integers_from/2` process, as shown in Figure 6.4. The number `1` in the message `reduce-1 primes(_0003E419)` shows that the `primes/1` call committed to the *first* clause of the procedure (unsurprisingly in this case, since there is only one clause in the `primes/1` procedure).

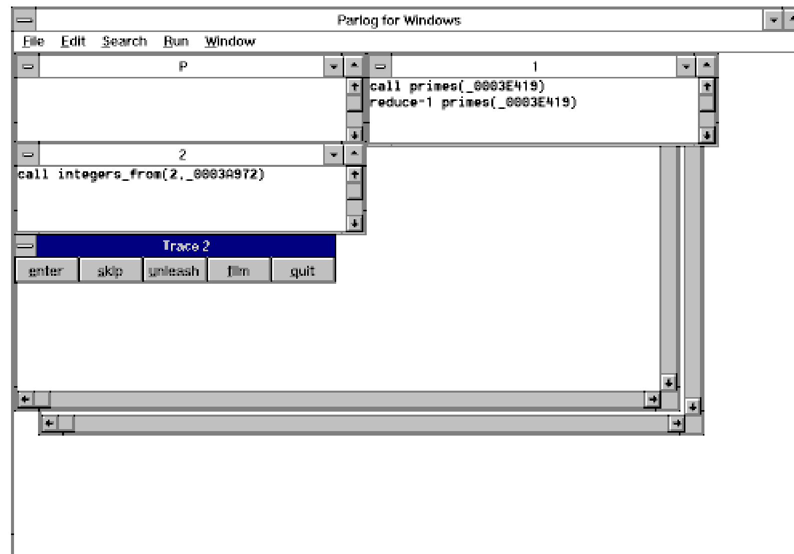


Figure 6.4

Select `enter` to trace the `integers_from/2` process. The tracer will now report a new event in window 1, namely the call to `sift/2`; see Figure 6.5. This illustrates the concurrent execution: the traces of the `sift/2` and `integers_from/2` sub-processes chronologically interleave. Perhaps you expected the `sift/2` process to be traced in a new window. We promised earlier that the Parlog for Windows tracer *normally* creates a new

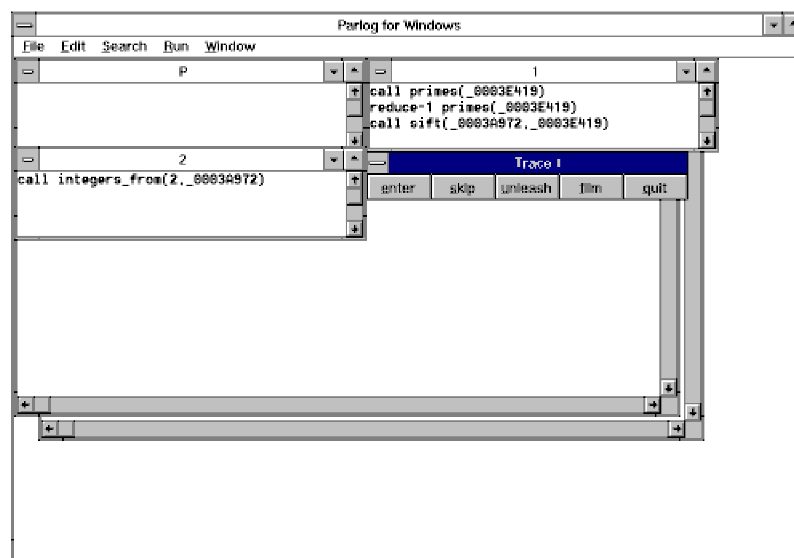


Figure 6.5

window for each process, but to do so without exception would risk an explosion of window numbers. One of the exceptions is the process evaluating the last (textual) call in the body of a clause. This process is traced in the window of its parent. Such a "sharing" of windows may seem an insignificant economy in the case of `primes/1` but with recursive processes it can make a big difference.

You are now tracing the two concurrent processes to which the `primes/1` process has become reduced: a `sift/2` process running in window 1 and an `integers_from/2` process running in window 2. The `call` messages in the trace windows show that the variable `_0003A972` is shared between these processes. It implements a channel which will carry integers from the `integers_from/2` process into the `sift/2` process. (In general, the address of a variable can change during execution so that the same variable could be referenced by different addresses in trace messages displayed at different times. This means that the variable names displayed by the tracer are not necessarily a reliable guide to shared variables.)

If you now select `enter` you will see that the `integers_from/2` process will itself spawn two sub-processes: an `is/2` process and a recursive `integers_from/2` process. A further trace window will be opened for the first of these, while the second will share the parent call's trace window, as Figure 6.6 shows.

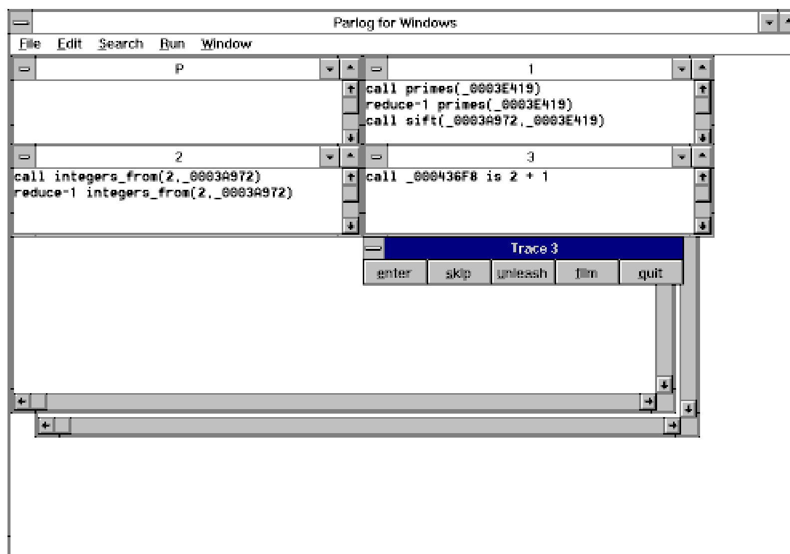


Figure 6.6

Select `enter` to trace the `is/2` call. The `is/2` process will quickly succeed and terminate. Whenever a process terminates, the window in which it is traced will be deleted, but only after you have been given a chance to examine the window's contents. A "close dialog", named '`Close 3`', now appears below window 3 (see Figure 6.7). When you select `close`, window 3 will be deleted.

Calls to primitives such as `is/2` are traced in the same way as other relations, with the exceptions of `data/1` and `ground/1`, which are invisible to the tracer. Of course, no trace events occur during the execution of a primitive call: after `enter`ing one, the next event reported will be `succeed` or `fail`.

Follow the evaluation through for some way until you feel comfortable with the tracer. If you carry on as far as a call to `filter/3` you will see that the tracer does not create new windows to trace guard calls: to restrain the proliferation of windows, these calls are traced directly in the window of the parent process. Another such economy, not illustrated here,

affects sequential conjunctions: two calls `A & B` will be traced in the same window.

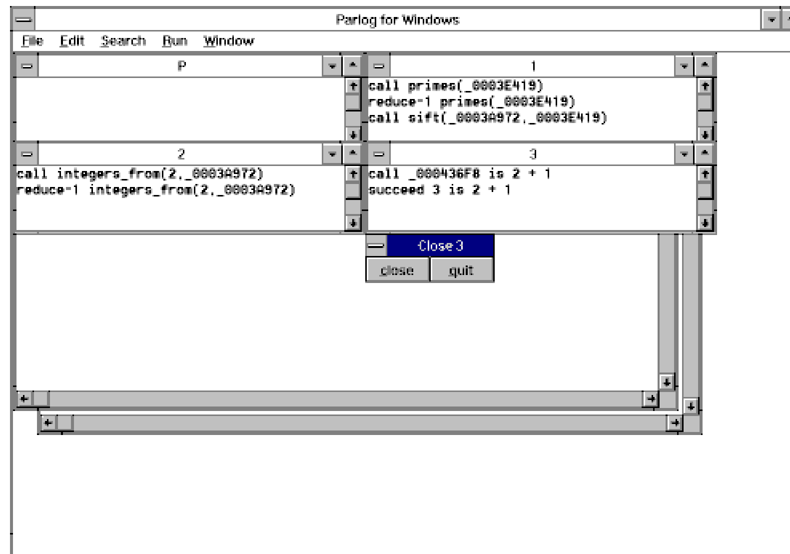


Figure 6.7

Selecting `enter` in response to each trace dialogue enables you to "single step" through the execution. Alternatively, you could choose `unleash` to trace the call exhaustively without further dialogue, or `skip` to execute the call with no tracing. These options are described in more detail below.

To stop a non-terminating process, such as the present `primes` example, select `quit` in response to any trace dialog or close dialog, select the 'Run/Quit' menu option, or type `Ctrl-break` from the keyboard.

6.4 Filming a call

Notwithstanding the tracer's efforts to economize on the growth of windows, it can easily happen during tracing that the number of windows becomes uncomfortably large. One way to keep the numbers down is to select `skip` in response to the trace dialogue to avoid tracing uninteresting calls. These calls will not then be traced, other than to report their eventual outcome. But this approach might be too extreme: for example, if we skipped either of the two sub-processes spawned by `primes/1` then it would be difficult to discover what data the processes generated, since they never terminate. Actually, even when a call is traced it can be far from easy to identify the data it produces because of the way in which streams are typically generated incrementally as processes reduce.

The tracer's *call filming* capability can help with both of these problems. To illustrate, run the `primes/1` program once again with tracing switched on. From the trace dialogue choose `enter` to trace the `primes/1` call as before. But this time, when you are offered the trace dialogue for the `integers_from/2` call, select `film`. This will produce a window named `integers_from/2` in which the `integers_from/2` call will be "filmed": repeatedly during the evaluation the state of the call will be written into the window, just as with the "film view" option for dynamically displaying a query variable.

Notice that the trace dialogue remains on display after selecting `film`. This is because the decision to film a call does not affect your freedom to choose whether to `enter`, `unleash`, or `skip` the call. In this case we shall choose `skip`. Make the same choices

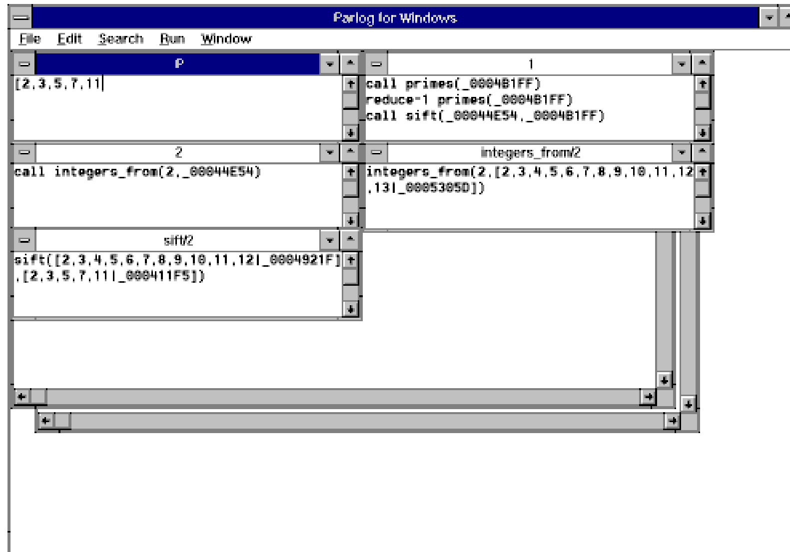


Figure 6.8

when the tracer offers you the `sift/2` call: select `film` and a window named `sift/2` will be created for the call, and then select `skip` and the filming of the call will proceed.

You can now sit back and watch the calls being filmed in their respective windows. At one point in mid-evaluation the windows will appear as in Figure 6.8.

Although this kind of display tells us little about the "glass-box" internal behaviour of a process it does make visible the "black-box" external behaviour, in which the process receives data on some arguments and generates data on others. In effect, call filming gives access to channel spying from within the tracer; channel spying is explained in Section 6.8.

Of course, having opted to skip the tracing of each of the sub-processes we can expect no further intervention from the trace dialog. This means that the only way to interrupt the evaluation is to type `Ctrl-break` or select the '`Run/Quit`' menu option. More typically, the process evaluating a filmed call will terminate eventually, and when that happens the Parlog for Windows tracer announces the event with a dialogue which enables the film window to be closed.

When you no longer want tracing to be done, switch off trace mode by typing the query:

```
notrace.
```

6.5 Process spying

In query tracing we have the ability to follow through the entire evaluation of the top-level query process or processes. If our interest is more localized, in the sense that attention can be confined to some specific sub-processes, then the more selective form of tracing known as *spypoint tracing* may be more convenient. By setting a spypoint on some relation we can ensure that the tracer is invoked only when that relation is actually called. We are able to "spy" the process in which we are interested and ignore the rest.

To set a spypoint on a specified relation you need only enter a query such as:

`spy` *Relation*/*Arity*.

An alternative form is:

```
spy Relation.
```

which sets a spypoint on all relations with the specified name, regardless of their arity. Try this for the `integers_from/2` relation in the Primes Sieve example:

```
spy integers_from.
```

We shall refer to this kind of spypoint as a *process* spypoint. This is to distinguish it from the other form of spypoint, the *channel* spypoint, which we shall come to shortly. (Note that it is not possible to set a process spypoint on a primitive relation.)

Now run the `primes` query again, making sure to switch off trace mode (using `notrace`) if it is still switched on. In the absence of query tracing you will not now be offered the opportunity to trace the `primes/1` process. The trace dialog will appear only when a call is made to the `integers_from/2` relation, because of the spypoint on this relation. You can respond to the dialogs with any of the options `film`, `enter`, `unleash`, `skip`, or `quit`, as usual. If you select `skip`, the evaluation will continue without tracing, but the tracer will be invoked again in a new window the next time that `integers_from/2` is called. Of course there will be a great many of these calls since the relation is recursive. Eventually you will want to abort the execution by selecting `quit` from a trace dialog, by selecting the 'Run/Quit' menu option, or by typing `Ctrl-break`.

6.6 Selective process spying

It is possible to set more selective spypoints which offer to trace only calls that match a certain pattern, rather than all calls to a relation. To set a *conditional process spypoint* on the `integers_from/2` relation, first remove the existing spypoint by a query like:

```
nospyall.
```

Now add the new conditional spypoint:

```
spy integers_from(17,_).
```

When you run the `primes` query again, only the call to `integers_from/2` whose first argument is 17 will be traced; no other calls unify with the specified pattern and so these calls will not invoke the tracer. Parlog for Windows uses a *unification* test to determine which calls should be traced so, for example, a call to `integers_from/2` whose first argument is an unbound variable at the time of the call would also be traced.

Note that conditional process spypoints are additive: if there is more than one, the tracer will be invoked when a call is made which unifies with *any* one of the patterns. Of course, a general process spypoint set on a relation subsumes any conditional ones for the same relation; there is no point in having both kinds in force at the same time.

6.7 Clearing process spypoints

When you've finished spying the `integers_from/2` process you have a choice about what to do next. If what you wanted was only *temporarily* to cease spying then the best option might be to switch off *debug mode*, by the query:

```
nodebug.
```

That way, the spypoint would remain intact but it would have no effect until debug mode is switched on again, by the query:

```
debug.
```

Here though, let's suppose that the spypoint is to be finally discarded. It's the only one set, so there's nothing to be lost by removing all spypoints. Enter the query:

```
nospyall.
```

Had several spypoints been in force, the `nospy/1` primitive would probably have been more appropriate since it allows the removal of individual spypoints. There are three forms of `nospy/1`:

`nospy` *Relation/Arity*.

removes all spypoints (both general and pattern spypoints) from the specified relation, while:

`nospy` *Relation*.

removes them for all relations with the specified name, regardless of arity. Finally, conditional process spypoints can be removed individually. For example, the spypoint on calls unifying with `integers_from(17,_)` can be removed by:

```
nospy integers_from(17,_).
```

Note that all spypoints on a relation are permanently removed when it is deleted (using `kill/1` or `reinitialize/0`, or by the 'File/Close All' menu option); they will not be reinstated if the relation is subsequently redefined.

6.8 Channel spying

As explained earlier, channel monitoring differs from process monitoring in focusing on variables rather than relations. You have already used one form of channel monitoring: in Section 5 we saw how the '`::`', '`:::`', and '`::::`' operators in the query let you "spy" dynamically the bindings for any query variables.

The kind of channel spying which is offered in the query is special, in that a channel being spied is one which connects top-level processes. However, Parlog for Windows does not restrict the channel spying capability to the top level: you are allowed to set a channel spypoint on any program variable, in any procedure. At run time the system will automatically display the bindings which are made for such a spied variable in a specially created output window. The display format can be any one of the incremental, film, or snapshot formats described previously and options are available for specializing the circumstances in which channel spying will be invoked.

Channel spying and process spying can be freely mixed. It is even possible to have both kinds of spypoint set for the same procedure at the same time. However, it will simplify matters here if we assume that all process tracing is inactive, that is, that trace mode has been switched off (by `notrace/0`) and all process spypoints have been cleared (e.g., by

`nospyall/0`). It will also be useful, though not essential, to enter the query:

```
fastcode.
```

at this point, to make sure that the regular form of object code is generated, instead of the slower traceable form. Regular and traceable code is explained in Section 6.12.3.

The Primes Sieve program can be pressed into service once again, this time to illustrate channel spying. As noted earlier, a `primes/1` process reduces to two sub-processes which share the variable `Ints`. This variable acts as a communication channel which carries a stream of integers from the `integers_from/2` to the `sift/2` process. By setting a channel spypoint on the `Ints` variable we will be able to inspect this flow of data dynamically as it is produced; see Figure 6.9.

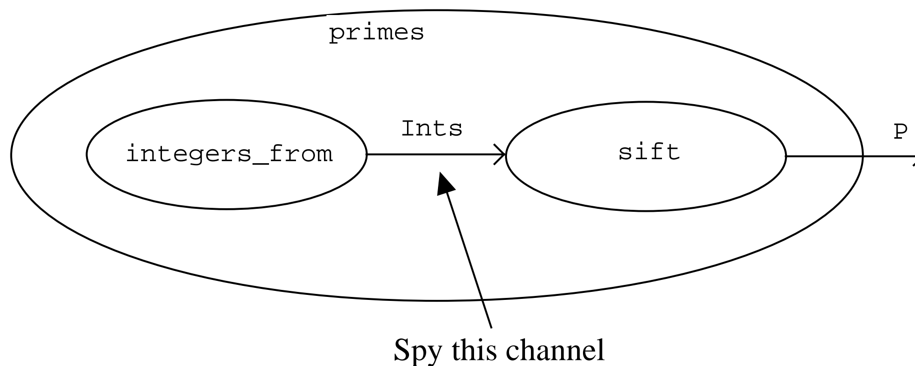


Figure 6.9

To set the channel spypoint, just enter the "query" :

```
Ints :: primes/1-1.
```

This is an example of a *channel spy query*. Although it has a syntax similar to that of a query, it is actually a request to set a channel spypoint.

Now, as before, run the query:

```
P :: primes(P).
```

thus selecting incremental output for variable `P` in the query. You will see two windows created, in which will be displayed, dynamically and incrementally, the bindings made for `P` in the query and for `Ints` in the `primes/1` clause. (At the same time, you may notice that the `primes/1` procedure is automatically recompiled. This is necessary because a channel spypoint has been set on a variable in this procedure.) At one point in the evaluation, which will continue until it is interrupted, these windows will appear as shown in Figure 6.10.

This display provides a dynamic view of the generation of the integers and of the prime numbers which are sifted from the integers one after another.

Notice that the name of the channel spypoint window is `Ints primes/1-1`. This identifies the window contents as representing the binding of the `Ints` variable in the first clause of the `primes/1` procedure.

More generally, a spypoint set on a variable `Var` appearing in clause *C* of some relation *R/A* has the following effect. On *every* occasion on which a call to *R/A* is made which commits to clause *C*, a window named `Var R/A-C` is created, into which is written the binding for `Var`. As soon as the clause body calls terminate — which might never happen,

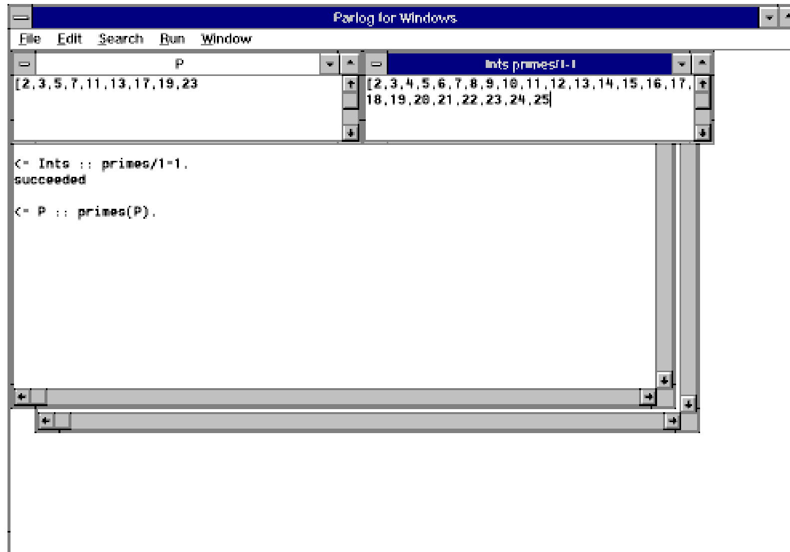


Figure 6.10

as in the case of the Primes Sieve example — a window appears which announces the event.

The channel spypoint window can display terms in film or snapshot format as well as in incremental format. To do this, simply use the '`:::`' operator (for film format) or '`::::`' (for snapshot format) in place of the '`::`' operator, in the channel spy query. It is also possible to select different formats for different variables, even in the same clause. For example, if you enter the channel spy query:

```
Primes :: Ints ::: primes/1-1.
```

the existing channel spypoint in the first clause for `primes/1` will be *replaced* by two new ones: on the clause variables `Primes` and `Ints`. Now you will see the dynamic generation of integers displayed in film format, while the list of primes is displayed incrementally.

If snapshot format is specified for one or more variables in the *C*th clause of relation *R*/*A*, these variables will be displayed together in a window named `variables R/A-C`.

6.9 Call contexts and selective channel spying

In the example above it was safe to assume that only one "copy" of the clause variable `Ints` could ever exist, since there were no calls made to `primes/1` other than the top-level (query) call. More generally, however, a clause variable could exist simultaneously in several *contexts*: one for each call which commits to the clause containing the spied variable. Below we illustrate how channel spying works in multiple contexts, and demonstrate the use of explicit conditions which allow you to select the context(s) in which channel spying should take place.

6.9.1 "First-context" channel spying

An easy, if somewhat contrived, way to illustrate channel spying with more than one context is with the query:

```
primes(P), primes(Q).
```

which creates two concurrent `primes/1` processes. If you run this query after setting a channel spypoint on the variable `Ints` then Parlog for Windows will create *two* spypoint windows: one for each of the two contexts in which the variable `Ints` exists. The windows will be named `Ints primes/1-1` and `Ints primes/1-1:1`, respectively. Of course, each window will display the same data.

In general, for a channel spypoint set on a variable *V* in the *C*th clause of a relation *R*/*A*, the windows corresponding to successive contexts will be named:

```
V R/A-C
V R/A-C:1
V R/A-C:2
```

and so on.

A channel spypoint could easily create many such windows where, for example, the spied variable occurs in a recursive clause. This can easily be avoided by setting a *first-context* channel spypoint instead of the default *all contexts* spypoint. To set a first context channel spypoint, simply add the words '`when first`' to the channel spy query, for example:

```
Ints :: primes/1-1 when first.
```

Now, if you rerun the query:

```
primes(P), primes(Q).
```

you will see just one spypoint window created for variable `Ints`, which will correspond to the first context in which `Ints` exists, i.e., the first call to `primes/1` which commits to the procedure's first clause.

In general, if the words '`when first`' are added to any channel spy query for clause *R*/*A*-*C*, the channel spypoint will take effect only if there is currently no active channel spypoint for that clause.

6.9.2 Selective channel spying

When setting a channel spypoint, we can be more selective by specifying some conditions under which the spypoint will be observed. A conditional channel spypoint is set by a channel spy query of the form:

```
Vars :: R/A-C when Conditions.
```

where `Conditions` is a condition, or a conjunction of conditions separated by commas.

By default, if the `when` is omitted, the condition is `true` which means "*always* observe the spypoint". The first-context channel spypoint described in Section 6.9.1 is another special case in which the only condition is `first` (meaning "observe the spypoint if there is no active spypoint for this clause").

Syntactically, each condition is either the word `first` or a unification test such as:

```
Term1 = Term2
```

Any variables included in a condition are taken as referring to variables in the appropriate clause if the names agree; otherwise they are treated as distinct variables.

To illustrate using the Primes Sieve example, let us set a channel spypoint which will

show the stream of integers entering and exiting the `filter/3` process when `5` is the term being filtered. The definition of `filter/3` is:

```
mode filter(prime?,list?,filtered_list^).
filter(Filter_num,[Num|List1],[Num|List2]) <-
     0 =\= Num mod Filter_num :
     filter(Filter_num,List1,List2).
filter(Filter_num,[Num|List1],List2) <-
     0 =:= Num mod Filter_num :
     filter(Filter_num,List1,List2).
```

We want to set spypoints on the variables `List1` and `List2` in the first clause for `filter/3`, specifying the appropriate selective conditions. We can do this by entering the channel spy query:

```
List1,List2 :: filter/3-1 when first,Filter_num = 5.
```

Now run the query:

```
primes(P).
```

as before. You will eventually see two windows created, in which will be displayed lists representing the input and output channels of `filter/3` when `5` is the integer being filtered out, as shown in Figure 6.11.

Notice how these spypoints are interpreted:

> On the first occasion on which a call commits to the first clause of the `filter/3` procedure with `Filter_num` having the value `5`, create a window for the display of `List1` and `List2` in incremental format.

You may like to convince yourself that the spypoints could have been set on the second clause rather than the first, with only a slight difference in the resulting window output.

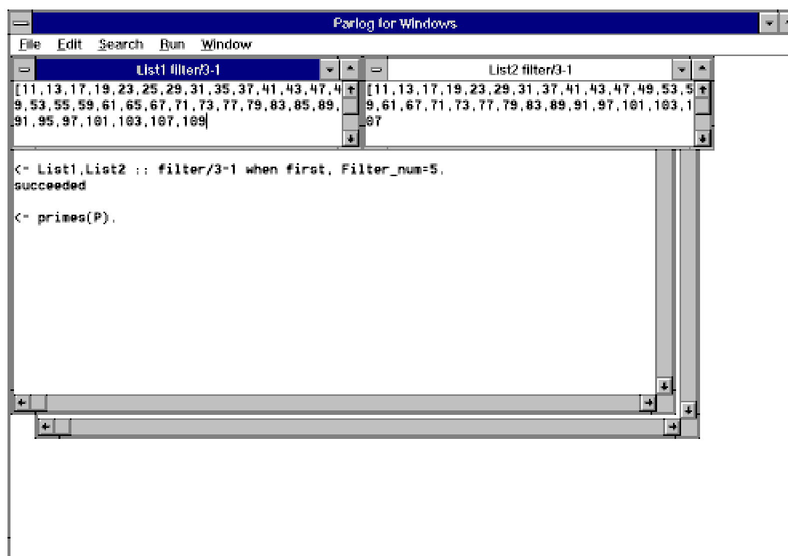


Figure 6.11

6.9.3 The Quicksort example

As a final channel spypoint example, load the 'QSORT.PAR' file on the Parlog for Windows distribution disk. The file contains a Parlog program for a classic list sorting procedure, Hoare's Quicksort, specified as a concurrent algorithm:

```
mode qsort(list?,sorted_list^).
qsort([N|Rest],Sorted) <-
     partition(N,Rest,LessN,MoreN),
     qsort(LessN,SortedLess),
     qsort(MoreN,SortedMore),
     append(SortedLess,[N|SortedMore],Sorted).
qsort([],[]).
```

Test that it works with a query such as:

```
Sorted :: qsort([5,1,7,9,3,4,11,6,0],Sorted).
```

Such a query creates four concurrent processes. As the `partition/4` process generates data for `LessN` and `MoreN` the `qsort/2` processes reduce recursively. By inspecting the procedures we can predict that the `append/3` process will be forced to suspend until the first of these `qsort/2` processes has computed the minimum list element.

We could study the evaluation behaviour by setting a process spypoint on the `qsort/2` relation. Instead, let us focus on the *flow of data* between the sub-processes: for the given query, the dataflow through the process network is illustrated in Figure 6.12.

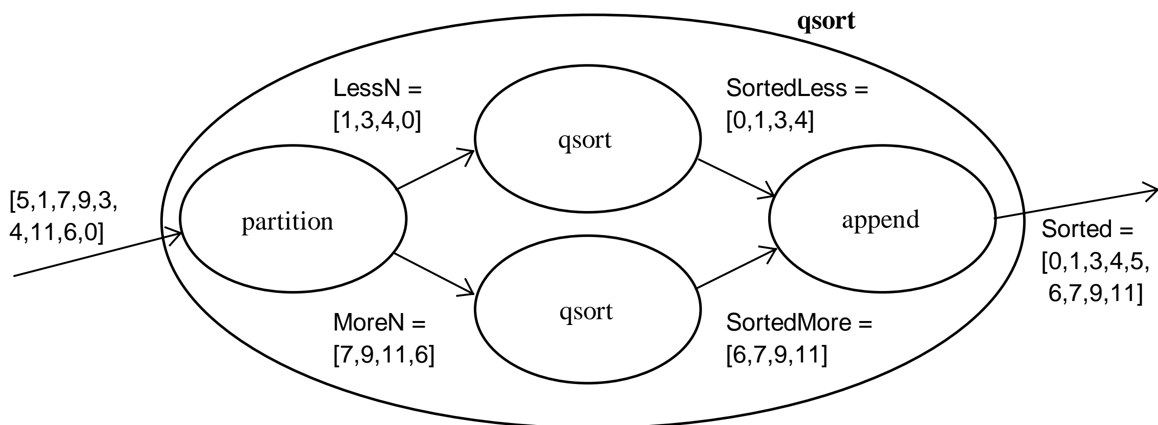


Figure 6.12

We can set channel spypoints on the variables `LessN`, `MoreN`, `SortedLess`, and `SortedMore` by the channel spy query:

```
LessN,MoreN,SortedLess,SortedMore :: qsort/2-1 when first.
```

Then run the query:

```
Sorted :: qsort([5,1,7,9,3,4,11,6,0],Sorted).
```

as before. (Notice that we could have viewed the `Sorted` channel by setting a channel spypoint on `Sorted` in the first clause for `qsort/2`, instead of by displaying the query

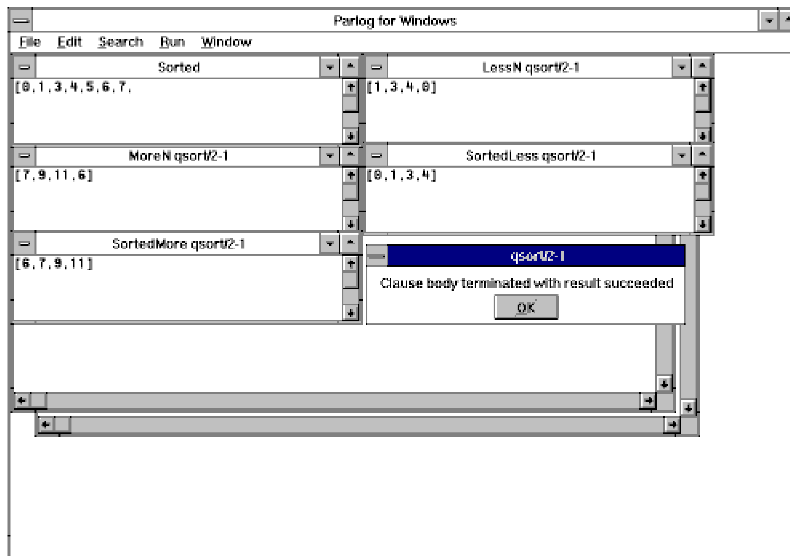


Figure 6.13

variable `Sorted` as we did here.)

Dynamic windows will appear, displaying each of the spied variables, as shown in Figure 6.13. You will also notice that, just before the evaluation ends, a "termination dialog" appears. This serves two purposes: one is to tell you that a clause body has terminated; the other is to allow you to inspect the contents of dynamic windows associated with this clause body before they are deleted. Click the dialog's `OK` button to make the dynamic windows disappear.

In general, whenever a call has previously committed in some context *N* to a clause *R/A-C* which contains variables for which a channel spypoint has been set, Parlog for Windows announces the termination of the call (i.e., the termination of the clause body) with a dialog named `R/A-C:N`, to identify the context in which a clause body has terminated.

Clicking on the dialog's `OK` button will remove all channel spypoint windows which have been created in context *N* to spy variables appearing in clause *R/A-C*. Of course, if the call's evaluation is endless, as with the Primes Sieve example, such a dialogue will never appear.

6.10 Clearing channel spypoints

When your channel-oriented debugging is all done you have a choice about what to do with the spypoints which have been set.

If what is required is only temporarily to desist from channel spying, it would be best to switch off debug mode by the query:

```
nodebug.
```

When debug mode is switched off, all currently set channel spypoints (as well as process spypoints) are ignored. The program will execute as if no spypoints existed, but it will only be necessary to switch debug mode back on and your spypoints will be recognized again.

The `nospyall/0` primitive discards all spypoints (of both kinds). There is no way to reverse this action other than by reinstating your spypoints one by one.

More selectively, channel spypoints can be removed by the `nospy/1` primitive:

`nospy` *Relation*/*Arity*.

removes all channel spypoints and process spypoints associated with the specified relation, while:

`nospy` *Relation*.

repeats this for all relations with the specified name, regardless of arity.

Alternatively, you can remove only the channel spypoints in a specific clause by entering a channel spy query with no variables specified. For example, to remove the channel spypoints for clause *C* of relation *R/A*, enter the query:

```
R/A-C.
```

Channel spypoints for a relation, like process spypoints, will be removed permanently whenever the relation is deleted, using `kill/1` or `reinitialize/0` or the '`File/Close All`' menu option.

6.11 Summary of channel spying

It is worth summarizing here the main points to remember when setting and clearing channel spypoints.

Channel spypoints can be set on any variables occurring in bodies of clauses. They can only be set a clause at a time, by a channel spy query, which takes the general form:

Inc :: *Film* ::: *Snap* :::: *R/A-C* when *Conditions*.

Each of *Inc*, *Film*, and *Snap* is a variable name, or a list of variable names separated by commas, or the word `all`. Any of the three variable lists may be omitted, provided the following operator is also omitted. *Conditions* is a condition (the word `first` or a unification test), or a list of such conditions separated by commas. *Conditions* may be omitted along with the preceding `when` operator.

A channel spy query will have no effect if relation *R/A* is not defined, or if its procedure has fewer than *C* clauses. In this case, the spypoints will not be remembered for later use. Otherwise, it first removes all channel spypoints from the *C*th clause of *R/A* and replaces them by spypoints on each of the variables in the lists *Inc*, *Film*, and *Snap*.

The dynamic display format of each variable may be specified individually: variables in *Inc* are displayed incrementally, those in *Film* are displayed in film format, while variables in *Snap* appear in snapshot format. Any variable that is listed but does not appear in the specified clause will be ignored; a variable that appears in both *Inc* and *Film* will be displayed in incremental format only. The word `all` is shorthand for the list of all variables in the body of clause *R/A-C*.

Conditions specifies the conditions under which the channel spypoints will be observed, and these apply to *all* spypoints in the clause. If *Conditions* is omitted, the spypoints will be observed every time a call to *R/A* commits to clause *C*. Otherwise, they will be ignored unless all tests in *Conditions* are satisfied. A unification test *T1*=*T2* is satisfied if *T1* and *T2* unify at the time of commitment, where variables in *T1* or *T2* are taken to be variables in clause *R/A-C* if the names are identical, and otherwise are new variables. The condition `first` is satisfied if no spypoint for clause *R/A-C* is currently active.

Channel spypoints can be removed from the clause *R/A-C* by a channel spy query:

```
R/A-C.
```

Alternatively, all channel spypoints on a given relation can be cleared by the `nospy/1` primitive, or channel spypoints can be removed globally by `nospyall/0`.

Channel spypoints will not be observed unless debug mode is on. Debug mode can be switched on and off by `debug/0` and `nodebug/0`.

6.12 Pragmatics of debugging

This section describes some pragmatic aspects of debugging. In particular, it describes Parlog for Windows's process tracing model in some detail and it explains the difference between regular code and the "traceable" code that is needed by the process tracer. The information provided here is not vital for most purposes and you may prefer only to skim it briefly on a first reading.

6.12.1 The process tracing model

Consider again the process tracer's dialogue. This offers the following options:

`film` Create a special window in which the call term will be filmed, then restore the dialogue with the options `enter`, `unleash`, `skip`, `quit`.

`enter` Trace the evaluation of this call (process), but present a trace dialogue each time a call in the guard or body of this procedure is entered.

`unleash` Trace the evaluation of this call exhaustively and without offering any further trace dialogues.

`skip` Don't trace the evaluation of this call (except for any spypoints which might be encountered) but only report its eventual success or failure.

`quit` Abort the evaluation of the query. This is equivalent to typing `Ctrl-break` during execution.

The trace dialogue allows the programmer a fair amount of control over the quantity of trace information which is written into the trace windows during an evaluation. You will have noticed that a typical trace looks like this:

```
call integers(1,_7C3A,_7C3D)
call 1 < _7C3A
suspend integers(1,_7C3A,_7C3D)
retry integers(1,5,_7C3D)
succeed 1 < 5
reduce-2 integers(1,5,_7C3D)
call integers(2,5,_979D)
```

Altogether there are six kinds of trace message: `call`, `suspend`, `retry`, `reduce`, `succeed`, and `fail`. These correspond to the events in the "lifecycle" of a Parlog for Windows process, which can be understood from the diagram in Figure 6.14.

`CALL` marks the creation or spawning of the process.

`SUSPEND` marks the state in which the process has not yet found a candidate clause. For at least one clause, input matching with the call has suspended and/or a guard evaluation has not yet terminated.

`RETRY` marks Parlog for Windows's effort to make progress in finding a candidate clause. Parlog for Windows uses a "busy-waiting" mechanism for process suspension, in which the suspended call is retried on each timeslice until (if ever) a candidate clause is identified.

`REDUCE` marks the commitment to a clause body following the discovery of a candidate clause. A message of the form `reduce-`*C* identifies the *C*th clause of the appropriate procedure as the clause selected for commitment.

`SUCCEED` marks the successful termination of the process.

`FAIL` marks the termination of the process in failure.

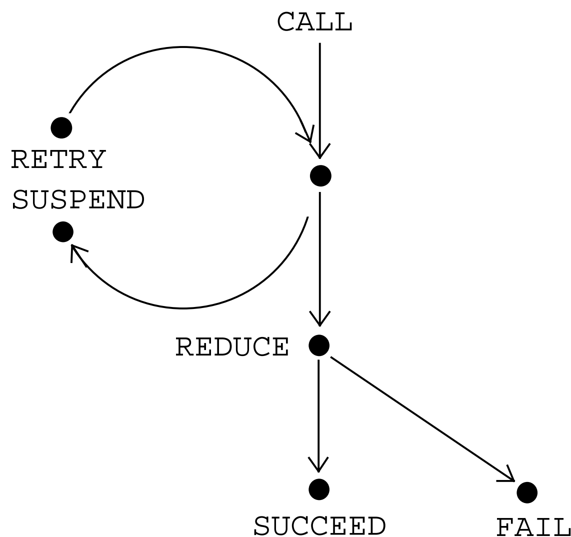


Figure 6.14

6.12.2 Configuring the trace model

By default, the Parlog for Windows tracer reports all the events in the lifecycle of a traced process. But this preset trace model can be changed. You may not want to trace `suspend` and `retry`, for instance, since these events may represent only the "busy waiting" status of a process which is awaiting the arrival of some data from elsewhere.

To reconfigure the trace model, use the `debug_options/3` primitive:

`debug_options(`*Suspend_retry*`,`*Reduce*`,`*Succeed_fail*`).`

Each of *Suspend_retry*, *Reduce*, and *Succeed_fail* may be either `on` or `off`. These determine the trace model; by default they are all `on`. If *Succeed_fail* is `on`, all `succeed` and `fail`

events are traced (otherwise these events are unreported). If *Reduce* is `on`, `reduce` events are traced. If *Suspend_retry* is `on`, `suspend` and `retry` events are traced. Note that the `call` event is always traced.

As we have seen, the tracer normally creates windows dynamically to trace newly spawned processes. This has the advantage that the traces of different processes appear in different windows and can therefore be clearly distinguished. One disadvantage is the memory overhead which the multiple windows incur. As an alternative, all trace messages can be sent to the main screen. As well as reducing the demands on memory, this option also helps by making explicit the chronological interleaving of the execution steps of different processes.

To direct trace information to the main screen, enter the query:

```
nowindow_debug.
```

You can return to tracing in dynamic windows by the query:

```
window_debug.
```

6.12.3 Traceable and regular code

Perhaps you noticed that when tracing was first invoked for the Primes Sieve example, Parlog for Windows automatically recompiled the program. The recompilation was indicated by messages of the form:

```
primes/1 ... compiled
```

which were displayed in a special "status box" during the evaluation. This recompilation is necessary because the Parlog for Windows tracer — which is actually a modified run-time system — cannot operate with the regular object code that is generated by the compiler. It requires a special "traceable" form of code which contains the information needed for writing trace messages.

Parlog for Windows makes the change from regular to traceable code *automatically*. As soon as query tracing is initially invoked, or when first a process spypoint is set for some relation, the system itself will set an internal switch so that traceable code will be generated for use in future evaluations.

When once the traceable code switch has been set all evaluations are supervised by the special trace run-time system. This is the case whether or not query or spypoint tracing is actually being performed. When a relation is called for which only regular object code exists, its procedure is automatically recompiled to the traceable form of code.

Although Parlog for Windows will automatically activate the traceable code switch as required, it *never* automatically deactivates the switch. To do so could result in unnecessary recompilation, for example, between removing one process spypoint and setting another. It is your responsibility to reverse the setting manually at some later time, by the query:

```
fastcode.
```

This will deactivate the traceable code switch, so that future compilation will restore the regular form of code. However, `fastcode/0` will have no effect while trace mode is switched on or process spypoints still exist.

Traceable code is logically equivalent to regular code. You could just leave traceable code selected after a tracing session and stick with it forever. The disadvantage of this is that traceable code is much slower and more demanding in use of memory. Usually therefore, you will want to use `fastcode/0` to restore the regular form of code as soon as your debugging activities are done.

6.12.4 Lazy and eager recompilation

The "silent" recompilation of code in mid-evaluation mentioned above is known as *lazy compilation* or *compilation by need*. It is "lazy" in the sense that the revised form of code that is required for a relation is not regenerated until the last possible minute, that is, until the relation is called. As indicated previously, it is essential that a procedure should be recompiled if the relation is called after the compiler has been switched from regular code to traceable code, or vice-versa; recompilation is also required if a channel spypoint has been set for, or removed from, a relation since the relation was last compiled. The idea behind lazy recompilation is to try to reduce the recompilation effort — only that code which is necessary for the current evaluation is affected.

The possible disadvantages of lazy compilation are that compilations which take place in mid-evaluation occupy additional time and memory, and in some circumstances they could be distracting. Where this matters you can always force Parlog for Windows into an immediate recompilation of all code just by entering the query:

```
compile.
```

In a sense, this command actually does two things: it generates original object code from newly loaded or edited procedures and it immediately updates all existing object code according to the current trace and spypoint settings. So if you use the `compile/0` primitive immediately before running a query you can be sure that compilations will not occur in mid-evaluation.

7 Incremental keyboard input

The Parlog for Windows query system implements a convenient top-level interface to Parlog. For keyboard input it is sufficient for many purposes simply to type directly into the query the initial data which is to be processed by the query calls. For example, to test a `qsort/2` procedure we might enter a query like:

```
Out :: qsort(In,Out), In = [56,34,89,67,44,12,67].
```

But in testing concurrent programs we will often want to supply keyboard input to some process or processes *incrementally* as the process is executing. This section describes how incremental keyboard input can be programmed in Parlog for Windows. The primitives which are introduced here are documented formally in Section 13.7.

7.1 Programming interactive input

One way in which a program can input a term is by calling a primitive such as `read/1`. The behaviour of a call such as:

```
read(Tm).
```

is to wait for the user to type in a term; on typing `Return`, the `read/1` process terminates with the variable `Tm` bound to the term typed, provided it is syntactically correct. This term could be of arbitrary type, for example it might be a list.

We should mention here a possible problem with `read/1`. For as long as a `read/1` process is active, that is, until `Return` is typed, any other concurrent Parlog processes are temporarily locked out. In effect the interactive input monopolizes the computer's solitary processor.

7.2 Incremental interactive input

Something as simple as a single call to `read/1` may suffice for certain applications. But in many interactive programs it is desirable that the value of the variable `Tm` should be a stream, i.e., a list of terms which is generated over a period of time, typically for consumption by some other concurrent process. A first attempt to program this might be the following procedure:

```
mode read_terms(inputs^).
read_terms([Tm|Tms]) <-
     read(Tm) &
     read_terms(Tms).
```

but the problem with this is that the whole Parlog for Windows system will suspend each time `read/1` is called, as mentioned above, even if there are other runnable processes.

Because of this property of the `read/1` primitive, the `read_terms/1` procedure defined above implements *synchronous* interactive input. The program and the user are synchronized at each input event, because each term will be read in only when the program requests input (by calling `read/1`) *and* the user supplies it (by typing a term).

7.3 Asynchronous interactive input

Synchronous input is quite acceptable in some cases, but often it is preferable to accept input from the user *asynchronously*, i.e., whenever the user is ready to input.

Sections 7.5 and 7.6 describe some high-level primitives which fulfil this requirement. However, it is interesting to see how we could implement the behaviour with a procedure of our own, perhaps something like this one:

```
mode async_read_terms(inputs^).
async_read_terms([Tm|Tms]) <-
     async_read(Tm) &
     async_read_terms(Tms).


mode async_read(input^).
async_read(Tm) <-
     key(_) &
     read(Tm).
```

The call to the `key/1` primitive is very important. Its behaviour is to suspend until any key (or mouse button) is pressed, and then succeed. The character typed is returned as the call's argument; in this example it is simply ignored. With `async_read_terms/1` a user can summon an input dialogue by pressing a key whenever a new term is to be added to the stream: the other processes will be interrupted only when necessary, i.e., during each input, using `read/1`. An elaboration of the definition would permit the stream ultimately to be closed, perhaps after some specially recognized term is read.

Incidentally, notice the use of the sequential conjunction operator '&' in both the `async_read_terms/1` procedure and the `async_read/1` procedure. Had we used the parallel operator in `async_read_terms/1`, a large number of calls to `key/1` would soon be concurrently active; in fact the quantity of such calls would grow without limit (or rather, subject only to memory limits). Obviously this would be grossly inefficient. Worse still, it would become effectively a matter of chance as to which call consumed a new keypress. Hence the order of characters output on the stream would be unlikely to agree with the order in which keys were pressed. In general, it is not useful to have more than one `key/1` process active at any one time.

The `key/1` primitive is implemented by keyboard polling which is carried out at quite a low machine level. A call to it will not lock out other concurrent Parlog processes, as would a call to a "higher-level" input primitive such as `read/1` for example. In consequence, a process could run in parallel with our `async_read_terms/1` process, which could consume the stream of characters incrementally as they are generated at the keyboard.

7.4 Stream character input

Almost all of the input primitives (such as `get0`, `read`, etc.; see Section 13.7) that read from the channel named `user` actually obtain their input from the console window. When such an input primitive is called, any characters typed are displayed in the console window and can be edited at leisure before typing `Return`.

As mentioned above, a `key/1` call obtains its character directly from the keyboard (or mouse). There is another primitive that works in a similar way: a call to `getkey/1` reads

and returns the first character that has been typed at the keyboard (or mouse). In both cases, the character is read invisibly without displaying it on the screen, and without the need for `Return` to be typed. The difference between `key/1` and `getkey/1` is that the latter will, like `read/1` etc., cause the Parlog for Windows system to hang until a character is typed.

The `getkey/1` primitive can be used for synchronous input, but it is particularly useful in conjunction with `key/0` to program asynchronous character input. For example, it is easy using this primitive to define a relation which generates a stream of characters typed:

```
mode chars(^).
chars([Char|Chars]) <-
     key &
     getkey(Char) &
     chars(Chars).
```

You may like to define this procedure and try a query such as:

```
C :: chars(C).
```

Press a few keys and watch what happens. The simple version which we have defined here is non-terminating, so eventually you will need to hit `Ctrl-break` in order to interrupt the process.

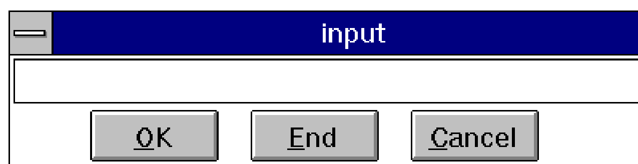
7.5 The `in_stream/1` primitive

In reality, there is no need to define your own stream input procedure like `async_read_terms/1` above since Parlog for Windows provides a primitive `in_stream/1` which implements the required behaviour. The definition of `in_stream/1` is essentially an elaborate extension of `async_read_terms/1`.

The simplest way to illustrate the primitive is to try the query:

```
Tms ::: in_stream(Tms).
```

At first after entering this query nothing appears to happen, but as soon as you press any key an input dialog like this one pops up:



Type any term into the dialog followed by a period character, and then click on the `OK` button. The binding for `Tms` in the film window will then be updated to show the entered term. The input dialog disappears, but it can be reinstated at any time by hitting another key, and you can continue in this way to bind `Tms` incrementally, over an arbitrarily long period of time. One possible interaction sequence might be like this:

Characters typed	**Button clicked**	**Binding for `Tms`**
`1.`	`OK`	`[1|_45C8]`
`2.`	`OK`	`[1,2|_40C1]`
	`End`	`[1,2]`

An in_stream/1 process terminates when the End button is clicked, in which case any term typed into the dialog is ignored. In this example Tms is left with the final value [1,2]. In passing, note that the Cancel button permits the removal of the input dialog without affecting the value of the variable, in case the input dialog should ever be summoned by mistake.

Of course, there is not much point in running an in_stream/1 process by itself. More typically, there will be some other concurrent process which consumes the data that is being entered. As an example, you may like to load the squares/2 procedure which is in the 'SQUARES.PAR' file on the distribution disk. Then a query such as:

```
all ::: in_stream(Ints), squares(Ints,Squares).
```

will enable you to enter a stream of numbers over a period of time, hitting any key as before to summon the input window whenever you like. As the numbers are entered the corresponding list of squares will be computed for the Squares variable and this list will be made visible by the dynamic display. Clicking the input window's End button will terminate the processes.

Finally, note that the system definition of in_stream/1 includes a call to the key/1 primitive, which polls the keyboard. Since key/1 responds to *any* keypress it will not normally be useful to have more than one in_stream/1 process (or an in_stream/1 process with a key/1 process) active concurrently. However, there is a way to generate multiple input streams incrementally; see the description of the in_streams/1 primitive below.

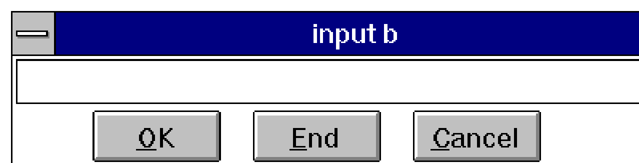
7.6 Multiple input streams

As mentioned above, it is not useful to have more than one in_stream/1 process active concurrently. However, Parlog for Windows provides another primitive in_streams/1 (note the plural) which does support multiple input stream generation from the keyboard.

To demonstrate this primitive, run the query:

```
all ::: in_streams([a(A),b(B),c(C)]).
```

To enter terms, you can now hit one of the "wakeup" keys 'a', 'b', or 'c'; any other key will produce only a beep from the computer's speaker. In response to one of the three wakeup keys Parlog for Windows will generate an input dialog which will accept terms for the corresponding stream. For example, if your first action is to hit key 'b' then a dialog like this one will pop up:



This input dialog is similar to that described above for in_stream/1, but it is selective in that terms typed into it will be output to the stream B (i.e., to the stream associated with the wakeup key included in the dialog name). Suppose that you enter the integer 1 followed by a period, and then click OK. Then in the film window for the variable B you will see a binding of the form [1|_9ADF]. The input dialog disappears and now you may make your next

choice, hitting one of the keys 'a', 'b', or 'c' as before. By continuing in this way over a period of time the three variables A, B, and C become incrementally bound to the entered streams of terms.

As with in_stream/1, concurrently active Parlog processes are not locked out by the presence of an in_streams/1 process except during the periods when an input window is on display. Once a stream has been closed, by clicking the appropriate input dialog's End button, the process ceases to recognize that stream's wakeup key. The process will terminate only when *all* of its streams have been closed.

A simple way to test in_streams/1 is to load the 'FMERGE.PAR' example, provided on the Parlog for Windows distribution disk. This program defines the "fair" version of the merge relation fair_merge/3 which is described in Chapter 6 of *Programming in PARLOG*. Enter the query:

```
all :: fair_merge(A,B,Merged), in_streams([a(A),b(B)]).
```

By pressing key 'a' or 'b' you will be able to supply data for the inputs to the fair_merge/3 process in any desired sequence. The resulting binding for the merged list will be visible in the dynamic display window for variable Merged.

In general, a call to in_streams/1 can specify any list of terms each of the form c(V) where c is a single character atom representing the wakeup key and V is the variable which is to receive the binding. A term other(V) may also be included at the end of the list, in which case any key other than those named elsewhere on the list will act as a wakeup key for the variable V. Try a query such as:

```
all :: in_streams([x(X),y(Y),other(Other)].
```

Supply some data, noting that any key other than 'x' or 'y' will act here as a wakeup key for the variable Other, and investigate the behaviour for yourself.

8 The database interface

Database relations, sometimes called *all-solutions* relations, have the property that a call to such a relation can compute multiple solutions. This contrasts with relations defined in Parlog, for which a call can only ever generate at most a single solution. See one of the books mentioned in Section 1.1 for a full account of the role of database relations in Parlog.

In Parlog for Windows a program can contain any mixture of Parlog procedures and database procedures. A database procedure comprises any set of assertions preceded by a database declaration, of the form:

```
database R/A.
```

where *R/A* is the name and arity of the database relation defined by the subsequent clauses.

Database procedures are *not* Parlog procedures: they have no mode declarations, for example, and they may not be called directly from a Parlog query. A database relation is accessible to a Parlog computation only through the `set/3` and `subset/3` primitives, both of which are documented in Section 13.6.

8.1 Entering a database procedure

Database procedures are loaded in the usual way, via program windows or directly, using the `load/1` primitive. Here is an example which you will find in the file 'DATABASE.PAR' on the Parlog for Windows distribution disk:

```
database sales/3.
sales(bulbs,10,date(10,jan)).
sales(fuses,3,data(10,jan)).
sales(batteries,14,date(14,jan)).
sales(fuses,2,date(15,jan)).
sales(elements,1,date(28,jan)).
sales(bulbs,2,date(28,jan)).
sales(fuses,5,date(3,feb)).
sales(solder,1,date(6,feb)).
```

In general, a program file may contain any mixture of procedures for both Parlog and database relations, in any order: they are distinguished by their distinct declarations. However, it is not possible to have both Parlog and database procedures for the same relation (i.e., the same relation name and arity). As with Parlog procedures, the clauses of an individual database procedure must be contiguous.

8.2 Commands for database relations

The functionality of the Parlog for Windows commands extends, where appropriate, to cover database procedures. For example, programs which include database procedures can be `loaded`, `saved`, and `listed` in the same way as pure Parlog programs.

Externally devised text-file databases can be imported into Parlog for Windows. In particular, text files of assertional Prolog clauses can be read into Parlog for Windows after adding a suitable `database` declaration. Imports from other database systems will

generally require that records be converted to clause form.

8.3 Interrogating database relations

Database relations such as `sales/3` in the example shown above cannot be called directly. For instance, a query:

```
sales(bulbs,Qnty,D).
```

entered directly will *not* deliver the quantities and dates of sales of bulbs. Access to `sales/3`, and to any other database relations, is solely via the `set/3` and `subset/3` primitives; these are the eager and lazy set constructors as described in the books by Conlon and Gregory.

The `set/3` primitive allows us to formulate a query which will return the required information. For example:

```
Dates : set(Dates,D,sales(bulbs,Qnty,D)).
```

will give the solution:

```
Dates = [date(10,jan),date(28,jan)]
```

which is a list of all individual solutions to the `sales/3` relation call.

An alternative query, to the `subset/3` primitive, which could be used to solve the same problem is:

```
D1,D2 : subset([D1,D2],D,sales(bulbs,Qnty,D)).
```

This query will bind the "demand variables" `D1` and `D2` to the respective solutions:

```
D1 = date(10,jan)
D2 = date(28,jan)
```

The distinguishing feature of `subset/3` is that its solutions are computed *lazily*, that is, in response to demands which typically are generated by some other concurrent process. Thus a `subset/3` process could be long-lived. In contrast, a `set` process is typically short-lived: it eagerly (and fairly rapidly) computes the list of all solutions.

Only a single call to a database relation may appear within `set/3` and `subset/3` calls: conjunctions of calls are not allowed. Of course, calls to `set/3` and `subset/3` may appear within the bodies of Parlog clauses, thus enabling database information to be processed by Parlog for Windows programs in a fully general way. The file 'DATABASE.PAR' on the Parlog for Windows distribution disk illustrates some of the possibilities.

9 Miscellaneous features

9.1 Comments in programs

A Parlog for Windows program may contain comments. A comment is any text that begins with `/*` and ends with `*/`, or any text between a '`%`' character and the end of the line. These comments are ignored when a program is read into memory. This means that if you use the `save/1` or `save/2` primitive to write the program back into a file, any comments will have disappeared.

Parlog provides another form of commenting: atomic identifiers can be inserted in mode declarations to document the role and type of a relation's arguments. For example:

```
mode insert(integer?,ordered_list?,inserted_list^).
```

These mode identifiers are preserved in the internal form of a program stored in memory, and are therefore output by the `save/1` or `save/2` primitives.

9.2 Queries in programs

A Parlog for Windows program may include *queries* as well as procedures. A query is a term of the form:

`<-` *Conjunction* `.`

When the program is loaded into memory (using the '`Run/Load All`' menu option or the `load/1` primitive) the Parlog conjunction *Conjunction* is executed immediately. Success or failure is not reported: whether the query succeeds or fails, the loading will continue. However, an error during the evaluation of *Conjunction* will cause the loading to be aborted with an appropriate error message.

One use of queries in programs is to make operator declarations. If a program contains a procedure which uses non-standard operators, these should first be declared by a call to the `op/3` primitive. The best way to make this declaration is by a query embedded in the program file, textually preceding the procedure. The operator will then be declared by the time the procedure is read in.

Another important use for queries in programs is to issue directives; see Section 9.2.

9.3 Setting compiler directives

One class of Parlog for Windows primitives is known as *directive* primitives: these are `optimize/1` and `optimize/2`, used to select or deselect certain compile-time optimizations, and `depth/1` and `depth/2`, which can be used to change the call evaluation depth.

Like several other primitives (such as `op/3` and the debugging primitives described in Section 6), directive primitives change the state of the Parlog for Windows system. They differ from most other primitives in that they affect the way in which the Parlog for Windows compiler works: after using a directive primitive, some or all procedures may need to be recompiled before a query can be run. (Parlog for Windows takes care of this compilation automatically, of course.)

Directive primitives have the effect of setting or removing *compiler directives*. A compiler directive specifies a *value* for one of the two *options* `optimize` and `depth`.

There are two types of directive: a *local directive* associates a value with an option for a specific relation; a *global directive* associates a value with an option for all relations *except* those for which a local directive exists. When compiling a procedure, the compiler checks whether a local directive exists for the relation. If so, the specified value for the option is used, otherwise it looks for a global directive. If a global directive exists, the specified value is used, otherwise the compiler uses the default value for the option.

A global directive can be changed by a call of the form:

`<-` *Option* *Value*`.`

which replaces any existing global directive for *Option* by a global directive that associates *Value* with *Option*.

A local directive can be added by a call of the form:

`<-` *Option*`(`*R*`/`*k*`,`*Value*`).`

which replaces any existing local directive for *Option* for relation *R*/*k* by a local directive that associates *Value* with *Option*.

A local directive for *Option* can be removed from relation *R*/*k* by a call:

`<-` *Option* *R*`/`*k*`.`

When a program is listed or saved (using the `listing` or `save` primitives), queries to directive primitives are embedded in the program if necessary, so that if the saved program is reloaded the state of the directives is automatically reconstructed.

The options available, and their possible values, are explained in detail in Section 13.12.

9.4 Windows

Windows and dialogs are dynamically created by Parlog for Windows for several purposes, as described in this Guide. There are also a few primitives, `crwind/5`, `cuwind/1`, and `dialog/4`, which allow Parlog programs to create windows and dialogs; see Section 13.8. Note that user-created windows should not be named by atoms that begin with a space character, since these names are reserved by the system.

9.5 Configuring Parlog for Windows

The memory space of Parlog for Windows is divided into six areas:

Backtrack stack.
Local stack.
Reset stack.
Heap (stores structured terms).
Text heap (stores atoms).
Program heap (stores program code).

The initial size of each area is fixed at the time the Parlog for Windows system is started; these are displayed in the banner (to the nearest Kbytes).

As the system is used to load and run programs some of the memory space is used up. You can find out how much space remains in each of the six areas at any time by calling the `free/6` primitive. Normally you need not be concerned about this, but eventually you may encounter one of the "memory full" errors listed in Section 11.3. When this happens, you can restart Parlog for Windows with a different configuration, increasing the size of the area that has filled up. Before doing this, you might like to use the `free/6` primitive to find out which area(s) need expanding. Most Parlog programs will not make much use of the backtrack or reset stacks, so these areas could probably be *reduced* in size.

Parlog for Windows can be configured by specifying the desired size of each of the memory areas, in "switches" that follow the word `PARLOG` on the DOS command line. These switches are:

`/b`*B*	set the backtrack stack space to	*B*-1 Kbytes.
`/l`*L*	set the local stack space to	*L*-1 Kbytes.
`/r`*R*	set the reset stack space to	*R* Kbytes.
`/h`*H*	set the heap space to	*H*-4 Kbytes.
`/t`*T*	set the text heap space to	*T*-135 Kbytes.
`/p`*P*	set the program heap space to	*P*-797 Kbytes.

Note that to set a memory area to a certain size, you have to specify a larger size in the switch. This is because the Parlog for Windows system itself uses some of each area, especially text space (135 Kbytes) and program space (797 Kbytes).

The default values are as specified by the switches:

```
/b64 /l64 /r64 /h128 /t256 /p2048
```

which provides approximately 63K backtrack stack space, 63K local stack space, 64K reset stack space, 124K heap space, 121K text heap space, and 1251K program heap space.

9.6 Porting Parlog for Windows programs to other systems

MacParlog (also a product of PLP Ltd.) is an implementation of Parlog for the Macintosh™ family of machines. Its programming environment differs somewhat from Parlog for Windows, and there are also differences between the sets of primitives supported by these systems, notably concerning I/O. However, the MacParlog compiler is basically the same as that for Parlog for Windows and, if machine-specific features are avoided, it should be straightforward to exchange source programs between these systems (although of course the disk formats are different!).

Some Parlog implementations, and also certain Parlog derivatives such as Strand and KL1, do not support "deep" guarded clauses. In these languages guards must be "flat": they may contain only calls to language primitives. If you intend to port a Parlog for Windows program to one of these non-standard languages you should check for such restrictions in advance, since they can imply a considerable effort in translation from full Parlog.

Other languages closely related to Parlog include Guarded Horn Clauses (GHC) and Concurrent Prolog (CP). The book *Programming in PARLOG* contains suggestions on how Parlog programs need to be transformed in order to run under GHC or CP.

10 Syntax and semantics

Parlog for Windows is consistent with the standard Parlog syntax and semantics used in the book *Programming in PARLOG*. However, some aspects are specific to this implementation. This section describes Parlog for Windows syntax in detail, first informally and then formally in BNF. It also explains precisely how the test-commit-output-spawn model of process behaviour which is described in the book is implemented in Parlog for Windows, including the "busy waiting" implementation of process suspension, the process-to-processor scheduling mechanism, and the role of the compiler directives.

10.1 Syntax of conjunctions

A Parlog for Windows conjunction is constructed from calls to Parlog relations, using the infix operators '`,`' (parallel conjunction) and '`&`' (sequential conjunction). In the query:

```
call1, call2.
```

call1 and `call2` are evaluated concurrently, timesharing between the calls. Bounded depth-first scheduling is used, so each call will only be executed to a certain depth in its timeslice. By using '`&`' instead of '`,`':

```
call1 & call2.
```

we can ensure that `call1` will be evaluated to completion before `call2` is started. The '`&`' operator has a higher precedence than '`,`', i.e., '`&`' is less tightly binding. So, in:

```
call1, call2 & call3.
```

call1 and `call2` are evaluated concurrently, but must terminate before `call3` is started.

Parentheses can be used to group calls in a conjunction, so '`&`' and '`,`' can be mixed arbitrarily, e.g.:

```
(call1 & call2), (call3 & call4).
```

The form of a relation call is:

```
R(t1, ..., tk)
```

where `R` is the relation name, its arity is `k`, and `t1`, _, `tk` are terms (which could be numbers, atoms, variables, lists or structures). Each call must be either a call to a Parlog primitive or to a user-defined Parlog relation.

10.2 Syntax of programs

A Parlog for Windows program comprises a set of procedures defining Parlog relations. A Parlog procedure consists of a mode declaration followed by one or more valid Parlog clauses. The mode declaration takes the form:

```
mode R(m1, ..., mk).
```

which declares a Parlog relation with name `R` and arity `k`.

Note that Parlog for Windows imposes a slight restriction on relation names: they must not begin with a space character, e.g. `' add'`; neither may you redefine a Parlog for Windows primitive by declaring a procedure with the same relation name and arity.

Each `mi` is either '?' (to indicate that this argument is input) or '^' (output). Optionally, the '?' or '^' may follow a mode identifier (an atom) as a postfix operator. These identifiers are ignored by the system; they are for the purpose of documentation only.

The clauses that follow the mode declaration take the form:

```
R(t1, _, tk) <- Guard : Body.
```

in which the `Guard` and `Body` are Parlog conjunctions, as described above. If the body is empty, it must be replaced by the symbol `true`. If the guard is empty, it can be omitted together with the '`:`'. If the guard and body are both empty, the '`<-`' operator can also be omitted.

The clauses in a Parlog procedure may be combined using '`.`' (parallel search) and '`;`' (sequential search) operators. The '`;`' operator has a higher precedence, i.e., it binds less tightly, than '`.`'. Clause groups may not be parenthesized. The role of '`;`' is to divide a procedure into two or more groups of clauses: clauses within a group are tried concurrently, while clauses in subsequent groups will not be tried until all clauses in the group are found to be non-candidates.

10.3 Examples

Here are some examples of Parlog for Windows procedures:

```
mode mults(^).
mults(X) <-
     timeslist(2,[1|X],X2),
     timeslist(3,[1|X],X3),
     timeslist(5,[1|X],X5),
     amerge(X2,X3,X23),
     amerge(X23,X5,X).


mode amerge(list1?,list2?,merged_list^).
amerge([U|X],[U|Y],[U|Z]) <-
     amerge(X,Y,Z).
amerge([U|X],[V|Y],[U|Z]) <- U < V :
     amerge(X,[V|Y],Z).
amerge([U|X],[V|Y],[V|Z]) <- V < U :
     amerge([U|X],Y,Z).


mode ontree(item?,labelled_tree?).
ontree(U,t(U,_,_)).
ontree(U,t(_,L,_)) <- ontree(U, L) : true.
ontree(U,t(_,_,R)) <- ontree(U, R) : true.
```

```
mode swrite_list(list?).
swrite_list([H|T]) <-
     data(H) &
     write(H) &
     swrite_list(T).
swrite_list([]).
```

10.4 Operators

The following operators are predeclared in the Parlog for Windows system:

```
1200  fx   :-             1200  fx   ?-
1200  xfx  :-             1200  xfx  -->
1150  fx   dynamic        1150  fx   initialization
1150  fx   meta_predicate 1150  fx   multifile
1150  fx   public         1150  fx   volatile
1100  xfy  ;              1100  xfy  |
1075  fx   <-             1075  xfx  <-
1050  fx   database       1050  fx   debuggable
1050  fx   depth          1050  fx   mode
1050  fx   optimize       1050  fx   prolog
1050  fx   version        1050  xfy  :
1050  xfy  ::             1050  xfy  :::
1050  xfy  ::::           1050  xfy  ->
1025  xfy  &              1025  xfx  when
1010  xfx  @=             1000  xfy  ,
900   fx   one            900   fy   \+
900   fy   nospy          900   fy   not
900   fy   spy            850   yfx  <~
850   yfx  ~>             700   xfx  =:=
700   xfx  =\=            700   xfx  <
700   xfx  =<             700   xfx  >
700   xfx  >=             700   xfx  =
700   xfx  \=             700   xfx  ==
700   xfx  \==            700   xfx  <=
700   xfx  =..            700   xfx  @<
700   xfx  @=<            700   xfx  @>
700   xfx  @>=            700   xfx  is
500   fx   +              500   fx   -
500   yfx  +              500   yfx  -
500   yfx  /\             500   yfx  \/
400   yfx  *              400   yfx  /
400   yfx  //             400   yfx  <<
400   yfx  >>             300   xfx  mod
10    xf   ?              10    xf   ^
```

Additionally you may define your own operators using the `op/3` primitive; see Section 13.13.

10.5 BNF syntax definition

Parlog syntax is defined here in a variant of BNF. Terminal symbols are enclosed in single quotes; non-terminal symbols have upper case initials. The metasyntactic constructs used are:

|	disjunction
[*item*]	0 or 1 occurrences of *item*
{ *item* }	0 or more occurrences of *item*

A Parlog procedure is defined as `Procedure`:

```
Procedure              = Mode_declaration '.'
                         { Parallel_group ';' }
                         Parallel_group '.'

Parallel_group         = { Clause '.' }
                         Clause

Mode_declaration       = 'mode'
                         Atom
                         [ '(' { Arg_mode ',' }
                           Arg_mode ')' ]

Arg_mode               = [ Atom ] '?'
                       | [ Atom ] '^'

Clause                 = Literal
                         [ '<-' [ Conjunction ':' ]
                           Conjunction ]

Conjunction            = { Parallel_conjunction '&' }
                         Parallel_conjunction

Parallel_conjunction   = { Unit_conjunction ',' }
                         Unit_conjunction

Unit_conjunction       = Literal
                       | '(' Conjunction ')'

Literal                = Atom
                       | Structure

Term                   = Number
                       | Atom
                       | Variable
                       | List
                       | Structure

List                   = '['
                         [ { Term ',' } Term [ '|' Term ] ]
                         ']'
```

```
Structure              =  Atom
                          '(' { Term ',' } Term ')'
```

The above BNF leaves several symbols undefined:

`Variable`: an identifier beginning with an upper case letter.
`Integer`: an integer.
`Number`: an integer or floating point number.
`Atom`: an identifier that is not a variable,
or any character string in single quotes.

Note that the `List` syntax is just shorthand for a term composed of the binary functor '`.`' and the atom `[]`. So, for example, the following equivalences hold:

```
[a]     =  .(a,[])
[a,b]   =  .(a,.(b,[]))
[a|X]   =  .(a,X)
[a,b|X] =  .(a,.(b,X))
```

There is another shorthand form. A string of characters enclosed in double quotes represents a list containing the numeric codes of those characters, for example `"bob"` is equivalent to the list `[98,111,98]`.

10.6 Operational semantics

The purpose of this section is to enable you to write Parlog for Windows programs that perform efficiently, in terms of execution speed or memory usage, and to avoid some problems that are occasionally encountered. This requires an understanding of some aspects of the Parlog for Windows implementation, which we describe first.

10.6.1 Matching and guard execution

Many Parlog clauses include some implicit matching operations (one-way unification and test unification) which are treated as part of the guard. For example, the clause:

```
r(k(X),X) <- g : b.
```

(in mode `r(?,?)`) can be rewritten as:

```
r(A1,A2) <- g' : b.
```

where `g'` includes the implicit matching calls `k(X) <= A1` and `X == A2` as well as the original guard `g`. In fact, exactly this transformation is one of the early steps in the compilation of the clause.

In standard Parlog, as described in *Programming in PARLOG*, the implicit matching calls are all concurrently evaluated with the guard, i.e., the transformed clause is:

```
r(A1,A2) <- k(X) <= A1, X == A2, g : b.
```

The effect of this is that, if any part of the matching fails, the whole guard will fail.

For the sake of efficiency, the Parlog for Windows compiler departs from this rule. The way in which the matching is added to the guard is complicated and depends on several factors, including the current setting of the `optimize` directive. For example, the arguments might be tested sequentially in some order, or a guard might suspend until all arguments it requires are available before testing any of them. The important point is that the implicit matching calls are not necessarily added in parallel with the guard. The result is that a guard might suspend in cases where, in standard Parlog, it would fail.

For example, given a procedure containing a single clause:

```
mode p(?,?,?).
p(c(X),d(X),X).
```

each of the calls:

```
p(e,A,B).
p(A,e,B).
p(c(e),d(f),X).
p(c(e),A,f).
p(A,d(e),f).
```

should fail in standard Parlog. In Parlog for Windows they either fail or suspend.

This behaviour only affects the suspension or failure of guards: a guard that succeeds in standard Parlog is guaranteed to succeed in Parlog for Windows. Normally this also means that a call which can be reduced (to some clause body) in standard Parlog will also be reduced in Parlog for Windows. However, if the sequential clause search operator ('`;`') is used in a procedure, a call may suspend (because a guard before the '`;`' is suspended) instead of reducing (using a clause after the '`;`').

To ensure that Parlog for Windows programs behave as expected, you should not make any assumptions about the order in which the implicit matching is performed. If the standard Parlog (concurrent) order is essential, it should be written explicitly in the program. For example, the above clause can be rewritten as:

```
p(A1,A2,A3) <-
     (c(X1),d(X2)) <= (A1,A2), (X1,X1) == (X2,A3) :
     true.
```

10.6.2 Concurrency in Parlog for Windows

All Parlog systems include a *scheduler*, which allocates Parlog processes to the fixed number of processors available — one, in the case of Parlog for Windows. A process might be a call in an and-parallel conjunction or a guard in an or-parallel clause search, for example. Most Parlog systems, including Parlog for Windows, provide a guarantee of fairness:

and-fairness: In a parallel conjunction, every call will be executed eventually.

or-fairness: In a parallel clause search, every guard will be executed eventually.

This ensures that a non-terminating process (which may have been created accidentally) will not cause the entire computation to hang indefinitely.

The Parlog for Windows scheduler ensures fairness by a *bounded depth-first* strategy, which is described in the next two sections.

10.6.3 Fair and-parallelism

Every and-parallel conjunction (there is one for the query, and there may also be lower-level ones inside sequential conjunctions, guards, or metacalls) is implemented by a *queue of processes*: one process for each conjunct. The scheduler repeatedly takes a process from the front of the queue, runs it for a while, then defers any unfinished processes by returning them to the rear of the queue. The latter step ensures that the subsequent processes in the queue will eventually also be executed. A *scheduler cycle* is complete when all processes in the queue have been treated in this way. To run the top-level query, the scheduler repeatedly cycles through the process queue until it is empty, or a process fails.

In more detail: a process which is a member of an and-parallel conjunction may be one of the following:

- A call.
- An uncommitted clause search.
- A sequential conjunction.
- A control metacall.

Every call has an associated *depth allowance*, which is initialized to the depth bound specified at compile time. When a call is scheduled, the system tries to find a candidate clause, as described in the next section. Assuming that the call is successfully reduced to some clause, it will be replaced at the front of the process queue by the calls in the clause body; these body calls have a depth allowance which is one less than that of the parent call. Next, the scheduler will pick the leftmost of the body calls to execute and this leftmost call will be replaced at the front of the process queue by its selected clause's body calls, and so on. Eventually the depth allowance of the calls at the front of the queue will reach zero. When the scheduler encounters a call with a depth allowance of zero it defers the call by immediately moving it to the rear of the process queue, resetting its depth allowance to the depth bound as it does so.

When a sequential conjunction is scheduled, its first conjunct (in general this is a parallel conjunction) is executed by performing one scheduler cycle through the corresponding process queue. If the queue becomes empty, the remainder of the conjunction is immediately executed, otherwise the sequential conjunction is deferred with its first conjunct in the advanced state.

When a control metacall is scheduled, one scheduler cycle is performed on the process queue representing its sub-computation. Then if the new queue is empty the metacall succeeds, otherwise it is deferred with this new queue.

10.6.4 Fair or-parallelism

Or-parallelism is implemented in a very similar way. When a call is executed, it becomes a clause search process. Such a process keeps a clause queue: a queue containing one process queue for each non-failed guard in the procedure. The clause search proceeds by executing one scheduler cycle of each process queue in turn. If some guard's process queue becomes empty, the search commits: output unification is performed and the clause body is immediately executed. If a guard fails, it is ignored, otherwise it is added with its new process queue to the rear of the clause queue. After one cycle through the clause queue, the clause search process fails if the queue is now empty, otherwise it is deferred with its new

clause queue.

The above mechanism ensures that a non-terminating guard cannot indefinitely delay the execution of a rival guard, or of a call running in and-parallel with the clause search process. (In the case of procedures with only empty guards there is no danger of this and a more efficient method is used.)

Fair or-parallelism ensures that any clause with a guard that could succeed will eventually be recognized as a candidate clause (unless another clause commits first). It says nothing about *which* candidate clause is chosen if there is more than one; in this case Parlog for Windows may be biased towards a certain clause, probably the first one textually. For a discussion of fair and biased behaviour, see Section 6.10 of *Programming in PARLOG*.

10.6.5 Busy waiting

The above account has ignored the question of how to handle suspension. In some Parlog systems a suspended process is removed from the scheduler's process queue and associated with the variable(s) it is waiting for; it will be reawoken only when one of these variables is bound. In contrast, Parlog for Windows simply adds a suspended process to the rear of the runnable queue. This means that it will be retried after one scheduler cycle, whether or not it can make any further progress. This scheduling strategy is aptly named *busy waiting*.

10.6.6 Viewing the scheduler's behaviour

Parlog for Windows's process tracer shows some aspects of the behaviour of the scheduler. The event `call r(t1,_,tn)` indicates that a call process has been scheduled. An event `suspend r(t1,_,tn)` shows the suspension of a clause search process, while `retry r(t1,_,tn)` indicates that such a process is being rescheduled. An event `reduce r(t1,_,tn)` signals the commitment of a clause search process.

Because of the busy waiting strategy, a suspended process is shown as repeatedly suspending and retrying. However, this does not necessarily indicate that the computation is frozen: between an event `retry r(t1,_,tn)` and a subsequent event `suspend r(t1,_,tn)`, the clause search process might have actually made some progress, for example, matching some of the arguments `t1,_,tn`, or partly executing some of the clause guards. An event `suspend r(t1,_,tn)` might not even mean that the clause search is suspended: it may be that a deep guard has executed as far as allowed by the depth bound, so the search is being deferred.

Whenever an uncommitted clause search process suspends, as shown by the event `suspend r(t1,_,tn)`, it remembers the exact state of its search so that when it is rescheduled no work will be repeated.

10.6.7 Tuning the scheduler's behaviour

The depth bound may be changed to any value not less than 1, using the `depth` (directive) primitive: see Section 9.3. The default value, 1, yields breadth-first execution, while a very large value approximates to depth-first execution. Adjusting the depth bound provides a way to experiment with some of the different orders of execution steps permitted by a program.

A smaller depth bound gives a more satisfying illusion of parallelism because each process is tried more frequently, and it is more economical on evaluation memory space. However, a larger depth bound is generally faster if some or all processes can do a lot of computation (as opposed to suspension) because the process switching overhead is reduced.

This is especially true if there are many other processes suspended, because it reduces the proportion of time spent in busy waiting.

10.6.8 Execution speed and how to increase it

The `ticks/1` primitive can be used to measure a call's execution time. The query:

```
ticks(Start) & my_prog & ticks(End).
```

for example, will return values for `Start` and `End` such that `End-Start` represents the approximate execution time of `my_prog` in "ticks". See Section 13.13 for more details on `ticks/1`.

There are several ways to increase the execution speed of a Parlog for Windows program. First, the depth bound can be increased, as explained above. The danger with this is that eventually the evaluation space might overflow; see below.

Second, because of the overhead of busy waiting, the number of suspended processes should be minimized. If a parent process spawns several new processes that then suspend on the same event, it is better to change the program so that the parent suspends *before* spawning the new processes. It is usually faster to sequence calls by using the sequential conjunction operator than to use mode constraints for sequencing.

Finally, deep guards should be avoided as far as possible. If the `optimize` directive is switched on (the default), Parlog for Windows performs a faster method of clause search for procedures whose guards are empty. Directives are described in Sections 9.3 and 13.12.

10.6.9 Minimizing evaluation memory use

As noted in Section 9.5, the Parlog for Windows system allocates a fixed amount of memory to any program evaluation; program execution will be aborted if this overflows, so it should be used economically.

Evaluation space is used to store essential items such as processes and data structures, but these should not normally cause an overflow unless your program is very large. Overflow is more commonly caused by (a) heavy use of deep guards, and (b) a large depth bound, both of which can consume unexpectedly large amounts of evaluation space. To avoid the problem, deep guards should be used only where necessary and the depth bound kept small; see above.

10.6.10 Minimizing code size

There are several reasons for minimizing the size of the object code to which a Parlog for Windows procedure is compiled. The obvious reason is to save program memory. Another reason is that a compile-time error such as "`Too Many Variables`" may result if the code is too large. Even if the procedure compiles successfully, its large code size may necessitate additional garbage collection during execution, which could dramatically reduce the speed of the execution.

Unfortunately, the size of the object code is not simply related to the size of the source Parlog procedure but is influenced by several factors. Specifically, the object code tends to be large if the `optimize` directive is on *and* the sequential clause search operator ('`;`') is heavily used, especially if the clause(s) before the '`;`' contain a lot of matching or large guards. To a lesser extent, code size is increased by use of the sequential conjunction operator ('`&`').

To minimize the object code size, you have three options:

1. Make sure that you do not unnecessarily use sequential search or sequential conjunction. In any case this is bad Parlog programming style. Replace them by parallel operators if at all possible.

2. If you do need to use sequential clause search, design the program so that the procedures with sequential search are as simple as possible. This again is good programming style.

3. If you encounter a compile-time error and all else fails, try switching off optimized compilation. See Sections 9.3 and 13.12 for details. The problem is that the speed advantages of optimized compilation will also be lost.

11 Error messages

This section describes the main error messages that you could get from Parlog for Windows.

Notice in particular that there is no error message for unsafe guards. (An unsafe guard is one which might bind a variable in the parent call.) Parlog for Windows leaves to the programmer the responsibility for guard safety: the system itself performs no check. If a program is allowed to run which does not have the safe guards property then the result could be failure or other unexpected behaviour, depending on the circumstances.

Also, Parlog for Windows does not provide an error message for deadlock; in the event of deadlock, the system simply hangs.

11.1 Program structure errors

Errors in the structure of a program are indicated by messages displayed in the console window. These errors are non-fatal, in the sense that the computation is not immediately aborted. The messages are of three kinds: indicating errors in the form of a query, a procedure, or a directive.

11.1.1 Query errors

Whenever a query is read, either from the supervisor or from a program file (during a `load`), Parlog for Windows checks its structure. The following message is displayed if the query *Term* is invalid: i.e., either it has unbound variables in the place of relation calls, or it has the variable output operators (’`:`’, ’`::`’, ’`:::`’, ’`::::`’) in the wrong order. After the error message is displayed, the query fails.

`*** invalid query:` *`Term`*

11.1.2 Procedure errors

Procedure errors may be displayed during the execution of `load/1`, which is the primitive that reads in and checks programs. If an erroneous term is read, the offending term is displayed in an error message; the `load` then continues.

`*** invalid declaration:` *`Term`*

Term is an invalid mode declaration: i.e., either you are attempting to redefine a Parlog for Windows primitive, or the relation name begins with a space character, or the mode specifiers are invalid.

`*** empty procedure:` *`Term`*

No clauses have been defined for the procedure declared by the mode declaration *Term*. This message will also appear if all clauses in the procedure are invalid for any reason.

`*** undeclared clause:` *`Term`*

The clause *Term* is not preceded by a mode declaration. This message may also appear as a consequence of a previous "`invalid declaration`" error.

`*** incompatible clause:` *`Term`*

The clause *Term* is incompatible with the preceding mode declaration: i.e., it has a different relation name or arity.

`*** variable clause:` *`Term`*

The clause *Term* is an unbound variable.

`*** variable head in:` *`Term`*

The clause *Term* has a head which is an unbound variable.

`*** bad 'if' operator in:` *`Term`*

The clause *Term* has an implication operator '`:-`' instead of '`<-`'.

`*** bad rhs in:` *`Term`*

The clause *Term* has an invalid right hand side (guard and body): i.e., it has unbound variables in the place of relation calls.

11.1.3 Directive errors

Invalid calls to directive primitives (Section 13.12) do not just fail: first, an explanatory message is displayed in the `syntax` window. A directive call is invalid if it is for an undefined relation (in the case of a local directive), or its value is out of range. The message displayed is:

`*** invalid directive:` *`Term`*

11.2 Compiler errors

The Parlog for Windows compiler itself should never fail, though run-time errors (see below) may exceptionally occur at compilation time. However, if the compiler does fail, the following message will be displayed in the console window:

```
!!! compiler failed
```

followed by a run-time error with code 99. This should never happen; if it does, please inform PLP.

11.3 Run-time errors

A run-time error may occur at any time. If it does, the evaluation is aborted immediately and control returns to the supervisor. An error message is displayed, of the form:

```
error(52):
Arithmetic Overflow at _000444A4 is 1 / 0
```

The message includes a numeric error code, a description of the error, and the call that is at fault. The error codes range from 0 to 35, and the code 99, and each has a corresponding

description which should be self-explanatory. A few of the most common errors are described below.

```
error(1): Backtrack Stack Full
error(2): Local Stack Full
error(3): Reset Stack Full
error(4): Heap Space Full
error(5): Text Space Full
error(6): Program Space Full
```

These indicate that one of the six memory areas has become full. Section 9.5 explains the ways around this problem.

```
error(42): Syntax Error
error(48): End Of File
```

These errors can occur at any time while Parlog for Windows is reading input. "`Syntax Error`" indicates that a term being read is syntactically illegal, while "`End Of File`" indicates that a term being read is incomplete.

Both errors are particularly likely to happen while loading a program into memory. They are more fundamental errors than those described in Section 11.1.2, since they cause the loading to be abandoned. However, if these errors occur during input performed by the Parlog for Windows system, rather than by a Parlog program, an informative message is displayed which includes the text of the term being read up to the point where the error was detected. This should guide you to the cause of the error, so that you can quickly correct it and reload the program.

```
error(45): Too Many Variables
```

This error may occur at compilation time, if the Parlog for Windows procedure being compiled is particularly complex. The causes of this error, and ways to overcome it, are explained in detail in Section 10.6.10.

For the sake of completeness, we list below all of the error messages that could be encountered. Some of them are not used in the current version of Parlog for Windows, and many others are very unlikely to occur in practice.

```
 1   Backtrack Stack Full
 2   Local Stack Full
 3   Reset Stack Full
 4   Heap Space Full
 5   Text Space Full
 6   Program Space Full
 7   External Space Full
 8   Machine Stack Full
 9   Console Space Full
10   Window Handling Error
11   Keyboard Break
12   Mouse Handling Error
13   Graphics Handling Error
20   Predicate Not Defined
21   Control Error
22   Instantiation Error
```

```
23   Type Error
24   Domain Error
25   Too Many Arguments
26   Term Too Big
30   File Handling Error
31   File Not Found
32   Path Not Found
33   Too Many Files Open
34   File Access Denied
35   Disk Full
40   Format Not Defined
41   Format Field Overflow
42   Syntax Error
43   Binary Format Error
44   Checksum Error
45   Too Many Variables
46   String Too Long
47   Atom Too Long
48   End Of File
50   Function Not Defined
51   Arithmetic Underflow
52   Arithmetic Overflow
53   Arithmetic Error
60   Instruction Not Defined
61   Bad Number Of Arguments
62   Bad Argument Type
63   Bad Register Number
64   Label Not Defined
65   Label Already Defined
66   Too Many Labels
67   Bad Form Of Clause
68   Predicate Protected
99   Parlog for Windows Compiler Error
```

12 Menus

The menus of Parlog for Windows are summarized in this section.

12.1 The `File` menu

12.1.1 The `New...` option

The '`File/New...`' option is used to create a new program window and associated disk file.

A directory dialog box is invoked, listing all files with a '.PAR' extension, to select a disk file to be associated with the new window. When a file is selected, an empty file is created with the specified name and path, and a new empty window appears on the screen. The title of the new window will be the same as the filename. If the selected file is one that already exists, a warning message will appear, asking whether the file should be replaced.

12.1.2 The `Open...` option

The '`File/Open...`' option is used to open an existing program file in a window.

A directory dialog box is invoked, listing all files with a '.PAR' extension, to select a disk file. When a file is selected, a program window appears on the screen with the same name as the file, and the contents of the file are copied to the window. If the selected file does not exist, an error message is displayed in the console window.

12.1.3 The `New Group...` option

The '`File/New Group...`' option is used to create a new program group file and open the group. This option is enabled only when no program windows are currently open.

A directory dialog box is invoked, listing all files with a '.GRP' extension, to select a disk file. When a file is selected, an empty file is created with the specified name and path, and a new empty program group is opened. This means that the group's name (the same as its filename) is added to the main window title, and all program files subsequently opened will become part of the group. If the selected file is one that already exists, a warning message will appear, asking whether the file should be replaced.

12.1.4 The `Open Group...` option

The '`File/Open Group...`' menu option is used to open an existing program group file. This option is enabled only when no program windows are currently open.

A directory dialog box is invoked, listing all files with a '.GRP' extension, to select a disk file. When a file is selected, the program group stored in the file is opened. This means that all program files belonging to the group are opened, the group's name (the same as its filename) is added to the main window title, and all program files subsequently opened will become part of the group. If the selected file does not exist, an error message is displayed in the console window.

12.1.5 The Save All option

The 'File/Save All' menu option is used to save the program windows and group (if any) that are currently open.

All open program windows that have been edited since they were last opened, or since the last 'File/Save All' command, are saved by copying them into their respective associated disk files. In addition, if a program group is open, it is saved into its associated disk file if program windows have been opened or closed since the last 'File/Save All' command. This option is enabled only when there are program windows that need saving or the open group needs saving.

12.1.6 The Close All option

The 'File/Close All' menu option is used to save the program windows and group (if any) that are currently open and close them.

This option begins by automatically performing a 'File/Save All' command. In addition, all open program windows are closed, the open program group (if any) is closed, and the internal form of the program is deleted from memory.

12.1.7 The Exit option

The 'File/Exit' menu option is used to save the program windows and group (if any) that are currently open and exit from Parlog for Windows.

The effect of this command is the same as 'File/Save All'. Following the save, the system exits.

12.2 The Edit menu

The 'Edit' menu contains the usual Windows text editing operations: Undo, Cut, Copy, Paste, Clear, and Select All. These can be used to cut and paste text within or between windows, or between Parlog and other Windows applications.

12.3 The Search menu

12.3.1 The Find option

The 'Search/Find' menu option is used to search for a text string in one or all windows.

A dialog is invoked, containing an edit box, a Find button, and three radio buttons, labelled 'Selected Text', 'Current Window', and 'All Windows'. The dialog is modeless, so it can be left on display while other operations are performed. A text string should be typed into the edit box and one of the radio buttons selected; when the Find button is clicked, the search will begin. If 'Current Window' is chosen, the search will be limited to the current window, which could be a program window, the console window, or any other window within Parlog for Windows. If 'All Windows' is chosen, all program windows (those created by the File menu options) will be searched. If the string is found, the corresponding text in a window is highlighted. If the Find button is clicked again, the search will continue for further occurrences of the text string. When there are no further

occurrences, a message will report this but the dialog will remain on display until closed.

12.4 The Run menu

12.4.1 The Load All option

The 'Run/Load All' menu option is used to load the currently open program windows into memory.

This option begins by automatically performing a 'File/Save All' command. Then, all open program windows that have been opened or edited since the last 'Run/Load All' command are loaded. This option is enabled only when there are program windows that need loading.

The loading process checks the syntax and structure of the source program, and stores it in memory in an internal form, which preserves all aspects of the source program except its layout and comments. If a procedure already exists with the same relation name and arity as one being loaded, the new procedure silently replaces the old one, with no warning message.

If a program contains syntax errors, the loading is aborted and an error message appears in the console window. If an error is found in the structure of the program, a message is again displayed in the console window but the loading will continue. (See Section 11.1.2 for an explanation of these error messages.) In either case, the 'Run/Load All' option remains enabled.

12.4.2 The Quit option

The 'Run/Quit' menu option is used to abort the execution of the current query, or other activity, and return to the supervisor prompt.

This has the same effect as typing Ctrl-break, but can be easier to use in some circumstances, particularly when dynamic windows are active.

12.4.3 The Tidy Windows option

The 'Run/Tidy Windows' menu option is used to delete all dynamically created windows.

As described in Sections 5 and 6, Parlog for Windows can display variable bindings and trace messages in dynamically created windows, whose initial size and position can be specified by the windows/5 primitive. The lifetime of these windows is the duration of a query, but they are not necessarily closed when the query terminates: this is so that their contents can be viewed at leisure. (They *are* automatically removed before the query is executed.) This menu option can be used to immediately delete the windows, reducing screen clutter, if they are no longer required.

12.5 The Window menu

12.5.1 The Cascade option

This cascades the windows that are not currently iconized, placing each window below and to the right of the previous one.

12.5.2 The `Tile` option

This tiles the windows that are not currently iconized, placing the windows adjacent to each other so that they do not overlap.

12.5.3 The `Arrange Icons` option

This arranges any iconized windows in the bottom left of the main window.

12.5.4 The lower section

The remainder of the `Window` menu contains the names of all windows. Selecting a window name from this menu will bring that window into focus. This is particularly useful if the window has been completely hidden behind other windows.

13 Primitives

The primitives of Parlog for Windows are relations which have implicit, system-defined procedures. They are documented in this section.

Primitives are grouped according to the function they perform. For each primitive is described the (implicitly declared) *modes*; the *synchronization*, that is, the conditions under which a call to the primitive suspends; and the *behaviour*, which is typically specified in terms of bindings that are made to the call arguments in those cases where the call succeeds, along with any possible side-effects. In addition, *examples* of calls are provided for some primitives.

13.1 Arithmetic primitives

The arithmetic primitives include the equality and inequality relations `=:=/2`, `=\=/2`, `</2`, `>/2`, `=</2`, `>=/2`. All of these take two arguments which are arithmetic expressions constructed using variables, numbers (real or integer), and certain predeclared operators. The expression evaluator `is/2` takes just one expression argument and evaluates it. All of these primitives suspend until their expression arguments are ground.

The operators that may appear in expressions include the arithmetic functions '+', '−', '*', '/', and `mod`, which are all infix operators, and a pseudo-random number generator `rand`: `rand(Limit)` evaluates to a random floating point number between `0` and `Limit`.

`X^ is Expression?`

Synchronization Suspends until `Expression` is ground.
Behaviour Evaluates the arithmetic expression `Expression` and unifies the result with `X`.
Examples
`X is 3+2*5-4` succeeds with `X = 9`

`E1? =:= E2?`

Synchronization Suspends until `E1` and `E2` are ground.
Behaviour Succeeds if arithmetic expressions `E1` and `E2` evaluate to the same value.

`E1? =\= E2?`

Synchronization Suspends until `E1` and `E2` are ground.
Behaviour Succeeds if arithmetic expressions `E1` and `E2` do not evaluate to the same value.

`E1? < E2?`

Synchronization Suspends until `E1` and `E2` are ground.
Behaviour Succeeds if the result of evaluating arithmetic expression `E1` is less than that of `E2`.

E1? > E2?

Synchronization Suspends until E1 and E2 are ground.
Behaviour Succeeds if the result of evaluating arithmetic expression E1 is greater than that of E2.

E1? =< E2?

Synchronization Suspends until E1 and E2 are ground.
Behaviour Succeeds if the result of evaluating arithmetic expression E1 is less than or equal to that of E2.

E1? >= E2?

Synchronization Suspends until E1 and E2 are ground.
Behaviour Succeeds if the result of evaluating arithmetic expression E1 is greater than or equal to that of E2.

13.2 Unification related primitives

Parlog for Windows supports all of the standard Parlog unification-related primitives =/2 (full unification), ==/2 (test unification), <=/2 (one-way unification), data/1, and var/1. In addition, there are three others provided for convenience: \=/2, ground/1, nonvar/1, and same/2.

Of these primitives, only ==/2, <=/2, data/1, and ground/1 ever suspend; the others always succeed or fail immediately.

Term1? = Term2?

Synchronization No suspension.
Behaviour Succeeds if terms Term1 and Term2 unify, and unifies them.

Term1? \= Term2?

Synchronization No suspension.
Behaviour Succeeds if terms Term1 and Term2 do not unify.

`Term1? == Term2?`

Synchronization Suspends if `Term1` and `Term2` could be unified only by binding variables in either term.
Behaviour This is the test unification primitive of Parlog. It unifies `Term1` and `Term2` without binding variables in either term. It succeeds if `Term1` and `Term2` are identical terms (even if they are not ground); it fails as soon as `Term1` and `Term2` fail to match.
Examples

`f(1,Y) == f(X,2)`	suspends
`f(X) == f(Y)`	suspends
`f(X) == f(X)`	succeeds
`f(1) == f(1)`	succeeds
`f(k,1) == f(Y,2)`	fails

`same(Term1?,Term2?)`

Synchronization No suspension.
Behaviour This performs an immediate, non-suspending check of whether two terms are identical. It succeeds if `Term1` and `Term2` are identical terms (even if they are not ground), otherwise it fails.
Examples

`f(1,Y) == f(X,2)`	fails
`f(X) == f(Y)`	fails
`f(X) == f(X)`	succeeds
`f(1) == f(1)`	succeeds
`f(k,1) == f(Y,2)`	fails

`Term1? <= Term2?`

Synchronization Suspends if `Term1` and `Term2` could be unified only by binding variables in `Term2`.
Behaviour This is the one-way unification, or matching, primitive of Parlog. Unifies `Term1` and `Term2` without binding variables in `Term2`. It succeeds if `Term2` is a substitution instance of `Term1`; it fails as soon as `Term1` and `Term2` fail to match.
Examples

`f(1) <= f(X)`	suspends
`f(X) <= f(1)`	succeeds with `X = 1`
`f(X) <= f(Y)`	succeeds with `X = Y`
`f(k,1) <= f(Y,2)`	fails

`var(Term?)`

Synchronization No suspension.
Behaviour Succeeds if `Term` is an unbound variable, fails if `Term` is instantiated.

`nonvar(Term?)`

Synchronization No suspension.
Behaviour Succeeds if `Term` is instantiated, fails if `Term` is an unbound variable.

`data(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds.

`ground(Term?)`

Synchronization Suspends until `Term` is ground.
Behaviour Succeeds.

13.3 Type checking primitives

The type checking primitives are `atom/1`, `integer/1`, `float/1`, `number/1`, and `atomic/1`. Each suspends until its argument is instantiated, when the primitive succeeds or fails according to the argument's type.

`atom(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds if `Term` is an atom.

`integer(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds if `Term` is an integer (i.e., has a zero fraction part).

`float(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds if `Term` is a floating point number (i.e., has a non-zero fraction part).

`number(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds if `Term` is a number, either integer or floating point.

`atomic(Term?)`

Synchronization Suspends until `Term` is instantiated.
Behaviour Succeeds if `Term` is either an atom or a number.

13.4 Metalevel primitives

The metalevel primitives include those which are used to construct and dissect terms. They include the structure manipulators `arg/3` and `functor/3`, the list/structure converter `=../2`, the string/character code list converter `name/2`, and an atom concatenating primitive `cat/2`.

Another group deals with variables inside terms: `varsin/2` can be used to find out the variables in a term, and `toground/3` replaces a term's variables. Both of these, like `var/1` and `nonvar/1`, should be used carefully, as their behaviour depends upon the current binding state of a term, which will change if its variables are bound by another process. `tohollow/3` performs the converse function of `toground/3`: they can be used together to convert between ground and hollow forms of a term.

`arg(N?,Term?,Arg^)`

Synchronization Suspends until `N` and `Term` are instantiated.
Behaviour Unifies `Arg` with the `N`th argument of `Term`.

`functor(Term?,Functor?,Arity?)`

Synchronization Suspends until `Term` is instantiated or `Functor` and `Arity` are instantiated.
Behaviour If `Term` is instantiated, `Functor` and `Arity` will be unified with the function name and arity, respectively, of `Term`.
If `Functor` and `Arity` are instantiated, `Term` will be unified with the most general term having the specified function name and arity.

`Term? =.. List?`

Synchronization Suspends until `Term` is instantiated or `List` is bound to a finite-length list whose head is instantiated.
Behaviour If `Term` is instantiated, `List` is unified with a list whose head is the functor of `Term` and whose remaining members are the arguments of `Term`.
If `List` is instantiated to a finite-length list whose head is instantiated, `Term` is unified with a term whose functor is the head of `List` and whose arguments are the remaining members of `List`.
Examples

`T =.. [likes,bob,Z]`	succeeds with `T = likes(bob,Z)`
`foo(a,b,c) =.. L`	succeeds with `L = [foo,a,b,c]`
`T =.. [X|Y]`	suspends

name(Atomic?,String?)

Synchronization Suspends until `Atomic` is instantiated or `String` is ground.
Behaviour If `Atomic` is instantiated (it must be an atom or number), unifies `String` with the list of character codes in its name.
If `String` is ground (it must be a list of character codes), unifies `Atomic` with the number, if possible, otherwise the atom, containing those characters.
Examples

`name(bob,S)`	succeeds with `S = [98,111,98]`
`name(X,"bob")`	succeeds with `X = bob`
`name(X,"123")`	succeeds with `X = 123`
`name(Char,[65])`	succeeds with `Char = 'A'`
`name(A,S)`	suspends

cat(Atoms?,Atom^)

Synchronization Suspends until `Atoms` is ground.
Behaviour Unifies `Atom` with an atom constructed by concatenating all atoms in the list `Atoms`.

varsin(Term?,Vars^)

Synchronization No suspension.
Behaviour Unifies `Vars` with a list of all of the variables in term `Term`. Each variable will appear exactly once in `Vars`.
Examples

`varsin(bob,V)`	succeeds with `V = []`
`varsin(f(X,Y,a,X),V)`	succeeds with `V = [X,Y]`

toground(Term?,Gterm^,Varnames^)

Synchronization No suspension.
Behaviour Unifies `Gterm` with a ground copy of term `Term`, constructed by replacing each of the variables in `Term` by an atom taken successively from the sequence 'A', 'B', 'C', etc. Variables that appear only once in `Term` are replaced by the atom '_'. A list of the variable names (atoms) used is unified with `Varnames`.
Examples

`toground(X,G,V)`	succeeds with `G = '_', V = ['_']`
`toground(f(X,Y,a,X),G,V)`	succeeds with `G = f('A','_',a,'A'),` `V = ['A','_']`

`toholLow(Gterm?,Term^,Varnames?)`

Synchronization Suspends until `Gterm` and `Varnames` are ground.
Behaviour Unifies `Term` with a "hollow" copy of term `Gterm`, constructed by replacing each of the atoms in `Gterm` that also appear in `Varnames` (a list of atoms) by a variable; all occurrences of an atom are replaced by the same variable. Also, the '_' atom is replaced by a unique variable, regardless of whether it appears in `Varnames`.
Examples

`tohollow('_',T,[])` succeeds with `T` unbound
`tohollow(f('A','_',a,'A'),T,['A','_'])`
succeeds with `T = f(X,Y,a,X)`

13.5 Control primitives

The control primitives are those which (with the exception of `true/0` and `fail/0`) take arguments which represent relation calls to be executed. The standard `call/1` primitive does no more than evaluate the call and then succeed or fail depending on the outcome. `call/3` is the *control metacall* which never fails and which makes possible tight supervision over an evaluation. Both versions of `call` suspend until their arguments are terms that could represent valid Parlog for Windows conjunctions (i.e., conjunctions which do not have unbound variables in the place of relation calls), as does `not/1`, which implements negation as failure.

`true`

Synchronization No suspension.
Behaviour Succeeds.

`fail`

Synchronization No suspension.
Behaviour Fails.

`not Conj?`

Synchronization Suspends until `Conj` is bound to a valid Parlog conjunction.
Behaviour Evaluates the Parlog conjunction `Conj`. Succeeds if `Conj` fails, fails if `Conj` succeeds.

`call(Conj?)`

Synchronization Suspends until `Conj` is bound to a valid Parlog conjunction.
Behaviour Evaluates the Parlog conjunction `Conj`. Succeeds if `Conj` succeeds, fails if `Conj` fails.

`call(Conj?,Status^,Control?)`

Synchronization Suspends until `Conj` is bound to a valid Parlog conjunction.
Behaviour Evaluates the Parlog conjunction `Conj`. If `Conj` succeeds, `Status` is unified with the atom `succeeded`. If `Conj` fails, `Status` is unified with the atom `failed`.
If `Control` is bound to the atom `stop`, the evaluation of `Conj` is aborted and `Control` is unified with the atom `stopped`.
If `Control` is bound to a list `[suspend|C]`, `Status` is unified with `[suspend|S]` and the evaluation of `Conj` is suspended if it is active. If `Control` is bound to a list `[continue|C]`, `Status` is unified with `[continue|S]` and the evaluation of `Conj` is reactivated if it is suspended. In either case the evaluation proceeds with `S` and `C` replacing `Status` and `Control`.
If `Control` is bound to any other term, the metacall fails.

13.6 Database primitives

The `set/3` and `subset/3` primitives implement the interface between "single-solution" Parlog relations and "all-solutions" database relations.

`set(Solnlist^,Term?,Dbcall?)`

Synchronization Suspends until `Dbcall` is instantiated.
Behaviour This is the eager set constructor primitive. The behaviour of the call is to bind `Solnlist` to a complete list of terms each representing an individual solution to the database relation call `Dbcall`. The relation for `Dbcall` is expected to be defined by code compiled from a modeless set of assertions (unit clauses each terminated by a period) declared by a `database` declaration. If no such code exists, or if there are no solutions to `Dbcall`, then `Solnlist` will return the empty list.
The evaluation of a call to `set/3` is a sequential search through the defining clauses for the relation, with full unification used for all pairs of arguments. During this evaluation (which is quite fast) no concurrent activity with any other process is performed.
Examples
If `likes/2` is a database relation with the definition:

```
likes(jill,mary).
likes(karen,jill).
likes(jill,ian).
```

then the computed solution for `S` to the call:

```
set(S,Person,likes(jill,Person))
```

is:

```
S = [mary,ian].
```

subset(Solnlist?,Term?,Dbcall?).

Synchronization Suspends until `Dbcall` is instantiated.
Behaviour This is the lazy set constructor primitive. As with `set/3`, `Solnlist` is a list of terms each representing an individual solution to the database relation call `Dbcall`. However, with `subset/3` the terms are produced *lazily*, that is, they are produced only in response to demands represented by variables supplied for `Solnlist`, and the final value of `Solnlist` may represent only a proper subset of the complete set of solutions. A call to `subset/3` succeeds and terminates when the consumer process closes this list.
If the supply of variables for `Solnlist` exceeds the availability of solutions then the surplus variables are bound to the constant `end`.
The evaluation of a call to `subset/3` is a lazy sequential search through the defining clauses for the relation, with full unification used for all pairs of arguments. The progress of the evaluation is constrained by the supply of demand variables and, in general, other processes can be active concurrently with `subset/3`.
Example
If `likes/2` is a database relation with the definition:

```
likes(jill,mary).
likes(karen,jill).
likes(jill,ian).
```

then the computed solution to the query:

```
S = [P1,P2,P3], subset(S,Person,likes(jill,Person)).
```

is:

```
P1 = mary
P2 = ian
P3 = end
```

13.7 Input and output primitives

Most of the input and output primitives of Parlog for Windows are available in two forms: one that does I/O to the default channel and another to do the I/O to a named channel. The default channel for input and output is the console window. The second form of I/O primitive takes an extra argument which is the name of the desired channel. A channel name can be the name of a file or a window, or the reserved atom `user`, which names the default channel. A named file should be opened before, and closed after, performing I/O to it; see Section 13.8.

There are four main input primitives: the term input primitive `read`, and the character-oriented primitives `get0`, `get`, and `skip` (all of which are available in both one-argument and two-argument forms). Another primitive, `gread/2`, is the same as `read/2` except that it treats any variables in terms as atoms.

All of these can be used to input from files, windows, or the default channel `user`. When reading from `user`, the input is actually done via the console window: the user can type the input and edit it, typing `Return` to terminate. In contrast, the `getkey/1` primitive obtains its input (a character) invisibly: it is not displayed in any window and cannot be edited before it is consumed by the system.

All of the input primitives cause the entire Parlog for Windows system to suspend until the specified input is complete. However, there is a special primitive `key/1`, which suspends until any key is pressed; this is useful because its suspension will not cause the

suspension of other concurrent processes.

A higher-level facility which provides asynchronous incremental input of streams is provided by the `in_stream/1` and `in_streams/1` primitives; these are also described in some detail in Section 7. They are intended to give support to program testing and debugging, rather than acting for application software I/O.

There are six main output primitives: the term output primitives `write`, `writeq`, and `display`, and the character-oriented primitives `nl`, `put`, and `tab` (all of which are available in two forms: with and without a channel argument). The other one, `incwrite/2`, outputs a ground term incrementally to a named channel, suspending whenever it reaches an unbound variable.

`read(Term^)`

Synchronization No suspension.
Behaviour Unifies `Term` with the next term read from `user`, i.e., from the console window. Until the input is complete, any concurrent processes will be suspended.

`read(Channel?,Term^)`

Synchronization Suspends until `Channel` is instantiated.
Behaviour Unifies `Term` with the next term read from `Channel`. A call `read(user,Term)` is equivalent to `read(Term)`. For a file or window, the atom `end_of_file` is returned when the end of the channel has been reached.

`getkey(N^)`

Synchronization No suspension.
Behaviour Reads the next character directly from the keyboard, i.e., without echoing it and without going through the console window, and unifies the character code of this with `N`. Until the input is complete, any concurrent processes will be suspended.

`get0(N^)`

Synchronization No suspension.
Behaviour Reads the next character from `user`, i.e., from the console window, and unifies the character code of this with `N`. Until the input is complete, any concurrent processes will be suspended.

`get0(Channel?,N^)`

Synchronization Suspends until `Channel` is instantiated.
Behaviour Reads the next character from `Channel` and unifies the character code of this with `N`. If the end of `Channel` has been reached, in the case of a file or window, `N` is unified with -1. A call `get0(user,N)` is equivalent to `get0(N)`.

get(N^)

Synchronization No suspension.
Behaviour Reads the next character that is not a space or control character from `user`, i.e., from the console window, and unifies the character code of this with `N`. Until the input is complete, any concurrent processes will be suspended.

get(Channel?,N^)

Synchronization Suspends until `Channel` is instantiated.
Behaviour Reads the next character that is not a space or control character from `Channel` and unifies the character code of this with `N`. If the end of `Channel` has been reached, in the case of a file or window, `N` is unified with -1. A call `get(user,N)` is equivalent to `get(N)`.

skip(N?)

Synchronization Suspends until `N` is ground.
Behaviour Reads characters from `user`, i.e., from the console window, until the next character whose ASCII code is `N`. `N` must be either an integer or an arithmetic expression that evaluates to an integer; a particularly useful form of this is a single-character double-quoted string. For example, `skip(".")` is equivalent to `skip(46)` and will advance to immediately after the next occurrence of the period character.

skip(Channel?,N?)

Synchronization Suspends until `Channel` is instantiated and `N` is ground.
Behaviour Advances the channel pointer for `Channel` to immediately after the next character whose ASCII code is `N`. `N` is either an integer or an arithmetic expression that evaluates to an integer. A call `skip(user,N)` is equivalent to `skip(N)`.

gread(Channel?,Term^)

Synchronization Suspends until `Channel` is instantiated.
Behaviour Unifies `Term` with a "grounded" (variable-free) copy of the next term read from `Channel`. Differs from `read` in that variables such as `X` are read not as variables but as atoms with the same name (e.g. `'X'`). For a file or window the atom `end_of_file` is returned when the end of the channel has been reached.

key(N^)

Synchronization Suspends until any key is pressed.
Behaviour Reads the next character directly from the keyboard, i.e., without echoing it and without going through the console window, and unifies the character code of this with `N`. Unlike `getkey/1`, this primitive does not force the suspension of other processes.

in_stream(Stream^)

Synchronization No suspension.
Behaviour Binds Stream to a list containing terms typed at the keyboard. The terms may be supplied incrementally into an input dialog which is summoned when any key is pressed.
The user may type any term into this dialog. When the OK button is clicked, the dialog is removed and the terms are entered into the stream. If instead Cancel is clicked, the dialog is removed without affecting the stream in any way. In either of these cases, the input dialog can be restored at any time in the future by pressing any key. If End is clicked, the stream is closed (ignoring any term typed) and the in_stream/1 process terminates.
Use the in_stream/1 primitive (instead of read/1, for example) whenever you want to test a program which is to be incrementally supplied with data. This primitive is "concurrent friendly" in that other processes, possibly including one or more processes which are consumers of Stream, may be active concurrently with in_stream/1 during the periods when the input dialog is not on display.
The system definition of in_stream/1 includes a call to key/1 (described in this section). This means that it is not normally useful to have more than one in_stream/1 process, or an in_stream/1 with a key/1 process, active concurrently. Use in_streams/1 to program multiple stream keyboard input.

in_streams(Streams?)

Synchronization Suspends until Streams is bound to a finite-length list all of whose members are instantiated.
Behaviour This primitive is a multiple-stream version of in_stream/1. Streams should be a list of terms each of the form c(C) where c is a single character atom representing a "wakeup" key and C is the variable which is to receive the corresponding binding. No two terms in the list may have the same wakeup key. The list may include a term other(V), which *must* appear at the end of the list; then every key other than those named elsewhere on the list acts as a wakeup key for the variable V. The call fails unless Streams specifies a list of this form.
To enter terms onto one of the streams, press its wakeup key. For example, if Streams includes the term b(B), typing 'b' will produce an input dialog into which a term can be typed; this term will be output on list B. Similarly, list B can be closed by typing 'b' and then clicking End.
Once a stream has been closed, the in_streams/1 process ceases to recognize that stream's wakeup key. An in_streams/1 process will terminate only when all of its streams have been closed by the user.
It is not useful to have an in_streams/1 process active concurrently with another in_streams/1 process, or an in_stream/1 process, or a key/1 process.

write(Term?)

Synchronization No suspension.
Behaviour Writes the term Term to user, i.e., to the console window. Current operator declarations are reflected. Atoms that would need to be quoted on input (such as 'a name' or 'Henry') are not quoted. Variables are displayed as underscore names.

write(Channel?,Term?)

Synchronization Suspends until Channel is instantiated.
Behaviour Writes the term Term to Channel, in the same format as write/1. A call write(user,Term) is equivalent to write(Term).

writeq(Term?)

Synchronization No suspension.
Behaviour Writes the term Term to user, i.e., to the console window. Atoms are quoted if necessary, so that they can be read back in by read, and current operator declarations are reflected.

writeq(Channel?,Term?)

Synchronization Suspends until Channel is instantiated.
Behaviour Writes the term Term to Channel, in the same format as writeq/1. A call writeq(user,Term) is equivalent to writeq(Term).

display(Term?)

Synchronization No suspension.
Behaviour Writes the term Term to user, i.e., to the console window, in prefix notation, i.e., ignoring operator declarations, and with atoms quoted if necessary, so that they can be read back in by read.

display(Channel?,Term?)

Synchronization Suspends until Channel is instantiated.
Behaviour Writes the term Term to Channel, in the same format as display/1. A call display(user,Term) is equivalent to display(Term).

nl

Synchronization No suspension.
Behaviour Starts a new line on user, i.e., the console window.

nl(Channel?)

Synchronization Suspends until Channel is instantiated.
Behaviour Starts a new line on Channel. A call nl(user) is equivalent to nl.

put(N?)

Synchronization Suspends until N is ground.
Behaviour Writes the character whose ASCII code is N to user, i.e., to the console window. N is either an integer or an arithmetic expression that evaluates to an integer; a particularly useful form of this is a single-character double-quoted string. E.g., put(".") is equivalent to put(46) and will output a period character.

put(Channel?,N?)

Synchronization Suspends until Channel is instantiated and N is ground.
Behaviour Writes the character whose ASCII code is N to Channel. N is either an integer or an arithmetic expression that evaluates to an integer. A call put(user,N) is equivalent to put(N).

tab(N?)

Synchronization Suspends until N is ground.
Behaviour Writes N spaces to user, i.e., to the console window. N is either an integer or an arithmetic expression that evaluates to an integer.

tab(Channel?,N?)

Synchronization Suspends until Channel is instantiated and N is ground.
Behaviour Writes N spaces to Channel. N is either an integer or an arithmetic expression that evaluates to an integer. A call tab(user,N) is equivalent to tab(N).

incwrite(Channel?,Term?)

Synchronization Suspends until Channel is instantiated.
Behaviour Writes Term to Channel incrementally, suspending if it attempts to write an unbound variable, until the variable is bound. The format used is the same as display/2, i.e., current operator declarations are ignored and atoms are quoted if necessary, so that the term can be read back in by read.

13.8 File, window, and dialog handling primitives

The primitives below are concerned with providing access to files and windows rather than actually performing the I/O (for which see Section 13.7).

To perform file input and output, the name of the file (which must first be opened by a call to open/1 or create/1, and closed afterwards) must appear as the first argument of one of the I/O primitives.

Output to a window is done in a similar way, opening the window first by crwind/5 and closing it afterwards, by close/1.

Another primitive in this section, dialog/4, allows the creation of a simple dialog, which is extremely useful for multi-window applications, while title/1 allows the main window title to be changed.

open(File?)

Synchronization Suspends until `File` is instantiated.
Behaviour Opens `File`, which must name a file that already exists, for read access.

create(File?)

Synchronization Suspends until `File` is instantiated.
Behaviour Creates a new file `File` and opens it for write access. If a file of that name already existed, it is replaced: *no* backup copy of it is created.

crwind(Name?,R?,C?,Depth?,Width?)

Synchronization Suspends until `Name`, `R`, `C`, `Depth`, and `Width` are ground.
Behaviour Creates a new window named `Name`, with depth `Depth` rows and width `Width` columns, its top left corner positioned at row `R`, column `C`. Note that the window coordinates are rows and columns, rather than pixels. The screen area is considered to comprise 25 rows, numbered from 0 (top) to 24 (bottom), and 80 columns, numbered from 0 (left) to 79 (right). A call to `crwind/5` does not change the current window.

cuwind(Name?)

Synchronization Suspends until `Name` is instantiated.
Behaviour Changes the current window to `Name`. The default current window is the console window, which is named by the integer 1, so to return to normal, call `cuwind(1)`.

close(Name?)

Synchronization Suspends until `Name` is instantiated.
Behaviour Closes the file or window named `Name`. Always use this primitive when you are finished writing to a file. The effect of closing a window is to delete the window and all of its contents.

dialog(Name?,Message?,Options?,Selection^)

Synchronization Suspends until `Name`, `Message`, and `Options` are ground.
Behaviour Creates a modeless dialog named `Name` (an atom), which displays a message and a number of buttons. The message is the concatenation of the ground terms contained in the list `Message`. There is one button for each member of the list `Options`, each labelled by the corresponding member (an atom) of `Options`.
After displaying the dialog, Parlog for Windows waits until one of the buttons is clicked, and then unifies `Selection` with the button label (an atom) for the clicked button.

`title(Title?)`

Synchronization Suspends until `Title` is instantiated.
Behaviour Replaces the title of the main Parlog for Windows window by `Title`, an atom.

13.9 Program handling primitives

This section describes primitives that allow you to add, delete, inspect, and edit source programs. Many of them take a `Relations` argument to specify the relation(s) to be listed, etc. This argument takes the form *Relation name/Arity*, or *Relation name*, or a list of these, e.g.:

```
filter
merge/3
[go/0,filter,merge/3]
```

Any relations named may be either Parlog relations or database relations.

`load(Channel?)`

Synchronization Suspends until `Channel` is instantiated.
Behaviour Reads in a Parlog for Windows program from `Channel`, which must be `user` or the name of a file. If no file extension is specified, '.PAR' is assumed. The syntax and structure of a program is checked as it is read in; any syntax error will abort the `load/1` process, whereas errors in the structure of the program simply cause an error message to be displayed (in the console window) and the offending term ignored.
If `Channel` is `user` you can type a program into the console window. Type `Escape` to terminate the editing and load the program; this is not recommended, as the source program is not stored in a disk file and can easily be lost.
A Parlog for Windows program may comprise Parlog procedures (preceded by a `mode` declaration), database procedures (preceded by a `database` declaration), and queries. If a program contains a procedure for a relation that is already defined, the existing procedure will be replaced by the new one, without warning.
A query in a program (a term of the form):
`<-` *Conjunction* `.`
is executed at the time that the program is loaded. No indication is given of its success or failure, though an error will cause the `load/1` call to be abandoned with an appropriate error message.

`listing`

Synchronization No suspension.
Behaviour Lists the whole of the loaded program to the console window. Global directives are listed, as queries, at the beginning of the program. Local directives associated with a relation are listed immediately after the procedure for that relation.

`listing(Relations?)`

Synchronization Suspends until `Relations` is ground.
Behaviour Lists the procedures for those relations specified by `Relations` that are currently defined, to the console window. Global directives are not listed; local directives associated with a relation are listed immediately after the procedure for that relation.

`save(Channel?)`

Synchronization Suspends until `Channel` is instantiated.
Behaviour Lists the whole of the loaded program to `Channel`, in the same format as `listing/0`. `Channel` must be `user` or the name of a file. If no file extension is specified, '.PAR' is assumed. `save(user)` is equivalent to `listing`.

`save(Channel?,Relations?)`

Synchronization Suspends until `Channel` is instantiated and `Relations` is ground.
Behaviour Lists the procedures for those relations specified by `Relations` that are currently loaded, to `Channel`, in the same format as `listing/1`. `Channel` must be `user` or the name of a file. If no file extension is specified, '.PAR' is assumed. `save(user,Relations)` is equivalent to `listing(Relations)`.

`kill(Relations?)`

Synchronization Suspends until `Relations` is ground.
Behaviour Deletes procedures, and local directives, for the relations specified in `Relations`.

`reinitialize`

Synchronization No suspension.
Behaviour Deletes the entire program currently defined, including all directives. Also resets the Parlog for Windows system to its initial state, except that (unlike the '`File/Close All`' menu option) program windows are not closed.

`defined(Relation?)`

Synchronization Suspends until `Relation` is ground.
Behaviour Succeeds if `Relation` is a term of the form *Relation name/Arity* that names a relation that is currently defined by a Parlog procedure. Note that this primitive succeeds only for user-defined Parlog relations, not for primitives or database relations.

13.10 Compilation primitives

The compilation of Parlog for Windows programs is normally invisible but, for convenience,

a `compile/0` primitive is provided, to force immediate compilation. Another primitive affecting compilation is `fastcode/0`.

Neither of these primitives is necessary, but `compile/0` can be useful to avoid the distraction of lazy compilation, while `fastcode/0` can be used to increase execution speed following a debugging session.

`compile`

Synchronization No suspension.
Behaviour Compiles all relations currently defined that need compiling. A relation needs compiling if its source form and object form are inconsistent, i.e., if, since the relation was last compiled:

- its procedure has been changed (by means of `load`), or
- a directive affecting the relation has been changed, or
- channel spypoints have been set or removed on this relation, or
- trace mode has been switched on, or
- process spypoints have been set for the first time (on any relation).

`fastcode`

Synchronization No suspension.
Behaviour Causes the Parlog for Windows compiler to generate regular (rather than traceable; see Section 6.12.3) code in future compilations, though it does not immediately do the compilation. It will have no effect if trace mode is switched on or if any process spypoints are set, because process tracing requires the presence of traceable code. It is worth calling this primitive after a process tracing session has ended, because regular code is faster than traceable code, and the change is not made automatically.

13.11 Debugging primitives

The debugging facilities of Parlog for Windows are described in detail in Section 6. Here, we summarize the primitives that are used to control the action of the debugger. These include primitives to switch various debugging parameters on and off and to configure the trace model. The other primitive described in this section is `windows/5`, which can be used to configure the layout of the windows that are dynamically created for both tracing and dynamic output of variable bindings.

The default values of the tracing parameters are as though the following calls were executed when the Parlog for Windows system was (re)initialized:

```
notrace,
debug,
nospyall,
debug_options(on,on,on),
window_debug,
windows(1,5,40,4,39).
```

`trace`

Synchronization No suspension.
Behaviour Switches on trace mode. When the system is in trace mode, the process tracer is invoked for each query issued from the supervisor, except for queries comprising just a single primitive call. Queries embedded in programs are not traced.

`notrace`

Synchronization No suspension.
Behaviour Switches off trace mode.

`debug`

Synchronization No suspension.
Behaviour Switches on debug mode. When the system is in debug mode, both process and channel spypoints are observed, otherwise they have no effect.

`nodebug`

Synchronization No suspension.
Behaviour Switches off debug mode.

`spy Relation?`

Synchronization Suspends until `Relation` is instantiated.
Behaviour Sets a process spypoint on the relation(s) specified by `Relation`. If `Relation` takes the form *Relation name/Arity* a spypoint is set on the specified relation if it is defined (not a primitive); if `Relation` is just a relation name, a spypoint is set on all currently defined relations with the given name, regardless of their arity. If `Relation` is any other term, a conditional process spypoint is added, to trace calls that unify with the term.

`nospy Relation?`

Synchronization Suspends until `Relation` is instantiated.
Behaviour Removes process spypoints and channel spypoints from the relation(s) specified by `Relation`. `Relation` may take the form *Relation name/Arity*, or just a relation name, in which case spypoints are removed from all relations with the given name, regardless of their arity. If `Relation` is any other term, all conditional process spypoints unifying with the term are removed.

`nospyall`

Synchronization No suspension.
Behaviour Removes all (process and channel) spypoints.

debug_options(Suspend?,Reduce?,Succeed?)

Synchronization Suspends until `Suspend`, `Reduce`, and `Succeed` are ground.
Behaviour Each of `Suspend`, `Reduce`, `Succeed` must be either `on` or `off`; otherwise the call fails. This primitive is used to configure the trace model (see Section 6.12.2). If `Suspend` is `on`, `suspend` and `retry` events are traced. If `Reduce` is `on`, `reduce` events are traced. If `Succeed` is `on`, `succeed` and `fail` events are traced. In any case, `call` events are always traced.

window_debug

Synchronization No suspension.
Behaviour Switches on window-debug mode. When the system is in this mode, all trace messages are displayed in dynamically created windows, one window for each process.

nowindow_debug

Synchronization No suspension.
Behaviour Switches off window-debug mode. Trace messages are now all displayed in the current window. This is more confusing than using dynamic windows, but it has the advantage that the relative speeds of the processes are more obvious. Also, it saves memory.

windows(S?,R?,C?,Depth?,Width?)

Synchronization Suspends until `S`, `R`, `C`, `Depth`, and `Width` are ground.
Behaviour `S`, `R`, and `C` must be non-negative integers, `Depth` must be an integer between 1 and 24 inclusive, `Width` must be between 1 and 79 inclusive; otherwise the call fails.
This primitive is used to configure the size and positions of the dynamically created windows. These windows are used both for trace messages (when window-debug mode is switched on), and for dynamic display of variable bindings in incremental or film format; see Sections 5 and 6.
`Depth` and `Width` specify the size of the scrolling area of each window, i.e., the size excluding borders. The other three arguments control the positioning of the windows. The first dynamic window is always placed at the top left corner of the screen. If `C` is not 0, subsequent windows are placed in the same vertical position, but each is shifted right by `C` columns.
If `C` is 0, or if the windows have reached the right side of the screen, the next window will be placed at the left of the screen, but shifted down by `R` rows (unless `R` is 0).
If `R` is 0, or if the windows have reached the bottom of the screen, the next window will be placed at the left of the screen near the top, but shifted downward by `S` rows relative to the topmost window(s) on the previous screen.

13.12 Directive primitives

As explained in Section 9.3, the directive primitives are used to set or remove global or local

directives, which associate a *value* with one of two *options*: `optimize` and `depth`.

These options, and their possible values, are explained in this section. If a directive primitive is called with an invalid value, or (in the case of a local directive) a relation that is not defined, an error message "invalid directive" is displayed in the console window, and the call then fails.

The default values of the directives are as though the following calls were executed when the Parlog for Windows system was (re)initialized:

```
(optimize on), (depth 1).
```

`optimize Value?`

Synchronization Suspends until `Value` is ground.
Behaviour If `Value` takes the form *Relation name*/*Arity*, any local directive for `optimize` will be removed from the named relation.
Otherwise, `Value` must be `on` or `off`; this is a global directive. `optimize on` causes the compiler to perform certain optimizations when compiling procedures; these usually increase the run-time speed and reduce the size of the object code generated. `optimize off` causes unoptimized code to be generated; there should normally be no reason to select this, but see Section 10.6.10.

`optimize(Relation?,Value?)`

Synchronization Suspends until `Relation` is ground and `Value` is instantiated.
Behaviour `Relation`, of the form *Relation name*/*Arity*, must name a relation that is currently defined. `Value` must be `on` or `off`. `optimize(Relation,on)` or `optimize(Relation,off)` adds a local directive that determines whether optimized code is generated for `Relation`, and overrides any global directive for `optimize`.

`depth Value?`

Synchronization Suspends until `Value` is ground.
Behaviour If `Value` takes the form *Relation name*/*Arity*, any local directive for `depth` will be removed from the named relation.
Otherwise, `Value` must be a positive integer; this is a global directive. The code will be compiled in such a way that the evaluation depth (see Section 10.6.7) of each call is bounded to the depth specified by `Value`.

`depth(Relation?,Value?)`

Synchronization Suspends until `Relation` is ground and `Value` is instantiated.
Behaviour `Relation`, of the form *Relation name*/*Arity*, must name a relation that is currently defined. `Value` must be a positive integer. `depth(Relation,Value)` adds a local directive that sets the evaluation depth (see Section 10.6.7) to `Value` for `Relation`, and overrides any global directive for `depth`.

13.13 Miscellaneous primitives

This section documents primitives which have not been described elsewhere.

`free(B^,L^,R^,H^,T^,P^)`

Synchronization No suspension.
Behaviour Returns the amount of memory space (in Kbytes) remaining in each of the four areas: `B` (backtrack stack space), `L` (local stack space), `R` (reset stack space), `H` (heap space), `T` (text space), `P` (program space).

`remember(Id?,Term?)`

Synchronization Suspends until `Id` is instantiated.
Behaviour Associates `Term`, an arbitrary term, with the atom `Id`; it can subsequently be retrieved by `recall/2`. Any variables in `Term` are copied (replaced by new variables). This provides a form of global assignment.

`recall(Id?,Term^)`

Synchronization Suspends until `Id` is instantiated.
Behaviour Unifies `Term` with the term that is currently associated with the atom `Id`; this must have been done previously by `remember/2`. Fails if there is no term associated with `Id`.

`op(Precedence?,Type?,Name?)`

Synchronization Suspends until `Precedence` and `Type` are instantiated and `Name` is ground.
Behaviour Declares the atom `Name` to be an operator with the specified precedence and type. Type should be one of `fx`, `fy`, `xf`, `yf`, `xfx`, `xfy`, `yfx`. `Precedence` should be an integer between 1 and 1200. `Name` should be an atom, or a list of atoms, in which case all atoms in the list are declared as operators with the same precedence and type.
The Parlog for Windows system starts with many operators already declared. These are listed in Section 10.4.

`current_op(Precedence?,Type?,Name?,Ops^)`

Synchronization No suspension.
Behaviour Finds all (or some) of the currently declared operators. `Ops` is unified with a list containing a term `op(P,T,N)` for each declared operator with precedence `P`, type `T` and name `N` such that `op(P,T,N)` unifies with `op(Precedence,Type,Name)`. Any of the arguments `Precedence`, `Type` or `Name` may be uninstantiated variables.
Example

```
current_op(_,_,(<-),Ops)
```

succeeds with

```
Ops = [op(1075,fx,(<-)),op(1075,xfx,(<-))]
```

`ticks(Time^)`

Synchronization No suspension.
Behaviour Unifies `Time` with an integer representing the current time. A *tick* is a unit of about 1/4660 seconds.

`halt`

Synchronization No suspension.
Behaviour Exits from Parlog for Windows.

Index of primitives

< /2, 65
<= /2, 67
= /2, 66
=.. /2, 69
=< /2, 66
== /2, 67
=:= /2, 65
=\= /2, 65
> /2, 66
>= /2, 66
\= /2, 66
arg/3, 69
atom/1, 68
atomic/1, 68
call/1, 71
call/3, 72
cat/2, 70
close/1, 79
compile/0, 82
create/1, 79
crwind/5, 79
current_op/4, 86
cuwind/1, 79
data/1, 68
debug/0, 83
debug_options/3, 83
defined/1, 81
depth/1, 85
depth/2, 85
dialog/4, 79
display/1, 77
display/2, 77
fail/0, 71
fastcode/0, 82
float/1, 68
free/6, 85
functor/3, 69
get/1, 75
get/2, 75
get0/1, 74
get0/2, 74
getkey/1, 74
gread/2, 75
ground/1, 68
halt/0, 86
in_stream/1, 76
in_streams/1, 76
incwrite/2, 78
integer/1, 68
is/2, 65
key/1, 75
kill/1, 81
listing/0, 80
listing/1, 80
load/1, 80
name/2, 70
nl/0, 77
nl/1, 77
nodebug/0, 83
nonvar/1, 68
nospy/1, 83
nospyall/0, 83
not/1, 71
notrace/0, 83
nowindow_debug/0, 84
number/1, 68
op/3, 86
open/1, 78
optimize/1, 85
optimize/2, 85
put/1, 77
put/2, 78
read/1, 74
read/2, 74
recall/2, 86
reinitialize/0, 81
remember/2, 86
same/2, 67
save/1, 81
save/2, 81
set/3, 72
skip/1, 75
skip/2, 75
spy/1, 83
subset/3, 73
tab/1, 78
tab/2, 78
ticks/1, 86
title/1, 79
toground/3, 70
tohollow/3, 71
trace/0, 82
true/0, 71
var/1, 67
varsin/2, 70
window_debug/0, 84
windows/5, 84
write/1, 76
write/2, 77
writeq/1, 77
writeq/2, 77