The Parallel Parlog User Manual V1.5.9

Parlog Group, Department of Computing, Imperial College, London
9th October 1989
Revised 14th June 1990, 28 September 1993

**by Jim Crammond, Andrew Davison, Alastair Burt,
Matthew Huntbach, Melissa Lam
Yannis Cosmadopoulos and Damian Chu**

1 User Manual

1.1 Introduction

The first part of this document is a user manual in tutorial form, and may be read while trying out the examples at a terminal. The second part is a reference manual, and contains a definition of Parallel Parlog and documentation on system predicates. The reader is assumed to have some knowledge of Parlog and Unix. In the rest of document, we will use the notation `pred/n` to denote a predicate with name `pred` and number of arguments (also known as the *arity*) `n`. The notation '`<CR>`' indicates that the 'return' key should be typed.

1.2 About Parallel Parlog

Parallel Parlog is an implementation of Parlog for shared memory multiprocessor machines. Versions currently exist for the Sequent Balance and Symmetry, BBN Butterfly, Encore Multimax and Alliant FX/8. Uniprocessor versions are also available for Sun3 and Sun4 workstations. The syntax is standard Parlog and is compatible with the SPM [Foster et al. 1986] and MacParlog [Conlon and Gregory 1989]

The language is as defined in [Gregory 1987], except for the *non-flat guard restriction* (see Section 1.9 [GuardSafety], page 10), the restricted form of the metacall (see Section 2.9.6 [Metalevel], page 25), and the introduction of a simple module scheme (see Section 1.8 [Modules], page 8).

This manual describes the third release of the Parallel Parlog system, V1.5, which contains a number of improvements over previous releases V1.0 and V1.4. Any program compiled on V1.0 will need to be recompiled in order to be executed on the new release of the system.

The main feature of release V1.4 of the Parallel Parlog system is a new memory management system that allows the different data areas within the emulator [Crammond 1990] to dynamically expand (and contract) as required. On machines that allow shared memory to be expanded during execution this makes the new Parlog system somewhat more robust, particularly when using many processors, and also more economical when executing small programs, particularly on a single processor. The latest release V1.5 adds many useful primitives and fixes many of the bugs reported by users (see Section 2.16 [Changes], page 46).

1.3 Entering and Exiting the Parallel Parlog System

The system can be entered by typing the command '`parlog`'. This command has one option '`-nx`' which specifies that the system should execute on '`x`' processors. On startup, the Parlog system displays the following message on the screen:

```
Parallel Parlog V1.5  -  10 Jan 1993
Copyright (C) 1993, Imperial College

| ?-
```

To exit Parlog, type the end of file character (usually '`CTL-D`') or the command '`halt.`' followed by `<CR>`. Note that the full-stop must be typed at the end of each command.

1.4 Running Queries

In response to the '`| ?-`' prompt the user may enter any conjunction of Parlog calls terminated by a full-stop. This conjunction is evaluated as a query, and if it is successful the bindings for the query are printed out and the word '`yes`' appears. Bindings are not printed for void variables (denoted by an underscore). If the query fails then this is indicated with '`no`'. This is illustrated by the following simple example:

```
| ?- X is 1 + 2.
 X = 3

yes
| ?- _ is 1 + 2.

yes
| ?- X is 1 + 2, X is 2 + 2.

no
| ?- halt.
ncalls/utime = 461 / 0.02 = 23050.0 RPS
```

Normally a query is executed in the foreground which means that the prompt for the next query will not appear until the last query has run to completion (either succeeded, failed or deadlocked). In some cases, it is desirable to run background queries so that more than one query can be executed in parallel. This feature is particularly useful for queries which are expected to suspend or take a long time to finish as it allows the user to enter new queries without waiting for the result. To execute a query in the background, prefix the query with a '`&&`'. Note that the whole query must

be run in the background. It is not possible to mix foreground and background processing within a single query.

```
| ?- create_channel(ch1,X) & data(X) & X=[H|_] & name(H,List).
data(_1511)
[deadlock]

| ?- && create_channel(ch2,X) & data(X) & X=[H|_] & name(H,List).

| ?- write_channel(ch2,hello).

yes
| ?-  X = [hello|_2081]
 H = hello
 List = [104,101,108,108,111]

yes
```

The evaluation of a query can be interrupted by typing the interrupt character (usually '`CTL-C`'). This should produce the prompt '`Parlog interruption ?`'. By typing '`y<CR>`' to this prompt, the execution of the query will be aborted; typing '`n<CR>`' will cause the execution to resume.

Note: if the interrupt prompt does not appear on the screen within a few seconds of typing an interrupt then the system has "hung" in some way. The only rescue from this situation is to type the quit character (usually '`CTL-\`').

1.5 Loading and Compiling

Users may define their own Parallel Parlog predicates by placing them in a text file. The correct syntax for predicates is defined in the Reference Manual (see Section 2.1 [Syntax], page 13). To make them available within the system, a Parallel Parlog object code file should be generated by using the `compile/1` predicate, and this should then be loaded with the `load/1` predicate.

It is usual for the Parlog text file to have a '`.par`' extension and the object file an '`.o`' extension, as in the following example, but this is optional (see Section 2.9.9 [CompilationAndLoading], page 31).

```
tutorial% cat tst.par

mode biased_merge(?,?,^).
biased_merge([A|As],Bs,[A|Cs])<-
    biased_merge(As,Bs,Cs).
```

```
biased_merge(As,[B|Bs],[B|Cs])<-
    biased_merge(As,Bs,Cs).
biased_merge([],Bs,Bs).
biased_merge(As,[],As).

tutorial% parlog
Parallel Parlog V1.5  -  10 Jan 1993
Copyright (C) 1993, Imperial College

| ?- biased_merge([1,2],[3,4],X).
[Warning: the procedure biased_merge/3 is undefined]

no
| ?- compile(tst) & load(tst).
[loading compiler]
Compiling file tst.par
biased_merge / 3
File tst compiled.

yes
| ?- biased_merge([1,2],[3,4],X).
 X = [1,2,3,4]

yes
| ?- halt.

ncalls/utime = 5432 / 0.56 = 9700.0 RPS
tutorial% ls
tst.o                tst.par
tutorial%
```

Terms in the source file with the form '`<-ConjunctionOfGoals`' are treated as commands, which are executed automatically as the object file is loaded.

1.6 Tracing

The Parallel Parlog system supports rudimentary tracing facilities, through the use of spypoints. A spypoint may be placed on a predicate using the `spy/1` predicate. Thereafter, each time the predicate is called, the goal and the current state of its arguments is displayed on the screen preceded by a '`Call:`' message. The user is then prompted, and four kinds of response are possible.

1. When the user types '`<CR>`', the traced call is executed and a further message is displayed on termination of the call. This message is either '`Exit:`' or '`Fail:`', followed by the goal and its arguments.

2. Typing '`n<CR>`' will cause the spypoint to be removed before executing the call. Thereafter, further calls to the predicate will not be traced (but existing calls will still be traced when they terminate). Spypoints can also be removed with the `nospy/1` predicate.
3. Typing '`a<CR>`' will abort the call.
4. Typing '`c goal<CR>`' will cause `goal` to be executed. This can be useful for setting further spypoints before proceeding with this call. Once `goal` has completed, the user is prompted again for action on this call.

```
| ?- spy(biased_merge/3).

yes
| ?- biased_merge([1,2], [], [1,3]).
Call: biased_merge([1,2],[],[1,3]) ?
Call: biased_merge([2],[],[3]) ?
Fail: biased_merge([2],[],[3]) ?
Fail: biased_merge([1,2],[],[1,3]) ?

no
| ?- biased_merge([1,2], [], X).
Call: biased_merge([1,2],[],_2454) ?
Call: biased_merge([2],[],_2546) ?
Call: biased_merge([],[],_2594) ?
Exit: biased_merge([],[],[]) ?
Exit: biased_merge([2],[],[2]) ?
Exit: biased_merge([1,2],[],[1,2]) ?
 X = [1,2]

yes
| ?- biased_merge([1,2], [], X).
Call: biased_merge([1,2],[],_2911) ? a

| ?- biased_merge([1,2], [], X).
Call: biased_merge([1,2],[],_3102) ? n
 X = [1,2]

yes
| ?-
```

Notes:

1. The emulator will continue to execute runnable goals while waiting for the prompt to a traced call; thus potentially altering the scheduling behaviour, particularly on one processor.

   A consequence of this is that parallel invocations of a spied predicate result in multiple '`Call:`' messages appearing, each requiring a response. This is rather unsightly at times.

2. A common fault that causes programs that work on one processor to fail on multiple processors is the incorrect use of `var(X)` in a guard. Such a guard may succeed 'unexpectedly' because it runs in parallel with a goal that eventually binds X. Remember, `var(X)` is time dependent and must be used with care in parallel programs.

1.7 Error Messages

There are four sorts of error messages which the user may encounter :

1.7.1 Builtin Errors

When the system predicates are given incorrect input they generate messages of the form :

```
[Error:  message: builtin name and arguments]
```

1.7.2 Syntax Errors

The `read` predicates (see Section 2.9.7.3 [TermIO], page 29) return error messages when they are used with input that does not conform to Parlog syntax. Since the compiler and the top level query evaluator use these predicates, the user will find the same messages displayed here. The format is:

```
**Syntax Error: offending term
```

The offending term is listed, and the place where the error occurred is marked by the word '`<<here>>`'.

1.7.3 Compiler Errors

The compiler may generate the following messages :

`**Syntax Error:` *Term*
: see above.

`**Mode Declaration Error:` *Mode*
: The mode is wrongly formatted; usually a '`^`' or '`?`' is missing.

`**Compiler Error - Name/Arity mismatch:` *Name/Arity*
: The clauses grouped together as one procedure with the sequential search operator '`;`' do not share the same name and arity.

`**Compiler Error:`*Name/Arity*
: Any other compiler error.

`warning: is/2 before "&" in body cannot be optimised`
: This indicates that a call to `is/2` can potentially be optimised by replacing the sequential conjunction operator '`&`' by the parallel conjuction operator '`.`'.

`warning: parallel deep guards detected`
: The compiler has detected two or more parallel non-flat clauses(see Section 1.9 [Guard-Safety], page 10). This warning may be ignored if the programmer is sure that only one of the clauses will commit. If this cannot be guaranteed, then the clauses should be separated using '`&`'.

`warning: some clause(s) have been ignored`
: The compiler has detected some redundant clauses. For example, in the relation

```
mode p(?).
p(X) <- X > 0 : positive(X).
p(X) <- X < 0 : negative(X).
p(X) <- X == 0 , zero(X).
```

: the third clause can always commit since it has no guard. The first two clauses are thus ignored. This is probably not what the programmer intended so the compiler gives the warning. The fix is to change the parallel clause separator '`.`' at the end of clause 2 to a sequential one '`;`', or to put a commit operator '`:`' between the two body goals in clause 3.

After processing the source file, a summary of all of the error messages is displayed. This is illustrated by the example session given below, which captures a syntax error in the first predicate of the '`kkqueen.par`' file.

```
tutorial% cat kkqueen.par

mode gen(?, ^).
gen(N, [N|Xs]) <-
  N > 0 :
  N1 is N-1,
  gen(N1, Xs).
gen(0, []).

mode count(?, ?, ^).
```

```
count([], M M).
count([X|Xs], M, Total) <-
  M1 is M + 1,
  count(Xs, M1, Total).

tutorial% parlog
Parallel Parlog V1.5  -  10 Jan 1993
Copyright (C) 1993, Imperial College

| ?- compile(kkqueen).
[loading compiler]
Compiling file kkqueen.par
gen / 2

** Syntax Error: count([],M <<here>> M).
count / 3
File kkqueen partially compiled:
One or more syntax errors in or before:
count / 3
These clauses missing from the object code!

yes
| ?- halt.

ncalls/utime = 4978 / 0.51 = 9760.78 RPS
tutorial%
```

1.7.4 Deadlock

When the evaluation of a query cannot terminate because there are calls suspended on input matching constraints, the system prints out all the suspended calls and then the message :

```
[deadlock]
```

Note: goals which were suspended before a previous deadlock, or were executing before an interrupt was typed, may be displayed among the list of goals suspended during this deadlock.

1.8 Modules

Parallel Parlog supports a simple module system to control the name space of predicates. This is based on the notion of a '`current module`', which is the place where a query currently looks for

predicate definitions. The predicate `current_module/1` returns the name of the current module, and `module/1` changes it (see Section 2.9.9 [CompilationAndLoading], page 31).

When predicates are loaded into the system, they are placed in the current module. Thus a command such as '`<- module exmpl.`' at the top of the source file will cause the predicates in the object file to be loaded into module `exmpl`. However, after the load is finished, the current module becomes the module which was being used prior to the load.

By default, predicates loaded into a module are private to that module and can only be called by goals within that module. Predicates can be made visible to all other modules by the `public/1` command, e.g. '`public a/1, b/2, c/3.`', which means that it can be invoked from any module. A public predicate is read-only in all modules other than the one in which it is defined, so the definition of a public predicate not in the current module cannot be replaced in subsequent loads by an alternative (private) one.

Predicates can also be imported into modules without making them public to all modules. This is a two stage process: firstly the predicates are made exportable in the module in which they are defined with `export/1`, and secondly they are imported with `import/2`, from the module in which they are defined into the current module. For example, suppose the current module is set to `foo`, then '`export([a/1,b/2,c/3])`' makes predicates `a/1`, `b/2` and `c/3` all exportable. Now in module `bar`, the predicate `b/2` can be imported from `foo` using '`import(foo,[b/2]).`'.

There are two predefined modules: `system` and `user`. All Parlog systems predicates, as defined in the reference manual, are defined in module `system` and are made public. In this way, the module mechanism is transparent to the user who does not wish to use it. The query evaluator executes by default in module `user` so that private predicates used by the system and compiler are invisible to the user.

The binding of a call to a predicate in the appropriate module is done at load time, thus incurring no overhead at run time. However, the binding for interpreted calls, using `call/1`, is done at run time in the context of the current module. Thus, the predicate must either be in the current module or imported into it. A predicate within a given module can be called explicitly using '`module#goal`' syntax, e.g.

```
| ?- system#assemble(file).
```

1.9 Guard Safety Considerations

The usual *guard safety* restrictions apply, which mean that bindings of variables in a guard are not made visible to the body of the clause until the guard has committed.

In addition, Parallel Prolog has a *non-flat guard restriction* which means that OR-parallel search only works for clauses with flat guards. Flat guards are guards which use only system predicates that can be compiled in-line and include all unification primitives (except `ground/1` and `\==/2`), all type tests, `is/2`, term comparison and arithmetic comparison. A guard is non-flat if it contains user defined predicates or non-flat system predicates. Non-flat guards are also called deep guards. More details on flat and non-flat guards can be found in [Gregory 1987].

The consequence of this non-flat guard restriction is that the programmer should ensure that all clauses containing deep guards (except the last) should end with a sequential clause separator `';'`. This coding strategy will ensure that a candidate non-flat clause will not be tried in parallel with other candidates.

With this restriction, the if-then-else form of deep guard is still possible (as seen in examples 1 and 2), but the or-parallel search technique - two alternative guards executing in parallel - is not (example 3).

```
(1) mode p1(?,^).
    p1(X,S) <- try(X) : S=ok ;
    p1(X,S) <- S=failed.

(2) mode p2(?,?).
    p2([A|B],Y) <- g1(A), g2(A) : b1(B,Y);
    p2(X,[A|B]) <- g1(A), g2(A) : b2(B,X);
    p2(X,Y)     <- b3(X,Y).

(3) % this is not allowed:
    mode p3(?,?,?,^).
    p3(X,L,R,T) <- find(X,L,T1) : T=T1.
    p3(X,L,R,T) <- find(X,R,T2) : T=T2.
```

A warning message is issued by the compiler when parallel deep guards are detected, such as in example 3.

There are transformations that automatically convert general Parlog into the restricted form, and future releases may perform these during compilation. At present, the user is responsible for any alterations to his code. Example 3 could be rewritten using `call/3` as :

```
p3(X,L,R,T) <-
    call(find(X,L,T1),Res1,Con1),
    call(find(X,R,T2),Res2,Con2),
    wait_for_result(Res1,Res2,Con1,Con2,T1,T2,T).

mode wait_for_result(?,?,^,^,?,?,^).
wait_for_result(success,_,_,stop,T1,_,T) <- T=T1.
wait_for_result(_,success,stop,_,_,T2,T) <- T=T2.
```

There is another programming restriction which is the result of the compiler trying to optimize code by compiling flat system predicates to in-line code. A side-effect of this is that the ordering of calls in the guard can sometimes be very important. e.g.

```
(4) longer(L1,L2) <- length(L1,A), length(L2,B), A>B : ...
(5) longer(L1,L2) <- length(L1,A), length(L2,B) & A>B : ...
(6) p <- X==Y, X=1, Y=2 : ...
(7) p <- X=1, Y=2, X==Y : ...
```

Examples 4 and 6 will deadlock under Parallel Parlog. A workaround is to either force a sequential evaluation (example 5) or re-order the calls (example 7). Note that since flat system predicates are compiled in-line, the separators between the system predicates in example 7 are notionally converted to '`&`' by the compiler.

2 Reference Manual

2.1 Syntax

A Parlog program consists of a finite set of *guarded clauses* and *mode declarations*. A guarded clause has the form:

```
H <- G1, ..., Gm : B1, ..., Bn  (m,n >= 0)
```

where `H` is called the *clause head*, `G1`, ..., `Gm` is its *guard* and `B1`, ..., `Bn` its *body*. The `:` is a *commit operator*. If the guard is empty (i.e. `m` = 0) then the commit operator can be omitted.

The clause head `H` and each of the `G1`, ..., `G` and `B1`, ..., `Bn` are *goals*. A goal has the form:

```
R(T1, ..., Tk)       (k >= 0)
```

where `R` is the *relation name* and `T1`, ..., `Tk` are its *argument terms*. The number of arguments, `k` is called the goal's *arity*. If the goal has no arguments (i.e. `k` = 0) then the brackets are omitted. A *relation* or *procedure* `R/k` comprises all clauses with the relation name `R` and arity `k`.

A *mode declaration* is associated with a Parlog procedure. For a relation `R/k` this has the form:

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

where each `m` is either `?` or `^`. A `?` denotes an *input* mode and a `^` denotes an *output* mode. If a mode declaration is omitted for a relation `R` then all of its argument terms are assumed to have an input mode. A mode `m` can be prefixed with argument names; these are comments of no semantic significance.

A *term* can be a variable, constant or structured term. A *variable* is denoted by an identifier beginning with a capital letter or the underscore character `_`. Examples are `X` and `Tree`. The underscore character may also be used on its own to denote a unique *anonymous* or *singleton* variable. A *constant* is either a number or an *atom* (a name denoted by a character string), which must be enclosed in single quotes when it can be confused with other types of term; examples are `jim`, `'Hello World'` and `'$primitive'`. Non-printable character may be included in quoted atoms by using *escape characters* (see Section 2.2 [EscapeSequences], page 15).

Numbers may be integers or floating-point numbers. Numbers may also be represented using radix notation where the radix is given by a single digit, followed by the quote character (') and the number in that base. Some examples will help to clarify this.

```
4'100 (base 4) = 16 (base 10)
8'123 (base 8) = 83 (base 10)
```

Radix 0 has a special meaning. It performs character to ASCII code conversion. e.g.

```
0'A = 65 (decimal) 'A' is ASCII 65
0'( = 40 (decimal) '(' is ASCII 40
```

A *structured term* is denoted by:

```
F(T1, ..., Tn)      (n >= 1)
```

where `F` is the *functor name* and the *arguments* `T1`, ..., `Tn` are terms. An example of a structured term is `point(X,Y,Z)`, whose functor name is `point` of arity 3, with arguments `X`, `Y` and `Z`. Infix notation may be used for certain structured terms where the functor is defined as an operator; thus the term `=(X,Y)` can be written as `X=Y`.

A special syntax is used for the type of structured term known as *lists*. The notation `[H|T]` is used to denote the list with the head `H` and tail `T`. A nested list term `[X|[Y|Z]]` may be abbreviated as `[X,Y|Z]` and if `Z` is the empty list or `nil` term (denoted by constant `[]`) then this can be abbreviated to `[X,Y]`.

A shorthand for representing a list of ASCII codes is to write the ASCII characters within double-quotes ("). Thus the string

```
"ABC"
```

is the same as the list [65,66,67].

A Parlog program is invoked by executing a *query*. A query is a conjunction of goals:

```
?- Q1, ..., Qn      (n >= 1)
```

Goals in a clause or query may be separated by the *parallel conjunction* operator, `','`, or the *sequential conjunction* operator, `'&'`. Similarly, clauses in a relation may be separated by Parlog's

parallel and *sequential clause search* operators, denoted as `'.'` and `';'` respectively. In addition, the last clause of a relation is always terminated by `'.'`.

Programs may include comments, which are defined as any text following a `%` character up to the next newline character.

2.2 Escape Sequences

In quoted atoms, double-quote delimited lists or base 0 notation, escape sequences can be used to denote characters. This is used to avoid ambiguity and to enter unusual characters. The following escape sequences are defined :

`\a` alarm/bell (ASCII 7)

`\b` backspace (ASCII 8)

`\t` tab (ASCII 9)

`\n` newline (ASCII 10)

`\v` vertical tab (ASCII 11)

`\f` formfeed (ASCII 12)

`\r` carriage return (ASCII 13)

`\e` escape (ASCII 27)

`\s` space (ASCII 32)

`\d` delete (ASCII 127)

`\<octal string>`
: the character with ASCII code `<octal string>` base 8. e.g. `\007` is the bell character and `\040` is the space character (ASCII 32).

`\^<control char>`
: the character whose ASCII code is the `<control char>` mod 32. e.g. `\^P` is `CTL-P`.

`\<CR>` no character. This is most useful when splitting atoms over two lines where the ignored newline character is preceded by a backslash.

`\c` no character, also all characters up to, but not including, the next non-layout character are ignored. Layout characters are characters with ASCII code =< 32 or >= 127.

`\<other>` the character `<other>`, where `<other>` is any character not defined above.

2.3 Semantics

Parlog differs from Prolog in three important respects : concurrent evaluation, don't-care non-determinism, and its (optional) use of mode declarations to specify communication constraints on shared variables.

During the evaluation of a call '`r(a1,...,ak)`', all of the clauses for the predicate `r` can be searched in parallel for a candidate clause. The clause :

```
r(t1, . . ., tk)  <-  < guard >  :  < body >
```

is a candidate clause if the head '`r(t1,...,tk)`' matches the call '`r(a1,..., ak)`' and the guard succeeds. It is a non-candidate if the match or the guard fails. If all clauses are non-candidates the call fails, otherwise one of the candidate clauses is selected and the call is reduced to the substitution instance of the body of that clause.

There is no backtracking on the choice of candidate clause since we "don't care" which candidate clause is selected. In practice, the language implementation dictates that the first one (chronologically) to be found is chosen. The absence of backtracking in Parlog is compensated for by guard tests which can be used to guarantee that the correct clause is chosen. This idea has been more formally stated in Gregory's *sufficient guards law* [Gregory 1987], which identifies the situations under which a Parlog computation returns a solution to a query, if a solution exists. It states that Parlog predicates should be written so that guards in each clause guarantee that if a clause is selected to reduce a goal then either a solution to the call can be computed using this clause, or no solution can be found using any other clause.

The search for a candidate clause can be controlled by using either the parallel clause search operator '`.`' or the sequential clause search operator '`;`' between clauses. For instance, if a predicate is defined by the clauses :

```
Clause1  .
Clause2  ;
Clause3.
```

Then Clause3 will not be tried for candidacy until both Clause1 and Clause2 have been found to be non-candidate clauses.

Non-variable terms that appear in input argument positions in the head of a clause can only be used for input matching. If an argument of the call is not sufficiently instantiated for an input match

to succeed, the attempt to use the clause suspends until some other process further instantiates the input argument of the call. If all clauses for a call are suspended, the call suspends but a candidate clause can be selected even if there are other suspended clauses.

Parlog utilises unification for output bindings which means that bindings to output variables will be unified with any values in those argument positions in the goal.

2.4 Command Line Options

`-n x` Specifies the number of processors to execute. The default value is '`-n1`'.

`-m x` [BBN Butterfly and Encore Multimax only] Specifies the number of memory segments (i.e. 64Kbyte blocks) to allocate on startup. The default value is `7 + 5n`, where '`n`' is the number of processors.

`-C x` [BBN Butterfly only] Specifies the number of memory segments (i.e. 64Kbyte blocks) to allocate for the private code area. The default value is 6.

On machines that allow shared memory to be expanded during execution there should be no reason to use the '`-m`' option as the system can allocate extra memory if needed automatically. Currently the Alliant, Sequent Balance and Symmetry and the uniprocessor versions of the system support this memory expansion feature. On the BBN Butterfly and the Encore Multimax, you may need to re-run your program using the '`-m`' option when you get the error message

```
can't extend shared memory, use -m option to get more
```

2.5 Environment Variables

The Parallel Parlog system uses three environment variables - `PARLOGDIR`, `PARLOG_PATH` and `PARLOG_INIT`.

`PARLOGDIR` is the top level directory of the Parlog run-time system. This is normally set automatically by the script which starts the Parallel Parlog system. You will only need to set this variable explicitly if you wish to use an alternative run-time system located in another directory.

`PARLOG_PATH` is a colon-separated list of directories in the style of the `PATH` environment variable. This is used to give a default list of directories to search for when looking for files. Normally, this

environment variable is not set, and only files in the current directory will be found. To access files in other directories, a relative or full pathname must be given. If you set `PARLOG_PATH`, then you will be able to access files in those directories by specifying the filename only (see Section 2.7 [FilenamesAndPaths], page 18).

`PARLOG_INIT` is the name of a file containing commands to be executed at startup time (see Section 2.6 [StartupFile], page 18).

2.6 Startup File Processing

When Parallel Parlog is started, commands in a startup file are executed. The commands should be of the form :

```
<- ConjunctionOfGoals.
```

For example, the following startup file will display a welcome message and load a customised library file. Note that lines not beginning with a `'<-'` are ignored.

```
<- nl &
   write('Welcome to Parlog') & nl.

<- load(mylib).

load(otherlibs).  % this line is not executed.
```

The default startup file is '`~/.parlog`' but this may be altered by setting the environment variable `PARLOG_INIT`. If the file specified by `PARLOG_INIT` does not exist, the system behaves as if the file was empty.

2.7 Filenames and Paths

There are a few primitives which take a filename as an argument. The most commonly used are `open/3`, `load/1` and `compile/1` for opening a file, loading compiled code and compiling Parlog source respectively. Filenames are atoms, so they are usually quoted since they commonly contain non-alphanumeric characters such as `'/'`, `'.'` and `'~'`. In accordance with Unix conventions, if the filename begins with a `'~'` character, the `'~'` is substituted by the home directory of the current user. If the filename begins with `'~<user>'` then the home directory of `<user>` is substituted

instead. ‘.’ signifies the current directory and ‘..’ denotes the parent directory of the path name so far.

By default, filenames are searched for in the current directory only. This may become tedious since files not in the current directory will need to be specified by giving the full path name. By setting the environment variable `PARLOG_PATH` (see Section 2.5 [Environment], page 17), a list of directories may be supplied for filename searching. This list is a colon-separated list of path names. e.g. if `PARLOG_PATH` has the value

```
.:~/parlog:/usr/local/parlog/source
```

then filenames will be searched for in those three directories in the order specified. Note that if ‘.’ is not included in `PARLOG_PATH`, then files in the current directory will not be visible.

The value of `PARLOG_PATH` may be changed dynamically by the `setenv/2` primitive.

2.8 System Operator Declarations

The following list specifies the operators defined in the system operator set.

```
op(system, 1400, xfy, ';')  &
op(system, 1350, xfy, '..') &
op(system, 1300, xfy, '\\') &
op(system, 1250,  fx, (&&)) &
op(system, 1200, xfx, (<-)) &
op(system, 1200,  fx, (<-)) &
op(system, 1150,  fx, [(mode),(public),(prolog)]) &
op(system, 1150, xfx, :)    &
op(system, 1100, xfy, &)    &
op(system, 1050, xfy, and)  &
op(system, 1000, xfy, ',')  &
op(system,  700, xfx, [is,=,=..,\=,==,\==,<=,=@=,\=@=,@<,@>,
                       @=<,@>=,=:=,=\=,<,>,=<,>=])   &
op(system,  600, xf, [?,^])   &
op(system,  500, yfx, [+,-,/\,\/,\]) &
op(system,  500,  fx, [+,-,\])  &
op(system,  400, yfx, [*,/,//,<<,>>]) &
op(system,  300, xfx, [div,mod])      &
op(system,  200, xfx, #).
```

2.9 System Predicates

This section describes the behaviour of the predicates which are already built into the system. The predicates are grouped under categories for easy reference. For each predicate, the conditions for synchronisation are also detailed.

2.9.1 Unification Related

`Term1 = Term2`

synchronisation: No suspension.

behaviour: Output unification. Attempts to unify Term1 with Term2. Succeeds if Term1 and Term2 unify; else fails.

`Term1 == Term2`

synchronisation: Suspends if Term1 and Term2 can be unified only by binding variables in both of them.

behaviour: Test unification. It unifies Term1 and Term2 without binding variables in either term. Succeeds if Term1 and Term2 are identical terms; fails as soon as Term1 and Term2 fail to match.

`Term1 \== Term2`

synchronisation: As for `==/2` above.

behaviour: Tests Term1 and Term2 without binding variables in either. Fails if Term1 and Term2 are identical terms; succeeds as soon as Term1 and Term2 fail to match. Note that there is a subtle difference with `\=@=/2` as shown by the following examples.

```
g(X) \== g(Y)
```

suspends since X and Y may potentially be different, but

```
g(X) \=@= g(Y)
```

succeeds since the variable X is different from the variable Y.

`Term1 <= Term2`

synchronisation: Suspends if Term1 and Term2 could only be unified by binding variables in Term2

behaviour: Input matching. Unifies Term1 and Term2 without binding variables in Term2. Succeeds if Term2 is a substitution instance of Term1. Fails as soon as Term1 and Term2 fail to match.

`var(X)` *synchronisation:* No suspension.

behaviour: Succeeds if X is an unbound variable. Fails if it is instantiated.

`nonvar(X)`

synchronisation: No suspension.

behaviour: Fails if X is an unbound variable. Succeeds if it is instantiated.

`ground(X)`

synchronisation: Suspends until X is ground.

behaviour: Succeeds when X is ground.

`data(X)` *synchronisation:* Suspends until X is instantiated.

behaviour: Succeeds when X is instantiated.

2.9.2 Type Tests

`atom(X)` *synchronisation:* Suspends until X is instantiated.

behaviour: Succeeds if X is an atom; otherwise fails.

`atomic(X)`

synchronisation: Suspends until X is instantiated.

behaviour: Succeeds if X is an atom or a number; otherwise fails.

`integer(X)`

synchronisation: Suspends until X is instantiated.

behaviour: Succeeds if X is an integer; otherwise fails.

`float(X)` *synchronisation:* Suspends until X is instantiated.

behaviour: Succeeds if X is a floating point number; otherwise fails.

`number(X)`

synchronisation: Suspends until X is instantiated.

behaviour: Succeeds if X is an integer or a floating point number; otherwise fails.

2.9.3 Arithmetic Expressions

`Value is Expression`

synchronisation: Suspends until Expression is ground.

behaviour: Expression is evaluated as an arithmetic expression, and the resulting number is unified with Value. If any of the variables are bound to non-numeric values, then the goal fails. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. The functors available in Parlog are given below. In the following, X and Y are considered to be arithmetic expressions.

`+X` Results in the positive value of X. The type of the result is the same as the type of the operand.

`-X` Results in the negative value of X. The type of the result is the same as the type of the operand.

`\X` Results in the complements of the bits in X.

`integer(X)`
Results in X if it is an integer. Otherwise, if it is a float, the result is the nearest integer that is between it and 0.

`float(X)` Results in X if it is a float. Otherwise, if it is an integer, the result is the floating point equivalent.

`X+Y` Results in the sum of X and Y. If both operands are integers, the result is an integer; otherwise the result is a float.

`X-Y` Results in the difference of X and Y. If both operands are integers, the result is an integer; otherwise the result is a float.

`X*Y` Results in the product of X and Y. If both operands are integers, the result is an integer; otherwise the result is a float.

`X/Y` Results in the quotient of X and Y. The resultant value is always a float, regardless of the types of the operands.

`X//Y` Results in the integer quotient of X and Y. The resultant value is always a (truncated) integer.

`X mod Y` Results in the remainder after the integer division of X by Y. The result always has the same sign as X.

`X/\Y` Results in the bitwise conjunction of X and Y.

`X\/Y` Results in the bitwise disjunction of X and Y.

`X<<Y` X is left-shifted Y places.

`X>>Y` X is right-shifted Y places with sign extension.

2.9.4 Term Comparison

Term comparison is based on a lexical ordering. The lexical ordering provides a way of ordering all terms using the following criteria :

```
variables @< numbers @< atoms @< structures
```

- variables are compared using their addresses
- numbers are compared numerically e.g. -3 `@<` 1 `@<` 1.5 `@<` 2
- atoms are compared using their ASCII codes
- structures are first ordered by arity, then functor, then 1st argument, 2nd argument etc.
- A list is considered as a structure with functor '.' and arity 2 e.g. f(1) `@<` [1] `@<` f(1,2)

`X=@=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically equal to Y; else fails.

`X\=@=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically not equal to Y; else fails. Note that the suspension only occurs at the top level; e.g.

```
X \=@= Y
```

suspends but

```
g(X) \=@= g(Y)
```

succeeds. This primitive can therefore be used as a non-suspending equality test by wrapping the two term to be compared inside a structure such as `g/1` .

`X@<Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically less than Y; else fails.

`X@>Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically more than Y; else fails.

`X@=<Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically less than or equal to Y; else fails.

`X@>=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: Succeeds if X is lexically more than or equal to Y; else fails.

`compare(Op,Term1,Term2)`

synchronisation: Suspends until Term1 and Term2 are instantiated.

behaviour: Unifies Op with one of '`=`', '`<`' or '`>`' depending on whether Term1 is lexically equal, less than or greater than Term2 respectively.

`X=:=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if they are equal.

`X=\=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if they are not equal.

`X<Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if X is strictly less than Y.

`X>Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if X is strictly greater than Y.

`X=<Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if X is less than or equal to Y.

`X>=Y` *synchronisation:* Suspends until X and Y are instantiated.

behaviour: X and Y must be numbers, and the goal succeeds if X is greater than or equal to Y.

2.9.5 Term Manipulation

`Term =.. List`

synchronisation: Suspends until either Term or List is instantiated.

behaviour: List is a list whose head is the atom corresponding to the principal functor of Term and whose tail is a list of the arguments of Term. If Term is uninstantiated, then List must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number.

`functor(Term,Functor,Arity)`

synchronisation: Suspends until Term is instantiated, or Functor and Arity are instantiated.

behaviour: If Term is instantiated, Functor and Arity will be bound to the functor and arity of Term. If Functor and Arity are instantiated, Term will be bound to the most general Term having this name and arity.

`arg(N,Term,Arg)`

synchronisation: Suspends until N and Term are instantiated.

behaviour: Initially, N must be instantiated to a positive integer and Term to a compound term. The result of the call is to unify Arg with the Nth argument of Term. (The arguments are numbered from 1 upwards). If the initial conditions are not satisfied or N is out of range, the call fails.

`atom_chars(Atom,String)`

synchronisation: Suspends until either Atom or String is instantiated.

behaviour: If Atom is instantiated (it must be an atom), then String is unified with the list of character codes in its name. If String is ground (it must be a list of character codes), Atom is unified with the atom containing those characters. Note that in Parallel Parlog `"add"`, for example, is shorthand for [97,100,100] (see Section 2.1 [Syntax], page 13).

`number_chars(Number,String)`

synchronisation: Suspends until either Number or String is instantiated.

behaviour: If Number is instantiated (it must be an integer or floating-point number), then String is unified with the list of character codes in its name. If String is ground (it must be a list of character codes), Number is unified with the number containing those characters.

`name(Atomic,String)`

synchronisation: Suspends until either Atomic or String is instantiated.

behaviour: If Atomic is instantiated (it must be an atom or a number), then String is unified with the list of character codes in its name. If String is ground (it must be a list of character codes), Atomic is unified with the number if possible, otherwise the atom containing those characters.

`concat_atom(List,Atom)`

synchronisation: Suspends until List is instantiated.

behaviour: List must be instantiated to a list of atomics (numbers or atoms). Atom is unified with the concatenation of the names of atomics in List.

`varoccurs(Term,VarList)`

synchronisation: No suspension.

behaviour: Unifies VarList with a list of variable occurrences in Term. A variable may appear in VarList more than once if there are multiple occurrences of it in Term.

`varsin(Term,VarList)`

synchronisation: No suspension.

behaviour: Unifies VarList with a list of variables that occur in Term. This list may be different from `varoccurs/2` because duplicates are removed. Each variable occurs in VarList only once regardless of how many occurrences there are in Term.

`copy_term(Term1,Term2)`

synchronisation: No suspension.

behaviour: A copy of Term1 with new variables is unified with Term2.

2.9.6 Metalevel

`call(Call)`

synchronisation: Suspends until Call is instantiated.

behaviour: If Call is instantiated to an atom or compound term, then the goal call(Call) is executed exactly as if that term appeared textually in its place. If Call fails, then call(Call) fails.

`call(Call,Status,Control)`

synchronisation: Suspends until Call is instantiated.

behaviour: This primitive always succeeds binding Status to `success` or `failed` depending on whether Call succeeded or not. If Control is instantiated to `stop`, execution of Call is terminated, binding Status to `stopped`.

Note: Due to the nature of the implementation, there is an uncorrectable bug such that when a process is '`stopped`' in this way, Call continues to execute until it succeeds or fails. Status will still be bound correctly to the value `stopped`. This bug is manifested most evidently when Call writes output.

`Module#Call`

synchronisation: Suspends until Module and Call are bound.

behaviour: Evaluates Call as it is defined in Module. Succeeds if Call succeeds. Fails if Call fails.

`fail` *synchronisation:* No suspension.

behaviour: Fails.

`not(Call)`

synchronisation: Suspends until Call is bound.

behaviour: This fails if the goal Call has a solution, and succeeds otherwise. This is not real negation, which is not possible in Parlog, but negation-by-failure meaning that P is not provable.

`halt` *synchronisation:* No suspension.

behaviour: Terminates the Parallel Parlog session.

`true` *synchronisation:* No suspension

behaviour: Always succeeds.

2.9.7 Input and Output

Input and output is dealt with by utilising `streams`. A stream can refer to a file or to a terminal, and must be either an input or an output stream. At any time, there is one *current input* stream and one *current output* stream. Input and output predicates fall into two categories:

1. Those that use the current input and output stream.
2. Those which take an explicit stream argument.

The standard streams are identified by `user_input`, `user_output`, or `user_error`. Other streams are identified by the term returned by `open/3` in its Stream argument (see below).

The current input and output streams are set to `user_input` and `user_output` upon entry to the Parallel Parlog system.

2.9.7.1 Streams

`open(File,Mode,Stream)`

synchronisation: Suspends until File and Mode are instantiated.

behaviour: If Mode is the atom `read`, and File is an existing file for which the user has read permission then it succeeds, unifying Stream with a term representing an input stream for File. If mode is the atom `write` and the user has permission to write to the file then it succeeds, unifying Stream with a term representing an output stream for File. If mode is the atom `append` and File is an existing file for which the user has write permission then it succeeds, unifying Stream with a term representing an output stream for appending to File. Otherwise the predicate fails.

`close(Stream)`

synchronisation: Suspends until Stream is instantiated.

behaviour: If Stream is an open stream then it succeeds and closes the stream; otherwise it fails. Failure will also occur if Stream is the stream for standard input or output.

`set_input(Stream)`

synchronisation: Suspends until Stream is instantiated.

behaviour: If Stream is an open stream then it succeeds, setting the current input to the stream. Otherwise it fails.

`set_output(Stream)`

synchronisation: Suspends until Stream is instantiated.

behaviour: If Stream is an open stream then it succeeds, setting the current output to the stream. Otherwise it fails.

`current_input(Stream)`

synchronisation: No suspension.

behaviour: This causes Stream to be unified with the current input stream.

`current_output(Stream)`

synchronisation: No suspension.

behaviour: This causes Stream to be unified with the current output stream.

`clear_input(Stream)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Flushes the input buffer associated with the given stream. For terminal input this means throwing away the rest of the current line. The current line being the sequence of characters input but not yet read off the stream.

`flush_output(Stream)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Flushes the output buffer associated with the given stream.

2.9.7.2 Character I/O

`get0(Stream,N)`
`get0(N)` *synchronisation:* Suspends until Stream is instantiated.

behaviour: Unifies N with the ASCII code of the next character from the input stream Stream. If end of file is encountered, N is unified with the atom `end_of_file`. If the end of stream was reached on a previous input command, then this call will give an error message and abort the computation. The one argument version reads from the current input stream.

`get(Stream,N)`
`get(N)` *synchronisation:* Suspends until Stream is instantiated.

behaviour: Unifies N with the ASCII code of the next non-space character from the input stream Stream. Space characters are all those with ASCII codes less than or equal to 32; this includes space, tab, linefeed and all control characters. If end of file is encountered, N is unified with the atom `end_of_file`. If the end of stream was reached on a previous input command, then this call will give an error message and abort the computation. `get/2` should not be used with `user_input`. The one argument version reads from the current input stream.

`skip(Stream,N)`
`skip(N)` *synchronisation:* Suspends until Stream and N are instantiated.

behaviour: Advances the input stream Stream to immediately past the character represented by the ASCII code N (N must be an integer). If end of file is encountered N is unified with the atom `end_of_file`. If N is not an integer, or is an integer outside the range 0..127, then the call fails after issuing an error. The one argument version reads from the current input stream.

`put(Stream,N)`
`put(N)` *synchronisation:* Suspends until Stream and N are instantiated.

behaviour: Writes the character represented by the ASCII code N to the stream represented by Stream. If N is not an integer, or is an integer outside the range 0..127, then the call fails after reporting an error. The one argument version writes to the current output stream.

`nl(Stream)`
`nl` *synchronisation:* Suspends until Stream is instantiated.

behaviour: Starts a new line on the output stream Stream. It is equivalent to `put(Stream,10)`. The zero argument version writes to the current output stream.

`tab(Stream,N)`
`tab(N)` *synchronisation:* Suspends until Stream and N are instantiated.

behaviour: Writes N spaces to the output stream Stream. N must be an integer; otherwise an error is issued and the call fails. The one argument version writes to the current output stream.

2.9.7.3 Term I/O

`read(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Reads the next Parlog term from the input stream Stream and unifies it with Term. The term must be terminated by a full-stop followed by a space character or a newline. These will be consumed by the call but not returned in Term. When a syntax error is encountered, an error message is printed which indicates where the error occurs in the input, and the call then fails. If the end of the file is encountered then the atom `end_of_file` is unified with Term. If the end of stream was reached on a previous input command, then this call will produce an error message and abort the computation.

`read(Stream,OpSet,Term,Variables)`

synchronisation: Suspends until Stream and Opset are bound.

behaviour: As for `read/2` except that the operator set given by OpSet is used to parse the term, and Variables is unified with a list of (Atom,Variable) pairs giving the atom representation of each variable in the term.

`display(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Writes the term Term to the output stream Stream in prefix notation, with no quotes for atoms that need quoting in order to be read back in.

`write(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Writes the term Term to the output stream Stream according to the operator precedences of the `system` operator set, with no quotes for atoms that need quoting in order to be read back in (see Section 2.8 [SystemOperatorDeclarations], page 19).

`writeq(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Writes in the same way as `write/2`, but with quotes around atoms that need quoting in order to be read back in.

`write_canonical(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Writes the term Term to the output stream Stream, with quotes around atoms that need quoting in order to be read back in.

`print(Stream,Term)`

synchronisation: Suspends until Stream is instantiated.

behaviour: Writes the term Term to the output stream Stream. If a user-defined `portray/2` is defined in the user module, `portray/2` is used to write the term. If it is not defined or if no clause matches, `print/2` behaves the same as `write/2`. The two arguments of `portray/2` are the stream and the term to be written.

The predicates below are equivalent to their counterparts above, except that they act upon the current output stream :

```
display(Term)          write(Term)
writeq(Term)           write_canonical(Term)
print(Term)
```

As expected, `read/2` has a short form for reading from the current input stream :

```
read(Term)
```

2.9.8 Channels

Channels are a way of naming stream variables. A channel is associated with a stream variable via an identifier. Terms which are written to the channel successively instantiate the tail of the stream variable. This channel mechanism can therefore be used as an efficient way of implementing `merge/3` since many processes can write to the same channel identifier.

Another use of channels is as a means for process communication without sharing variables explicitly. One process may instantiate a stream variable, which an independent process (i.e. does not share any variables) can read by getting the channel via a pre-defined channel identifier.

Since multiple processes may write to the same channel, and multiple processes may get the channel, this provides a simple mechanism for one-to-many, many-to-one and many-to-many process communication.

Channel identifiers are either numbers generated by the system or atoms specified by the user. User-defined channel identifiers provide a way of accessing stream variables created by other processes.

`create_channel(ID, Var)`

synchronisation: No suspension.

behaviour: Create a new channel and assigns a name ID to a stream variable Var. Var must be an uninstantiated variable. If ID is a variable, it will be bound to a system generated channel identifier (a number). If ID is an atom, that atom will be used as the channel identifer. If ID is neither a variable nor an atom, the predicate fails.

`write_channel(Ch, Term)`

synchronisation: Suspends until Ch is instantiated.

behaviour: The term Term is written to the channel Ch, or more precisely the tail of the stream variable associated with the channel is instantiated to `[Term|_]`. Ch is either a system generated channel identifier or a user-defined one. The channel must have been previously opened using `create_channel/2` and it must not be closed yet, otherwise it fails.

`get_channel(Ch, Var)`

synchronisation: Suspends until Ch is instantiated.

behaviour: Var is unified with the stream variable associated with the channel. This primitive enables the programmer to access the complete history of the stream variable. The primitive fails if the channel is not valid.

`close_channel(Ch)`

synchronisation: Suspends until Ch is instantiated.

behaviour: The channel Ch is closed. The tail of the stream variable associated with the channel is instantiated to `[]`. The primitive fails if the channel is not valid.

2.9.9 Compilation and Loading

`compile(File)`

synchronisation: Suspends until File is instantiated.

behaviour: Compiles a Parlog source code file. Initially, the name File is sought as the input file; if that does not exist then File is tried with '`.par`' appended to it. If still no match is found, an error is issued and the call succeeds. If the compiler is not already loaded, then a call to `compile/1` will automatically load it. The filename is searched in a list of default directories (see Section 2.7 [FilenamesAndPaths], page 18).

`load(File)`

synchronisation: Suspends until File is instantiated.

behaviour: Loads a Parlog object code file. Initially, the name File is sought; if that does not exist then File with '`.o`' appended is tried. If still no match is found, an error is issued and the call fails. The filename is searched in a list of default directories (see Section 2.7 [FilenamesAndPaths], page 18).

`load_foreign(Predicates,Obj,Libraries)`

synchronisation: Suspends until all three arguments are instantiated.

behaviour: Loads C predicates into the system. See see Section 2.10 [ForeignRoutines], page 37.

`load_foreign(Predicates,Obj)`

synchronisation: Suspends until both arguments are instantiated.

behaviour: As for `load_foreign/3` but with no libraries specified.

`load_foreign_io(Predicates,Obj,Libraries)`

synchronisation: Suspends until all three arguments are instantiated.

behaviour: Loads C predicates into the system that can only be executed by the master processor. This makes a difference only on multiprocessor implementations where Input/Output is handled by one master processor to preserve ordering. On uniprocessors (implementations on Suns), this primitive is the same as `load_foreign/3` (see Section 2.10 [ForeignRoutines], page 37).

`load_foreign_io(Predicates,Obj)`

synchronisation: Suspends until both arguments are instantiated.

behaviour: As for `load_foreign_io/3` but with no libraries specified.

`defined(Pred)`

synchronisation: Suspends until Pred is instantiated.

behaviour: Succeeds if a definition for the predicate Pred exists, fails otherwise. Predicates may be system defined or loaded by the user. A predicate may be specified using the `Name/Arity` notation or a sample call.

```
| ?- defined(append/3).

yes
| ?- defined(append(_,_,_)).

yes
| ?- defined(append(_,_)).

no
| ?-
```

`public Preds`

synchronisation: Suspends until Preds is ground.

behaviour: Preds is expected to be a sequence of Name/Arity pairs that represent predicate names, otherwise the call fails. If a call to `public/1` succeeds, these predicates become visible to all other modules.

`export(Predicates)`

synchronisation: Suspends until Predicates is ground.

behaviour: The predicates list is expected to consist of Name/Arity pairs that represent predicate names, otherwise the call fails. This enables the predicates listed (which should be private to this module) to be imported into other modules with `import/2`.

`import(Module,Predicates)`

synchronisation: Suspends until Module and Predicates are ground.

behaviour: The Predicates list is expected to consist of Name/Arity pairs that represent predicate names, otherwise the call fails. This imports the definitions of Predicates from Module into the current module and can be used by code subsequently loaded into the current module.

`module(Module)`

synchronisation: Suspends until Module is instantiated.

behaviour: Makes Module the current module.

`current_module(Module)`

synchronisation: No suspension.

behaviour: Unifies Module with the current module.

2.9.10 Operators Related

`op(OpSet,Precedence,Type,Op)`

synchronisation: Suspends until all four arguments are instantiated.

behaviour: Inserts atoms into the operator set OpSet with the specified precedence and type. Precedence should be an integer between 1 and 1200. The lower the precedence, the tighter binding the operator. Type should be one of fx, fy, xf, yf, xfx, xfy or yfx, and Op should be an atom or a list of atoms. The types are defined as follows :

`fx`	non associative prefix operator
`fy`	associative prefix operator
`xf`	non associative postfix operator
`yf`	associative postfix operator
`xfx`	non associative infix operator

`xfy` right-associative infix operator

`yfx` left-associative infix operator

Note that an operator may be declared to be prefix, postfix and infix all at the same time, though this is confusing and not recommended.

The precedence and type of an operator may be changed by re-declaring it. To remove an operator, re-declare it with precedence 0 and the relevant non associative type (i.e. fx, xf or xfx).

`current_op(OpSet,Precedence,Type,Op)`

synchronisation: Suspends until OpSet,Type and Op are instantiated.

behaviour: Returns the precedence of the Op operator declared in the OpSet operator set. Type should be one of fx, fy, yf, xfx, xfy or yfx, Op should be an atom, and OpSet should be an operator set. If no match is found in OpSet, but an operator is found in the system operator set, then the precedence of the system operator is returned.

The following two system predicates are equivalent to their counterparts above, except that they act upon the operator set system :

`op(Prec,Type,Op)`	`current_op(Prec,Type,Op)`

2.9.11 Spypoints

`spy(Pred)`

synchronisation: Suspends until Pred is instantiated.

behaviour: Pred must be a Name/Arity pair representing a predicate that has already been loaded. If this is the case, `spy/1` succeeds, setting spypoints on the named predicate; otherwise it fails.

`nospy(Pred)`

synchronisation: Suspends until Pred is instantiated.

behaviour: Pred must be a Name/Arity pair representing a predicate that has already been loaded. If this is the case, `nospy/1` succeeds, removing spypoints from the named predicate; otherwise it fails.

2.9.12 Miscellaneous

`util(Package)`

synchronisation: Suspends until Package is bound.

behaviour: Loads predicates from the specified package in the utilities directory; examples are the foreign interface package (see Section 2.10 [ForeignRoutines], page 37), the tracer (see Section 2.11 [AlternativeTracer], page 39), the Parlog Shell (see Section 2.12 [ParlogShell], page 42) and the SPM compatibility package (see Section 2.13 [SPMCompatibility], page 42).

`statistics(Key,Val)`

synchronisation: Suspends until Key is bound.

behaviour: When Key is one of the atoms `suspensions`, `gcs`, `heap`, `stack`, `process`, `program`, `memory`, `runtime` or `calls` the appropriate values will be unified with Val and the predicate will succeed; otherwise it will fail. For memory values (when Key has the value `heap`, `stack`, `process`, `program` or `memory`), Val is bound to a term of the form `[memory_used, memory_free]`. These values are given in 'word' (4 byte) units except for `process`, which is given in 'process' (32 byte) units, and `memory` which returns the total size of the allocated and free memory segments in bytes. When Key is `runtime`, Val is bound to a list of the form `[total, Ilast]`, where `total` is the user cpu time since the Parallel Parlog system was invoked, and `last` is the time since the last call to `statistics/2` or `statistics/0`.

`statistics`

synchronisation: No suspension.

behaviour: Prints out all the values generated by `statistics/2` onto the current output stream.

`unix(Command)`

synchronisation: Suspends until Command is instantiated.

behaviour: Can be used in the following ways:

```
unix(cd)                % change working directory to home
unix(cd(Path))          % change working directory to Path
unix(shell)             % invoke an interactive shell
unix(system(Cmd))       % pass Cmd to sh(1)
unix(access(File))      % test whether File is readable
```

`setenv(Env,Val)`

synchronisation: Suspends until Env and Val are instantiated.

behaviour: Set the environment variable Env to the value Val. Env must be instantiated to an atom and Val must be either an atom or a number.

`getenv(Env,Val)`

synchronisation: Suspends until Env is instantiated.

behaviour: Val is unified with the value retrieved from the environment variable Env. The value is either an atom or a number. Note that Val is bound to a number if the

value can be converted to an integer or a floating point number. This has the side effect that atom values whose name resemble numbers (e.g. `'123'`) cannot be retrieved successfully if stored in an environment variable.

`clock(T)` *synchronisation:* No suspension.

behaviour: T is unified with the current time represented as the number of seconds since 00:00:00 1st January, 1970.

`absolute_file_name(F,Abs)`

synchronisation: Suspends until F is instantiated.

behaviour: Abs is unified with the absolute path name of the file F. The filename F is searched in the directories specified in `PARLOG_PATH` (see Section 2.5 [Environment], page 17). Meta-characters in F are recognised (see Section 2.7 [FilenamesAndPaths], page 18). The primitive fails if F is not found.

`stat(F,Abs,ModTime,Size,Mode,Uid,Gid)`

synchronisation: Suspends until F is instantiated.

behaviour: This primitive retrieves information about the file F. The filename F is searched in the directories specified in `PARLOG_PATH` (see Section 2.5 [Environment], page 17). Meta-characters in F are recognised (see Section 2.7 [FilenamesAndPaths], page 18). The primitive fails if F is not found.

If F is found, Abs is unified with the absolute path name, ModTime is the time of last modification (see `clock/1`), Size is the file size, Mode is the file permissions, Uid and Gid are unified with the user and group ID of the file respectively.

The file permissions are represented by a single integer in which each bit indicates whether the specified permission is set. The permissions and the corresponding bit values are as follows :

```
octal  decimal  permission
=====  =======  ==========
 001        1   Execute (search) by others.
 002        2   Write by others.
 004        4   Read by others.
 010        8   Execute (search) by group.
 020       16   Write by group.
 040       32   Read by group.
 100       64   Execute (search in directory) by owner.
 200      128   Write by owner.
 400      256   Read by owner.
1000      512   Sticky bit.
2000     1024   Set group ID on execution.
4000     2058   Set user ID on execution.
```

`append(List1,List2,List3)`

synchronisation: Suspends until List1 is instantiated.

behaviour: List3 is unified with the result of appending List2 to the end of List1.

`merge(List1,List2,List3)`

synchronisation: Suspends until either List1 or List2 is instantiated.

behaviour: List3 is unified with the result of merging the two lists List1 and List2. This primitive implements a fair merge in the sense that if elements are available on both List1 and List2, then elements in List3 are taken alternately from List1 and List2.

2.10 Foreign Routines

The Parallel Parlog system allows C predicates to be loaded dynamically into the system either as predicates to be executed on any processor (using `load_foreign/3 ),` or predicates only executable by the master processor (using `load_foreign_io/3 ).`

To aid integration of C routines into the Parlog environment, a *foreign* package is provided, for generating the interface routines that convert Parlog data types into C data types and invokes the C routines. The interface routines are generated from a file of `foreign/2` declarations, which have a similar format to those used by Quintus Prolog.

2.10.1 Foreign Declarations

The syntax of the foreign declarations is as follows :

```
foreign( Function, PredicateSpecification ).

PredicateSpecification = PredicateName(ArgSpec, ArgSpec,...)

ArgSpec =  Arg?          input argument
        |  Arg^          output argument
        |  [Arg^]        argument for the function return value

Arg = short | integer | float | string | atom
```

In this context, string refers to a Parlog atom which is converted to a C string; whereas atom refers to the internal representation of a Parlog atom (a tagged Word) which is passed directly to the C routine as a long unsigned integer.

The following conversions are carried out according to the ArgSpec type :

```
Parlog: short?        -->       C: integer
Parlog: integer?      -->       C: integer
Parlog: float?        -->       C: double
Parlog: string?       -->       C: char *
Parlog: atom?         -->       C: Word

Parlog: short^        <--       C: integer *
Parlog: integer^      <--       C: integer *
Parlog: float^        <--       C: double *
Parlog: string^       <--       C: char **
Parlog: atom^         <--       C: Word *

Parlog: [short^]      <--       C: return (integer)
Parlog: [integer^]    <--       C: return (integer)
Parlog: [float^]      <--       C: return (double)
Parlog: [string^]     <--       C: return (char *)
Parlog: [atom^]       <--       C: return (Word)
```

Some examples of foreign declaration are :

```
foreign( sin, sin(float?, [float^]) ).
foreign( rename, rename(string?, string?) ).
foreign( test, tester(integer?, [integer^], atom?, atom^) ).
```

2.10.2 Adding Foreign Routines

The procedure for intergrating a C routine into the Parallel Parlog environment is as follows:

1. Consider a C routine `strcat()` which takes two strings as its arguments and returns the concatenated string. It is defined in '`strcat.c`' and is compiled to an object file in the usual way :
   ```
   cc -O -c strcat.c
   ```
2. A foreign declaration for this C routine is then placed in a Parlog file, called '`iface.par`' :
   ```
   foreign(strcat, strcat(string?, string?, [string^])).
   ```
   This states that the Parlog predicate `strcat/3` will call the C routine `strcat()` and return its result as a Parlog atom in its 3rd argument.
3. Now the interface file for this declaration can be generated by loading the foreign package and calling `make_interface/1` in the following way :
   ```
   | ?- util(foreign).
   [loading foreign]

   yes
   | ?- make_interface(iface).
   ```

```
yes
```

This produces a C file called '`iface.c`' which must be then be compiled to an object file :

```
cc -O -c iface.c
```

4. The interface and C routine object files can now be loaded in to the Parlog system with the `load_foreign/3` command :

```
| ?- load_foreign([strcat/3], 'iface.o', 'strcat.o').
```

`strcat/3` is the name of the Parlog interface predicate, '`iface.o`' is the object file for the C interface and '`strcat.o`' is the object file for the original C routine. Now the `strcat/3` predicate can be invoked :

```
| ?- strcat(foo, bar, X).
X = foobar

yes
```

In practise, an interface file can specify the interfaces to C routines in several files. In that situation, the 3rd argument of `load_foreign/3` or `load_foreign_io/3` will contain several files and/or libraries, e.g.

```
| ?- load_foreign_io([a/1, b/1, ..], 'iface.o', 'win1.o win2.o -lXlib').
```

2.11 An Alternative Tracer

The *trace* utility provides a more comprehensive tracing facility which allows the user to execute a Parlog program with set break points and trace points. At a trace point the user is informed of some action which took place during execution. At a break point execution is halted and the user is able to investigate the current state of the system.

The basis behind this tracer is that each clause in a Parlog program may be considered as a rewrite rule. Any point in the execution of a query by the program may be represented by a list of goals. This list starts as the initial query, and the rewrite rules are continually applied to it until either it is empty, in which case the query succeeds, or no rewrite rule may be applied, in which case the query fails. The tracer also halts at deadlock, when no goal is sufficiently instantiated to allow a commitment to a rewrite to take place.

2.11.1 Loading and Using the Tracer

The tracer is loaded with the command: `util(trace)`. The program to be traced should be in one or more files named '`<name>.par`'. To enter the tracer type :

```
partrace([<name1>,<name2>, ..., <namen>]).
```

where `[<name1>,<name2>, ..., <namen]` is a list of the filenames which contain the program to be traced (without the '`.par`' suffix). The brackets may be omitted if there is only one file. The program will then be parsed by the tracer after which the tracer will query the user for the sort of tracing required.

The system will first prompt with:

```
Full trace?
```

The user should reply '`y`' for yes or '`n`' for no. In full trace mode every time a goal commits to a clause the reduction made is printed out, a break is made in computation, and the user is prompted for a trace command.

If the user chooses not to use full trace mode the system will first ask for a list of predicates to be traced on reduction. These should be entered one to a line and followed by an empty line. If there is any ambiguity over arity a predicate name may be followed by a slash and its arity. For example :

```
pred1
pred2/2
pred2/3
pred3
```

If the user just types '`<CR>`' following the prompt no predicate will be traced. At any point in the computation when a goal for one of the predicates listed for tracing commits to a clause the reduction will be printed in the form:

```
<goal> => <body of new clause with variable bindings passed through>
```

If output modes are used, any unifications giving output are also printed in the form:

```
<variable> <= <term>
```

Following the prompt for tracing, the user is prompted for a list of predicates to break on reduction. The list is entered in the same way as above. When a goal for a predicate in this second list commits, the reduction will be printed as before, but execution will also halt, and the user will be prompted for a trace command.

Whether in full trace mode or not, the user will also be asked for a list of predicates to be traced on completion. This is a form of tracing based on the procedural view of Parlog. It prints out the final form of a goal when it has completed execution. A goal has completed execution either if it reduces to the empty list or if it has reduced to a list of subgoals all of which have completed execution. A completion is indicated by the trace:

```
* Completed * <goal>
```

2.11.2 Trace Commands

At a break point, the user is asked for a command with the prompt:

```
Trace command :
```

A '<CR>' typed at this point will display the state of computation (see below), otherwise one of the following commands are can be typed:

`c`	continue execution until next trace point
`f`	give full display, including all goals following a sequential '`&`'
`g`	give standard display and allow user to investigate goals
`h`	give list of available commands
`q`	quit execution
`t`	reset trace information
`u`	undo trace points and continue execution until completion

2.11.3 Displaying State of Computation

At any point in the computation goals in the list of current goals will be of three types:

1. Those on which no attempt has been made to reduce, but which could be chosen for reduction.

2. Those on which an attempt to reduce has been made, but which have not yet committed, possibly due to a suspension while awaiting a variable instantiation.
3. Those which cannot be reduced as they follow a sequential '`&`', and the goal before the '`&`' has not completed execution.

The standard display only prints goals of the first two sorts, numbering each goal. The full display prints every goal and includes the bracketing to indicate the mix of sequential and parallel execution.

Following the 'investigate goals' command, '`g`', the list of goals is displayed and the user may choose to investigate it in more detail. The system gives the following prompt:

```
Enter goal number (0 to restart tracing) :
```

If an integer greater than 0 is entered, the system will give further details on the goal with that number. If no attempt has been made to reduce the goal, it will just be printed. Otherwise the current state of reduction is given. The current state of the guard computation, and any input variables on which a possible commitment is suspended are given, followed by the subgoals to which the goal would reduce if commitment were made. In the case of a sequential clause search, clauses which would be tried if the current commitment cannot be made are displayed.

If '`0`' is entered instead of a goal number, the system returns to the computation. Alternatively, the list of the goals can be redisplayed by entering '`l`', and can be displayed in full form by entering '`f`'.

2.12 Parlog Shell

To be written.

2.13 SPM Compatibility

SPM [Foster et al. 1986] and Parallel Parlog are syntactically very similar, and the system predicates in both languages offer much the same functionality. However, many SPM builtins are not present in Parallel Parlog, but this obstacle can be overcome by loading the compatibility package with the command '`util(spm)`' It contains predicate definitions for emulating certain SPM

predicates from six broad categories : I/O, unification, arithmetic, type testing, term manipulation and utilities. The predicates defined in this package are :

```
I/O.
        fcreate/2       fclose/1        fopen/3
        loutput/1       loutputnl/1     p/1
        pnl/1           print_term/1    print-term/2
        read_char/2     write_char/2    load_files/1

Unification.
        equal/2         unify/2         assign/2

Arithmetic.
        add/3           div/3           eq/2
        less/2          lesseq/2        mod/3
        mul/3           sub/3

Type testing.
        con/1           list/1          tuple/1

Term manipulation.
        a_to_s/2                        s_to_a/2
        list_to_struct/3                struct_to_list/3
        l_to_i/3                        i_to_l/2
        sappend/2                       predefined_op/1

Utilities.
        sleep/1
```

The SPM file manipulation primitives `access/1`, `assem/1` and `edit/3` are not supported.

In the area of I/O, the `get_term/2` and `put_term/3` primitives for reading terms to and from a disk are not implemented in Parallel Parlog, and `intok/3` and `iowait/2` are also not available.

The `set/3` and `subset/3` all-solution primitives are absent from Parallel Parlog, but all of the type testing primitives are present (albeit after '`util(spm)`' is loaded). However, care must be taken when transferring user-defined predicates of this sort across from the SPM, since SPM does not treat [] as an atom while Parallel Parlog does.

Parallel Prolog has neither a `freeze/2` nor a `melt/2` primitive, but does offer `copy_term/2` for which they are most often used.

Unavailable control primitives include `controlc/0`, `key/1` and `space/3`.

Many SPM utilities are absent from Parallel Parlog, including `date/3`, `show_date/0`, `time/3`, `instream/1`, `load_comp/0`, `sedit/0`, `sinput/2` and `username/1`. In addition, the utilities for type checking, manipulating binary trees and the *xref* program are not supported. Finally, Parallel Parlog does not support any window predicates, although an X based window interface is planned for the future.

Parallel Parlog differs from SPM in having more predicates for testing types, a set of lexical comparison primitives, a module system, a spypoint mechanism, a C and UNIX interface, and statistic predicates. In addition, Parallel Parlog enables extra operator sets to be defined, and for commands to be included in source files.

Some SPM programs may need to be rewritten to take into account the nonflat guard restriction of Parallel Parlog. However, transformation techniques exist to do this and these may be put into future releases.

2.14 MacParlog Compatibility

MacParlog [Conlon and Gregory 1989] is very similar to Parallel Parlog except in the area of I/O, where it contains numerous primitives for manipulating the Macintosh user interface.

Some versions of MacParlog can use arithmetic functions from MacProlog [Clark et al. 1987], for such things as transcendental and exponential calculations.

In the area of metalevel predicates, MacParlog allows terms to be converted between ground and hollow versions using `toground/3` and `tohollow/3`, and `varsin/2` permits the variables in a term to be accessed. `primitives/1` and `private_macparlog_names/1` have no equivalents in Parallel Parlog.

I/O in MacParlog is based on *channels*, which are like Parallel Parlog *streams* except that channels can be connected to Macintosh windows. MacParlog input can utilise the mouse, and `key/0` is available for waiting on input. `gread/3` reads terms and returns the variables contained within them, but is less flexible than `read/4` in Parallel Parlog. `skip/1` and `skip/2` are slightly different in MacParlog since the 'N' argument can be a character as well as an integer.

File handling in MacParlog utilises the Macintosh hierarchical filing system rather than UNIX directories. However, the resulting predicates are still quite similar to those in Parallel Parlog. The main difference is the use of file dialogues in `create/2`, `new/3` and `old/2` to access a particular file on a volume.

MacParlog has a number of extra utility predicates not present in Parallel Parlog; namely `beep/1`, `recall/2`, `remember/2`, `ticks/1` and `time/3`. Also, the language permits deep guards to be tried in parallel, and so does not impose the nonflat guard restriction found in Parallel Parlog. MacParlog has a number of window handling primitives and dialogue primitives, none of which are available in Parallel Parlog.

Parallel Parlog has primitives for the lexical comparison of terms, for gathering statistics, and accessing C and UNIX, which are not found in MacParlog. Also, MacParlog does not support user-defined operator sets, there is no module system, and compiling and loading is done via pull down menus. In addition, MacParlog does not allow commands to be included in source files.

2.15 Changes Between V1.0 and V1.4

Most of the differences between the first and second releases of Parallel Parlog are internal and transparent to the user. This section outlines the areas where the user may notice the changes.

As already mentioned, the memory management is quite different; in V1.0 the sizes of the four main data areas – code, heap, argument stack and process stack – were determined at startup time, and controlled by `-C`, `-H`, `-S` and `-P` options. In V1.4 each area is made up of a linked list of segments or 64Kbyte blocks, which are allocated from a pool of segments. The initial number of segments can be controlled by the `-m` option, but on machines in which (shared) memory is expandable, the system will automatically allocate more when needed.

C predicates can now be executed on any processor if loaded with `load_foreign/3`, allowing Parlog to control the parallel execution of C routines. Predicates which must only run on the master processor (e.g. I/O predicates or perhaps window predicates) should be loaded using `load_foreign_io/3` instead.

There have been some changes to the compiler and the code it generates. One noticeable change is that calls to `is/2` in the body can now be executed 'inline', with a full goal call only being made if the inline call cannot proceed because input arguments are unbound. This optimisation cannot be done if the call to `is/2` precedes a sequential conjunction operator ('`&`') and is not in parallel with any other goal calls - the compiler provides a warning message about this.

The `public/1` predicate has been changed so that it makes the specified predicates visible in all modules. Before, only public predicates in the system module were visible to all modules; public predicates in other modules still had to be explicitly imported.

There were a few problems related to the use of floating point numbers in V1.0 which have been fixed in V1.4.

2.16 Changes Between V1.4 and V1.5

* "channel" abstraction for merging streams (see Section 2.9.8 [ChannelIO], page 30)
* recognition of escape characters within atoms (see Section 2.2 [EscapeSequences], page 15)
* primitives to access environment variables (see Section 2.9.12 [Miscellaneous], page 34)
* ability to define a search path for files (see Section 2.5 [Environment], page 17)
* recognition of meta-characters in filenames (see Section 2.7 [FilenamesAndPaths], page 18)
* background processing (see Section 1.4 [Running], page 2)
* startup file processing (see Section 2.6 [StartupFile], page 18)
* New primitives `concat_atom/2`, `number_chars/2`, `atom_chars/2`, `append/3`, `merge/3`, `stat/7`, `absolute_file_name/2` `defined/1`, `compare/3`, `clock/1`, `print/1`, `print/2`, `varoccurs/2`, `varsin/2`.
* Improved compiler messages/warnings
* Numerous bug fixes

2.17 Bugs/Features

Suspended guard calls for a candidate clause, which has been rejected in favour of another clause, are not removed by the system. Such suspended guard calls are also called *dead* processes.

Deadlock and interrupts do not kill or remove processes; they only restart the top level command interpreter.

I/O predicates produce a segmentation error when called with an inappropriate stream argument.

It is possible to overflow a heap by unifying a variable with a very large term in the source code (e.g. a list of over 128 elements) when the top heap segment is almost full. This can result in a

segmentation error. There is no problem with terms constructed dynamically during execution and a workaround is to break up the source code term into smaller parts, e.g.

```
g(X) <- X = [1,2,....,128|X1], g1(X1).
g1(X1) <- X1 = [129,130,....,256].
```

Similarly, it is possible to overflow an argument stack by calling a large number of goals in a parallel conjunction when the top stack segment is almost full, if the sum of all the arguments in the parallel conjunction exceeds 256. The workaround is to replace some of the goals in the conjunction with a single call to a predicate that invokes the replaced goals. In theory, the compiler could be made to detect and fix both of these cases automatically.

Loading C predicates does not work on the Alliant and Multimax versions.

3 Interface with IC-Prolog II

This chapter describes the extra primitives available when Parlog is run under the IC-Prolog II system. These primitives are not available in the standalone Parlog system.

In IC-Prolog II, a thread is a sequential Prolog process. IC-Prolog II is a multi-threaded system so there may be more than one thread running concurrently.

3.1 Calling Prolog

To enable Parlog to call Prolog, there is a suite of Parlog primitives which execute Prolog goals. This is achieved by executing the Prolog goals in separate threads in the multi-threaded IC-Prolog II system.

There is more than one such primitive because the goal may be executed in a new thread or an existing server thread, the goal may produce one or more solutions which can be generated eagerly or lazily, and we may or may not be interested in the answer bindings.

`prolog_cmd(G)`

synchronisation: Suspends until G is instantiated.

behaviour: A new thread is created to execute the Prolog goal G. G is a command so no results are expected back to Parlog i.e. any variables in G will not be bound.

`prolog(G)`

synchronisation: Suspends until G is instantiated.

behaviour: A new thread is created to execute the Prolog goal G. Variables in G will be bound to Prolog's first solution and the values communicated back to Parlog.

`prolog(G,P)`

synchronisation: Suspends until G is instantiated.

behaviour: A new thread is created to execute the Prolog goal G. All the solutions of G will be available by successively reading from the port P using `read_pipe/2`. When there are no more solutions, reading from port P will return `end_of_file`.

`prolog(G,P,Th)`

synchronisation: Suspends until G is instantiated.

behaviour: Same as `prolog/2` with the extra argument returning the thread ID of the newly created thread.

`lazy_prolog(G,P,Th)`

synchronisation: Suspends until G is instantiated.

behaviour: Same as `prolog/3` except that the solutions are generated lazily and it is the programmer's responsibility to get the next solution by explicitly resuming the suspended Prolog thread by calling `resume(Th)`.

`qprolog(G)`

synchronisation: Suspends until G is instantiated.

behaviour: Similar to `prolog/1` but quicker to execute since it uses the background Prolog server thread instead of creating a new thread. Note that G should be a simple query only since long queries will monopolise the Prolog server and cause a bottleneck.

`resume(Th)`

synchronisation: Suspends until Th is instantiated.

behaviour: Resume execution of the specified thread. This is used to wake up explicitly suspended Prolog threads. If Th is 0, the next runnable thread is resumed. `resume/1` returns an error if Th is not a currently valid thread.

`kill_thread(Th)`

synchronisation: Suspends until Th is instantiated.

behaviour: Kills the specified thread and reclaims the memory space.

3.2 Pipe Communication

In the description below, a pipe is a communication channel between two threads or processes. A pipe has two end which are called ports. An output port is where terms enter the pipe (i.e. written out by a thread), and an input port is where the terms leave the pipe (i.e. read in by a thread). The same mechanism can be used for Prolog to Prolog, Parlog to Parlog, Prolog to Parlog or Parlog to Prolog communication.

`pipe(OP,IP)`

synchronisation: No suspension.

behaviour: Creates a new pipe and binds OP to the output end and IP to the input end of the pipe. Terms which are written to OP may be read from IP. The structure of a port term is `port(N)` where N is an integer.

`write_pipe(OP,T)`

synchronisation: No suspension.

behaviour: Outputs the term T to the output port OP. OP must be an output port owned by the current thread, otherwise the primitive fails. Ownership of a port is

automatically set to the current thread if there is no current owner. Note that creation of a pipe does not set ownership for the ports. Thus a pipe may be created by one thread and used by another thread. If any thread is suspended on a read, `write_pipe/2` wakes it after T is written out to the pipe.

`look_pipe(IP,T)`

synchronisation: No suspension.

behaviour: Copies a term T from the input port IP. The term is not removed from the pipe. IP must be an input port owned by the current thread, otherwise the primitive fails. Ownership of a port is automatically set to the current thread if there is no current owner. The pipe is 'locked' before the term is read. This is to prevent multiple readers accessing the pipe simultaneously. The lock is not released by this primitive (see `commit_read/1`). `look_pipe/2` suspends if the port is already locked or if the pipe is empty. If the output port of the pipe is closed, `look_pipe/2` binds T to the constant `end_of_file`.

`commit_read(IP)`

synchronisation: Suspends until IP is instantiated.

behaviour: Removes the term previously read by `look_pipe/2` from the pipe and unlock the pipe. IP must be an input port owned by the current thread, otherwise the primitive fails. `commit_read/1` fails if the pipe is not locked (i.e. no previous `look_pipe/2`).

`read_pipe(IP,T)`

synchronisation: No suspension.

behaviour: Reads a term T from the input port IP. Semantically, this is equivalent to `look_pipe(IP,T)` followed sequentially by `commit_read(IP)`.

`empty_pipe(IP)`

synchronisation: Suspends until IP is instantiated.

behaviour: Succeeds if there is no data ready to be read from input port IP.

`is_iport(Port)`

synchronisation: Suspends until Port is instantiated.

behaviour: Succeeds if Port is an input port.

`is_oport(Port)`

synchronisation: Suspends until Port is instantiated.

behaviour: Succeeds if Port is an output port.

`unlock(Port)`

synchronisation: Suspends until Port is instantiated.

behaviour: Unlocks a pipe which has been locked by a previous `look_pipe/2`. The port must be owned by the current thread, otherwise the primitive fails. This is used

in the case where the term read is not the one expected. The pipe is unlocked so that another thread can re-read the same term.

`close_port(Port)`

synchronisation: Suspends until Port is instantiated.

behaviour: Closes the specified port. The port must be owned by the current thread, otherwise the primitive fails. `close_port/1` suspends if the pipe is locked and fails if the port has a suspended operation pending. If an output port is closed, subsequent (and suspended) reads from the input port will return `end_of_file`. Closing an input port does not affect the output port. If both ports are closed, space used for the pipe is reclaimed.

`release_port(Port)`

synchronisation: Suspends until Port is instantiated.

behaviour: Releases ownership of the port to allow other threads to use it. The port must be owned by the current thread, otherwise the primitive fails. `release_port/1` suspends if the pipe is locked.

3.3 Network Communication Primitives

Parlog under IC-Prolog II has a large set of primitives for communication over a network. The set consists of an interface to TCP/IP library calls and an alternative communication model based on the notion of mailboxes.

The Parlog primitives are identical to those available under the Prolog component of IC-Prolog II. Please refer to the Prolog manual for further details.

3.4 Miscellaneous

`prolog` *synchronisation:* No suspension.

behaviour: Switches the query evaluator from Parlog to Prolog in the case where Prolog and Parlog share the same window. Normally, Parlog and Prolog will each have their own windows. This is only used interactively as a top level goal.

`parlog(G)`

synchronisation: Suspends until G is instantiated.

behaviour: Start executing the goal G as a separate Parlog process. This is similar to meta-call `call/1` except that variables are not instantiated and `parlog/1` returns

immediately regardless of the how long it takes to execute G or indeed whether it succeeds or fails. This primitive is most useful for setting up 'background' goals.

`timeslice(N)`

synchronisation: Suspends until N is instantiated.

behaviour: If N is the integer 0, time-slicing is switched off. This has the effect of preventing runnable Prolog threads from executing, thus allowing Parlog exclusive use of the processor. If N is the integer 1, time-slicing is resumed. This primitive is used when atomicity is required or when critical code regions are entered.

`ensure_loaded(F)`

synchronisation: Suspends until F is instantiated.

behaviour: Loads the file F if not already loaded. This primitive can be safely used instead of `load/1` to ensure that the file is loaded once and once only.

4 References

Clark, K., McCabe, F.G., Johns, N., and Spenser, C. 1987., "LPA MacPROLOG Reference Manual", Logic Programming Associates Ltd, London.

Conlon, T., and Gregory, S. 1989., "Hands on MacParlog 1.0", Parallel Logic Programming Limited, London, January.

Crammond, J.A. 1990., "The Abstract Machine and Implementation of Parallel Parlog", Research Report, Dept. of Computing, Imperial College, London.

Foster, I., Gregory, S., Ringwood, G., and Satoh, K. 1986., "A Sequential Implementation of PARLOG", 3rd International Conference on Logic Programming, London, July.

Gregory, S. 1987., "Parallel Logic Programming in PARLOG : The Language and its Implementation", Addison-Wesley.

Predicate Index

#

#/2 26

=

=../2 24
=/2 20
=:=/2 23
==/2 20
=@=/2 23
=\=/2 23
=</2 24

@

@=</2 23
@>=/2 23
@>Y/2 23
@</2 23

>

>/2 24
>=/2 24

\

\==/2 20
\=@=/2 23

<

</2 23
<=/2 20

A

absolute_file_name/2 36
append/3 36
arg/3 24
atom/1 21
atom_chars/2 24
atomic/1 21

C

call/1 25
call/3 26
clear_input/1 27
clock/1 36
close/1 27
close_channel/1 31
close_port/1 52
commit_read/1 51
compare/3 23
compile/1 31
concat_atom/2 25
copy_term/2 25
create_channel/2 31
current_input/1 27
current_module/1 33
current_op/3 34
current_op/4 34
current_output/1 27

D

data/1 21
defined/1 32
display/1 30
display/2 29

E

empty_pipe/1 51
ensure_loaded/1 53
export/1 33

F

fail/0 26
float/1 21
flush_output/1 28
functor/3 24

G

get/1 28

get/2 28
get_channel/2 31
get0/1 28
get0/2 28
getenv/2 35
ground/1 21

H

halt/0 26

I

import/2 33
integer/1 21
is/2 21
is_iport/1 51
is_oport/1 51

K

kill_thread/1 50

L

lazy_prolog/3 50
load/1 32
load_foreign/2 32
load_foreign/3 32
load_foreign_io/2 32
load_foreign_io/3 32
look_pipe/2 51

M

merge/3 37
module/1 33

N

name/2 25
nl/0 28
nl/1 28
nonvar/1 21
nospy/1 34
not/1 26
number/1 21
number_chars/2 25

O

op/3 34
op/4 33
open/3 27

P

parlog/1 52
pipe/2 50
print/2 30
prolog/0 52
prolog/1 49
prolog/2 49
prolog/3 49
prolog_cmd/1 49
public/1 32
put/1 28
put/2 28

Q

qprolog/1 50

R

read/1 30
read/2 29
read/4 29
read_pipe/2 51
release_port/1 52
resume/1 50

S

set_input/1 27
set_output/1 27
setenv/2 35
skip/1 28
skip/2 28
spy/1 34
stat/7 36
statistics/0 35
statistics/2 35

T

tab/1 29
tab/2 29

timeslice/1 53
true/0 26

U

unix/1 35
unlock/1 51
util/1 35

V

var/1 20
varoccurs/2 25
varsin/2 25

W

write/1 30
write/2 29
write_canonical/1 30
write_canonical/2 30
write_channel/2 31
write_pipe/2 50
writeq/1 30
writeq/2 29

Short Contents

1 User Manual 1
2 Reference Manual 13
3 Interface with IC-Prolog II 49
4 References 55
Predicate Index 57

Table of Contents

1 User Manual 1

1.1 Introduction 1
1.2 About Parallel Parlog 1
1.3 Entering and Exiting the Parallel Parlog System 2
1.4 Running Queries 2
1.5 Loading and Compiling 3
1.6 Tracing 4
1.7 Error Messages 6
1.7.1 Builtin Errors 6
1.7.2 Syntax Errors 6
1.7.3 Compiler Errors 6
1.7.4 Deadlock 8
1.8 Modules 8
1.9 Guard Safety Considerations 10

2 Reference Manual 13

2.1 Syntax 13
2.2 Escape Sequences 15
2.3 Semantics 16
2.4 Command Line Options 17
2.5 Environment Variables 17
2.6 Startup File Processing 18
2.7 Filenames and Paths 18
2.8 System Operator Declarations 19
2.9 System Predicates 20
2.9.1 Unification Related 20
2.9.2 Type Tests 21
2.9.3 Arithmetic Expressions 21
2.9.4 Term Comparison 22
2.9.5 Term Manipulation 24
2.9.6 Metalevel 25
2.9.7 Input and Output 26
2.9.7.1 Streams 27
2.9.7.2 Character I/O 28
2.9.7.3 Term I/O 29
2.9.8 Channels 30
2.9.9 Compilation and Loading 31

2.9.10 Operators Related 33
2.9.11 Spypoints 34
2.9.12 Miscellaneous 34
2.10 Foreign Routines 37
2.10.1 Foreign Declarations 37
2.10.2 Adding Foreign Routines 38
2.11 An Alternative Tracer 39
2.11.1 Loading and Using the Tracer 40
2.11.2 Trace Commands 41
2.11.3 Displaying State of Computation 41
2.12 Parlog Shell 42
2.13 SPM Compatibility 42
2.14 MacParlog Compatibility 44
2.15 Changes Between V1.0 and V1.4 45
2.16 Changes Between V1.4 and V1.5 46
2.17 Bugs/Features 46

3 Interface with IC-Prolog II 49

3.1 Calling Prolog 49
3.2 Pipe Communication 50
3.3 Network Communication Primitives 52
3.4 Miscellaneous 52

4 References 55

Predicate Index 57