GETTING STARTED WITH PARLOG (DEC-10 PROLOG, ICOT version)

S. Gregory

October 1983

1. INTRODUCTION

The PARLOG system described in this document comprises a compiler and an interpreter which are combined in one interactive system. Both compiler and interpreter are written in DEC-10 PROLOG (which is compatible with MPROLOG).

The document begins with features which are independent of the implementation and becomes more specific. Section 2 is a general description of the PARLOG language. Section 3 is relevant only to the PROLOG implementation of PARLOG and describes how to use the PROLOG language from PARLOG. Sections 4 and 5 are specific to the PARLOG system written in DEC-10 PROLO (and MPROLOG): section 4 gives the concrete syntax of PARLOG programs and section 5 lists the PARLOG system commands. Section 6 tells how to access the PARLOG system running on the ICOT DEC-20.

The appendix lists some files containing useful primitive relations.

2. PARLOG LANGUAGE FEATURES

This section briefly describes the features of PARLOG which are supported by this implementation. It is not intended as an introduction to programming in PARLOG; for this purpose the definitive paper [ClG 83] should be consulted. The PARLOG paper also contains many example PARLOG programs.

In this section we use the "publication syntax" of PARLOG, which is also used in [ClG 83].

2.1 Syntax of a PARLOG program

A PARLOG program consists of a set of relation definitions (reln_defn), each of which is the keyword 'relation' followed by a relation declaration (reln_decl) followed by a relation body (reln_body), i.e.

```
reln_defn = 'relation' reln_decl reln_body
```

The relation body is constructed from clause groups (cl_grp) by the operators '.' and ';':

```
reln_body = cl_grp '.' reln_body |
            cl_grp ';' reln_body |
            cl_grp
```

The '.' operator specifies parallel search for a candidate clause, while ';' specifies searching in sequential left-right order.

A clause group is either a clause or a parenthesized relation body. This allows an arbitrarily nested combination of '.' and ';' in a relation body:

```
cl_grp = clause |
         '(' reln_body ')'
```

A clause consists of a head atom, a guard conjunction and a body conjunction. The guard or body conjunctions may be empty:

```
clause = head |
         head ':-' body |
         head ':-' guard '|' body

head = atom

guard = conj

body = conj
```

A conjunction (conj) is an arbitrarily nested structure of literal groups (lit_grp) using the operators ',' (parallel "and") and '&' (sequential "and"):

```
conj = lit_grp ',' conj |
       lit_grp '&' conj |
       lit_grp

lit_grp = literal |
          '(' conj ')'
```

A literal is a positive or negative atom or a "lazy call" (see later):

```
literal = atom |
          '~' atom |
```

```
'lazy' atom
```

An atom has the same form as a PROLOG atom. Variables are identifiers beginning with a lower case letter. Other identifiers, and quoted strings, can be used as constructor function names.

The relation declaration takes the form

```
r(m1,...,mn)
```

where r is a relation name and m1,...,mn give the mode of the relation definition. Each mi is either '?' (for an input argument) or '^' (for an output argument). Every clause in the corresponding relation definition must have the same number (n) of arguments.

Note that if a relation is to be used in more than one mode, the whole relation definition must be repeated with an appropriate relation (mode) declaration for each mode, e.g.

```
relation merge(?,?,^)
merge([u|x],y,[u|z]) :- merge(x,y,z) .
merge(x,[u|y],[u|z]) :- merge(x,y,z) .
merge([],[],[])

relation merge(^,^,?)
merge([u|x],y,[u|z]) :- merge(x,y,z) .
merge(x,[u|y],[u|z]) :- merge(x,y,z) .
merge([],[],[])
```

Although in this case the relation definitions are the same for each mode, they may in general be different.

2.2 PARLOG modes

In PARLOG programs, every relation call and definition has a particular mode associated with it. The mode of each call is determined at compile time by an analysis of the direction of communication through variables. Details of this mode analysis can be found in [ClG 84]. Here we will illustrate the mode analysis by means of examples.

Consider the definition of the 'seqplus' relation, where seqplus(x,y,z) is the relation "z is a list in which each element is the sum of the corresponding elements in lists x and y":

```
relation seqplus(?,?,^)
seqplus([u|x],[v|y],[w|z]) :- plus(u,v,w), seqplus(x,y,z) .
seqplus([],[],[])
```

In the first 'seqplus' clause, variables u, x, v and y occur in the input arguments and so must be input to the body calls. w and z occur in output arguments so they must be output from the body calls. From this analysis, the compiler can deduce the mode (?,?,^) for both the 'plus' and 'seqplus' calls in the body.

The 'fibonacci' relation can be defined in terms of 'seqplus':

```
relation fibonacci(^)
fibonacci([1|x]) :- seqplus([0,1|x],[1|x],x^)
```

Here the mode of the 'seqplus' call is (?,?,^) since the first and second arguments of the call are non-variable and the third argument is a '^' annotated variable. If the '^' were omitted, the leftmost occurrence of x in the body would be taken as the "generator" of x and the third argument of 'seqplus' would therefore be input.

In the following definition of the 'multiples' relation, all calls in the body have mode (?,?,^):

```
relation multiples(^)
multiples(x) :- timeslist(2,[1|x],r),
                timeslist(3,[1|x],s),
                timeslist(5,[1|x],t),
                amerge(r,s,rs), amerge(rs,t,x^)
```

2.3 Back communication

It is often necessary to override the direction of communication imposed by the modes to allow "back communication" or "incomplete messages" [Sha 83]. This is done by means of head annotations.

To indicate that an output argument of a relation contains a variable which will be instantiated by the consumer of the argument, we annotate the variable by a '?' suffix on its occurrence in the output argument. In the definition of the consumer relation, the corresponding variable must be annotated by a '^' suffix.

An interesting example of back communication is a 'readlist' relation which reads terms from the terminal in response to demands from a user process:

```
relation readlist(?)
readlist([u^|x]) :- read(u) & readlist(x) .
readlist([])
```

This relation has a single input argument which is incrementally instantiated to a list "skeleton" by a user process. Each time the skeleton is extended by one element, a term is read and bound to the variable u. This is why u must be annotated by '^' in the head.

The following is a program to read a sequence of terms from the terminal and print them:

```
: proc(x,y), readlist(x), printlist(y).

relation printlist(?)
printlist([u|x]) :- printnl(u) & printlist(x) .
printlist([])

relation readlist(?)
readlist([u^|x]) :- read(u) & readlist(x) .
readlist([])

relation proc(^,^)
proc([u?|x],y) :- proc1(u,x,y)

relation proc1(?,^,^)
proc1('end_of_file',[],[]) :
proc1(u,x,[u|y]) :- proc(x,y)
```

Notice that the definitions of 'readlist' and 'printlist' are almost identical except that only 'readlist' does "back communication". The sequential "and" is necessary in both definitions due to the side effects of the 'read' and 'printnl' calls. Here, printnl(u) prints term u followed by a new line.

In general, any subterm(s) of an input argument in the head of a clause can be annotated by a '^' and variables occurring in this subterm can be marked as input by a '?' annotation. For example, Kowalski's "admissible pairs" example [Kow 79] can be programmed as follows:

```
: y = [<1,v>|x], adm(y), printlist(y).

relation adm(?)
adm(x) :- double(x), triple(x)

relation double(?)
double([<u,v^>|x]) :- times(2,u,v) & double(x)

relation triple(?)
triple([<u,v>|[<u1,v1?>|x7]^]) :- times(3,v,u1) &
                                  triple([<u1,v1>|x])
```

In this program, both 'double' and 'triple' calls have the same argument as input, and they are both provided with an initial value [<1,v>|x] for this argument. The two calls then cooperate in producing the remainder of the data structure. The 'double' call waits for a list with a head tuple whose first argument is a number, then binds the second argument of the

tuple. The 'triple' call waits for a list with a head tuple whose second argument is a number and then further instantiates the list with a new tuple whose first argument is now known.

This program does not terminate. If we assume that the input to 'adm' will be a list of pairs of variables of a particular length, used in a query such as

```
: y = [<1,y1>,<x2,y2>,<x3,y3>], adm(y), printlist(y).
```

then 'double' and 'triple' can be defined as:

```
relation double(?)
double([<u,v^>|x]) :- times(2,u,v) & double(x) .
double([])

relation triple(?)
triple([<u,v>,<u1^,v1>|x]) :- times(3,v,u1) & triple([<u1,v1>|x]) .
triple([<u,v>])
```

2.4 Lazy relation calls

An extra feature in this implementation of PARLOG is the ability to delay the evaluation of a relation call until all other activity has ceased. If a call is prefixed by the operator 'lazy', the call will be added to a special "lazylist" instead of being evaluated. When the interpreter detects deadlock it first examines the lazylist and if this is non-empty, all processes on it are reactivated and the lazylist is emptied.

A major use of lazy relation calls is to implement a 'readlist' relation which reads terms from the terminal only as a last resort. This allows the user to observe the effect of one input term before typing the next. This version of 'readlist' can be defined as follows:

```
relation readlist(^)
readlist(x) :- lazy read(u) & readlist1(u,x)

relation readlist1(?,^)
readlist1('end_of_file',[]) ;
readlist1(u,[u|x]) :- readlist(x)
```

Note that this relation has mode readlist(^) in contrast to the definition given earlier. This means that both definitions can be used in the same program.

3. INTERFACE TO PROLOG

All logic programming systems are provided with a set of "built-in relations" (sometimes called "evaluable predicates"), i.e. relations which are programmed not in the language itself, but in the host language. In this section, which is specific to the PROLOG implementation of PARLOG, we describe how to provide PARLOG relations which are programmed in the host PROLOG system.

The interface to PROLOG is provided by a special relation 'prolog', together with a new head annotation '!'. Calls to the 'prolog' relation may have any number of arguments and their mode is ignored. The first argument of a 'prolog' call is the name of a PROLOG relation and subsequent arguments are arguments of that PROLOG relation call. When a 'prolog' call is encountered, the relevant PROLOG relation is called with the given arguments.

For example, the PARLOG 'read' relation can be defined as follows (we shall use DEC-10 PROLOG relation names in this section but the same principle applies in any PROLOG system):

```
relation read(^)
read(u) :- prolog('read',u) |
```

It should be pointed out that unbound PARLOG variables are not necessarily the same as unbound PROLOG variables; if there are consumers suspended waiting for a PARLOG variable to be instantiated, the variable will not be an unbound PROLOG variable. It will actually have a value that contains a pointer to a list of suspended processes. Therefore, if a 'prolog' call is to bind a variable, then that variable should not be accessible from another PARLOG call until PROLOG has bound the variable. This is why, in the above example, the 'prolog' call appears in the guard.

Many PROLOG relations require certain arguments to be given at the time of call. In PROLOG programs this is ensured by evaluating conjunctions sequentially in a certain order. Due to the parallel evaluation of PARLOG, we need a different mechanism to ensure that arguments are known before evaluating a 'prolog' call. For this purpose, we use the '!' annotation on variables in head arguments of PARLOG relations. If '!' occurs on a variable, the evaluation will suspend until that variable is instantiated (to any term).

Note that the '!' annotation is not normally necessary in user programs. It is only needed for programming up primitives in PROLOG.

The following definition provides the PARLOG 'plus' relation in all useful modes:

```
relation plus(?,?,^)
plus(a!,b!,c) :- prolog('is',c,a+b) |

relation plus(^,?,?)
plus(a,b!,c!) :- prolog('is',a,c-b) |
```

```
relation plus(?,^,?)
plus(a!,b,c!) :- prolog('is',b,c-a) |

relation plus(?,?,?)
plus(a!,b!,c!) :- prolog('is',c,a+b)
```

Notice that all input arguments are '!'-annotated to ensure that arguments are instantiated before evaluating the 'prolog' calls. Also all 'prolog' calls which do output are placed in guards. Other arithmetic relations required in PARLOG can be provided in a similar manner.

Another example is a simple definition of the 'printnl' relation used earlier which will print its argument term as soon as it is instantiated:

```
relation printnl(?)
printnl(u!) :- prolog('write',u) & prolog('nl')
```

This definition is inadequate for printing structured terms that are constructed incrementally. A completely general 'printnl' relation is defined below:

```
relation printnl(?)
printnl(t) :- print(t) & prolog('nl')

relation print(?)
print(t!) :- prolog(=..,t,t!) & printargs(t!)

relation printargs(?)
printargs([f]) :- prolog('write',f) .
printargs([f,a|as]) :- prolog('write',f) &
                       prolog('write','(') &
                       printas([a|as]) &
                       prolog('write',')')

relation printas(?)
printas([a]) :- print(a) .
printas([a|as]) :- print(a) &
                   prolog('write',',') &
                   printas(as)
```

4. CONCRETE SYNTAX FOR PARLOG PROGRAMS

In this section we describe the concrete syntax of programs as used in the DEC-10 PROLOG and MPROLOG implementations of PARLOG.

4.1 Concrete syntax

Each relation definition is represented by a clause for the PROLOG relation 'relation':

```
RELN_DEFN = relation(RELN_DECL,RELN_BODY).
```

The relation declaration is the same as in the publication syntax, i.e. a term of the form r(m1,...,mn) where each mi is '?' or '^'.

The relation body is a PROLOG term constructed using the operators 'par' (which replaces '.') and 'seq' (which replaces ';'). 'par' and 'seq' have equal priority.

```
RELN_BODY = CL_GRP 'par' RELN_BODY |
            CL_GRP 'seq' RELN_BODY |
            CL_GRP

CL_GRP = CLAUSE |
         '(' RELN_BODY ')'
```

In a clause we use the operators '::-' instead of ':-' and ':' for the guard bar '|':

```
CLAUSE = HEAD |
         HEAD '::-' BODY |
         HEAD '::-' GUARD ':' BODY
```

Note that if the body is empty it must be replaced by the constant 'true':

```
CLAUSE = HEAD '::-' GUARD ':' 'true'

HEAD = ATOM

GUARD = CONJ

BODY = CONJ
```

For conjunctions, '//' is used for parallel "and" and '&' is used for sequential "and". Both operators have the same priority:

```
CONJ = LIT_GRP '//' CONJ |
       LIT_GRP '&' CONJ |
       LIT_GRP

LIT_GRP = LITERAL |
          '(' CONJ ')'
```

```
LITERAL = ATOM |
          '~' ATOM |
          'lazy' ATOM
```

Atoms are the same as PROLOG atoms, and may be written using infix, prefix or postfix operators if these operators are suitably defined by the user. The operators already defined by the PARLOG system are listed below.

Within atoms, terms are the same as PROLOG terms except that variables are PROLOG constants preceded by the prefix operator '*'. Variables may be annotated by the postfix operators '?', '^' and '!', which have a higher priority (less tightly binding) than '*', in the same way as in the publication syntax.

4.2 Operators defined

```
op(1200,fx,[:]).
op(760,xfy,[par,seq]).
op(750,xfy,[::-,:,::]).
op(740,xfy,[//,&]).
op(730,fx,[~,lazy]).
op(500,fx,[parcomp,list,listp,delete]).
op(200,xf,[?,^,!]).
op(100,fx,[*]).
op(650,xfy,[.]).
```

4.3 Example programs

Some of the programs given in sections 2 and 3 are repeated here in concrete syntax.

```
relation(seqplus(?,?,^),
seqplus([*u | *x],[*v | *y],[*w | *z]) ::-
      plus(*u,*v,*w) // seqplus(*x,*y,*z)  par
seqplus([],[],[])  ).

relation(fibonacci(^),
fibonacci([1 | *x]) ::- seqplus([0,1 | *x],[1 | *x],*x^)  ).

relation(multiples(^),
multiples(*x) ::- timeslist(2,[1 | *x],*r) //
                  timeslist(3,[1 | *x],*s) //
                  timeslist(5,[1 | *x],*t) //
                  amerge(*r,*s,*rs) // amerge(*rs,*t,*x^)  ).
```

```
relation(read(^).
read(*u) ::- prolog(read,*u) : true  ).

relation(plus(?,?,^).
plus(*a!,*b!,*c) ::- prolog(is,*c,*a + *b)  ).
```

5. INTERACTING WITH THE PARLOG SYSTEM

This section, like the preceding one, is specific to the DEC-10 PROLOG and MPROLOG implementations of PARLOG.

Since a PARLOG program is represented in the concrete syntax by a PROLOG program for relation 'relation', a PARLOG program can be entered by the usual 'consult' mechanism and then manipulated by the PROLOG editor, if one is provided. In addition, certain commands are provided by the PARLOG system for manipulating and running programs and these are described below.

5.1 PARLOG system commands

list R.

Lists on the terminal the PARLOG code for all relations with relation specifier R. A relation specifier is a term which matches a relation declaration r(m1,...,mn).

For example, to list the definition of plus(?,?,^):

```
list plus(?,?,^).
```

To list all definitions of plus with three arguments:

```
list plus(A,B,C).
```

To list all relation definitions in the program:

```
list X.
```

list.

Same as "list X".

parcomp R.

Compiles all relations with relation specifier R. Each time the compilation of a new relation begins, the message

```
compiling r(m1,...,mn)
```

appears on the terminal. As each clause in the relation definition is begun, the clause number is displayed:

```
clause 1
clause 2
...
```

When the relation definition has been successfully compiled, the message

```
r(m1,...,mn) compiled
```

appears. If this message does not appear, then the compilation has been unsuccessful, and the error has occurred in the clause whose number was last displayed. In some cases an explanatory message will be printed, beginning with '!!!'; this will help to locate the error.

If the compilation of a relation definition is successful, two new clauses will be asserted. One of these clauses is for relation 'paraml' and contains the PARLOG Abstract Machine Language for the relation. The other is a clause for relation 'crelation' and contains the interpretable code for the relation. This is the code that is directly executed by the PARLOG interpreter. The 'paraml' clauses can be viewed by the 'listp' command (see later).

parcomp.

Same as "parcomp X".

listp R.

Lists on the terminal the PARLOG Abstract Machine Language for all relations with relation specifier R. This abstract machine code is in the form of a binary tree and is displayed in an indented layout to show the structure.

listp.

Same as "listp X".

delete R.

Deletes the PARLOG code for all relations with relation specifier R. Note that only the 'relation' clauses are deleted, not the 'paraml' or 'crelation' clauses; the latter are replaced only when the relation is recompiled.

delete.

Deletes all PARLOG, PARLOG Abstract Machine Language, and interpretable code from the program, i.e. all clauses for 'relation', 'paraml' and 'crelation'.

: C.

Compile and execute a query.

This command has two phases: firstly the PARLOG conjunction C is compiled and the compiled code is stored, then the PARLOG interpreter is invoked to execute the query.

When the compilation begins, the message

```
compiling query
```

is displayed. If the compilation is successful,

```
query compiled
```

will be displayed and the interpreter will be started.

The action of the interpreter will be explained later.

T : C.

Compile and execute the query C, then print the term C. Defined by the clause

```
(T : C) :- (: C & prolog(untrace) & printnl(T)).
```

list T : C.

Compile and execute the query C, then print the list T as a sequence of terms with one

term printed on each line. Defined by the clause

```
(list T : C) :- (: C & prolog(untrace) & printlist(T)).
```

T :: C.

Compile and execute the query C in parallel with printing the term T. Defined by the clause

```
(T :: C) :- (: C // printnl(T)).
```

list T :: C.

Compile and execute the query C in parallel with printing the list T as a sequence of terms. Defined by the clause

```
(list T :: C) :- (: C // printlist(T)).
```

partrace.

Switches on the PARLOG level trace. This will cause the PARLOG interpreter to print trace information.

untrace.

Switches off the PARLOG level trace.

5.2 The PARLOG interpreter

The PARLOG interpreter executes the most recently compiled query using the currently compiled program. Provided deadlock does not occur, the result of the evaluation will be displayed in the form

```
*** answer: A
```

where A is the result: true, false or error(E). The latter reports a run time error. The only run time error currently detected is

```
nodefinition(RN,MN)
```

where RN and MN are the relation name and mode name respectively. For example, if plus(?,?,^) is called but not defined, we would see the result

```
*** answer: nodefinition(plus,??^)
```

The interpreter gathers certain statistics and displays these at the end of the evaluation:

```
*** cycles: C
*** reductions: TR
*** ave r/c: AR
*** max r/c: MR
*** calls: N
```

C is the number of cycles, i.e. the number of times that the process list has been rotated during the evaluation. This figure is proportional to the execution time on a hypothetical parallel architecture with a sufficient number of processors.

TR is the number of process reductions at the PARLOG abstract machine level. As an approximate guide, there are usually about 20 process reductions for each PARLOG call.

AR and MR are the average and maximum number of reductions per cycle. AR gives a measure of the amount of parallelism in a program, while MR indicates how many processors would be necessary to achieve the execution time given by C.

Finally, N is the number of PARLOG calls reduced in the evaluation. This figure can be divided by the PROLOG execution time to compute the speed of the interpreter in LIPS.

5.3 Tracing

If PARLOG tracing is switched on, a trace message is printed at certain points:

```
--- call: L
```

traces a call to atom L. This message indicates that the interpreter is beginning to search for a candidate clause to solve the call L. If a candidate clause is eventually found, the call will be reduced and the message

```
--- reduce(N): L
```

will be displayed. This means that L has been reduced by using the Nth clause for the relation of call L. If no candidate clause is found, a fail message will be displayed:

```
--- fail: L
```

Similar trace messages will be displayed for 'prolog' calls, except that a 'prolog' call will either succeed or fail:

```
--- prolog: L
--- succeed: L
--- fail: L
```

All variables in trace messages are displayed using a special routine. 'V' is displayed for each variable except those on which some consumer processes are waiting, which are displayed as 'S' (for "suspension").

5.4 Example traced PARLOG evaluation

```
| ?- list *t : amerge([1,2,4],[2,3],*t).
compiling query
query compiled

--- call: amerge(??^,.(1,.(2,.(4,[]))),.(2,.(3,[])),V)
--- call: less(??,1,2)
--- call: less(??,2,1)
--- reduce(1): less(??,1,2)
--- reduce(1): less(??,2,1)
--- prolog: <(1,2)
--- succeed: <(1,2)
--- reduce(2): amerge(??^,.(1,.(2,.(4,[]))),.(2,.(3,[])),V)
--- prolog: <(2,1)
--- fail: <(2,1)
--- call: amerge(??^,.(2,.(4,[])),.(2,.(3,[])),V)
--- reduce(1): amerge(??^,.(2,.(4,[])),.(2,.(3,[])),V)
--- call: less(??,2,2)
--- call: less(??,2,2)
--- reduce(1): less(??,2,2)
--- call: amerge(??^,.(4,[]),.(3,[]),V)
--- reduce(1): less(??,2,2)
--- prolog: <(2,2)
--- fail: <(2,2)
--- prolog: <(2,2)
--- fail: <(2,2)
--- call: less(??,4,3)
--- call: less(??,3,4)
--- reduce(1): less(??,4,3)
--- reduce(1): less(??,3,4)
--- prolog: <(4,3)
--- fail: <(4,3)
```

```
--- prolog: <(3,4)
--- succeed: <(3,4)
--- reduce(3): amerge(??^,,(4,[]),,(3,[]),V)
--- call: amerge(??^,,(4,[]),[],V)
--- reduce(5): amerge(??^,,(4,[]),[],V)
--- prolog: untrace
1
2
3
4
*** answer: true
*** cycles: 175
*** reductions: 272
*** ave r/c: 1
*** max r/c: 5
*** calls: 23
```

6. ACCESSING THE PARLOG SYSTEM

To access the PARLOG system on the ICOT DEC-20, 'run' the file PARLOG.EXE in directory US1:<PARLOG>.

Some built-in relations are provided in three files in directory US1:<PARLOG>:

ARITH.PAR contains some arithmetic relations,
READ.PAR contains input relations,
PRINT.PAR contains output relations.

These files are listed with comments in the appendix.

ACKNOWLEDGEMENTS

Research on the PARLOG language and its implementation is supported by the UK Science and Engineering Research Council.

The PARLOG system was further improved while the author was a visiting scientist at ICOT - the Institute for New Generation Computer Technology in Tokyo, where this document was written.

REFERENCES

[ClG 83] K.L. Clark and S. Gregory,
PARLOG: a parallel logic programming language.
Research report DOC 83/5, Imperial College, London, May 83.

[ClG 84] K.L. Clark and S. Gregory,
Title to be decided.
In preparation.

[Kow 79] R.A. Kowalski,
Logic for problem solving.
North Holland, 1979.

[Sha 83] E.Y. Shapiro,
A subset of Concurrent Prolog and its interpreter.
Technical report TR-003, ICOT, Tokyo, January 83.

APPENDIX: SOME PRIMITIVE RELATIONS

File US1:<PARLOG>ARITH.PAR

```
% arithmetic relations:  less, lesseq, divides, plus, times

relation(less(?,?),
        less(*u1,*v1) ::- prolog(<,*u,*v)                    ).

relation(lesseq(?,?),
        lesseq(*u1,*v1) ::- prolog(=<,*u,*v)                 ).

relation(divides(?,?),
        divides(*u1,*v1) ::- prolog(is,0,*v mod *u)          ).

relation(plus(?,?,^),
        plus(*u1,*v1,*w) ::- prolog(is,*w,*u + *v) : true    ).

relation(plus(^,?,?),
        plus(*u,*v1,*w1) ::- prolog(is,*u,*w - *v) : true    ).

relation(plus(?,^,?),
        plus(*u1,*v,*w1) ::- prolog(is,*v,*w - *u) : true    ).
```

```
relation(times(?,?,^),
        times(*u!,*v!,*w) ::- prolog(is,*w,*u * *v) : true       ).

relation(times(^,?,?),
        times(*u,*v!,*w!) ::- prolog(is,*u,*w / *v) : true       ).

relation(times(?,^,?),
        times(*u!,*v,*w!) ::- prolog(is,*v,*w / *u) : true       ).
```

File US1:<PARLOG>READ.PAR

```
% readlist(x?):  x is an input list of variables, each variable is bound
%                to the next term read from the terminal
relation(readlist(?),
        readlist([*u^ | *x]) ::- read(*u) & readlist(*x)  par
        readlist([])      ).
% readlist(x^):  x is an output list of terms read from the terminal
relation(readlist(^),
        readlist(*x) ::- lazy read(*u) & readlist1(*u,*x)    ).

relation(readlist1(?,^),
        readlist1(end_of_file,[])  seq
        readlist1(*u,[*u | *x]) ::- readlist(*x)             ).

% read(x^):  x is the next term read from the terminal
relation(read(^),
        read(*u) ::- prolog(read,*u) : true                  ).
```

File US1: <PARLOG>PRINT.PAR

```
% printlist(x?):  prints a list of terms as a sequence of terms on the
%                 terminal
relation(printlist(?),
        printlist([*u | *x]) ::- printnl(*u) &
                                 printlist(*x) par
        printlist([])                                        ).

% printnl(t?):  prints term t on the terminal followed by a new line
relation(printnl(?),
        printnl(*t) ::- print(*t) & prolog(nl)               ).
```

```
% print(t?):  prints term t on the terminal
relation(print(?),
        print(*tl) ::- prolog(=..,*t,*tl) &
                       printargs(*tl)                              ).

relation(printargs(?),
        printargs([*f]) ::- prolog(write,*f) par
        printargs([*f,*a | *as]) ::- prolog(write,*f) &
                                     prolog(write,'(') &
                                     printas([*a | *as]) &
                                     prolog(write,')')              ).

relation(printas(?),
        printas([*a]) ::- print(*a) par
        printas([*a | *as]) ::- print(*a) &
                                prolog(write,',') &
                                printas(*as)                        ).
```