MU-PROLOG 3.0 reference manual

Lee Naish
Melbourne University
July 1983

Herein is a description of the facilities of version 3.0 of the MU-PROLOG interpreter. This is a reference manual only, not a guide to writing PROLOG programs.

/

1. Introduction

MU-PROLOG is (almost) upward compatible with DEC-10 PROLOG and UNIX PROLOG. The syntax and built-in predicates are therefore very similar. A small number of DEC-10 predicates are not available and some have slightly different effects. There are also some MU-PROLOG predicates which are not defined in DEC-10 PROLOG. However most DEC-10 and UNIX PROLOG programs should run with few, if any, alterations.

However, MU-PROLOG is not intended to be a UNIX PROLOG look-alike. MU-PROLOG programs should be written in a more declarative style. The non-logical "predicates" such as cut (!), \=, not and var are rarely needed and should be avoided. Instead, the soundly implemented not (~), not equals (~=) and if-then-else should be used and wait declarations should be added where they can increase efficiency.

2. Using MU-PROLOG

To use MU-PROLOG type the shell command "prolog". The interpreter will print a short message followed by a prompt. You are now at the top level of the interpreter and if you type PROLOG commands they will be executed. For example, if you type "[myfile]." then the file myfile in your current directory will be consulted (loaded). If you then type "ls." the program will be listed. Commands are PROLOG terms terminated by a full stop and a carriage return.

After you type a goal without variables the top level will print yes or no (depending on whether the goal succeeded or failed). If a goal with variables succeeds then the bindings of the variables are printed, followed by a question mark prompt. If you want to see the rest of the solutions then type a semicolon followed by a return; otherwise just hit return. It is possible that some subgoals get delayed and are never woken. If this happens then a message indicating how many calls have not been executed is printed. The interpreter has not proved or disproved the goal but it is often indicative of an infinite number of solutions.

Because commands are often repeated, especially during debugging, the system saves the most recent ones you type. Each top level prompt contains a command number. If the command you type is a positive number then the command of that number is repeated. If you type "-1." then the previous command is printed and "-2." means the one before that etc. The h (for history) command lists the saved commands.

Under Berkeley UNIX (at least), an alternative way to run MU-PROLOG is possible if you have a saved state (created by the save predicate) in a file (say savefile). If you type

```
                    prolog savefile arg1 arg2 ...
```

or equivalently (with Berekely UNIX, at least), just

```
                    savefile arg1 arg2 ...
```

then the state is restored and the arguments may be accessed with the argv predicate. In this mode, no message is written and interrupts are not trapped. There is a utility available on the system to save PROLOG programs. If you type "plsave f.pl dir", then a save file named f is created in directory dir (the default is the current directory). The program in f.pl should have a procedure called main, with one argument. When the saved file is run, main is called with

/

the command line list (from argv) as the argument. To save space, some of the facilities (the debugging package, for example) are not provided by plsave.

3. Built-in Predicates

Here are brief descriptions of the predicates supplied by the interpreter. There are also a number of library predicates which should be used freely, rather than re-inventing the same predicates with different definitions and names. The system also has many predicates with names starting with dollar signs. These are protected from the user but to avoid confusion, you should not start predicate names with dollars.

3.1. Internal Database Predicates

The following predicates are used in accessing and updating the database of clauses stored in main memory.

assert(X)
: Adds clause X (a rule or fact) to the database. It is the same as assertz.

asserta(X)
: Adds X at the start of a procedure.

assertz(X)
: Adds X at the end of a procedure.

clause(X,Y)
: There is a clause in the database with head X and body Y. The body of a fact is "true".

consult(X)
: The file with name X is consulted. X must be an atom. All clauses and definite clause grammar rules in the file are added to the database and goals are executed. Goals are written in the form "?-goal.".

deny(X,Y)
: Equivalent to retract((X :- Y)).

hidden
: Used in conjunction with hide(X) to hide the implementation of some predicates and make some procedures local to a file.

hide(X)
: X is a procedure name or list of procedure names in the form <procname>(<number of args>). If hide is called at the start of a file being consulted and hidden is called at the end then the effect is to make the procedure(s) local to that file. They cannot be accessed except by other procedures in that file. When hidden procedure names are printed they are followed by an underscore.

/

lib X
: Reconsults the file named X in the PROLOG library.

libdirectory(X)
: X is the atom whose name is the UNIX directory where the PROLOG library resides.

protect(X)
: X specifies a number of procedures, in the same way as with hide. The procedures are protected, so they can still be called but not listed or altered in any way.

reconsult(X)
: Reads clauses and grammar rules from file X to supersede existing ones (like consult but previous definitions of predicates are retracted).

retract(X)
: The first rule or fact that matches X is removed from the database. On backtracking, the next matching clause is removed.

retractall(X)
: Retracts all clauses whose head matches X.

[file1, file2, ...]
: Consult file1, file2, etc. If "-file1" is used then that file will be reconsulted instead.

3.2. I/O Predicates

These predicates are related to input and output. There are more low level I/O predicates in the UNIX section of this manual.

display(X)
: Write term X on the current output in prefix format. Equivalent to "writef(X,2'1011)".

eof(X)
: Equivalent to "X = (?-end)" (but more portable).

get(X)
: Reads characters from current input and returns X, the first printing character.

get0(X)
: Returns the next character, X, from the current input. At the end of file it returns 26 (^Z).

next(X,Y)
: Changes the standard input to X and the standard output to Y. If X(Y) is "user" then the input(output) is not changed. After calling next the new standard input and output are still referred to as "user". The old input and output are lost. By default read and write use the standard files but this can be overridden by using see and tell. These should generally be used in preference to next.

/

nl

A newline is printed on the current output.

op(X,Y,Z)

Declares Z, an atom (or list of atoms), to be an operator of type Y and precedence X. The standard operator declarations are nearly all the same as in DEC-10 PROLOG are as listed at the end of this manual.

portraycl(X)

Write clause X in a suitable format.

portraygoals(X)

Write goal X in a suitable format.

print(X)

Print X on the current output. If the user has defined a predicate called "portray", with one argument, then this is called. If "portray(X)" fails then "write(X)" is called.

printf(X,Y)

List Y is printed with format X. Y is a list of strings, constants and integers. X is a string specifying the format. Exactly the same conventions are used as with printf in the C language. For strings and constants a %s format is used, %d for decimal output of integers, %o for octal, %c for characters (small positive integers in PROLOG). Field widths can be specified in the same way as in C. The types of the elements of Y <u>must</u> match the format string.

putatom(X)

Atom X is printed on the current output.

put(X)

Character X is written on the current output. X may also be a string.

read(X)

Read a term terminated by a full stop and a whitespace from the current input. If the term contains variables these are considered distinct from all other variables. At end of file read returns "?-end". X must be a variable. If a syntax error in encountered, a message is printed and read fails.

see(X)

Switches current input to file X. If X has not already been opened by see then it is, otherwise the old file descriptor is used. After calling see, all calls to read, get, get0 and skip cause X to be read. X must be an atom.

seeing(X)

X names the current input file.

seen

Closes the current input file and reverts to the standard input.

skip(X)

Reads characters from current input until X appears or end of file is reached. If X is a list, then it reads until a member of the list is found.

/

tab(X)
    Prints X spaces on the current output.

tell(X)
    Switches current output to file X.

telling(X)
    X is the current output file.

told
    Closes current output and reverts to standard output.

wflags(X)
    If X is an integer, the write flags are set to X. If X is a variable, it is bound to the current value of the write flags. Write interprets the value as a bit string. If the 1 bit is set, terms are written in prefix format. If the 2 bit is set, names containing non-alphanumeric characters are quoted. If the 4 bit is set, lists of integers between 32 and 126 are written as strings. If the 8 bit is set, level numbers are written after variables, to distinguish between different variables with the same name. If the 16 bit is set, lists are written with the dot notation, instead of brackets. The write flags are initially set to 2'01100.

write(X)
    Writes term X on current output, taking into consideration current operator declarations and write flags. Write is currently written recursively, so for deeply nested terms some systems may have problems with the stack size.

writef(X,Y)
    Write term X using flags Y, rather than the current write flags.

writeln(X)
    The same as "write(X), nl".

3.3. Interactive Predicates

These predicates are usually called from the top level of the interpreter, or used for debugging, rather than being part of programs.

abort
    Aborts execution of the current goal and reverts to top level.

backtrace(X)
    Write the X most recent ancestors.

backtrace
    Write the 10 most recent ancestors.

break
    Causes a new invocation of the top-level interpreter. When this has finished, the previous computation is resumed.

debugging
    Lists all current spypoints.

/

h
    Print a history of the top level commands that have been saved.

ls
    Write all predicate definitions (except hidden and protected ones).

listing
    Write all predicate definitions (except hidden and protected ones).

ls X
    Write all definitions of predicates named X (except hidden and protected ones). X may be a list of predicate names.

listing X
    Write all definitions of predicates named X (except hidden and protected ones). X may be a list of predicate names.

nodebug
    Removes all spypoints. See "spy".

nospy X
    Removes any spypoints on procedure(s) X.

notrace
    Equivalent to trace(0).

restore(X)
    Restores the prolog state saved in file X. If X is not a compatible save file, restore fails. No files (other than the standard ones) should be open when restore is called.

save(X)
    Saves a copy of the current prolog state in file X. X is made an executable file which, when run or used as the first argument to prolog, restores the state and continues as if the save had just succeeded. No files (other than the standard ones) should be open when save is called.

spy X
    Places a spypoint on procedure(s) X. X is a procedure name or list of procedure names in the form <procname> or <procname>(<number of args>). When a procedure with a spypoint is called a message is printed and the user is able to trace and control the execution.

trace(X)
    Turns universal tracing on/off. X is an integer interpreted as a bit string. If the least significant bit (1) is one then each time the interpreter tries to match a call with a procedure head, a message is printed. The second bit (2) causes messages when backtracking occurs. The third bit (4) causes messages when a calls delay. The fourth bit (8) causes messages when delayed calls are woken. Some system predicates turn tracing off.

trace
    Equivalent to "trace(2'1111)".

/

3.4. Arithmetic Predicates

The following predicates deal with integers or integer expressions. Integer expressions may contain integers, variables bound to integer expressions, strings of length one and arithmetic operators. The allowable binary operators are +, -, *, /, mod, /\ (bitwise and), \/ (bitwise or), ^ (exclusive or), << (shift left), >> (shift right), and (logical and), or (logical or) and the relational operators <, =<, >, >=, =:= and =\=. The valid unary operators are - and \ (bitwise not). Logical expressions evaluate to one (true) or zero (false). All predicates which use integer expressions delay until all variables in the expression(s) are bound.

maxint(X)
: X is the largest integer possible in the system. The smallest is - X - 1.

X < Y
: Integer expression X is evaluates to less than integer expression Y.

X =< Y
: Integer expression X is less than or equal to expression Y.

X > Y
: Integer expression X is greater than expression Y.

X >= Y
: Integer expression X is greater than or equal to expression Y.

X =:= Y
: Integer expressions X and Y are equal.

X =\= Y
: Integer expressions X and Y are not equal.

X is Y
: Integer expression Y evaluates to X (a variable or integer). Remember that it delays if Y contains an unbound variable so it can only be used "in one direction", unlike plus.

plus(X,Y,Z)
: X + Y = Z. If at least two arguments are variables it will delay. If two are integers then the third will be calculated. If all three are integers it acts solely as a test.

length(X,Y)
: X is a list of length Y. X or Y may be variables. If they are both variables, then the call delays.

/

3.5. Control and Meta Level Predicates

Those predicates in this section which are non-logical, should be avoided where possible.

ancestor(X,Y)
: Y is the Xth ancestor (not including "call", ";" or ",") of the current call.

depth(X)
: X is the number of ancestors of the current call.

arg(X,Y,Z)
: The Xth argument of term Y is Z (delays if X or Y are variables).

functor(X,Y,Z)
: X is a term whose functor is Y and arity Z. Delays if X and either Y or Z are variables.

name(X,Y)
: Y is the list of characters in the name of atom X. If X and Y are variables, it delays.

atom(X)
: X is an atom (non-logical). If X is currently a variable it fails.

atomic(X)
: X is an atom or integer (non-logical).

int(X)
: X is an integer. If X is currently a variable it delays.

integer(X)
: X is an integer (non-logical). If X is currently a variable it fails.

var(X)
: X is currently a variable (non-logical).

nonvar(X)
: X is not currently a variable (non-logical).

not X
: If X succeeds then not X fails and if X fails then not X succeeds. The result is suspect if X succeeds and binds any variables (use ~ instead).

~X
: Sound implementation of negation as failure. If X contains variables then it delays, otherwise if X succeeds then ~X fails and if X fails then ~X succeeds.

occurs(X,Y)
: Term Y contains a subterm X. X must be a variable, an atom or an integer (non-logical).

error(X)
: Called when an error occurs (see section 5 of this manual).

/

errhandler(X)
    Called by "error" if the user-defined procedure "traperror(X,Y,Z)" fails (see section 5).

repeat
    Always succeeds, even on backtracking.

true
    Succeeds. On backtracking it fails.

fail
    Always fails.

wait X
    X is a procedure head with all its argument either 1 or 0. The corresponding wait declaration is added to the procedure. See section 4 of this manual.

call(X)
    The goal represented by the term X.

X
    The goal given by the binding of the variable X (meta-variables).

X, Y
    X and Y (the bodies of clauses are in this form).

X ; Y
    X or Y (use sparingly - use an extra clause instead).

X =.. Y
    Y is a list made up of the functor of X followed by the arguments of X. The call delays if X and Y are insufficiently instantiated. If X is an integer it fails.

X == Y
    Terms X and Y are identical. That is, they can be unified without binding any variables (non-logical).

X \== Y
    X and Y are not identical (non-logical).

X = Y
    X equals Y (X and Y are unified).

X \= Y
    The same as not(X = Y). ie. unsound implementation of not equals.

X ~= Y
    Sound implementation of inequality. If X and Y do not unify it succeeds. If X and Y unify without binding any variables then it fails. If X and Y unify but variables need to be bound then it delays. Underscores in the call are not treated as other variables are. It is assumed that they are universally quantified. For example, X ~= f(_) means for all possible values of _, X does not equal f(_) (so it fails if X equals f(Y), whatever Y is).

if X then Y

If X contains variables then it delays, otherwise if X succeeds then Y is called. If X fails the goal succeeds.

if X then Y else Z

If X contains variables then it delays, otherwise if X succeeds then Y is called and if X fails then Z is called.

X -> Y

Unsound implementation of "if X then Y". It does not delay if X contains variables (non-logical).

X -> Y ; Z

Unsound implementation of "if X then Y else Z". It does not delay if X contains variables (non-logical).

!

The cut operation. It succeeds, but on backtracking everything fails up to and including the most recent ancestor which is not ";", "call" or ",". No atoms to the left of the cut are retried and no more clauses in that procedure are tried. It can be used to implement many of the unsafe features of PROLOG (for example, not, \=, var and ==). It should be used as sparingly as possible.

3.6. UNIX-related Predicates

These predicates provide an alternative interface to the file system and other facilities provided by UNIX. The number of predicates in this section is likely to grow with demand.

open(X,Y,Z)

Opens file X (an atom) on channel Y in mode Z. There are currently twenty file channels available. Numbers zero to four are reserved for standard input, standard output, standard error output, current input and current output, respectively. If Y is a variable, then it is bound to the highest free channel. The mode must be 'r' (read), 'w' (write) or 'a' (append).

getc(X,Y)

Y is bound to the value of the next character read on file channel X.

putc(X,Y)

Character Y is written on file channel X.

read(X,Y)

A term Y is read from file channel X.

write(X,Y)

Y is written on file channel X.

write(X,Y,Z)

Y is written on channel X using flags Z.

writeln(X,Y)

Equivalent to "write(X,Y), putc(X,10)".

/

fprintf(X,Y,Z)
    Print list Z on file channel X using format Y.

pipe(X)
    Creates a pipe named X. X must be an atom. X may then be used with "see", "tell" and "next".

pipe(X,Y)
    Creates a pipe using file channel X for the input from the pipe and file channel Y for output to it. If X or Y are variables they are bound to the highest free channels.

fork
    Creates another prolog process, with the same core image. The only difference is that the call to fork in the parent process succeeds but the call in the child process fails. Care must be taken to ensure the two processes do not compete for input from the terminal or other open files.

system(X,Y)
    Command line X (an atom) is passed to UNIX to execute and Y is then bound to the exit code returned.

argv(X)
    X is a list of atoms which were the command line arguments typed by the user. The first element of the list will be the name of the save file.

csh
    Invokes a new copy of the UNIX shell (csh). The prolog process is suspended until the shell process terminates.

sh
    Invokes a new copy of the UNIX shell (sh).

shell(X)
    Calls the UNIX shell with the string X as a command line.

more(X)
    Lists file X using the "more" command.

edit(X)
    The editor "ed" is used to edit file X. When the editing is completed, the file is reconsulted.

exit(X)
    The prolog process terminates with exit code X (an integer).

getuid(X)
    Binds X to the uid of the user running prolog.

isuser(X)
    Checks the password file for user-name X (X must be a string).

sig(X,Y)
    Used to trap, ignore or set to default, the various UNIX signals. X is the number of a signal as defined in <signal.h>. Y must be zero, to set the signal to the default, one, to ignore the signal, or two, to trap the signal. The most common use is "sig(2,2)" which traps interrupts. The

/

library file "signal" gives a more high level interface.

4. Wait Declarations

Procedures without wait declarations behave as normal PROLOG procedures - calls to them succeed or fail. If wait declarations are added, calls may also delay. This happens when the call unifies with the head of some clause but in doing so, certain of variables in the call are bound. Wait declarations can be used to prevent infinite loops and to enable coroutining between sub-goals, which often increases efficiency.

Wait declarations are most easily added by putting goals in the files containing programs. A typical predicate definition would have a couple of goals to add wait declarations ("?-wait ....") to the clauses for the procedure. The argument to the call to wait should look like the head of one of the clauses with each argument being a one or a zero. A one signifies that the corresponding argument in a call to the procedure may be constructed and a zero means that it may not.

As each argument of a call is being unified with the corresponding argument in a procedure head we check if it is constructed. An argument is constructed if a variable in it is unified with a non-variable or another variable in the call. If the unification succeeds, the result is a mask of ones and zeros, representing which arguments were and were not constructed. This is then compared with the wait declarations. If there is a wait declaration with ones corresponding to all ones in the mask then we succeed; otherwise we delay.

When a call is delayed the bindings are removed. Thus from a logical point of view nothing has happened. However, the variables that were bound are marked and when any of them are bound by some other call, the delayed call is woken. A call may bind several marked variables and each one may have been marked several times so any number of delayed calls may be woken at the same time. The order of subsequent calls is as if the woken calls were all at the start of the clause that just matched, in the order that they were delayed. For further discussion of the use of wait declarations, see "An Introduction to MU-PROLOG". There is also a library program, called genwait, which can produce reasonable wait declarations for most procedures.

5. Errors

Whenever an error occurs, procedure error is called with the error code and the call that caused the error as arguments. The call to error in effect replaces the call that caused the error. This allows you to simulate failure, success and replacing the call with another call. When error is called it always prints a message (on standard error) and depending on the error it will do various things.

If the program is almost out of memory then an error is generated, the last ancestors are written and abort is called. If you hit the interrupt key an error is generated. If you hit it again it will get you back to the top level of the interpreter (abort is called). It is an error to call a predicate for which no clauses or wait declarations have ever been added. In such cases, a warning is printed and (in effect) the call fails.

Other errors will cause the error code (and/or some explanation) and the call to be printed, followed by a question mark prompt. The simplest thing to do is to hit interrupt, which generates an abort. If you type a goal it will be executed in place of the call (eg "fail." will cause a failure). If you type a clause then the call is unified with its head and then the body is executed. This can save some typing and allows you to change the call but still use the same variables.

MU-PROLOG also allows users to write their own error handlers, rather than rely on the default one outlined so far. Procedure error is defined in (something like) the following way:

```
        error(Ecode, Call) :- traperror(Ecode, Call, X), !, call(X).
        error(Ecode, Call) :- errhandler(Ecode, Call).
```

User programs may contain definitions of traperror and other predicates to trap and handle errors, respectively. For example, if you want "is" to fail if it encounters any errors and you want to query the user if an undefined predicate is called, then the following code would do.

```
        traperror(_, _ is _, fail).
        traperror(enoproc, Call, askabout(Call)).

        askabout(Call) :- ...
```

The following error codes are currently used:

eelist	list expected
eeconst	constant expected
eeint	integer expected
eefunct	functor expected
eestring	string expected
ee01	one or zero expected
eevar	variable expected
eerwa	r, w or a expected
euint	unexpected integer
eufunct	unexpected functor
euvar	unexpected variable
enoproc	undefined procedure called
eprotect	protection violation
eopen	cant open file
efile	invalid file specification

/

6. Standard Operator Declarations

```
?- op(1200, fx, (?-)).
?- op(1200, fx, (:-)).
?- op(1200, xfx, (:-)).
?- op(1200, xfx, (-->)).
?- op(1170, fx, (if)).
?- op(1160, xfx, (else)).
?- op(1150, xfx, (then)).
?- op(1100, xfy, (->)).
?- op(1050, xfy, (;)).
?- op(1000, xfy, ',').
?- op(900, fy, ls).
?- op(900, fy, listing).
?- op(900, fy, wait).
?- op(900, fy, ~).
?- op(900, fy, not).
?- op(900, fy, \+).
?- op(900, fy, nospy).
?- op(900, fy, spy).
?- op(900, fy, lib).
?- op(700, xfx, =).
?- op(700, xfx, ~=).
?- op(700, xfx, \=).
?- op(700, xfx, is).
?- op(700, xfx, =..).
?- op(700, xfx, ==).
?- op(700, xfx, \==).
?- op(700, xfx, =:=).
?- op(700, xfx, =\=).
?- op(680, xfy, or).
?- op(660, xfy, and).
?- op(630, xfx, <).
?- op(630, xfx, >).
?- op(630, xfx, =<).
?- op(630, xfx, >=).
?- op(600, xfy, '.').
?- op(500, yfx, +).
?- op(500, yfx, -).
?- op(500, yfx, /\).
?- op(500, yfx, \/).
?- op(500, fx, (+)).
?- op(500, fx, (-)).
?- op(500, fx, \).
?- op(400, yfx, *).
?- op(400, yfx, /).
?- op(400, yfx, <<).
?- op(400, yfx, >>).
?- op(300, xfx, mod).
?- op(200, xfy, '^').
```