JWL

MU-PROLOG (Version 1)

USER MANUAL

March 1981

J.W. LLOYD

Contents

1. Introduction

This manual briefly describes the facilities available in MU-PROLOG. Version 1 is experimental and rather primitive. Later versions will improve the user interface and provide many more facilities. Any problems should be reported to the author.

MU-PROLOG (Melbourne University PROLOG) is an interpreter written in PASCAL for the programming language PROLOG (PROgramming in LOGic). PROLOG is based on first order logic. In fact, the interpreter is essentially a theorem prover, highly specialized so that it can efficiently run user programs. For the background on PROLOG and logic programming, the reader is referred to [2], [3], [4], [5].

A user runs the interpreter in a similar way to an interactive LISP system. In fact, PROLOG has close similarities to LISP, although based on a different foundation (first order logic rather than the λ-calculus). A user can load previously edited programs and run them in an interactive way. Various debugging aids are provided. Furthermore, a number of built-in system predicates are available, which provide integer arithmetic capabilities.

MU-PROLOG, like nearly every other PROLOG interpreter, uses backtracking as the search strategy. Backtracking is implemented in the standard way using a stack. This stack contains the goal clauses which are generated during the computation. In fact, a computation is essentially just an interleaved sequence of stack pops and pushes. The computation is successful when the NIL clause is generated.

2. Command Predicates

The interpreter is called up by the command

```
prolog
```

The system responds with the reply

```
MU-PROLOG version 1

<-
```

The system prompt is always <-. The user is now in the top level of the interpreter. At this level the user may load programs, trace, run programs etc. The generic term for predicates which do such things as load programs and so on is command predicate. The following are those command predicates available in version 1.

load

The load command has the form

```
ld(filename).
```

Note the full stop. ld takes one argument. The interpreter looks for a

file called filename.pl in the current directory and tries to load it as a PROLOG program. Syntax errors, if any, are reported. When the command is finished, the interpreter responds with <-.

list

The list command has the form

ls.

This command lists the currently loaded program (there can be only one at a time). It also numbers the clauses in the program. Other command predicates refer to these numbers. Program clauses appear 12 at a time. To get the next lot, type y return (or simply return twice). To cease listing, type n return.

trace

By default the trace facility is off. To turn it on, the command is

tr.

and to turn it off, the command is

tf.

When the trace is on, the interpreter displays on the screen the goal clauses as they are generated during the running of the program. The number on the left of the clause refers to the program clause which was one parent of the current goal clause. The other parent is always the first atom in the immediately preceeding goal clause.

Subscripting is used to distinguish variables with the same identifier in a particular goal clause. Thus x:14 is to be read x_{14}.

The trace command is very expensive because the interpreter has to reconstruct each goal clause from a rather complicated internal representation. However, trace provides a very useful debugging aid and also a check that a program is performing as expected.

Goal clauses are displayed 5 at a time. To discontinue the trace, type n return. To continue the trace, type y return (or return twice). Stack pops are indicated by

**** pop ***

Proof

This command has the form

pr.

The proof command displays goal clauses (5 at a time) starting with the initial goal clause and following all the way down the branch which led

to the successful computation, finally ending with NIL. (This is to be compared with trace which prints all goal clauses as the interpreter traverses the search tree during the computation).

next

The next command has the form

nx.

This command is used to find the next solution, and by repetition, all solutions to a problem. Next pops the last goal clause (NIL) off the goal stack and continues the computation until the next solution is found. This solution will generally be different to the previous solution, although, depending on the program, it need not be. The command can be repeated. When there are no more solutions, the interpreter responds with "no".

Exit

To exit the interpreter, the command is

ex.

Occur

The command predicate to turn the occur check on is

oc.

The command to turn it off is

of.

By default, the occur check is off. When the occur check is on, during unification the interpreter checks whether a term, which is about to be bound to a variable, contains that variable. If it does the proposed unification should fail. With the occur check off (the default), the interpreter will quite happily make the binding (and subsequently get itself into a dreadful tangle!)

However, it is important to realise that only the most contrived PROLOG programs will run into this problem and, since the occur check is very expensive, current interpreters do not make the check.

If you need the occur check to make a program run, there is probably something wrong with your program!

3. System Predicates

The interpreter contains a number of system predicates. These are built-in predicates mainly relating to arithmetic capabilities. These may be used in programs. When called, they are handled specially by the interpreter with PASCAL procedures.

Command predicates and system predicates are contrasted with user predicates, which are predicates defined by clauses in user programs.

Arithmetic Predicates with 2 Arguments

These predicates are

lt	less than
le	less than or equal to
eq	equal to
ne	not equal to

Each of the above predicates has 2 arguments which must both be bound to integers at the time the predicate is encountered during the computation. If one or both of the arguments are not bound to integers an error occurs and the computation ceases. The predicate succeeds when the stated relation holds, otherwise it fails.

Arithmetic Predicates with 3 Arguments

These predicates are

plus	(integer) addition
minus	subtraction
mult	multiplication
div	division
mod	remainder modulo

Thus, plus (x,y,z) means x+y=z and so on. Each of these predicates expects the first two arguments to be bound to integers at the time the predicate is encountered during the computation. Otherwise, an error occurs and the computation ceases with an error message. The third argument can either be a ("free") variable or be bound to an integer. In the first case, the variable is bound to the value resulting from the operation, and the predicate succeeds. Otherwise, the interpreter checks whether the stated relation holds between the 3 integer arguments, succeeds if it holds and fails if it does not hold.

Write and Newline

The write predicate has the form

write(arg1. arg2. arg3. ... argn)

When the write predicate is encountered during a computation, it transmits output to the terminal. For those arguments which are variables with a current binding the predicate writes out the current bind-

ing. Arguments with no current binding are simply written out directly. Arguments which are integers cause that number of blanks to be written. The dots in the write predicate cause a single blank to be output. Thus if x is currently bound to 34.123.847.NIL, then:

```
write(5.Sorted.list.is.x)
```

causes the message

␣␣␣␣␣Sorted␣list␣is 34.123.847.NIL

to be output.

One can cause newlines to be output by the predicate

```
nl
```

fail

The system predicate

```
fail
```

always fails when encountered as a goal during the computation.

4. Control Predicate

Apart from the ordering of program clauses and the ordering of atoms in a program clause, one other control feature is provided. This is the "cut" control predicate, written !.

! is inserted into the body of a program clause like an ordinary atom. However, it has a special behaviour. When first encountered as a goal during the computation, cut succeeds immediately. If backtracking should later return to the cut, the effect is as follows. The interpreter will pop clauses all the way back to the parent clause which first caused the cut to be introduced. (That is, this clause matched a clause which contained the cut in its body). The parent clause is itself then popped (i.e. failed). For further discussion of the use of the cut predicate the reader is referred to [6].

5. Syntax

A MU-PROLOG program consists of a sequence of clauses which are essentially first order logic Horn clauses of the form

```
p <- q, r, ..., z.
```

where the p,q, ... are atoms. The only variation from logic notation is that "," denotes conjunction and each clause must end with a full stop. In general, the right hand side can be empty in which case the clause is written

```
p.
```

The left hand side of the clause is called the clause (or procedure) head and the right hand side is called the clause (or procedure) body.

A loaded program is run by giving it an initial goal clause of the form

s, t, ..., x.

If the computation succeeds and there are variables in the goal clause, their values are printed out at the end. If there are no variables and the computation succeeds the interpreter responds with "yes", otherwise it responds with "no".

A primitive commenting facility is available. Just insert in the program, at any point between clauses, something like

comment (this.is.a.very.primitive.comment).

There is one reserved function available. This is the cons function, familiar from LISP. cons(x,y) is the list with x as the first element and y as the rest. For notational convenience, the dot notation for cons is provided. Thus

cons(x,y) can be written x.y

Hence, cons(A, cons(B, cons(C, NIL))) is the list

A.B.C.NIL

NIL is conventionally used as a list terminator. The cons and dot notation are interchangeable. Thus, one could write for example

cons(A,B.NIL).

The interpreter always outputs the dot notation. The only restriction is that it does not understand, for example

((A.B).C).

Instead, one must write:

cons(cons(A,B),C).

To distinguish between variables and constants in a program the following convention is used. A function with no arguments is a constant if its first letter is upper case, otherwise it is a variable.

Identifiers must be no more than 10 characters long.

6. Error Messages

A fairly comprehensive list of errors is reported. The following is a complete list of error messages with their explanation.

Syntax error in goal - the initial goal clause has a syntax error.

Since the syntax for MU-PROLOG is so trivial the precise problem will be evident.

Syntax error in clause n - the nth clause in the program being loaded has a syntax error.

Unexpected eof - the file containing the program does not contain a complete program.

Predicate not found: identifier - the identifier is not known as a predicate to the interpreter.

Wrong no of args in goal - a predicate or function in the goal contains the wrong number of arguments. For a user defined predicate or function, this means that an earlier occurrence had a different number of arguments. It could also mean that a predicate or function had > maxargs-1 arguments. Maxargs-1 is the maximum number of arguments allowed.

Wrong no of args in clause n - the nth clause has a predicate or function with the wrong number of arguments.

Predicate table overflow - the program being loaded has too many predicates to fit into the predicate table.

Function table overflow - the program being loaded has too many functions to fit into the function table.

Goalstack overflow - the stack containing the goal clauses generated during the computation has become full. Your program probably has a bug!

Command predicate illegally used - it is not legal to mix command and user predicates in a goal clause. Also it is not legal to have a goal with more than one command predicate in it. So, for example,

```
ld(fred),ls.
```

is illegal.

Too many vars in goal - the goal clause contains too many variables to fit into the variable table.

Predicate not defined in program: identifier - a predicate has been encountered during the computation which is not a system predicate and does not appear on the left hand side of any program clause.

Error in system predicate - one of the arguments of one of the arithmetic predicates is not bound to an integer. The offending predicate can be found by trace.

7. A Note on the Implementation

The main implementation technique used in the program is the structure

sharing idea of Boyer and Moore [1]. In fact, the basic unification routines in the interpreter are essentially the algorithms given in [1]. However, they have been partly improved by removal of recursion and addition of lots of lovely goto's!

The only real disadvantage of their scheme is that goal clauses on the goal stack have to be reconstructed from their component parts before they can be printed, as in a trace, for example. Otherwise, their scheme gives a fast and space efficient implementation.

The internal data structures used are given in Appendix 2. This should be read in conjunction with a program listing. The main data structure is the predicate table, which is a table of predicates, user, system and command. Program clauses with the same predicate in the head are put on a linked list which hangs off the corresponding predicate in the predicate table.

8. A Sample Session

```
$ prolog
MU-PROLOG  version 1
<-ld(member).
<-ls.
   1  member(x,x,y) .
   2  member(x,y,z) <- member(x,z) .
<-member(x,123,4,324,NIL).
x = 123
<-nx.
x = 4
<-nx.
x = 324
<-pr.
    2  member(x:2,4,324,NIL) .
    2  member(x:3,324,NIL) .
    1  NIL
<-nx.
no
<-ld(front).
<-ls.
   1  front(n,z,x) <- append(x,y,z) , length(x,n) .
   2  length(NIL,0) .
   3  length(u.x,n) <- length(x,n1) , plus(n1,1,n) .
   4  append(NIL,x,x) .
   5  append(u.x,y,u.z) <- append(x,y,z) .
<-tr.
<-front(2,A,B,C,D,NIL,z).
    1  append(z,y:1,A,B,C,D,NIL) , length(z,2) .
    4  length(NIL,2) .
*** pop ***
    5  append(x:1,y:2,B,C,D,NIL) , length(A,x:1,2) .
    4  length(A,NIL,2) .
    3  length(NIL,n1:1) , plus(n1:1,1,2) .


    2  plus(0,1,2) .
*** pop ***
*** pop ***
*** pop ***
    5  append(x:1,y:3,C,D,NIL) , length(A,B,x:1,2) .
    4  length(A,B,NIL,2) .
    3  length(B,NIL,n1:1) , plus(n1:1,1,2) .
    3  length(NIL,n1:1) , plus(n1:1,1,n1:2) , plus(n1:2,1,2) .


    2  plus(0,1,n1:3) , plus(n1:3,1,2) .
    0  plus(1,1,2) .
    0  NIL
z = A,B,NIL
<-pr.
    1  append(z:2,y:1,A,B,C,D,NIL) , length(z:2,2) .
    5  append(x:1,y:2,B,C,D,NIL) , length(A,x:1,2) .
    5  append(x:1,y:3,C,D,NIL) , length(A,B,x:1,2) .
    4  length(A,B,NIL,2) .
    3  length(B,NIL,n1:1) , plus(n1:1,1,2) .


    3  length(NIL,n1:1) , plus(n1:1,1,n1:2) , plus(n1:2,1,2) .
    2  plus(0,1,n1:3) , plus(n1:3,1,2) .
    0  plus(1,1,2) .
    0  NIL
<-ex.
$
```

9. References

[1] R.S. Boyer and J.S. Moore, The Sharing of Structure in Theorem-proving Programs, MI7, 101-116.

[2] R. Kowalski, Logic for Problem Solving, North Holland, 1979.

[3] R. Kowalski, Algorithm = Logic + Control, CACM, 22(1979), 424-436.

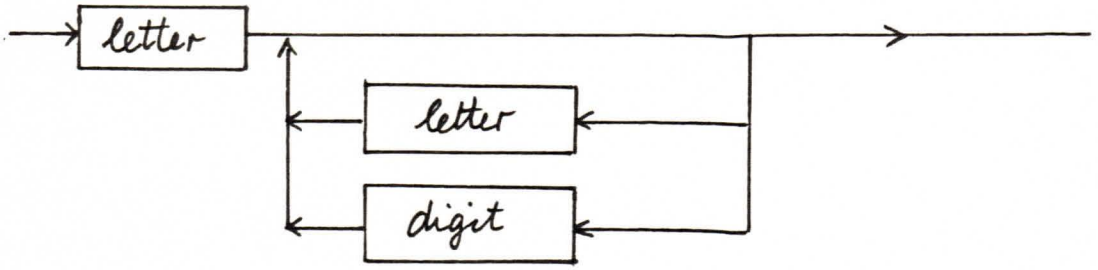
[4] R. Kowalski, Predicate Logic as Programming Language, IFIP74, 569-574.

[5] D.McDermott, The PROLOG Phenomenon, SIGART, 1980.

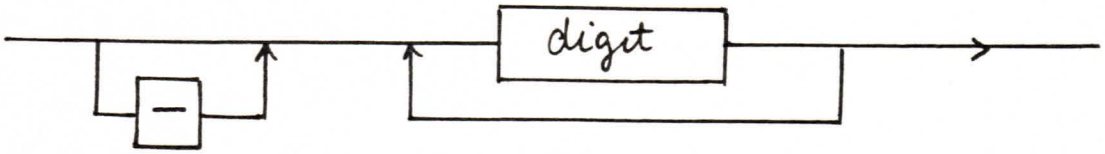
[6] D.H. Warren, Implementing PROLOG - compiling predicate logic programs, Vol. 1 and 2, D.A.I. Research Report Nos. 39 and 40, Edinburgh University, 1977.

Appendix 1. Syntax Diagrams

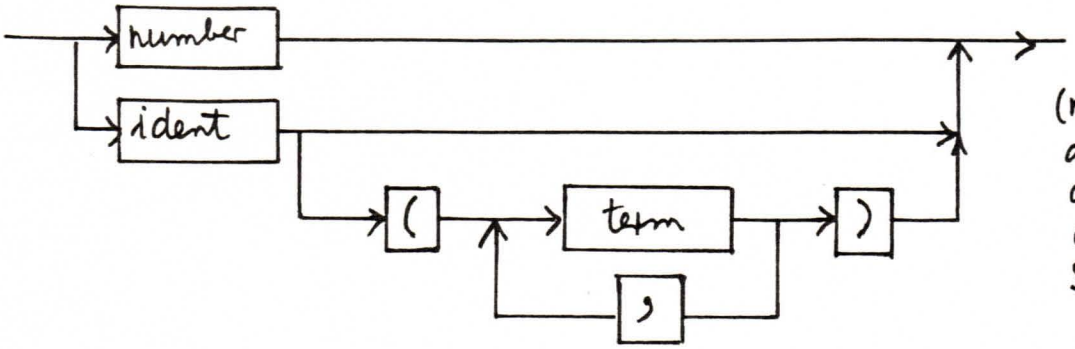
ident



number

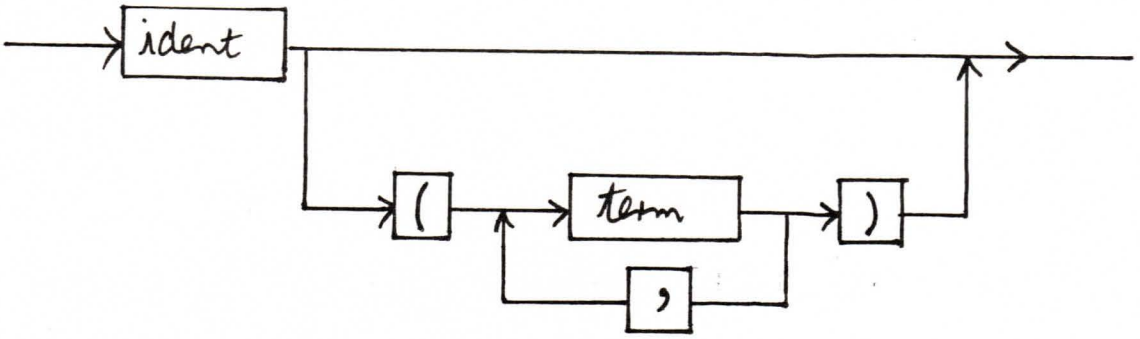


term

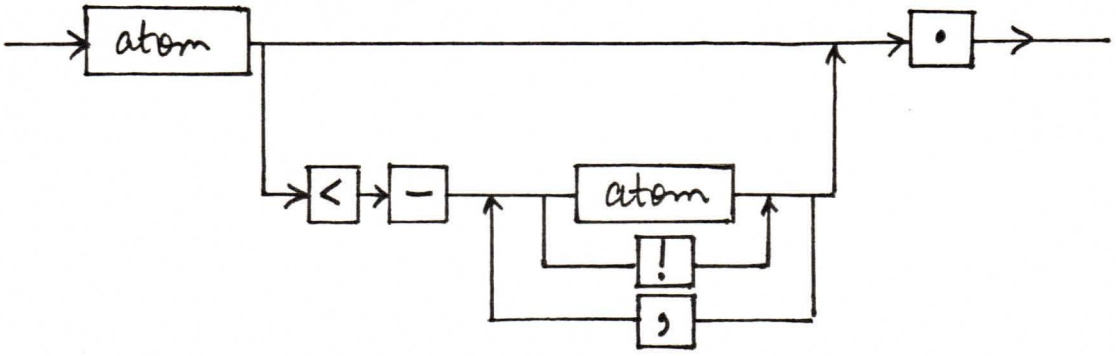


(note: cons also allows alternative dot notation. See section 5)

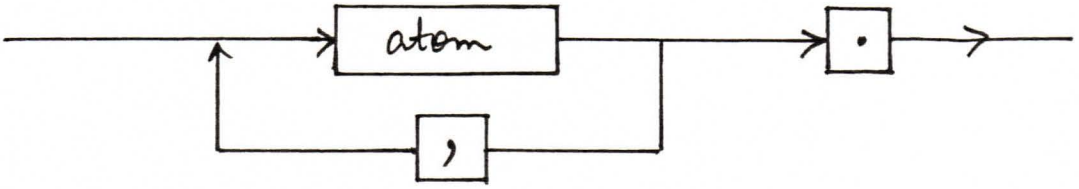
atom



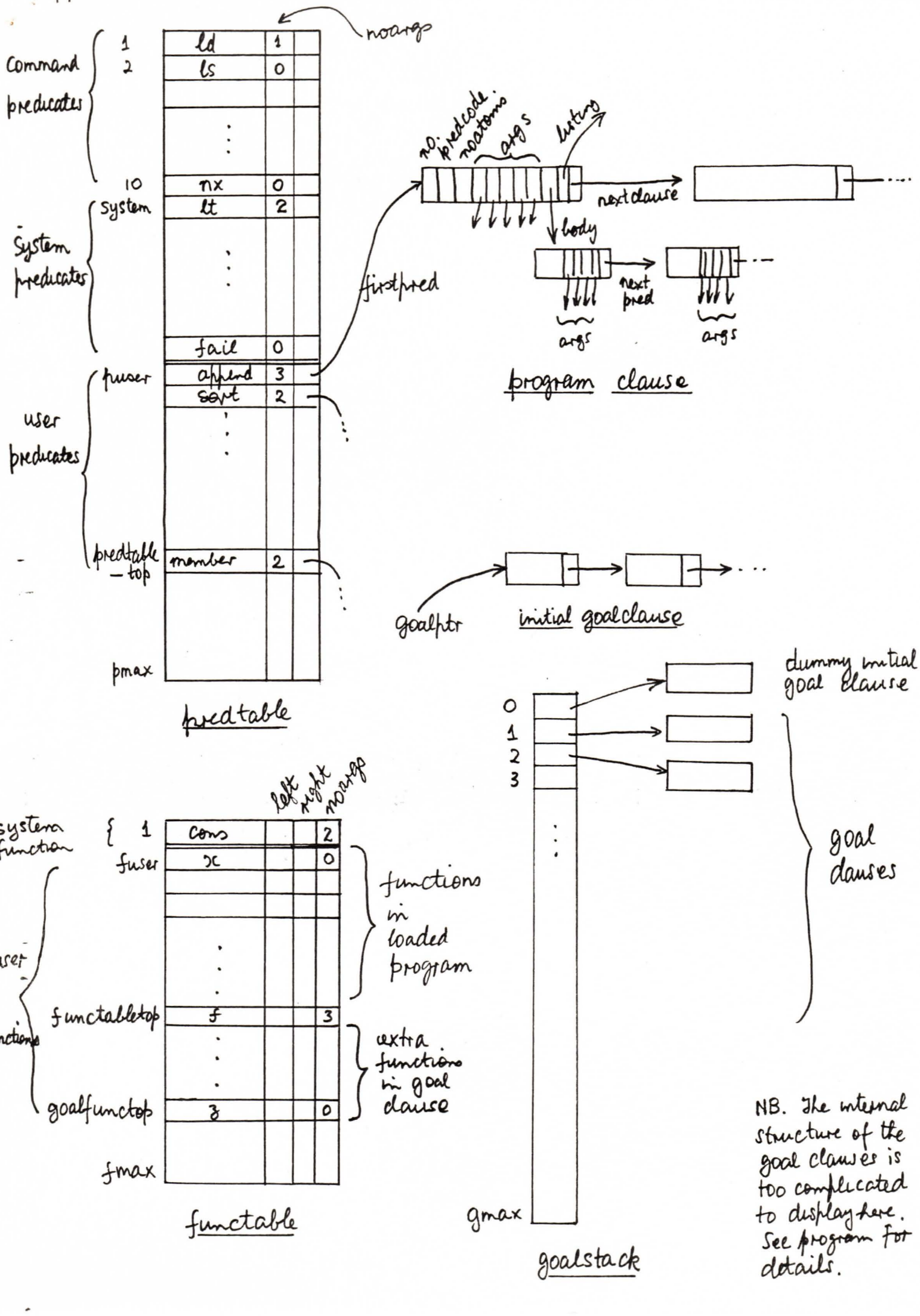
program clause



goal clause



Appendix 2. Internal Data Structures



NB. The internal structure of the goal clauses is too complicated to display here. See program for details.

Appendix 3. Current Values of Constants

pmax = 200
pmax is the length of the predicate table. 22 locations are taken up by command and system predicates.

fmax = 200
fmax is the length of the function table.

maxargs = 7
maxargs -1 is the maximum number of arguments allowed in any predicate or function.

gmax = 1000
gmax is the goal stack depth.

vmax = 10
vmax is the maximum number of variables allowed in any initial goal clause.

In addition, as a rough way of preventing endless loops, the interpreter writes a message after a certain large number of successful resolutions have been completed during a computation. It is possible, by typing y or n return, to continue or discontinue the computation. The current limit is 1100.

All the above constants are subject to change in the future. In particular, gmax will most likely be increased as programs get larger.

Addendum to MU-PROLOG (version 1) user manual

JWL

p3. Current version is 1.1

p3. load command can also have form

ld(f1. f2. f3).

This loads files f1.pl, f2.pl, f3.pl

One can also write

ld ("filename").

Only the first 10 characters of string are used and the .pl extension is not added.

p6 The write statement has been replaced by the following:

write (x) outputs the term x.

So if x is bound to 1.2.NIL, then

write (func (x)) produces func (1.2.NIL).

Similarly, write ('string') produces string

The convention of outputting blanks when an integer is met in a list has disappeared.

p8. End of line comments are supported. So one could have

append (NIL, x, x). # This is a comment

p15. maxargs has been increased to 8.

strings as indicated above, strings are now supported

'<any string of printable chars>'

is a special type of constant

To put a quote in a string, write two quotes, thus

'This isn''t wrong'

is an acceptable string.