micro-PROLOG 2.12
Programmer's Reference Manual

CP/M Version *

F.G.McCabe

Revised Second Edition

This manual describes the micro-PROLOG system from the programmer's point of view. It describes the syntax of micro-PROLOG, the various built-in features, and how to interact with the system. Also included is a chapter which describes the machine code level interface, and how to augment the system with built-in predicates. In so far as it is specific to any particular micro-PROLOG implementation, this manual describes the CP/M version.

The specification of micro-PROLOG is subject to change without notice.

September, 1980
May, 1981

2nd Printing

Contents

Chapter 1

Introduction

This manual describes the micro-PROLOG programming system. PROLOG is a computer language based on predicate logic, in particular the clausal form of logic [Robinson 1965,1979]. The procedural interpretation of logic was conceived in 1972 by Kowalski [1974], this enabled logic to be viewed as a concrete programming language.

The first PROLOG (which stands for PROgramming in LOGic) was implemented in 1972 in Marseilles by Colmeraur and Roussell [Colmeraur 1973] in the medium level programming language ALGOL-W. A more efficient and improved implementation was made in 1973 [Roussell 1975], this time in FORTRAN. This implementation reached a wide audience in countries as far afield as Poland, Hungary, U.S.A., Canada, Sweden, Portugal, Belgium and the U.K. Subsequently to the Marseilles PROLOG various other implementations have been built, the principal ones being in London [Clark and McCabe 1979], Edinburgh [Warren et al. 1978], Waterloo [Roberts 1977] and a new implementation from Marseilles [Kanoui & Van Caneghem 1980].

micro-PROLOG is a small, disk based implementation for the Z-80 microprocessor operating under CP/M. The objective of micro-PROLOG has been to build a very basic system, which can easily be extended by the user. As part of this philosophy the built-in syntax is very simple and basic, but also very flexible. Extensibility is further enhanced by the provision of an interface for adding sub-programs written in other languages, for example ASSEMBLER and FORTRAN.

This manual is not a primer for logic programming, but it is intended to be used as a reference manual by the programmer who already has some basic knowledge about logic programming. It describes the micro-PROLOG system in some detail, but does not attempt to teach the principles of logic programming to the novice. For an introductory text on Logic as a programming formalism see the primer "A micro-PROLOG primer" by [Clark, Ennals & McCabe 1981], and the book "Logic for Problem Solving" [Kowalski 1979]; for a more formal treatment see "Predicate Logic as a computational Formalism", by Clark [1979].

The reader should also have some knowledge of the CP/M operating system, or be able to access a CP/M manual.

Chapter 2

Syntax of micro-PROLOG

The syntax of micro-PROLOG is very simple, making up in generality what it lacks in sophistication. Most mainframe PROLOG systems nowadays have very powerful grammars built into them; typically some kind of operator precedence grammar. While this might have been desirable, space limitations make this difficult to do in micro-PROLOG. Instead we modelled the syntax of micro-PROLOG on LISP syntax [McCarthy 1962].

There are only four different kinds of syntactic objects that micro-PROLOG knows about: Numbers, Constants, Variables and Lists; which are all kinds of *term*. Note that there are no general facilities for arbitrary function symbols, the only one allowed is the list constructor and for which there is a specially condensed syntax c.f. DEC-10 PROLOG and LISP.

2.1 Character set

micro-PROLOG uses the 7 bit ASCII character set, together with extra characters, such as special graphics characters, that may be supported by the underlying machine. Characters are represented internally by 8 bit numbers in the range 1..126. The characters corresponding to 0 and 127 are not legal in micro-PROLOG, and are ignored if used.

2.2 Numbers

Numbers are integers in the range $-2^{15}..2^{15}-1$. A positive number is written as a contiguous sequence of digit characters, with no leading sign character, eg. 0 30 1025 32767.

A negative number is written with the leading sign character "-" contiguously followed by a positive number. For example, -1 -30 & -32767 are all negative numbers. If a sign character does not have a positive number contiguously following it, then it is not regarded as the sign character of a number. Thus "-" on its own is a valid syntactic object, differing from any number.

2.3 Constants

Constants are the simple unstructured objects of micro-PROLOG. They are used to name individuals in the program, such as "fred", "A1" etc., but they are also used to label programs with predicate symbols, such as "Append", "P" etc.

A constant is normally written as a alphabetical letter, followed by a sequence of letters and digits (though see definition of variable below). This is similar to the way identifiers are written in conventional programming languages. The "-" character also counts as an alphabetic character; this can be used to split up long names with several English words. Some examples of constants are:

```
A1   Fred   A-1   All-sol
```

Constants can also be written using the non alphanumeric characters such as "." "!" "," etc. This kind of constant is written as a sequence of graphical characters, which include such characters as:

```
! # $ % & ' = - ^ ~ § @ ` [ ] { } ; : , . / + * < > ?
```

Finally, a constant can also be written as a quoted string, in which case there are no restrictions on the characters that can be used in the constant. A quoted string consists of a sequence of characters surrounded by the double quote character: ". If the quote character is itself to be part of the string then it is typed twice: "". Using this string notation means that we can have constants of the form:

"1" "The man" "A$" "$100" "A ""quoted"" string"

If an ASCII control character occurs in a constant then it is normally displayed by the micro-PROLOG system as a "^char" sequence. For example the Control-A character is displayed by micro-PROLOG as [illegible] to reflect the key combination needed to enter control characters at the keyboard.

2.4 Variables

Variables are represented by alphanumeric names which consist of a single letter followed by a sequence of digits. The first character must be one of the variable prefix characters, which in standard micro PROLOG are: "x", "y", "z", "X", "Y" & "Z". So, for example, "x", "X1" and "Y30" are variable names since they consist of a variable prefix character followed by only digits, whereas "yes", "x12c" are not variable names. The number which follows the variable prefix character should be in the range 1..127, otherwise two apparently distinct names will be mapped to the same variable.

When a variable is recognised on input it is always converted to an internal form, which means that when a variable is printed the original name of the variable is not used. Instead, the variables are displayed in the sequence "X", "Y", "Z", "x", "y", "z", "X1", .., "z1", ... The set of variable prefix characters can be altered by the user to select from one of the popular variable conventions (see Appendix C).

Variables and constants are generally seperated from each other by one or more of the seperator characters: space, carriage-return/new-line, and tab. The actual number of seperators between constants and variables is not important, and they are ignored by the micro-PROLOG system.

See Section 2.10 for a fuller description of micro-PROLOG's lexical syntax.

2.5 Lists

The only function symbol recognised in micro-PROLOG is the binary function symbol "|". Function terms using this symbol are written in fully bracketed infix form.

We generally view the "|" function symbol as a list constructor in the same way that "." is used in LISP, and in other PROLOG systems. As in LISP and in DEC-10 PROLOG there is a more convenient list notation when a term names a list of items. Such a list is written as the sequence of its elements separated by spaces or carriage returns, and enclosed with brackets.

For example, the list of numbers from 1 to 5 would be written as

```
(1 2 3 4 5)
```

which is identical to the term:

```
(1|(2|(3|(4|(5|())))))
```

and both forms are acceptable as input to the system.

Note that the special characters "(", ")" and "|" need no spaces on either side of them, whatever context they are in. Of course these characters may appear in a quoted constant in which case they are treated just as if they were ordinary characters.

The empty list just consists of two brackets: "()" and it is logically interpreted as a single constant.

When a term is displayed by the **micro-PROLOG** system it is be displayed in list notation rather than in fully bracketed form.

If a term is not exactly a list, for example if the 'tail' of the term is a variable, then a bar ("|") is interposed between the last element of the list and the term naming the tail of the list. This allows partial use of the list notation, reverting to the original bar notation when necessary. For example, the list structure:

```
(A B C|x)
```

refers to the list whose first three elements are A, B and C, and whose tail is named by the variable "x". In full bar notation this term would be written as:

```
(A|(B|(C|x)))
```

Elements of lists are in general arbitrary terms, and in particular can themselves be lists. The list of unit lists of numbers from 1 to 5 is written as:

```
((1) (2) (3) (4) (5))
```

In full bar notation this is written as:

```
((1|())|((2|())|((3|())|((4|())|((5|())|())))))
```

Although the bar is the only function symbol in **micro-PROLOG** other function symbols can easily be simulated by using a prefix notation. Instead of writing "f(t1,...,tn)" write the list "(f t*1 .. t*n)" where the function symbol is named by a constant, and forms the first element of a list. The rest of the elements t*1 .. t*n of the list correspond to the arguments t1,...,tn of the function term.

In fact this method is not much less efficient in terms of space used, and time of execution, though it is perhaps less convenient to write. Of course, we can also have terms where the function symbol is no longer named by a constant but by a variable or a list structure. For example the list "(x t1 .. tn)", has a 'function symbol' which is a variable, can be interpreted as a term 'for all possible values of "x" '.

Alternatively, the function symbol may have structure:

```
((RECORD PAYROLL) employee salary)
((RECORD INVOICE) customer total)
```

All **micro-PROLOG** constructs are expressed in terms of the four types of term discussed so far: Numbers, Constants, Variables and Lists. The higher level syntactic constructs such as atoms and clauses make use of these basic objects, and they are generally list structures.

2.6 Atoms

An atom is a term written using the special prefix form described above for functions. For example, an atom which has predicate symbol P and arguments A, B, C and D would be written as the list:

```
        (P A B C D)
```

For implementation reasons the predicate symbol of an atom must be a micro-PROLOG constant or variable. In particular predicate symbols with structure, such as "((F x) A)", are not allowed.

Some atoms, in particular those which refer to certain built-in programs, can have a syntax which is slightly different from that of the normal atom. If the built-in program can execute without any arguments and it is an program implemented in assembler, then the atom can be just a constant rather than a unit list. For example, the terms:

```
        (PP)
        PP
```

are equivalent atoms if they occur in the body of a clause (where PP is a built-in program for printing terms, see Chapter 5).

This ability to use just the constant name of a relation is restricted to the class of built-in programs which are written in assembler and are normally executed without arguments. (The feature is described here for the sake of completeness: it is most heavily used by micro-PROLOG itself.)

2.7 Clauses

A clause is represented by a term which consists of a list of atoms, the first atom being the head of the clause, and the rest of the list being the body of the clause. The predicate symbol of the head atom must be a constant. For example, a simple assertion consists of a unit list of its head atom:

```
        ((Pred x y))
```

If there is a body to a clause then, unlike conventional PROLOGs, there is no implication arrow written between the head and the body. For example, the program for appending two lists together is written:

```
        ((Append () x x))
        ((Append (x|X) Y (x|Z))
             (Append X Y Z))
```

Furthermore, there are no explicit connectives like "&" or "and" between the atoms in the body.

2.8 Comments

micro-PROLOG has no built-in means of handling comments in user programs. However, one suggestion for adding comments is to have dummy clauses in the program, clauses which the programmer knows will never unify. For example:

```
        ((Pred) this is a comment)
```

The advantage of writing comments in this way is that the system 'knows' where each comment is to be kept, and it always prints in the same relative position in the program. The disadvantage is a slight slowing down in execution speed as the system tries, and fails, to use the comment clause.

2.9 Meta-variable

The reader may skip this section on a first reading of the manual.

As an extension to the clausal syntax described above we allow variables in the clause to name 'meta-level' components of the clause. A variable can be used in place of the predicate symbol of an atom in the body, it can name a whole atom in the body, or it can be used to name the 'rest' of a body. These various uses of variables in the bodies of clauses are called the 'meta-variable' facility. This is to indicate that at run-time the variables concerned will be bound to terms which name the relevant components of the clause.

The meta-variable is very important to the usability of micro-PROLOG, it enables many of the second order programs found in LISP (say) to be also available in micro-PROLOG as PROLOG programs. The meta-variable facility is still a first order logic construct however, and it does not affect the semantics of logic.

The interpreter checks a call atom for the correct form. If the form is incorrect, or if the variable is unbound for some reason, then the system reports a "Control Error" and aborts the execution.

2.9.1 Meta-variable as Predicate Symbol

A variable can be used as the predicate symbol of an atom in the body of a clause. In this case the variable must be bound to a constant when the atom is called. The constant is taken as the predicate symbol of the atom for this call, and it must have associated with it a program of one sort or another as in a normal atom; otherwise the execution aborts with a "Clause Error" or "Control Error" message.

This form of the meta-variable can be used to implement the equivalent of the MAP functions in LISP. It is also similar to the facilities to pass procedures as parameters commonly found in more conventional programming languages like Pascal, ALGOL etc. In the example program below "Apply" applies a test to each of the elements of its list argument. A call to Apply takes the form: "(Apply OK <list>)" and it succeeds if "(OK <el>)" is true of each element of "<list>".

```
((Apply x ()))
((Apply x (y|Y))
     (x y)
     (Apply  x Y))
```

2.9.2 Meta-variable as an atom

A variable can also be used to name an atom in the body of a clause. In this case the variable must be bound to a term which names an atom when the variable is 'called'. An atom is, of course, just a list whose first element is a constant (the predicate symbol) followed by the atom's arguments.

This variant of the meta-variable is used to implement some of the meta-level extensions to the language. For example, the following program 'evaluates' a list as though it were a list of atoms:

```
((Eval ()))
((Eval (x|X))
     x
     (Eval X))
```

This use of the meta-variable does not have a direct counterpart in

Pascal, it would correspond to passing an expression as a parameter to a procedure; the closest comparison is with the call-by-name mechanism of ALGOL [Naur 1962]

2.9.3 Meta-variable as the body of a clause

The final variant of the meta-variable is its use as the body of a clause. In particular it names the tail of the body of a clause. A variable in this case represents a list of procedure calls, rather than just a single call. For example, the following program encodes the disjunctive operator OR (available as a built-in program):

```
((OR x y) | x)
((OR x y) | y)
```

The use of the bar in these two clauses implies that the variables "x" & "y" name lists of atoms, and during execution they must be bound to lists of the correct format. Each list is interpreted as the body of the clause.

Again, this use of the meta-variable has a loose counterpart in conventional programming languages; in particular the closest comparison is with the label parameter passing mechanism of ALGOL 60, the replacement body is 'jumped to' rather than being called as with the meta-variable as atom.

2.10 Lexical syntax

In this section we describe in more detail the lexical syntax that micro-PROLOG uses; the section should be omitted on a first reading of the manual. The lexical syntax determines how the sequence of characters input to micro-PROLOG (either from the console or from a disk file) are grouped together into tokens.

In some sense the notion of token is a generalisation of word; in that tokens form the smallest groups of characters that can have a meaning associated with them; for example numbers, names and special symbols like "(" are all tokens. The lexical rules themselves however, do not attach meaning to tokens, they merely define what tokens are. In micro-PROLOG there are five different types of token: special tokens, numeric tokens, alpha-numeric tokens, graphic tokens, and quoted strings.

2.10.1 Token separators

The boundaries between tokens are determined by separator characters and by certain changes in token type. For example, a number token can be immediately followed by a graphic token since they are of different type, however two successive number tokens must be separated by at least one separator character. The separator characters are space, carriage return, line feed and tab. Apart from their role as token separators, separator characters are ignored on input (but see quoted strings below).

2.10.2 Special tokens

Special tokens consist of single characters, called special characters. They therefore need no particular consideration as to token boundaries: they can be grouped together with no intervening separators. The special tokens recognised are:

```
( ) |
```

2.10.3 Alpha-numeric tokens

Alpha-numeric tokens are defined in a similar way to identifiers in normal programming languages. They consist of a letter (lower or upper case letter) followed by a sequence of letters, digits. The sign character can also be used in alpha-numeric tokens to aid readability. Some example alpha-numeric tokens are:

```
A  A1  x  A1b3fred  All-sol  A-1  -A
```

2.10.4 Number tokens

Numeric tokens are tokens which name integers. They consist of sequences of digits with possible a leading negative sign character. Some example number tokens are:

```
0 1 -3 100345678 -9009
```

Note that if a alpha-numeric token occurs immediately to the left of a number token then they must be separated by at least one separator character, but an alpha-numeric token can occur immediately to the right of a number.

2.10.5 Graphic tokens

Graphic tokens are names which are built up from the non alpha-numeric (and non special) characters. The sign character can also appear in graphic tokens. This includes such characters as "<" "=" "%" etc. Some example graphic tokens are:

```
<-  =  <  '-%  &
```

2.10.6 Quoted strings

The final kind of token is the quoted string. This is used when the lexical roles of characters need to be ignored; it allows arbitrary characters to be grouped together as a single token. The quoted string is defined as a quote character (") followed by an arbitrary sequence of characters (excepting the quote character itself) and terminated by another quote character. The quote character can itself be represented in the string by doubling it: by writing two quote characters in succession. Thus the string:

```
"A quoted """ string"
```

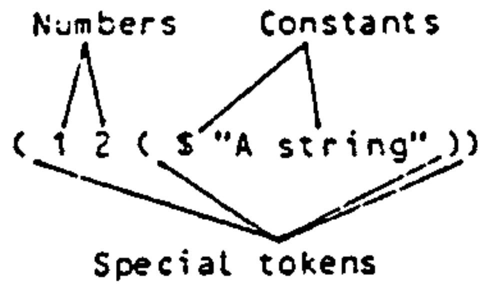
has the text:

```
A quoted " string
```

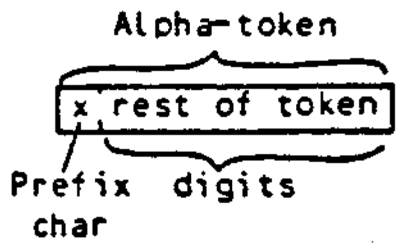
2.10.7 The lexical rules

The parser in micro-PROLOG has the fairly simple task of parsing the list syntax described above. It has merely to recognise the list construct, and to distinguish between variables, numbers and constants.

There is a fairly close correspondence between the lexical types recognised and the distinctions the parser needs to make: numbers are made from numeric tokens, graphic tokens and quoted strings form constants.



The remaining kind of token: the alpha-numeric token is recognised either as a variable or as a constant by the parser. micro-PROLOG recognises variables be examining the first character of alpha-numeric tokens (called the prefix character). If this character is a variable prefix character and the rest of the token is made up of digits then the token is read as a variable, otherwise it is taken to be a constant:



In standard micro-PROLOG the variable prefix characters are "x", "y", "z", "X", "Y" and "Z". Appendix D gives details of how the set of variable prefix characters can be modified so that the various popular variable conventions can be implemented: lower case/upper case variables, variables prefixed by the "*" character and so on.

Chapter 3

Interacting with the micro-PROLOG system

The micro-PROLOG supervisor is a PROLOG program which provides a simple operating environment for the user. It allows programs to be entered, executed, edited, saved and loaded on disk files. The supervisor also provides some extensions to the language, in the form of built-in programs which would otherwise have to be programmed by the user. In this chapter we describe the user interface to the supervisor.

The standard micro-PROLOG disk contains the following files:

```
PROLOG.COM
TRACE.LOG
SIMPLE.LOG
EDIT.LOG
```

The file "PROLOG.COM" is the micro-PROLOG system itself, and the other files are actual micro-PROLOG programs. To start micro-PROLOG you should

1. Insert the micro-PROLOG disk (or a disk with micro-PROLOG on it) into a suitable disk drive (the B drive say).

2. Log in to the disk (not strictly necessary but it is easier)

3. Type the command "PROLOG".

For example, if your computer has two disks: the "A" disk and the "B" disk, to execute micro-PROLOG from the "B" disk type:

```
A> b:
B> prolog
```

When micro-PROLOG is started up the following banner should appear at the console:

```
Micro-PROLOG r.vv  S/N *****
(C) 1981 Logic Programming Associates Ltd.
99999 Bytes Free
&.
```

The first two lines form the micro-PROLOG banner, and give details of the release (r) and version (vv) number. The message "99999 BYTES FREE" indicates how much memory is allocated for work space. The allocation is divided into two fixed areas: approximately 12% of the available memory is allocated to the storage of text for the dictionary (where the names of constants in the dictionary are stored), and the rest forms the heap and stack space. This latter region is where the user programs are stored, and where evaluation of programs takes place.

The last line which starts with a "&." shows the system level prompt which is output by the supervisor. It indicates that the system is waiting for input from the console keyboard.

micro-PROLOG can also be invoked with an initial command line. Any characters that are typed on the command line after the word "prolog" are taken to be the initial input to the system (i.e. without waiting for the prompt). For example,

```
     B> prolog load file
```

is equivalent to:

```
     B> prolog
          .
          .
     &. LOAD FILE
```

CP/M automatically converts characters appearing in the command line to upper case before passing them on to micro-PROLOG.

3.1 Keyboard control

Older versions of micro-PROLOG used the line edit facilities built into CP/M; the main problem with this is that those facilities are not especially convenient or powerful. The line editor now built into Micro-PROLOG is better; it is based on the line editor in Micro-SOFT BASIC (V.5); though with certain simplifications.

When reading from the keyboard the system prompts the user for input with a "." prompt (this is part of the "&." prompt that micro-PROLOG gives at the top level). Any characters typed in are stored in an internal buffer and are only 'read' by micro-PROLOG after the carriage return is pressed. This corresponds to the 'input mode' of the line editor. The following control keys have special significance in the input mode:

<Backspace> or <Rubout>	will delete the last character typed in
<Return>	exits the entry mode and process input
<Escape>	echos a "$" and enters edit mode (see below)
<Control-P>	toggles the print device (as in CP/M)
<Control-Q>	quotes the next key (ignore key function)

Certain other control keys provided by CP/M are not supported by this line editor; these include:

<Control-R>	Review the line
<Control-X>	Cancel input
<Control-U>	Same

Edit mode

In the edit mode of the line editor edit commands are entered using single letters. These letters can be in either upper or lower case and are never echoed to the screen. The edit commands provide fairly simple character level editing functions such as cursor movement, replacing, searching etc.

In the descriptions of the commands below we shall talk about a 'cursor'. This is similar in principle to the cursor on a screen, except that since the line editor is one dimensional the cursor can only move to the left or to the right. The cursor can only be 'over' an existing character in the keyboard buffer. Any attempt to move it outside existing text will cause the bell to be sounded on the terminal.

Similarly, if a character is typed as an edit command which is not recognised, or is illegal for some reason the bell is sounded on the console, and the command ignored. The edit commands are summarised as follows:

i	Insert mode (start accepting characters)

<Return>	Echoes the rest of the input buffer and exits the editor and allows micro-PROLOG to process the edited line.
<space>	Move 'cursor' one character to the right. The character is echoed to the screen. If already at the end of the line then the bell is sounded instead.
<backspace> or <Rubout>	Move the 'cursor' left one character. A backspace is echoed to the screen. If already at the start of the line then the bell is sounded. Note that unlike in input mode the backspace does not delete the char under the cursor.
s <char>	Searches the keyboard buffer from the current position for the <char>. The characters between the cursor and the target are displayed on the screen. If the <char> is not found then the bell is sounded and the cursor is left at the end of the line.
c <char>	Replaces the character under the cursor with <char>
d	Deletes the character under the cursor. Characters which are deleted are enclosed in "/"s.
k <char>	Similar to search, except that the characters between the cursor and the target are deleted. As with delete, the deleted characters are enclosed by "/"s.
l	Lists the rest of the line and positions the cursor at the beginning of the line.
p	Toggles the print mode; analogous to the Control-P key when in insert mode. (A "p" will toggle a <Control-P> typed in insert mode).
x	This is used to extend the line. The rest of the line is displayed and insert mode is entered.
z	This cancels the rest of the line from the cursor position and enters input mode. Useful when retyping a whole line.

Any number of terms can be typed on a line, and a term can be spread over many lines. Any excess terms (terms are read one at a time) are saved in the buffer until the next 'read console' is executed, in which case the buffer is read without disturbing the user. Constants and variable names must not be split across lines.

When entering a term, which is spread over more than one line, each successive line is prompted with "n.", where n is the number of unmatched left brackets typed so far. For example we might have the following typed in:

```
&.((
2.App(
3.)x x
2.))
&.
```

3.2 Supervisor commands

From the user's point of view the supervisor consists of a set of commands. Commands are typed in when the supervisor has issued a "&." prompt. Usually a given command can also be accessed by the appropriate procedure call. Furthermore, the supervisor allows you to define and invoke your own commands. In this section we describe the various commands available. The general format of supervisor commands is:

```
&.<Command verb><Command Data>
```

where the "command verb" is a constant, and the command data is a term whose exact form is dependent on the actual command.

3.2.1 Entering clauses

The simplest supervisor command consists of just a clause. The clause is added to the program at the end of the set of clauses defining the appropriate relation. For example:

```
&.((Parent  Mary  John))
&.((Parent  Peter  John))
&.((App()x  x))
&.((App(x|X)Y(x|Z))
1.(App  X  Y  Z))
&.
```

3.2.2 Listing the program

The PROLOG program currently in the workspace can be listed at the console with the "LIST" command. When a program is listed it is displayed in an indented format to aid readability of the program.

There are two variants of the LIST command:

```
LIST ALL
```

lists the entire program, whereas

```
LIST (pred1 pred2 .. predn)
```

lists the programs pred1, pred2,.. ,predn:

```
&.LIST ALL
((App () X X))
((App (X|Y) Z (X|x))
     (App Y Z x))
((Parent Mary John))
((Parent Peter John))
```

```
&.LIST (Parent)
((Parent Mary John))
((Parent Peter John))
&.
```

3.2.3 Executing programs

micro-PROLOG programs are executed by using the run command "?". This takes as argument a goal statement. A goal statement has the same form as the body of a clause, it consists of a list of goal atoms; the run command executes each of the atoms in the goal in turn.

If the evaluation is completed successfully then the supervisor displays its normal prompt:

```
&.?((Parent  x1  x))
&.
```

otherwise if the evaluation fails then a "?" is displayed before the next prompt:

```
&.?((Parent x1 x2)(Parent x2 x1))
?
&.
```

micro-PROLOG does not automatically print any response if the evaluation of the goal succeeds. If output is needed then it has to be explicitly programmed into the goal, for example to find a Parent of John, and print the name we could use the goal statement:

```
&.?((Parent x John)(PP The parent of John is x))
The parent of John is Mary
&.
```

3.2.3.1 Commands in the Supervisor

The supervisor also provides an alternative way of invoking certain kinds of evaluation. Namely, in the special case where the goal is a single atom, and that atom is unary (i.e. only one argument) then the goal can be executed by mentioning the predicate symbol and then its argument; this is without any extra parentheses and question marks:

```
&.<predicate> <argument>
```

This allows you to define new 'commands' to the system. For example suppose that we had the clause:

```
((Exprint  x)  (? x)(PP  x))
```

By typing:

```
&.Exprint  (goal1 goal2 ..  goaln)
```

the Exprint program is invoked as though it was called with

```
((Exprint  (goal1  goal2  ..  goaln))
```

It executes the list of goals and then prints the list in its successful form, i.e. with its variables replaced by their answer bindings.

The "?" command that we saw above is itself defined by the clause:

```
((? X)|X)
```

This clause uses the meta-variable feature where the body of the clause is replaced by the value of the variable "X". Other supervisor commands are themselves just micro-PROLOG programs that are invoked using this facility.

3.2.3.2 Controlling Execution

To interrupt the execution of a goal two interrupt keys are provided. If the Control-S key is pressed, then execution is suspended. Execution continues as soon as Control-S is typed again.

To break into an execution the Control-C key is used. When a break is detected the message

```
BREAK!
```

is displayed at the console, the current execution is stopped and the system returns to the supervisor. These interrupt conventions are in line with those for CP/M as a whole.

When an evaluation is aborted, for whatever reason (see Appendix A for possible abort conditions) the reason for the abort and the procedure call currently being executed are displayed.

3.2.4 Tracing Execution

A trace package is provided with the system (as a separate file). The trace program allows, interactively, selective tracing of the execution of a goal.

Before invoking the trace facility it is necessary to load the trace package (see section below on loading programs). Then instead of the normal "?" for executing goals, use: "??" to trace a goal. The format of a traced goal is:

```
&.??((goal1)..(goaln))
```

and each of the subgoals are traced and executed in turn.

When entering a subgoal for the first time the message

```
ENTER (pred <seq. of args of call>).
```

is displayed at the console. At this point one of the trace commands described below may be entered. The term printed represents the procedure call just before any attempt at evaluation, with all of the known values of variables substituted in place.

If the predicate symbol of the call refers to a built-in program then the call is immediately executed. (This means that it is only possible to trace user programs.) If the call is for a user program then the trace package reads a trace command from the console.

The trace commands allow selective tracing of the program. For example, low level (or already debugged) programs can be skipped: i.e. executed without tracing. The allowed trace commands are:

Continue, Skip, FINISH and FAIL

1. Skip If the Skip command is typed, or if the procedure call is of a built-in relation, the sub-goal is executed without tracing. In this case one of two things normally happen: either the sub-

computation _fails_ or it _succeeds_. If the sub-computation fails then the message

```
FAIL (pred <seq. of arguments of call>)
```

is displayed at the console, and the system backtracks. Of course this may cause calls that previously succeeded to fail, in which case more than one FAIL message may appear.

If the sub-computation succeeds then the message

```
FINISH (pred <seq. of arguments of call>)
```

is displayed. In this case the procedure call is printed with the answer bindings substituted, so that you can see the result of the sub-computation invoked. If the call was the last in the body of a clause then the calling computation is also succeeded, in which case a "FINISH" message is displayed for it too, and so on up to the goal.

After finishing a sub-computation, any uncompleted sub-goals are entered and traced in turn, until the top-level goal ultimately fails or succeeds.

2. _Continue_ The _C_ontinue command allows the trace to be continued inside the newly entered sub-computation. When this command is used the trace program looks for a clause to match the call, and when it finds one it traces each of the atoms (if any) that occur in the body of the clause selected. If no clause unifies, or if backtracking causes failure of the call the _fail_ message for the call is displayed as described above.

   If the clause that unified with the call was an assertion, then since there are no (more) atoms to trace inside the sub-computation, the call is succeeded and a "FINISH" message displayed. Otherwise each of the atoms in the body of the clause are entered and traced in turn.

3. _Finish_ The _FINISH_ trace command allows one to arbitrarily succeed a call. This command is most useful when developing programs top down, in which case the low level programs can be simulated during a traced execution without causing the "CLAUSE ERROR" message.

4. _Fail_ The _FAIL_ trace command allows the user to arbitrarily fail the call, and cause the system to backtrack. The fail command causes the "FAIL" message to be displayed before the system backtracks. Like the FINISH command, FAIL is most useful when developing programs, where it can fail a particular call without causing an abort message.

The trace program insists that a legal trace command be typed. If an erroneous command is input the program displays the message:

```
ENTER S C FINISH OR FAIL
```

and prompts the user again.

Since the trace package is itself written in micro-PROLOG it would be possible to implement a more sophisticated version of this simple program.

3.2.4.1 Warning

The use of the trace program greatly increases space demands on the workspace, thus programs which run without tracing may well run-out of space when traced.

3.2.5 Loading and saving programs from Disk

The supervisor allows the user program to be saved onto a disk file, and subsequently loaded back into the workspace. The two commands "SAVE" and "LOAD" respectively save and load the user's programs.

The formats of the load and save commands are:

```
LOAD <file-name>
SAVE <file-name>
```

3.2.5.1 File specification

The file name is a constant which describes a file in the CP/M style. The general format of a CP/M file descriptor is:

```
<Drive Letter>:<File name>.<File type>
```

The drive letter and file type are optional, in which case the colon and dot (respectively) are ommited. micro-PROLOG uses the file type "LOG" as the default file type if one is not given, and it uses the current 'logged-in' disk drive if a drive is not specifically specified. Since the colon and the dot are not alphabetic characters a file name using them must be written inside string quotes. Some example constants describing files are:

```
"A:TRACE.LOG"
TRACE
"TRACE.LOG"
"OTHER.ASM"
"B:OTHER"
```

The "LOAD" command reads a program from the file specified and adds the program into the user's workspace as if it were typed in. Any program already in the system is not disturbed in any way by the load: the new clauses are added to the end of any existing relations. In this way the programmer can have a library of programs, on a number of different files and load from them when building up a new program.

The "SAVE" command saves the program currently in the workspace into the named file. The entire workspace is saved, this may include such extra programs as the trace package, if it had been loaded. If the program is saved onto an already existing file, then the old file is renamed with file extension ".BAK". This automatically ensures that back-up copies of files are created.

3.2.6 Exiting the system

To leave the micro-PROLOG system use the command:

```
QT.                 {The "." is arbitrary - it can be anything}
A>
```

3.3 The micro-PROLOG Editor

The micro-PROLOG editor allows the PROLOG programmer to edit programs within the micro-PROLOG environment. It is a context editor which takes into account the list structure of micro-PROLOG clauses and terms. Since the editor is itself written in micro-PROLOG it is easy to extend and modify, should the need arise. The source of the editor program is given in Appendix E.

The editor's context consists of a current term, and the immediate sub-term that the current term is in. To act as an 'aide de memoire' the editor uses the current term to form its prompt when the editor is ready to accept a command. At the top-most level of editing a program (where the current term is one of the clauses of the program) the editor prefixes the prompt with a number; this number indicating the index, within the program, of the current clause. If at any time the current term 'pointer' is not a term or clause in the program then the editor displays "No term" or "No clause" as its prompt.

3.3.1 Edit Commands

Before using the editor it is necessary to LOAD it in to the workspace using the "LOAD" command. In the standard micro-PROLOG system the editor is in the file "EDIT.LOG"; so to LOAD it in type:

```
&.LOAD EDIT
```

The editor is invoked using the command "Edit program", for example to edit the "likes" program type:

```
&.Edit likes
```

The editor uses the first clause in the program (if the program is non-empty) as the initial current term pointer. In the case of the likes program this could be:

```
[1]((likes John Mary)).
```

When the editor displays its prompt it is ready to accept an edit command, which at the top level can be any of the insert, append, kill, next, back, enter and out commands. The edit commands are divided into two groups: those which move the current term pointer of the structure of the terms being edited, and those which change the terms in some way.

3.3.2 'Cursor Movement' Commands

There are four commands which can be used to walk over the program; these are next, back, enter and out.

1. The n (next) command changes the current term to the next term to the right in the immediate context. At the top level this means move to the next clause. So, for example, if the current term is "(A B)", and the immediate context is "(C (A B)(D))" then the "n" command moves the current term to "(D)":

```
(A B).n
(D).
```

If the current term was already at the last term in the immediate context, or if it was the last clause in the program, the pointer is stepped on, but the current term is "No term", (or "No clause" at the top level) indicating that it is not actually pointing to a term. It is impossible to step beyond this point.

2. The b (back) command is the inverse of the n command, it is used to step back to the term to the left of the current term in the immediate context, or to step to the previous clause. To undo the effect of the previous "n" command above:

```
(D).b
(A B).
```

If the current term were already the first term, then the b command steps back to in front of it, again causing the prompt to become "No term" ("No clause"). e.g.

```
(A B).b
C.b
No term.
```

It is not possible to move before this point.

3. The e (enter) command steps 'into' a term or clause so as to edit its components. The term being stepped into must be a list structure (there being no concept of the inside of a number or constant). The immediate context becomes the list just entered, and the current term is the first element (if any) of that list. So in our example if we enter the list "(A B)" we change our immediate context and point to "A":

```
(A B).e
A.
```

or

```
[1]((likes John X)(likes X Mary)).e
(likes John X).
```

4. The o (out) command is the inverse of the enter command. The current immediate context becomes the current term, and the 'old' immediate context (prior to the corresponding e command) is reestablished as the immediate context. The o command is also used to exit the editor, when at the top level:

```
(likes John X).o
[1]((likes John X)(likes X Mary)).o
Edit of likes finished
&.
```

The o command may fail if the entered term has been incorrectly changed, in particular if on returning to the outer level the predicate symbol of the head of the clause has been changed. When an edit command fails the editor responds with a "?" and re-prompts at the appropriate level.

With these four cursor control commands any sub-term of a program can be reached. In the next section we look at those edit commands that directly change the current term.

3.3.3 Edit change Commands

There are five commands which directly affect the current term; these are insert a new term, append a new term, kill the current term, substitute it by another and text edit the term.

1. The i (insert) command inserts a term before the current term. The i is followed by the term to insert as in:

```
(A B).i (F)
(F).
```

The new term just inserted becomes the new current term, the old one can be regained by stepping on to it with the n command.
At the top level the i command inserts a new clause into the program. In this case the form of the clause is checked to ensure that at least the predicate symbol of the clause is the appropriate one.

2. The a (append) command appends a new term (or clause) after the current term. Otherwise it is like the i command.

3. The k (kill) command deletes the current term from the immediate context. The previous term (or clause) to the left becomes the new current term, if there is not a previous term (or clause) then the current term becomes "No term" ("No clause"). We can delete a particular element of a list by using a sequence of cursor movement commands to move to the required term and then using the k command.
For example, to delete the third element of "(A B C D)":

```
(A B C D).e
A.n
B.n
C.k
B.o
(A B D).
```

4. The s (substitute) command replaces the current term with a new term. The argument to s is a pair:

```
(t1 t2)
```

The term t1, is unified with the current term, and then the current term is replaced by t2. The use of unification allows quite powerful pattern matching, but more importantly the specification of the replacement can make use of variables bound in this match. For example to reverse the first two elements of a list:

```
(A B C D). s((x y|z)(y x|z))
(B A C D).
```

Note. The s command is not available at the top level.

5. The t (text) command allows the current term to be changed using the line editor. It works by displaying the term on a new line and positioning the cursor underneath the first character. The edit mode commands described above are then available to modify the text of the term. Upon typing the <return> key the system reads the text back in and replaces the current term by it.
Note that variables are not handled properly by this command, in particular any variables become new variables after processing. This is not a serious deficiency when using the "t" command at the clause

level of the editor, or when only changing constants. (future versions of micro-PROLOG will fix this problem)

3.3.4 Restructuring Lists

Given the importance of lists in micro-PROLOG, it is especially important to be able to repair an arbitrarily damaged list. Where it is just a sub-term of a list that is damaged, the above commands are sufficient. However, a problem arises if some brackets have been put in the wrong places. For example, a left bracket can be easily missed as in:

```
[1] ((Prog A X) PR X Y)
```

and a right bracket could be put in too far to the left, as in

```
[2] ((Prog A X)(PR) X Y)
```

or too far to the right, as in

```
[3] ((Prog A X (PR X Y)))
```

The editor has two simple primitives which can be used to repair this kind of global damage: wrap and unwrap.

1. The w (wrap) command takes a number of terms from the immediate context and wraps them up into a list, which becomes the current term. The w command has an argument: the number of terms to wrap starting from the current term. If 0 (zero) is used then no terms are wrapped, i.e. the empty list "()" is inserted. If 1 (one) is used then the current term only is wrapped, if 2 (two) then the current plus the next term are wrapped, and so on up to the number of the remaining terms in the immediate context. For example, to wrap up the middle two elements "(A B C D)":

```
(A B C D).e
A.n
B.w 2
(B C).o
(A (B C) D).
```

2. The u (unwrap) command is the inverse of the wrap command. The current term must be a list, the effect is to remove the outer pair of brackets of the list. The first element of the list becomes the current term, and the other elements are inserted into the immediate context. To undo the effect of the wrap above we could perform the following sequence:

```
(A (B C) D).e
A.n
(B C).u
B.o
(A B C D).
```

Now we we can see how to use these two commands to repair the various terms we showed above:

a) ((Prog A X) PR X Y)

This case is quite simple, we wrap up the sub-list "PR X Y" into a single list, so that it is put into the correct form:

```
[1]((Prog A X) PR X Y).e
(Prog A X).n
PR.w 3
(PR X Y).o
[1]((Prog A X)(PR X Y)).
```

b) ((PR A X)(PR) X Y)

The second example is a little more complex, a right bracket has been inserted too far to the left. To repair this we need to unwrap the list "(PR)", and re-wrap including the missing arguments:

```
[2]((Prog ~~PR~~ A X)(PR) X Y).e
(Prog A X).n
(PR).u
PR. w 3
(PR X Y).o
[2]((FR[illegible]A X)(PR X Y)).
```

c) ((Prog A X(PR X Y)))

In this example we have first to wrap up the sub-list "Prog A X" to form an atom of the right form:

```
[3]((Prog A X (PR X Y)).e
(Prog A X (PR X Y)).e
Prog.w 3
(Prog A X).o
((Prog A X)(PR X Y)).
```

Now we have one too many pairs of brackets at this level, so we unwrap:

```
((Prog A X)(PR X Y)).u
(Prog A X).o
[3]((Prog A X)(PR X Y)).
```

This last unwrap has 'removed' the right bracket that was too far to the right.

3.3.5 Further Extension

This editor represents a first attempt at the development of a term oriented structure editor for micro-PROLOG. Further possibilities for improvement are context searching and combining commands together with a repeat count. Since the editor is itself written in micro-PROLOG these enhancements should be quite straight forward.

3.4 Pragmatic Considerations for Programmers

The principal limiting resource in micro-PROLOG is space. To help to conserve space micro-PROLOG incorporates a number of space saving features. To maximise their effect the programmer should be aware of them, so this section describes some of them and how they operate. Note that space saving does not affect the logic of the running program; it may only affect whether a program can run in the space available.

The features of micro-PROLOG which affect the space used by a program are:

1. Organization. The evaluation area in micro-PROLOG is organised as a stack and a heap. The stack contains the activation records and the variables of the execution. This grows with recursion and pops normally only on backtracking. The heap contains the values of variables, clauses and other permanent data objects.
   Periodically the stack and heap collide, at which time the heap is garbage collected. The garbage collector is actually called whenever the stack and heap grow too close to each other; the point at which this is done is automatically computed by the system depending on the available memory and the relative sizes of the stack and heap.
   The garbage collector is a 'Mark and Collect' garbage collector, which means that all the free space in the heap is collected together into a list. The heap is also 'cut down' if there is free space at the end of it. This has the effect (hopefully) of leaving a clear region of memory between the stack and heap, allowing execution to continue.
   If the garbage collector fails to find sufficient space then the evaluation aborts with the message "SPACE ERROR".
   Note that the allocation algorithm means that as memory gets tight the garbage collector gets called more and more often; this can have a dramatic effect on the performance of the sytem. Normally garbage collection takes a very short time (about 0.25 secs) and isn't a big overhead.

2. Success Popping. micro-PROLOG performs special actions in certain circumstances, when a procedure call has been deterministic. When such a call completes it is popped off the stack just like a normal recursive call in a more conventional programming language. (The record of the evaluation is not needed for backtracking purposes.)
   The alternative situation, where micro-PROLOG cannot detect that a computation is deterministic, results in the record of the evaluation being left even after the successful completion of a procedure call.
   The expert programmer can give more information about when a program is deterministic by inserting the "/" control primitive in suitable places in his program (see Chapter 5).

3. Tail recursion is the name given to that form of recursion which is actually equivalent to a loop. micro-PROLOG can detect this special case of recursion, and when a tail recursion is also deterministic then micro-PROLOG optimises the call; it does not grow the stack when entering the call. For example, if all the calls preceding the last call are deterministic then, when entering Qn in:

   ```
   ((Pred ...)(Q1 ...)  ..  (Qn ...))
   ```

   the stack is not grown at all for the first evaluation step of Qn. Initially the stack grows in the normal way, but since it is the last call and since the evaluation is deterministic then micro-PROLOG 'knows' that it will eventually be able to success pop the activation record for "Pred". Further, since there are no more references into the procedure "Pred" micro-PROLOG pops the "Pred" entry from the stack as the last call "(Qn ...)" is entered. The net effect is not to grow the stack at all for the last call in a procedure when in a completely deterministic evaluation.

A classic example of the power of tail recursion in saving space is in append. If we write the append program as:

```
((append () x x))
((append (x|X)Y(x|Z))
    (append  X Y Z))
```

then for normal calls to this append program (appending two lists together for example) the stack does not grow during execution <u>for any length of input.</u> In this way the recursive definition of append is executed as though it were written in a WHILE loop.

Often, there may be several calls in the body of a clause all of which execute deterministically. In this situation after each of the calls in the body, except for the last call, have completed then the stack is popped leaving no trace of the evaluation of the stack. For the last call the top of the stack is overwritten with the new record, hence the stack does not grow at all for the last call. This has the effect of turning a recursive evaluation of the sub-goal into a loop evaluation.

4. <u>Non-structure sharing</u>. micro-PROLOG is a so-called 'non-structure sharing' implementation. Briefly this means that when a variable is bound during unification its value is explicitly computed and placed, if necessary, in the heap.

   The effect of this, together with garbage collection, success popping and tail recursion, is to limit the amount of data currently in the work-space to that which is actually needed, though it does lead to an overall increase in memory turn-over. It also has a space benefit in that for certain simple, but common, cases the value of a variable takes actually less space than in the more normal 'structure-sharing' implementations.

To take full advantage of these space saving optimizations the programmer should try to ensure that micro-PROLOG can always detect determinism in a program. This means, for example, putting the base case of a program (such as append) before the general case, and using the conditional form where it is applicable (see Chapter 5). Programs optimised for space in this way tend to be less optimal with respect to speed of execution, and vice versa.

3.5 <u>Summary</u>

Here is a summary of the supervisor commands we have discussed:

<clause>	Add a clause.
?(<Goal sequence>)	Execute a goal.
??(<Goal sequence>)	Trace execution of a goal.
C	Continue trace.
S	Skip trace - execute subgoal without tracing.
FAIL	Arbitrarily fail subgoal.
FINISH	Arbitrarily succeed a sub-goal.
Edit <predicate symbol>	Edit a program.
n	next term/clause
b	previous term/clause
e	enter term/clause
o	exit term/editor

i t	insert new term/clause
a t	append a new term/clause
k	delete term
s (t1 t2)	unify current term with t1 and replace by t2
t	text edit the current term with line editor
w n	wrap n terms into a sub-list
u	unwrap sub-list
LOAD <file descriptor>	Load a program from the file.
SAVE " "	Save a program to the file.
QT.	Exit micro-PROLOG.
LIST ALL	List the program on the console.
LIST (P1 P2 .. Pn)	List the programs named.
^C	Abort execution of a goal
^S	Suspend execution until ^S is pressed again.
^P	Toggle printer on/off
^Q	Quote the next key pressed on the keyboard

Chapter 4

Simple PROLOG

In this chapter we look in detail at a way of extending the basic micro-PROLOG system by using a front end program. This technique allows us to write PROLOG programs using a more friendly syntax. Sentences of this *surface* syntax have a simple structure and often many fewer parentheses. The file "SIMPLE.LOG" contains a module which defines a set of commands which allow one to write and use Simple PROLOG programs rather than micro-PROLOG programs.

4.1 Syntax of Simple PROLOG sentences

A Simple PROLOG sentence (or clause) consists of either an *atomic* sentence or a *molecular* sentence. Atomic sentences are just atoms, and can have two forms:

```
     John likes Mary
     PRED(2 3 x)
```

i.e. an atom is either a binary predicate ("likes"), in which case it is written in infix form with no parentheses; or a non-binary predicate, in which case it is written as the predicate symbol ("PRED") followed by a list of arguments surrounded by parentheses and separated by spaces.

Molecular sentences consist of an atomic head followed by the word "if" followed by a conjunction of *literals* separated by "&" or "and". Literals are either atoms:

```
     x likes y if y likes x1
```
and
```
     PRED(x y z) if PQ(x y z) and y LESS z
```

or negated atoms which are written as "Not" followed by a conjunction enclosed in parentheses, as in:

```
     x likes y if Not(y likes Peter)
```
,
```
     x GE y if Not(y LESS x)
```
and
```
     PRED(x y z) if x LESS y and Not( PR(y z x) & QUALIFY(x))
```

4.2 The Simple PROLOG System

Simple PROLOG is implemented as a micro-PROLOG module that is LOADed in at the beginning of a session. It provides a small set of commands that enable you to interact with the micro-PROLOG system as though it were a Simple PROLOG system.

Thus to start a Simple PROLOG session enter micro-PROLOG using the LOAD command:

```
     A>prolog load simple
     Micro PROLOG r.vv S/N  xxxxx
     (C) 1981 Logic Programming Associates Ltd.
     99999 Bytes Free
     &&.
```

In Simple PROLOG the various functions to add new sentence etc. are invoked by a set of commands, some of which are described below. For a more detailed description of how to use this system see the micro-PROLOG Primer.

4.2.1 Add

The "Add" command allows you to add a simple PROLOG sentence into the workspace. The format of the command is:

```
&. Add (Peter likes x if Not(x likes John))
&.
```

Notice that the sentence to be Added to the program is surrounded by brackets. This makes it a single list argument to the "Add" command. A simple PROLOG command can be typed in whenever the micro-PROLOG system displays its "&." prompt.

Sentences are usually added to the end of the program for the appropriate relation. To add into the middle of a program the form

```
Add n sentence
```

is used, where the number "n" refers to where in the relation the new clause is to be added. For example, to add to the beginning of the "likes" relation use

```
&. Add 0 (Peter likes John)
&.
```

4.2.2 List

The "List" command displays the program on the console. To display the whole of your program type

```
&. List All
Peter likes John
Peter likes X if
     Not (X likes John)
&.
```

To display just a single relation, the "likes" relation say, use

```
&. List likes
```

To print a simple program on the printer use the ^P toggle function before "List"ing the program. This will automatically print the program on the printer as it is displayed on the screen.

4.2.3 Delete

"Delete" a single clause from the program. Its usage is

```
&. Delete likes 3
```

which deletes the third likes sentence.

4.2.4 Kill

"Kill" will delete an entire relation; to remove the likes relation type

```
     &. Kill likes
```

4.2.5 Does

The "Does" command makes a YES/NO query of the program. It has as argument a conjunction of literals, like the body of a molecular sentence. If the goal is successful the command responds with

```
     YES
```

otherwise it responds with

```
     NO
```

For example

```
     &. Does (John likes Mary)
     YES
     &. Does (SUM(2 3 x) & x LESS 5)
     NO
```

4.2.6 Which

The "Which" query attempts to find answers to questions. The form of the answer required is specified by the question. The syntax of this command is

```
     Which (term body)
```

where term denotes the form of the answer required, and body is the query to be evaluated. For example to list those people that like John use:

```
     &. Which (x x Likes John)
     Answer is Peter
     Answer is Mary
     No (more) answers
     &.
```

To list the pairs of people who like each other:

```
     &. Which ((x y) x likes y and y likes x)
     Answer is (Peter Mary)
     No (more) answers
```

To compute the sum of 3 and 5

```
     &. Which (x SUM(3 5 x))
     Answer is 8
     No (more) answers
```

4.2.7 One

The "One" query is similar to "Which" except that it prompts after each solution. If you respond with "C" then the next solution is sought, otherwise use "F" to finish looking for solutions:

```
     &.One (x x likes John)
```

```
Answer is Peter.C
Answer is Mary.C
No (more) answers
&.
```

4.2.8 Save

The Simple PROLOG program in the memory can be saved for later use via this command. The form of the "Save" command is

```
&. Save file-name
```

where file-name is a file name in the normal micro-PROLOG form.

4.2.9 Load

The "Load" command is used to re-load a previously Saved Simple PROLOG program. For example:

```
&. Load fred
```

4.2.10 Accept

If entering a lot of data the "Accept" command can be used as an aid in generating large relations. It enables binary atomic sentences to be added without using the "Add" command all the time. It is restricted to binary relations. The "Accept" command is used as follows:

```
&. Accept likes
likes.(John Mary)
likes.(John Peter)
    .
    .
likes.(P S Q)
What is (P S Q)?
likes.
    .
    .
likes.End
&.
```

The "Accept" command prompts for each pair with the name of the relation involved. If a pair is not entered the response is queried and you are reprompted for another pair.

4.2.11 Edit

The line editor can be used to edit an individual sentence locally by using this "Edit" command (Do not confuse with the structure editor). The Edit command is invoked as follows:

```
&.Edit likes 1    (the predicate symbol followed by the clause number)
```

The system responds with the sentence (surrounded by brackets) which can then be edited in a similar manner to the "t" command in the structure editor.

4.3 All Solutions

A special Simple PROLOG predicate "Is-All" can be used to mimic the action of "Which". Instead of displaying the answers however, it puts then in a list and returns the list of solutions. The general form of the predicate is:

```
     x Is-All (term conjunction)
```

For example in:

```
     &. Which (x x Is-All (y y likes John))
```

the query "(z z likes John)" looks exactly like a "Which" query in itself. But instead of printing the answers, they are put into a list and bound to x:

```
     Answer is (Mary Peter)
     No (more) answers
     &.
```

4.4 Summary

Simple PROLOG illustrates how easily the basic micro-PROLOG system can be augmented by the use of a front end program. In this case Simple merely adds a few commands which compile to and from the syntax of Simple- and micro-PROLOG. In Chapter 6 we look at how this is done in more detail.

Chapter 5

Built-in Programs

In some ways the character of a PROLOG system is determined more by the built-in programs than by any other single factor. The selection provided, their flexibility and efficiency are all key factors determining the final usability of the system. In micro-PROLOG this problem is made worse by the severe space constraints on a micro-computer.

A special feature of the built-in programs in micro-PROLOG is that they model as closely as possible normal relations. For example the SUM relation can be viewed as a collection of addition sums, and the PROD relation consists of the various 'times tables'. Consequently the built-in programs must attempt to simulate all the various possible patterns of use of the relation; and the SUM built-in program must be able not only to add up numbers, but also to subtract them.

For reasons relating to efficient implementation micro-PROLOG compromises to some extent and generally allows some of the uses of its built-in programs but not all. In particular the assembler coded built-in programs only implement the deterministic uses of the relations they represent.

So, in general each built-in program may have several uses. This helps to mimimise the number of names the programmer has to know, and also helps to keep micro-PROLCG programs 'reversible'.

If a particular call to a built-in program has an illegal use (for example if SUM is called with two or more arguments as variables) then the system reports a "Control Error" and aborts the execution. An error of this kind usually occurs only if there are too many variables in the call.

The 35 or so built-in programs are divided into a number of functional groups: the arithmetic operations, string operations, input/output operations, type predicates, data base operations, logical operators, module construction facilities, program library operations and miscellaneous programs. We take each group in turn and describe the formats and semantics of each built-in program.

5.1 Arithmetic Relations

The three arithmetic relations SUM, PROD, and LESS cater for the normal operations on integers of addition, subtraction, multiplication, division and comparison.

5.1.1 SUM

```
(SUM x y z)            "x+y=z"
```

When used with numeric arguments the SUM program can:

1. Check a sum. If all arguments are numbers then SUM succeeds only if the first two numbers add up to the third. (16 bit arithmetic is used, with an overflow error trap.) For example, (SUM 20 30 50) succeeds.

2. Add two numbers together. If the first two arguments are numbers and the third a variable then the call succeeds by binding the third argument to the sum of the first two. For example, (SUM 30 -2 x) binds "x" to 28.

3. Subtract two numbers. If the third argument is a number, and either the first or the second also a number (with the remaining argument a variable) then the call succeeds by binding the variable in the call to the result of subtracting the first number (or second) from the third. For example, (SUM x 3 15) binds "x" to 12, as does (SUM 3 x 15).

If an addition or subtraction results in an overflow, then micro-PROLOG reports an "Overflow" error, and aborts the current evaluation.

5.1.2 PROD

```
(PROD  x y z {u})   "x * y {+u} = z, u is optional"
```

The PROD program implements multiplication and division. The allowed uses are:

1. Check a product. If PROD is called with three arguments, all of which are numbers, then the product of the first two numbers is checked against the third number. If they are the same then the call succeeds, otherwise it fails. For example, (PROD 3 4 12) succeeds.

2. Multiplication. If PROD is called with just the first two arguments known (numbers), and the third argument a variable, the call succeeds by binding the variable to the product of the two numbers. For example, (PROD 3 -4 x) results in "x" being bound to -12.

3. Division. There are two forms of the PROD program which can be used for division. For so-called perfect division where the divisor divides exactly into the dividend the three argument form is used:

   ```
   (PROD x 10 30),
   (PROD 10 x 30)
   ```

   In this form the PROD call only succeeds if the division is perfect, in which case the variable is bound to the quotient.
   In the second form the PROD program has four arguments, the fourth argument forms the remainder of the division. For example, (PROD 3 x 17 y) results in "x" being bound to 5 and "y" to 2, as does (PROD x 3 17 y).

5.1.3 LESS

```
(LESS x y)          "x is less than y"
```

The LESS built-in predicate implements the inequality test for numbers. Only one arithmetic usage is allowed, where both arguments are numbers. In this case the call succeeds if the first number is numerically less than the second; if they are equal or if the first number is greater than the second the call fails. For example, (LESS 3 2) ~~succeeds.~~ fails.

5.2 String operations

In micro-PROLOG strings are represented by constants. There are two built-in programs that manipulate the names of constants. The LESS predicate performs a textual comparison of two constants' names and STRING is used to transform lists of characters to constants and vice versa.

5.2.1 LESS

```
(LESS x y)            "x lexicographically less than y"
```

Like the inequality test for numbers, this test for constants tests that the first argument (which is a constant) is textually less than the second (which must also be a constant). The ordering used is the lexicographical ordering, based on the ordering of the underlying character set (namely ASCII), and compares the names of the constants.
For example, (LESS FRED FREDDY) succeeds since "FRED" is lexically less than "FREDDY".

5.2.2 STRING

```
(STRING <list> <constant>)      the <list> of letters forms <constant>
```

The STRING built-in program enables the programmer to take apart a constant into its constituent characters, and vice versa to pack a list of characters into a single constant. There are essentially two uses of the program:

1. Unpacking: to produce a list of characters from a constant. In this use the second argument should be bound to a constant (not a number or list), and the second unbound. The effect of the program is to bind the first argument to a list; if the empty constant "" is used then the result is the empty list. So, for example, "(STRING x fred)" results in "x" being bound to "(f r e d)", and "(STRING x "A*")" binds "x" to "(A *)".
   The first argument may be partially instantiated; this would allow some comparison of the list of characters and the constant, as well as being able to 'pick off' some specific characters of the constant. In the case where the two arguments are fully gound a check is performed to see that the list does name the constant. Some examples of this use are:

   ```
   (STRING (f r x d) fred) x = e
   (STRING (f r|x) fred)   x = (e d)
   (STRING (f r|x) gerry)  fails
   ```

   The list of characters must be just that: a list of constants which have single character names.

2. Packing. This takes a list of letters and produces a constant from it. It is the inverse of the unpack use of STRING. Some example uses:

   ```
   (STRING (f r e d) x)     x = fred
   (STRING () x)            x = ""
   ```

5.3 Console Input/Output Operations

The input/output facilities are divided into two groups: Console I/O and Disk I/O. Console I/O refers to terms read from the keyboard and displayed on the console, and disk I/O transfers terms to and from the disk system. The I/O facilities described here are the first example of a non-logical feature of micro-PROLOG, this is because they depend on their behaviour (reading and writing terms) for their meaning.

There are four built-in programs for dealing with Console I/O: Read, Print, PrettyPrint term and RFILL, which respectively read a term from the console, print a list of terms, pretty print a list of terms and 'pre-fill' the keyboard buffer with a list of terms.

5.3.1 R

```
(R x)     "read a term from the keyboard and bind to x"
```

The Read program reads a single from the keyboard and binds its argument to the term it reads in. It must be called with a variable as its argument, otherwise a "Control Error" is reported.

Any variables typed in appear as variables in the term, though they are completely new variables differing from any others that may appear in the program.

See Section 3.1 for details of how **micro-PROLOG** accepts terms from the keyboard.

5.3.2 P

(P <seq. of terms>) "print the **<seq. of terms>** on the console"

The Print program prints its arguments on the console output device. Each character in the terms of its arguments are printed as they are, rather than being specially 'exhibited'. There is no implicit new line after the print operation has completed: though it can of course be programmed by printing control characters.

For example, if the console uses Control-L to clear the screen then

```
(P ^L)
```

clears it. Any variables occurring in the list of terms are displayed as "X", "Y", "Z",...,"z","X1",...,"z1","X2" and so on, corresponding to the order that the variables are encountered during the Print. Variables appearing in more than one term will have the same print name. Note that if there are more than 128 different variables, then subsequent variables are displayed as "???".

5.3.3 PP

(PP <seq. of terms>) "Pretty Print **<seq. of terms>**"

The PP program displays its arguments in a pretty printed format. This is probably the best way to view the structure of terms constructed by the your program. It is used for example by the LIST program when displaying clauses.

The PP program uses a rudimentary algorithm for distributing terms accross lines. Each time a term of odd nesting depth is encountered it is put onto a new line with some indentation which depends on the depth. For example the term "((P (t))(Q (t1 t2)))" is displayed as:

```
((P (t))
   (Q (t1
        t2)))
```

If a term that happens to name a clause is displayed in this manner, it has the effect of displaying the head of the clause on a single line and each atom in the body on a separate line and slightly indented, exhibiting the structure of the clause.

Any control characters occurring in terms that are pretty printed are displayed in the "^<char>" format. If a quoted string type of constant is pretty printed then it is displayed in its quoted form, with two quote characters on either side. For example the call:

```
(P "(The man")
```

displays:

```
(The man
```

on the console, whereas the call:

```
(PP "(The man")
```

displays:

```
"(The man"
```

on the console. The pretty print operation is always completed by a carriage return/line feed, so a new line on the screen is performed by: "(PP)" or "PP".

5.3.4 RFILL

(RFILL <seq. of terms>)   "Fill keyboard buffer with <seq. of terms>.

The RFILL program is used to 'pre-fill' the keyboard buffer prior to keyboard entry. It is similar to the "PP" built-in program, in that it takes a list of terms as arguments and 'writes' these terms into the keyboard buffer. This program is very useful to implement editors in micro-PROLOG as in the structure editor and in the Simple front end.

For example, to pre-fill the keyboard buffer with a sentence such as "John likes Mary" we would call: "(RFILL (John likes Mary))".

When a subsequent read "(R x)" is executed it is as though this term had already been typed in at the keyboard. The "R" program detects that a previous "RFILL" has been executed, and instead of starting the input with the input mode as is normal, it starts in edit mode. The contents of the buffer are also displayed. In general it is as though the editor were primed with a "l" command, (as opposed to the "i" command it is normally primed with).

RFILL also has the side effect of clearing any previous contents of the keyboard buffer. If several terms had been typed on a line then any 'unused' terms would be ignored.

5.4 Disk I/O

micro-PROLOG supports files of text under CP/M, i.e. files of ASCII characters. These are accessed sequentially via the built-in programs READ, W and WRITE or randonmly via the SEEK program. Up to four files may be active during an evaluation at any one time, any attempt to have more results in the "Too many files opened" message.

5.4.1 OPEN

(OPEN file-name)   "open file for reading"

This built-in program opens a file for reading. The "file-name" is a constant which names a file according to the CP/M file naming conventions (see Section 3.2.5.1 for details).

If the file was previously open for writing then the file is first 'flushed' and closed down before starting the read. This means that it is not possible to simultanously read and write to a file. The first character of the file is read in by the OPEN program, so if the file is empty or if it is not there then the message: "File not found" is printed, and the OPEN call fails.

If the file was already open for reading then the file is 'rewound' to the beginning by the OPEN program.

5.4.2 CREATE

(CREATE file-name) "create a new file for writing"

The CREATE program opens a new file for writing. Any old file of the same name is first re-named to have extension ".BAK", and the new file is then created. This means that files are automatically backed up.

If the file is already open for writing then it is merely rewound to the beginning.

5.4.3 CLOSE

(CLOSE file-name) "close down the file"

In CP/M there are no special actions associated with closing a file which is being read only; however a file which is being written to must be explicitly closed down, otherwise the disk file may not contain the right data. This CLOSE program performs this operation and releases the file from micro-PROLOG. It is also used to close down files opened for read access.

5.4.4 READ

(READ file-name x) "read a term from <file-name> into x"

The READ program reads a single term from the named file and binds its argument to the term. If the call reads past the end-of-file then the READ call fails.

5.4.5 WRITE

(WRITE file-name <seq. of terms>) "write <seq. of terms> to the file"

The WRITE program writes the terms in the <seq. of terms> into the file, in the same form as the PP program. This ensures that any term written onto a disk file can be subsequently read back in as the same term, with the exception of variables which are renamed.

5.4.6 W

(W file-name <seq. of terms>) "write <seq. of terms> to the file"

The W program is the same as the WRITE program except that the terms are written in the same way that "P" displays terms on the console. Note that terms written using "W" may not be re-readable as terms.

5.4.7 SEEK

(SEEK file-name pos) "file is at pos"

The SEEK program identifies 'where' in a file the program is currently pointing. There are two modes, either the 'pos' argument is unbound on entry to SEEK, or it is a number. The 'pos' number is in terms of the number of characters from the start of the file.

If the position argument is unbound then SEEK returns the current position in the file; if the argument is given then the file is positioned to the position given.

Extreme care should be exercised if writing into the middle of a file, as there is no protection against overwriting already existing terms on the file.

SEEK can be used to implement a system where one or more programs can be on disk instead of in memory. The following program will do this automatically:

```
((RPRED x y z)          {file x contains clauses of the form (y|Y) }
     (SEEK x z)         {starting from z}
     (READ x y1)        {read in candidate clause}
     (SEEK x z1)        {where are we now?}
     (OR                {either its the right clause:}
       ((EQ (y|Y) y1)|Y) {and execute the body of clause just read}
       ((RPRED x y z1)))){or continue down the x file until y is found}
```

To use this program (ignore the bits between the curly brackets) to keep the clauses for likes (say) on the disk file LIKES (say) replace the clauses for "likes" in the program with the single clause:

```
((likes|x)
     (RPRED LIKES (likes|x) 0))
```

The file LIKES must also have been OPENed prior to using the program for "likes". The LIKES file can be generated either by using SAVE (see below) or specially written by another program. The likes clauses in "LIKES" can be general (including recursive); and that there can be more than one file with the "likes" clauses on it (just use more than one rule as above).

5.5 Type Predicates

The type predicates test a single argument for their type: whether it is numeric, constant etc.

5.5.1 NUM

(NUM n) "n is a number"

The NUM built-in predicate tests to see if its single argument is numeric or not. If it a number the call succeeds, if the argument is a variable then the evaluation aborts with a "Control error", otherwise the call fails. For example, (NUM 3) succeeds.

5.5.2 CON

(CON c) "c is a constant"

The CON built-in predicate test to see if its single argument is a constant. For example, (CON CON) succeeds, whereas (CON ()), and (CON 1) both fail. If the argument is a variable then the evaluation aborts with a control error.

5.5.3 SYS

(SYS t) "t names an atom which is built-in"

SYS tests to see if its argument is a built-in program or not. It succeeds if it is, fails if it is a user defined program or if there is no such program.

5.5.4 VAR

(VAR x) "at the time of the call x is a variable"

Strictly non-logical, the VAR built-in type predicate checks to see if its argument is currently a variable. (It is non-logical because a successful call is invalidated if the variable is subsequently bound.)

5.6 Logical operators

The basic clausal form of logic programs is extended to include some other connectives via the logical operator built-in programs. These programs implement disjunction (OR), negation-as-failure (NOT) conditionals (IF) and identity (EQ). These can be used to increase the efficiency and (sometimes) the readability of **micro-PROLOG** programs.

5.6.1 OR

(OR goal1 goal2) "either goal1 or goal2 is true"

The disjunctive operator OR has two arguments: each of which is a list of atoms. The OR program succeeds by succeeding either of the two goals in the call. For example, (OR ((SUM 3 2 x)(P x OK))((P not ok))) succeeds in the first branch, and prints "5 OK" on the console.

An empty goal (named by the empty list "()") always succeeds, and hence if used as an argument to OR acts as a 'true' branch.

5.6.2 NOT

(NOT pred arguments)

The NOT operator implements negation-as-failure [Clark 1978]. A negated atom has the form "(NOT pred args)" where "(pred args)" is the corresponding un-negated atom.

Negation-as-failure is not logical negation, but corresponds to the rule:

> If a goal is unprovable (i.e. any attempt to prove it ends in finite failure) then assume that the goal is false.

For a great number of cases (especially when using negation to test a condition) negation-as-failure is adequate, and coincides with the classical concept of negation.

The negated atom should contain no variables, otherwise the NOT program may behave incorrectly. In particular if the atom succeeds by binding a variable in it then the NOT program fails, whereas the system should abort, since logically the negated atom can neither be false nor true. A simple example illustrates the difficulties that can arise: given the two clauses:

```
        ((Not-test-A B))
        ((Not-test-B A))
```

and the goal statement:

```
        ?((NOT Not-test-A  x)(Not-test-B  x))
```

This goal logically should succeed, but with our NOT it fails because the NOT call fails. However the equivalent goal:

```
        ?((Not-test-B  x)(NOT Not-test-A  x))
```

succeeds, with "x" equal to "A". A logic program should not be this sensitive to ordering of sub-goals.

5.6.3 IF

```
    (IF atom goalA goalB)  "If atom is provable then IF is
                                    provable if goalA is
                                   else it is
                                    provable if goalB is"
```

The IF program implements the conditional form. It is primarily used for efficiency because in practice the excessive use of conditionals hampers readability. A statement of the form:

```
        ((A ...)(IF (C ...) (bodyA) (bodyB))
```

is (almost) equivalent to the pair of clauses:

```
        ((A ...) (C ...) bodyA)
        ((A ...) (NOT C ...) bodyB)
```

together with control information which says that the conditional test "(C ...)" need not be executed more than once.

Conditionals are a mixed blessing because less use can be made of unification. For example in the two clauses for A above it could be that the heads of the two clauses would naturally be slightly different. This means that when the conditional form is used the head must be the 'most general' of the two, with extra equalities in the conditional branches to bind the variables.

5.6.4 EQ

```
    (EQ t1 t2)   "t1 is identical to t2"
```

The EQ program implements the identity component of the equality axioms: viz x=x. For example, (EQ (x1 x2) (A B)) results in "x1" = "A", and "x2" = "B".

5.7 Data base operations

micro-PROLOG has a collection (3) of programs that enable clauses to be accessed, added and deleted from the user's work-space at run-time. Note that these facilities are dangerous (they are non-logic) and should be used with care. They are included because the supervisor needs them and also because they facilitate certain language extensions. (See Chapter 6 for examples of how they are used in this way.)

5.7.1 CL

```
(CL <clause>)                           "clause is a clause in the program"
OR (CL <clause> <start-#> <index-#>)    "clause after start-# at index-#"
```

This program accesses clauses from the user's work-space. This is one of the few built-in programs that is at all non-deterministic as it can be used to backtrack through an entire relation. The only input constraint on CL is that the predicate symbol of the clause is known. For example, (CL ((At|x)|X)) succeeds if there are any "At" clauses in the workspace, in which case the variable "x" is bound to the arguments of the head atom of the clause, and the variable "X" is bound to the (possibly empty) list of atoms that make up the body of the clause.

The three argument form of "CL" can be used to find particular clauses in the program, the start-# is the first clause to start looking at, and the index-# is the actual index of the clause found in the relation. For example this can be used to find where a clause is in a relation:

```
(CL ((likes John |x)|y) 1 X)
```

returns in "X" where the first clause that matches "((likes John|x)|y)" is in the "likes" program. (If the three argument form is used the second argument _must_ be given). We can also use it to see what the n^{th} clause is as in:

```
(CL ((likes|x)|y) 4 4)
```

5.7.2 ADDCL

```
(ADDCL clause {n}) "add clause to the program {after nth clause }"
```

The ADDCL program is used to insert clauses into the workspace. The clause to be added is named by a term constructed according to the syntax of clauses: a list of atoms, with the first atom being the head.

From a logical point of view, the syntax of clauses is such that the term which names a clause in the meta language is identical to the term which is the clause in the object language. Any variables that appear in the term naming a clause appear as variables in the clause that is added to the workspace. For a detailed examination of the relationship between meta level terms and object level terms and clauses see chapter 12 of Kowalski [1979].

This way of naming clauses (using 'real' variables to name variables of the clause) is actually logically incorrect, but PROLOG has always been done in this way.

If ADDCL is used with a single argument then the clause is added to the end of the appropriate program. Otherwise it is inserted after the n^{th} clause in the program. For example, if "0" is used as the clause number, the clause is inserted at the front of the program. For example, (ADDCL ((append () x x))) adds the clause

```
((append () x x))
```

to the end of the "append" program.

There is a restriction on the use of ADDCL: namely that clauses can only be added to user defined programs, not to built-in programs. This means that it is not possible to redefine the built-in programs, whether they are implemented in micro-PROLOG or assembler.

5.7.3 DELCL

```
(DELCL clause)    "delete clause from the workspace"  or
(DELCL pred n)    "delete clause n from program pred".
```

The program DELCL is used to delete clauses from the user's workspace. In the first form the program is searched for the clause, and if it is found (unified with the clause) then the appropriate clause is deleted. In the second form the clause to be deleted is specified by a predicate symbol and an offset. For example, (DELCL ((append | x1) | x2)) looks for an "append" clause and deletes it, the variables "x1" and "x2" are unified with the head arguments and body of the clause respectively.

The restrictions that apply to ADDCL also apply to DELCL: only user programs can have clauses deleted from them. If the clause is not part of the program then the call to DELCL fails.

5.8 Library procedures

The program library facilities from the supervisor are also available as built-in predicates. These allow saving of programs, loading programs and listing programs at the console.

5.8.1 LIST

```
   (LIST ALL)
or (LIST <list of predicate symbols>)  "list program(s) at console"
```

The LIST program either lists the entire user program at the console, or, if given a list of predicate symbols as a parameter, just individual programs.

```
For example, LIST ALL      lists the whole program,
         (LIST(Likes Fred Angie)) lists programs "Likes", "Fred" & "Angie"
```

5.8.2 SAVE

```
    (SAVE file)
or  (SAVE file <list of predicate symbols>) "save program on disk"
```

The SAVE program saves the user's program on the disk file. The second form of the SAVE program allows selective saving of the user's program onto disk. For example, (SAVE "A:TEST.LOG") saves the entire work space on the disk file "TEST.LOG" on drive "A". The old file, possibly from a previous "SAVE", is renamed with extension ".BAK", ensuring automatic backing up of programs.

5.8.3 LOAD

```
(LOAD file)   "load program from disk file"
```

This program reads the disk file, and loads the program contained on it. For example, (LOAD TRACE) loads the program in the disk file "TRACE.LOG" on the logged in disk.

5.9 Module Construction Facilities

micro-PROLOG offers a limited facility for constructing modules. These enable programs to be put together from different sources while avoiding name clashes.

A module has four components: a name, (which is a micro-PROLOG constant) a local dictionary which are the symbols appearing in the module which are private to that module, a list of names which are being 'made available' by the module: the Export list, and a list of names that the module imports from the outside: the Import list.

Modules are LOADed and SAVEd automatically by the LOAD and SAVE programs. For the LOAD the function is completely automatic: files containing modules have a different structure to ordinary program files, and for the SAVE program there is the extension:

```
      (SAVE file module-name)
```

which saves the module named (and its sub-modules) on the specified file.

The LIST program can also be used to list the contents of a module; by using

```
      (LIST module-name)
```

At any one time there is what is called the 'current' module. This defines what programs are currently available. In the current module the symbols that are available are those appearing in the Import/Export lists as well as the local names. However when the current module is LISTed, or SAVEd only those programs from the Export list and the local programs are displayed.

The supervisor uses the current module's name as its prompt, so if the current module were called "simple" then instead of the "&." prompt we get the prompt:

```
      simple.
```

There are three programs that are provided to control the uses of modules: CRMOD, OPMOD and CLMOD.

5.9.1 CRMOD

```
      (CRMOD module-name <Export list> <Import list>) "create module"
```

CRMOD creates a new module of name module-name, and enters it; i.e. makes it the current module. The Export list and the Import list are as defined above.

5.9.2 OPMOD

```
      (OPMOD module-name) "open module"
```

OPMOD enters the already existing module named; i.e. it becomes the new current module.

5.9.3 CLMOD

```
      CLMOD "close module"
```

CLMOD drops out of the current module back into what was the previous current module. It is not possible to drop out of the base module.

5.10 Miscellaneous Predicates

In this section we draw together a rag-bag of functions not covered above. These include the dictionary relation and some control functions.

5.10.1 DICT

(DICT module <Export list> <Import list> <seq. of constants>)

The DICT program contains the dictionary of the current module.

5.10.2 QT

QT   "exit to monitor"

Execution of this program will cause an exit from the micro-PROLOG system into CP/M.

5.10.3 /

/   for controlling backtracking

The slash program is used to control backtracking. Its effect is to eliminate backtracking between the call which invokes the clause, and the "/" evaluation. For example, in the program

```
((P ...)(A ...)(B ...)/(C ...))
```

the atoms (A ...) and (B ...) are executed in the normal way. If they succeed then the slash is executed. Slash always 'succeeds'; it is used for its side effect.

It suppresses further backtracking in the evaluation of the (A ...) and (B ...) atoms and it also removes any alternative clauses for P that have not yet been tried. In other words if (C ...) fails, then the call (P alternative ways of executing the (A ...) and (B ...) atoms, and there may have been more (P ...) clauses; none of which will be tried.

The slash can also be used to 'tell' micro-PROLOG that certain programs are actually deterministic, allowing the system to make space optimizations.

5.10.4 FAIL

FAIL   "false".

The FAIL predicate always evaluates to false. This is used to fail a branch of the proof, FAIL has no clauses, but the interpreter knows about it and does not report a "CLAUSE ERROR".

5.10.5 <SUP>

<SUP> is the supervisor.

The supervisor appears as the program for the predicate symbol "<SUP>". It is a very simple program with just a few clauses. Its main function is to call user programs, and other built-in functions. The actual program is shown here:

```
(("<SUP>")
     (CMOD Y)        {find the current module name (system use only)}
```

```
        (P Y)             {print module name as prompt}
        (R X)             {read in command word}
        ("<>" X)/         {process command deterministically}
        ("<SUP>"))        {tail recurse on supervisor (look for next command}

  (("<>" X)
        (CON X)           {if command is a constant..}
        (R Y)             {read in single argument}
        (X Y))            {execute command (using meta-variable)}

  (("<>" (X|Y))           {if command is a micro-PROLOG clause..}
        (ADDCL (X|Y)))    {add it to the program}

  (("<>" X)               {come here if command fails, or illegal input}
        (PP ?))
```

Chapter 6

Implementing High-level features with low-level primitives

This chapter is dedicated to the would be supervisor writer: that is the programmer who wishes to extend micro-PROLOG with high level features. To illustrate how one might go about extending the system we take a number of fairly simple extensions and describe in some detail how they are implemented using the low-level primitives given.

A common kind of extension involves extending the language towards a richer notation which is closer to full first order standard form [Kowalski 1979]. However, since we tend to rely on the operational semantics of the interpreter for the implementation of a new feature, we generally have to compromise to some extent on its logical nature to achieve a reasonable efficiency. Usually, a subset only of the full power is implemented.

The extensions we consider are negation, 'One-of', conditionals, lists of solutions. Negation and conditionals are already in the supervisor. Finally we describe how to extend the system by implementing a front end to the sytem, in particular we look at the "Simple" system discussed in Chapter 4.

6.1 Negation

Most PROLOG systems do not support full logical negation, but rather attempt to implement 'negation-as-failure'. This can be shown, under certain circumstances, to be equivalent to classical negation and for a full treatment of this see Clark [1978]. In this section we concern ourselves only with its implementation.

The essence of the negation-as-failure rule is that an atom is true with respect to a logic program only if its provably true, and false otherwise. So negation can be implemented by the simple trick of reversing the normal failure and success pattern of the interpreter. We fail a negated call if the call succeeds, and we succeed the negated call if the call fails. A simple micro-PROLOG program that does this is:

```
((NOT x) x / FAIL)
((NOT x))
```

Operationally (the only way this program can be understood), what occurs when NOT is called, for example:

```
(NOT (EQ A B))
```

is that the argument of NOT (which names an atom) is itself called using the meta-variable facility.

If the atom in the NOT call succeeds then the slash is executed. The effect of the slash is to remove all the choice points inside the called atom. Moreover, it also removes the second NOT clause from consideration. After the slash the built-in predicate FAIL is executed, which of course fails. Now, since there are no more choices left for NOT and for the called atom (they have been explicitly removed) the call to NOT also fails. Hence if the call to the atom succeeds, then the call to NOT fails.

If however, the call to the atom fails, then the second clause for NOT is tried. This clause always, unconditionally, succeeds. So the NOT call succeeds if the atom fails:

```
&.??((NOT (EQ A B)))
ENTER (NOT (EQ A B))).C
ENTER (EQ A B).C
FAIL (EQ A B)
FINISH (NOT (EQ A B))
&.
```

Roughly then, this is the behaviour we want from the NOT program. However there is an important case where the program goes wrong. If the call to the argument atom succeeded, but only by binding a variable in it then it is wrong to fail the NOT call, what we should do is abort the execution with a "CONTROL ERROR".

It is too expensive to do the required check though, and for cases where the atom has no variable in it the check is not necessary anyway, so we do not program it up.

Finally a syntactic convenience: as it stands the NOT program above requires a single argument that itself names an atom. This leads to two extra brackets when it is used, compared with a normal atom. A more elegant approach is to allow NOT to accept a list of arguments, this list naming an atom: the first argument is the predicate symbol, and the rest are the arguments to the atom. The result of this is that we can write our negated atoms with no extra brackets:

```
(NOT EQ A B)
```

The modified program that accepts this is:

```
((NOT | x) x / FAIL)
((NOT | x))
```

6.2 One-of

The 'One-of' operator is a useful way of telling the system that a particular call to a program is actually functional: there is only one solution. This then allows the system to economise on space by not saving the backtracking points left after finding the first solution.

In writing this program we choose the same format as for negation: the 'One-of' symbol comes at the front of the atom involved. For example,

```
(One-of Member (B x) ((A 32)(B 1)(C 3)(B 2)))
```

only finds the first "(B x)" pair in the list.

The 'One-of' program is simple: we find the first solution, by using the meta-variable facility as in NOT, and then remove all the remaining backtracking points by slash:

```
((One-of | x) x /)
```

6.3 Conditionals

Conditionals are used to express conditional branches in a program clause: one of two branches is executed depending on the result of some test. The ideal we want to aim for is for a statement of the form:

```
((A ...)(IF (C ...)((THEN ...))((ELSE ...))))
```

to be logically identical to the pair of clauses:

```
((A ...)(C ...)(THEN ...))
```

```
        ((A ...)(NOT  C  ...)(ELSE  ...))
```

together with control information that the test "(C ...)" only needs to be performed once. However we have to compromise, so we actually implement the conditional to be equivalent to the pair of clauses:

```
        ((A  ...)(One-of  C  ...)(THEN  ...))
        ((A  ...)(NOT  C  ...)(ELSE ...))
```

The IF program has two clauses. The first calls the conditional test, and if that succeeds it removes the second clause, by using slash, and executes the THEN branch. The second IF clause (which is only entered if the conditional test fails) simply executes the ELSE branch.

```
        ((IF  x X Y)
              x
              /
              | X)
        ((IF  x X Y)  |  Y)
```

A more complete solution, which allows backtracking on the conditional test, would call the conditional test twice: once in such a way as to avoid binding any variables in it, and if that succeeds then to call it again in the 'then' branch to bind the variables. The problem with this solution is that it calls the conditional test twice which is exactly what we wanted to avoid! For this reason the simpler program is the one actually embedded in the supervisor as the IF program.

6.4 Lists of solutions

A common requirement in logic programs, particularly for data-base applications, is for a complete list of answers to be constructed, rather than a single one at a time. To answer this need we extend the language to allow a primitive 'set' construct.

What we would like is for an expression of the form "{< , , >|P}" which signifies a set of tuples each corresponding to a different way of solving P, to be written in micro-PROLOG. What we settle for is a program All which has the form "(ALL x y z)" which unifies "x" with a list of terms "y" each of which corresponds to a solution of the atom "z". Note that the termination of a call to All is entirely governed by whether all of the answers to "z" can be found in a finite time, and that duplicate solutions are not removed.

The strategy we follow when programming All is to call the atom "z" and every time it succeeds add to a global variable the term "y" and then artificially fail. When all the alternatives for the atom are exhausted the call to All succeeds by binding the list constructed to the answer variable.

Although PROLOG is an applicative (i.e. side effect free) language, the primitives ADDCL and DELCL can be used to program up global variables which can be overwritten. For example to maintain a variable called 'Global' we have in the workspace a clause of the form:

```
        ((Global  Value))
```

where Value is the current value of the Global variable. To access the variable's value we simply have a call to Global:

```
        (Global  x)
```

To update the value of the global variable we first delete the current clause using DELCL, and then ADDCL a new clause with the updated value.

```
        (DELCL ((Global  oldvalue)))(ADDCL ((Global  newvalue)))
```

If we wish to simultanecusly access and delete the value of a global variable we can do so by using the DELCL to unify a variable with the old value:

```
        (DELCL  ((Global  x)))
```

For example in our All program we maintain as a global variable the list of solutions found so far. When a new solution is found we want to add it to the list, to do this we access the list and remove the global variable at the same time using

```
        (DELCL  ((Global  X)))(ADDCL  ((Global  (newterm|X))))
```

The program for finding all solutions can be written as:

```
        ((All x y z)
         (ADDCL ((Global ())))
         z
         (DELCL ((Global Y)))
         (ADDCL ((Global (y|Y))))
         FAIL)
        ((All x y z)
         (DELCL ((Global x))))
```

The difficulty with this program is that it does not allow for the possibility of more than one All call being invoked at once: an All call may itself have All calls imbedded in it.

The program for All presented below is a little more complicated since it keeps track of different calls to the same program (even nested calls). To to this we have a second global variable All-num which counts the number of times the All program has been called, and we have renamed the original global variable to All-list.

```
        ((All x y z)
         (DELCL ((All-num X1)))
         (SUM X1 1 X2)
         (ADDCL ((All-num X2)))
         (ADDCL ((All-list X1())))
         (All-find X1 x y z))

        ((All-find X1 x y z)
         z
         (DELCL ((All-list X1 Y)))
         (ADDCL ((All-list X1 (y|Y))))
         FAIL)

        ((All-find X1 x y z)
         (DELCL ((All-list X1 x))))

        ((All-num 0))
```

6.5 The Simple PROLOG front end

In this section we look at the implementation of the Simple PROLOG system described in Chapter 4. It illustrates how a more friendly system can be built from the micro-PROLOG basics.

The main function of the front end program is to compile between Simple PROLOG sentences and micro-PROLOG clauses, and to invoke evaluation of Simple PROLOG queries. Of course the correspondence is very close: Simple PROLOG is essentially syntactic sugar for micro-PROLOG. The principles behind the implementation apply to more complex transformations, such as a natural language front end system.

6.5.1 The translator

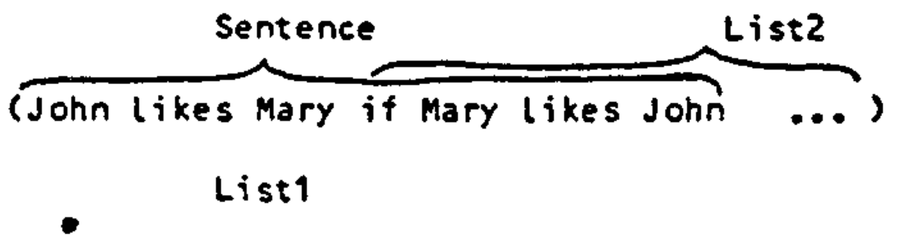
The kernel program in Simple is "parses-to", this translates between Simple PROLOG and micro-PROLOG. Here we shall look at a simplified version of the program, one that only knows about binary predicates.

A Simple PROLOG sentence has a very simple (sic) structure: it consists of a list of terms which are grouped into atoms; the atoms of the sentence are seperated by some constants which act as key words. On the other hand micro-PROLOG clauses have a list structure: a clause consists of a list of atoms, each atom is itself a list the first element of it being the predicate symbol.

We can represent a Simple PROLOG sentence as a list of terms:

```
(John likes Mary if Mary likes John)
```

or, more generally, we can represent it by a pair of lists, the *difference* between them being the Simple PROLOG sentence:



We can name this difference between the two lists by a term of the form "(list1 to list2)":

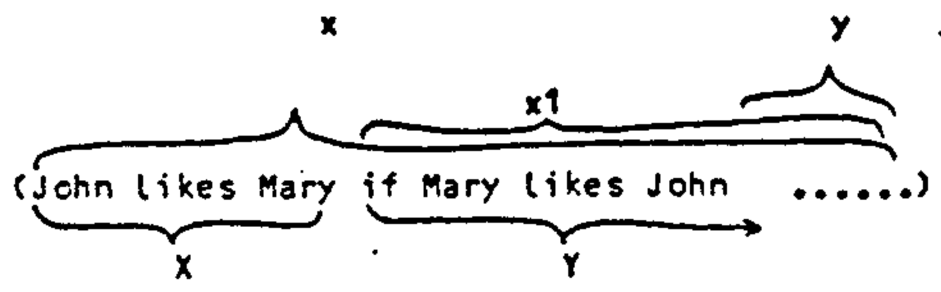
```
((John likes Mary if Mary likes John ...) to ...)
```

This latter representation is very useful when parsing the Simple PROLOG sentence; we can write the parser as:

```
((parse (x to y) (X|Y))
     (is-atom (x to x1) X)
     (is-body (x1 to y) Y (if)))
```

The English reading of this is:

> To parse the sentence represented by the list "x to y" into the clause with head X and tail Y, first find an atom X between x and x1, then parse the body Y from x1 to y using the keyword "if".



Since we are only parsing binary predicates of Simple, the program for determining what an atom is is easy:

```
     ((is-atom ((x1 x x2|y) to y) (x x1 x2)))
```

This sentence says that three terms define an atom, they are represented using the pair of lists "(x1 x x2|y)" and "y", the difference between them is the list "(x1 x x2)", and this list is the micro-PROLOG atom "(x x1 x2)". Notice how easy this transformation is, the first three terms of the list representing the Simple PROLOG sentence only need to be slightly re-ordered to put them into the micro-PROLOG representation of an atom.

The body of a Simple PROLOG sentence is a sequence of atoms, each atom is preceded either by the keyword "if" for the first atom, or one of "&" or "and" for each remaining atom in the body. This makes the definition of the "is-body" program straight forward:

```
     ((is-body ((x|x1) to x2) (X|Y) z)
          (in-key-list x z)
          (is-atom (x1 to x3) X)
          (is-body (x3 to x2) Y (& and)))
```

The base case, where the body of the sentence is empty is equally straight forward:

```
     ((is-body (x to x) () z))
```

The empty list is named here by the pair "(x to x)"; which says that for any list x the difference between it and itself is empty.

Although the above programs are written in Simple PROLOG the actual translator is of course written in micro-PROLOG. The program is equally good at translating from Simple into micro-PROLOG as it is at translating from Micro to Simple PROLOG.

6.5.2 Organization of Simple PROLOG programs

As we mentioned above the Simple front end program compiles Simple PROLOG programs into micro-PROLOG format. It also stores the Simple programs as micro-PROLOG clauses. This allows the micro-PROLOG system itself to be used for evaluating Simple PROLOG queries.

However, since Simple is itself a micro-PROLOG program some way has to be found of separating the two programs: the Simple PROLOG program and the front end program itself. The reason for wanting to searate them is for program list .g and saving programs: the Simple PROLOG user does not want to see the ´ont end program when his Simple PROLOG program is Listed or Saved in a file.

This separation is achieved by using the module construction tools described in Chapter 5. The front end program is formed into a module called "Simple" which effectively hides the program from the user. All that remains visible is the user's Simple PROLOG program. This also allows us to use the built-in SAVE and LOAD programs to Save and Load

Simple PROLOG programs.

6.5.3 Evaluations of Simple PROLOG queries

In this section we look at how the "Which" command is implemented. "Which" takes a goal and a term in Simple format, converts the goal to micro-PROLOG format and then, using the meta variable, evaluates the query. If the query is successful the term is printed and the next solution to the query is found by artificially failing.

"Which" uses the program "is-body" above to parse the Simple query. However "is-body" expects either the empty body or a keyword, such as "if" or "and" on the front of the body; so "Which" tacks-on a suitable keyword to the query.

```
((Which (x|y))
     (is-body ((if|y) to ()) z (if))
     (exec x z))
```

where "exec" executes the **micro-PROLOG** query "z" using "?" and displays the result "x" for each solution:

```
((exec x z)
     (? z)
     (P x)
     PP
     FAIL)
((exec x z)
     ((PP No (more) Solutions))
```

The "One" command is implemented in essentially the same manner. The main difference is that instead of just failing after each solution is printed the console is queried; if the user replies "C" (for continue) then the next solution is sought, otherwise the "One" command just terminates.

6.5.4. Summary

This section illustrates some of the techniques which can be used to implement a front end system to **micro-PROLOG**. Simple is a module which defines some new commands which implement the various necessary functions. It translates from the Simple PROLOG syntax into **micro-PROLOG** syntax, and Simple programs are actually stored in **micro-PROLOG** form. Queries are answered by first translating to **micro-PROLOG** form and then using the meta variable facility to evaluate them.

The listing of the full Simple program is given in Appendix D.

Chapter 7

Adding assembler coded subroutines

It is part of the philosophy of micro-PROLOG that it should be as extensible as possible. This is reflected in the flexibility of the syntax, as well as in the inherent extensibility of PROLOG. A further kind of extension provided for in micro-PROLOG is the ability to add programs to the system that are written in other languages, in particular assembly coded programs, and have them automatically executed by the system like any other program.

To this end we have an interface which, if followed exactly, allows a 'foreign' program to be called by the system and parameters to be passed between it and micro-PROLOG. This interface is also used by the bulk of the built-in programs, so this chapter also gives a flavour of how they are implemented.

A user coded program is invoked in the normal way, by an atom in a goal statement, or in a clause. Like the built-in machine coded programs the extent to which it behaves like a normal program, written as clauses, depends on how many of the program's uses have been catered for, though the interface only handles deterministic uses. If a non-deterministic use is to be handled, then it can be programmed up using a micro-PROLOG program that explicitly sequences through the non-deterministic choices.

The principal interface between micro-PROLOG and a user coded (or any other) machine language program consists of three components. A number of data registers are provided through which parameter values are passed between micro-PROLOG and the machine coded program. A Type tree is used to specify what types of arguments the program can accept, how many of them, and what patterns of use are supported. The type tree also specifies the actual entry points into the program, so that depending on the particular call different entry points may be entered. The third component of the interface is the predicate symbol declaration. This declares to micro-PROLOG a constant which describes the name to be used to access the program and its initial entry point.

To illustrate the method for inserting a new program into micro-PROLOG we take as a simple case study a psuedo random number generator.

The algorithm we use is based on Knuth[1968]. The most important properties of this algorithm are that it passes every statistical test for randomness, and it is guaranteed to have a cycle length of 2^{16} (The maximum possible length of cycle with a 16 bit number).

The formula for computing the next random number in a sequence is:

```
rand(n+1) = 13849 + 16385 * rand(n)
```

The format of a call to our random number generator is:

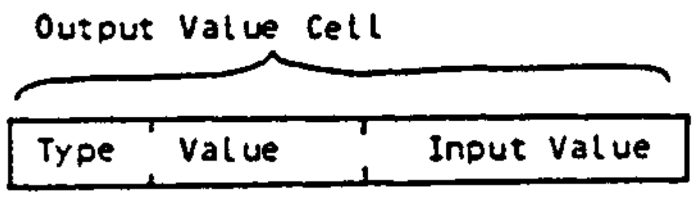
```
(RND var)
```

and it succeeds by binding the variable to the next random number in the sequence.

7.1 Data registers

There are eight of these registers provided in the system, corresponding to up to eight arguments in a call. No user coded program may have more than eight arguments, though the system does not check this. Of course most programs have considerably fewer than eight arguments, in which case not all of the registers are used. However, those that are not used must not be altered in any way by the user program.

A Data register has two separate components, one for input data to the user program, and one for returning results. The input side is two bytes long: sufficient for a 16 bit number, or a pointer. The output side is three bytes long and comprises a 'value cell'. A Value cell has a one byte type field, and a two byte data field. The format of a data register is:



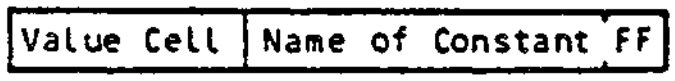
The input component of the data register is determined by the type tree. It can be either a variable, number, constant or list pointer. The type is not made explicit as it is assumed that the type checking of the type tree enables the selected entry point to 'know' the type of data in a given input register. The user must under no circumstances affect the value in the input register: the value can be read, but not modified.

The output component has its type field initially set to OFFH. To pass back a value it is necessary to assign the type of the answer to this field, and to place the appropriate value in the value field. If the type field is left at OFFH then no value is passed back, and the corresponding variable is left unbound.

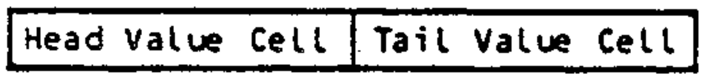
The various data types that are recognised by micro-PROLOG include:

NUMBER	=	4	Data field holds a 16 bit number in two's complement form.
NIL	=	16	The empty list. Data field ignored.
CONSTANT	=	8	Data field points to a constant structure
LIST	=	3	Data field points to a list cell

A Constant has a structure of the form:



A list cell consists simply of two value cells contiguous in memory, with the head cell first and on an even byte boundary.



7.1.1 Warning

There are other types recognised by micro-PROLOG, but all other values are reserved by micro-PROLOG. The type field is checked at various points in the system, and an invalid type may result in micro-PROLOG being aborted.

In fact it is envisaged that the type most commonly used by user programs is that of number. It is for the sake of completeness that the other types have been described.

7.2 Type tree

Type checking of arguments to a machine coded program is controlled by the type tree. This is a data structure that is part of the program, and must be provided with it. Only if no arguments are expected to a call may this tree be omitted, but if the programmer wants the system to check that it is called with no arguments then a tree can be specified to check for this.

Each node in the tree has 5 fields, corresponding to the possible types that an argument in the call can have. Each different type of argument leads to a sub-tree of the type tree, with an empty subtree signifying that a particular type of argument is not allowed. The empty subtree is marked by having the value 0FFH in the corresponding field of the node. The depth of the tree corresponds with the argument position in the call: the root of the tree deals with the first argument, and the nodes in the second level (i.e. those immediately descended from the root) deal with the second argument position.

Type tree node:

```
     Leaf Num Con List Var
```

For example the type tree for the SUM predicate may be represented as follows:

```
                        |
              ---------------------
              |                   |
             num                 var            1st Arg
              |                   |
       -------------              |
       |           |              |
      num         var            num            2nd Arg
       |           |              |
  ------------     |              |
  |          |     |              |
 num        var   num            num            3rd Arg
  |          |     |              |
 leaf       leaf  leaf           leaf
 check      add   subtract     subtract         Sub-programs
```

And for our RND program it is:

```
                          |
                         var                                1st Arg
                          |
                         leaf
```

1. The "Leaf" field refers to the end of the argument list: i.e. no argument. The subtree rooted at the leaf field is actually an entry point into the code of the user program proper. At this point all of the parameters to the call will have been parsed and the appropriate values placed in the data registers. Furthermore, since the path from the root node of the tree to the entry point is unique the code can simply access the values in the knowledge that the types are as expected. All that is left for the program to do is to compute the answer values, place the result in the output halves of the data registers and return.

When returning from a leaf program (by executing a "RET" instruction) the system checks the return code for success or failure. If the "Z" flag is set then the system assumes success, and the variables are bound as specified, if however, the "Z" flag is reset then the call is assumed to have failed. In this case the micro-PROLOG system backtracks in the normal way.

2. The "Num" field has rooted from it a non-empty subtree if a number was allowed in the current argument position. If a number is present, and the number subtree is non-empty then the number in the call is loaded into the appropriate data register, the number subtree followed and the next argument considered.

3. If the "Con" subtree is non-empty then a constant is allowed as a legal argument. If a constant appears as an actual parameter then the constant's address is loaded into the input data register.

4. If the "List" subtree is non-empty then a list is allowed as an actual parameter. Note that this means that a non-empty list as well as the empty list is allowed. If a list is encountered as an actual parameter then a pointer to a value cell which points to the list (or has NIL as a type) is placed in the data register: not a pointer to the list itself.

5. If the "Var" subtree is non-empty then a variable is allowed as an actual parameter. If a variable is used where one is not allowed then a "CONTROL ERROR" is reported, similarly if only a variable is allowed but a variable not used as an actual parameter, then a "CONTROL ERROR" also results. The Variable subtree is used when the programmer expects to return a result in that argument position, although there is no actual compulsion to return a value. Note that, of course, values can not be returned other than through a variable!

Each field in the node is a single byte unsigned number in the range 5..255. If a non-empty subtree is rooted at a particular field then the number in the field is a relative offset: it is the distance, in bytes, between the target node or entry point and the base of the current node. If an actual parameter is of a type which has no legal subtree for it, for example if a number is supplied as a parameter but a number is not allowed, then the call fails, unless the actual parameter was a variable, or a variable was the only type allowed.

To set up the type tree interface, for our RND program for example, we must start the initial entry point of the program as follows:

```
        ORG <BOS>               ;Where <BOS> is the contents of BOS
RNDPRG: LD IX,RNDTRE            ;load IX with type tree for this prog
        JP TRWALK               ;entry point inside micro-PROLOG
                                ;which processes the type tree
```

The actual RND program, together with its type tree is given here:

```
RNDTRE: DEFB -1,-1,-1,-1,RNDT2-RNDTRE
RNDT2:  DEFB RNDENT-RNDT2,-1,-1,-1,-1

RNDENT: LD HL,(SEED)            ;Get last random number generated
        LD DE,16385
        CALL MLTPLY             ;multiply by factor
        LD DE,13849             ;add in offset
        ADD HL,DE              ;New random number generated
        LD (SEED),HL           ;Store it for the next call
        LD (DATA1+OUTDTA),HL   ;Store in output register for answer
        LD A,4                 ;Set up the answer type in the reg.
        LD (DATA1+OUTYPE),A
        CP A                   ;Set successful return code
        RET                    ;Return to micro-PROLOG

SEED:   DEFS 2                 ;Storage for random number seed.
```

7.3 Predicate symbol declaration

The predicate symbol declaration is used to describe the name of the new program, and its initial entry point, to the micro-PROLOG system. The predicate symbol is defined by a constant structure like that seen above. Note that the declaration of a constant is not in itself sufficient, since micro-PROLOG does not yet 'know' about it. The new constant has to be patched into the system dictionary before the program can be used.

The constant declaration for our RND program can be coded in assembler as follows:

```
RND:    DEFB 4                  ;Number type
        DEFW RNDPRG             ;Initial entry to RND program
        DEFM 'RND'              ;Text string of name of predicate symbol
        DEFB 0FFH               ;Byte terminator of name string
```

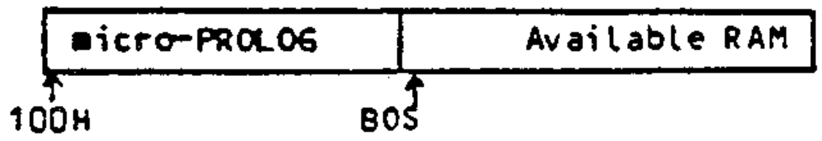
The entry in the dictionary takes the form of a list cell, and can be coded as:

```
NEWDCT: DEFB 8                   ;Constant type
        DEFW RND                 ;Point to new constant
        DEFB 3                   ;List type
        DEFW <SDICT>             ;Point to top of system dictionary
                                 ;Where <SDICT> is the contents of SDICT
```

7.4 Inserting a program

In the micro-PROLOG system there are two pointers which are necessary to adjust and know about when inserting a new program. The first is "BOS", which points to the first available byte of memory.

Appendix B gives a table of useful addresses within the micro-PROLOG system (including BOS).



The second is a pointer to the top of the system dictionary (SDICT).

The procedure for adding a new program to micro-PROLOG (once it is assembled into the right location) involves loading the new program starting at (BOS), updating the "BOS" pointer, and updating the system dictionary pointer.

Note It will be appreciated that any one attempting to augment micro-PROLOG by adding new assembler built-in programs should be reasonably proficient in (a) programming in assembler, (b) interfacing to CP/M (in particular be able to use that effectively and (c) using micro-PROLOG. Finally, be very careful as it is very easy to damage micro-PROLOG. (Never modify in any way the original distribution disk.) The interface described above is a very simple and powerful one; it is used by the great majority of the built-in programs in standard micro-PROLOG.

Adding built-in programs does not invalidate the licence agreement; however it is not permitted to sell or otherwise distribute an augmented version of micro-PROLOG without the written permission of the copyright holders of micro-PROLOG. Of course any augmentations that you build are _not_ the property of the copyright holders.

Appendix A

Error conditions and messages

Error message	Condition
Overflow Error	Arithmetic overflow in an arithmetic operation.
Clause Error	When a call is made to a program with no clauses.
Control Error	When a call is made to a system function. with too many variables, or when a meta-variable is used and it is not in the correct form.
Space Error	The heap has run out of space. (usually only occurs after extensive calls to garbage collector)
Dict Error	The dictionary area has overflowed. Try to rewrite program with fewer constants.
Syntax Error	Badly formed term. The read operation is restarted.
Break!	The user has interrupted the execution of the system.
Too many files opened	Too many active files for Micro-PROLOG.
File not found	The file specified in a OPEN call was not on disk.
Directory full	The directory space on the disk is full.
Disk full	The disk is full.
File closing error	An error on closing a file.
File Error	Attempted to write or read from an unopened file.
Not write mode	Shouldn't happen.
System Abort	Shouldn't happen. (Fatal error in Micro-PROLOG)

Appendix B

Useful addresses

Name	Address	Entry Parameters	Effect
DIVIDE	0115	DE=dividend,BC=divisor	DE=quotient,HL=remainder
INTCHK	0109	None	Polls for interrupts
LEXTYP	0118	A reg. ASCII character.	C contains type byte.
LINKDE	010C	DE points to value cell	DE dereferenced.
LINKHL	010F	HL " " " "	HL "
MLTPLY	0112	HL=multiplier,DE=m'cand	HL=product
MSG	0106	Call followed by text,0	Message displayed on console
PROLOG	0100	None	Micro-PROLOG cold start
TRWALK	0103	IX points to type tree	Execute built-in predicate

Data Areas

DATA1	01A5	Data registers 1..8 (five bytes each)
DATA2	01AA	
DATA3	01AF	
DATA4	01B4	
DATA5	01B9	
DATA6	01BE	
DATA7	01C3	
DATA8	01C8	
EOS	0006	Pointer to end of available memory (two bytes)
BOS	0121	Pointer to base of available memory (two bytes)
SDICT	011C	Pointer to top of system dictionary (two bytes)
LEXTAB	0124	Lexical type table (128 bytes)
NOVARS	0123	Number of variable prefix characters (one byte)
ERRCHR	01A4	Contains character used to print error variables (one byte)

Appendix C

Changing the lexical rules

The Micro-PROLOG tokeniser is a table driven system that separates the sequence of characters in the input into tokens. The table it uses, which is called LEXTAB, describes the character set in terms of different subsets: the separator characters, the special characters, the digits, the letters, the graphic characters, the sign and quote characters, and the variable prefix characters.

Each character's membership of these subsets is represented by a single byte in the table, with each bit in the byte representing a different set. If the appropriate bit in the byte is on (i.e. "1"), then it signifies that the character belongs to that set.

By modifying these subsets the lexical rules can be made to look very different; for example by merging the graphical and letter sets into one (the letter set) then the distinction that Micro-PROLOG makes between the two sets is ignored. This would allow such tokens as:

```
$A   %1   A'B
```

However, the subset most likely to be of interest is the variable prefix character set. This subset defines the conventions that Micro-PROLOG uses to distinguish variables from constants. In standard Micro-PROLOG the variable prefix subset is:

```
{"x" "y" "z" "X" "Y" "Z"}
```

Tokens beginning with these letters are recognised as variables. By changing this set we can implement different conventions for variables. This approach is a response to the current multiplicity of ways of recognising variables.

To implement the convention of tokens beginning with lower case letters being recognised as variables, and upper case as constants (as in IC-PROLOG [Clark & McCabe 1979] all that is necessary is that the variable prefix character set be changed to:

```
{"a" "b"  ..  "z"}
```

To implement the DEC-10 convention of upper case variables, lower case constants the variable prefix set should be changed to:

```
{"A" "B"  ..  "Z"}
```

Finally to implement the convention, found in the original Marseilles PROLOG and in Waterloo PROLOG, of using the character "*" in front of a token to signal a variable, the variable prefix character set should be:

```
{"*"}
```

Note that in this case the character "*" will also have to be made a letter.

Apart from reading variables, it is necessary to print them, preferably in the format that variables are read in. In Micro-PROLOG all variables are printed with a variable prefix character (possibly) followed by a sequence of digits.

This is of course the kind of token that would subsequently be read as a variable. The prefix character used is taken from the table of lexical types, each variable prefix character defined in the table will be used when printing variables, in the order that they appear in the table. Thus the first seven variables (in standard Micro-PROLOG) are printed as:

```
X  Y  Z  x  y  z  X1  ...
```

To actually change types of the various characters it is necessary to use the CP/M utility "ddt" to modify the prolog program. This utility is a general debugging package and is part of standard CP/M. We need to examine and modify certain locations in prolog. ddt is executed by using the CP/M command:

```
A> ddt prolog.com
```

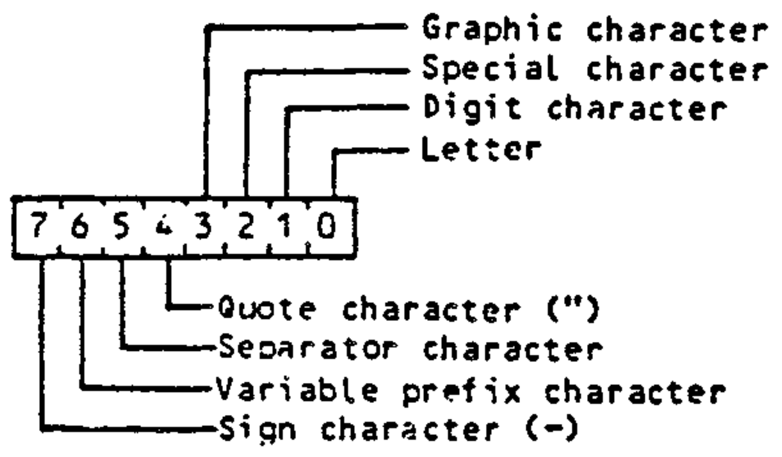
This loads the prolog system into memory and enters the command mode of ddt. The initial response of ddt is like:

```
DDT VER 2.XX
Next  pc
2F80  0100
```

Be careful to note down the number under "next" as it is needed later. ddt works entirely in hex arithmetic. The various addresses we use below are absolute memory addresses, as this is how CP/M is organized; in particular you should be careful about modifying memory locations other than described below as "ddt" allows you to change <u>any</u> memory location including ddt and CP/M! The table LEXTAB consists of 128 single byte entries. Each character in the ASCII character set has an entry associated with it; the entry is found by adding the value of the character to the base address of LEXTAB.

Each bit in the entry corresponds to one of the subsets discussed above, if the bit is on then the character is said to belong to the appropriate set, if off then the character does not belong to the subset in question. Note that the two characters corresponding to 0 (Nul) and 127 (Del) are illegal, and belong to none of the subsets.

Each byte in LEXTAB is organised as:



Some example entries of LEXTAB are:

Char	Hex	SVSQGSDL		Hex
"-"	2DH	10001001	=	89H Sign, Graphic & Letter
"A"	41H	00000001	=	01H Letter
"x"	6DH	01000001	=	41H Var. prefix & Letter
" "	20H	00100000	=	20H Separator
""""	22H	00010000	=	10H Quote

```
"("  26H    00000100    =   04H Special
"0"  30H    00000010    =   02H Digit
```

So, to implement our "*" convention for variables we have to change the table entries for "*", "X", "Y", "Z", "x", "y" & "z". The six letters have existing table entries of "41" (hex) which reflects that they are both letters and variable prefix characters; these entries have to be changed to be just letters. The letters are in two groups of three successive bytes in the table: "X", "Y" & "Z" and "x", "y" & "z". The table entry for any character can be found by adding the value of the ASCII representation of the character to the base address of LEXTAB (which is 0124 in hex). For example "X" in the 88th character in the ASCII sequence, (which is 58 in hex), so the entry for "X" is 124 + 58 (hex). We can use the ddt "h" command to do this hexadecimal arithmetic for us:

```
-h58,124
017C FF34
```

The first number printed is the address in memory (ignore the second number) of the table entry for "X", the entries for "Y" and "Z" immediately follow it. To modify the entry we use the ddt "s" command. This command enables memory locations to be modified in sequence. The old value of the byte is printed, and it is changed by entering (in hex) the new value of the byte. After carriage return is pressed the next memory location is examined, allowing it to be changed too. The "s" command is terminated by using "." instead of the new value of a byte. To change the "X", "Y" and "Z" entries we can type as in:

```
-s17C
017C 41 1
017D 41 1
017E 41 1
017F XX .
```

The "x" character is "78" (hex) in the ASCII code, so to change "x" & "y" and "z" to be just letters we do:

```
-h78,124
019C FF54
-s19C
019C 41 1
019D 41 1
019E 41 1
019F XX .
```

Now we have to declare the "*" character as a variable prefix character. We must also change it from being a graphic character to being a letter, this is so that the tokeniser treats "*123" as a single alphanumeric token. The "*" character must therefore have the code 41 (hex) as its entry in LEXTAB. To change the entry we do what we did for the letters, we add the ASCII value of "*" to the base address of LEXTAB and use the "s" command to change the entry byte. Now "*" has ASCII value 2A (hex), so the required entry is computed by:

```
-h124,2A
014E 00FA
```

And we change the entry by:

```
-s14E
014E 08 41
014F XX .
```

A special counter (NOVARS) contains the number of variable prefix characters in the table that are to be used when printing variables. This single byte counter should be changed, if the variable prefix character set is changed, to reflect the number of prefix characters. This number should <u>never</u> be greater than the actual number of variable prefix characters in the table, and it should also be at least one. So we have to change this location to 1 (one) which is the new number of variable prefix characters:

```
-s124
0124 06 1
0125 00 .
```

This completes the changes to make the variable convention "*" followed by digits. All that is now required is to exit ddt and save the memory image in the file "prolog.com", which has the effect of updating the old prolog system with the changes. ddt is exited by typing:

```
-^C
A> Save 47 prolog.com
```

The number in the save command is found by converting the original "next" value printed out by "ddt". When "ddt" is first entered it gives the length of prolog in pages. The save command expects this number in decimal form, whereas "ddt" displays it in hex, so you have to convert it.

<u>Warning</u>

Some of the entries in LEXTAB should not be changed. In particular the characters that are special should not be removed from the special set (otherwise the syntax analyser may not be able to recognise terms properly), and no new characters should be added to the special set. Furthermore, the sign character subset should always be {"-"}, and the quote character subset should remain {"""}. Apart from these restrictions, and from the obvious condition that the digit set should be {"0" "1" .. "9"}, there are no contraints.

Secondly, you should be careful about choosing which letters you use as variable prefix characters, since the "R" program converts single letter tokens into variables if the single letter is a variable prefix character. This means that certain programs which expect single letter responses (such as the editor which uses "e", "u" and "w" (among others)) to be constants may have to be modified. In particular if the letters "u", "v" and "w" were added to the standard set of variable prefix characters then the editor will have to be changed to use something other than "u" and "w" for the unwrap and wrap commands.

Note that the various PROLOG programs supplied with Micro-PROLOG will also have to be changed to reflect the new variable conventions you have implemented. This has to be done using a conventional text editor, such as the CP/M "ed" editor.

Appendix D

The Simple PROLOG front end program

```
Simple
(Add List Kill Delete Does One Which Save Load Accept Edit
 All Not Is-All For-All)
(End dict C & and if)
((version 2.12b))
((Add X)
  (NUM X) / (R Y) (Add X Y))
((Add X) /
  (Add 32767 X))
((Add X Y)
  (parse ((Z|x)|y) Y) (declare Z) (ADDCL ((Z|x)|y) X))
((Edit x)
  (dict x) (R y) (NUM y) (CL ((x|x1)|x2) y y)
  (parse ((x|x1)|x2) X) (RFILL X) (R Y)
  (parse ((x|X1)|X2) Y) (ADDCL ((x|X1)|X2) y) (DELCL x y))
((List X)
  (NOT EQ X All) / (List-pred X))
((List All)
  (CL ((dict x))) (List-pred x) FAIL)
((List All))
((Which (X|Y))
  (is-body (?) Z (?|Y)) (Whichex X Z))
((One (X|Y))
  (is-body (?) Z (?|Y)) (Oneex X Z))
((Does X)
  (is-body (?) Y (?|X)) (IF (? Y) ((PP YES)) ((PP NO))))
((Load X)
  (LOAD X))
((Save X)
  (SAVE X))
((Delete (x|y))/
  (parse z (x|y))
  (OR ((DELCL z))((PP No such sentence))))
((Delete X)
  (CON X) (R Y) (IF (DELCL X Y) () ((PP No such sentence))))
((Kill X) (DELCL X 1) (Kill X))
((Kill X)
  (P Program X deleted) PP)
((Accept X)
  (declare X) (Acceptin X))
((parse (X|Y) Z)
  (Atom Z X x) (is-body (if) Y x))
((is-body X () ()))
((is-body X (Y|Z) (x|y))
  (Mem x Y) (Literal Y y z) (is-body (and &) Z z))
((Literal X x y)
  (Special-Atom X x y)/)
((Literal X x y)
  (Atom x X y))
((Atom (X ()|Y) (X) Y)
  /)
((Atom (X Y Z|x) (Y X Z) x)
  (CON Y)/)
((Atom (X (Y|Z)|x) (X Y|Z) x))
((Special-Atom (Not|x) (Not y|z) z)
  (is-body (?) x (?|y)))
((Special-Atom (Is-All x (y|z)) (x Is-All (y|Z)|Y) Y)
  (is-body (?) z (?|Z)))
((Special-Atom (For-All x (y|z)) (X For-All(y|Z)|Y) Y)
  (is-body (?) x (?|X))
  (is-body (?) z (?|Z)))
((List-pred X)
  (CL ((X|Y)|Z)) (Rev-parse ((X|Y)|Z) x) (P|x) PP FAIL)
((List-pred X))
((Rev-parse (x|y) z)
  (Atom z x z1)
  (Rev-body y z1 "if
        "))
((Rev-body () () x))
((Rev-body (x|y) (z|Z) z)
  (Literal x Z Z1)
  (Rev-body y Z1 "and
        "))
((Oneex X Y)
  (? Y) (P Answer is X) (R Z) (IF (EQ Z C) (FAIL) ()))
((Oneex|X)
  (PP No (More) answers))
((Whichex X Y)
  (? Y) (P Answer is X) PP FAIL)
((Whichex X Y)
  (PP No (more) answers))
((Acceptin X)
  (P X) (R Y)
  (OR ((EQ Y End))
      ((OR ((EQ (Z x) Y) (ADDCL ((X Z x))))
           ((P What is Y ?)PP))
       (Acceptin X))))
((Mem X (X|Y)) /)
((Mem X (Y|Z))
  (Mem X Z))
((declare x)
  (OR ((CL((dict x))))((ADDCL ((dict x))))))
((Not|X)
  (? X) / FAIL)
((Not|X))
((Is-All X (Y|Z))
  (DELCL ((All-num x))) (SUM x 1 y) (ADDCL ((All-num y)))
  (All-find x X Y Z))
((For-All x (y|z))
  (NOT ?((? z) (NOT ? x))))
((All-find X Y Z x)
  (? x) (ADDCL ((All-list X Z))) FAIL)
((All-find X Y Z x)
  (Collect X Y))
((All-num 0))
((Collect X (Y|Z))
  (DELCL ((All-list X Y)))/
  (Collect X Z))
((Collect X ()))
CLMOD
```

Appendix E

The Micro-PROLOG Editor

```
Ed(Edit)(t n s b e o u w k i a)
((Version 2.12))
((D-C () () ()))
((D-C (X|Y) (X) Y))
((Rev-list () X X))
((Rev-list (X|Y) Z x)
  (Rev-list Y (X|Z) x))
((DownC X (() Y Z))
  (NOT VAR X)
  (D-C X Y Z))
((App-C () X X))
((App-C (X|Y) Z (X|x))
  (App-C Y Z x))
((BackC ((X|Y) Z x) (Y (X) y))
  (App-C Z x y))
((BackC (() (X) Y) (() () (X|Y))))
((NextC (X Y (Z|x)) (y (Z) x))
  (App-C Y X y))
((NextC (X (Y) ()) ((Y|X) () ())))
((Delete-in-C (() X Y) (() () Y)))
((Delete-in-C ((X|Y) Z x) (Y (X) x)))
((Front-C 0 () X X))
((Front-C X (Y|Z) x (Y|y))
  (LESS 0 X)
  (SUM 1 z X)
  (Front-C z Z x y))
((DisplayC (X () Y))
  (P No term)
  /)
((DisplayC (X (Y) Z))
  (P Y))
((Edit-in-C (x y z) i (x (Z) Y))
  (R Z)
  (App-C y z Y))
((Edit-in-C (x y z) a (x1 (Z) z))
  (R Z)
  (App-C y x x1))
((Edit-in-C X k Y)
  (Delete-in-C X Y))
((Edit-in-C (X (Y) Z) s (X (x) Z))
  (R y)
  (EQ y (Y x)))
((Edit-in-C (X (Y) Z) t (X (x) Z))
  (RFILL Y)
  (R x))
((Edit-in-C X n Y)
  (NextC X Y))
((Edit-in-C X b Y)
  (BackC X Y))
((Edit-in-C (X Y Z) w (X (x) y))
  (R z)
  (App-C Y Z X1)
  (Front-C z x y X1))
```

```
((Edit-in-C (X (Y) Z) u (X (x) y))
  (NOT VAR Y)
  (App-C Y Z (x|y)))
((Edit-in-C (X (Y) Z) e (X (x) Z))
  (DownC Y y)
  (Edit-term y x))
((UpC (X Y Z) x)
  (App-C Y Z y)
  (Rev-list X y x))
((EdC X Y o)
  (UpC X Y)
  /)
((EdC X Y Z)
  (Edit-in-C X Z x)
  /
  (Edit-term x Y))
((EdC X Y Z)
  (PP ?)
  (Edit-term X Y))
((Edit-term X Y)
  (DisplayC X)
  (R Z)
  (EdC X Y Z))
((Insert-in-P 0 1 Y)
  (ADDCL Y 0))
((Insert-in-P X X Y)
  (SUM 1 Z X)
  (ADDCL Y Z))
((Append-in-P 0 1 Y)
  (ADDCL Y 0))
((Append-in-P X Y Z)
  (SUM 1 X Y)
  (ADDCL Z X))
((Goto-P X Y ((X|Z)|x))
  (CL ((X|Z)|x) Y Y)
  /)
((Goto-P X Y "No clause")
  (LESS -1 Y))
((E-in-P X Y Z i x y)
  (R y)
  (Insert-in-P Y x y))
((E-in-P X Y Z a Y1 Z1)
  (R Z1)
  (Append-in-P Y Y1 Z1))
((E-in-P X Y Z k x y)
  (DELCL X Y)
  (SUM 1 x Y)
  (Goto-P X x y))
((E-in-P X Y Z n x y)
  (NOT ? ((EQ Z "No clause") (LESS 0 Y)))
  (SUM 1 Y x)
  (Goto-P X x y))
((E-in-P X Y Z b x y)
  (SUM 1 x Y)
  (Goto-P X x y))
((E-in-P X Y (Z|x) e Y ((X|y)|z))
  (Edit-term (() (Z) x) ((X|y)|z))
  (DELCL X Y)
  (SUM 1 X1 Y)
  (ADDCL ((X|y)|z) X1))
((E-in-P X Y (Z|Z1) t Y x)
  (?FILL (Z|Z1))
  (R x)
  (EQ x ((X|x1)|x2))
  (ADDCL x Y)
  (DELCL X Y))
((EdP X Y Z o)
  (PP Edit of X finished)
  /)
((EdP X Y Z x)
  (E-in-P X Y Z x y z)
  /
  (Edit-P X y z))
((EdP X Y Z x)
  (PP ?)
  (Edit-P X Y Z))
((Edit-P X Y Z)
  (P [ Y ] Z)
  (R x)
  (EdP X Y Z x))
((Edit X)
  (NOT SYS X)
  (OR ((CL ((X|Y)|Z) 1 1)
      (Edit-P X 1 ((X|Y)|Z))) ((Edit-P X 0 "No clause"))))
CLMOD
```

References

Clark, K.L., [1978], Negation as Failure. Logic and Data Bases, (H.Gallaire and J.Minker, Eds.), Plenum Press, New York, pp. 293-322.

Clark, K.L., [1980], Logic as a programming Calculus. To be published in 1981 by Springer-Verlag, New York.

Clark, K.L., McCabe, F., [1979], Control facilities of IC-PROLOG. Expert systems in the Micro-Electronic Age. Ed D.Michie Edinburgh Univ.Press.

Clark K.L., Ennals, J.R., McCabe, F., [1981], A Micro-PROLOG Primer, Logic Programming Associates Ltd.

Colmerauer, A.,[1973], Les systemes-Q ou un Formalisme pour Analyser et Synthetiser des Phrases sur Ordinateur. Publication Interne No.43, Dept. d'Informatique, Universite de Montreal.

Colmerauer, A.,[1978], Metamorphosis Grammars. Natural Language Communication with Computers, (L. Bolc, Ed.), Lecture Notes in Computer Science No. 63, Springer-Verlag, pp. 133-189.

Kanoui H., Van Canaghem M., [1980], Implementing a very high level language on a very low cost computer. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

Knuth D.E., [1968] The Art of Computer programming. pp 147-151. Addison Wesley. Volume II, Semi-numerical algorithms.

Kowalski, R.A., [1974], Predicate Logic as Programming Language. Proc. IFIP 74, North Holland Publishing Co., Amsterdam. pp. 569-574.

Kowalski, R.A., [1979], Logic for Problem Solving. Artificial Intelligence series, North Holland Inc., New York.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., [1962], LISP Programmers Manual. MIT Press. Cambridge, Mass.,

Moss, C.D.S., [1979], A New Grammar for Algol 68. Dep. Rep. 79/6, Imperial College, London.

Naur, P., ed. [1962] Revised Report on the Algorithmic Language Algol 60. IFIP 1962

Roberts G.W., [1977], An implementation of PROLOG. MSc thesis. Waterloo, Ontario, Canada.

Robinson, J.A., [1965] , A Machine Oriented Logic Based on the Resolution Principle. J. ACM 12 (January 1965), pp. 23-41.

Robinson, J.A., [1979], Logic: Form and Function. Edinburgh Univ.Press.

Roussel, P., Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, Sept. 1975.