ACORNSOFT

Micro PROLOG

on ROM for the BBC Microcomputer

Micro PROLOG

on ROM for the BBC Microcomputer

Reference manual

D R Brough
K L Clark
F G McCabe
C S Mellish

ACORNSOFT

Acknowledgement

The authors would like to express their gratitude to R.A. Kowalski without whom this project would have been impossible.

ISBN 0 907876 31 5

Produced by Acornsoft Limited

Note: Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

SECOND EDITION

Printed in the United Kingdom

Acornsoft Limited, Cambridge Technopark, 645 Newmarket Road, Cambridge CB5 8PD.

Contents

1 Introduction

This manual describes the micro-PROLOG logic programming system. PROLOG is a computer language based on predicate logic, in particular the clausal form of logic and resolution inference [Robinson 1965,1979]. Micro-PROLOG is an interpreted version of PROLOG tailored for interactive use on microcomputers. It is the end product of five years of continuous development in the field of PROLOG systems on micros.

The first PROLOG (which stands for *PRO*gramming in *LOG*ic) was implemented in 1972 in Marseilles by Colmerauer and Roussell as an interpreter in the medium level programming language ALGOL-W. A more efficient and improved interpreter was then built 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. Building on and extending the implementation techniques of the Marseilles PROLOG, there have been several other implementations, the principal ones being the Edinburgh DEC-10 PROLOG [Warren *et al.* 1978], Waterloo PROLOG [Roberts 1977], the Hungarian M-PROLOG [Szeredi 1982], a new implementation from Marseilles [Kanoui & Van Caneghem 1980] and the first implementation of micro-PROLOG [1980]. The Edinburgh implementation incorporates a compiler, the first to be written for the language. Recently, interest in PROLOG has been stimulated by the Japanese decision to use it as the core language for which they will design their fifth generation computers. The Japanese interest derived from the 'academic export' of the DEC-10 implementation to Japan.

1.1 micro-PROLOG

The core of micro-PROLOG is a small built-in supervisor program which accepts programs, commands and queries written in an elegantly simple but expressive standard syntax very similar to that of LISP. However, it is very easy to extend the facilities of the built-in supervisor by loading micro-PROLOG programs wrapped up as *modules*. By using a suitable supervisor extension, one can program in any syntax. Two such supervisor extensions, SIMPLE and MITSI, are supplied with the system. They allow beginning programmers to program in a more user friendly syntax.

SIMPLE is fully described in chapter 5 and is the program development system used by the introductory books [Clark & McCabe 1984], [Conlon 1985], [Ennals 1984]. MITSI is described in chapter 8. [Briggs 1984] is a tutorial introduction

to micro-PROLOG programming using MITSI. It is intended for school level educational use.

The penalty of using either of these front-end systems is a slight loss of programming power. We recommend that you use them in conjunction with one of the tutorial books only whilst you are learning about micro-PROLOG programming. We then recommend that you move on to programming in the standard syntax which experienced micro-PROLOG programmers prefer. MICRO, described in chapter 6, is a program development system offering all the facilities of SIMPLE but for programs written in standard syntax. After using SIMPLE we recommend that you move over to using MICRO to familiarise yourself with standard syntax programming. Any program developed under SIMPLE can be loaded and used under MICRO.

Finally, you can dispense with MICRO and use your own micro-PROLOG implemented aids to program development. [Clark & McCabe 1984] and [de Saram 1985] both give introductions to using and extending the built-in supervisor.

In addition to the three program development systems, SIMPLE, MICRO and MITSI, several other tools are provided with the system. These are:

EDITOR	{a structure editor for standard syntax programs},
TRACE	{a full trace package},
SPYTRACE	{allows spy-tracing of individual programs},
MODULES	{facilitates construction and editing of modules},
EXREL	{a package that allows disc files to be used as virtual memory for programs},
SIMTRACE	{a trace package for use with SIMPLE},
DEFTRAP	{a simple error handler},
ERRTRAP	{a full error handling and recovery utility}.

Modules are an important program structuring facility of micro-PROLOG which is one of a very few implementations of PROLOG to support modules. Micro-PROLOG modules are named collections of relation definitions that communicate with other programs via import/export name lists. All other names used in the module are local to the module. This localised name feature enables modules developed at different times, or by different programmers, to be used together without fear of a name clash. Finally, exported names of relations defined in the module can be used in other programs as though they were primitive relations of micro-PROLOG. Hence the role of modules in extending the facilities of the supervisor.

The EDITOR, TRACE, SPYTRACE, MODULES and EXREL programs are described in chapter 4. SIMTRACE and DEFTRAP are described in chapter 5

along with SIMPLE, and ERRTRAP is described in chapter 6 along with MICRO.

All of these tools are utility modules which can be optionally loaded as extensions to the built-in supervisor and deleted when they are no longer needed.

1.2 micro-PROLOG 3.1

This reference manual is for version 3.1 on 6502 machines.

The 6502 version of micro-PROLOG 3.1 incorporates special primitives for exploiting the unique facilities of your machine. Appendix C gives details of these special primitives.

1.3 User preparation

To be able to exploit the full power of micro-PROLOG on your micro we recommend that you study the introductory leaflet supplied with the software, then read the whole of the manual. This effort to become fully conversant with all the facilities of micro-PROLOG will be well rewarded.

1.4 Notation conventions

The manual includes many examples of small micro-PROLOG programs to illustrate the use of some facility. A special typeface (OCRB) is used to denote example programs and also the names of relations and modules within the text.

In specifying the syntax of some micro-PROLOG construct the notation

<type>

is used to indicate that a field or sub-expression can be any instance of the indicated type. For example,

`LOAD` <file name>

is used to indicate that any allowed file name can be used in the `LOAD` command. <type1> and <type2> will be used when two uses of the same type *can* be different. The use of the different subscripts does not imply that they *must* be different.

The list constructor (see 2.5.1) is the character ! (vertical bar symbol) but the form .. (two adjacent dots) is an acceptable alternative on most micros. Throughout the manual, and in all the introductory books, ! is used.

One syntactic category, that of <term>, is used so often that the abbreviations *t*, *t1*, *t2* etc. are used for <term>, <term1>, <term2>. The form

t1 ... tk

is used to indicate a sequence of k terms. Unless the condition k>0 is explicitly given, this form includes the case when there are no terms in the sequence. Again, the use of the different subscripts does not imply that the terms *must* be different. It implies only that they *may* be different.

Finally, some example programs will be given with associated comments between "{", "}" brackets. They are not part of the program and should not be typed if you enter the program. Micro-PROLOG does not accept comments in this form.

Warning

This manual is not an introduction to micro-PROLOG programming. It is intended to be used as a reference manual by the programmer who already has some knowledge of PROLOG programming. For an introductory text on the general ideas of logic programming we recommend the book *Logic for Problem Solving* [Kowalski 1979].

Trade marks

Micro-PROLOG is a trade mark of Logic Programming Associates Ltd.

2 Standard syntax of micro-PROLOG

The standard syntax of micro-PROLOG is modelled on LISP syntax [McCarthy *et al.* 1962]. If a richer syntax is desired it has to be supported by a front end micro-PROLOG program that compiles into the standard syntax. In this chapter we describe the standard syntax.

In chapter 5 a more user friendly syntax that is compiled into the standard syntax by the SIMPLE extension for the supervisor is described. The SIMPLE supported syntax is just one option for a more elaborate syntax. The MITSI syntax described in chapter 8 is another.

There are only four different kinds of syntactic objects that micro-PROLOG knows about: *Numbers*, *Constants*, *Variables* and *Lists*. They are all different kinds of *Term*.

The only data constructor in micro-PROLOG is the list constructor denoted by | (vertical bar) or . . (two dots). Lists constructed using | can be written in a specially condensed syntax as in LISP. Other PROLOG systems do allow other kinds of data constructors; these can be easily simulated (with no great loss of efficiency) using lists.

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..254. The characters corresponding to 0 and 255 are illegal in micro-PROLOG, and are ignored if used.

2.2 Numbers

Numbers are integers or are floating point numbers.

A *positive integer* is written as a contiguous sequence of digit characters, with no leading sign character, e.g. `0 30 1025 32767`.

A *negative integer* is written with the leading sign character - contiguously followed by a positive integer. For example, `-1 -30 & -32767` are all negative numbers. If a sign character does not have a positive integer contiguously following it, then it is not regarded as the sign of a number. Thus

- on its own is a valid syntactic object, differing from any number. Integers must be in the range -10^9+1 to 10^9-1.

A *floating point number* can have up to 9 digits of precision, and can range from 2^{-39} to 2^{38}. They are written in a fairly conventional notation; some example floating point numbers are:

```
2.34  10.E15  12.003E-20  -0.9
```

One and only one period must be present in a floating point number. As with an integer, it must start with a digit or a minus sign and a digit. The *E* exponent is optional, but if used it must be contiguous to the number and must be contiguously followed by an integer. The following are *not* floating point numbers:

```
.9        {starts with .}
34 E3     {space before E}
-.7E45    {no digit after -}
56E4.8    {exponent not an integer}
```

No matter how they are entered, numbers are always displayed in a standard format, which depends on the magnitude of the number. If the magnitude of the number is in the range 0.1 to 999999999 then it is displayed with no exponent as in:

```
4.2345678 -50.35
```

Otherwise floating point numbers are displayed in the standard scientific notation:

```
3.4E-4 1.23E10
```

Always only the significant digits are displayed – trailing 0s after the . are suppressed. A . is not printed if the number is an integer.

Only negative numbers have a sign character in front of them, positive numbers *do not* have a sign character. If you enter `+6.86` micro-PROLOG will interpret this as a sequence of two terms, the constant + and the positive number `6.86`.

2.3 Constants

Constants are the simple unstructured objects of micro-PROLOG. They are used to name individuals such as `fred`, `A1`. They are also used to name relations such as `member-of`, `father-of`.

A constant is normally written as an alphabetical character (a 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 in a sequence of letters and digits. This can be used to split up long names with several English words.

Constants can also be written using the non-alphanumeric characters such as `. ! ,` etc. This kind of constant is written as any sequence of symbolic characters (or graphic characters supported by the machine) other than:

`( ) |` or `..`

which have a special syntactic role, and

`{ } [ ] < >`

which are always interpreted as single character constants.

The symbols that can be used to form a symbolic constant include:

`! # $ % & ' = -^~ \ @ £ ; : , . / + * ?`

Thus,

`** ! ?? -- :=`

are all symbolic constants.

Finally, a constant can also be written as a quoted sequence of characters, in which case there are no restrictions on the characters that can be used in the constant. A quoted constant consists of a sequence of characters surrounded by the double quote character: ". Examples of quoted constants are:

`"1" "The man" "?ERROR?" "$100" "<SUP>"`

A quoted constant can include a space, as in `"The man"`, it can also include a new line or a tab. Outside a quoted string, these act as separators. Thus,

`The man`

is a sequence of *two* constants.

ASCII control characters between 1 and 31 can also be included in a quoted constant. These can often be used to invoke certain special functions when output to the screen. (See appendix B for details.)

In a quoted constant @ acts as an escape character causing micro-PROLOG to interpret the next character in a special way. To enter @ in a string you must use @@. Thus, "@@" is the constant comprising one character: an @.

An @ character followed by the quote character means a quote character in the string.

`"A quoted @" string"`

has the text:

```
A quoted " string
```

Although constants can be of any length, only the first 60 characters of a constant are significant, and only these 60 characters are stored. This applies to quoted constants as well as other kinds of constant.

2.4 Variables

Variables are represented by alphanumeric names which consist of a single letter optionally followed by a contiguous positive integer which has the role of a subscript. 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 contiguously followed (if at all) only by digits, whereas `yes`, `x12c` are *not* variable names. The integer 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.

Warning
`x` and `x0` are the same variable because `x` is implicitly subscripted with `0`. Similarly, `x3` and `x03` are the same variable because `3` and `03` are the same integer subscript. The subscript is not the digit sequence but the positive integer that the sequence denotes.

When a term is read in, and a variable is recognised in the term, the variable is converted into a special internal form and the actual variable name used is discarded. All occurrences of the same variable name in the term are converted into the same internal form. However, when the term is printed the original name of the variables are not available. Instead, the variables in the term are displayed with names taken from the sequence `X`, `Y`, `Z`, `x`, `y`, `z`, `X1`, .., `z1`, ... The first variable in the term is given the name `X`, the second is given the name `Y`, and so on.

2.4.1 A note on separators

Variables, constants and numbers are generally separated by one or more of the separator characters: space, carriage-return/new-line, tab. The actual number of separators is not important.

See section 2.10 for a fuller description of micro-PROLOG's lexical syntax. A separator character is not always necessary in order for micro-PROLOG to determine the end of one syntactic object and the beginning of another. On the other hand, insertion of a separator never does any harm.

2.5 Lists

Lists are the structured objects of micro-PROLOG. The simplest list is the empty list

```
()
```

which is written as a left (round) bracket followed by a right (round) bracket (separated by any number of spaces, including none). If there are terms inside the brackets we have a non-empty list. Thus

```
(2 3 5)
```

is a list of the three integers *2*, *3*, *5*. The spaces between the integers are important. The list

```
(235)
```

is the list of one integer *235*.

The objects in a list can be any term, including another list. For example,

```
((2 3) apple x (-2.3 &&))
```

is a list of four objects:

a sublist `(2 3)` of the two integers *2*, *3*
the constant `apple`
the variable `x`
a sublist of two objects: the floating point number `-2.3` and
the graphic constant `&&`.

As in LISP, sublists can be nested to arbitrary depth, e.g.

```
((a (b c)) ((d)) ((e) f) h))
```

is a list of three sublists.

2.5.1 The list constructor ;

Internally, non-empty lists are represented as structures built up from the empty list by a sequence of ;-constructions. ; is the micro-PROLOG list constructor analogous to the LISP dot. For example,

```
(a b c)
```

is represented internally as

```
(a ; (b ; (c ; ())))
```

The innermost expression `(c ; ())` represents the list constructed from the empty list by adding `c` as a front element. That is, `(c ; ())` is the internal representation of the single element list `(c)`. So `(b ; (c ; ()))` is the list `(c)`

with *b* added as a front element. In other words, it is the internal representation of the list *(b c)*. Finally,

(a | (b | (c | ())))

is the list *(b c)* with *a* added as a front element, which is the list *(a b c)*.

The notation *(t1 t2.....tn)* for a list of *n* terms is just an accepted abbreviation for the explicitly constructed list

(t1 | (t2 | ....(tn | ())....))

More generally,

(t1 t2....tn | t)

is an accepted abbreviation for the explicit construction

(t1 | (t2 | .... | (tn | t)....))

when *t*, as well as each of *t1 .. tn*, is any term i.e. a number, constant, variable or list. The | should be read as: followed by. The above term is the list of the k elements *t1, t2,..,tk followed by t*.

Examples

(a b c | x)

is the list comprising the sequence *a, b, c followed by x*. A list term of the form

(t1 | t2)

is the list comprising the term *t1 followed by t2*. In LISP parlance, *t1* is the head (or CAR) of the list and *t2* is the tail (or CDR) of the list.

Not all keyboards can generate the vertical bar character (|), and so micro-PROLOG contains an alternative list constructor which is two full stops **..**. This is a special reserved graphics token (see 2.10 for more information about tokens). Examples of its use are:

(john mary .. X) is equivalent to: *(john mary | X)*
(nice going ..) is a syntax error: *(nice going |)*
(nice going ...) is a list of three constants.

Note that micro-PROLOG does not store the original text of your program, and when you list a program any dot–dot list constructors will be replaced with vertical bars.

Note: Throughout this manual | vertical bar is used for documentation purposes but remember **..** can always be used as an alternative.

2.5.2 List patterns

Any list term with variables is a list pattern. The above

```
(a b c | x)
```

is a list pattern representing any list that begins with the constants *a*, *b*, *c* in that order. Since *x* may be the empty list, the list `(a b c)` is covered by this pattern. List patterns have a crucial role in micro-PROLOG. Relations over lists are defined using list patterns. Further examples of list patterns are:

(i) `(x | y)`

represents any list comprising at least one element *x*. Again, since *y* can be the empty list, this includes the single element list `(x)`. Note that `(x)` is also a list pattern, representing any list of exactly one element.

(ii) `((x | y) | z)`

represents any list that starts with a non-empty list (the pattern `(x | y)`. Since *z* and *y* can both be the empty list, the list pattern `((x))`, denoting a singleton list with a singleton list as its only element, is a special case of the pattern `((x | y) | z)`.

(iii) `(x1 x2 (x3 x4))`

is a list of three elements, whose third element is a list of two elements.

(iv) `(x1 x2 | y)`

is a list of at least two elements *x1* *x2*, in that order. It covers the case of a list of exactly two elements since *y* may be `()`.

In each of the above patterns the variables represent any term, including lists. There are no types in micro-PROLOG over and above types of list structures represented by list patterns. Variables always represent any term.

(v) `(x1 x2 | (x3 x4))`

is the list of two elements *x1*, *x2 followed by* the list of two elements *x3*, *x4*. In other words, it is the list

```
(x1 x2 x3 x4)
```

which is an equivalent pattern. Both are represented internally as

```
(x1 | (x2 | (x3 | (x4 | ()))))
```

Although the bar is the only data constructor in micro-PROLOG, which is a major difference between it and other PROLOGs, other constructors can easily be simulated by using list notation. In most PROLOGs, any constant *f* can be introduced as a data constructor with *f*-structures denoted as terms of the form

`f(t1,...,tn)`

In micro-PROLOG, you can represent this, as in LISP, as the list

`(f T1 ... Tn)`

where `Ti` is the list representing the term `ti` and the terms are separated by spaces not commas.

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

Atomic formulae, or atoms, are the primitive sentence forms out of which program statements and queries are constructed. In the usual syntax of predicate logic an atomic formula is an expression of the form

`R(t1,...,tn)`

where R is a relation name (more formally, *predicate symbol*) and `t1,..,tn` are its argument terms. In micro-PROLOG, the atomic formula is written as the list

`(R t1 .. tn)`

comprising the relation name followed by the sequence of arguments. There are no separating commas. Spaces are used if needed (see section 2.10).

Example

`(father-of tom bill)`

is the atom which expresses the relation of `father-of` between `tom` and `bill`.

The relation name of an atom, that is the head of the atom list, must be a constant. (But see section 2.9.)

2.7 Clauses

All PROLOG program statements correspond to a restricted form of sentence of predicate logic called *Horn implications*, or *definite clauses*. These are sentences of the general form

<atom> if <atom1> and <atom2> and...and <atomk>, k> 0

in which each variable appearing in the sentence represents any term. In the micro-PROLOG standard syntax the logical connectives `if` and `and` are

dropped, and the clause becomes the list of atoms

`(<atom> <atom1>...<atomk>)`

Since each atom is itself a list, a clause is a list of sublists in which each sublist is a list that begins with a constant which is a relation name. (But see section 2.9 for exceptions.) The first atom is the *head* of the clause, the remaining atoms comprise the *body* of the clause.

Examples

(i) `((father-of tom bill))`

is the unconditional statement that the atom

`(father-of tom bill)`

represents a true fact.

(ii) `((App () x x))`

is the unconditional statement that for all `x`

`(App () x x)`

is an instance of the Append relation.

(iii) `((App (x | y) z (x | y1)) (App y z y1))`

is the conditional statement that for all x, y, z and y1

`(App (x | y) z (x | y1))`

is an instance of the App relation if

`(App x y y1)`

is an instance of the App relation. These are both true statements when `(App x y z)` is understood to mean that `z` is the result of appending (concatenating) a list `x` to the front of a list `y`. Note the role of the patterns `(x | y)` and `(x | y1)` in the second clause. They tell us that the clause deals with the case of a non-empty front list and the repeated `x` in the two patterns tells us that the first element of the front list becomes the first element of the concatenation.

2.8 Comments

Since micro-PROLOG is first and foremost an interactive system, comments in the form of text in a source file that is ignored on loading are not supported. However, comments can be inserted in a program as statements about some `comment` relation, or as comment conditions in a clause.

2.8.1 Comment conditions

The built-in relation `/*` is a multi-argument relation that is always true. Thus an atom with `/*` as relation name, followed by any term or sequence of terms, is always skipped over by the micro-PROLOG interpreter.

Example

```
((App () x x)
  /* App is the concatenation relation for lists))
```

is a commented assertion about the `App` relation. It will be slightly slower in use than the uncommented assertion.

2.8.2 Comment clauses

The relation name used for comment clauses can be chosen by the programmer, however it cannot be `/*`. No clauses are allowed that make statements about the primitive relations of micro-PROLOG. An attempt to enter such a clause results in an error.

Suppose the programmer uses the relation name `comment`. The clause

```
((comment App (the concatenation relation for
lists))
```

adds the same information about the `App` relation. The main difference is that it is not embedded in the `App` program and so is not automatically shown in a listing of `App` program. The comment for `App` must be retrieved by either listing the `comment` relation or by querying the `comment` relation. We deal with listing and querying in the next chapter.

2.9 Meta-variable

As an extension to the clausal syntax described above variables can appear in certain positions in a 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 the 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 become 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 expressed succinctly in micro-PROLOG.

When the micro-PROLOG interpreter makes use of a clause containing a

meta-variable the variable must have been bound by the time that the meta-variable condition is reached. Moreover the term binding must be a valid syntactic item for the position of the meta-variable in the clause.

So, if the meta variable replaces a relation name, it must have been bound to a constant; if it replaces an atom it must have been bound to a list that is an atom; if it replaces the remainder of the body, it must have been bound to a list of atoms. If this is not the case, an error is signalled.

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. (The head atom of a clause must always have a constant as the predicate symbol. This names the relation that the clause is about.) A predicate symbol meta-variable must be bound to a constant before the atom is evaluated. The constant is taken as the predicate symbol of the atom for this call.

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 y)` is true of each element `y` of `<list>`. The `x` argument, the relation to apply to each element of the list, must be given in any call to `Apply`.

```
((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 instead of 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'.

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 in ALGOL [Naur 1962].

2.9.3 Meta-variable as the tail of the body of a clause

The final variant of the meta-variable is its use as the tail of the body of a clause or as the entire 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 can 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 disc 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 `(`, `:` and `)` 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 tokens.

2.10.1 Token boundaries

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 tokens below).

2.10.2 Special tokens

Special tokens consist of single characters, called special characters. Three special tokens form part of the list syntax of micro-PROLOG:

```
( ) |
```

while the others are always regarded as single character constants by micro-PROLOG (even though they are also bracket characters):

```
[ ] < > { }
```

The use of one of these special characters always marks a token boundary. They do not need to be preceded or followed by any separator.

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 father-of A-1 -A
```

An alpha-numeric token must be separated from a preceding alpha-numeric token and from a following number token by at least one separator.

2.10.4 Number tokens

Numeric tokens are tokens which denote integers or floating point numbers. They are described in section 2.2.

A numeric token must be separated from a preceding number or alpha-numeric token and from a following number token.

2.10.5 Graphic tokens

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

```
- = : :: ?? /+ £-% &
```

Graphic tokens need only be separated from preceding or following graphic tokens. The token .. is a reserved graphics token (see 2.5.1).

2.10.6 Quoted tokens

The final kind of token is the quoted token. 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. A quoted token is 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 be inserted in the token by prefixing it with the escape character @. (See section 2.3.)

Quoted tokens do not need to be preceded or followed by a separator.

2.10.7 The lexical rules

The parser in micro-PROLOG has the fairly simple task of recognising tokens, converting tokens into variables, numbers and constants and constructing lists out of the token sequences that begin with (and end with).

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

Numbers Constants

```
| | | |
| | | |
( 1 2 ( $ "A string" ))
\ | //
\ | //
```

Special tokens

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 by 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:

Alpha-tokens

Variables constants

x x123 yes zoo

Prefix digits letters
char

In standard micro-PROLOG the variable prefix characters are x, y, z, X, Y and Z. Finally, when the parser encounters a (it constructs a list out of the sequence of terms parsed from the following token sequence that terminates with the corresponding).

3 The micro-PROLOG supervisor

Micro-PROLOG is an interactive system. The top level interaction with the user is controlled through a special built-in micro-PROLOG program called the supervisor.

The micro-PROLOG supervisor provides a simple operating environment for the user. It allows programs to be entered, executed, edited, saved and loaded on disc files using various micro-PROLOG primitive relations as commands. In this chapter we describe the user interface to the supervisor and we illustrate the use of several primitives of micro-PROLOG that serve as useful supervisor commands.

The supervisor is an integral part of the micro-PROLOG interpreter, and is called as soon as you enter micro-PROLOG. It is a non-terminating program which controls all your interactions with micro-PROLOG.

3.1 Entering clauses and issuing commands

At this stage we suggest that the reader refers to Appendix A for details of how to enter micro-PROLOG. He can then follow the examples of this chapter using a computer.

As mentioned in appendix A the

```
&.
```

prompt that you get when you enter micro-PROLOG comprises a & printed out by the supervisor to tell you that it is ready to accept a clause or a command. The . is then the read prompt displayed by the keyboard read primitive of micro-PROLOG. It is displayed because the supervisor, which is a micro-PROLOG program, immediately tries to read user input from the keyboard.

The supervisor accepts two kinds of user input. It accepts clauses or single argument commands. A command comprises the name of some single argument relation (a *unary* relation) followed by its single argument. The single argument relation can either be one of the primitive single argument relations of micro-PROLOG, all of which are fully described in chapter 7, or it can be a user-defined single argument relation. The supervisor does not distinguish between the two kinds of relations. The effect of this is to allow the user to define new supervisor 'commands'.

3.1.1 Entering clauses

To enter a clause you simply type the clause in response to the *&.* prompt and then press the **RETURN** key. The clause is added to the program at the end of the sequence of clauses currently defining the relation of the head of the clause. For example:

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

There is no supervisor command to enter a clause at a position other than at the end of the current sequence of clauses for its relation. To do this, you have to use the primitive *ADDCL* relation in a query command (see section 3.2 below), or the clause EDITOR (described in chapter 4), or the MICRO extension to the supervisor (chapter 6).

Notice that the last clause is entered over two lines and that at the beginning of the second line the prompt is *1.* instead of *&.*. The *1.* prompt is issued by the keyboard read primitive because it knows that the term that is the clause for *App* is not yet finished even though the **RETURN** key has been pressed. It knows this because the brackets of the first line do not balance. There is one right bracket to come. The *1* of the prompt tells you it is waiting for this one extra right bracket.

For more information on the entering of clauses, or any list term, over several lines, we refer the reader to appendix B.

The general format of a supervisor command is:

```
&.<Command name><Command argument>
```

where the command name is a constant, and the command argument is a term whose exact form is dependent on the actual command. The command argument is what would be the single argument if the command name were used as a unary relation in a program. The supervisor knows when you have entered a command because the first thing it reads is a constant which is the command name. If the first thing you enter in response to the *&.* prompt is a list, it assumes that this is a clause and tries to add it to your program using the *ADDCL* primitive. If the list term does not satisfy the syntactic constraints of a clause you will get the *Error 4* error message and the entered list will be ignored. If the first thing that you enter is a number the supervisor will display a *?* response and ignore the number. *?* is the supervisor response that you will also get whenever a command fails.

3.2 The query command ?

Micro-PROLOG programs can be queried using the primitive query relation ?. This takes as its single argument a list of atoms which represents a conjunction of conditions to be solved. The way in which micro-PROLOG tries to find a solution to the conjunction of conditions is fully described in the introductory book.

If a solution to the ? query is found then the supervisor displays its normal prompt:

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

The query to show that someone *x1* is the parent of someone *x* succeeds.

If a solution cannot be found, i.e. the query fails, then a ? is displayed before the next prompt:

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

The query to show that someone *x1* is the parent of someone *x2* and that the found *x2* is also the parent of the found *x1* has failed.

Micro-PROLOG does not automatically print any response if the evaluation of the ? query succeeds. If you want to see the values of variables for the found solution you must add an extra *PP* condition/call to the list. For example, to find the name of a parent of John use the query:

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

PP is a built-in multi-argument relation that always succeeds with the side-effect of displaying its sequence of arguments at the screen.

In chapters 5 and 6 other forms of query are described that automatically display the solutions.

3.2.1 Interrupts and errors

To break into an execution the **ESCAPE** key is used (see appendix B to determine what the **ESCAPE** key is on your computer). When this is detected, Break (error 11) is signalled and the execution is interrupted. Appendix E gives a complete list of the signalled error conditions of micro-PROLOG.

Section 3.7 below describes how a signalled error can be trapped by a program for the reserved relation name *"?ERROR?"*. It gives an example definition of

the relation which provides a program that displays a short description of the error together with the call being evaluated when the error occurred.

3.2.2 Other uses of the query command

The primitive relation *ADDCL* can be used in a query command to add a clause at a specified position in the current sequence of clauses for its relation.

Thus,

```
?((ADDCL ((Parent John James)) 1))
```

will add the clause

```
((Parent John James))
```

after the current first clause for the *Parent* relation. Since we already have

```
((Parent Mary John))
((Parent Peter John))
```

the new clause will be inserted between these two clauses.

Individual clauses can be deleted using *DELCL*. This has a unary form and a binary form. In the unary form the single argument is the clause to be deleted. In this form it can be used as a command, e.g.

```
DELCL ((Parent Peter John))
```

In the binary form its arguments are a relation name and the position of the clause to be deleted. In this form it must be used inside a query command, e.g.

```
?((DELCL Parent 2))
```

deletes the current second clause for the *Parent* relation.

Finally, individual clauses can be edited using the *ED* primitive, e.g.

```
?((ED parent 2))
```

will cause the second clause for *Parent* to be displayed ready for copy-editing using the cursor keys of your computer (see appendix B). When you have constructed the new version of the clause (after the read prompt that followed the displayed clause), hit **RETURN**. The new version will replace the old clause.

If you want to abandon the edit, hit the **ESCAPE** key (see appendix B).

See chapter 7 for a complete description of the *ADDCL*, *DELCL* and *ED* relations. See also chapter 4 for an explanation of how to add and delete and reposition clauses using the EDITOR.

3.3 User-defined relations as commands

As we remarked earlier, any unary relation can be used as a command including relations defined by our own micro-PROLOG programs. This allows us to define new 'commands' to the system. For example, suppose that we have added the clause:

```
((Show x) (? x)(PP x))
```

which uses the supervisor query command *?* as an ordinary unary relation. By using *Show* as a command relation with a command of the form:

```
&.Show (<atom1> <atom2> .. <atomk>))
```

the *Show* program is invoked as though it was called with query:

```
?((Show (<atom1> <atom2> .. <atomk>)))
```

It evaluates the list of atoms and then prints the list in its solved form, i.e. with variables replaced by their answer bindings.

The *?* command that we saw above is itself defined as a unary relation by the clause:

```
((? X)↑X)
```

This clause uses the meta-variable feature described in section 2.9 in which the body of the clause is replaced by the value of the variable *X*. Many other supervisor commands, such as *LIST*, are themselves just built-in micro-PROLOG programs for unary relations.

3.4 Multi-argument commands

Sometimes it is convenient to enter a command which has several arguments. One way to do this is to define it as a unary relation that takes a list of its parameters as its single argument. *?* is rather like this. Its single argument is a list of any number of conditions to solve. Alternatively, one can define the command as a program for a unary relation that immediately reads in its extra arguments.

As an example,

```
((addcl X)
  (R Y)
  (ADDCL Y X))
```

is a program for a relation that can be used as though it were a two argument command. An example use is:

```
&.addcl 1 ((Parent John James))
```

When executed, the first argument, the 1, is read in by the supervisor and becomes the single argument to the `addcl` call. This invokes the single clause for the relation which immediately reads in the next term (the clause to be added) using the primitive `R` relation. Then the two argument `ADDCL` is called with the position `X` and the clause `Y`. Compare this with the direct use of `ADDCL` using `?` that we gave earlier.

3.5 Basic supervisor commands

3.5.1 The LIST command

The micro-PROLOG clauses that you have entered 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.

```
LIST ALL
```

lists the entire program.

```
LIST <relation name>
```

lists all the clauses (in the current program) for the named relation. Finally,

```
LIST (<relation name1> ... <relation namek>)
```

lists the programs for each of a list of named relations.

```
&.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))
&.
```

The variables that appear in a listed clause will usually be different from the ones you used when you entered the clause. This is because, as we noted in section 2.4, variables are converted into a special internal form when a term such as a clause is read in. The name of the variable is lost, only its positions within the term are remembered.

When the term is listed, each variable is given a print name from the sequence of names `X`, `Y`, `Z`, `x`, `y`, `z`, `X1`, ..etc. The first variable in the term is displayed as `X`, the next as `Y` and so on. That is why the second `App` clause is listed as above.

Note: see appendix B for information on how to pause listing to the screen and how to direct output to a printer.

3.5.2 LOAD and SAVE commands

A user program can be saved onto a disc file, and subsequently loaded back into the workspace. The two supervisor 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>
```

The <file name> must be different from that of any program relation and different from the name of any currently loaded module. If it is not you will get a signalled error on trying to obey the command. If you get an error during a *LOAD* you should *CLOSE* the file with a

```
CLOSE <file name>
```

command because the file which was automatically opened for the *LOAD* will have been left open.

The *LOAD* command reads a program from the file specified and, if the program is not a module, it 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: each new clause is added to the end of the current sequence of clauses for its relation. In this way the programmer can have a library of programs, on a number of different files, which are loaded in as required 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, i.e. all the clauses that are listed in response to a LIST ALL command. A subsequent *LOAD* of the file containing the saved program will add the program to the workspace.

To save the programs for named relations only you need to use the two-argument form of *SAVE* in a query command.

```
?((SAVE FAMILY (Parent Male Female)))
```

will save all the clauses for the relations *Parent*, *Male* and *Female* in the file FAMILY of the currently logged in drive. The second argument to *SAVE* is a list of the relations to be saved.

SAVE does not delete any program from the PROLOG workspace. To do that you need to use *DELCL* or the *KILL* command described below.

However, if you *SAVE* a file with the same name as one already existing in the filing system, then the previous version will be lost.

On some computers you can use the commands of the filing system to delete or rename a file or to see what files you have, without leaving micro-PROLOG. See appendix C for details of this.

3.5.3 File names

A file name is a constant which describes a file in the style of the disc system you are using. Please consult appendix D and also your disc system documentation for more details of the conventions required. Note that quotes are needed around any name that is not recognised by micro-PROLOG as a single token (see 2.10).

But beware the problem mentioned above; your file name as given in the micro-PROLOG *LOAD* or *SAVE* command must be different from that of any relation or module name. Using only upper case letters in a file name helps to avoid that problem.

3.5.4 The KILL command

The *KILL* command can be used to delete all the clauses for a particular relation or list of relations.

```
KILL Parent
```

deletes all clauses for *Parent* and

```
KILL (Parent App)
```

deletes all clauses for *Parent* and *App*.

```
KILL ALL
```

will delete all clauses in the current workspace. It should be used with care, and usually only after the use of *SAVE*. (However, it will **not** delete any LOADed modules.) A final form of use of *KILL* is

```
KILL <module name>
```

This does get rid of a loaded module.

3.5.5 Exiting the system

To exit micro-PROLOG either switch the machine off making sure any discs have been removed first or use your machine specific exit primitive described in appendix C. When you switch it on again, you will have to follow the start up procedure described in appendix A.

3.6 The dictionary relation

All constants that are used in a user program are stored in a dictionary and uses of the constant in the program become pointers into the dictionary. So do not worry about using long relation names. You can see this dictionary by executing

```
LIST DICT
```

What you will see is something of the form

```
((DICT & () (...) App Parent John .............)
```

which is a clause for the relation *DICT* which is a multi-argument relation. The first three arguments are:

1. the constant *&* which is part of top-level *&.* prompt and is the name of the workspace area,
2. the empty list,
3. a list that will be empty unless you have loaded any modules, if you have, the second list will be all the names exported by currently loaded modules.

The remaining arguments are all the constants used in the workspace program.

The workspace name *&* is there because this is also the form in which you will see the dictionary of a module once the module has been opened (see 7.11). *&* is the name of the special root module which is the workspace area. The first argument of *DICT* is always the name of the current module. For any module other than the root module the second argument will not be the empty list. It will be the list of all the exported names of that module.

The garbage collector clears constants from the workspace dictionary once there is no longer a reference to them in the program.

3.6.1 The SDICT relation

About 80 constants never appear in the *DICT* relation: these are the system constants which comprise the names of micro-PROLOG's built-in programs, supervisor commands and special files. As with user constants, the system constants are maintained in a dictionary.

The *SDICT* relation contains the full list of these constants, which can be seen by executing:

```
LIST SDICT
```

... which will result in:

```
((SDICT NUM DEF VAR ... FORALL))
```

The 80 or so arguments of *SDICT* are the system constants. System constants are all those and only those constants of which the relation *SYS* is true.

3.7 Trapping error conditions by micro-PROLOG programs

Micro-PROLOG has relatively few error conditions, and most of them can be trapped by a micro-PROLOG program. The errors are divided into two groups – the numbered errors and the message errors. All the numbered errors can be trapped. (See appendix C for a complete description of the numbered and message errors.)

When a numbered error occurs, if there are clauses for the relation *"?ERROR?"*, the program for the relation is called with a call of the form:

```
("?ERROR?" <error number> <error call>)
```

This call replaces the erroneous call which is passed as the second argument, this is the call that was being evaluated when the error occured. Thus, if the call to the *"?ERROR?"* program succeeds, possibly binding variables in its error call second argument, the error call is assumed to have succeeded and the evaluation continues with the next call. If the call to the *"?ERROR?"* program fails, the error call is assumed to have failed. The call to the error handler may also result in an *ABORT* to the supervisor.

If there are no clauses for the *"?ERROR?"* relation, the message

```
Error: n
```

where *n* is the error number, is displayed. The current evaluation is then aborted and you are returned to the supervisor.

The following program is an example of a simple error handler which just reports the error with a short message and then aborts to the top-level supervisor:

```
 (("?ERROR?" X Y)
  (P-code X Z)
  (P Z)(PP Y)
  ABORT)
((P-code 0 "Arithmetic Overflow or Underflow"))
((P-code 2 "Relation Not Defined"))
((P-code 3 "Too many variables or invalid form"))
((P-code 4 "Relation Protected"))
((P-code 5 "File Handling Error"))
((P-code 11 "Break!"))
((P-code 12 "Module Handling Error"))
((P-code X (Error X)))
```

A simple alteration will allow the error 2 (which occurs when attempting to execute a program for which there is no definition) to be treated as a failure. All that is needed is to add the following clause to the *"?ERROR?"* program:

```
(("?ERROR?" 2 x)
/
FAIL)
```

This must be placed before the general *"?ERROR?"* clause. The */* followed by the *FAIL* causes the error call given as the *x* argument to fail.

A further refinement of this is to treat error 2 as a failure only if the clause *((data-rel <rel>))* in the user's program declares that it should be so treated. The extra clause for *"?ERROR?"* is then

```
(("?ERROR?" 2 (X|Y))
  (DEF data-rel)
  (data-rel X)/
  FAIL)
```

The error handlers which are defined in the SIMPLE and MICRO front end programs (*errmess-mod*, *deftrap-mod* and *errtrap-mod*) all treat error 2 in this way. The use of *DEF* prevents another error 2 arising should there be no *data-rel* clauses. See the description of *DEF* in chapter 7 of this Manual.

It is generally a good idea to have some definition for *"?ERROR?"* present. The ERRTRAP file contains a module that exports a definition for this relation. It provides a sophisticated error trap and recovery utility. Its facilities are described in the MICRO chapter, chapter 6. However, you can load and use it separately from the MICRO system.

3.8 The user-defined supervisor

A significant feature of the supervisor is the ability of the user to redefine the supervisor program which runs for the duration of a micro-PROLOG session. It enables the user to have complete control of the system, even after *Space Error*, *Dictionary Full*, or where the ABORT primitive has been invoked.

3.8.1 The "<SUP>" relation

To explain the use of this facility, a few words need to be said about the actual default supervisor. This is a program, written in micro-PROLOG, which performs a very simple loop indefinitely. Called *"<SUP>"*, its definition is shown below:

```
(("<SUP>")
  (DEF "<USER>")
  ("<USER>")
  FAIL)

(("<SUP>")
  (CMOD Y)
  (P Y)
  ("<R>" X)
  ("<>" X)
 /("<SUP>"))
```

The second clause is this loop. *CMOD* is used to find the name of the current module (initially &), and this is printed by *P*. *"<R>"* is a special form of the *R* predicate which is fail-proof, and merely reads a term into X. The *"<>"* predicate processes this term as follows:

```
(("<>" X)
  (CON X)
  (R Y)
  (X Y))

(("<>" (X.Y))
  (ADDCL (X.Y)))

(("<>" X)
  (PP ?))
```

If the term read by *"<R>"* is a constant, the first clause of *"<>"* reads a second term and uses the meta-variable facility of micro-PROLOG to run the terms as a query or command. If *"<R>"*'s term was a list (X . Y), the second clause attempts to add it as a program clause. If either of the first two clauses fail for any reason, or if the term returned by *"<R>"* was a number or a variable, then the third clause of *"<>"* succeeds by printing the fail message *?*.

3.8.2 The "<USER>" relation

The first clause of *"<SUP>"* (see above) is what provides the user supervisor facility. On each loop of the *"<SUP>"* program, this first clauses tests to see whether a program called *"<USER>"* is defined. If not, it fails and the default loop described above takes over. As soon as *"<USER>"* is defined, however, it is called and takes over from the default supervisor.

As with *"<SUP>"*, any program called *"<USER>"* should be a tail-recursive loop (see appendix F or Chapter 7 of [Clark & McCabe 1984]). If *"<USER>"* does fail or succeed, no harm is done, but the first clause of *"<SUP>"* fails, and the default second clause is entered for at least one iteration. This provides

enough time to edit the bad *"<USER>"* program or to kill it.

A simple example of the use of *"<USER>"* is given below:

```
(("<USER>") {there are no arguments}
  (SPACE x)
  (P x kb Free:)
  ("<R>" y)
  ("<>" y)
 /("<USER>"))
```

This performs a loop analogous to the default supervisor, except that it displays the amount of free space left as its prompt, rather than the current module name.

Warning
The *"<USER>"* program is always entered in place of the default supervisor, even after any error short of *System Abort*. If the definition of *"<USER>"* contains a micro-PROLOG error, or if *"<USER>"* loops without allowing user interaction, then the user no longer has any control of the system.

For example, the following bad definition:

```
(("<USER>")
  (SUM X Y Z))   {a Control Error}
```

will cause micro-PROLOG to generate error messages forever, since after each error message aborts the execution, *"<USER>"* is again entered and the error occurs a second time.

Again, the definition:

```
(("<USER>")
 ("<USER>"))
```

will loop forever, and no error conditions, including the **ESCAPE** 'Break!' can abort the loop. You will have to switch the machine off and start again.

3.8.3 Turnkey systems and security

Two main reasons exist for including a user supervisor option. The first relates to the ability to write turnkey systems. If a program is loaded from disc, and it includes a definition of *"<USER>"* such as:

```
(("<USER>")
   (run rest of system){or any other start command}
   (PP Goodbye)        {print log-out message}
   (loop)
((loop) (loop))
```

then the program will run automatically, and any ABORT error conditions will result in the system being run again from the start. When the run is complete, the program will sit in an infinite loop until the machine is switched off.

Related to the above is the ability to provide secure systems where the user cannot break in and look at the clauses created by the system while it was running. If the start-up command for the system is of the form:

```
((start up)
   (ADDCL (("<USER>")
            ((PP Private Program - Aborted)
            (loop)))
   (carry on))
```

then once the program is under way, any error, including *Space Error*, *Dictionary Full*, and **ESCAPE**, will leave the user with the bland message, and unable to do anything more. Data structures or clauses created by the system while running can therefore remain private.

4 Utility modules

4.1 Tracing execution

A trace program is provided with the system as a module with the name *trace-mod* in the file TRACE. The trace module allows, interactively, selective tracing of the execution of a query.

To use the trace program execute a *LOAD TRACE* command. This will load the module that is in the file. To get rid of the program when you have finished tracing do a *KILL trace-mod*. Note the asymmetry. To bring in a module you do a *LOAD* using the name of the file that contains it. To delete it you do a *KILL* using the name of the module. Nearly all the modules provided on the distribution disc have a name of the form: *<name>-mod* where <NAME> is the name of the file that contains them.

The module exports the relation name *??*. This is used in exactly the same way as the supervisor *?* query command, e.g.

&.??(<atom1>..<atomk>

The difference is that you are led through the evaluation of the query step-by-step, a call at a time.

When entering a call for the first time a message of the form

<call id>(R <seq. of args of call>)

is displayed at the console. The term printed represents the call/condition just before any attempt at evaluation, with all of the known values of variables substituted in place. The <call id> is a list of numbers identifying the ancestry of the call back to the original query. The length of the list corresponds to the depth of the evaluation. As an example, the list

(1 3 2)

identifies the call as the first condition of the body of a clause invoked to solve the third condition of the body of a clause invoked to solve the second condition of the original query.

If the relation name *R* of the call refers is one of the micro-PROLOG primitives (all described in chapter 7) then the call is immediately executed. (The logical primitives *IF*, *OR*, *NOT*, *ISALL*, and *FORALL* are an exception. These can be traced – see below.) If the call is for a user-defined relation then the message

trace? is also printed. One of the allowed trace responses must then be entered.

The trace responses allow selective tracing of the program. For example, low level (or already debugged) programs can be skipped by responding:*n*, i.e. no tracing. The allowed trace responses are:

```
n(for no), y(for yes)
s(for succeed), f(for fail),
q(for quit), /(followed by any command)
```

1. **n**

If *n* is entered the call is executed without tracing. In this case one of two things normally happens: either the call is solved or the attempt to solve it fails. If it fails then the message

```
<call id> failing (R <seq. of arguments of call>)
```

is displayed at the console, and the system backtracks. This failure may cause backtracking on calls that were previously solved if there are no untried ways of solving these calls. Each time there is backtracking on a call, the call is displayed with the *failing* message.

If the evaluation of a call succeeds then the message

```
<call id> solved (R <seq of instantiated arguments
of call>)
```

is displayed. In this case the call is printed with the answer bindings substituted, so that you can see the result of the evaluation just completed. If the call was the last in the body of a clause then the immediate ancestor of the call is also solved, in which case a *solved* message is displayed for it too.

After solving a call, remaining unsolved calls are entered and traced in turn. This continues until the last call of the original query is solved or until its first call is failed.

2. **y**

The *y* response allows the tracing of the evaluation of the call. The trace program looks for a clause to match the call. If no clause matches the call, the *failing* message described above is displayed. If there is a clause that matches (i.e. a clause whose head atom unifies with the call) the message

```
<call id> matches clause n
```

is displayed. The *n* is the position of the clause in the sequence of clauses for relation of the call.

If the clause that matches the call is an assertion, then since there are no atoms to trace inside the body of the clause, the call has been solved and a *solved* message displayed. Otherwise each of the atoms in the body of the clause is entered and traced in turn. On entering the body of any clause that matches the call the <call id> is extended by a first number that identifies the position of the call in the body of the clause. The clause being used is also identified by a message of the form

3. **s**

Entering *s* allows one to arbitrarily solve a call. The immediate response is a printing of the call with the *solved* message. The trace program does not try to use any clauses for the call, nor does it instantiate any variables in the call.

4. **f**

Entering *f* allows the user to arbitrarily fail the call, and to cause the evaluation to backtrack. The *failing* message for the call is displayed.

5. **q**

Entering *q* quits the trace evaluation. It is used to obtain a quick exit. You are returned to the supervisor.

6. */*

When you get the trace prompt *trace?* you can enter any command providing it is preceded by the escape symbol /. You can use this facility to LIST the clauses for the relation of the condition about to be traced e.g.

```
(1 2) (parent ...) trace?/LIST parent
.
. <clauses for parent>
.
(1 2) (parent ...) trace?
```

or to EDIT the program using the EDITOR described later in this chapter. The trace program insists that a legal trace response is entered. If an erroneous response is input the program displays the message:

```
ENTER y n s(for succeed) f(for fail) q(for quit) or
   / followed by a command
```

and prompts the user again. When you are asked whether you want tracing inside one of the logical operators *IF*, *OR*, *NOT*, *ISALL* or *FORALL* the allowed responses are only *y* or *n*. But this restriction is indicated by the prompt *trace?(y/n)* that you will get in this case. To this prompt any response but *y* is taken as *n*.

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

As we mentioned earlier, the trace program, being a module, will not be loaded into the user program workspace. So when you do a `LIST ALL` you will not see any of the trace program. (However, if you do a `LIST ??` you will be able to see the clauses for `??` since this is an exported relation of the module.) Loading the trace module does, however, reduce the amount of space available for user programs. It should therefore be deleted when you have finished using it with the `KILL trace-mod` command. You can always re-load it when required.

4.2 Spypoint tracing

Sometimes, all that you want to trace is the entry/exit behaviour of some of your programs. A utility module called `spytrace-mod` enables you to spypoint trace a program in this way. To use the module do a `LOAD SPYTRACE`. To get rid of it when you have finished, do a `KILL spytrace-mod`. It is a very small utility program and can be used when your own program is quite large and you have not got enough space left to LOAD and use the full TRACE program.

The module exports three relations: `spy`, `unspy`, `spying`. The first two are unary relations that are used as commands. The `spying` relation has two uses, as a command to turn the tracing on and off, and as a two-argument relation to give entry/exit information for a particular call.

4.2.1 The spy command

If you want to have entry/exit information for all calls for a user-defined relation <rel>, do a

```
spy <rel>
```

command. If you now `LIST <rel>` you will find an extra clause has been added to the front of its program. The `spy` command has added the clause.

```
((<rel>|X)
  (spypoints on)          {check that spy relations should be traced}
  (/)                     {execute / to prevent other clauses being
                           used to solve this call}
```

```
(spying <rel> X))        {evaluate (<rel>↕X) call using the other
                          clauses giving some trace information}
```

(The "{","}" bracketed comments will not appear.) If the *((spypoints on))* clause is not in the workspace, the user-provided clauses for <rel> will be used to solve the call in the normal way. If it is present, the call *(<rel>↕X)* is evaluated by the call *(spying <rel> X)* to the *spying* relation.

You cannot *spy* a micro-PROLOG primitive relation or any relation exported by a module. The attempt to do so will result in the "Cannot add clauses for" error being signalled as the *spy* command tries to add the extra clause.

4.2.2 The spying relation

This takes two arguments, the name of the relation *<rel>* of the call to be entry/exit traced and the list *X* of the arguments of the call.

The evaluation of

(spying <rel> X)

is equivalent to the evaluation of

(<rel>↕X)

except that some trace information is given as the call is evaluated.

On entry, a message of the form:

<call id> : <rel> <list X of arguments of call>

is displayed. The <call id> is an integer. The <call id> is increased on each call of *spying* and decreased if the call fails. During the evaluation the call to *<rel>* it is identified in subsequent trace messages by the <call id>.

As each user-supplied clause that matches the call is tried, the message

<call id> matches clause <clause number>

is displayed. The <clause number> is the position of the clause in the listing of all the clauses for *<rel>* including the clause added by the *spy* command. So the first user clause for *<rel>* has clause number 2.

If the call is solved using this clause the message

<call id> solved <rel> <list X of instantiated arguments>

is displayed giving any solution bindings for the call.

Finally, if the call fails, the message

```
<call id> backtracking on <rel> <list of original
arguments X>
```

is displayed. After this message, the <call id> will be re-used to identify a different *spying* call.

The *spying* relation can be used directly in user-supplied clauses and queries. For example, the query

```
?((spying father-of (tom X))(spying male (X)))
```

will give you trace information on the evaluation of the two calls even if *father-of* and *male* are not *spy* specified relations.

4.2.3 The spying command

The command

```
spying off
```

switches off the *spying* of all the *spy* specified relations.

The command

```
spying on
```

has the reverse effect. The default when you LOAD the SPYTRACE program is that spying is on.

4.2.4 The unspy command

The command

```
unspy <rel>
```

will remove the extra clause added for <rel> by the *spy* command. It stops all the automatic *spying* of calls to <rel>.

4.3 The micro-PROLOG structure editor

The micro-PROLOG structure editor allows the PROLOG programmer to edit programs within the micro-PROLOG environment. It is an 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.

At any moment the editor is focused on a *current term* which has an *immediate context*, which is the list that the current term is in. To act as an 'aide-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 and the context is the list of clauses for a relation) the editor prefixes the prompt with a number; this number indicating the position, within the list of clauses, 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. The commands allow changing the current term, changing the context, moving or deleting or modifying the current term, and inserting new terms.

4.3.1 Edit commands

Before using the editor it is necessary to LOAD the editor module (called `editor-mod`) using the command:

```
LOAD EDITOR
```

To delete it when you have finished use:

```
KILL editor-mod
```

The editor is invoked using the command:

```
EDIT <relation name>,
```

for example to edit the `likes` program enter:

```
EDIT likes
```

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

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

If there are no clauses for `likes` you will get

```
[0] No clause.
```

in which case the first thing you should do is add a clause using the *a* command. The editor offers a handy tool for building new programs as well as for modifying existing ones.

When the editor displays its prompt it is ready to accept an *edit command*, which at the top level can be any of the *i*(insert), *a*(append), *k*(kill), *f*(fix), *v* (vars), *n*(next), *b*(back), *m*(move), *c*(copy), *t*(text), *e*(enter) and *o*(out) commands. The edit commands are divided into two groups: those which move the current term pointer in the structure of the terms being edited, and those which change the terms in some way.

4.3.2 Cursor movement commands

There are four commands which can be used to walk over the program; these are *n*, *b*, *e* and *o*.

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. As an 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).
```

whereas at the top-level we get the next clause:

```
[1] ((likes John Mary)).n
[2] ((likes X Y) (knows X Y) (likes Y X))
```

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 becomes `No term`, (or `No clause` at the top level) indicating that it is not actually pointing to a term or clause. 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).
```

```
[2] ((likes X Y) (knows X Y) (likes Y X)).b
[1] ((likes John Mary)).
```

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.
```

```
[1] ((likes John Mary)).b
[0] No clause.
```

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. (You cannot get 'inside' a number, constant or variable using these structure move commands. To change these you must use the *t* (text) or the *s* (substitute) command.) The immediate context becomes the list term 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

```
[2] ((likes X Y) (knows X Y) (likes Y X )).e
(likes X Y).n
(knows X Y).n
(likes X Y).
```

Note what has happened as we stepped along the entered clause. As we reach each atom it is printed with the variables in the atom assigned print names from the list *X*, *Y*, *Z*, *x*, ...This means that sometimes it will appear that the editor has changed the variable names.

As we moved to the *(knows X Y)* atom of the clause we actually got the same names displayed by accident. When we moved to the *(likes Y X)* atom of the clause this is displayed as *(likes X Y)*. Within this atom, the *Y* that we saw when the whole clause is displayed is the *first* variable, so its print name is *X*. The variable we now *see* as *X* is the variable that we *saw* as *Y* at the whole clause level.

This change of names as we enter and then step along a clause can be a little disconcerting. It does not matter unless you want to change a variable in a clause. To do this you should use the *t* text edit command at the whole clause level or at least at a term level in the clause that covers all the occurrences of the variable you want to change. Alternatively, you should use the *f* and *v* commands described below to temporarily fix the variables of a clause as constants. Changing or adding a new variable is then achieved by changing or adding new *vars* constants.

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

```
(likes X Y).o
[2 ](likes X Y ) (knows X Y) (likes Y X)).o
Edit of likes finished
&.
```

The *o* command may fail if the entered term has been incorrectly changed. This will happen if on returning to the outer level the predicate symbol of the head of the clause has been changed to a variable or to the name of a primitive relation. When an edit command fails the editor responds with a *?* and re-prompts at the appropriate level. If you change the relation that a clause is about to another allowed program relation name you will be asked if you want the clause added to the program for the new relation; if you respond *yes*, you will be asked for the position for the clause. This should be a positive integer. The clause will be inserted after the current clause at that position.

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.

4.3.3 Edit change commands

There are nine commands which directly affect the current term; these are *i*(insert) a new term, *a*(append) a new term, *k*(kill) the current term, *s*(substitute) another term, *m*(move) the term, *c*(copy) a clause, *t*(text) edit the term, *f* fix the variables as *vars* constants in a clause and *v* introduce variables for the *vars* constants of a clause. The *c*, *f* and *v* commands can only be used at the top clause level and the *s* command is only available below the clause level.

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 onto 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 clause is for the relation currently being edited. If it is not, or the clause is incorrectly formed, the insert fails and the old term is re-displayed as the prompt.

```
[1] ((likes John Mary)).i((likes bill mary))
[1] ((likes Bill Mary)).n
[2] ((likes John Mary)).
```

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 term (or clause) to the left of the deleted term in the immediate context

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 screen editor. It displays the term on one or more lines of the console. See appendix B for details of using the screen editor on your machine. You are now expected to type in a modified version of the displayed text to replace the current term.

If you use *t* to modify a term which is a component of a clause be careful not to change the displayed names of any variables that also occur outside the term. If you leave the names unchanged then the link between these 'global' variables with their occurrences outside the term will be restored on exit from the text edit. This is possible because the primitive *RFILL* relation (see chapter 7) used by the *t* command remembers the print name it assigns to each variable (which has a unique internal name) before it puts the text of the term into the keyboard buffer. When it reads in the text it replaces the occurrences of these remembered print names with the original internal names of the variables. This re-establishes the link with the variable occurrences in the rest of the clause when the reconstructed term is inserted in the clause. Alternatively, you can fix all the variables of the clause as constants using the *f* edit command.

Because of restrictions on the use of *RFILL* (see appendix C for your computer) the *t* command below the clause level may only be able to correctly display terms that can be printed on one or two display lines.

6. The *c* (copy) command allows the copying of a clause. It can only be used at the clause level. It is very useful when building up a program in which two or more clauses differ only slightly. You insert one, then copy it using *c*. *c* places the copy after the copied clause and makes it the current clause. You then edit the copy to give the variant of the first clause.

7. The *m* (move) command allows you to move a term within the current context. At the top level it repositions a clause within the sequence of clauses for its relation. *m* takes one argument, an integer which gives the displacement to the right or left within the current context. A positive integer n moves the term to the right (forwards) n positions, a negative integer −n moves it to the left (backwards) n positions. You cannot move the term outside the current context. If you give too large a right move the term is simply placed at the right end, too large a left move puts it at the left end. The moved term remains the current term after the move.

```
[1] ((likes Bill Mary)).m 1
[2] ((likes Bill Mary)).b
[1] ((likes John Mary)).n
[2] ((likes Bill Mary)).m -4
[1] ((likes Bill mary)).
```

8. The *f* command allows you to temporarily fix all or some of the variables of a clause as constants. It can only be used at the top clause level and it can only be used once. (A second *f*, applied to the same clause before the effect of the first has been undone with the *v* command described below will fail.) Its form of use is

```
f (constant1 constant2 ... constantk)
```

The result is that the first k variables of the clause will be replaced by the k constants given in the list following the *f* and when the clause is displayed an extra atom

```
(vars constant1 constant2 ... constant3)
```

will appear as a new last condition of the clause to remind you which constants in the clause are really variables. To change a variable you now change the constant that has replaced the variable. To add a new variable, use a new constant and then record that it is a constant standing for a variable by editing the *vars* atom so that it includes the new constant.

Example

```
[2] ((likes X Y) (knows X Y) (likes Y X)).f
(person1 person2)
[2] ((likes person1 person2) (knows person1
person2)
(likes person2 person1) (vars person1 person2))
```

9. The *v* (vars) command reverses the *f* command. Again it can only be used at the clause level. It replaces all the constants given in the *vars* last condition of the current clause by new variables thoughout the rest of the clause. Each occurrence of the same *vars* constant gets replaced by the same new variable. After the substitution, the *vars* atom is removed from the displayed clause.

Example

We can continue the above example use of *f* in order to add a new condition which requires person2 to be female.

```
[2] ((likes person1 person2) (knows person1
person2)
      (likes person2 person1) (vars person1
person2)).e
      (likes person1 person2).a (female person2)
      (female person2).o
[2]  ((likes person1 person2) (female person2)
       (knows person1 person2) (likes person2
       person1) (vars person1 person2)).v
[2] ((likes X Y) (female Y) (knows X Y) (likes Y
X))
```

To add this extra condition without using *f* and *v* we would have to use the *t* command at the clause level to text edit in the new condition (female Y).

Exiting the edit without issuing a v command

If you exit the edit of the definition of a relation in which you have used *f* but you have not undone its effect with a *v* command you will still be able to use the clause in the normal way. If you *LIST* the program you will see that all the *vars* constants have actually been replaced by new variables as when you do a *v* but there will be an extra */** comment condition as the first condition of the clause. This will be of the form

```
(/* vars (X1 | constant1) .. (Xk | constantk))
```

where constant1 .. constantk are the *vars* constants of the clause that were given in the *vars* atom just before you exited the edit and X1 ... Xk are the variables that now appear in place of these constants.

Example

Suppose that in the above example we exit the edit before doing the *v*. If we list *likes* the second clause will be displyed as

```
((likes X Y)
   (/* vars (X|person1) (Y|person2))
   (female Y)
   (knows X Y)
   (likes Y X))
```

In fact, it was saved in this form even during the edit. The editor automatically converts clauses that have a first condition, which is a comment associating constants with variables, into the special form in which the variables are replaced by the associated constants when you are about to edit the clause. It also maps the */** comment into the *vars* last atom of the clause. The *f* command adds such a condition to the clause, relying on the editor to display it in the right form, and the *v* command simply deletes the comment. This means that if we now start to re-edit *likes* we will see the clause in the form in which it was last displayed by the editor.

```
EDIT likes
[1] ((likes John Mary)).n
[2] ((likes person1 person2) (knows person1
person2)
      (likes person2 person1) (vars person1 person2))
```

Finally, if we use the editor to develop programs we can even enter clauses which use constants instead of variables. All we need do is make sure that this special use of constants is signalled to the editor by the appropriate *vars* condition at the end of the clause.

Example

```
EDIT append
[0] No clause.a
((append () list list)
  (vars list))
[1] ((append () list list) (vars list)).a
((append (h|t) list (h|t1))
  (append t list t1)
  (vars h t list t1))
[2] ((append (h|t) list (h|t1)) (append t list t1)
(vars h t t1 list)).o
Edit of append finished
&.LIST append
```

```
((append () X X)
  (/* vars (X|list)))
((append (X|Y) Z (X|x))
  (/* vars (X|h) (Y|t) (Z list) (x t1))
  (append Y Z x))
&.
```

4.4 Editing modules

We refer the reader to section 7.11 for a description of modules and the problem of editing a user program that has been wrapped up as a module. Only workspace programs can be easily edited using the above structure editor, so what is needed is a utility that allows modules to be unwrapped into workspace programs, and then, after editing, allows them to be wrapped up again as modules.

The file MODULES of the distribution disc contains such a utility program as the module *modules-mod*. To use it do a *LOAD MODULES*. To get rid of it, do a *KILL modules-mod*.

The module exports two unary relations *wrap* and *unwrap* that are used as commands. Both commands use the currently logged in disc for writing files, so this disc should have enough space for a file of all the clauses in the module and it should not be write protected.

4.4.1 The unwrap command

To transfer the clauses of a module named *M* into the workspace for editing first *SAVE* any existing workspace program, then do a *KILL ALL* to clear the workspace, then do an

```
unwrap M
```

command. All the clauses owned by the module will enter the workspace along with an extra clause

```
((Module M <export list> <import list>))
```

which gives the name *M* of the unwrapped module and its export and import name lists. You can now edit or add to the clauses in the workspace using the structure editor described above. You can also edit the extra *Module* clause to change the name of the module or to change its export/import lists, but do not delete it. It is used by the *wrap* command to reconstruct the module.

The *unwrap* command uses the current logged-in disc to store a temporary file which it deletes when all the clauses are in the workspace.

4.4.2 The wrap command

When you have finished editing the *unwrapped* module do a

```
wrap <file name>
```

command. This command uses the information in the *Module* clause in the workspace to create a module containing all the other clauses in the workspace. It also saves the new module on the named file.

By explicitly adding a *Module* clause to a workspace program you can use *wrap* as a quick way of creating new modules.

4.4.3 The save-mods command

This is a command that allows you to save several modules in a single file. When the file is *LOAD*ed all the modules it contains will be loaded. Its form of use is

```
save-mods <file name> <list of module names>
```

e.g.

```
save-mods SUITE (first-mod second-mod third-mod)
```

will save the three currently loaded modules

```
first-mod second-mod third-mod
```

in the single file

SUITE

of the currently logged in disc.

You can also include in the list of modules to be saved names of relations and even lists of relations. The clauses for the named relations will be saved on the file in the positions specified between the saved modules.

e.g.

```
save-mods SUITE (first-mod rel1 second-mod (re2
rel3) third-mod)
```

will save the module first-mod followed by the clauses for the relation rel1 followed by the module second-mod followed by the clauses for rel2 and rel3 followed by the module third-mod. So when the program is *LOAD*ed from this file the clauses and modules will be loaded in this order.

4.5 Using external relations – the EXREL utility

Note that the *EXREL* facility depends upon the *SEEK* primitive of micro-PROLOG. Please check in appendix C that this is implemented for your computer. If it is not available then please ignore this whole section.

On occasions, especially when the definition of some of your relations comprises a lot of single atom clauses, there is very little space left when the entire program is loaded.

When space is at a premium, you can trade time for space and have some of your relation definitions reside in a specially created disc file. The file is created using the *external* command of the utility module *exrel-mod* which is in the program file EXREL of the distribution disc.

The *external* command saves all the current clauses for one or more relations in a disc file. It then deletes them from the workspace and adds a single new clause for each relation. This new clause 'gives' the segment of the disc file in which the defining clauses for the relation are to be found. During a query evaluation these defining clauses are automatically accessed from the file as and when required.

We now describe the commands and facilities of *exrel-mod* in more detail. It can be used to set up disc file data bases of micro-PROLOG relation definitions which interface with the query evaluation system in a transparent way.

To load the module do a *LOAD EXREL* command. To delete it do a *Kill exrel-mod* command. The module must be present whenever you want to query *external* relations. This is because it exports the definition of the interface program *RPRED*.

4.5.1 External

This has two forms of use:

external <file name> <relation name>

sets up a file containing the defining clauses for a single relation.

external <file name> (<relation name1> .. <relation namek>

sets up a file containing the current defining clauses for each of the list of relations. The disadvantage of packing several relations onto one file comes when we want to edit one of the relations – more on this later. The advantage is that it reduces the number of external files. Since the filing system only allows a small number of files to be open at any one time, only a small number of external files can be accessed during a single query evaluation.

The command deletes all the saved clauses from the workspace program. For each externally stored relation a single new defining clause is added. This links the relation with the segment of the file where the defining clauses were saved.

Example

Suppose that the workspace program contains a sequence of clauses of the form

```
((worker-ID <name list> <identifying number>))
```

e.g.

```
((worker-ID (Tom Jones) 2334))
((worker-ID (Bill J Bell) 2867))
.
.
((worker-ID (Jane Roberts) 900))
```

and a sequence of clauses of the form

```
((salary <identifying number> <dept> <annual
salary>))
```

e.g.

```
((salary 2334 invoicing 12000))
.
.
((salary 3456 design 14500))
```

and a conditional clause

```
((salary x y 8000) (LESS x 1000))
```

expressing the company 'rule' that all trainee clerks have an ID number less than 1000 and a fixed salary of 8000. (This is not meant to be realistic. The general rule is there to emphasise that any form of clause can be used in the definition of a relation that is made external. You are not restricted to relations defined only by single atom clauses.)

We can make the definitions for both relations external with

```
external EXFILE1 (salary worker-ID)
```

A file named EXFILE1.LOG will be created on the currently logged-in disc drive and all the current clauses for the two relations will be saved on the file. They are deleted from the workspace and replaced by clauses such as:

```
((salary:x) (RPRED EXFILE1 (0 72) (salary:x)))
((worker-ID x) (RPRED EXFILE1 (72 510) (worker-
ID:x)))
```

These are the clauses for *salary* and *worker-ID* that you will see if you now try to *LIST* these relations. To see the old defining clauses you have to use the *LISTEX* command described below.

If you now want to pose queries that will access these two relations do an *open EXFILE1* command. This is more fully described below. Basically, it opens the external file for access by *RPRED* (using the micro-PROLOG primitive *OPEN*) and records the fact that the file is open by adding an *opened* clause for the relation to the workspace.

4.5.2 The RPRED relation

RPRED is a relation which converts a query condition for an external relation into a search through the clauses saved on a file segment. The segment is identified by its first three arguments – the file name, the start position and the stop position. As the segment is searched, each clause found is matched against the atom given as the fourth and last argument. The *RPRED* definition of *salary* given above can be read:

> To solve any atom of the form *(salary↓x)* (the x represents any list of arguments) find a clause in the opened file EXREL1 in the segment between character position 0 and character position 72 that matches the atom *(salary↓x)*

In general there will be several clauses in the segment that match *(salary↓x)* for the given *x*, so there will be several solutions to the query.

Chapter 7 includes a definition of a simplified version of *RPRED* in terms of the file primitives of micro-PROLOG.

4.5.3 The open command and the opened relation

The use is

open <file name>

It is equivalent to the primitive *OPEN* command of micro-PROLOG with the added feature that it 'remembers' the opened files by adding an *opened* assertion to the workspace. You can therefore find out which are the currently opened files by *LIST*ing the *opened* relation. You should always use *open* before posing a query that will access an external relation. Use it instead of the *OPEN* primitive because the *opened* assertion it generates provides useful information for the various listing commands described below.

If you do not open the external relation file before querying any of its relations you will get an error. If you are using the *errtrap-mod* error handler

described in chapter 6 you will be able to recover from the error by *opening* the file and continuing. Otherwise, the query will be aborted. You must *open* the file and repose the query.

Micro-PROLOG only allows four files to be open at any one time. This limits the number of different external relations you can access during a single query. You increase the number by putting more external relations on each file. When you have finished accessing all the external relations on a file you should *close* the file.

4.5.4 The close command

The use is

close <file name>

It is the opposite of *open*. It closes the file (using the primitive *CLOSE*) and deletes the *opened* assertion for the file from the workspace.

4.5.5 The internal command

This is almost the opposite of *external*. The difference is that it handles one relation at a time and leaves other external relations on the same file unchanged.

The use is

internal <relation name>

Notice that the file name is not given – it is not needed. The command picks up the clause defining the named relation in terms of *RPRED*, deletes this clause, and loads all the clauses on the file segment identified in the *RPRED* condition of the deleted clause.

The *RPRED* definitions for any other relations on that file are left unchanged, they can still access the appropriate segments of the file. However, there is now an unaccessible segment – a hole in the file – from which the definition of the *internalised* relation has been read. All its defining clauses are now in the workspace and can be added to or edited before the relation is again made external on *another* file: it must be another file if there are still accessible relations on the old file on which it was kept. This is the disadvantage of packing more than one relation onto a file.

4.5.6 The LISTEX, LISTFILE commands

If you *LIST* a relation that has been made external you see its *RPRED*

definition. To see its actual defining clauses you need to use *LISTEX*. Its use is

LISTEX <relation name>

where the named relation is external (i.e. is defined by an *RPRED* clause in the workspace program) e.g.

LISTEX salary

will produce listing of all the old clauses for the salary relation. This command automatically opens the file on which the clauses are kept and will close it again unless there is an *opened* assertion for the file.

Hence the importance of using *open* rather than the primitive *OPEN* to open external files for querying. If you use *open*, listing the clauses for the external relation will not result in the file being automatically closed after the listing: it will be left open for the queries.

To see all the saved clauses on a file use *LISTFILE*.

LISTFILE <file name>

will list all the clauses on the file. This can be used for any file on which a micro-PROLOG program has been saved, not just the files holding the external relations; so you can use it to list a normally *SAVE*d program. Again, the command automatically opens and closes the file unless there is an *opened* assertion for the file left by the *open* command.

4.5.7 The listex and listfile commands

These are alternatives to the *LISTEX* and *LISTFILE* commands which can only be used when the *query-mod* module of the SIMPLE front-end system is present. They require the query-mod module to be present because they use one of its exported relations, "?REV-P?" to map the clauses on the file back into SIMPLE sentence form. Otherwise, they have exactly the same effect as *LISTEX* and *LISTFILE*; *LISTFILE* should only be used to list a file of clauses that have been entered and saved using SIMPLE, similarly *listex* should only be used to list the clauses for external relations entered as SIMPLE sentences.

4.5.8 Changing the definitions of external relations

If you need to edit or delete some clause about a relation that is external then you must use the *internal* command described above to bring all the defining clauses back into the main memory. When you have made the changes, make the relation external again. You must use a different file if there are still external relations on the old one.

If the change to the definition is just the adding of some new clauses, you can just enter these in the normal way, leaving the old clauses on the file. If you *ADDCL* them before the *RPRED* definition for the relation, the new clauses will be used before the file clauses for the relation.

You can add them so that they are positioned after the *RPRED*.

When you eventually do an *internal* command for the relation, the *RPRED* definition is deleted, no matter where it is, and all the external clauses are added at the *end* of the remaining clauses. A new *external* command will now save all the defining clauses onto the specified file.

4.6 The file LOGIC

The file LOGIC contains a single module *logic-mod* which exports a definition for the logical operator: *ISALL*. It is fully described in section 7.8.

The module is also included in the files SIMPLE and MICRO.

5 SIMPLE PROLOG

SIMPLE provides a quite elaborate extension to the facilities of the built-in supervisor. It provides commands to add, delete and text edit programs and supports several high level forms of query. The most important feature is that it allows clauses to be entered and queries to be posed using a much more user friendly syntax than the standard syntax described in chapter 2. The SIMPLE syntax for clauses is much closer to the syntax of English. SIMPLE is therefore a very suitable system to use for programmers beginning micro-PROLOG. It is the system described in the books [Clark & McCabe 1984] and [Conlon 1985].

The form of clause accepted by SIMPLE is called a *sentence* because its syntax is so different. The program development commands of SIMPLE compile sentences into clauses and map the clauses back into sentences when they are displayed. For most uses, the programmer does not need to know anything about the syntax of clauses, only the syntax of SIMPLE sentences. However, you can always see the sentences in their compiled clause form by using the `LIST` supervisor command. In contrast, the `list` command of SIMPLE maps the clauses back into sentence form before displaying them.

To use SIMPLE you execute a `LOAD SIMPLE` command. (Note the warning at the beginning of the next chapter on MICRO. None of the MICRO modules should be present when you load in SIMPLE.) The file SIMPLE which will be loaded contains four modules called:

query-mod
program-mod {also in the file PROGRAM}
logic-mod {also in the file LOGIC}
errmess-mod

`query-mod` being the module which defines and exports all the relations that compile and de-compile SIMPLE sentences and it also includes the definitions of the query commands and other pre-defined relations supported by SIMPLE; `program-mod` is the module that defines all the program development commands, for example, the commands to add, delete and edit SIMPLE sentences. It can be optionally killed, with a

```
KILL program-mod
```

command, when a program has been developed in order to free space for the evaluation of queries. When the program needs to be edited or added to, this

module can then be loaded from the file *PROGRAM*, which only contains the module, with a

```
LOAD PROGRAM
```

command; *logic-mod* exports the definition of *ISALL* which is used to evaluate *isall* conditions in SIMPLE sentences and queries.

Finally, *errmess-mod* is a very simple error-handling module. It exports the definition of a relation called *"?ERROR?"*. When this relation is defined the micro-PROLOG interpreter calls its program on encountering a run-time error. The *"?ERROR?"* program in *errmess-mod* displays the condition currently being evaluated together with a more meaningful error message than the default error messages. For more information on error handling consult section 3.7. The messages given by *errmess-mod* are very similar to those of the example error handler given in that section. The module can be killed and replaced by the *errtrap-mod* error handler described in 6.3 of the MICRO chapter, or by one called *deftrap-mod* which is described below.

Two other modules in the files *EXPTRAN* and and *TOLD* can be optionally loaded and used with SIMPLE. The first file contains the module *exptran-mod*. When this is present expressions can be used in sentences and these are compiled into a sequence of conditions that evaluate the expression. The second file contains the module *told-mod* which defines and exports the relation *is-told*. This can be used in a straightforward way to write interactive programs, programs that ask for information by posing queries to the user. The *is-told* program can also be invoked from the error handlers *deftrap-mod* and *errtrap-mod*.

5.1 Syntax of sentences accepted by SIMPLE

A *sentence* is either

1. a simple sentence or
2. a conditional sentence.

5.1.1 Simple sentences

Simple sentences have four forms: infix, postfix, prefix and single condition.

infix simple sentences have the form:

```
<term> R <term>
```

where *R* is a constant which is the name of a two argument (binary) relation. The terms on either side of *R* are the two arguments of the binary relation.

e.g. *John likes Mary*
x LESS 45

postfix simple sentences have the form:

```
<term> R
```

where R is a constant which is the name of a single argument (unary) relation.

e.g. `Tom male` {male is the unary relation}
`x a-father` {a-father is the unary relation}

prefix simple sentences have the form

```
R(<term1> <term2> .. <termk>)
```
or
```
R <variable>
```

where *R* is a constant which is the name of an argument relation. The list of terms following *R* are the k arguments of the relation and are separated only by spaces.

e.g. `SUM(X 5 34)`
`APPEND( (2 3) (4 5) X)`

The form which is a relation name followed by a variable is a special meta-variable form. For an explanation and example of its use see the description of `true-of` given in 5.3.7 below.

Sentences about binary relations and unary relations can be entered in prefix form but will always be displayed by SIMPLE in the infix or postfix form.

e.g. `Bill likes x` can be entered as `likes(Bill x)`
`x male` can be entered as `male(x)`

Single condition simple sentences have the form

```
C
```

where *C* is a constant that names a 0-argument relation or condition.

e.g. `FAIL`
`/`
`Bill-likes-Mary` {the hyphens make it one constant}

5.1.2 Conditional sentences

Conditional sentences have the form:

```
<simple sentence> if <conjunctive condition>
```

where a conjunctive condition is

1. a condition or
2. a conjunction of the form:

```
<condition> and <conjunctive condition>
```

& is an accepted abbreviation for *and*.

A *condition* is

1. a simple sentence or
2. a complex condition.

A *complex condition* is one of the following:

(a) a *negated condition* of the form:

```
not <condition>
```
or
```
not (<conjunctive condition>)
```

e.g.
```
not x male
not (x male and x likes Mary)
```

(b) an *isall condition* of the form:

```
<term> isall (<term> : <conjunctive condition>)
```

e.g.
```
x isall (y : Bill father-of y and y male)
```

(c) a *forall condition* of the form:

```
(forall <conjunctive condition> then <conjunctive
condition>)
```

e.g.
```
(forall Tom father-of y then y male and y
married)
```

The outer brackets are essential.

(d) an *or condition* of the form:

```
(either <conjunctive condition> or <conjunctive
condition>)
```

e.g.
```
(either x likes Peter or Peter likes x & Sally
likes x)
```

Again the outer brackets are essential.

(e) an *equality condition* of the form

```
<expression> = <expression>
```

(f) an *expression condition* of the form:

```
R # (<expression> <expression> ... <expression>)
```

Expressions are terms of a special form. The syntax of expressions is given in 5.3 below. Expressions can only be used if the optional EXPTRAN has been loaded.

The syntax of terms is as described in chapter 2. There is one difference. The control characters that are allowed in quoted constants in clauses should not be used in sentences. If they are, they will not be displayed in the correct form by the SIMPLE `list` command and may well upset the way the sentence is displayed.

Example

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

x likes y if y female and not y likes Peter

x GE y if not y LESS x

PRED(x y z1) if x LESS y and not(PR(y z x) &
QUALIFY(x))
and z1=(x*y+z)

x all-the-sons-of y if x isall (z : y parent-of z &
z male)

only-has-sons(x) if (forall x parent-of y then y
male)

x parent-of y if (either x father-of y or x mother-
of y)
```

5.2 The relations and commands defined by program-mod

We briefly describe the program development commands defined in and exported from `program-mod`. For a more tutorial introduction to the use of SIMPLE we refer the reader to [Clark & McCabe 1984] or [Conlon 1985].

5.2.1 add

The `add` command allows you to add a sentence to the current workspace program. The sentence to be added must be enclosed in brackets. (Thus the argument to `add` is a list of terms that conforms to the syntax of a sentence.) One form of the command is:

```
add (<sentence>)
```

e.g.

```
&. add (Peter likes x if not x likes John)
&.
```

The sentence will be compiled into a clause and added to the end of the current list of clauses for the relation that the sentence is about. In this case, the sentence is about the relation *likes*.

To add into the middle of a program the form

```
add n (<sentence>)
```

is used, where the number *n* refers to the position that the new sentence should occupy in the listing of the sentences for its relation. For example, to add to the beginning of the *likes* relation use

```
&. add 1 (Peter likes John)
&.
```

Whenever you add a sentence about a new relation the relation name will be recorded in a *dict* assertion which is automatically added to your workspace program by the *add* command. So to see all these relation names you *list* the *dict* relation.

5.2.2 delete

This deletes a single clause from the program. It has two forms of use:

1. delete (<sentence>)
2. delete <relation name> n

where n is the position of the sentence to be deleted.

e.g.

```
&. delete likes 3
&. delete (tom likes mary)
```

5.2.3 kill

kill will delete an entire relation using the form:

```
kill <relation name>
```

e.g. to delete all the clauses for the likes relation type

```
&. kill likes
```

To get rid of the entire workspace program (you should normally only do this after a *save*) use

```
&.kill all
```

You will be asked if you really want to do this with the question

```
Entire program ?(yes/no)
```

The *yes* response causes all relations recorded by a *dict* assertion to be killed.

Finally, to get rid of a module use the form : kill <module-name>

```
&.kill exptran-mod
```

will get rid of the expression parser after you have used it to compile expressions in some *add*ed sentences.

All the uses of *kill* report the successful deletion of the program or module. When a program for a relation is *kill*ed its entry in the *dict* relation is also deleted. After a *kill all* the *dict* relation will be empty.

5.2.4 accept

The *accept* command can be used as an aid in adding a lot of simple sentences for a relation. It enables a sequence of simple sentences to be added without the need to repeatedly use the *add* command. An example use is:

```
&. accept likes
likes.(John Mary)
likes.(John Peter)
likes.end
&.
```

The *accept* command prompts for a simple sentence (in prefix form) with the name of the relation involved. The list of arguments of the sentence are then entered. You can continue entering sentences in this way until you enter *end*. The above example is equivalent to:

```
&.add(likes(John Mary))
&.add(likes(John Peter))
```

5.2.5 edit

The line editor can be used to edit an individual sentence by using this *edit* command. (See appendix B for details of the line editor.) The *edit* command is invoked as follows:

```
&.edit likes 1   {the relation followed by the sentence number}
1 (Peter likes John)
```

The sentence (surrounded by brackets) will be displayed preceded by the number which is its position and both the sentence and its position can then be edited using the line editor commands. Like *list*, *edit* maps clauses back

into sentence form before displaying them. It then re-compiles them into clauses on exit from the edit when the *ENTER* or *RETURN* is pressed.

You can change the position number of the sentence in order to re-position the sentence whether or not you have edited the sentence. In the above example, changing the position number 1 to 2 will move the sentence to the second position in the listing of *likes*. The old second sentence will therefore become the new first sentence. You should never delete the position for the sentence from the beginning of the edit line since this is used when you exit the edit to determine where the edited sentence should be added. The old sentence is deleted just before the edited version is added to the program. If you change the name of the relation that the sentence is about you will be told that this has been done with the message

```
relation changed to ....
```

on exit from the edit.

You can only *edit* relations with names recorded in the *dict* relation. See appendix B for details on the use of your machine's line editor and any limitations on the size of sentences that can be edited.

5.2.6 cedit

Exactly the same form of use as *edit*. The difference is that the old version of the sentence is not deleted. The command is very useful for building programs in which two are more sentences differ only slightly.

Example

```
&.add (y greater-of (y z) if not y LESS z)
&.cedit greater-of 1
1 (X greater-of (X Y) if not X LESS Y)
```

This line can now be edited to

```
2 (Y greater-of (X Y) if not Y LESS X)
```

as an alternative to explicitly adding it as a second sentence.

5.2.7 function

This is the command that is used to declare that a relation is to be used as a function in expressions. Its use is fully described in section 5.4.

5.3 The relations and commands exported by query-mod

5.3.1 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
&.
```

Note that the clauses are displayed in sentence form. The *list* command de-compiles the clauses back into sentences before displaying them. You will see all the sentences for the relations recorded in the *dict* relation.

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

```
&. list likes
```

This uses the alternative form of the command:

```
list <relation name>
```

You can only *list* relations recorded by a *dict* assertion.

To see all the currently defined relation names use:

```
&.list dict
```

You will see all the relation names about which you have *added* a sentence, providing you have not *killed* the relation.

5.3.2 is

The *is* command makes a YES/NO query of the program. It has the form:

```
is (<conjunctive condition>)
```

with the single argument a bracketed conjunctive condition. If the conjunctive condition can be solved, the response is *YES* otherwise it is *NO*.

For example,

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

5.3.3 which (*all* an accepted synonym)

The *which* query finds all answers to some condition of a specified form. The syntax of the command is

```
which (<sequence of terms> : <conjunctive
condition>)
```

where *<sequence of terms>* denotes the form of the answer required, and the *<conjunctive condition>* is the query to be evaluated. For example, to find all the people that like John use:

```
&. which(x : x Likes John)
Peter
Mary
No (more) answers
&.
```

To find the pairs of people who like each other:

```
&. all(x y like each other : x likes y and y likes
x)
Peter Mary like each other
No (more) answers
```

Notice the use of the constants in the sequence of answer terms to make up an answer message. The sequence of answer terms can be of any length and can contain any form of term. The answers are displayed by the micro-PROLOG primitive *PP*.

To compute the sum of 3 and 5

```
&.which(x : SUM(3 5 x))
8
No (more) answers
```

5.3.4 one

The *one* query is similar to *which* except that it prompts after each solution has been found with

```
more(y/n)?
```

a *y* response gives the next answer if there is one, a *n* response terminates the query evaluation.

```
&.one(x : x likes John)
Peter
more(y/n)?.y
Mary
more(y/n).y
No (more) answers
&.
```

5.3.5 save

The entire workspace program can be saved for later use with this command. It is saved in clause form, not in sentence form. The use of the *save* command is:

```
&. save <file name>
```

where <file name> is a file name in the normal micro-PROLOG form. (See section 3.5.3 and appendix D.) *save* is just a lower case synonym for the supervisor *LOAD*.

The *<file name>* must be different from any relation name in the program. The *saved* program can subsequently be reloaded using *load*.

5.3.6 load

The *load* command is used to re-load a previously *saved* program. Its use is

```
&. load <file name>
```

Like *save* it is just a lower case synonym for the supervisor *LOAD*.

5.3.7 APPEND, ON, true-of

The *query-mod* module includes and exports definitions of two useful list processing programs for the relation names *APPEND* and *ON*. In sentence form, their definitions are:

```
APPEND(() X X)
APPEND((X|Y) Z (X|x))
    if APPEND(Y Z x)

X ON (X|Y)
X ON (Y|Z)
  if X ON Z
```

The *APPEND* relation has many uses. It can be used in exactly the same way as the *append* relation of the Primer. The relation *ON* can be used to find members of a list or test for membership in the same way as the *belongs-to* relation of the Primer.

The *true-of* relation can be used when the relation name of a prefix form

condition or its argument list is to be given as the value of a variable. Its form of use is:

```
<variable or relation name> true-of <list pattern>
```

The `<variable>` *must* be a relation name `R` by the time that the condition is evaluated. The `<list pattern>` represents the list of arguments for the relation; `true-of` gives the power of the predicate symbol and argument list meta-variables of the standard syntax (see chapter 2) to programs entered in sentence form.

For the clause form definition of `true-of` see the next chapter. The definition that is exported from `query-mod` cannot be given in sentence form.

Example

```
x true-of-all Y if (forall y ON Y then x true-of
(y))
```

defines a relation `true-of-all` that can used to test if all elements of a list `Y` have some property given as argument `x`. Notice that since `x` names a property – a unary relation – the second argument of `true-of` is a unit list `(y)` comprising the element to be tested.

```
is( male true-of-all (Peter John Sam))
```

checks that Peter, John and Joan are all recorded as males.

```
which(x : x dict & x true-of (tom|y))
```

will find all the names of the relations recorded in `dict` such that `tom` 'satisfies' the relation. That is, if the relation is a property, then `tom` can be shown to have the property, and if it is a relation with more than one argument, then `tom` can be shown to be related to some other things by that relation.

The `y` in the list pattern `(tom|y)` represents the remaining arbitrary number of arguments of the relation `x`. When the `true-of` condition is solved `y` will be bound to a list of these extra arguments. When `x` is a property, `y=()`.

The `<list pattern>` of a `true-of` condition can be a variable representing a list of any number of arguments. The use of variable in this way is not restricted to `true-of` conditions. Any simple sentence, written in prefix form, can have its list of arguments represented by a variable. As an example,

```
which(X : father-of X)
```

can be used to find all the pairs in the `father-of` relation. It is a shorthand for

```
which((x y) : father-of(x y))
```

The use of a single variable as the argument following a relation is always a shorthand for a list of different variables, one for each argument of the relation. This, and *true-of*, are the only 'meta-syntax' forms allowed in SIMPLE programs.

5.3.8 CONS, @, #, =

CONS and @ are used in expressions which are dealt with in the next section. The # and = relations are only used together with expressions and are also described in the next section; *query-mod* does not actually contain a definition of =, but it cannot be defined in a user program because it has a special role in the syntax of sentences.

5.3.9 *, %, +, −

The *query-mod* module exports definitions for the auxiliary arithmetic relations *, %, +, −. The auxiliary relations *,%,+,− are recognised as arithmetic operators in expressions. The *query-mod* definitions of the operators are:

```
+(X Y Z) if SUM(X Y Z)
-(X Y Z) if SUM(Y Z X)

*(X Y Z) if TIMES(X Y Z)
%(X Y Z) if TIMES(Y Z X)
```

5.3.10 defined, reserved

defined can be used to check if a relation name has any defining sentences. It can only be used in this checking mode.

```
&.is(likes defined)
YES

&.is(animal defined)
NO
```

reserved can be used to find all the SIMPLE reserved words.More precisely, it returns a list of all the names exported by the currently loaded modules with the names *dict*, *func* and *data-rel* appended to the front. Although these last three are not exported by any of the SIMPLE modules they have a special role in SIMPLE programs and should not be used as normal relation names. We have already encountered *dict*. The role of *func* will be explained in the next section and the role of *data-rel* in section 5.5 on *errmess-mod*.

5.3.11 Parse-of-S, Parse-of-ConjC, Parse-of-SS, Parse-of-Cond, Parse-of-CC

These are the relations used by *query-mod* to compile and de-compile sentences into clauses.

Parse-of-S

```
X Parse-of-S Y
```

holds when *X* is the clause version of the sentence list *Y* e.g.

```
((likes X Y)(likes Y X)) Parse-of-S (X likes Y if Y
likes X)
```

It can be used to parse a sentence list into a clause with a condition of the form

```
X Parse-of-S <sentence list>
```

or to de-compile a clause into a sentence list with

```
<clause> Parse-of-S Y
```

Parse-of-ConjC

```
X Parse-of-ConjC Y
```

holds when *X* is the atom list form of the conjunctive condition list *Y* e.g.

```
((likes Tom Y)(NOT male Y)) Parse-of-Conj (Tom
likes Y & Y male)
```

It can be used for parsing conjunctive conditions with a condition of the form

```
X Parse-of-ConjC (<conjunctive condition list)
```

or for de-compiling lists of atoms into conjunctive conditions

```
<atom list> Parse-of-ConjC Y
```

Parse-of-SS

```
Parse-of-SS(X Y Z)
```

holds when *X* is the atom form of the simple sentence which is formed from the sequence of terms on the list *Y* up to the list *Z*. That is, *X* is the atom form of the difference between the lists *Y* and *Z*. (See chapter 6 of [Clark & McCabe 1984] for more information on difference pairs.) For example,

```
Parse-of-SS((likes X Y) (X likes Y if Y likes X)
(if Y likes X))
```

It can be used for finding and parsing the front simple sentence of a list with a condition of the form

`Parse-of-SS(X <sentence list> Y)`

`X` will become the atom and `Z` the remainder of the sentence list.

To map an atom into a simple sentence use the condition of the form

`Parse-of-SS(<atom> Y ())`

where the remainder is given as the empty list. A condition

`Parse-of-SS(<atom> Y Z)`

will give `Y` the list pattern value `(<simple sentence form of atom> | Z)` e.g.

`Parse-of-SS((likes x y) Y Z)`

gives `Y` the value (`x` likes `y` | `Z`).

Parse-of-CC

`Parse-of-CC(X Y Z)`

holds when `X` is the atom form of the complex condition represented by the difference between the lists `Y` and `Z`. It has the same uses as `Parse-of-SS`.

Parse-of-Cond

`Parse-of-Cond(X Y Z)`

holds when `X` is the atom form of the condition which is the difference between lists `Y` and `Z`. It has the same uses as `Parse-of-SS`. Reflecting the syntax definition of a condition that we gave at the beginning of this chapter `Parse-of-Cond` is defined by the two rules

```
Parse-of-Cond(X Y Z) if Parse-of-SS(X Y Z)
Parse-of-Cond(X Y Z) if Parse-of-CC(X Y Z)
```

5.3.12 "FIND:", "?VARTRANS?"

Both these relations are used internally by the parse programs and are of little use in user programs. They are exported from `query-mod` only because they are used by other modules.

5.3.13 "?REV-P?"

This is the relation that is used to map each clause for a relation into a list which when displayed using the `P` primitive of micro-PROLOG will be the sentence form of the clause formatted with each condition of the sentence on a new line. It does this by inserting control character constants in the sentence list that it generates from the clause. It is used by the `list` command. The `Parse-of-S` relation described in the next section is more general in that it

can be used to generate clauses from sentences as well as sentences from clauses. When "Parse-of-S" generates a sentence from a clause no control character constants are inserted so that if the sentence is then displayed using *P* or *PP* it will not be formatted.

The form of use of *"?REV-P?"* is

```
X "?REV-P?" Y
```

where X is a clause and Y is a variable.

Example

```
&.?(("?REV-P?" ((likes X Y)(likes Y X)(female Y))
x) (P | x))
X likes Y if
Y likes X and
Y female
&.
```

This query uses the primitive *?* described in chapter 3. The condition *(P | x)* is used rather than *(P x)* so that terms of the sentence list x will be treated as a sequence of different arguments and so displayed without the outer brackets. If *(P x)* had been used the display would be

```
(X likes Y if
      Y likes X and
      Y female)
```

5.4 Using expressions in sentences – the module exptran-mod

Expressions can be used in equality conditions and expression conditions in sentences and queries when the optional module *exptran-mod* has been loaded with a

```
load EXPTRAN
```

command. The expressions are actually compiled into a relational form by the single relation *Epression-Parse* exported by exptran-mod.

5.4.1 Expression conditions

Expression conditions have the form

```
R # (E1 E2 .. Ek)
```

where *R* is the name of a relation and *E1* ... *Ek* are expressions. Expressions are just terms of a certain form. The # is the signal that *E1* ... *Ek* are not normal arguments but that some or all of them contain arithmetic operators

and function calls. The syntax of expressions is formally defined below. For now we shall just look at examples.

Example

```
LESS # ((2*x) (5+y))
```

is a `LESS` condition with arguments the values of the expressions `(2*x)`, `(5+y)`. When it is evaluated, the variables `x` and `y` should have been given values.

The relational form into which the condition is compiled is:

```
(X LESS Y) # (* (2 x X) and + (5 y Y))
```

The # in this form should be read as `where`. So this condition is read as

```
X LESS Y where X is 2 * x and Y is 5 + y
```

This is the sentence syntax form of what is produced by `Expression-Parse`. It is what will be displayed if you kill `exptran-mod` after the expression condition has been entered or if you add `rel-form` to your program and `list` the sentence in which it is used (see 5.3.5 below).

The relational form of an expression condition

```
R # (E1 E2 .. Ek)
```

is

```
(R(t1 t2 .. tk)) # (<conjunctive condition>)
```

The evaluation of the conjunctive condition will produce values for the variables in the terms `t1, t2, .. tk` so that they become the values of the original expressions `E1, E2, .. Ek`. In our `LESS` example, the terms are the variables `X`, `Y` and the evaluation of the conjunctive condition

```
* (2 x X) and + (5 y Y)
```

will result in `X` having the value of `(2*x)` and `Y` the value of `(5+y)`.

When a compiled # condition is evaluated, the conjunctive condition that produces the values of the expression arguments is evaluated first, then the `R` condition. On backtracking, only the `R` condition will be re-tried for alternative solutions, since there will be no alternative solutions for the evaluation of the expression values.

Example

The condition

```
salary # (x (12*157))
```

is compiled into the relational form:

```
(x salary X) # (* (12 157 X))
```

The * condition computes the value of (12*157) and the evaluation of the # condition will reduce to the evaluation of

```
x salary 1884
```

in order to find an x with recorded salary of 1884. Backtracking will result in different values for x being sought but will not cause the re-computation of the value 1884.

For details of the standard syntax form of a compiled expression condition and for the definition of # that is exported from query-mod, and which is used to evaluate # conditions, see the next chapter. The MICRO definition of # relation is exactly the same as the query-mod definition.

5.4.2 Equality conditions

An equality condition has the form

```
E1 = E2
```

where E1 and E2 are expressions. It is a shorthand notation for the expression condition

```
EQ # (E1 E2)
```

Thus, the two expression arguments E1, E2 of an equality condition are evaluated and then their values are checked for identity using the primitive EQ relation. EQ is itself defined by the sentence

```
EQ(X X)
```

so, if one of the expressions is a variable, this will result in the variable being given the value of the other expression.

Examples

```
x=(y * 67 + z)
```

can be used to give x the value of the bracketed expression if y and z have values at the time that the condition is evaluated. If they do not, there is a "Too many variables" error message displaying either a call to SUM or a call to TIMES.

```
0=(2*x*x + 7*x - 23)
```

can be used to check that the value of x satisfies the equation:

$2x^2 + 7x = 23$

Note: It cannot be used to find the roots of the equation.

5.4.3 Syntax of expressions

An *expression* is

1. an arithmetic expression
2. a function call or
3. some other term

An *arithmetic expression* is a list of the form

(<expression> <operator> <expression>)

where the operator is one of the following:

*	for multiply
% or /	for divide
+	for addition
– or ~	for subtraction (the ~ is safer because of the other syntactic roles of -. If you use - you should always surround it with spaces.)

The outermost brackets of an arithmetic expression are essential – so an arithmetic expression is just a three-element list whose second element is an operator. However, if the expression arguments of this operator are also arithmetic expressions the inner brackets around them may be dropped in accordance with the following precedence

`* / %` equal – left associative

greater than

`+ ~ –` equal – left associative

Examples

```
(x * y + 3 / z)    equivalent to ((x * y) + (3 / z))
(x + y / (5 + z)) equivalent to (x + (y / (5 + z)))
(x * y / 5 + z)    equivalent to ((x * y) / 5) + z))
```

`query-mod` exports definitions for the operators `* % + -`. Uses of `/` and~ in expressions are mapped into conditions for `%` and `-` respectively. Any use of ~ or `/` in an expression will subsequently be displayed as a use of `-` or `%`.

A *function call* is a list of the form

```
(R E1 ... En-1)
```

where `E1...En-1` are expressions and `R` is a n-ary relation name that has been declared a function with the command

```
function R
```

The expressions are the first n-1 arguments of what would otherwise be given

as an expression condition of the form

```
R # (E1 ... En-1 x)
```

The *x* found by the evaluation of this condition is the value denoted by the function call `(R E1 .. En-1)`. Notice that this means that relations should be declared as functions only if the value of the last argument of the relation is uniquely determined by values for the preceding arguments.

Example

Suppose the relations `div` and `mod` are defined by the rules:

```
div(x y z) if TIMES(y z1 x) & INT(z1 z)
mod(x y z) if div(x y z1) & z=(z~y*z1)
```

Note: the essential use of ~. If ~ had been used the `z-y` would have been parsed as a constant resulting in the call to `TIMES` failing; this would cause the evaluation of `mod` to fail. However `z~y` is recognised by the lexical analyser as three terms `z`, ~ and `y` and hence as a use of the subtraction operator.

`INT` is a primitive of micro-PROLOG that can be used to test if a number is an integer or to find the integer part of a number, as here. So, `div(x y z)` can be used to find the integer divisor `z` of `x` and `y` and `mod(x y z)` can be used to find the remainder `z` of the integer division of `x` by `y`. For both `div` and `mod` the last argument is functionally determined by the first two arguments. So, we can declare these relations as functions

```
function div
function mod
```

and then use them in expressions:

```
x=((mod 85 23)*34)
LESS #((div 500 x) 23)
```

If you forget to declare that some relation `R` is a function before using it in an expression the expression parser of `exptran-mod` will assume that the function call is just a list that has as its first element a constant. However, it will tell you this by giving you the message

```
R assumed not to be a function
```

If the expression was in a query you will get the wrong answers. If the expression was used in an `add`ed sentence you can easily recover from the error. Declare `R` as a function and `Edit` the sentence. Just call the line editor and then immediately exit it with **RETURN**. The editor de-compiles the clause for the original sentence, mapping compiled expressions back into source form. It re-compiles the expressions on exit. This time it will recognise the use of `R` as a function call because of the declaration.

A `function R` declaration causes a "R func" sentence to be added to the user workspace program. It is this that causes the expression parser to treat a list beginning with `R` as a function call when it appears in an expression. You can examine what functions have been declared with a

```
which(x : x func)
```

query. Alternatively, you can `list` the `func` relation. If you `kill` a relation that has been declared a function the `func` sentence for the relation will be automatically deleted. (This only happens if you use the `kill` command of `SIMPLE`. If you use the supervisor `KILL` neither the `func` sentence nor the `dict` sentence for the relation will be deleted.)

5.4.4 Warning on the use of ! in expressions

From the above discussion on the problem of undeclared function names it will be obvious that lists can be given as arguments to function calls in expressions. As an example

```
x = (APPEND (1 2) (APPEND (3 4) (5 6)))
```

can be used to append three lists. As we remarked earlier, the relation `APPEND` is defined in and exported from `query-mod`. It is also recognised by the expression parser as a function name so you do not need to declare it as a function in order to use it in expressions. However,

```
x=(3 ! (APPEND y z))
```

cannot be used to make `x` the list `3` followed by the concatenation of the lists `y` and `z`. This is because, as a micro-PROLOG term, the list

```
(3 ! (APPEND y z))
```

is just another way of writing the list

```
(3 APPEND y z)
```

in which the function call sublist has disappeared. So the expression parser sees a list of four elements and leaves it unchanged. The moral is that the term following a ! in an expression can never be a function call. When you do want to denote the rest of a list by a function call you must use an explicit `CONS` function call (as in LISP) to construct the list instead of the primitive !. The above = condition needs to be re-expressed as

```
x = (CONS 3 (APPEND y z))
```

Like `APPEND`, `CONS` is defined in and exported from `query-mod` and is recognised as a function in expressions. Its definition, in sentence form, is

```
CONS(X Y (X!Y))
```

Finally, there is one other predefined function. It is the apply function @. Its definition, which can only be given in clause form, is given in 6.1.16 of the next chapter. It applies its first argument to the rest of its arguments. An example use is in the expression

```
x = (@ y 3 4)
```

If *y* is bound to * when this is evaluated *x* will be bound to *12*.
The @ is needed. If instead you use

```
x = (y 3 4)
```

then *(y 3 4)* is not recognised as a function call. The equality is compiled into

```
EQ(x (y 3 4)) # ()
```

with an empty condition for the argument evaluation. It will result in *x* being bound to the list *(* 3 4)* when evaluated. @ is the expression equivalent of the *true-of* sentence condition and the predicate meta-variable (see chapter 2) for atoms.

5.4.5 The relation Expression-Parse

Expressions occurring in equality conditions and expression conditions are compiled into relational form using the relation *Expression-Parse* that is exported from the module *exptran-mod*. This same relation is used to de-compile the expressions when a clause is *list*ed or *edit*ed.

```
Expression-Parse(X Y Z)
```

holds when *Y* is the term *X* with all the expression sublists replaced by new variables and *Z* is the list of atoms the evaluation of which will give these new variables the values of the expressions of *X* e.g.

```
Expression-Parse((x *y + (fact z)) X ((* x y x1)(fact z x2)(+ x1 x2 X)))
```

It can be used to parse expressions with a condition of the form

```
Expression-Parse (<expression> X Y)
```

or to de-compile expressions with

```
Expression-Parse(X <term> <atom list>)
```

5.4.6 When the expression handler is not needed

If you kill the *exptran-mod* module after you have entered some sentences that used expressions, the expressions will then be displayed in the compiled relational form even by the *list* and *edit* commands. This is because a

check is made to see if the module is present before *Expression-Parse* is used to de-compile expressions.

If the *exptran-mod* module is present then the expressions are displayed in the normal expression form. This means that you can have the convenience of using expressions in sentences whilst you are developing a program. Then, when you want to start using the program, you can get rid of *exptran-mod* with a

```
kill exptran-mod
```

command to make available more space for the query evaluations. Of course, you will not be able to use expressions in the queries to the program.

If you do enter a sentence or query that uses expressions with the *exptran-mod* module not present you will get an error message of the form

```
No definition for relation
trying: Expression-Parse(<expression> <variable>
<variable>)
```

If you are using one of the alternative error handlers *deftrap-mod* or *errtrap-mod* you can recover from this error and return to the interrupted compilation of the expression – more on this later. Otherwise you should load the *exptran-mod* module with a

```
load EXPTRAN
```

command and re-enter the sentence or query.
Incidentally, you can see the compiled relational form of any expression in a program, even when *exptran-mod* is present, by adding the sentence *rel-form* to your program with

```
&.add (rel-form)
```

Now, when you *list* or *edit* the program, compiled expressions will not be put back into expression form but will be displayed in relational form. (The relational form of an expression condition can be edited and it will be compiled back into clause form on exit from the *edit*.) The display of the relational forms will continue until you *delete* the *rel-form* sentence. It is useful if you want to check that you did not make a mistake in an expression and that what you intended is what has been recognised and compiled.

5.5 The error handler errmess-mod

On a run-time error the error handler *errmess-mod* that is loaded as part of *SIMPLE* just prints out a message identifying the error (the messages are similar to the ones given by the example error handler in chapter 3) along with

the condition it was trying to evaluate when the error occurred. The current query or command is aborted and you are then returned immediately to the supervisor (you will get the *&.* prompt for a new query or command).

An alternative error handler, that allows you to recover from the very common error of not having a definition for a relation of the condition about to be evaluated (through forgetfulness or misspelling) is in the file DEFTRAP of the distribution disc. You can switch to this error handler by executing the following pair of commands in the *order* given.

```
kill errmess-mod
load DEFTRAP
```

It is important that you do the *kill* first. This is because *errmess-mod* and the module *deftrap-mod* that is loaded from the file DEFTRAP both export definitions for the error trap relation *"?ERROR?"* (see chapter 3). If you do the *load* first you will get the error: "Illegal use of modules".

The only difference between *errmess-mod* and *deftrap-mod* is that the error "No sentences for relation" is specially handled by *deftrap-mod*. The condition that was being evaluated, which contains the undefined relation, is displayed followed by the prompt:

```
error&(? for info).
```

error& is the prompt you will continue to get whilst in this error state with the evaluation in which the error occurred suspended. Entering *?* produces the message

```
to quit enter: q
or enter: tell (see manual)
or enter: / <any command> (eg / add (sentence), /
load file)
to continue enter: c
```

If you enter any other first response than *q* / or *c* you will again get the *?* info.

The *q* causes an *ABORT* to the top-level supervisor which is what normally happens with the *errmess-mod* handler.

tell invokes *is-told* (see below) with argument the offending condition. It enables the condition to be answered interactively during the current query evaluation. This is useful for 'top-down' development of programs. You can define the higher-level relations in terms of lower-level relations whose programs are yet to be constructed. Then, when you query the partly developed program and get the "No sentences for relation" message for a lower level relation you can supply the answer to the condition using *tell*.

But note, your answers are not remembered. If a condition using the undefined relation is again encountered you will be prompted to answer the condition even if it is identical to one which you have already answered.

The only way to avoid this is to *add* sentences defining the relation or to *load* a file that contains a definition of the relation. You then enter the *c* to continue the suspended query evaluation. You can *add* any number of sentences for the undefined relation before entering the *c*, in fact you can enter any number of commands providing each is preceded by the */*. The */* is the escape symbol that allows a command to be entered whilst in the error state.

Loading a file is the appropriate recovery response if you have *killed* a module such as *exptran-mod* and you get an error message such as

```
No sentences for relation
trying: Expression-Parse((X*Y) Z x)
error&(? for info).
```

This will happen if you forgetfully use a = or # condition in a query, or in a sentence that you are adding to the program, after you have got rid of the *exptran-mod* module. The recovery response is

```
/ load EXPTRAN
c
```

and the parsing will continue from the point that the error occurred.

A final example concerns the appropriate recovery action if you have misspelt a relation name, say you have used *fatherof* instead of *father-of*, you can recover by adding a sentence that defines *fatherof* as *father-of*. When the query evaluation is over you can edit the program to correct the spelling error and delete the definition of *fatherof*.

```
No sentences for relation
trying : fatherof (X bill)
error&(? for info). / add (x fatherof y if x father-
of y)
error&. c
```

Notice that the error handler displays the condition in the prefix form.

For an even more sophisticated way of handling errors, kill *errmess-mod* and load the MICRO error handler described in the next chapter. This is in the file ERRTRAP.

data-rel relations

Sometimes, usually when a relation is being used to record facts, it is more convenient if micro-PROLOG interprets having no sentences for a relation as a

failure to solve the condition rather than as an error. The *errmess-mod*, *deftrap-mod* and *errtrap-mod* error handlers can be informed that they are to treat a particular relation R in this way simply by adding the sentence

```
R data-rel
```

to your program. Before displaying the *No sentences for relation* error message each error handler checks to see if there is such a sentence about the relation. If there is, the message is not displayed and the evaluation continues as though there had been sentences but none had matched the condition being evaluated e.g.

```
&.add(jolly data-rel)
&.which(x : Tom father-of x & x jolly)
No (more) answers
```

even if we have no sentences for *jolly*.

5.6 Using the relation is-told

is-told is a multi-argument relation exported from the module *told-mod* which is in the file *TOLD*. To use *is-told* do a

```
LOAD TOLD
```

To delete its definition when you have finished using it do a

```
kill told-mod
```

When evaluated, an *is-told* condition displays its sequence of arguments followed by a ? and waits for a response. If there is just one argument, which is a list, this is displayed without the outer brackets. The responses and their effects are:

response	*effect*
yes	The *is-told* condition for the displayed argument is assumed to be true. Backtracking will not cause the question to be posed again.
no	The *is-told* condition is assumed false (i.e. it fails).
ans ...	The ... is a sequence of terms, one for each *different* variable in the displayed message. The *is-told* condition is solved for values of the variables given in the response. The ith term in the response sequence becomes the value for the ith variable in the displayed message in the left to right order of the text. For example, if the *is-told* condition is `(X likes Y) is-told`

the message and response

```
X likes Y ? ans tom bill
```

makes `X=tom` and `Y=bill`. Backtracking will result in the message being re-displayed when an alternative solution can be given. This repeated prompting for new solutions on backtracking continues until you enter `no` or `just`.

just ... The same as `ans` except that on backtracking you are not asked for another solution. It is assumed to be the last solution to the `is-told` condition.

5.6.1 Example use of is-told

1. `which(percent z : (mark x outof y) is-told & z=(x/y*100))`

sets up an interaction that can be used to convert pairs of numbers to percentages. A example interaction is:

```
mark X outof Y ? ans 20 40
percent 50
mark X outof Y ? ans 15 60
percent 25
mark X outof Y ? just 40 120
percent 3.3333333E1
No (more) answers
```

2. `x is-male if x known-male`

```
x is-male if not x known-male &
                (x a male) is-told & (x known-male)
add
```

defines `is-male` in such a way that the user is queried whenever an `is-male` condition is encountered with argument given but not recorded as a `known-male`. A `yes` response to the question such as

```
keith a male ?
```

results in the `keith is-male` condition that provoked the question being solved and a `keith known-male` sentence being added to the program. A `no` response results in the condition failing.

5.6.2 Using is-told from DEFTRAP error handler

The `tell` response of the `DEFTRAP` error handler will invoke `is-told` with argument the error condition for the undefined relation which you can then

answer interactively. The use of the *tell* response is actually equivalent to the pair of responses

```
/ add(R X if (R X) is-told)
c
```

where *R* is the undefined relation. That is, *tell* adds a sentence that defines *R* as an *is-told* relation to be answered by the user. You should delete this sentence before you *add* the actual defintion for *R*.

If you use the *tell* response and you have not loaded TOLD you can still recover from the error by loading the file when you get the second error message. The interaction will be something like:

```
No sentences for relation
trying : father-of (tom X)
error&(? for info).tell
No sentences for relation
trying : is-told (father-of (tom X))
error&(? for info). / load TOLD
error&.c
father-of (tom X) ? .
```

and you can now continue by answering the *father-of(tom X)* question in the manner described above. You have recovered from an error encountered in an error recovery action! You are now out of the error state.

5.7 Using external relations – the EXREL utility

Note that the *EXREL* facility depends upon the *SEEK* primitive of micro-PROLOG. Please check in appendix C that this is implemented for your computer. If it is not available then please ignore this whole section.

On occasions, especially when the definition of some of your relations comprises a lot of simple sentence facts, there is very little space left when the entire program is loaded. Incidentally, you can always find out the amount of free space left – for query evaluation or the loading of extra sentences – by querying the primitive *SPACE* relation.

```
which(x : x SPACE)
```

will give you the amount of free space as a number of K bytes.

When space is at a premium, you can trade time for space and have some of your relation definitions reside in a specially created disc file. The file is created using the *external* command of the utility module *exrel-mod* which is in the program file EXREL of the distribution disc. This can be loaded and used in conjunction with the SIMPLE system.

The use of this utility is described in chapter 4. Note that there are two commands supported by EXREL – *listex* and *listfile* that can only be used when the *query-mod* module is present. They give a listing of an external relation and a program file in sentence form.

5.8 Tracing SIMPLE queries using SIMTRACE

The TRACE program described in chapter 4 can be loaded and used to trace evaluations of programs developed using SIMPLE. The drawback of using this program, especially for beginning programmers, is that its *??* query form has the query expressed as a list of atoms and during the trace conditions are displayed as atoms. In fact, to use TRACE none of the SIMPLE modules need be present. They are best deleted to make available more space for the trace.

There is a version of the trace facility called SIMTRACE, that makes use of some of the exported relations of query-mod which allows the user to trace *is* and *all* queries posed using the SIMPLE syntax. It also gives more information during the evaluation – it gives information about the failed matches as well as the successful matches – and it displays the conditions being evaluated as SIMPLE syntax conditions. It is a useful program for a beginner to use to sharpen his understanding of how micro-PROLOG evaluates queries using backtracking. But because the *query-mod* module must also be present (*program-mod* and *logic-mod* can and should be killed before loading SIMTRACE and re-loaded from the files PROGRAM and LOGIC when you have finished using it), and because tracing takes up a lot of space, only relatively small programs can be traced using SIMTRACE.

The trace program is in the file SIMTRACE of the distribution disc. To use it do a *load SIMTRACE*. It contains one program module called *simtrace-mod*, so to get rid of the tracer when you have finished do a *kill simtrace-mod*.

The SPYTRACE program described in chapter 4 can also be used to set up spypoints on programs developed and queried using SIMPLE. That there are spypoints on a program will not prevent the use of SIMTRACE. However, you will not be able to trace the evaluations of conditions for spypoint relations. You have to *unspy* the relation if you want to trace it using SIMTRACE.

5.8.1 The relations exported by simtrace-mod

The *simtrace-mod* module exports two relations: *is-trace* and *all-trace*. They are used in exactly the same way as the *is* and *all* queries of *query-mod*. To trace the query

```
all(x : Tom parent-of x & x male)
```

use

```
all-trace(x : Tom parent-of x & x male)
```

The trace will take you through the evaluation step by step. As each condition is reached, the condition is displayed in the form

```
<condition identifier> : <condition>
```

where the condition identifier is a list of integers that also gives the 'history' of the condition back to the original query. All the conditions of the original query have a single integer identification which is the position in the query. In the above query `Tom parent-of x` will have the identifier `(1)` and `x male` the identifier `(2)`. Whenever a rule is applied, the identifier grows by one number. The identifier `(2 1)` tells you that the condition is the second condition in the rule currently being used to evaluate the first condition of the original query. The identifier `(3 2 1)` tells you that it is the third condition of the rule currently being used to solve the second condition of the rule being used to solve the first condition of the original query.

When the condition displayed is for a relation in your program (one recorded in the `dict` relation) you will also get the prompt

```
trace ?
```

and the tracing will be suspended until you respond. The responses are the same as those for the TRACE of chapter 4. They are:

y	to trace the evaluation of the condition
n	not to trace it, only the solutions to the condition will be shown
q	to quit the trace entirely
s	to resume the trace with the condition, as displayed, assumed solved
f	to resume the trace with the condition, as displayed, assumed failed, i.e. assumed to have no solution
/	followed by a command, e.g. /list rel

If you enter any other response you will be reminded of the allowed responses and prompted again.

The tracer will also prompt you when it reaches a complex condition such as an `isall`. This time the prompt will be

```
trace ?(y/n)
```

indicating that *y* and *n* are the only responses. In fact, any other response than *y* is taken as *n*. A *y* response enables you to trace inside the evaluation of the complex condition.

When a condition for one of your program relations is traced you will be told which sentence is being used to try to solve the condition with a message of the form

```
matching I : <condition> with head of N :
<conclusion of sentence>
```

where *I* is the condition identifier and *N* is the number of the sentence being used. (It is the position in the listing of the sentences for the relation of the condition.) You will be told whether the sentence matches the condition. If it does the result of the match will be displayed. Then, if the matched sentence has preconditions, these will be displayed as

```
new query : <precondition(s) of the matched
sentence>
```

and the trace will continue with the evaluation of each of the conditions of the new query. The identifier for each of these conditions will be the identifier for the condition just matched with an extra condition number at the front.

When a condition is solved you will get the message

```
<condition identifier> solved : <condition in
solved form>
```

When backtracking results in an alternative solution for a condition being sought, you will get the message

```
retrying <condition identifier> ..
```

Finally, when a condition cannot be solved, or when all solutions have been tried and the evaluation is backtracking to find alternative solutions to preceding conditions, you will get the message

```
<condition identifier> failing: <condition>
```

6 The MICRO extension to the supervisor

The MICRO file of the distribution disc contains three modules called *micro-mod*, *logic-mod* and *errtrap-mod*. The last two are also provided on the distribution disc in the single module files LOGIC and ERRTRAP respectively. Two other modules *exptran-mod* and *told-mod* in the files TOLD and EXPTRAN can be optionally loaded; more about this later.

micro-mod is the main module. It exports several relations that can be used as auxiliary supervisor commands or directly in workspace programs when the module is present.

The commands are similar to those provided by the SIMPLE system of chapter 5. The major difference is that the clauses that can be entered and edited using MICRO are essentially standard syntax clauses as described in chapter 2. The one elaboration is that expressions can be used as arguments to = and # conditions when *exptran-mod* is present.

logic-mod exports the definition of *ISALL* (see chapter 7). The *errtrap-mod* module should not be deleted unless it is replaced by an alternative error handling module. It exports the relation called *"?ERROR?"*, which if defined, is always called by the micro-PROLOG interpreter when it encounters a run-time error (see section 3.7 for more information). The error handling allowed by *errtrap-mod* is quite sophisticated and is described below.

As mentioned in the last chapter, *exptran-mod* exports a relation called *Expression-Parse* that is used to compile expressions into relational form. This module need not be loaded if the use of expressions is not required.

Finally, the *told-mod* module exports a relation called *is-told* which can be used in a straightforward way to ask questions about certain conditions whilst a query is being evaluated. The use of *is-told* is documented in chapter 5 and so will only be briefly described in this chapter. It can be called from the error handler *errtrap-mod* in the same way that it can be invoked from the *deftrap-mod* error handler described in chapter 5.

Any program developed under the SIMPLE system can be loaded and queried when using MICRO.

Warning
If you want to switch from using SIMPLE to using MICRO, or vice versa, you must first kill all the modules of the SIMPLE system before you load MICRO. This is because the modules of the two systems export common relation names. The attempt to load a module that exports a name already exported by another module gives an error.

6.1 The relations exported by micro-mod

To use the facilites of MICRO do a `LOAD MICRO` command. To get rid of it you must kill each of its modules with

```
KILL logic-mod
KILL errtrap-mod
KILL micro-mod
```

6.1.1 add

`add` has two uses:

1. `add <clause>`

will add the clause to the end of the list of clauses for its relation. Except when the `exptran-mod` module is present, and the clause contains expressions (see later), it is equivalent to just entering the clause in the manner described in chapter 2.

2. `add n <clause>` n a positive integer

This adds a clause at the position indicated by the positive integer. It is equivalent to

`?((ADDCL <clause> m))` where m is n-1

if the clause contains no expressions. This means that if there are already at least n-1 clauses for the relation the added clause becomes the new nth clause. It is inserted before the old nth clause if there is one. If there are not already n-1 clauses for the relation it is added as a new last clause. Notice that this is slightly different from the `Add` of SIMPLE.

For the MICRO `add` the number is the position *before* which the clause is added whereas for the SIMPLE `Add` the number is the position *after* which the clause is added. The difference is related to the different kind of clause editor in MICRO.

6.1.2 delete

This also has two usages:

1. `delete <clause>`

Again, except when the clause contains expressions, it is equivalent to

```
DELCL <clause>
```

with the added feature that if there is no such clause you get the message "No such clause".

2. `delete <relation name> n` n a positive integer

will delete the current nth clause for <relation-name>. It is equivalent to

```
?((DELCL <relation name> n))
```

with the added feature that if there is no current nth clause, it displays the message "No such clause".

6.1.3 edit

The `edit` command can be used to text edit a clause and/or reposition a clause. Its use is:

`edit <relation name> n` n a positive integer

The result is that the nth clause for the named relation together with the number n will be displayed as

```
n <clause>
```

and both can be edited using the line editor. If n is changed, it is taken as the new position for the edited clause, i.e. the position that would be used in an `add` command. If the relation that the clause is about (i.e. the relation of the head atom) is changed then the new n is taken to be the position for adding the changed clause to the sequence of clauses for the new relation. The old clause is always deleted.

The main difference between `edit` and the `t` command of the structure editor is that any compiled expressions in the clause are de-compiled back to expressions before the clause is displayed. This de-compiling only takes place if `exptran-mod` is present. If it has been killed then the clause will be displayed with all expresssions in their compiled form.

6.1.4 cedit

This has the same form of use as `edit`. The difference is that the old clause is *not* deleted. This is useful for building up a definition of a relation where the clauses have common components (cf. the `c` command of the structure editor). As with `edit`, compiled expressions are de-compiled and displayed as expressions if `exptran-mod` is present. They are then re-compiled on exit from the edit.

6.1.5 kill

This has the same uses as the primitive *KILL* command of the supervisor. The only difference is that to delete the entire workspace program

kill all (note the lower case *all*)

is used and you are asked to confirm that the entire program is to be deleted. The program space that is available when it is cleared is also displayed. This space is the value given by the built-in micro-PROLOG *SPACE* relation. The other uses of *kill* are:

kill <relation name>
kill <module name>
kill <list of relation names>

They also report the successful deletion of the corresponding programs.

6.1.6 list

Again, this has the same uses as supervisor command *LIST* with the exception that a request to list the entire workspace program is

list all (with lower case *all*)

Compiled expressions are displayed in their compiled form.

6.1.7 load

load <file name>

This has the same effect as supervisor *LOAD <file name>* command. The only difference is that after a load the free space left is given.

6.1.8 save

save <file name>

This has exactly the same meaning as the supervisor *SAVE <file name>* command. It saves the entire workspace program on the named file.

6.1.9 reserved

A call *(reserved x)*, *x* a variable, will result in *x* being bound to a list of all the names exported by the currently loaded modules. Useful for reminding you of the names of these exported relations which are names you cannot use for your own program relations.

The attempt to add a clause for a primitive relation or a relation exported by a loaded module results in the *Cannot add clauses for ...* error message. Nor can you avoid this error by adding clauses for the exported

relation before you load the module. This will result in the `Illegal use of modules` error message. The programs inside modules are protected. To change them you should use the module utility described in chapter 4.

6.1.10 space

The command

`space.` (the `.` or some other argument term is needed)

will give the current space left as a number of *K* bytes. It is equivalent to the query:

```
?((SPACE x)(PP x K free))
```

which uses the primitive `SPACE` relation.

6.1.11 *is*

Use is

```
is <list of atoms>
```

except when the list of atoms contains expressions, which it compiles before evaluating the atom list, its effect is almost the same as the supervisor query command

```
? <list of atoms>
```

However, if the evaluation is successful, `YES` is displayed. If the evaluation fails, `NO` is displayed.

6.1.12 which, all

Use is

```
which (<term> <atom1> <atom2>...<atomk>)
```

Again, any expressions are compiled before the command is executed. The effect is the display of `<term>` for each different solution of the query

```
?(<atom1> <atom2>...<atomk>)
```

Example

```
which((x y) (APPEND x y (1 2 3))
```

produces the answer

```
(() (1 2 3))
(1) (2 3)
(1) (2 3)
```

```
(1 2) (1)
(1 2 3) ( )
No (more) answers
```

APPEND is a relation defined in and exported from *micro-mod*. (See below.)

all is an accepted synonym for *which*

Example

```
all((y son of bill) (parent-of bill y) (male x))
```

produces an answer such as

```
(john son of bill)
(peter son of bill)
No (more) answers
```

6.1.13 one

Use is similar to *which*

```
one(<term> <atom1> <atom2>...<atomk>)
```

It will also display *<term>* for each solution of the sequence of atoms given in the query. The difference is that after each solution is displayed, it waits for a prompt to continue. If you enter *c*, it will give the next solution, if any. Any other response stops the evaluation. Again any expressions in the query condition are first compiled.

6.1.14 accept

accept enables a sequence of single atom clauses to be entered for a relation to be entered quite quickly. Its use is :

```
accept <relation name>
```

You will then receive the relation name as a prompt and you need only enter the list of arguments for the single atom clause that you want to enter. You can continue in this way until you enter *end*.

Example

```
&.accept male
male.(tom) {(tom) is entered, male. is the prompt}
male.(bill)
male.(john)
male.end
```

will add the clauses

```
((male tom))
((male bill))
((male john))
```

to the end of the *male* program.

6.1.15 APPEND, ON, true-of

We have already mentioned that *micro-mod* defines and exports the list processing relation *APPEND*. Its definition is:

```
((APPEND () X X))
((APPEND (X ! Y) Z (X ! x))
  (APPEND Y Z x))
```

micro-mod also exports another useful list processing relation *ON*. It can be used to find members of a list or test for membership. Its definition in *micro-mod* is:

```
((ON X (X ! Y)))
((ON X (Y ! Z))
  (ON X Z))
```

true-of has the definition:

```
((true-of X Y)
  (X ! Y))
```

You never need to use this relation in programs entered using MICRO. Any use of a call *(true-of X Y)* can be replaced by *(X ! Y)*. However, the relation may have been used in some sentence form of a clause entered using the SIMPLE extension of the supervisor described in the last chapter. It is included in *micro-mod* so that programs developed using SIMPLE can be loaded and queried when using MICRO. (See the remark at the beginning of the chapter.)

6.1.16 CONS, @, #, =, function

The definitions of the *CONS* and @ relations are:

```
((CONS X Y (X ! Y)))
```

```
((@ X ! Y)
  (X ! Y))
```

They are used in expressions. The # and = relations are only used together with expressions and are described in the next section; *micro-mod* does not actually contain a definition of =, but it cannot be defined in a user program because it has a special role in the syntax of MICRO programs; *function* is the command that is used to declare a relation as a function before using it in expressions. (See section 6.2.3 below and 5.3.3.)

6.1.17 *, %, +, –

The *micro-mod* module exports the following definitions for the auxiliary arithmetic relations *, %, +, -. *TIMES* and *SUM* are primitive arithmetic relations of micro-PROLOG implemented in machine code. The auxiliary relations are recognised as arithmetic operators in expressions.

```
((+ X Y Z) (SUM X Y Z))
((- X Y Z) (SUM Y Z X))
((* X Y Z) (TIMES X Y Z))
((% X Y Z) (TIMES Y Z X))
```

The definitions are the same as those given in 5.3.9 which are exported from *Simple*.

6.2 Expressions in MICRO clauses

You can use expressions as arguments to certain calls in the body of clauses that are entered using the *add*, *edit* or *cedit* commands described above. The expressions can be used as arguments to calls that use the two relation names = and #.

The three commands just mentioned scan each clause before adding it to the workspace. If any call has the relation name =, or the relation name # followed by a first argument which is a constant, the expression arguments of the call are compiled into a list of relation calls which becomes an argument of a compiled form of the original = or # call. You see this compiled form when you *list* or *LIST* the clause.

However, when you edit the clause using *edit* or *cedit* the compiled call is mapped back into its source form before it is displayed. The *delete* <*clause*> command also compiles = and # calls before it scans for the clause to delete.

Expression arguments to = and # calls are also allowed in *ask*, *which* and *all* queries. Expressions are compiled and de-compiled using the relation *Expression-Parse* which is exported from the module *exptran-mod*. Before you use expressions you must load the module with a

```
load EXPTRAN
```

command.

6.2.1 The # relation

Source calls to #, i.e. calls that will be compiled, have the form:

```
(# R E1 E2..Ek)
```

where *R* is a relation name and *E1*,..,*Ek* are expressions. The expressions are

the arguments of the relation call, which is really to the relation *R*. The # signals that these arguments to *R* are not just ordinary terms, but are expressions, terms that contain calls to functions and arithmetic operations. An # atom in a MICRO clause is the equivalent of an expression condition in a SIMPLE sentence. The atom *(R # E1 ... Ek)* would be the expression condition *R#(E1 ... Ek)*

Example

```
(# LESS (2 * 3) (5 + 7))
```

is a call to the relation *LESS* with arguments the values of the expressions *(2 * 3) (5 + 7)*. This source call will be compiled into the target call

```
(# (LESS X Y) (( * 2 3 X) (+ 5 7 Y)))
```

in which a list of atoms that will evaluate the expressions *(2*3)(5+7)* appears as the second argument. The first argument is the call to the relation *LESS* with arguments the variables which have the values of *(2 * 3)* and *(5 + 7)* when this list of atoms is evaluated.

This # call is logically equivalent to the three ordinary calls:

```
(* 2 3 x) (+ 5 7 y) (LESS x y)
```

The definition of #, which is exported from *micro-mod*, is

```
((# X Y)
    (? Y)/
     X)
```

Thus # evaluates the atom list *Y* before it evaluates the call *X*. Note the use of the backtracking control primitive */*. This prevents backtracking on the atom list evaluation, i.e. on the evaluation of the arguments to the call *X*. However, it does not prevent backtracking on the evaluation of *X*.

Example

```
(# salary x (2 * y))
```

will be compiled into

```
(# (salary x z) ((* 2 y z)))
```

If the value of *y* is known at the time the call is evaluated, the * call will compute *z* and the evaluation of the # call will reduce to the evaluation of

(salary x N), N some number

Backtracking will result in different values being sought for *x*, but not in the re-computation of *N*.

6.2.2 The = relation

Uses of = have the form

```
(= E1 E2)
```

where *E1* and *E2* are expressions. It is equivalent to

```
(# EQ E1 E2)
```

i.e. to a call of the primitive *EQ* relation with arguments the values of the expressions *E1*, *E2*. An = atom in a MICRO clause is the equivalent of an equality condition in a SIMPLE sentence.

Example

```
(= (2 * x) (3 + y))
```

is equivalent to

```
(# EQ (2 * x) (3 + y))
```

and will be compiled into

```
(# (EQ X Y) ((* 2 x X) (+ 3 y Y)))
```

The *EQ* relation is a primitive of micro-PROLOG and is defined by the single clause

```
((EQ X X))
```

So, a = call evaluates its arguments and then unifies them. When the values are numbers, as here, this amounts to checking that they are identical. When one of them is a variable, as in

```
(= x (234/23))
```

it will result in *x* being bound to the value of the expression (234/23) i.e. 10.173913.

6.2.3 Syntax of expressions

We refer the reader to section 5.3.1 for the syntax of expressions as well as for information about how a relation *R* may be declared a function, using the

```
function R
```

command, and then used in expressions. This command adds a

```
((func R))
```

clause to the workspace program. The *func* clause is not automatically deleted when you *kill* the relation *R*, so you should delete it yourself if you do kill a relation that has been declared a function.

As with SIMPLE, you can recover from the error of not having declared a relation as a function before using it in an expression in a clause. You will get the warning message from *exptran-mod*

```
R assumed not to be a function
```

edit the clause and immediately exit from the editor. The expression is de-compiled before it is displayed and then re-compiled on exit. This time the use of *R* as a function name will be recognised.

6.2.4 Killing exptran-mod

The *exptran-mod* module occupies between 2 and 3K of program memory. After it has been used to compile expressions in the clauses of some entered program it can be killed. You will not of course be able to use expressions in any of the queries to the program. If you do, or if you forgetfully *add* a clause containing expressions when *exptran-mod* is not present, you will get error of the form

```
No clauses for (Expression-Parse <expression> <var>
<var>)
```

This is because the *add* command has found some use of an expression and has tried to call the *Expression-Parse* relation exported by *exptran-mod*. The error message has been displayed by the *errtrap-mod* error handler. You can recover from the error by loading the EXPTRAN file which contains just *exptran-mod*. The details are in the next section.

You will not get this error message when you try to edit a clause with compiled expressions even though *Expression-Parse* is normally also called by the edit commands to de-compile expressions. This is because before *Expression-Parse* is called to de-compile expressions a check is made to see if the *exptran-mod* module is present.

If *exptran-mod* is not present, the expressions are displayed in their compiled form. This means that you can use *exptran-mod* whilst you are developing a program and want to use the convenient shorthand provided by expressions. Then, for serious use of the program where space might be at a premium, you can kill the module. You can re-load the module as required.

6.3 The error handler errtrap-mod

The module *errtrap-mod* exports the relation *"?ERROR?"* as described in section 3.7. If this relation is defined by some loaded module or workspace program, the micro-PROLOG interpreter will call the *"?ERROR?"* program when it encounters a run-time error. The program for *"?ERROR?"* given in *errtrap-mod* then displays a message of the form:

<short phrase describing the error> <atom of the call>
`error&(? for info).`

The short phrase for the error is similar to the one given by the example `"?ERROR?"` program given in section 3.7. The `error&` is the prompt that you will continue to get whilst in an error trap state. The query evaluation that caused the error is suspended. There are now various options that allow recovery action to be taken and the suspended evaluation to be resumed.

You can obtain a brief description of these options by entering `?`. You will then get displayed:

```
to quit enter : q
to fail call enter : f
to succeed call enter : s
to line edit call and resume enter : e
or enter / <any command> (eg / add <clause> , /
load file)
or enter : tell
to resume enter : c
error&.
```

q response

This quits the suspended evaluation and returns you to the supervisor. After the `q` you will get the normal supervisor prompt `&.` and you can use any of the supervisor or MICRO supported commands in the normal way.

f response

The suspended query evaluation is resumed but with the error-invoking call assumed to have no solution, i.e. to have failed.

s response

Same as for `f` except the call is assumed to have been solved with its current arguments.

e response

The offending call will be re-displayed to be edited with the BBC micro screen editor. You can edit the call in any way but if you want to leave variables in the call you should use the variable names displayed in the call. When you type RETURN, the evaluation will be resumed but with the offending call replaced by its edited form.

Note: you have not edited the program. You have only changed the call for this one execution in order to avoid the error. Generally, you should also have edited the clause that lead to the error using the `/ <command>` response, or

you should edit the program when the current query evaluation finishes.

/ <command> response

The */* is the escape character that enables you to enter any supervisor command or MICRO command. This must be the command name followed by its arguments. (You cannot just enter a clause. You must use *add* or *ADDCL* to do this.) You can thus edit your program, list it, load files, or add new clauses. An example of a */ load file* recovery response is given below.

tell response

This calls the *is-told* relation exported by the module *told-mod* with argument the offending call which can then be answered interactively in the manner described in section 5.5. It is useful for top-down programming. You can use relations in your program before defining them. Then, when you get the error *No clauses for* when the a call on the relation is reached you can provide the answer or answers to the condition in response to the *is-told* prompts without having to give the definition. But note that your answers are not saved. A subsequent use of the relation in the current query may force you to give the same answers. The only way to avoid this is to load a file with a program for the relation, or to add some clauses to define it. The use of *tell* is also in many cases an alternative to editing the call.

c response

This should only be given after you have executed one or more commands that will ensure that the error will not occur when the evaluation is resumed. For example, you have added some clauses for a relation after getting the *No clauses for* error message and are now ready to continue with the suspended evaluation. The evaluation is resumed at the point where the offending call was tried, so the call is re-evaluated.

If you enter any other response you will get the *?* message.

6.3.1 Example error recovery

1. Suppose that you have killed the *exptran-mod* that exports the *Expression-Parse* relation and then you use an expression in a query or added clause. You will get an error message of the form:

```
No clauses for (Expression-Parse <expression> <var>
<var>)
error&(? for info).
```

The *.* is the prompt for you to enter a response. The following will recover from the error.

```
error&(? for info). / load EXPTRAN
error&.c
```

2. Suppose that you have misspelled a relation name in some call of a program clause, using say `parentof` instead of `parent-of`. You will get an error message of the form:

```
No clauses for (parentof .. ..)
error&(? for info).
```

There are several ways to recover.

(a) You can use `e` to edit the call and change `parentof` to `parent-of`; but remember this change only applies to this call. If the clause with the misspelled relation is used again you will get the error again.

(b) You can add a clause defining `parentof` as `parent-of` and then resume with the responses:

```
/ add ((parentof X Y)(parent-of X Y))
c
```

This avoids the error on this and all subsequent uses of the incorrect clause. After the current query evaluation is over you can find the misspelling, edit it and then delete this added clause. This is perhaps the best recovery response.

(c) You can list the clauses in which you think the misspelling appears and then edit the offending clause. This removes the problem for subsequent uses of the clause but not for any currently active uses which includes the current error. To avoid a recurrence of the current error, either do (b), deleting the new clause when the current evaluation is finished, or edit the call, or enter `tell` to invoke `is-told` with argument the offending call.

6.4 Using the is-told relation

`is-told` is a multi-argument relation exported from the module `told-mod`. When evaluated it displays its sequence of arguments followed by a `?` and waits for a response. If there is just one argument, which is a list, this is displayed without the outer brackets. The responses and their effects are fully described in 5.5. Before using `is-told` you must load the module with a

```
load TOLD
```

command.

6.4.1 Example use of is-told

These are the same examples as given in 5.5 but using the query commands of MICRO and its clause syntax:

1. `all( (percent z) (is-told mark x outof y) (= z (x/ y*100)))`

sets up an interaction that can be used to convert pairs of numbers to percentages. A possible interaction is:

```
mark X outof Y ? ans 20 40
(percent 50)
mark X outof Y ? ans 15 60
(percent 25)
mark X outof Y ? just 40 120
(percent 33.333333)
No (more) answers
```

2. `((is-male x) (known-male x))`

```
((is-male x) (NOT known-male x)
             (is-told (x a male)) (ADDCL ((known-
             male x))))
```

defines `is-male` in such a way that the user is queried whenever an `is-male` condition is encountered with argument given but not recorded as a `known-male`. A `yes` response to the question such as

```
keith a male ?
```

results in the `(is-male keith)` condition that invoked the query being solved and a `((known-male keith))` assertion being added to the program. A `no` response results in the condition failing.

6.4.2 Killing told-mod

If you do not want to use `is-told` in your programs or queries, or use the `tell` error response, you can get rid of the `told-mod` module saving about 1K of program space. Even if you do this and then forgetfully use the relation you can recovery from the error by re-loading the module from the TOLD file. You can even recover if you use the `tell` error recovery response with the module deleted. The interaction will be something like:

```
No clauses for (father-of tom X)
error&(? for info).tell
No clauses for (is-told ((father-of tom X)))
error&(? for info). / load told
error&.c
(father-of tom X) ? .
```

and you can now continue by answering the `(father-of tom X)` question in the manner as described above. You have recovered from an error

encountered in an error recovery action! You are now out of the error state.

6.5 External relations

Note that the *EXREL* facility depends upon the *SEEK* primitive of micro-PROLOG. Please check in appendix C that this is implemented for your computer. If it is not available then please ignore this whole section.

The EXREL utility described in 4.5 can be loaded and used in conjunction with MICRO. The only constraint is that the *listex* and *listfile* commands must not be used as they require *query-mod* module of SIMPLE to be present. Their clause form equivalents *LISTEX* and *LISTFILE* can be used instead.

6.6 Tracing and structure editing

Both the TRACE and SPYTRACE utilities described in chapter 4 can be loaded and used in conjunction with MICRO. If you use TRACE it might be useful to kill the optional modules *exptran-mod* and *told-mod* before tracing in order to gain space. You can re-load them after the trace.

The structure editor in EDITOR can also be loaded and used with MICRO. It is described in chapter 4.

7 Built-in programs

A special feature of the built-in programs in micro-PROLOG is that they model as closely as possible program-defined relations. For example, the *SUM* relation can be viewed as though it were defined by a set of facts about addition, and the *TIMES* relation as though it were defined by the various 'times tables'. This is because the built-in programs attempt to simulate the different patterns of use of the relation; and the *SUM* built-in program is able not only to add up numbers, but also to subtract them.

For reasons relating to efficient implementation micro-PROLOG compromises on the ideal of supporting every possible use and generally allows only some of the possible uses of its built-in programs. In particular, the assembler-coded built-in programs only support the deterministic uses of the relations they represent.

However, in general, each built-in program has several uses. This helps to minimise the number of names the programmer has to know, and also helps to keep micro-PROLOG programs 'invertible' – able to support different patterns of use for the relations they define.

If a particular call to a built-in program is an illegal use (for example if *SUM* is called with two or more arguments as variables) then the system raises an error 3 which signals an invalid form of use. An error of this kind usually occurs only if there are too many variables in the call.

The 60 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. For the relations that are implemented entirely as micro-PROLOG programs embedded in the supervisor we also give its defining program. It is the program that will be displayed if you *LIST* the relation.

In introducing each relation we give the form of use, a comment (inside "{}" brackets) which gives a brief description of its meaning, and the restrictions, if any, on its use. The restrictions concern the arguments that must be known, or the arguments that must be variables, when a condition for the relation is evaluated. When the argument can be any term we indicate this by using a "t" or "t1" etc. in that argument position in the form of use. When the argument must always be known at the time of evaluation, and must be a particular type

of value, we give the value type in the form of use. For example, if a particular argument to a relation must be a number at the time of evaluation the type `<number>` will appear in that argument position in the form of use.

Remember that each primitive unary relation can also be used as a supervisor command – see chapter 3.

7.1 Arithmetic relations

The arithmetic relations *SUM*, *TIMES*, *SIGN*, *INT* and *LESS* cater for the normal operations on integer and floating point numbers of addition, subtraction, multiplication, division and comparison.

7.1.1 SUM

`(SUM x y z)` {x+y=z} at least two arguments must be given, given arguments must be numbers

SUM is true of three numbers when the first two add up to the third. The numbers involved can be integer, floating point or a mixture of the two types. The *SUM* primitive can be used to:

1. *Check a sum:* If all arguments are given then SUM succeeds only if the first two numbers add up to the third. For example, `(SUM 20 30 50)` is true, as is `(SUM 1.5 0.3 1.8)` and `(SUM 23.6 -0.6 23)`.

2. *Add two numbers together:* If the first two arguments are numbers and the third a variable then the call succeeds by instantiating the third argument to the sum of the first two. For example, `(SUM 30 -2 x)` binds *x* to 28, and `(SUM 30 2.5 y)` will instantiate *y* to 32.5.

3. *Subtract two numbers:* If the third argument is a number, and either the first or the second is 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, `(SUMx315)` binds *x* to 12, as does `(SUM 3 x 15)`.

If an addition or subtraction results in an overflow (or underflow), then micro-PROLOG signals an arithmetic overflow (underflow) error. This causes any resident *"?ERROR?"* program to be invoked as for other kinds of signalled error (see appendix E).

Micro-PROLOG uses 9 digits of accuracy in its arithmetic calculations, and numbers can range from 2^{-39} to 2^{38} in absolute value. Rounding is performed if the number of significant digits in a calculation is more than 9.

7.1.2 TIMES

(TIMES x y z) {x * y = z}

At least two arguments must be given; given arguments must be numbers.

The *TIMES* relation can be used to multiply, divide or to check a multiplication.

1. *Check a product:* If *TIMES* is called with all three arguments 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, *(TIMES 3 4 12)* is true, as is *(TIMES 0.5 -3 -1.5)*

2. *Multiplication:* If *TIMES* is called with just the first two arguments given, and the third argument a variable, the call succeeds by binding the variable to the product of the two numbers. The call *(TIMES3-4x)* results in *x* being bound to −12.

3. *Division:* There are two forms of the *TIMES* program which can be used for division, both involve the division of the third argument of the *TIMES* call by either the first or second argument depending on which is known at the time. The remaining argument is instantiated to the quotient of the two given numbers.

(TIMES x 10 30), *x* is bound to 3
(TIMES 10 x 25) *x* is bound to 2.5

The division of an integer by another integer may of course result in a floating point number. Similarly if two floating point numbers are divided then it is possible that the quotient is a integer. Any necessary conversions between the two kinds of numbers are performed automatically.

If a division by zero is attempted then an *arithmetic overflow* error is signalled.

7.1.3 LESS

(LESS <number1> <number2>)

{<number1> is less than <number2>}

The *LESS* built-in predicate implements the inequality test for numbers. Only one **arithmetic** usage is allowed, where both arguments are given and 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, *(LESS23)* is true, as is *(LESS -1 1.32)*; but neither *(LESS 10 9)* nor *(LESS 4.4 4.4)* are true.

LESS can also be used to compare constants, see below.

7.1.4 INT

(INT <number> y) {y the nearest integer to <number>
not greater than <number>}

y must be a variable at call

The *INT* primitive can be used to find the nearest integer to a given number. The nearest integer (truncating towards zero) is returned as the value of y. This two argument form of *INT* can only be used in this single mode. A one-argument form of use is described in 7.7.

If the nearest integer to a number cannot be represented exactly in 9 digits, it will be rounded. However, if the number is 2^{31} or more in absolute value, an *arithmetic overflow* error is signalled.

7.2 String operations

In micro-PROLOG strings can be represented and manipulated as lists of characters. A string in packed form is a constant. There is a primitive *STRINGOF* relation that can be used to convert between strings as character lists and constants. *LESS* can be used to test the lexicographical order of constants.

7.2.1 LESS

(LESS <constant1> <constant2>)

{<constant1> lexicographically less than <constant2>}

Similar to the inequality test for numbers, this test for constants tests that the first argument (which must be a given constant) is textually less than the second (which must also be a given constant). The ordering used is the lexicographical ordering, based on the ordering of the underlying character set (namely ASCII).

For example, *(LESSFREDFREDDY)* succeeds since *FRED* is lexically less than *FREDDY*.

7.2.2 STRINGOF

(STRINGOF x y) {x is a list of characters of constant y}

x must be a list or list pattern and *y* a constant
or *x* must be a character list and *y* a variable

The *STRINGOF* relation can be used convert a constant into a list of its constituent characters, or to pack a list of characters into a single constant. There are essentially two forms of use:

1. *Unpacking*: to produce a list of characters from a constant. In this use the second argument must be a constant (not a number or list) at the time of evaluation. The result is the unification of the first argument with the list of characters of the constant. If the empty constant "" is given, its list of characters is the empty list ().

`(STRINGOF x fred)` results in `x` being bound to the list `(fred)`, and `(STRINGOF x "A*")` binds `x` to the list `(A *)`.

For the unpacking use, the first argument may be given as a list or a list pattern. This allows comparison of a list of characters and the constant, and the use of a pattern allows a particular character of the constant to be picked up as the binding of a variable.

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

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

2. *Packing*. This takes a list of characters and produces a constant from it. It is the inverse of the unpack use.

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

7.2.3 CHAROF

`(CHAROF x y)` {x is character with code y}

at least one argument must be given

The `CHAROF` primitive implements a mapping between single character constants and the ASCII coding sequence. Three uses are allowed: where either `x` or `y` or both are known at the time of the call. Some examples of its use are:

```
(CHAROF A 65)
(CHAROF B x)    x is instantiated to 66
(CHAROF y 32)   y is instantiated to " "
```

CHAROF will fail unless its first argument (if already a constant) is exactly one character long.

7.3 Console input/output

The input/output facilities are divided into two groups: Console I/O and Disc I/O. Console I/O reads terms from the keyboard and displays them on the console. Disc I/O transfers terms to and from the disc system. In fact the console

I/O primitives are defined in terms of the disc I/O primitives as we shall see below.

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. However, in large measure the power of PROLOG as a systems programming language arises from its combination of declarative and imperative features.

There are four built-in programs for dealing with Console I/O: *R*(Read), *P*(Print), *PP*(P-Print) and *RFILL*, which respectively read a term from the console, print a sequence of terms, pretty print a sequence of terms and 'pre-fill' the keyboard buffer with a sequence of terms. *RFILL* allows the sequence of terms to be edited using the line editor on your computer (see appendix B for details).

7.3.1 R

```
(R x)   {x the next term typed on keyboard}
```

x must be a variable at time of call

The *R*ead program reads a single term 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 signalled. If there is already a term in the keyboard buffer (see chapter 3) this will be the value returned, otherwise the read prompt . is displayed and the evaluation will suspend until a term is entered.

Any variables in the entered term are converted to the special internal form for variables in which the variable name used in the term is discarded. However, different occurrences of the same variable in the term will be converted into the same internal form. The internal form for each variable of the term will be different from the internal form of any other variable in the program i.e. it will be an entirely new variable. It will also be different from the internal form assigned to a variable of any other term that has been read in, or will be read in, even if the variables have the same name. In micro-PROLOG the *scope* of a variable name is the term in which it appears.

See appendix B for details of how micro-PROLOG accepts terms from the keyboard.

The *R* primitive side-effects the keyboard buffer removing the first term from the buffer. A call to *R* only has one solution. A backtracking return to the call will not result in an attempt to find an alternative solution by reading in the next term. The second attempt to solve the call will fail. The call will be resolved only if an earlier call has an alternative solution and this leads to a fresh

attempt to solve the *R* call. At that point, the next term in the buffer will be read.

7.3.2 P

(P t1 t2 .. tk) {display t1 t2 .. tk on the console}

The *P*rint program displays its sequence of any number of arguments on the console output device. It takes any number of arguments each of which can be any term. The terms will be displayed separated by a single space. There is no automatic new line on completion. A subsequent *P* or *PP* will therefore start one character position after the preceding *P* finishes its display.

The *P* primitive displays constants in its argument terms as the sequence of their component characters. Thus a constant that would have to be quoted on input (see chapter 2) will be dispayed without the quotes. This means that any control characters in the constant, which appear as @<*char*> combinations in the quoted form, are sent to the output device as control characters. See appendix B for details for your micro.

Variables occurring in the sequence of displayed argument terms are given the names *X*, *Y*, *Z*, *x*, *y* ,*z*,*X1*,..,*z1*,*X2* and so on, corresponding to the order in which the variables are encountered during the *P*rint. (The first variable encountered is given the name *X*, the second one the name *Y* and so on.) The same internal form variable appearing in more than one term of the displayed sequence of terms will have the *same* print name. So, if the first variable that appeared in *t1* also appears in *t2*, it will be displayed as *X* in both terms. If there are more than 128 different variables, then subsequent variables are displayed as *???*.

7.3.3 PP

(PP t1 t2 .. tk) {Pretty Print t1 t2 .. tk on the console}

The *PP* program displays its sequence *t1* .. *tk* of arguments in a standard format. It is very similar to the *P* program except that a carriage return/line feed automatically occurs after the terms are displayed. In other words, PP always terminates the display of its arguments by sending the <CR> and <LF> pair of characters to the terminal. A subsequent *P* or *PP* starts at the beginning of a new line. Also *PP* displays constants that would need to be quoted on input in quoted form. This means that *PP* displays the terms in such a way as to guarantee that if it they were read back the same terms would be re-constructed (except for variables which are always new).

Control characters occurring in the quoted constants are displayed in the @<*char*> format. As an example of the difference between *P* and *PP*,

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

displays:

```
(The man
```

on the console, whereas the call:

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

displays:

"(The man" <CR><LF>

on the console.

Since the *PP* program always finishes by sending a return/line feed pair a call to *PP* with no arguments:

```
(PP)
```

will just move the cursor to the beginning of a new line.

7.3.4 RFILL

```
(RFILL (t1 t2 .. tk) x)
```

{clear then pre-fill keyboard buffer with t1 t2 .. tk and then read a term into x}

x must be a variable at time of call

The *RFILL* program is used to invoke the computer's line editor with the sequence of terms *t1 t2 .. tk* given as the elements of its first (list) argument displayed, on the screen. When editing is terminated the first term (usually the edited form of *t1*) is automatically read in to become the binding of the variable *x*. The edited forms of *t2 .. tk*, if they were given in the call, must be explicitly read in using *R* since they will be left in the keyboard buffer. *RFILL* displays constants in the *PP* format so that they can be read back in. Please see appendix B for details of using the line editor on your computer and see appendix C for information on any restriction there may be on the use of *RFILL*.

RFILL remembers the print names given to each internal form variable in *t1*. When the edited version of the displayed *t1* is read back in as the binding of *x*, each of these remembered variable names is converted back into the internal form variable that appeared in the original *t1*. Thus, variables names in the edited form of *t1*, which are the same after editing, become *the same internal variables* as those that appeared in the *t1* argument to *RFILL*.

This remembering of variable names is necessary for the use of *RFILL* for editing programs, particularly when it is used from within a structure editor to

edit a component of a clause. *RFILL* is used in this way by the structure editor described in chapter 4. It is invoked by the *t* command. *RFILL* is also used by the *e* error recovery command of the *errtrap-mod* module described in chapter 6. Again for this use, the remembering of variable names is essential.

Note that although a list of terms must be given as the second argument to *RFILL*, the list is displayed without the outer level of parentheses.

RFILL also has the side effect of clearing any previous contents of the keyboard buffer. If several terms had previously been typed on a line, then any 'unused' terms will be discarded when *RFILL* is invoked.

As we have already mentioned, *RFILL* is used by the *t* command of the structure editor. It is also used by the *edit* commands of both *SIMPLE* and *MICRO*. As an example, a simplified version of the *edit* of *MICRO* is defined by:

```
((edit x)                              {command: edit relation x}
      (R y)                            {read the clause number}
      (CL ((x|z)|z1) y y)              {find the yth clause of x}
      (RFILL (((x|z)|z1)) z)           {invoke line editor with the clause in
                                        the buffer}
      (ADDCL z y)                      {add the edited clause z to the
                                        program}
      (DELCL x y))                     {delete the old clause for x}
```

For an explanation of the other primitives see section 7.9.

7.4 File I/O

Micro-PROLOG supports disc files of text i.e. files of characters. These are accessed sequentially via the primitives *READ*, *W* and *WRITE* or randomly using the *SEEK* primitive (if implemented on your computer, see appendix C). According to the filing system, a certain maximum number of files may be open during an evaluation at any one time, an attempt to have more files open results in an error being signalled. Please see appendix D for details of file handling on your micro.

7.4.1 OPEN

(OPEN <file-name>) {open named file for reading}

This built-in program opens a file for reading. The <file-name> argument is a constant which names a file according to the file naming conventions.

If the file was previously open for writing then the file buffer is 'flushed' and the file is closed before being re-opened for the read. This means that it is not possible to simultaneously read *and* write to a file.

If the file was already open for reading then the file is 'rewound' to the beginning; however, it is not good practice to rely on this feature. If the file cannot be found, the OPEN call fails.

7.4.2 CREATE

(CREATE <file-name>)

{create and open a new file for writing}

The *CREATE* program opens a new file for writing.
If the file is already open for writing then an error is signalled.

7.4.3 CLOSE

(CLOSE <file-name>) {close down the named file}

There may be no special actions associated with closing a file which is being read; however, a file which is being written to must be explicitly closed down, otherwise the disc file may *not* contain the right data.

The *CLOSE* primitive performs this operation and releases the file from micro-PROLOG. It is also used to release, from micro-PROLOG, files opened for read access.

7.4.4 READ

(READ <file-name> x) {*x* next term on the named file}

x must be a variable at time of call

The *READ* primitive reads the next term from the named file and binds its argument to the term. It finds the term by reading characters from the current character position for the file. If the current character position is at the end of file, the *READ* fails. After the read, the current character position is immediately after the last character of the read-in term. The *READ* primitive is the same as the *R* primitive, except that it reads a term from a named file.

7.4.5 WRITE

(WRITE <file-name> (t1 t2 .. tk))

{write the sequence of terms t1 t2 .. tk on the named file in *PP* format}

The sequence of terms *t1 t2 .. tk* are written onto the file in the same form as the *PP* program. This ensures that any term written onto a disc file can be subsequently read back in as the same term, with the exception of variables which are renamed. At the end of the transfer of *t1 .. tk* a linefeed/return two-character sequence is sent to the file. So the file position at the end of the *WRITE* is not immediately after the character text for the last term *tk*, but is two more characters beyond that.

7.4.6 W

(W <file-name> (t1 t2 .. tk))

{write the sequence of terms t1 t2 .. tk on the named file in *P* format}

Exactly the same effect as *WRITE* except that the terms are written in the same way that *P* displays terms on the console. Note that terms written using *W* will not be re-readable as terms if they contain any constants that would need to be quoted on input. The *W* does not complete the transfer with a linefeed/return. The file position on completion is therefore immediately after the text for *tk*.

Both *W* and *WRITE* handle variables in the same way as the *P* and *PP* primitives. Each occurrence of a variable throughout the list of terms *(t1 .. tk)* is given the same print name in the *sequence* of terms written to the file.

7.4.7 SEEK

(check if implemented on your micro in appendix C)

(SEEK <file-name> x) {named file is at position *x*}

x a variable or file position at call

There are two modes of use, either the *x* argument is unbound on entry to *SEEK*, or it is a file position previously returned by an earlier call to *SEEK*.

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

The *SEEK* primitive can be used for files that are either opened for reading or for writing. However, 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 disc instead of in memory. The following is a simplified form of the *RPRED* definition exported by the EXREL utility described in chapter 4.

```
((RPRED x y z)                  {find clause with head y on opened file x
                                 starting search at position z}
      (SEEK x z)                {move file to z}
      (READ x y1)               {read in a candidate clause y1}
      (SEEK x z1)               {find new file position z1}
      (OR                       {either}
```

```
((EQ (y|Y) y1)|Y)        {unify candidate clause y1 with (y|Y)
                          and evaluated the body of the clause Y}
((RPRED x y z1))))       {or find another clause starting at z1}
```

To use this program to keep the clauses for a relation *likes* on the disc file LFILE we replace the clauses for *likes* in the program with the single clause:

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

The file LFILE must also have been OPENed prior to using the program for *likes*. The LFILE file can be generated either by using *SAVE* (see below) or be specially generated by a program that uses *WRITE*.

The *likes* clauses in LFILE can be quite general, including recursive clauses, because different invocations of *RPRED* can be accessing different segments of the file simultaneously. The use of *SEEK* means that each different use of *RPRED* remembers where it is in the file, and then explicitly repositions the file to that position when it is getting its next clause.

You can spread the clauses for a relation over several files (just use more than one rule as above) and you can mix *RPRED* definitions of relations with ordinary clauses. So, new information about a 'data base' relation can be recorded by a clause in memory until the backing store file can be updated. Programs like *RPRED* can be used to build intelligent data base systems within micro-PROLOG.

7.4.8 Special files

Your computers standard default input and output files (usually the keyboard and screen) are denoted by the special file name "CON:". This special file name can be used in any of the above file I/O primitives.

Finally, the *R*, *P* and *PP* commands are defined by the following supervisor programs using *READ*, *W* and *WRITE* and the special file name *CON:*.

```
((R X)
  (READ "CON:" X))

((P|X)
  (W "CON:" X))

((PP|X)
  (WRITE "CON:" X))
```

Note how the use of the meta-variable as the list of arguments (see chapter 2) makes *P* and *PP* multi-argument relations. The sequence of argument terms given in the call is passed on as a single list of terms to the *W* and *WRITE* primitives.

7.4.9 IOB

(IOB <file-name> X) {X is ASCII code of char in file}

Use 1: X an unbound variable.

The *IOB* primitive reads the next character from the named file and binds its second argument to the small integer that is the ASCII code of the character. After the read, the current character position is incremented by 1.

Use 2: X an ASCII code.

IOB writes the character whose ASCII code is supplied by the second argument, at the time of the call, to the named file. The code must be in the range 0 to 255. After the call to *IOB*, the current character position in the file is incremented by 1.

Warning:
Since files are read-only or write-only you cannot mix the two uses of *IOB*.

7.5 Computer specific primitives

Micro-PROLOG provides a number of special relations to allow access to the special facilities of your particular micro. See appendix C for your computer for details of these.

7.6 Record I/O

Formatted reading and writing (*FREAD* and *FWRITE*) are not supported by the 6502 version of micro-PROLOG.

7.7 Type predicates

The type predicates test a single argument for its type i.e. whether it is a number, constant, list or unbound variable.

7.7.1 NUM

(NUM x) {x is a number}

The *NUM* built-in predicate tests to see if its single argument is numeric or not. If it is a number the call succeeds, if it is an unbound variable, a constant or a list then the call fails. For example, (NUM 3) and (NUM -1.3e9) succeed (are true), whereas (NUM bill) and (NUM *x*), with *x* unbound, fail (are false).

7.7.2 INT

(INT x) {x is an integer}

The *INT* predicate, in its single argument form, checks its argument for being an integer. For a number whose absolute value is 2^{31} or more, the precision does not suffice to determine whether it represents an integer, and an *arithmetic overflow* is caused.

7.7.3 CON

(CON x) {x is a constant}

The *CON* built-in predicate tests if its single argument is a constant. For example, *(CON foo)* succeeds, whereas *(CON())*, and *(CON1)* both fail. If the argument is not a constant (including the case where it is a variable) then the call fails.

7.7.4 LST

(LST x) {x is a list}

The *LST* predicate is true of lists. This includes the empty list. If the single argument is not a list, (again including the case where it is an unbound variable), the call fails.

7.7.5 SYS

(SYS t) {t is an atom for a primitive relation}

SYS tests to see if *t* is one of the relations described in this chapter. It succeeds if it is, it fails if it is an atom for any other relation.

7.7.6 VAR

(VAR x) {at the time of the call x is an unbound 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.)

7.7.7 DEF

(DEF t) {t is an atom naming a defined program}

DEF tests to see whether its argument is a call to any program currently defined: this includes built-in programs and user-defined programs. It succeeds if it is, and fails if it is any other atom.

For example:

(DEF DICT)	will succeed, and
(DEF (fred X))	will succeed if a user program called *fred* has been defined.

7.8 Logical operators

The basic clausal form of logic programs, which is limited to the implicit 'and' between the calls in the body of a clause, is extended via various logical operators of micro-PROLOG. There are built-in supervisor programs that implement: disjunction *OR*, identity *EQ*, *NOT*, *IF*, *FORALL* and ! . In addition the extra logical operator *ISALL* is implemented as a micro-PROLOG program exported from the module *logic-mod* in the file LOGIC of the distribution disc. This module is also included in both MICRO and SIMPLE.

The logical operators can be used to increase the efficiency and the readability of micro-PROLOG programs. They also raise the level of the language, making it more powerful and expressive.

7.8.1 OR

```
(OR <atom list1> <atom list2>)
```

{either the query ?<atom list1>
or the query ?<atom list2> can be solved (is true)}

The disjunctive operator *OR* has two arguments: each of which is a list of atoms. An *OR* call succeeds if either of its component queries is solved. For example,

```
(OR ((father-of x y)) ((mother-of x y)))
```

succeeds if either `(father-of x y)` or `(mother-of x y)` succeeds (is true).

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

The supervisor definition of *OR* is

```
((OR X Y) | X)
((OR X Y) | Y)
```

This uses the 'meta-variable as the rest of the clause' form (see chapter 2). From the definition we can see that the 'either' branch is tried first (it is the first clause). Only when there are no more solutions to this branch will a solution to the 'else' branch be sought. As we saw in the example program of 7.6.2, the else branch can be used to specify the failure alternative to the 'either' branch. A / placed in the 'either' branch will cut out the 'else' branch option when it is evaluated. Thus, the / in

```
(OR ((test X)/(process X Y)) ((default X Y)))
```

will cut out the 'else' branch if `(test X)` succeeds. See below for more information on /. A / in any of the logical operators only has an effect on the backtracking within the operator. It has no effect on the backtracking

evaluation of the other conditions in the clause or query in which the logical operator appears.

7.8.2 NOT

(NOT <relation-name> t1 t2 .. tk) (A)

{the condition (<relation-name t1 t2 .. tk) fails (is false)}

The *NOT* operator implements negation-as-failure [Clark 1978]. It denotes the negation of the call

(<relation-name> t1 t2 .. tk) (B)

The negation (A) succeeds if, and only if, the unnegated call (B) fails.

There is an implicit assumption that the program definition of the relation is *complete*; that all the positive instances can be inferred from the program definition. On this assumption, it is correct to assume that a condition is false (hence that its negation is true) if an attempt to prove the condition fails.

A negated condition can only be used for testing, it cannot be used to find any values for variables in the condition such that the condition is false. Indeed, the more correct reading of the call is

show that
there are no x1,..,xn such that (<relation-name> t1 t2 .. tk)

where the *x1,..,xn* are all the variables in the call *at the time that it is evaluated*. This means that the reading and effect of a program which uses *NOT* can be effected by the position of the *NOT*.

As an example,

?((parent-of john x)(NOT male x))

is read: show that john has a child who is not (proveable) male

Whereas

?((NOT male x)(parent-of john x))

must be read:

show that
there is no x such that x is male and show that there is an x
who is a child of john

This reading is forced because the *x* in *(NOT male x)* will be unbound when the condition is evaluated. The *(NOT male x)* condition will fail if there is at least one way of showing that the *x* is a male.

The fact that `NOT`, implemented as 'failure to prove', is only an approximation to the logical negation is evident in the use of a double `NOT`. The queries:

```
?((parent-of tom X)(PP X))
?((NOT NOT parent-of tom X)(PP X))
```

are *logically* equivalent. However, with the first, the name of any found child of `tom` will be printed, with the second the unbound variable `X` will be printed. The double `NOT` succeeds if `(parent-of tom X)` succeeds but it will leave the variable `X` unbound. The first query is read:

show that
there is an X such that (parent-of tom X)
and display the found X

The second is read:

show that there is an X such that (parent-of tom X) and
then show that there is an X that can be displayed.

The second reading derives from the fact that the `X` in the negated condition is unbound when it is evaluated.

The positive aspect of this imperfection of `NOT` is that double negations can be used to test if a condition succeeds *without* binding any variable in the condition.

Rule of thumb: Place negated, test conditions on a variable (or variables) *after* positive conditions/calls that can be used to find values for the variable(s).

The definition of `NOT` is

```
((NOT ! X) X
      / FAIL)          {If X is proveable then (NOT ! X) must fail}
((NOT ! X))            {else, (NOT ! X) is proven}
```

Note the essential use of `/` to prevent the use of the second clause if `X` succeeds. It also makes sure that `X` is only proven once. Only when there is no proof of `X` will the second clause be used. This confirms `(NOT X)` without instantiating any variables – hence the restriction of `NOT` to test use.

7.8.3 IF

```
(IF <atom> <atom list1> <atom list2>)
```

{either <atom> and <atom list1> can be solved (are true)
or (NOT ! <atom>) and <atom list2> can be solved (are true)}

The `IF` is a conditional alternative. Its use is equivalent to

```
(OR ((<atom> / <atom list1>)) ((atom list2))
```

which uses / to prevent the use of the 'or' branch if the <atom> test is solved. The IF form is more declarative. The declarative equivalent using OR would be

```
((OR ((<atom> <atom list1>)) ((NOT | <atom>) <atom
list2>)
```

with an explicit (NOT | <atom>) on the 'else' branch. This form is much less efficient than the IF or the OR using / because of the repeated evaluation of <atom> in the second branch.

A definition of a relation R of the form

```
((R ...)(IF (P ...) (<bodyA>) (<bodyB>))
```

is (almost) equivalent to a definition using the pair of clauses:

```
((R ...) (P ...) <bodyA>)
((R ...) (NOT P ...) <bodyB>)
```

The difference is that in the IF definition the test (P ...) is only ever evaluated once.

The use of conditionals in this way is a mixed blessing because less use can be made of unification. For example in the two clauses for R above it could be that the heads of the two clauses would naturally be slightly different. 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 in the head to the terms that they should be. For example,

```
((sort (x y|Z) (x |Z1))
      (LESS x y)
      (sort (y|Z) Z1))
((sort (x y|Z) (y |Z1))
      (NOT LESS x y)
      (sort (x|Z) Z1))
```

must be absorbed into

```
((sort (x y|Z) Z2)
       (IF (LESS x y)
            ((EQ Z2 (x|Z1)) (sort (y|Z) Z1))
            ((EQ Z2 (y|Z1)) (sort (x|Z) Z1))))
```

which is much less readable. A non-declarative solution is to use / after the test of the first clause and to drop the test in the second.

```
((sort (x y|Z) (x|Z1)) (LESS x y) / (sort (y|Z) Z1))
((sort (x y|Z) (y|Z1)) (sort (x|Z) Z1))
```

The definition of *IF* uses */* in just this way. It is:

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

As with *OR*, a */* placed inside the 'then' or the 'else' branch of an *IF* only effects the backtracking within that branch.

7.8.4 EQ

(EQ t1 t2) {t1 is identical to t2}

The supervisor definition of *EQ* is

```
((EQ X X))
```

so the effect of the evaluation of an *EQ* condition is the attempted unification of its two-argument terms.

(EQ (x1 x2) (A B)) results in x1=A, x2=B.
(EQ (a|z) (x y c)) results in x=a, z=(y c).

7.8.5 ?

(? <atom list>) {true if the sequence of atoms in <atom list> can be solved (are all true)}

The supervisor definition of *?* is

```
((? x)|x)
```

? can be used when the syntax for a condition requires a single atom but you want to 'pass' a 'conjunction' of atoms. An example is the *IF* condition of which the test must be an atom. The form,

```
(IF (? <atom list>) (...) (...))
```

enables you to have several conditions for the test.

As a supervisor command *?* is the primitive query form of micro-PROLOG. The more elaborate query forms of MICRO and SIMPLE are all defined using *?*. For example, the following is the definition of the *which* of MICRO:

```
((which (X|Y))                    {X is the answer pattern, Y the
                                   query}
  (? Y)                           {solve Y}
  (PP X)                          {display answer pattern X for this
                                   solution}
  FAIL)                           {backtrack to find next solution}
((which (X|Y))                    {when first clause fails}
  (PP No (more) answers))         {there are no more answers to Y}
```

7.8.6 FORALL

```
(FORALL <atom list1> <atom list2>)
```

{for all the solutions of <atom list1>, <atom list2> can be solved}

`FORALL` is a very high-level concept, and can often replace explicit recursions in a program.

The following program defines 'prime number' using `FORALL`. It is a specification that can be used as a prime checking program.

A positive prime number *x* is a number such that none of the integers *y* in the range $2 <= y < x$ divide *x*. That is, for each *y* in this range it is not the case that *y* divides *x*. This definition is formalised as:

```
((prime x)
      (FORALL ((in-range 2 x y))
              ((NOT divides y x))))
```

where in-range and divides are:

```
((in-range x y x))
((in-range x y z)
     (SUM x 1 x1)
     (LESS x1 y)
     (in-range x1 y z))

((divides x y)
     (TIMES x z y)
     (INT z))
```

This program is not a very efficient program for checking prime numbers, but it is an obviously correct one.

A `(FORALL <atom list1> <atom list2>)` condition is solved if, and only if, the pair of conditions:

```
(? <atom list1>) (NOT ? <atom list2>)
```

does **not** have a solution. In other words, if there is no way of solving <atom list1> so that <atom list2> *cannot* be solved. Hence, the definition of `FORALL` is

```
((FORALL X Y) (NOT ? ((? X)(NOT ? Y))))
```

After the evaluation of a `FORALL` all variables in the <atom list> arguments of the condition are left unbound.

7.8.7 ISALL (exported from the module *logic-mod*)

(ISALL t t1 <atom1> <atom2> .. <atomk>) k>0

{t is the list of all the t1s such that the query *?(<atom1> ... <atomk>)* is solved}

ISALL can only be used to instantiate varables in *t*

Each element in the list *t* is a copy of the value the term *t1* given by each different solution to the query. At the end of the evaluation all variables in *t1* and the *<atom>* conditions are left unbound.

The solutions found are neither unique nor sorted: if a different solution to the query results in the same value of *t1* then a second copy of this value appears on the list *t*. The values for *t1* appear on *t* in the order in which the solutions to the query condition are found.

The first value of *t1* on *t* corresponds to the first solution of the query, the last value of *t1* on *t* corresponds to the last solution of the query. *t* is usually given as a variable in the call and the evaluation binds the variable to the list of solutions. However, *t* can be a list or a list pattern. If it is a list, the elements on *t* must be in the order that they would be found by an evaluation in which *t* was given as a variable, which is the order in which solutions to the query are found. Since this order is not easy to predict, this checking role for *ISALL* is not very useful. Example uses of *ISALL* are:

```
(ISALL x y (father-of tom y)(male y))
```

{makes x the list of all the sons of *tom*}

```
(ISALL (x.y) (z1 z2) (gives bill z1 z2))
```

{checks that *bill* gives at least one thing *z1* to someone *z2* and makes *x* the last *(z1 z2)* pair found, *y* the other solutions}

```
(ISALL (x1 y1) y (mother-of mary y))
```

{checks that *mary* has exactly two children, and finds their names in the list (x1 y1)}

```
(ISALL (bill) y (father-of tom y))
```

{checks that *bill* is the only child of *tom*}

You can use *ISALL* to define a *count* relation that counts the number of different solutions to a query.

```
((count X Y.Z)
   (ISALL X1 Y.Z)
   (list-count X1 0 X))
```

```
((list-count () y y))
((list-count (X|Y) y z)
  (SUM y 1 y1)
  (list-count Y y1 z))
```

7.8.8 !

```
(! <relation name> t1 .. tk)
```

{the first solution to (<relation name> t1..tk)}

The evaluation of the *!* call reduces to the evaluation of the atom that follows the *!* in the call. However, it restricts the evaluation to just one solution. On backtracking, no further ways of solving the condition are sought. The call

```
(! likes x tom)
```

should be read "the first x who likes tom".

! provides useful control information when you know that there will only be one way of solving the condition. It cuts out the redundant search on backtracking. It is particularly useful for test calls that will be evaluated against a large number of assertions. In the MICRO `which` query

```
which(x (parent-of tom x)(male x))
```

each found child of `tom` is checked for being male. The backtracking search results in alternative and redundant 'proofs' of the `(male x)` condition being sought before another child of `tom` is found. In consequence, the entire set of `male` assertions will be scanned for each found child. By using

```
which(x (parent-of tom x)(! male x))
```

the scan of the `male` assertions is abondoned as soon the found *x* is confirmed as a male.

The supervisor definition of *!* is:

```
((! | X) X /)
```

It is the */* that prevents the search for a second solution.

7.9 Data base operations

Micro-PROLOG has three primitives that enable clauses to be accessed, added and deleted from the user's workspace at runtime. The ability to add and delete clauses is needed in order to implement extensions to the supervisor. It also enables the data base to be used as a scratch pad memory. An example of this latter use is the definition of `ISALL` given above. Being able to pick up the clauses for a relation allows the definition of alternative query evaluation strategies. An example of an alternative evaluator is given below.

7.9.1 CL

This has two forms of use, a single argument form and a three-argument form:

`(CL X)`	{X is a clause in the progam}
`(CL X Y Z)`	{X is a clause at position Z in the sequence of clauses for its relation with Z >= Y}

At time of evaluation of either call `X` must be a clause or a clause pattern in which at least the relation name of the head atom is given as a constant. For the three-argument form `Y` must also be given as an positive integer, `Z` can be a variable or be given.

This program accesses clauses from the user's workspace (or the currently opened module). It 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 important constraint on `CL` is that the relation name of the head atom of the clause to be found must be given.

For example,

```
(CL((At|x)|X))
```

succeeds if there are any `At` clauses in the workspace so this form of use can be used to test if a relation is defined before a call to the relation is made. (A call to an undefined relation signals an error.)

In the example call, if there are clauses for `At`, variable `x` is bound to the arguments of the head atom of the first clause, and the variable `X` is bound to the (possibly empty) list of atoms that make up the body of the clause. Backtracking will result in an alternative (later) clause being sought for the `At` relation.

For the most general use, the clause argument is a pattern of the form

```
((<relation name>|Y)|Z)
```

The three-argument form of `CL` can be used to find particular clauses in the program, the second argument must be given. It gives the position of the clause from which the search commences. The last argument is the position of the found clause.

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

binds `X` to the position of the clause that matches `((likes John|x)|y)` with the search starting at the first clause. Backtracking results in an alternative clause being sought that matches the clause pattern. If there is one, its (later) position will be given as the value of `X` and the rest of its structure will be given in the bindings for `x` and `y`.

We can also use the three-argument form to pick up a specific clause. To do this, we use the most general pattern for a clause for the relation and give the clause position.

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

will bind *x* and *y* to the argument list and the body of the 4th clause for `likes` if it exists. If it does not, the call fails.

The single argument `CL` behaves almost as if a micro-PROLOG program is stored as a sequence of `CL` clauses with the actual program clauses as argument terms. The evaluation of a `CL` call is then a search through these `CL` clauses. This is almost true. The sequence of `CL` clauses is implicitly repesented by the relation names in the dictionary which 'point' to the sequence of clauses that define them.

However, the restriction that the predicate symbol must be known at the time of the call means that `CL` can only be used to search through the clauses of a *single* relation, it `cannot` be used to search through all the clauses in the workspace.

`CL` can be used to define a query evaluator.

```
((question ())          {a question with no conditions is true}

((question (X|Y))       {a non-empty question is true}
   (CL (X|Z))           {if there is a clause (X|Z) matching X}
   (question Z)         {whose body Z is true}
   (question Y))        {and the remaining conditions Y are true}
```

This makes use of `CL` to find a clause that solves *X* in such a way that the rest of the question can be solved. Failure to solve *Y* will first result in backtracking on the evaluation of the body *Z*, and finally on the search for an alternative matching clause. For questions and programs that do not use `/`, it is equivalent to the supervisor `?`.

By collecting all the matching clauses as a list, we can try the clauses in an order different from their order in the program. An alternative recursive clause for `question` is:

```
((question (X|Y))
  (ISALL z (X|Z1) (CL (X|Z1))) {z are the clauses matching X}
  (select (X|Z) z)             {(X|Z) is a selected clause}
  (question Z)
  (question Y))
```

The definition of *select* determines the order in which clauses for the condition X are tried. As an example,

```
((select x z)
   (sort z z1)
   (member-of z z1))
```

can be used to select the clauses in order of increasing number of atoms in the body given suitable definitions of *sort* and *member-of*.

Elaborations of this approach enable one to program breadth first evaluation of queries (as distinct from depth first with backtracking).

7.9.2 ADDCL

This also has two uses:

(ADDCL X) {add X as a new last clause for its relation}

(ADDCL X <int>) {add X as a clause after clause <int> for its relation}

For both uses, at the time of evaluation *X* must be a list term that satisfies the syntax of a clause. In particular the relation name of the head atom must be given as a constant. For the second use, *<int>* must be a non-negative integer.

The clause added is a copy of the list term *X* in which any unbound variables of the term become variables in the added clause. This convention, that unbound variables in the 'name' *X* denote variables in the clause to be added, is logically not very satisfactory. However, all PROLOG implementations use this convention. It enables clauses to be read in using the *R* primitive and then added to the program using *ADDCL*.

R converts variable names in the read-in clause term into internal form unbound variables. *ADDCL* then copies the clause term, mapping the unbound variables into special variable names in the added clause. (This *R*, *ADDCL* cycle is how the supervisor accepts new program clauses – see the definition of *<SUP>* below.)

When a clause is picked up by the micro-PROLOG interpreter during an evaluation, or when it is retrieved using *CL*, the special variable names of the clause are converted back into entirely new internal form variables, just as though the clause had been read-in. So each use of the clause gets a fresh copy, with a new set of variables different from any other variables currently in the evaluation. This allocation of new internal form variables (which are actually locations where pointers to values for the variables will be stored when the variables are bound) for the variable names of an accessed clause is exactly the same as the allocation of new locations for the variables of a procedure that

takes place in a conventional recursive programming language.

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 clause whose position is given as the second argument. If *0* is used as the clause number, the clause is inserted at the front of the program.

```
(ADDCL ((append () x x)) 0)
```

adds the clause

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

to the front of the *append* program. If a position is given that is beyond the last clause for the relation the clause is added as a new last clause.

Restriction
Clauses cannot be added for primitive relations nor for relations exported by some loaded module. An attempt to do so signals an error.

7.9.3 DELCL

`(DELCL X)`	{delete the first clause matching X}
`(DELCL <relation name> <int>)`	{delete the <int>th clause for <relation name>}

X must satisfy the syntax of a clause, in particular, the relation name of the head atom must be given.

The primitive *DELCL* is used to delete clauses from the work space (or currently opened module). In the first form the program is searched for a clause matching *X*. The first one found is deleted. If none is found, the call fails. In the second form the clause to be deleted is specified by a relation name and a clause position. Again, if the specified clause does not exist, the call fails.

```
(DELCL ((append ! x1) ! x2))
```

will cause the first clause for *append* (if there is one) to be deleted. However, it will also bind the variables *x1* and *x2* to the head arguments and body of the clause respectively. So this form of *DELCL* can retrieve information from the deleted clause. An example of this use is in the definition of *ISALL* given above. In particular, the definition of the auxiliary relation *update* uses *ADDCL* and *DELCL* to manipulate the data base as a scratch pad memory.

Restriction
Clauses for primitive relations or relations exported from loaded modules cannot be deleted. An attempt to do so signals an error.

7.9.4 Undoing the effect of ADDCL or DELCL

The effect of an *ADDCL* or a *DELCL* is not undone on backtracking over the call. The adding or deleting of a clause is a side effect on the state of the program in the same way that *R* and *PP* are a side-effect on the state of the terminal. We can never undo the effect of a *R* or *PP*, but we can undo the effect of *ADDCL* or *DELCL* by doing the opposite.

The following clauses define 'soft' *addcl* and *delcl* relations whose effect is undone on backtracking. This is subject to the proviso that a */* has not been evaluated after the call that cuts out their 'else' branches.

```
((addcl X)
   (OR ((ADDCL X)) ((DELCL X) FAIL)))

((delcl X)
  (OR ((DELCL X)) ((ADDCL X) FAIL)))
```

Both relations need the clause to be unambiguously specified – to be uniquely determined by the argument *X*. If it is not, the *DELCL* of the *addcl* definition may delete a different clause that happens to match *X*, and the *ADDCL* of the *delcl* definition may add a more general clause than the one deleted. The alternative branch of each definition is the one evaluated when a failure causes backtracking. The FAIL transmits the failure to cause the necessary backtracking on preceding conditions.

Notice that *delcl* may not add the clause back in the same position so it is only useful when the position of the clause is not important. This is usually the case when you are manipulating a data base of single atom clauses.

7.9.5 KILL

(KILL R)	{delete all clauses for relation R}
(KILL (R1 .. Rk))	{delete all clauses for each of R1..Rk}
(KILL ALL)	{delete all clauses from workspace or all those owned by the currently opened module}
(KILL <module name>)	{delete the named module}

R and *R1 .. Rk* must be relation names.

The *KILL* primitive deletes all clauses for a single relation, all those for a list of relations, or all the clauses that can be deleted. When working in the workspace (the supervisor prompt is *&.*) only user relations in the workspace can be *KILL*ed. A *KILL ALL* will delete all the clauses from the workspace but will not delete any loaded module. This must be deleted with the last form of use *KILL <module name>*.

When working in an opened module (see below) *KILL* can be used to delete all clauses for relations that are owned by the module. The main use of *KILL* is as a supervisor command.

7.10 Library procedures

These allow saving of programs, loading programs and listing programs at the console.

7.10.1 LIST

(LIST R)	{list program for relation R}
(LIST (R1 .. Rk))	{list program for each of R1 .. Rk}
(LIST ALL)	{list programs for all workspace relations or all relations owned by currently opened module}
(LIST <module name>)	{list the named module}

R R1 .. Rk must be relation names

The *LIST* primitive lists the specified programs, in a standard format, at the console.

```
(LIST ALL)
```

lists the whole program, (apart from any loaded modules), and

```
(LIST (Likes Fred Angie))
```

lists the programs for *Likes*, *Fred* & *Angie*.

Modules are listed in the form in which they are *SAVE*d – see below.

7.10.2 SAVE

(SAVE <file name>)	{save entire program in named file}
(SAVE <file name> (R1 .. Rk))	{save only programs for given relations}
(SAVE <file name> <module name>)	{save named module in named file}

For the first form of use the entire program saved will be the workspace program if working in the workspace. If working in an opened module it will be all the clauses owned by the module. The supervisor command

```
SAVE TEST
```

saves the entire program on the disc file TEST.
Programs are saved as a sequence of clauses in the same format in which they

are dispayed by *LIST*. Modules are saved in a special format – see the next section on modules. It is also the format in which modules are displayed by *LIST*.

The second two forms of use cannot be used directly as supervisor commands as they have two arguments. For a command to save a module, you must use a query of the form:

?((SAVE <file name> <module name>))

SAVE does not delete the saved program(s). It also automatically opens the file for writing and then closes it on successful completion.

Warning
File names, relation names and module names must be distinct. If you inadvertently use a name for more than one of these roles you will get an error. If you attempt to save a program on a file, the file name you use must not be the same as any current relation name or any loaded module.

The error can occur when you are loading a file with the *LOAD* command described below and the *LOAD* reaches a clause for a relation that has the name of a loaded module or a currently opened file. On the error the *LOAD* will be abandoned with the file left open. So you should *CLOSE* the file from which you were *LOAD*ing if this occurs. If you are using an error handler (see appendix A) the offending relation name will appear in the *ADDCL* call that caused the error.

To avoid this type of error use different forms of name for relations, modules and files. Most of the micro-PROLOG modules on the distribution disc have a <NAME> of the form *<name>-mod*, where <NAME> is the file in which they are supplied. If you always use module names ending with *-mod*, you should avoid the error.

7.10.3 LOAD

(LOAD <file name>) {load the program from named file}

This program reads the disc file, and adds all the clauses in the file to the workspace or the currently opened module. If the file contains a saved module, the clauses are loaded as a module and do not enter the workspace.

Note: a module can only be loaded when working in the workspace (the supervisor prompt is *&.*). You cannot load a module when working in an opened module. The attempt to do so signals an error. You can only load into an opened module files consisting entirely of program clauses. The supervisor command

```
LOAD TRACE
```

loads the program in the disc file TRACE on the currently logged-in disc.

As each clause on the file is read in, it is added to the end of the current program for its relation. If there are clauses for the relation before the load, all the loaded clauses will be added after the existing clauses.

LOAD automatically opens the file and then closes it on successful completion of the *LOAD*. See the warning given in 7.10.2.

7.10.4 LISTP

`(LISTP <file name>)`	{list entire program to named file}
`(LISTP <file name> (R1 .. Rk))`	{list only programs for given relations to file}
`(LISTP <file name> <module name>)`	{list named module to named file}

LISTP is the built-in program used to implement both *LIST* and *SAVE*, and can be used to list the entire program, selected programs or individual modules to any file. Its arguments are exactly the same as in *SAVE*, as can be seen from the way *SAVE* is defined in micro-PROLOG:

```
((SAVE X↓Y)
   (CREATE X)
   (LISTP X↓Y)
   (CLOSE X))
```

For completeness, it is interesting to note that *LIST* is defined as:

```
((LIST ALL)
  /
  (LISTP "CON:"))
((LIST X)
  (LISTP "CON:" X))
```

LISTP is used whenever program clauses are to be written in a reasonably readable format, and is available so that user-defined versions of *SAVE* and *LIST* programs can be written.

7.11 Module construction facilities

Micro-PROLOG has facilities for constructing modules. These are named micro-PROLOG programs (sequences of clauses) which communicate with workspace programs and other modules via import/export name lists. Names used in the module that are not in the import/export lists are local to the

module and are invisible from outside the module. Different modules can therefore use the same local names with no name clash. On the other hand, constants that do need to be communicated across the module *must* be in the import or export list.

A module has five components:

name : which is a micro-PROLOG constant.

export list : a list of constants which are being 'made available' outside the module. These are usually the names of relations of the module. An exported relation can be used in a workspace program or command as though it were a primitive relation when the module is loaded. An exported relation name can also be imported by another module and used by the other module as though it were a primitive.

import list : a list of constants that the module imports from the outside. This must include the names of relations defined in the workspace or exported from some other module that need to be accessed from inside the module. It must also include all the 'data' constants used in the module that need to be communicated across the module e.g. constants which may be used in calls to exported relations of the module and must be recognised by the module or constants used in the module to calls to imported relations which will need to be recognised outside the module. Such communicated 'data' constants can be given in either the import or export list, but it is good practice to restrict the export list to exported relation names.

local dictionary : constants appearing in the module which are private to that module

module program : all the clauses owned by the module. These are the programs for all the relation names of the export list and the local dictionary.

The export list, the import list, and the local dictionary have no names in common. The relations accessible by a module are the relations defined by the clauses it owns and the relations with names in the import list.

Modules are `LOAD`ed and `SAVE`d automatically by the `LOAD` and `SAVE` programs. For the `LOAD` the same form of of call is used, `(LOAD <file name>)`, even if the file contains a module. This is possible because files containing modules have a different structure to ordinary program files and this is recognised by the `LOAD` which then handles the file in a special way. For the `SAVE` there is a special three-argument form for modules:

```
(SAVE <file name> <module name>)
```

This saves the named module on the specified file in the form:

```
<module name>
<export list>
<import list>
.
. {clauses owned by the module}
.
CLMOD
```

The local dictionary is not saved because this is automatically reconstucted by the *LOAD*.

Warning
if you use a text editor to develop a module program file the terminating *CLMOD* must be followed by a <return>.

The *LIST* program can also be used to list the contents of a module:

```
(LIST <module name>)
```

will display the module at the console in the form that it is saved on a file.

You can enter and exit loaded modules with the *OPMOD* and *CLMOD* commands described below. Entering a module makes it the *current* module.

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

```
module.
```

The top-level prompt *&.* is in fact the name of the root module *&* which is also the workspace. The root module has a special role. On loading micro-PROLOG the root module becomes the current module. It exports no names but it imports all the names exported by the loaded modules. As a module is loaded, the import list of *&* grows. Finally, other modules can only be loaded when *&* is the current module.

A supervisor *LIST ALL* will list all the clauses owned by the current module, and all clauses entered at the keyboard or added using *ADDCL* are added to the current module. An attempt to add or delete a clause for an imported name of the current module signals an error.

It **is** possible to have a program in a module which when called adds clauses to an imported relation of the module but the program **cannot** be called while the module is the *current* module. For example, the SIMPLE front-end program imports the relation name *dict*. It maintains this relation for the user by adding and deleting clauses from it. It can do this because when the *dict*

clauses are added and deleted the root module *&* is the current module and *dict* is not an imported relation of the workspace – it is a relation *owned by* the workspace.

The local dictionary of the current module is the dictionary used by all the I/O primitives of micro-PROLOG. When a constant is read-in, it is looked up in the local dictionary of the current module. If it is not present, it is added to the local dictionary. This means that any module program that is to be called from a workspace query, which reads in and tests for certain constants in the input, must import all the constant names it needs to recognise. For, when they are read-in, *&* will be the current module and the constants will enter the local dictionary of the workspace. The program in the module will not be able to recognise them unless it imports their names.

There are four primitives connected with modules: *CMOD*, *CRMOD*, *OPMOD* and *CLMOD*.

7.11.1 CMOD

(CMOD x) {x is the name of current module}

x must be a variable at call

The *CMOD* program simply returns the name of the current module. It is used, for example, by the supervisor to print out the name of the module as part of its top-level prompt.

7.11.2 CRMOD

(CRMOD <module name> <export list> <import list>)

{create an empty module and make it the current module}

CRMOD creates a new module with name *<module name>* and with the given export and import constant lists. It then enters it, i.e. makes it the current module. The *<export list>* and the *<import list>* are as defined above.

CRMOD can only be used when *&* is the current module. The *<module name>* cannot be the same as the name of any relation accessible by *&*. It must also be different from the name of any other current module and any opened file. Finally, none of the exported names can be the names of workspace relations or relations exported by other current modules. If any of these constraints are flaunted the *Illegal use of modules* error is raised.

Having created a module you can enter clauses into the module using all the facilities of the resident supervisor described in chapter 3. But note that you will not be able to *LOAD* and use the structure editor (of chapter 4) or any of the

other program development utilities whilst in the new module. The only way you can use the editor to help develop or modify a program inside a created module is to import the name *EDIT* and all the names of the edit commands into the module. The editor must be loaded whilst at the root module (*&*) level, but it can be called from inside the new module.

A more convenient method of developing a program that is to be wrapped up in a module is to develop it first as a workspace program which is saved in the normal way. You then use *CRMOD* to create the shell for the new module. On entry, *LOAD* the saved workspace program, then exit the module with the *CLMOD* described below. You can now *SAVE* the new module using the special form for modules. Alternatively, use the *MODULES* utility described in chapter 4 which supports the construction and modification of modules.

CRMOD is used by the *LOAD* primitive when it encounters a module name in the file. It creates a module using the name and the export and import lists that immediately follow the name. It then reads in all the clauses that follow up to the end of module mark *CLMOD* in the file. Each clause is added to the newly created module, which temporarily becomes the current module. On reaching the *CLMOD* the module is exited and the current module is again the *&* workspace module. If there is an error on loading, or if you interrupt the load with an ESCAPE, you will find that you are in the temporarily entered module.

7.11.3 OPMOD

(OPMOD <module name>) {make named module the current module}

OPMOD enters the already existing named module and makes it the new current module.

7.11.4 CLMOD

CLMOD {close current non-root module and return to *&*}

CLMOD drops out of the current module back into the root*&* module. It is not possible to drop out of the root module. As a supervisor command *OPMOD* must be given an (ignored) argument e.g.

```
CLMOD.
```

with the . the ignored argument.

7.12 Miscellaneous predicates

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

7.12.1 DICT

`(DICT x y z : X)` {y z and X are respectively, the export list, the import list, and the local dictionary of the current module x}

The `DICT` relation defines the dictionary of the current module. If you do a `LIST DICT` you will get a clause of the above form listed. You can also call `DICT` to pick up any of its arguments. Note that `DICT` is a multi-argument relation. After the import list `z` is a sequence `X` of all the local constants of the current module. Garbage collection will remove from `X` all constants no longer in use.

7.12.2 /

`/` {always true but with a side effect on evaluation}

The `/` primitive is used to control backtracking. When executed as a call in the body of an invoked clause its effect is to cut out all the, as yet, untried clauses for the call which invokes the clause, and all the alternative untried evaluation paths for the calls that precede the `/` in the clause.

Suppose that a clause of the form

```
((R ...)(R1 ...)(R2 ...)/(R3 ...))
```

is invoked by some `R`-atom call. The calls `(R1 ...)` and `(R2 ...)` are evaluated in the normal way with any necessary backtracking in order to find a solution to both calls. If they can both be solved, the `/` is executed. Slash always 'succeeds'; it is used for its side effect.

It suppresses all further backtracking on the evaluation of the `(R1 ...)` and `(R2 ...)` calls that would normally occur on a subsequent failure or when trying to find all the solutions to a condition. It also prevents any later, as yet, untried clauses for `R` from being used to solve the particular `R`-atom call that invoked the clause.

In other words, if the `(R3 ...)` call should now fail, then the call that invoked the clause also immediately fails. If the slash was not there, a failure of `(R3 .. )` would result in alternative ways of solving the `(R1 ...)` and `(R2 ...)` calls being explored, and then to alternative clauses for R being tried. The `/` cuts out all these alternatives.

The slash is useful for 'telling' micro-PROLOG that there are no alternative solutions to be found once the evaluation has reached the `/` point in the clause. This saves wasted time searching for non-existent alternative solutions, and also enables micro-PROLOG to save space.

The `/` actually side effects the evaluation stack. It discards all the activation

records back to the activation record for the invoked clause containing the /. This removes all the backtracking information for the evaluation of the calls that precede the / in the clause. Finally, it side-effects the backtracking information associated with the invoked clause so that it appears to be the last clause for the call. The / is also used in an essential way to implement some of the logical primitives of micro-PROLOG.

Used in a ? query a / prevents backtracking on the calls that precede it in the query once a solution to each of them has been found. This is because the ? is defined by

```
((? X) ; X)
```

and so a query

```
?(A1 .. Ak / ...)
```

becomes the evaluation of the body

```
A1 .. Ak / ...
```

for the single clause for ?. The / cuts out the backtracking options for A1 .. Ak. There are no other clauses for ? to be suppressed.

You can use a ? call when you want to cut out backtracking on preceding calls but not prevent the use of later clauses. Thus, the evaluation of the / in

```
((R ...) (? ((R1 ...)(R2 ...)/)) (R3 ...))
```

will cut out alternative solutions to (R1 ..)(R2 ..) but will allow the use of untried clauses for the invoking R call. A / placed inside an atom list argument to a logical primitive has a similar 'local' effect.

7.12.3 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 signal a "No clauses for" error.

7.12.4 ABORT

ABORT {abort the current supervisor command}

The ABORT primitive cancels the current supervisor command that is being executed and returns to the supervisor.

7.12.5 /*

(/* t1 t2 .. tk) {t1 ... tk is true}

The /* is the comment predicate. It ignores its sequence of any number of argument terms (including none) and always succeeds with *no* side-effect. /* can be used within clauses to provide comments or as a 'no-op' relation.

Its definition is equivalent to the program

```
((/* !X))
```

7.12.6 SPACE

(SPACE x) {x is the no. of K bytes left in workspace}

The *SPACE* primitive returns in its single argument the number of Kilo-bytes(1024 bytes) of space that are currently left in the workspace.

Before actually returning this number it calls the internal garbage collector, which has the effect of making sure that all the known garbage is removed.

7.12.7 <SUP>

<SUP> {the ever-running supervisor program}

The supervisor control program can be listed as the program for the predicate symbol *<SUP>*. It is a very simple program with just a few clauses. Its main function is to add clauses that are entered and to call other programs invoked by commands. Its definition is given in chapter 3. It should *not* be called from a user program.

7.12.8 ED

(ED <relation name> <clause no.>) {Edit a clause}

The designated clause for the named relation will be displayed ready for editing using the cursor keys of your computer (see appendix B). When you press **RETURN** the new version of the clause will be read in and will replace the old version. You can interrupt the edit by hitting the *ESCAPE* key for your computer (see appendix B).

8 The MITSI extension to the supervisor

by Jonathan Briggs

The file MITSI provides an alternative 'educational' environment in which to learn the concepts of programming in micro-PROLOG. This interface has been designed to provide most of the facilities of SIMPLE, with a uniform (object relation object) syntax, meaningful variable names and punctuation driven (rather than keyword driven) commands. Lower case is used throughout.

These notes are not intended as a complete guide to programming in micro-PROLOG using MITSI. A step-by-step tutorial, 'micro-PROLOG Rules!', designed for children and teachers, has been written by the author and is available from Logic Programming Associates Ltd.

8.1 Loading MITSI

To use micro-PROLOG with MITSI, enter micro-PROLOG and type the command

LOAD MITSI

The following prompt will be displayed:

```
>>>
```

This indicates that the MITSI supervisor is running. Once loaded it is not possible to escape to micro-PROLOG.

8.2 Leaving MITSI

To leave micro-PROLOG and return to the operating system, type:

```
>>> quit! {exclamation mark indicates this is a command}
```

MITSI will ask you to confirm that you really want to quit, allowing you to save your current programs.

8.3 The structure of sentences in MITSI

8.3.1 Facts

Using MITSI, facts have the form

object relation object

An object can be either an atom or a list enclosed in brackets. There are no

alternative prefix or postfix forms e.g.

```
John likes Mary
6 sum-of (2 4)
```

8.3.2 Rules

Rules have the similar form to those in SIMPLE :

<sentence> if <condition> and <condition> and ...

where sentence and condition both have the same (restricted) structure as facts. Additionally, conditions can have the form

not object relation object

e.g.

```
John is happy if Mary likes John
John friends-with Mary if John likes Mary and Mary
likes John
John is sad if John likes Mary and not Mary likes
John
```

8.3.3 Variables

MITSI uses the convention that words starting with some... or any... are variables e.g.

```
someone, anyone, some-temperature, any-x, some-1,
some, any
```

Warning
MITSI will reject sentences with micro-PROLOG variables `x, y, z, X, Y, Z` etc. These can be used as constants if enclosed in quotes.

8.4 Punctuation driven commands

Instead of SIMPLE keywords 'add', 'is' and 'which' punctuation is used to indicate the action required. Every line must end with a punctuation mark (. or ! or ?). The computer will not respond until one of these is typed. As with SIMPLE and MICRO, however, a sentence can be entered over more than one line and hence the punctuation mark may be typed even after ENTER or RETURN has been pressed.

8.4.1 Adding facts or rules

The fact or rule is typed followed by a full stop.

e.g.

```
>>> John likes Mary .

>>> John is happy if Mary likes John.

>>> someone is happy if Mary likes someone.
```

8.5 Commands

Commands are followed by an exclamation mark. Their action is similar to the same commands in SIMPLE.

8.5.1 list all!

To list all the sentences for all relations in workspace. MITSI displays these sentences numbered (within relations) preceded by relation names.

e.g.

```
>>> list all!

likes:
1 John likes Mary
2 Fred likes Mary

friend-of:
1 some-person friend-of some-one if
              some-person likes some-one and
              some-one likes some-person
```

Note that MITSI retains the variable names used.

8.5.2 list <relation>!

To list the sentences for one relation

e.g.

```
>>> list likes!

likes:
1 John likes Mary
2 Fred likes Mary
```

8.5.3 delete <relation> n!

This removes the nth sentence from the sentences about <relation>. The number can be found by listing the relation first.

e.g.

```
>>> delete likes 1!
```

will remove the first **likes** sentence.

8.5.4 kill all!

This removes all sentences from the workspace. MITSI asks for confirmation that everything is to be deleted.

e.g.

```
>>> kill all!
everything {yes/no} ? yes
Entire program deleted 17k free!
```

Typing *no* will leave the current program unchanged.

```
>>> kill all!
everything {yes/no} ? no
OK...nothing removed!
```

8.5.5 kill <relation>!

This removes all sentences about a <relation> from the workspace. MITSI removes these sentences without asking for confirmation.

e.g.

```
>>> kill likes!
```

will remove all sentences about the **likes** relation.

8.5.6 edit <relation> n!

This command allows you to alter the nth sentence for <relation>.

e.g.

```
>>> edit likes 1!
1 (John likes Mary)
```

The line is displayed preceded by its position in the relation (i.e. the first sentence has position 1). This number can be altered to change the position of the sentence. The name of the relation or the names of variables can also be changed.

The editor uses the same commands as those for SIMPLE and MICRO (see appendix B).

8.5.7 edit!

MITSI keeps a record of the last line typed. **edit!** allows this to be altered. There is no position number at the start of the line.

e.g.

```
>>> John likes Fred.
>>> edit!
(John likes Fred)
```

`John likes Fred` is not deleted by this form of `edit`, even if it is changed. `kill` or `delete` must be used to remove unwanted sentences.

8.5.8 load <filename> !

To load a previously saved program, type:

```
>>> load <filename> !
```

The filename conventions are the same as SIMPLE and MICRO. Programs loaded in this way are added to existing sentences in workspace.

8.5.9 save <filename> !

The current program can be saved for future use with this command. The <filename> must be different from any relations in the program. Saving a program does not delete it from the workspace.

e.g.

>>> **save fred!**

will save the current program in a file called FRED.LOG

8.6 Asking questions

Questions are asked by typing a question mark after the sentence.

The computer will answer the question in two different ways. If there are no variables in the question, the computer will respond `YES` or `DON'T KNOW` depending on whether the question can be answered from the facts and rules in the program.

e.g.

```
>>> John likes Mary?
YES
```

If there are variables in the question , the computer will find answers and will, in addition to YES, display the question with the answers filled in, in place of the variables.

e.g.

```
>>> John likes someone?
YES John likes Mary
No(more)answers
```

If there are no answers to a question containing variables, or when all answers have been displayed the computer will display this message `No(more)answers`.

A question can be in the form of a conjunction

e.g.

```
<question> and <question> and... ?
```

Second, and subsequent parts of a conjunctive question can be preceded by 'not'

e.g.

```
<question> and not <question> and....?
```

8.7 why?

MITSI includes a built-in **why** question that will try and explain answers to questions. MITSI remembers the last thing typed and if this is a question typing **why ?** will cause this to be explained.

e.g.

```
>>> John likes Mary?
YES
>>> why?
```

asks the computer to explain `why John likes Mary?`

`why` can also be used followed by a question.

e.g.

```
>>> why Fred friend-of Jim?
Fred friend-of Jim because

Fred likes Jim and
Jim likes Fred
```

MITSI uses the rules in programs to explain its answers. This means that `why` can be used to follow how a program is being executed. If `why?` is used following a question that produces many answers, only the first answer is explained.

e.g.

```
>>> some-one likes some-one-else?
YES John likes Mary
YES Fred likes Mary
No(more)answers

>>> why?
```

will explain `John likes Mary` only.

8.8 Formatted answers to questions

The answers to questions can be formatted by preceding the question by a template. The templated question has the form

printing <pattern> , <question> ?

The <pattern> can be made up of text or variables. The variables used must be the same as those in the question.

e.g.

```
>>> printing some-name, John likes some-name?
Mary
No(more)answers
```

```
>>> printing some-name this is text, some-name
likes Mary?
John this is text
Fred this is text
No(more)answers
```

8.9 Built-in programs

Instead of the built-in micro-PROLOG programs SUM, TIMES, LESS, EQ, etc., the following set of arithmetic and list processing programs have been provided.

8.9.1 sum-of

This is used to:
1. add two numbers together
2. check that two numbers added together produce a third
3. perform a subtraction

```
<number> sum-of (<number> <number>)
```

e.g.

```
>>> 6 sum-of (4 2)?
YES
```

```
>>> some-n sum-of (5 3)?
YES 8 sum-of (5 3)?
No(more)answers
```

```
>>> 7 sum-of (5 some-n)?
YES 7 sum-of (5 2)
No(more)answers
```

The same restrictions exist on *sum-of* as on the micro-PROLOG SUM. Only one of the numbers may be replaced by a variable.

8.9.2 prod-of

This is used to:
1. check that two numbers multiplied together give a third
2. find the result of multiplying two numbers together
3. perform a division

```
<number> prod-of (<number><number>)
```

e.g.

```
>>> 8 prod-of (4 2)?
YES
>>> some-n prod-of (5.3 2)?
YES 10.6 prod-of (5.3 2)
No(more)answers
>>> 64 prod-of (8 some-n)?
YES 64 prod-of (8 8)
No(more)answers
```

As with the micro-PROLOG program TIMES the use is restricted to cases with a maximum of one variable.

8.9.3 string-of

This is used to:
1. check that a list of characters make up a word
2. convert from a list of characters to a word
3. convert from a word to a list of characters

```
<list of characters> string-of <word>
```

e.g.

```
>>> (fred) string-of fred?
YES
>>> some-list string-of John?
YES (John) string-of John?
No(more)answers
>>> (Mary) string-of some-word?
YES (Mary) string-of Mary
No(more)answers
```

Only one of the arguments may be replaced by a variable.

8.9.4 less

This is used to:
1. check that one number is less than another

2. check that one letter precedes another in the alphabet
   in reverse to check 'greater'

```
<number> less <number>  or  <word> less <word>
```

e.g.

```
>>> 4 less 7?
YES

>>> Fred less John?
YES
```

This program can only be used for checking. It cannot be used to find values less than a certain number.

8.9.5 equals

This is used to check that two objects are the same

```
<object> equals <object>
```

e.g.

```
>>> John equals John?
YES

>>> 2 equals 3?
DONT KNOW
```

Only one of the arguments may be replaced by a variable.

8.9.6 belongs-to

This is used to:
1. check that an object is a member of a list of objects
2. find elements of a list

```
<object> belongs-to <list>
```

e.g.

```
>>> John belongs-to (Fred Mary John Sue)?
YES

>>> someone belongs-to (Fred Mary John Sue)?
YES Fred belongs-to (Fred Mary John Sue)
YES Mary belongs-to (Fred Mary John Sue)
YES John belongs-to (Fred Mary John Sue)
YES Sue belongs-to (Fred Mary John Sue)
No(more)answers
```

8.9.7 appends-to

This is used to:
1. check that two lists can be joined together
to give a third
2. find two lists that can be joined together
to give a third

```
(<list><list>) appends-to <list>
```

e.g.

```
>>> ((a b c) (d e f)) appends-to (a b c d e f)?
YES

>>> some-pair appends-to (1 2 3 4)?
YES (() (1 2 3 4)) appends-to (1 2 3 4)
YES ((1) (2 3 4)) appends-to (1 2 3 4)
YES ((1 2) (3 4)) appends-to (1 2 3 4)
YES ((1 2 3) (4)) appends-to (1 2 3 4)
YES ((1 2 3 4) ()) appends-to (1 2 3 4)
No(more)answers
```

Appendix A

Entering micro-PROLOG

The micro-PROLOG 3.1 distribution ROM comes with disc containing the following 16 files:

TRACE	SPYTRAC
EDITOR	MODULES
EXREL	LOGIC
SIMPLE	DEFTRAP
EXPTRAN	TOLD
SIMTRAC	PROGRAM
MICRO	ERRTRAP
MITSI	HILOG

All these files contain micro-PROLOG programs wrapped up as modules. They provide optionally loaded extensions to the built-in supervisor. Their facilities are described in various sections of this manual.

The facilities of the first six are described in chapter 4, the next six are described in chapter 5, and the next two (MICRO and ERRTRAP) are the subject of chapter 6. MITSI is described in chapter 8. Note that SPYTRAC and SIMTRAC are referred to as SPYTRACE and SIMTRACE in the main body of the manual.

To enter the micro-PROLOG system:

1. Make sure that the micro-PROLOG ROM is in a paged ROM socket of the machine.

2. Turn on the machine.

The following banner should appear at the console:

```
micro PROLOG 3.1
99999 Bytes Free
&.
```

The first two lines form the micro-PROLOG banner, and give details of the release and version number. The message "99999 BYTES FREE" indicates how much memory is allocated for the work space.

Warning
Once you are running micro-PROLOG, pressing the **BREAK** key will cause micro-PROLOG to restart but with all of your loaded programs lost. This can be a useful facility, but beware of pressing **BREAK** accidentally!

The free memory space 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 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.

There is no fixed division between the stack and the heap. They grow towards each other and garbage collection of the heap occurs when they are about to collide. When garbage collection does not create enough space to continue, you get a `No space left` error message and the current evaluation is aborted with a return to the supervisor.

Because of the trade-off between the heap and stack you can create space for an evaluation by deleting all unnecessary relation definitions and modules before the evaluation. Thus, utility modules that have been used to help develop and/or edit the program can be deleted to make room for the evaluation. This ability to load, then kill, then re-load support software as required is a significant feature of the program development environment of the micro-PROLOG system.

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

If the micro-PROLOG ROM is present in your machine together with other language ROMS, micro-PROLOG may not be entered immediately the machine is switched on. Micro-PROLOG can be entered from another language by passing the command PROLOG to the OS command line interpreter (see documentation for the language concerned). From BASIC this will be achieved by typing

```
* PROLOG
```

Note for 6502 Second Processor users
If you are using a 6502 Second Processor, you will be able to use a special "HI" version of PROLOG which is provided on disc. To do this, type

```
* HILOG
```

from BASIC, or

```
** HILOG
```

if you are already in PROLOG. This will give you about 15000 more bytes free for PROLOG.

Appendix B

Keyboard control and line editor

When reading from the keyboard the system prompts the user for input with a . prompt and then backspaces the cursor over the . (the top-level *&.* prompt that micro-PROLOG gives you on entry is made up of the workspace dictionary name *&* followed by the . read prompt). The first character typed always erases the . prompt.

As characters are typed in response to the read prompt they are stored in a special *keyboard buffer* and are only 'read' by micro-PROLOG after the **RETURN** is pressed. Until you hit **RETURN**, you can edit the sequence of characters using screen editing keys. In particular, you can construct the text to be entered using the arrow and **COPY** keys from any text currently displayed on the screen in addition to using the keyboard keys.

The keyboard buffer holds up to 250 characters, so a 'line' of up to 250 characters can be edited by the screen editor. All these characters will not appear on the same console *display line*. When the display reaches the end of a line, the display should automatically move to a new display line without the need to press **RETURN**.

The **DELETE** key will move back over different display lines, so, only press **RETURN** when you are confident that you do not want to edit the sequence of characters you have typed in response to the read prompt. Pressing **RETURN** makes that sequence an *entered line* which cannot be edited.

Note: the automatic transition from one display line to another is a function of the terminal/computer. Micro-PROLOG has no knowledge of the transition – it is not a token boundary (see 2.10). If the token that finishes at the right end of one display line needs to be separated from the one that starts the next line you need to type a space in front of the token on the second line.

Type ahead

Even before an input prompt is displayed, i.e. before an input request is given, you can enter up to about 20 characters by typing ahead. The characters will not be echoed until the input request is given. If more than 20 characters are typed ahead then some characters may be lost. It is generally best to wait for the *&.* prompt before entering commands or clauses.

Multi-line terms and right-bracket prompts

Constants, variable names and numbers *cannot* be split across entered lines because **RETURN** is a separator. List terms can be split over more than one entered line.

If you press **RETURN** before the list term is complete you will get a prompt of the form *n*., where *n* is the number of right brackets needed to complete the list. For example, we might have entered the *App* clause

```
((App () x x))
```

over four entered lines:

```
&((      RETURN
2.App(   RETURN
3.)x x   RETURN
2.))     RETURN
&.
^  (cursor position)
```

The *2*, *3* and *2* on the second, third and fourth lines are the bracket count components of the read prompts.

The . of these three lines will actually not be seen when the text is entered. It will have been erased by the first character typed. However, we shall always show the . in our example interactions.

The bracket count prompt is very useful when entering lists, in particular, clauses. If you are unsure about the number of closing right brackets you need, and you are sure that you do not want to do any editing of the current line, enter a **RETURN**. The prompt will tell you how many right brackets you need to finish of the list.

Copying to the printer

If you press **CTRL B** all screen output will be copied to the printer until you press **CTRL C**. So if you wish for a copy of your program to appear on the printer then you can use this before you LIST the program. Remember to issue the appropriate "Fx 5, n" command to set up your printer.

Suspending the listing

You can make the screen listing suspend after it has filled one display screen by using the Control-N facility. If you press **CTRL N** at any time the listing will be automatically suspended when a screen has been filled. You can get the next screen by pressing **SHIFT**. You turn off this screen at a time display mode by pressing **CTRL O**.

Control characters

Special effects can often be achieved by sending control characters to a particular device. `P` can be used to send a quoted constant containing the control sequence to the device.

To put a control character in a quoted constant enter~ `character` where `character` is the character of the key you would press in the control-key combination for the control character. The character must be a upper-case letter (between "`A`" and "`Z`") or one of the characters `"[" "#" "]", "|" "^" or "<-"`. Thus, to insert **CTRL A** enter `~A`. Micro-PROLOG will convert this `~A` in the quoted constant into ASCII control code for **CTRL A**. *When you list the program* the occurrence of **CTRL A** in the constant will be displayed as `~A`.

As an example, the call

```
(P "L")
```

or the supervisor command

```
P "L"
```

will clear the text area.

`IOB` can also be used to send control characters.

Interrupting execution

The ESCAPE key referred to in chapter 3 is the **ESCAPE** key on the keyboard. This breaks into a current evaluation and causes error 11 to be signalled.

Restarting micro-PROLOG

To restart micro-PROLOG press **BREAK**.

This will remove all loaded micro-PROLOG programs and modules. It is exactly as though you had just entered micro-PROLOG.

Appendix C

Special primitives

Micro-PROLOG provides a number of special relations to allow access to the unique facilities of the BBC Microcomputer. These allow the programmer to draw pictures, make sounds and make calls to the special routines of the MOS (Microcomputer Operating System). These sections only document the micro-PROLOG interface to these facilities.

All the standard primitives are as described in chapter 7.

Calling the MOS

1. `**`

`** <command>`

{give the command to the command line interpreter}

The `**` relation allows a command to be sent to the Command Line Interpreter (CLI). Command Line Interpreter commands can be used to select a filing system, give a command to the current filing system or select another programming language. The *command* argument must be a micro-PROLOG constant, and hence must be typed with " quotes if the command does not satisfy the conditions for being a single token. Here are some example uses of the `**` predicate:

`** BASIC`	{causes micro-PROLOG to be exited and BASIC to be entered}
`** "DELETE FL"`	{causes the disc file FL to be deleted (in the Disc Operating System version 1.00)}

Warning
Some filing system commands accessible via the Command Line Interpreter cause parts of the computer memory to be overwritten. These can cause your micro-PROLOG programs to be lost and the micro-PROLOG system to crash. See your filing system documentation for details. If in doubt, SAVE your programs before using the `**` relation.

2. *OSBYTE*

`(OSBYTE <number1> <number2> <number3> x y)`

{invoke the appropriate OSBYTE function}

The *OSBYTE* relation enables the operating system *OSBYTE* facility to be called. The three numerical arguments are sent to *OSBYTE* as the values of the machine registers A, X and Y, respectively. They must be in the range −256 to 255, or an *Arithmetic overflow* error will result. The numbers are rounded to integers, before being converted to 8-bit quantities. The final two arguments must be unbound; these are instantiated to the values in the X and Y registers when the *OSBYTE* call returns. They will be integers in the range −128 to 127. Here are some example calls:

```
?((OSBYTE 0 0 0 X Y))      {causes an error, numbered 247, whose
                            message is the version number of the
                            MOS}
?((OSBYTE 130 0 0 X Y))    {puts the machine high order address into
                            X and Y (X low byte, Y high)}
```

3. *OSWORD*

```
(OSWORD <number>)  {invoke a MOS OSWORD function}
```

The *OSWORD* facility can be used for a variety of applications, for instance, to read the dot pattern of a character or generate sound. The single argument to the *OSWORD* relation represents the number of the function required (it is passed to the routine in the machine A register). The number must be in the same form as the first three arguments of *OSBYTE*. All *OSWORD* calls require the use of a parameter block for the storage of extra arguments and results, and the micro-PROLOG system provides a dedicated parameter block of length 128 bytes for all uses of the *OSWORD* relation. Numbers are stored in and retrieved from this block using the *OSWORDARG* predicate (see below). Here are some example uses:

```
(OSWORD 3)      {reads the value of the interval timer into the first
                 5 bytes of the parameter block}
(OSWORD 4)      {writes to the interval timer, using the first 5 bytes
                 of the parameter block}
```

4. *OSWORDARG*

```
(OSWORDARG <number> x)
```

{get a byte from or put a byte in the parameter block}

The *OSWORDARG* predicate is used to place a byte into a specified position of the dedicated parameter block used by the *OSWORD* predicate, or to retrieve the value in a specified position. The *number* argument must be a number in the range 0 to 127, the bytes being numbered starting from 0; the number is converted to an integer if necessary. If the number falls outside the range, an *Arithmetic overflow* error is caused. If the second argument, *x*, is

bound, then it should be a number between −256 and 255, and this number, converted to integer and then into a single byte, is stored in the numbered location. If *x* is unbound, then the number in the location specified is retrieved and *x* is bound to it; *x* will be an integer in the range −128 to 127.

```
?((OSWORDARG 3 28))   {places 28 in byte number 3 (the fourth byte)}
?((OSWORDARG 3 X))    {binds X to the contents of byte number 3}
```

Making sounds

1. *Envelope*

```
(ENVELOPE <number1> <number2> ... <number14>)
```

{change the quality of sounds made}

The *ENVELOPE* relation is given 14 numerical arguments and works identically to the *ENVELOPE* statement in BASIC. Micro-PROLOG checks that each number is in the range −256 to 255; the numbers are converted to integers and then to 8-bit quantities. Any number out of range causes an *Arithmetic overflow*. Note that the sensible ranges for some of these arguments are actually somewhat more restrictive. A control error results if too many arguments are given; if too few are provided, the function is performed with undefined values for the missing arguments.

2. *Sound*

```
(SOUND <number1> <number2> <number3> <number4>)
```

{make a sound}

As with ENVELOPE, this is identical to the BASIC facility. *number1* specifies the channel, *number2* the loudness, *number3* the pitch and *number4* the duration of the sound. To be meaningful, the numbers must be convertible to 16-bit quantities, and an overflow error is caused if any number is not between −32768 and 32767.

```
?((SOUND 1 -15 53 20))   {makes a sound of maximum loudness on
                          channel 1, lasting for 1 second, the note
                          being middle C}
```

Graphics

1. *Mode*

```
(MODE <number>)      {change screen display mode}
```

In order to make use of graphics, it is necessary to change the graphics mode, which is the function of the *MODE* relation. Unfortunately, different graphics modes take up different amounts of memory space for keeping information

about the screen, which leave different amounts of space left for programs. If you run micro-PROLOG in mode 7, you have the maximum possible space for your program; using mode 0 gives you very little space (unless you have a 6502 Second Processor, BBC Microcomputer Model B+ or some other SHADOW memory device). Once micro-PROLOG has started up, it will not allow you to change the mode in such a way as to reduce your program space. This means that in order to use mode 5, say, you must already be in mode 5 when you enter micro-PROLOG. To do this, you must set the mode to 5 within BASIC (get to BASIC by the ** command above), and then return to micro-PROLOG (by giving the "*PROLOG" command to BASIC). Now that you are running micro-PROLOG in mode 5, the *MODE* predicate will allow you to change the mode to 7, and then back to 5 at will. At no time will you be allowed more program space than you initially have in mode 5, however.

```
?((MODE 5))
```
{change to graphics mode 5. The goal will fail if mode 5 requires more screen memory than the mode when micro-PROLOG was entered, but will otherwise succeed. The amount of space available for programs will not change}

2. *VDU*

```
(VDU <code1> <code2> ... <coden>)
```

{invoke graphics function by sending control codes to the VDU}

On the BBC microcomputer, graphics functions are invoked by asking the MOS to send special control codes to the *VDU* (the Console screen). The functions available include moving the cursor, changing the colours used and changing the areas of the screen used by graphics and text. The micro-PROLOG *VDU* predicate enables these functions to be used. Note that the PLOT predicate (see below) also operates by sending control codes to the *VDU* and is more convenient for certain more complex graphics functions. The *VDU* predicate can be given any number of arguments; these represent the control codes to be sent. The number of codes sent depends on the particular function required; the first code sent normally identifies the function and is followed by any other information necessary. Some *VDU* functions require that numbers greater than 255 be communicated, even though a control code cannot be this large. In these circumstances, the numbers must be sent as two control codes. If a particular *VDU* function is expecting a number sent as two codes, then two codes must be sent, even if the number happens to be small. You can indicate to micro-PROLOG that a number is to be sent as two codes by enclosing that number in a list. This corresponds to putting a semi-colon after a number in the BASIC *VDU* statement. Here are some examples of uses of the *VDU* predicate:

```
?((VDU 17 0))                          {in graphics mode 0, changes the text foreground colour to be black}
?((VDU 24 (150) (300) (1100) (700))    {sets up a graphics area in the centre of the screen. Note that graphics coordinates must be sent as two codes each}
?((VDU 16))                            {clears the graphics area}
```

Miscellaneous

1. *INKEY*

```
(INKEY <number> x)    {see if a key is pressed within a certain time period}
```

When the *INKEY* program is called, any characters that have been typed but not yet processed are forgotten and the system waits to see if a new key is pressed within a specified time period. The first argument to the predicate specifies the time period in hundredths of a second. If nothing is typed within the time period, the goal simply fails. Otherwise the second argument, *x*, which must initially be unbound, is bound to the ASCII code of the first character typed, and the goal succeeds. The number provided for the time period must be in the range 0 to 32767.

```
?((INKEY 100 X))    {see if a character is typed in the next second. if so, bind X to its ASCII code}
```

2. *ADVAL*

```
(ADVAL <number> x)    {get a number from an analogue input}
```

The relation ADVAL is used to obtain inputs from analogue input devices, such as games paddles and joysticks. The number provided as the first argument selects the channel of the device (a number between 1 and 4 inclusive); the second argument is bound to a number representing what that device is signalling. Various special numbers in the first argument position can be used for other purposes. See the documentation on the BASIC ADVAL keyword for more details.

3. *TIME*

```
(TIME x)
```

{set or read the interval timer}

The BBC micro comes with a built-in clock that can be used to time intervals. The clock counts in one hundredth of a second intervals. The TIME programs allows one to set the clock and to see what time it is indicating at any point. If the argument is bound, it must be a number between 0 and 2!32, and the clock is set to the time indicated. If the argument is not bound it will be instantiated to the time that the clock is currently reading.

```
?((TIME 0))          {set the clock to 0}
?((TIME X))          {see what time it is now}
```

Appendix D

File naming conventions

The ACORN disc systems expect a file name of the form:

```
{:<drivenumber>.} <name>
```

where *name* is the only compulsory part and consists of up to 16 characters. Remember that if you wish to save a file on drive 2 with name fred say, you must enclose the name in quotes for micro-PROLOG. Otherwise it will not be treated as a single token e.g.

```
SAVE ":2.fred"
```

or

```
** "CAT 1"
```

Appendix E

Error messages

See chapter 3 for information on error handling.

If there is no error handler present most errors are reported by a message

```
Error: n
```

where *n* is the error number. The current evaluation is then aborted and you are returned to the supervisor.

The numbered errors

The errors, and their numbers, are:

1. `Arithmetic overflow` or `underflow`

   If a call to an arithmetic primitive results in a number which cannot be represented then an `arithmetic overflow` is signalled. This includes the case of division by zero.

2. `Clause error`

   There are no clauses defined for the call being executed. Some PROLOG systems merely fail the call if there are no clauses for its relation. In micro-PROLOG you can simulate that behaviour by having a special clause in the `"?ERROR?"` program to `FAIL` the `"?ERROR?"` call for error number 2.

3. `Invalid form of use`

   The built-in primitives in micro-PROLOG often require a minimum number of arguments to be given at the time of the call. If the arguments to a call to such a primitive are underspecified, this error occurs. The error is also signalled if the evaluation has reached the use of a meta-variable (see chapter 2) and the meta-variable is unbound or has a value of the wrong form. There is a strong possibility that the calls in some clause or query are incorrectly ordered, and that the call that would bind the unbound variable has not yet been evaluated.

4. `Protected relation error`

   This error is signalled when you try to add or delete a clause for a relation name which is either a primitive of micro-PROLOG, or is imported to the current module, or is the name of a currently opened file.

5. *File error*

   This error is signalled when an error arises during a file operation. For example, if you try to open a file using a file name which is also the name of a program relation you will get this error. Other examples include *CREATE*ing a file already open, trying to *SEEK* to one of the special serial files like *CON:*, or trying to *SEEK* to an unopened file.

11. *Break!*

    This error is signalled when the user hits ESCAPE.

12. *Module error*

    This error is signalled whenever there is an illegal use of modules. It occurs when you try to *CRMOD* or *LOAD* a module other than at the root module level, or if the new module has a relation name in its export list and the relation is already defined by some program.

In addition to these, the BBC Microcomputer MOS and filing systems can also generate numbered errors during the execution of built-in programs. See the appropriate documentation for an explanation of the numbers used. These errors can be trapped in the same way as ordinary micro-PROLOG errors.

Errors numbered 0

There are some errors from which it is not possible to recover. When these errors occur, a message is displayed and and there is a default effect that cannot be changed. The error messages and the effects are:

Dict error	There is no space left in the dictionary for new constants. The evaluation is *ABORT*ed and you are returned to the supervisor.
Out of space	Garbage collection is not able to free enough space for the current evaluation to continue. You are *ABORT*ed to the supervisor.
Syntax error	Occurs when a term is being read in. It is displayed if there are too many right brackets or there is more than one term following a `.`. In either case the read continues with the extra brackets or terms ignored.
System abort	This arises when micro-PROLOG detects an internal inconsistency within the system. It means that a vital internal data structure has been destroyed and micro-PROLOG cannot precede. You must restart micro-PROLOG by pressing **BREAK**

You should actually never get a *system abort*, though there are situations where programmer action can cause one. The simplest case where a programmer can cause a system abort is in trying to execute the following program:

```
((abort-micro (x))
      (abort-micro x))
```

(This causes overflow in the garbage collector and cannot reliably be recovered from.)

Another situation which can cause system aborts is in certain uses of *DELCL* or *KILL* inside programs. If you are not careful you may delete a clause while the clause is still being used during the current query or command evaluation. If the garbage collector was called after the deletion of the clause and before micro-PROLOG's attempt to continue the evaluation by using the clause, you will get a system abort.

Appendix F

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 as follows:

Organisation

The evaluation area in micro-PROLOG is organised as a stack and a heap. The stack contains the activation records and the locations for the variables of the evaluation. The stack grows with recursion and pops normally only on backtracking. It records the state of the evaluation of the current supervisor command. The heap contains the *value* of variables. It also contains the program 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 `No space left.`

Because the garbage collector does not 'compact' the heap you may still get the `No space left` message even though the `SPACE` primitive indicates there is memory available. In this case, the only way to proceed is to `KILL` some of your definitions or modules, force a garbage collection by calling `SPACE`, and then re-load the `KILL`ed programs. Hopefully, this will have the effect of

shifting the free memory to the top of the heap so that you will have enough evaluation space.

As memory gets tight the garbage collector gets called more and more often; this can have a dramatic effect on the performance of the system. Normally garbage collection takes a very short time (about 0.25 seconds) and is not a big overhead.

Success popping

Micro-PROLOG performs special actions when a procedure call has been deterministic – when it has been solved using the last clause for its relation and the evaluation of each of the calls in the body of the clause has been deterministic. When such a deterministic call is solved, the activation record for the clause that solved the call is popped off the stack (it will always be at the top of the stack) in exactly the same way that the record of a procedure invocation is popped in a conventional recursive programming language. The activation record can be discarded because it will not be needed to determine what next clause to try in order to solve the call should there be backtracking to the call.

The alternative situation, where either the evaluation of one of the calls in the body of the clause was not deterministic or it is not the last clause, means that the activation record is left on the stack.

The expert programmer can give control information that allows a success popping even when the evaluation of the call appears to have been non-deterministic. This is when there are still untried clauses for the call or untried clauses for one or more calls solved during the evaluation of the call. He can do this by inserting the / backtracking control primitive in the clause.

Tail recursion

This is the name given to the form of recursion which is actually equivalent to a loop. micro-PROLOG can detect this special case of recursion, and when a tail recursive call is also deterministic micro-PROLOG does not grow the stack when entering the call.

A classical example of the power of tail recursion in saving space is during a list append. If we write the append program as:

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

then for all uses of the program the stack grows by only two records *for any length of input list.* The recursive definition of append is executed as though it were written as a WHILE loop.

The general conditions for a tail recursion optimisation are as follows. Consider a clause of the form:

```
((R t1..tk) (A1)...(An))
```

in which `(A1) .. (An)` are atoms in the body of the clause. Suppose that it is invoked by some calls `(R t'1..t'k)`. If the evaluation of the calls `(A1)..(An-1)` is deterministic, success popping of their evaluation traces will leave the activation record corresponding to the use of the clause at the top of the stack.

Further suppose that there are no other clauses to try for the call `(R...)` that invoked the clause, in other words, the above is either the last clause for *R* or a / was executed in the body of the clause. In this situation, the activation record for the clause is not needed for backtracking on the call. (If the last but one atom in the clause is / these first two conditions will be satisfied `(An-1)`.

Finally, let us suppose that `(An)` is now matched with the last clause for its relation, so the activation record of this new clause does not need to be left on the stack for possible backtracking on `(An)`. In this circumstance, the activation record is thrown away and replaced by the activation record of the clause invoked by the last call `(An)`.

Note that `(An)` need not be a call of the relation *R*. So the tail recursion optimisation of micro-PROLOG is a generalisation of the tail recursion that corresponds to iteration.

Chapter 7 of [Clark & McCabe 1984] has a more tutorial description of tail recursion.

Non-structure sharing

Micro-PROLOG is a so-called 'non-structure sharing' implementation. Briefly this means that when a variable is bound during a 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 stack and heap 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 variable takes occupies less space than in the more normal 'structure-sharing' implementation of PROLOG.

To take full advantage of the success popping and tail recursion space-saving optimisations 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 that for append) before the general case, and

using the *IF* condition and / where they are applicable (see chapter 7). Programs optimised for space in this way tend to be less optimal with respect to speed of execution, and vice versa.

References

Briggs J., [1984] *micro-PROLOG rules!* Logic Programming Associates Ltd., London.

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., McCabe F., [1984] *micro-PROLOG: Programming in Logic* Prentice-Hall International, Hemel Hempstead, England.

Clocksin W.F., Mellish C.S.,[1981] *Programming in Prolog* Springer-Verlag, New York.

Conlon T., [1985] *Start Problem Solving with PROLOG* Addison-Wesley, Wokingham, England.

van Emden M.H., Kowalski R.A., [1976] *The Semantics of Predicate Logic as a Programming Language* J. ACM, Vol 23, No 4, pp. 733-742.

Ennals R., [1984] *Beginning micro-PROLOG* Ellis-Horwood, Chichester, England and Harper and Row, New York.

Kanoui H., Van Canaghem M., [1980] *Implementing a very high level language on a very low cost computer* Groupe d'Intelligence Artificielle, Université 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.

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

Pereira L, Pereira F & Warren D. [1978] *User's guide to DEC system 10 PROLOG* Dept AI University of Edinburgh.

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 University Press.

Roussell P., *PROLOG: Manuel de référence et d'utilisation* Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Luminy, Sept. 1975.

de Saram H., [1985] *Programming in micro-PROLOG* Ellis-Horwood, Chichester, England.

Szeredi, [1982] *M-PROLOG* Szki, Institute for co-ordination of computer techniques, Budapest, Hungary.

Warren *et al.*, [1978] *DEC-10 PROLOG* Edinburgh University Press.

Micro PROLOG

SBD 32