Logic Programming Associates Ltd
36 GORST ROAD
LONDON SW11 6JE

A micro-PROLOG PRIMER

Clark Ennals
McCabe

Logic Programming Associates Ltd.

RÖHLING

A Micro-PROLOG Primer

ERRATA

Page	Line	Text	Change text to
26	9	(-77 38)	(38 77)
26	10	(-76 45)	(45 76)
49	18	and y=z1	and z1=y
50	25	(x.x)	(x:X)
50	25	x has-length y	X has-length y
51	9	SUM(y1 1 y)	SUM(y1 1 z1)
51	-9	y length-of X	z length-of Y
52	12	y length-of X	z length-of X
52	-13	4 length-of y	4 length-of x
53	14	parameters	parents
53	14	correct	connect
68	-13	z length-of y	x length-of y
68	-12	"z length-of y"	"x length-of y"
99	-19	(reduce SUM y 0 x)	(reduce SUM y x)
99	-12	(reduce y 0 x)	(reduce SUM y x)
126	12	(Y1 X1)	(y1 X1)

A positive line number is a line of text count from the top of the text, a negative number from the bottom of the text.

2041I

A micro-PROLOG Primer

K.L.Clark J.R.Ennals
F.G.McCabe

First Edition December 1981,
Second Edition April 1982

Published by:-

Logic Programming Associates Ltd., 36 Gorst Rd., LONDON SW11 6JE, ENGLAND

CONTENTS

Preface

This Primer is intended to serve as an introduction for the non-specialist to the micro-PROLOG system, which is implemented for microcomputers based on the Z80 microprocessor and the CP/M Operating System. The primer is a companion volume of the micro-PROLOG Programmer's Reference manual [McCabe 1981], which gives a complete description of the system but assumes knowledge of PROLOG programming as covered by this primer.

Since micro-PROLOG is one of the PROLOG family of logic programming languages (PROgramming in LOGic), the primer also serves as an introduction to the general concepts of logic programming. The differences between micro-PROLOG and the other PROLOGs are mainly syntactic.

Why program in logic?

Ever since von Neumann first described the form of the modern computer they have been programmed in essentially the same way. The first programming language was the binary language of the machine itself: machine code; then came assembler, which is symbolic machine code; then the so-called high level languages like FORTRAN, COBOL, BASIC, followed by today's more modern variants ADA and Pascal. All of these programming languages share a common characteristic: the programmer must describe quite precisely <u>how</u> a result is to be computed, rather than <u>what</u> it is that must be computed.

A computer program in one of these programming languages consists of a script of instructions each of which describes an action to be performed by the computer. For example, the <u>meaning</u> of the BASIC statement:

```
10 LET X = 105
```

is that the memory location whose name is X should have its contents changed to 105. They are <u>imperative</u> programming languages, statements in them are commands which specify actions to be performed. They are geared to the description of the <u>behaviour</u> needed to achieve the desired result. While undoubtedly we sometimes <u>think</u> behaviourally, most often we do not. For example, the first question we ask someone about a particular computer or program is:

"What does it do?"

not:

"How does it do it?"

Certainly the answer to the first question will not be:

```
1 INPUT X,Y
2 IF X>Y THEN 4
3 Z=X: X=Y: Y=Z
4 X=X-Y
5 IF X>0 THEN 2
6 PRINT Y
7 END
```

We shall not simply list the program. What we are more likely to do is to describe the <u>relation</u> between the input and output of the program. We might say, for example, "it prints the greatest common divisor of the two numbers read-in".

Similarly the most effective way to tackle a new programming task is to first develop a specification of "what the program has to do". This specification is also often a description of the relation of the output of the program to the input. Having described this relation, the program is then written as a sequence of actions which 'compute' the output that meets the specified relation to the input.

Given that people find difficulty in thinking purely in imperative ways (as is evidenced by the huge shortage of programmers) it seems archaic to program computers in this way. Computers are supposed to help solve problems, not to create more.

The net effect of forty years of development of programming languages seems to be that there are very few programmers, and that very few of these programmers have any solid confidence that their programs are correct. Programming is still essentially a craft activity. Compare that with almost any other modern production/design activity which is typically highly automated, with sophisticated (computer) aids for designing and manufacturing products.

One way of tackling the programming problem is to provide a programming language which is descriptive rather than prescriptive: a language in which programs are descriptions of the input/output relation to be satisfied. The execution of the program is then a use of this description to find an output that satisfies the relation. The way in which the description is used is the secondary, control aspect of the program. By taking into account the way the description is used we might choose one description rather than another. This is the pragmatics of programming in a descriptive language. But it will still remain the case that the program is primarily a description of what it is supposed to compute, rather than a prescription of how it should compute it.

LISP (at least pure LISP) is an early example of a descriptive language; PROLOG is another. A PROLOG program is essentially a set of sentences of symbolic logic that define the relation that we want to compute. PROLOG computation is the use of this definition to find an output that lies in the defined relation to the input. We shall see that it is often the case that a single description of some input/output relation can be used in the inverse mode. It can be used to find all the inputs that will give rise to a particular output! This invertability of use is only possibly because the program is descriptive. It is not limited to one use because it does not comprise a sequence of instructions that encode the behaviour of that use.

Finally, since a PROLOG program is a description of a set of relations, it blurs the distinction between data retrieval and computation. In PROLOG, they are both the finding of one or more arguments of a relation using the description of the relation provided by the program.

Chapter descriptions

Chapter 1 introduces micro-PROLOG by using it to develop and query a simple data base of facts. The ease with which one can construct and query such a data base is one of the prime features of the language. The chapter also introduces the arithmetic facilities of micro-PROLOG. These are quite different from those of a conventional programming language. We add and subtract by querying an (implicit) data base of facts about the addition relation, likewise we multiply and divide by querying a data base of 'times tables'.

Chapter 2 describes how the data base can be augmented by rules. Rules can be used to abbreviate queries. They can also be used to give a recursive definition of a relation.

In Chapter 3 we introduce lists and describe how they can be used to structure information, often compressing many statements into one. The elements of a list are accessed using special list patterns. This pattern processing of list structures is another unique feature of PROLOG. The chapter also introduces a primitive of the language that can be used to wrap up the set of answers to a database query as a list. This provides the interface between the use of PROLOG as a database language and its use as a list manipulation language.

In Chapter 4 we describe more complex forms of query. These include the use of not and for-all.

In Chapter 5 we discuss programs which use more complex list processing. These include the "appends-to" program, a list sorting and a simple parsing program.

In Chapter 6 we introduce the imperatives of micro-PROLOG. These are built in relations that have a side-effect when they are evaluated. An example is the built in relation that reads data from the terminal. Declaratively it means: something that can be read at the terminal. Procedurally, it always returns the next thing to be typed. The imperatives of micro-PROLOG detract somewhat from its descriptive nature, a program that uses them is not a purely descriptive program. However, as we shall see, the use of the imperatives can often be restricted to the definition of one or two auxiliary relations, the rest of the program being entirely descriptive.

In Chapter 7 we describe the internal syntax of a micro-PROLOG program. This is the form in which the facts and rules are accessed and evaluated by the micro-PROLOG interpreter. The user friendly surface syntax, the syntax used previously, is translated into the internal form by a special micro-PROLOG program called simple. The simple program is written in internal syntax. Any program can be written and entered in internal syntax form. (The micro-PROLOG reference manual uses the internal syntax.)

All micro-PROLOG programs are really just list structures. As in LISP, one can therefore write micro-PROLOG programs that manipulate lists that are other micro-PROLOG programs. The translator program, simple, is such a program. This ability to treat programs as data is an exceeding powerful tool. It enables one to write programs in micro-PROLOG to modify and extend the micro-PROLOG system. In Chapter 7 we show how this can be done and we introduce one or two features of micro-PROLOG that can only be used by programs written in internal syntax.

Applications of PROLOG

The current major uses of PROLOG are as a language for Artificial Intelligence research, as a language for implementing and querying data bases and in Education to teach both logic and the descriptive approach to programming. Within Artificial Intelligence it is being used for natural language understanding, problem solving and the implementation of expert systems.

Logic is particularly useful as a language for data bases where it has a number of advantages over the conventional data base systems. Logic can be used both to express data base queries, and to describe the data base itself. The result of this is that the data base implementor and user share a common language, enabling users to become programmers: common queries can be easily turned into an extension of the data base. Logic also plays a role in data bases in maintaining integrity. Integrity constraints can be expressed as special queries of the data base, which are tested whenever the data base is updated.

PROLOG is not particularly suited for applications which need a lot of routine numerical work, nor for some real time and some commercial data processing applications. However in these fields logic is still a suitable specification language, and PROLOG can be used to speedily implement and test a prototype program.

Acknowledgements

The research which underlies many of the ideas presented in this primer was supported by the U.K. Science & Engineering Research Council in a series of research grants held by R.A.Kowalski and K.L.Clark at Imperial College. Of particular relevance is the "Logic as a Computer Language for Children" project which is concerned with teaching the principles of logic programming to school children. This project uses micro-PROLOG and the user friendly surface syntax described in this primer was developed for the project.

We are also grateful to the groups of people in various parts of the country who have acted as hosts for demonstrations of micro-PROLOG, providing excellent opportunities for testing different methods of explanation to interested non-specialists.

Finally, the authors would like to thank Diane Reeve and Sandra Evans whose patient 'slaving over a hot word processor' made this primer possible.

1. Basic logic programming - facts and queries

1.1 Developing a simple data base program

In this chapter we introduce some of the basic ideas of logic programming by giving an example of the setting up and querying of a data base in micro-PROLOG. If the reader has access to a computer which has micro-PROLOG we recommend that he follows through the example using the computer. Instructions for the loading of the PROLOG system are given in Appendix A.

Adding facts

Let us suppose that we want to set up a data base describing the family relationships of the Tudor royal family. We will do this by making statements about these relationships, adding them one at a time to the data base.

The statements are expressed as sentences of symbolic logic. There are two kinds of sentences: simple and compound. To begin with we shall only need simple sentences which express basic facts.

In any family there are a number of basic facts about the relationships between individuals. Two such "Tudor" facts are:

Henry the 7th is the father of Henry the 8th (1)
Henry the 8th is the father of Mary (2)

There are many such facts, each of which describes an instance of one of the family relationships of the Tudors. Now these English sentences are almost sentences of micro-PROLOG! The simplest form of micro-PROLOG sentence has three components:

Name-of-Individual Name-of-relationship Name-of-Individual

In the two sentences (1) and (2) the Name-of-relationship is "is the father of". In micro-PROLOG we have to make this into one word by hyphenating, we must use: "is-the-father-of". Similarly, we must name individuals by a single word. Again we can do this by hyphenating, writing "Henry-the-7th", or by abbreviating, using "Henry7". Rewriting (1) and (2) in this way transforms them into simple sentences of micro-PROLOG.

```
Henry7 is-the-father-of Henry8
Henry8 is-the-father-of Mary
```

These two simple sentences in the data base are a direct representation of the two facts (1) and (2). We 'tell' the micro-PROLOG system about these facts by adding each to the data base. We type:

```
&. Add(Henry7 is-the-father-of Henry8)
&. Add(Henry8 is-the-father-of Mary)
```

Notice that the sentence to be added is surrounded by brackets. The "&." is not typed, it is the prompt printed out by micro-PROLOG to tell us it is ready to accept a new sentence. Moreover, each Add instruction must be terminated by a carriage return. Before typing the carriage return you can correct typing mistakes using the 'rubout' or 'backspace' key. After the carriage return any mistake in the form of the added sentence will produce a "?" response. If the "Add" is misspelt, you will get a "Clause Error" message. Both indicate that the sentence has not been accepted, so try again with a new Add command.

You do not have to type all of a sentence on a single line. It can be spread over several lines, but words cannot be split across lines. If you do type a sentence without finishing it, you will get the prompt

1

This merely indicates that micro-PROLOG is waiting for the right bracket that marks the end of the sentence to be added.

Different kinds of relationship

A relationship such as "is-the-father-of" holds between pairs of individuals, in this case between a 'father' and a 'child'. It is a binary relation. Not all relationships are between pairs, some relate three or more individuals, and some are properties that apply to single individuals. The genders "male" and "female" are properties. (More technically, they are unary relations.) The relation of someone giving something to someone else is a three place relation (a ternary relation). Simple sentences giving facts about these non-binary relations have a slightly different syntax. Instead of writing

```
individual-name  relation-name  individual-name
```

we write

```
relation-name(individual-name individual-name ... individual-name)
```

For example,

```
Male(Henry8)
Gives(Henry8 Mary book)
SUM(2 3 5)
```

We can also write the binary simple sentences in this way:

```
is-the-father-of(Henry7 Henry8)
```

but the original way of writing this is more readable. We shall use these 'non-Binary' simple sentences more often when we get to arithmetic in PROLOG.

A technical term - argument of a relation

A simple sentence tells us that certain individuals are related by some relation. In mathematics and logic the individuals are called the arguments of the relation. We also talk about the first argument, the second argument, etc., of the relation. This names the argument by its position in the list of arguments of the simple sentence. In the sentence

```
Gives(Henry8 Mary book)
```

"henry8" is the first argument, "Mary" the second and "book" the third.

A note on the use of spaces

The spaces between the names of the individuals are important. In micro-PROLOG spaces and new lines and tabs are separators. They are the only separators. The number of spaces you use does not matter, but failure to use a space may mean that micro-PROLOG makes into one name what you intended to have as two names. For more detailed information on what is or is not understood by micro-PROLOG as a word boundary, we refer the reader to the reference manual. If in doubt use a space. The converse of this is the need to hyphenate phrases such as "is the father of" in order to make it into one name, not 4.

1.1 Developing a simple data base program

Adding some more facts

Carrying on, let us enter more of the known facts concerning the family relationships of the Tudors:

```
&.Add(Elizabeth-of-York is-the-mother-of Henry8)
&.Add(Katherine is-the-mother-of Mary)
&.Add(Henry8 is-the-father-of Elizabeth)
&.Add(Ann is-the-mother-of Elizabeth)
&.Add(Henry8 is-the-father-of Edward)
&.Add(Jane is-the-mother-of Edward)
&.Add(Male(Henry7))
&.Add(Male(Henry8))
&.Add(Female(Elizabeth-of-York))
&.Add(Female(Katherine))
&.Add(Female(Mary))
&.Add(Female(Elizabeth))
&.Add(Female(Ann))
&.Add(Female(Jane))
&.Add(Male(Edward))
```

Notice that we slipped in some "is-the-mother-of" facts and some facts about who is male and female. We can add sentences of any relationship at any time using the "Add" command. The sentences are collected together by name of relationship. The vocabulary of a program consists of the names of the relationships and the names of the individuals; the vocabulary defines the "things" that a subsequent query can talk about. Our vocabulary so far is

```
Henry7
Henry8
Mary
Elizabeth-of-York
Katherine              Names of Individuals
Elizabeth
Ann
Edward
Jane

is-the-father-of
is-the-mother-of       Names of Relations
Male
Female
```

Listing and saving the program

We can display our program by using another command "List". This command displays on the screen all the sentences entered, or just those for specified relations. To list the full program we type:

```
&. List All
Henry7 is-the-father-of Henry8
Henry8 is-the-father-of Mary
Henry8 is-the-father-of Elizabeth
Henry8 is-the-father-of Edward
Elizabeth-of-York is-the-mother-of Henry8
Katherine is-the-mother-of Mary
Ann is-the-mother-of Elizabeth
Jane is-the-mother-of Edward
Male(Henry7)
Male(Henry8)
Male(Edward)
Female(Elizabeth-of-York)
Female(Katherine)
Female(Mary)
Female(Elizabeth)
Female(Ann)
Female(Jane)
&.
```

The sentences are listed according to name of relationship, not order of typing. However, the listing of the sentences for each relation does correspond to the order in which they were entered.

We can choose a particular name of relationship, and list that. For instance:

```
&.List is-the-mother-of
Elizabeth-of-York is-the-mother-of Henry8
Katherine is-the-mother-of Mary
Ann is-the-mother-of Elizabeth
Jane is-the-mother-of Edward
&.
```

By using the command:

```
&.List dict
```

a dictionary of the names of relationships used so far is listed, in this case we will get:

```
dict(is-the-father-of)
dict(is-the-mother-of)
dict(Male)
dict(Female)
&.
```

We can save the program on disk, giving it a unique name of our choice, as follows:

```
&. Save tudors
```

This copies all the sentences of the current program into a file named "TUDORS.LOG". (The name given in the Save command must be different from the name of any relation in the program.) The sentences still remain in the data base. However, on a subsequent occasion, we can retrieve these sentences and have them automatically added to any data base simply by typing:

```
&. Load tudors
```

<u>Simple editing</u>

Simple editing of the PROLOG program is performed by deleting a whole sentence and adding a new one. Let us suppose that the name of Elizabeth's mother has been misspelt, and that it should be "Anne". The simplest way to remove the sentence "Ann is-the-mother-of Elizabeth" is to use:

```
&. Delete(Ann is-the-mother-of Elizabeth)
```

This use of Delete is the opposite of Add. If the sentence given as the argument to the command is in the program, the Delete command removes it. If it is not in the program, you will get a "?" response. You will get this response if there is not an <u>exact</u> match between the sentence to be deleted and some sentence of the current data base.

There is another way to delete a sentence, we can refer to it by its position in the listing of the sentences for its relation. In the listing the relation "is-the-mother-of" given above the sentence "Ann is-the-mother-of Elizabeth" was the third sentence to be listed. So, instead of giving the sentence to delete we can use

```
&. Delete is-the-mother-of 3
```

Having deleted the sentence, using either form of the Delete command, we can add the new version:

```
&.Add(Anne is-the-mother-of Elizabeth)
```

If we now list the "is-the-mother-of" relation we will get:

```
&.List is-the-mother-of
Elizabeth-of-York is-the-mother-of Henry8
Katherine is-the-mother-of Mary
Jane is-the-mother-of Edward
Anne is-the-mother-of Elizabeth
&.
```

The new sentence

```
Anne is-the-mother-of Elizabeth
```

is now listed at the end of the relation because it was entered last.

Let us now correct the spelling of "Ann" in the "Female" relation. This time we will <u>replace</u> the sentence Female(Ann) with Female(Anne). We do this by deleting the old sentence and adding the new one so that it occupies the same position in the listing of "Female" sentences. The following are the commands (those preceded by "&.") and the PROLOG responses.

```
&.List Female
Female(Elizabeth-of-York)
Female(Katherine)
Female(Mary)
Female(Elizabeth)
Female(Ann)
Female(Jane)
&.Delete Female 5
```

```
&.Add 4 (Female(Anne))
&.List Female
Female(Elizabeth-of-York)
Female(Katherine)
Female(Mary)
Female(Elizabeth)
Female(Anne)
Female(Jane)
&.
```

We have used a variant of the Add command in which the position after which the sentence should be added is given. Add 4 (Female(Anne)) puts it after the fourth sentence about the Female relation, which is where the deleted sentence was. For a more sophisticated way of editing programs see the "Edit" command in Appendix B.

<u>Summary of program development commands</u>

<u>Add</u>

(i) Add (sentence)

will add the 'sentence' argument to the end of the list of sentences for its relation.

(ii) Add n (sentence)

will add 'sentence' after the n'th sentence in the list of sentences for its relation. If n = 0, the new sentence will be placed in front of these sentences.

<u>Delete</u>

(i) Delete (sentence)

will remove 'sentence' from the data base.

(ii) Delete relation n

will remove the n'th sentence in the current list of sentences for 'relation'.

<u>List</u>

(i) List relation

lists all the sentences for relation.

(ii) List All

lists all the sentences in the current program.

<u>Save</u>

Save name

will save all the sentences of the current state of the program in a file "name.log". "name" should be different from any relation of program.

<u>Kill</u>

Kill relation

deletes <u>all</u> sentences for 'relation'.

<u>Quit</u>

QT.

this command exits from PROLOG to CP/M. In general you should save your program before using it.

<u>Exercise</u> 1-1

If you are following the text with a computer, at this stage you should save the program that has been developed, using the command:


```
&.Save tudors
```

This and following exercises can be carried out with or without a computer.

1. Using the program developed above

   a. Show how you would edit the program to change the spelling of "Katherine" to "Catherine" in each sentence in which it appears. Do this in such a way that the new sentences are in the same positions in the program as those they replace.

   b. Add the two simple sentences necessary to express the information that Henry7 had a son called Arthur. Add these new sentences so that they will be listed at the beginning of the sentences for their relation. [Hint: if you give the sentence number 0 in the Add command it will add after the 0'th sentence and so place the new sentence at the beginning.]

2. Set up a database of simple sentences describing countries in different continents using the following vocabulary:

Names of Individuals

Washington-DC	USA	North-America
Ottawa	Canada	Europe
London	United-Kingdom	Africa
Paris	Italy	
Rome	Nigeria	
Lagos		

Names of Relations

capital-of
country-in

As examples, your data base should contain the sentences:

```
Washington-DC capital-of USA
USA country-in North-America
```

Save this data for future use using the Save command.

3. Set up a data base of simple sentences describing the books of different kinds written by different people, using the following vocabulary:

Names of Individuals

Tom-Sawyer	Mark-Twain
For-Whom-The-Bell-Tolls	Ernest-Hemmingway
Oliver-Twist	Arther-Miller
Great-Expectations	Charles-Dickens
Macbeth	William-Shakespeare
Romeo-And-Juliet	
Death-Of-A-Salesman	Novel
	Play

Names of Relations

type

written-by
writer

For example, you should have the sentences

Tom-Sawyer written-by Mark-Twain
Tom-Sawyer type Novel
writer(Mark-Twain)

in your data base. Save this data for future use with the Save command.

1.2 Queries

We now look at how a PROLOG program is queried. This is done via one of the question commands of PROLOG. The questions presented in the example are based on the Tudor family relationships data base that we developed in 1.1. If this data base is not in the computer (test this by trying to list the sentences for the "is-the-father-of" relation) load it with a Load tudors command.

Confirmation

The simplest form of query is the "Does" query which asks for confirmation of some fact. We explain this and other queries by posing some example questions in 'logicised' English. Below the questions we give the PROLOG equivalent and the answers given by the computer. A brief explanation is provided of points arising from the query.

English: Is it the case that Henry8 is the father of Elizabeth?
PROLOG: &. Does(Henry8 is-the-father-of Elizabeth)
YES

The query is asking about a particular instance of the "is-the-father-of" relation. As there is a match between the query sentence and the sentence

Henry8 is-the-father-of Elizabeth

in the data base, the answer is "YES", an abbreviation for "Yes, fact is confirmed".

English: Is it the case that Katherine is the mother of Edward?
PROLOG: &. Does(Katherine is-the-mother-of Edward)
NO

In this case there was no match between the query sentence and a sentence in the program, so the answer is "NO", short for "No, fact is not confirmed".

English: Do you know who the mother of Mary is?
PROLOG: &. Does(x is-the-mother-of Mary)
YES

In this query we are trying to find out whether the data base contains a sentence that records who the mother of Mary is. The "x" stands for the mother, whose name is unknown to us. PROLOG searches the sentences of the "is-the-mother-of" relation, looking for a simple sentence of the form

x is-the-mother-of Mary.

It finds the simple sentence

Katherine is-the-mother-of Mary

and so returns the answer "YES". It does not tell us that the unknown x is Katherine. To retrieve this information we use a different form of query.

Variables in queries

The letters x, y, z, lower or upper case, followed by one or more decimal digits, e.g. x1, y31, are the variables of micro-PROLOG. The variable in a query is a very simple concept: it stands for some unknown individual. It is a place holder, ready to be filled in by a name. Variables are the formal equivalent of pronouns in English. Where in English we would say something, someone, it or he, in PROLOG we use a variable. Just as pronouns are never used in English as proper names, so in PROLOG variables can never be used as proper names. You cannot enter a fact about an individual whose name is X! The variable names were chosen so that this problem is highly unlikely to arise.

Data Retrieval

To retrieve the names of unknown individuals we use the "Which" form of query.

English: Who is the x such that x is the father of Edward?
PROLOG: &. Which(x x is-the-father-of Edward)
Answer is Henry8
No (more) answers

A "Which" query has two arguments. The second argument is a query pattern, a sentence which contains variables. Here it is the pattern

x is-the-father-of Edward

The first argument is the answer pattern. Here it is the single variable x of the query pattern. More generally, the answer pattern is a list of variables that appear in the query pattern.

In answering the query micro-PROLOG finds all the instances of the query pattern that are facts that can be confirmed. In doing this it 'fills in' the variable slots of the query with the names of individuals, which are then printed in accordance with the answer pattern. In this case, there is only one instance of

x is-the-father-of Edward

that can be confirmed. This is the instance with x = Henry8. It is confirmed because

Henry8 is-the-father-of Edward

is a sentence of the data base. So we get printed out

Answer is Henry8

followed by the message that there are no more answers.

Compound queries

Queries with several component simple sentences can be expressed directly in both "Does" and "Which" form.

```
English:  Was Henry7 the father of Henry8 and of Edward?
PROLOG:   &. Does(Henry7 is-the-father-of Henry8 and
             Henry7 is-the-father-of Edward)
          NO
```

For a compound question prefaced by "Does" to receive the answer YES all of the simple sentences must receive the answer YES. Otherwise the answer NO is returned. In this case the second sentence is not contained in the data base, hence the answer to the combined query is "NO".

Notice how in PROLOG we must make explicit the question "was Henry7 the father of Edward" that is implicit in the English phrase "and of Edward".

```
English:  Who had Henry7 as a father, and was the father of Elizabeth?
PROLOG:   &. Which(x Henry7 is-the-father-of x and
                  x is-the-father-of Elizabeth)
          Answer is Henry8
          No (more) answers
```

```
English:  Who are the daughters of Henry8?
PROLOG:   &. Which(x Henry8 is-the-father-of x & Female(x))
          Answer is Mary
          Answer is Elizabeth
          No (more) answers
```

Notice that in this query we have used "&" as an abbreviation for "and". This is an abbreviation that PROLOG understands.

```
English:  Who is a mother (of somebody)?
PROLOG:   &. Which(x x is-the-mother-of y)
          Answer is Elizabeth-of-York
          Answer is Katherine
          Answer is Jane
          Answer is Anne
          No (more) answers
```

```
English:  Tell me all the father, son pairs that you know about?
PROLOG:   Which((x y) x is-the-father-of y & Male(y))
          Answer is (Henry7 Henry8)
          Answer is (Henry8 Edward)
          No (more) answers
```

In this query the answer pattern is the list "(x y)" of both variables appearing in the query pattern. They are the unknown father and unknown son referred to in the query pattern. Note that we must use the vocabulary of the program. The program does not include any facts that directly describe the father-son relationship, so we describe what we want using "is-the-father-of" and "Male".

Summary of querying commands

Does

```
Does(simple-condition [and ... simple-condition)
```

This query checks to see if the given (possibly compound) condition can be confirmed using the facts in the data base. It responds "YES" if it can, and "NO" if it cannot confirm the query.

Which

```
Which(P simple-condition [and ... simple-condition)
```

This query returns the answers to the query defined by the simple condition(s). Each answer is in the form: "Answer is P' " where P' is the answer pattern P with the variables replaced by the names that satisfy the condition. After all the answers have been found then the message "No (more) answers" is displayed at the console.

One

```
One(P simple-condition [and ... simple-condition)
```

The One query is similar to the "Which" query except that after each of the solutions is found the system prompts for input. If you respond with "C" then the next solution is found, with any other response the evaluation stops.

For example, we might ask the system for just one child of Henry8:

```
&.One(x Henry8 is-a-parent-of x)
Answer is Mary.C
Answer is Elizabeth.F
&.
```

Exercise 1-2

1. Using the Tudor royal family data base developed in this chapter, give the appropriate answers to the following PROLOG queries:
   a. Does(Jane is-the-mother-of Elizabeth)
   b. Does(Henry7 is-the-father-of x)
   c. Which(x Henry7 is-the-father-of x)
   d. Does(x is-the-mother-of Mary and Female(x))
   e. Which(x Henry8 is-the-father-of x and Male(x))
   f. Which((x y)x is-the-father-of z and z is-the-father-of y)

2. Using the vocabulary of the Tudor royal family data base, express these English questions as PROLOG queries:
   a. Was Katherine the mother of Edward?
   b. Who is a father?
   c. Was Jane the mother of anybody whose father was Henry7?
   d. Who had Henry8 as a father and Katherine as a mother?

3. Using the geographical data base started in Exercise 1, express these English questions as PROLOG queries:
   a. Is Rome the capital of France?
   b. Is Washington-DC the capital of a country in Europe?
   c. What are the capitals of countries in Europe?
   d. Is the capital of Italy known?
   e. For which North-American countries is the capital known?
   f. For which continents are the capitals of countries known?

4. Using the books data base started in Exercise 1-1, give the appropriate answers to the following PROLOG queries:

   a. Does(Oliver-Twist written-by William-Shakespeare)
   b. Does(x written-by Mark-Twain and x type Novel)

c. Which((x y) x type Play and x written-by y)
d. Which(x x type Novel and x written-by Charles-Dickens)
e. Which(x y written-by x)

1.3 Arithmetic

As we have remarked, PROLOG is not suited for applications which need a lot of routine numerical work. However, we can do simple integer arithmetic using the three primitive relations SUM, PROD and LESS.

We use these relations in exactly the same way as we use relations described by sentences of the data base. Although each relation is implemented in machine code, so as to make use of the hardware operations of the machine, we can think of them as being defined by an implicit data base of simple sentences.

SUM Relation

The SUM relation is a three argument relation such that

SUM(x y z) holds if and only if z = x + y.

The implicit data base describing the relation contains sentences such as SUM(2 3 5) and SUM(-3 10 7). We do addition & subtraction by querying the implicit data base.

Uses of the SUM relation

Checking:

```
&. Does (SUM(20 30 50))
   YES
```

Adding:

```
&. Which(x SUM(30 -2 x))
   Answer is 28
   No (more) answers
```

Subtracting:

```
&. Which(x SUM(x 3 15))
   Answer is 12
   No (more) answers
```

or:

```
&. Which(x SUM(3 x 15)
   Answer is 12
   No (more) answers
```

Restrictions on SUM queries

A query pattern for the SUM relation can have at most one unknown argument. This constraint would not apply if there was a real data base for the relation. It applies because the micro-PROLOG system simulates the data base and for efficiency supports only a restricted range of query patterns. This means that a query such as

```
&. Which((x y) SUM(x y 10))
```

will not be answered. It will result in a "Control Error" message. Try

it!

PROD relation

The PROD relation is such that

PROD (x y z) holds if z = x * y

Uses of the PROD relation

Checking:

```
&. Does (PROD(3 4 12))
   YES
```

Checking if one number divides another:

```
&. Does(PROD(3 y 9))
   YES
&. Does(PROD(3 y 10))
   NO
```

Multiplying:

```
&. Which(x PROD (5 4 x))
   Answer is 20
   No (more) answers
```

Exact division:

```
&. Which(x PROD(x 2 10))
   Answer is 5
   No (more) answers
```

We must be careful with the use for division. If there is no exact division we get no answer.

```
&. Which(x PROD(x 3 17))
   No (more) answers
```

For such a division we need to use a special four argument form of the PROD relation. This is the safe form to use for every division. The extra argument of the relation represents the remainder on division.

Inexact division:

```
&. Which((x y) PROD(3 x 17 y))
   Answer is (5 2)
   No (more) answers
```

Restriction on PROD queries

The restrictions on the use of the three argument form of PROD are the same as those for SUM. At most one argument can be a variable, but this can be any of the three arguments. This covers the use for multiplication and exact division. The four argument form can only be used for division. Thus the last argument, the remainder argument, must always be a variable and the second to last argument, the number to be divided, must be an integer. The divisor can be given as either the first or second argument, but then the other argument must be a variable representing the unknown quotient. So the above division query could have been given as:

&. Which((x y) PROD(x 3 17 y))
Answer is (5 2)
No (more) answers.

The uses PROD(x M N y), PROD(M x N y), M and N integers, will both return x and y values such that

x * M + y = N, y<M

LESS relation

The primitive LESS relation can only be used for checking. LESS(x y) holds if x is less than y in the usual ordering of the integers.

Uses of LESS

&. Does(3 LESS 4)
YES

&. Does(4 LESS 3)
NO

LESS can also be used for comparing two words. The ordering used is that of the dictionary. LESS(x y) holds for words x and y if x comes before y in a dictionary. Example:

&. Does(FRED LESS FREDDY)
YES

&. Does(ALBERT LESS HAROLD)
YES

&. Does(SAM LESS BILL)
NO

Exercise 1-3

1. Answer the following PROLOG queries:
   a. Does(SUM(9 6 15))
   b. Which(x SUM(4 18 x))
   c. Which(x SUM(x 23 40))
   d. Does(9 LESS 10)
   e. Does(SUM(9 8 x) and x LESS 19)
   f. Which(x PROD(9 7 x))
   g. Does(PROD(11 8 80))
   h. Which((x y) PROD(4 x 14 y)

2. Write PROLOG queries to ask the following English questions
   a. What is 9 plus 7?
   b. What is the remainder when 65 is divided by 7?
   c. What is the result if you add 29 and 53, and divide the total by 2?
   d. Can 93 be exactly divided by 5?
   e. Is the result of multiplying 17 and 3 less than 50?

1.4 Evaluation of queries

This is an appropriate point to say something about the way in which PROLOG evaluates queries.

When querying a data base of simple sentences we can, for the most part, ignore the way that queries are evaluated. However, we shall see that the ordering of the conditions in a compound query can effect the time that PROLOG takes to answer the query. Choosing an ordering that facilities the evaluation is part of the pragmatics of using PROLOG. Moreover, for certain compound queries, for example the query:

Which(x PROD(37 51 y) & SUM(y 73 x))

we must know about the order of evaluation of the component conditions. Does PROLOG answer the SUM or the PROD query first? If it is the SUM query we will get an error message because there are two unknown arguments y and x. If PROLOG answers the PROD query first there will be no problem providing the answer obtained for the unknown y is 'passed on' to the SUM query before it is answered. Fortunately this is exactly what PROLOG does.

Evaluation of simple "Does" queries

The simplest form of query is the "Does" query of the form

Does(S)   where S is a simple sentence

PROLOG evaluates this query by searching through the sentences in the data base for the relation of the sentence S. It does not search the whole data base. PROLOG stores the sentences about each relation in a list, the ordering of the sentences on the list being the order in which they are displayed by the List command. PROLOG runs down this list, comparing S with each sentence in turn. If it finds an exact match between S and a sentence in this list it abandons the search and gives the answer "YES". If it reaches the end of the list of sentences without finding a match, it displays the "NO" answer.

Example 1

Does(Male(Henry8))

The sentences in the Tudors data base about Male are stored in the order

Male(Henry7)
Male(Henry8)
Male(Edward)

because this is the order in which they are listed by the "List Male" command. First PROLOG compares the query sentence

Male(Henry8)

with the sentence

Male(Henry7)

that heads the list. The sentences do not match because "Henry8" and "Henry7" are different names. It then moves on to the next sentence. We now have an exact match, so PROLOG abandons the search and gives the answer "YES".

If we pose the query

Does(Male(Edward3))

PROLOG compares Male(Edward3) with each sentence in turn. In no case is there an exact match. So we get the answer "NO".

"Does" query with a sentence pattern

A "Does" query of the form

Does(S)   where S is a simple sentence pattern

is answered in much the same way. The only difference is that when looking for an exact match PROLOG is allowed to give each variable in S a value which is the name of some individual.

Example 2

Does(x is-the-father-of Elizabeth)

The sentences for the is-the-father-of relation are stored in the order

Henry7 is-the-father-of Henry8
Henry8 is-the-father-of Mary
Henry8 is-the-father-of Elizabeth
Henry8 is-the-father-of Edward

PROLOG compares the sentence pattern

x is-the-father-of Elizabeth

with each sentence in turn. There is an exact match with the third sentence when the variable x has the value "Henry8". At this point PROLOG abandons the search and gives the answer "YES".

Example 3

Does(x is-the-father-of x)

This query is asking whether the data base contains any fact that says that someone is their own father. PROLOG will give us the answer "NO", but it is instructive to see why.

PROLOG tries to match the sentence pattern

x is-the-father-of x

with each of the above sentences. It gets a partial match with the first sentence

Henry7 is-the-father-of Henry8

by giving x the value "Henry7". This makes the sentence pattern become the sentence:

Henry7 is-the-father-of Henry7

But it is not an exact match because by giving x this value PROLOG is implicitly replacing both occurrences of x by "Henry7". This creates a mismatch between the names of the children. The same thing happens in the attempt to match all the other sentences of the data base. So the query is answered, "NO".

Now consider the query

Does(x is-the-father-of y)

In answering this query, PROLOG does not encounter the same problem because it can give the different variables x and y different values. In fact there is an immediate match with x=Henry7 and y=Henry8.

In answering a query PROLOG can give different variables different values, but it may also give them with the same value. Thus, if we had a data base that contained just the single "likes" sentence

Tom likes Tom

then both

Does(x likes x)

and

Does(x likes y)

would be answered affirmatively. In the second query we are asking whether the data base knows anything about some x liking some y. It does, when x and y are the same person Tom. This convention that different variables can stand for the same unknown person PROLOG inherits from symbolic logic. To insist that different variables name different individuals we must add an extra condition that says just that. We shall see how we can do this in chapter 3.

Evaluation of simple "Which" queries

The simple "Which" query is of the form

Which(P S) where P is an answer pattern and S is a simple sentence pattern.

PROLOG takes the sentence pattern S and compares it with each of the sentences for its relation in the data base. A match of S with a sentence in the data base results in each variable of S being given a value. For each match the answer pattern P is displayed with its variables replaced by the values given for that match.

Example 4

&. Which(x Henry8 is-the-father-of x)

The sentences of the data base are compared with the query pattern in the listing order given above. There is no match with the first sentence

Henry7 is-the-father-of Henry8

because the fathers "Henry8", "Henry7" do not match. There is a match with the second sentence,

Henry8 is-the-father-of Mary

providing x=Mary. Because it has found a sentence that matches the query pattern PROLOG has found one answer to the query. It therefore prints out the answer pattern, x, with x replaced by the value "Mary". We get the answer:

Answer is Mary

The evaluation continues with the attempt to match the query pattern "Henry8 is-the-father-of x" with the remaining sentences:

Henry8 is-the-father-of Elizabeth
Henry8 is-the-father-of Edward

There is a match with the first of these providing x=Elizabeth. So we get the second answer:

Answer is Elizabeth

There is also a match with the last sentence, providing x=Edward. This gives us the last answer

Answer is Edward
No (more) answers

Evaluation of compound "Which" queries

We will illustrate the way that PROLOG answers compound queries by two examples.

Example 5

&. Which(x Henry8 is-the-father-of x & Male(x))

This query is a restriction on query of example 4 to find only the male children of Henry8. What PROLOG has to do is to find all the x's such that both

Henry is-the-father x

and

Male(x)

are sentences of the data base.

It finds all these x's by initially ignoring all but the first condition of the compound query. It starts by looking for all the x's that satisfy

Henry8 is-the-father-of x

We know that there are three sentences of this form, the first one being

Henry8 is-the-father-of Mary

PROLOG matches the query condition with this sentence and finds a possible answer, x=Mary, for the compound query. At this point PROLOG interrupts the search for solutions to the first condition in order to see whether this value for x is compatible with the second condition of the query, the condition Male(x). It sees whether it can find a successful match for Male(x) with x already given the value "Mary". This is equivalent to finding a successful match for the query condition

Male(Mary)

It tries to confirm this condition by searching the list of sentences about the "Male" relation. Since it does not find the sentence Male(Mary), it cannot confirm the extra condition on x, when x=Mary. It therefore returns to its interrupted search for all the solutions to

Henry8 is-the-father-of x

It finds the next solution to this with the match against the sentence

Henry8 is-the-father-of Elizabeth

This gives the value x=Elizabeth. Again, PROLOG interrupts the search for other solutions to this first condition to check if Male(x) can be confirmed when x=Elizabeth. That is, it checks to see if the condition Male(Elizabeth) can be confirmed. This attempt also fails. So PROLOG again returns to its interrupted search for all the x values that satisfy the condition

"Henry8 is-the-father-of x".

It finds the next value with the match against

Henry8 is-the-father-of Edward

which makes x=Edward. Interrupting the search once more, PROLOG tries to confirm

Male(x) (with x=Edward), which is Male(Edward).

This time it succeeds, for the sentence Male(Edward) is in the data base. PROLOG has at last found an answer to the compound query, which it prints out.

Since we want all solutions, PROLOG once more returns to its interrupted search for x's that satisfy "Henry8 is-the-father-of x". There are no more because PROLOG has already looked at all the sentences that match this pattern. It therefore prints out "No (more) answers".

Example 6

&. Which((x z) x is-the-father-of y & y is-the-father-of z)

This is a request for all the pairs of people in the paternal grandfather relation. The answers to this query are the names assigned to x and z for each solution to the compound condition query pattern:

x is-the-father-of y & y is-the-father-of z

A solution is an assignment of values to variables in this query pattern such that each of its sentences become facts in the data base. In this case, it is an assignment to x, y, z such that

x is-the-father-of y

and

y is-the-father-of z

are sentences of the data base.

Again, PROLOG searches for all the solutions to the compound query by initially ignoring all but the first condition

x is-the-father-of y.

It starts by looking for all the solutions to this condition. It finds the first solution with the match against

Henry7 is-the-father-of Henry8

which makes x=Henry7, y=Henry8. At this point PROLOG interrupts its search for all the solutions to the first condition. It now looks for all the solutions to the rest of the query which are compatible with this solution (x=Henry7, y=Henry8) to the first condition. In other words, it looks for all solutions to the condition

    y is-the-father-of z (with x=Henry7, y=Henry8)

which is the condition

    Henry8 is-the-father-of z.

There are three solutions to this:

    z=Mary, z=Elizabeth, z=Edward.

So PROLOG has found three solutions:

    x=Henry7, y=Henry8, z=Mary
    x=Henry7, y=Henry8, z=Elizabeth
    x=Henry7, y=Henry8, z=Edward

to the compound condition

    x is-the-father-of y & y is-the-father-of z.

As it finds each solution it prints out the answer pattern (x z) with the variables replaced by their solution values. Hence PROLOG gives us:

    Answer is (Henry7 Mary)
    Answer is (Henry7 Elizabeth)
    Answer is (Henry7 Edward)

as its first three answers.

Since PROLOG has found all the answers to the second condition "y is-the-father-of z" for y=Henry8 it can only find more answers to the query by returning to its interrupted search for all solutions to first condition "x is-the-parent-of y". The next solution it finds is

    x=Henry8, y=Mary

produced by the match with

    Henry8 is-the-father-of Mary.

PROLOG again interrupts the search for all the solutions to "x is-the-father-of y", to find all the solutions to the remaining conditions

    y is-the-father-of z (with x=Henry8, y=Mary)

which is

    Mary is-the-father-of z

There are no solutions to the condition for there are no matching sentences in the data base. So the x=Henry8,y=Mary solution to the first condition does not produce any solutions to the compound query.

Once more PROLOG returns to its search for solutions to "x is-the-father-of y". The last two solutions it finds are:

    x=Henry8, y=Elizabeth
    x=Henry8, y=Edward

On finding each solution PROLOG interrupts its search to look for all solutions of y is-the-father-of z with the y it has found. The first solution

causes it to look for all solutions to

    Elizabeth is-the-father-of z,

and the second causes it to look for all solutions to

    Edward is-the-father-of z.

In each case, there are no solutions; there are no values for z that make them sentences of the data base. So PROLOG finds no more answers to the original query.

General evaluation method

From these two examples we can see that micro-PROLOG satisfies the conditions of a compound query from left to right. When it finds a solution to the first condition it passes the solution on to the following conditions. It then finds all the solutions to the remainder of the query that are compatible with the solution to the first condition it has just found. To find more answers, it returns to look for the next solution to the first condition. It then finds all the solutions to the remainder of the query that are compatible with this second solution, and so on. The evaluation stops when micro-PROLOG can find no more solutions to the first condition. The evaluation method can be summarised by:

    To find all the solutions to a compound query:
        for each solution to the first condition
          (i.e. for each successful match of the first condition
           with a sentence in the data base)
        find all the compatible solutions of the remainder of the query.

If the remainder of the query is a compound condition this method of evaluation again applies. Notice that this means that the first condition in which a variable appears is the one that is used to find different candidate values for the variable. It is the generator of a set of possible values for the variable that are passed on and checked by the later conditions of the query.

Evaluation of compound "Does" queries

The evaluation of a "Does" query with a compound condition containing variables proceeds in exactly the same way as that of a compound "Which" query. PROLOG starts off as though it were trying to find all the solutions for the conjunction of conditions given in the query. It stops as soon as it finds one solution to the query, giving the answer "YES". If it completes the search for all solutions without finding one, we get the answer "NO".

So, to answer a "Does" query such as

    &. Does(Henry is-the-father-of x & Male(x))

PROLOG will again use the first condition, Henry8 is-the-father-of x, to find values for x that might satisfy both conditions of the query. As it finds each x satisfying this condition, it interrupts the search to check whether Female(x) can be confirmed for the x that has been found. If it can, it stops and gives us the answer "YES". If it cannot be confirmed, PROLOG returns to search for the next child of Henry8.

A "Does" query in which the compound query has no variables is checked in the same left to right fashion. In this case, since there are no variable values to find, it becomes a check to see if each query condition is a sentence in the data base. It checks them one at a time, in the left to right order in which they are given.

Exercises 1-4

1. We will add further sentences to our geographical database, giving information about the latitude and longitude of each city, using the form

   city location (latitude longitude)

   with figures given in degrees. Figures North and West are given as positive integers, figures South and East as negative integers.

   ```
   Washington-DC location (-77 38)   (38 77)
   Ottawa        location (-76 45)   (45 76)
   London        location (51   0)
   Paris         location (48  -2)
   Rome          location (41 -12)
   Lagos         location ( 6  -3)
   ```

   Given the PROLOG queries that correspond to the following English questions

   a. Which cities are North of London?
   b. Which cities are West of Rome?
   c. Is there a European country whose capital is North of Rome and South of London?
   d. Which countries in Europe have capitals that are East of London?
   e. In which country and continent is there a city that is South and West of Rome?

2. I have been sent on a shopping expedition, with a database describing the financial situation.

   ```
   Wallet contains 98
   Cheese costs    84
   Bread  costs    40
   Apple  costs    12
   ```

   Obtain answers to the following questions, using PROLOG queries:
   a. How many apples can I afford to buy?
   b. Can I afford to buy the bread and the cheese?
   c. How much is left in my wallet after I have bought the cheese and one apple?
   d. How much more money will I need in order to buy five apples and three loaves of bread?

3. Add information about the year of publication to the books data base using sentences such as:

   Oliver-Twist published 1849
   Great-Expectations published 1853
   Macbeth published 1623

   Guess the dates if need be.

   Pose the following as PROLOG queries:
   a. Was Oliver-Twist published in 1850?
   b. What was published in 1623?
   c. When was Tom-Sawyer published?
   d. Were Oliver-Twist and Great-Expectations published in the same year?
   e. Was Macbeth published before Romeo-And-Juliet
   f. What was published before For-Whom-The-Bell-Tolls
   g. Was anything published before 1600?

1.5 Efficient queries

Now that we know how PROLOG evaluates queries, particularly compound queries, we can see that the way in which we pose a query can effect the efficiency with which PROLOG finds the answers. Thus,

&. Which(x Henry8 is-the-father-of x and Male(x))

and

&. Which(x Male(x) and Henry8 is-the-father-of x)

are logically equivalent queries and will produce exactly the same set of answers. However, in answering the first query, PROLOG will use the condition Henry8 is-the-father-of x to find values for x that it checks with the Male(x) condition. In answering the second, it uses the condition Male(x) to find the different values for x which it then checks with the Henry8 is-the-father-of x condition. So the queries are not behaviorally equivalent. Since, in a more general data base, there will be far fewer children of Henry8 than males, the first query will be answered more efficiently. For each child of Henry8 it will do a search through all the sentences for "Male" relation. In evaluating the second query, for each male recorded in the data base it will have to search through all the sentences for the "is-the-father-of" relation. As a general rule, when a query has two or more conditions on a variable we should put first the condition with the fewest number of solutions.

We must also take into account the order of evaluation of compound queries when we use relations which have restrictions on their use, such as the arithmetic primitives. For example, the queries:

Which(x PROD(17 3 y) and PROD(y 3 x))

Which(x PROD(y 27 x) and PROD(17 3 y))

are logically equivalent but PROLOG will only give us an answer to the first query. We get an answer to this query because it first finds the only solution y=51, to the PROD(17 3 y) condition. It then passes this y value on to the second condition, PROD(y 27 x), which becomes PROD(51 3 x). For this it finds the single solution x=153, which it then gives as the only answer to the query.

In trying to answer the second query, PROLOG encounters the condition PROD(y 27 x) first. This it cannot answer because of the restrictions on the use of the PROD relation. So, when we use an arithmetic primitive in a compound query we should place it after other conditions that can be used to find values for its variables.

2. Basic Logic Programming - using general rules

Often we want to ask the same query many times, in which case it becomes tedious to be always repeating the same long question. Also we want to be able to draw conclusions from the basic information in the data base. For example, that Henry7 is the father of Henry implies that he is a parent of Henry8. We would like to be able to conclude "Henry7 is-the-parent-of Henry8" without having to have this as an explicit fact in the data base. To be able to draw conclusions and to abbreviate queries we need to use rules.

2.1 Turning queries into rules

If we look at exercise 1-2.1(f) we see that we are really asking about the paternal grandfather relation:

```
Which((x y) x is-the-father-of z and z is-the-father-of y)          (A)
```

In a sense the query defines this relation, the pairs (x y) which are produced as answers to the query are in the "paternal-grandfather-of" relation.

If we often wanted to find instances of this relation it would be more convenient if the data base recorded all the instances

```
(Henry7 Mary)
(Henry7 Elizabeth)
(Henry7 Edward)
```

that are given as answers to the query. A straightforward way to do this, is to explicitly record them by adding the simple sentences about the "paternal-grandfather-of" relation:

```
Henry7 paternal-grandfather-of Mary                                  (1)
Henry7 paternal-grandfather-of Elizabeth
Henry7 paternal-grandfather-of Edward
```

We could now get the effect of query (A) with the simpler query

```
Which((x y) x paternal-grandfather-of y)                             (B)
```

There is an alternative to this explicit recording of the instances of the new relation defined by a query. We can add just one sentence that links the new relation to the query pattern that defines it. This new sentence is a rule that gives an implicit definition of the new relation. The rule is expressed using a new form of sentence, the conditional sentence. The "Which" query:

```
Which((x z) x is-the-father-of y and y is-the-father-of z)
```

becomes the rule:

```
x paternal-grandfather-of y if x is-the-father-of z                  (2)
                            and z is-the-father-of y
```

A conditional sentence is "Add"ed to the program in just the same way that ordinary simple sentences are added:

```
&.Add(x paternal-grandfather-of y if x is-the-father-of z
1.                                 and z is-the-father-of y)
```

The rule (2) is equivalent to the set of simple sentences (1). When

used to answer query (B), it has the effect of transforming it into our original query (A).

The descriptive reading of the rule is:

```
x is a paternal grandfather of y if x is the father of z and
                                    z is the father of y, for some z.
```

The prescriptive or procedural reading reflects the way it is used. We should read it as:

```
To answer a query of the form x paternal-grandfather-of y,
    answer the compound query:  x father-of z and z father-of y
```

Using several rules

Sometimes it takes more than one "Which" query to completely 'cover' a relation. For example if we want a list of parents and children, because we do not have this information explicitly stated, we would have to use the two queries:

```
    Which((x y) x is-the-father-of y)                                (C)
and
    Which((x y) x is-the-mother-of y)                                (D)
```

We can reduce these queries to rules for the "is-a-parent-of" relation in the same way we did for the "paternal-grandfather-of" relation. Taking (C) and (D) in turn we get the two rules:

```
    x is-a-parent-of y if x is-the-father-of y                       (3)
and
    x is-a-parent-of y if x is-the-mother-of y                       (4)
```

Adding these to the program gives us two rules which together define the "is-a-parent-of" relation. Both rules contribute towards the definition: there is no sense of exclusive definition. In general, many rules can contribute towards a definition of a relation, and we can even describe a relation by a mixture of facts and rules.

In technical English our two PROLOG rules can be read:

```
x is a parent of y if x is the father of y        (rule 3)
                 and
x is a parent of y if x is the mother of y        (rule 4)
```

Providing the data base contains all the facts about the mother and father relationships for some group of people, the definition of the "is-a-parent-of" relation provided by these two rules is just as good as a set of simple sentences giving all the facts about the relation. micro-PROLOG uses the rules to answer queries about the new relation. The way they are used is indicated by the following **imperative** reading of the two sentences:

```
To answer a query of the form x is-a-parent-of y,
   answer the query: x is-the-father-of y.

To answer a query of the form x is-a-parent-of y,
   answer the query: x is-the-mother-of y.
```

Each rule gives us a different way of answering queries about the new relation "is-a-parent-of". Together, they cover all the instances of the relation implicitly given by the "is-the-father-of", "is-the-mother-of" facts of the data base. Thus, to answer the query:

Which(x x is-a-parent-of Elizabeth)

PROLOG will use both rules. Using the first rule transforms the query into:

Which(x x is-the-father-of Elizabeth)

and the second rule transforms it into:

Which(x x is-the-mother-of Elizabeth)

We therefore get the two answers:

Answer is Henry8
Answer is Mary

They come in this order, because the rule (3) was added before rule (4).

Changing variables in rules

If we list the rules for the relation we get:

```
&. List is-a-parent-of
X is-a-parent-of Y if X is-the-father-of Y
X is-a-parent-of Y if X is-the-mother-of Y
&.
```

Again the rules are listed in the order that they were added. But notice that micro-PROLOG has changed our lower case "x" and "y" to upper case "X" and "Y". It can do this because the actual variable names used in a rule are not important. It can replace a variable, without affecting the meaning of the rule, providing the replacement appears in exactly the same position as the variable it replaces. micro-PROLOG changes variable names but never violates this constraint. It actually 'forgets' the original variable names and remembers only the positions that they occupied in the rule.

Conditional Sentences

The rules we have used so far are examples of conditional sentences. A conditional sentence is a sentence of the form

simple sentence if simple sentence [and ... and simple sentence]

A conditional sentence is an implication. The conclusion (called the consequent) is the simple sentence on the left of the "if". The condition of the sentence (called the antecedent) is the simple sentence or a conjunction of simple sentences on the right of the "if".

Any sentence that contains variables is a rule. So far we have only used simple sentences without variables and conditional sentences with variables. The former we have called facts. We can have conditional sentences without variables, e.g.

Bill likes Jim if Jim likes Bill,

and we can have simple sentences with variables, e.g.

Bill likes x (Bill likes everyone).

In the next chapter we shall have frequent need of these simple sentence rules. For the time being we shall continue to use only facts

(simple sentences without variables) and conditional rules (conditional sentences with variables).

The set of all the facts in a PROLOG program is its data base. The conditional rules enable us to abbreviate queries by defining new relations in terms of the relations of the data base. When queried about these new relations PROLOG uses these rules to interrogate the data base.

Declarative reading of a conditional rule

Suppose we have a conditional rule of the form

S if C

Let y1,...,yk be the variables of the sentence that only appear in the antecedent C. We can read the rule as the implication:

S if C, for some y1,...,yk.

It is understood that each variable in the consequent S represents an arbitrary individual. The conclusion S is true whenever the condition C is true for some values of the variables y1,...,yk.

Procedural reading

The procedural reading of the rule is:

to answer a query of the form S, answer the query: C.

Exercise 2-1

1. Using the Tudor royal family data base, add rules to define the following relations:
   a. "is-maternal-grandmother-of"
   b. "is-a-grandparent-of"
   c. "is-a-grandchild-of"

2. Using the geographical example developed in exercises, complete these rules:
   a. x city-in Europe if .....
   b. x North-of London if .....
   c. x West-of y if .....

3. Using the books example developed in exercises, express the following information as rules added to the program:
   a. A book is classified as fiction if it is a novel or a play.
   b. Anything written by William Shakespeare or Charles-Dickens is a classic.
   c. Any book published after 1900 is contemporary literature.

4. Write a data base describing your own family tree, using appropriate names of relationships.

Rules can use rule defined relations

The relations that we have defined using rules can themselves be used in rules to define further relations. We can build up a hierarchy of such relations with the data base relations at the bottom. We can, for instance, define the relationship "is-a-grandparent-of". In semi-English we would say:

Somebody x is a grandparent of somebody y
    if x is the parent of z and z is a parent of y, for some z.

We can add a conditional sentence to our program expressing this rule:

```
x is-a-grandparent-of y if x is-a-parent-of z and z is-a-parent-of y
```

The imperative reading of the rule is:

To answer a query of the form x is-a-grandparent-of y,
  answer the query: x is-a-parent-of z and z is-a-parent-of y

These rules make use of the "is-a-parent-of" relation which is itself defined by rules. This does not matter. PROLOG can use this rule defining the grandparent relation independently of whether the parent relation is defined explicitly by facts in the data base, or implicitly by rules. It discovers which is the case, and behaves accordingly, when it reduces a query about "is-a-grandparent-of" to the compound query about "is-a-parent-of".

<u>The program so far</u>

Our program, from simple beginnings, has now grown somewhat. To conclude its development at present, let us list it in its current state, to see what our changes have produced.

```
& List All
Henry7 is-the-father-of Henry8
Henry8 is-the-father-of Mary
Henry8 is-the-father-of Elizabeth
Henry8 is-the-father-of Edward
Elizabeth-of-York is-the-mother-of Henry8
Katherine is-the-mother-of Mary
Jane is-the-mother-of Edward
Anne is-the-mother-of Elizabeth                        facts
Male(Henry7)
Male(Henry8)
Male(Edward)
Female(Elizabeth-of-York)
Female(Katherine)
Female(Mary)
Female(Elizabeth)
Female(Anne)
Female(Jane)

x paternal-grandfather-of y if x is-the-father-of z
                             and z is-the-father-of y
x is-a-parent-of y if x is-the-father-of y             rules
x is-a-parent-of y if x is-the-mother-of y
x is-a-grandparent-of y if x is-a-parent-of z
                           and z is-a-parent-of y
&.
```

<u>Exercise</u> 2-2

1. Give PROLOG rules that define
   a. x is-the-grandfather-of y
   b. x is-the-grandmother-of y

2. Answer the following PROLOG queries about the Tudor royal family data base:

   a. Which(x x is-a-parent-of y)
   b. One(x Henry7 is-the-grandfather-of x)
   c. Does(Henry8 is-a-parent-of x and y is-the grandfather-of x)
   d. Which(x x is-the-mother-of y and Henry8 is-the-father-of y)

3. Give the PROLOG queries that would be needed to translate the following English questions:
   a. Who was Edward's paternal grandmother?
   b. Who are the mothers of Henry7's grandchildren?
   c. Did Katherine have a male child?
   d. Who was the mother of a male child of Henry8?

4. Using the geographical data base, express the following questions as PROLOG queries:
   a. What cities are there in Europe?
   b. Is anywhere north of London?
   c. Which places are north of London and west of Rome?

5. With regard to your books program, express the following questions as PROLOG queries:
   a. Which books are classics?
   b. Who wrote books published before 1900?

<u>More on answer patterns</u>

So far answers to queries have just been values for variables given in the answer pattern of the query. We can also have text printed out with each answer. We simply insert the text in the answer pattern of the query. As an example, consider the query:

```
English: What are the names of mothers and their children?
PROLOG:  Which((x y) x is-the-mother-of y)
         Answer is (Elizabeth-of-York Henry8)
         Answer is (Katherine Mary)
         Answer is (Jane Edward)
         Answer is (Anne Elizabeth)
         No (more) answers
```

we just get the pairs of names, which is not very informative. It would be better to get the message:

```
Answer is (Elizabeth-of-York is the mother of Henry8)     and
Answer is (Katherine is the mother of Mary)               etc.
```

in which the inserted text "is the mother of" helps us to interpret the answer. Each of these answers are instances of the answer pattern

(x is the mother of y).

To get the message, we use this pattern instead of the pattern (x y) of the original query:

```
PROLOG:  Which((x is the mother of y) x is-the-mother-of y)
         Answer is (Elizabeth-of-York is the mother of Henry8)
         Answer is (Katherine is the mother of Mary)
         Answer is (Jane is the mother of Edward)
         Answer is (Anne is the mother of Elizabeth)
         No (more) answers
```

We have simply added text to affect the form of our printed answer. The text is only coincidentally similar to the query pattern "x is-a-mother-of y". We can insert <u>any</u> text into the list of variables of an answer pattern. It has no effect on the query evaluation. The only constraint is that the variables must be separated from the text by spaces. If they are not, they become part of the text and their values will not be printed.

2.2 How queries involving rules are evaluated

We shall just consider the case of the evaluation of "Which" queries. The other query forms are answered in exactly the same way. The only difference is that for a "One" query we can exit the evaluation each time an answer is found and for a "Does" query the evaluation is always stopped when one solution to the query condition is found. We shall also review the general method used by micro-PROLOG to find all the solutions to the conjunction of conditions of a compound query. This method applies whether the relations of the query are defined by a sequence of facts, by general rules or a mixture of the two.

A compound Which query is of the form:

&.Which(P S and S'...)

where S and S' are simple sentences. The query pattern S and S'... will contain variables, some or all of which will appear in the answer pattern P. What PROLOG must do is find all the solutions to the compound condition. It must find all the different ways in which the variables of the compound condition can be given values so that each of its simple sentences is in the data base, or can be inferred from the data base using the rules. For each solution that it finds, it prints out the answer pattern P.

PROLOG begins its search for all the solutions to the query by searching for a solution to the first condition S. As soon as it finds a solution it interrupts its search. If S contained variables the solution comprises values for these variables. PROLOG now looks for all the solutions to the rest of the compound query that are compatible with these values. In effect, it 'passes on' the values for the variables in S that solve S to the rest of the query. When it has found all the solutions to the rest of the query that are compatible with this first solution to S, it returns to find the next solution to S. On finding the next solution, it again immediately passes this solution on to the rest of the query. Only when it has found all the solutions to the rest of the query compatible with this second solution to S does it return to look for the next solution to S. It continues in this way until it can find no more solutions to S.

Backtracking

The way that PROLOG searches for all the solutions to a compound condition is called a backtracking search. When PROLOG finds a solution to the first condition S, and passes it on to the remaining conditions S'..., it is 'tracking forward'. When it returns to find the next solution for S, it is 'tracking backward', or backtracking.

The evaluation of a compound "Which" query is a forwards and backwards shuffle through the conditions of the query. Let us suppose that there are three conditions

S & S' & S".

PROLOG finds the first solution to S and passes it on to

S' & S".

It now looks for all the solutions to S' & S" that are compatible with this solution to S. It again starts by looking for a solution to the first condition S'. It tries to solve S' with the variable values given by the first solution to S. If it can do this, it moves forward to S". It tries to solve S" with the variable values given by the solution to S & S' that it has now found. When it has found all these solutions to S", it backtracks to look for the next solution to S'. It shuffles backwards and forwards between S' and S" until it has found all the solutions of

S' & S"

compatible with the first solution to S. At that point, it backtracks to look for the next solution to S.

The process of 'passing' on solutions to the rest of the query represents a flow of 'information' from left to right in the query. The first condition in which a variable appears is the generator of values for that variable. These values are passed on to the other conditions of the query in which the variable appears.

This backtracking search for all the solutions to a compound query applies irrespective of whether the relations in the query are defined by facts, rules or a mixture of the two. The difference occurs only when micro-PROLOG picks off a condition S in the query and starts to look for all the solutions for that condition.

Let us suppose that the condition S refers to a rule defined relation R. micro-PROLOG searches for solutions to the condition S as for a data base relation. It scans the list of sentences about R looking for a match with the query condition. It scans them in the order in which they were added to the program (the order in which they are listed by the "List" command).

The extra complication is that it now has to match the query condition with the consequent of a rule, which may contain variables. Then, even when it has found a match, it has not yet found a solution. It must interrupt its scan of the sentences for R to find a solution to the query given by the antecedent of the rule. Each solution to this auxiliary query is a solution to the condition S.

Each time it finds a solution to the auxiliary query micro-PROLOG interrupts its search to pass the solution on to any remaining conditions of the original query. Now, backtracking to find the next solution to S means backtracking to look for the next solution to the auxiliary query. When it has found each solution to the auxiliary query, it returns to its scan of the program sentences for the relation R. Each rule with a consequent that matches S gives rise to an auxiliary query. The solutions to each of these auxiliary queries combine to give all the solutions to S.

Example evaluation

Let us illustrate the invocation of rules during the evaluation of a query by a simple example. Consider the query:

Which(y Henry7 is-the-grandfather-of y). (E)

We shall assume that the rule

x is-the-grandfather-of y if x is-the-father-of z and
z is-a-parent-of y (5)

has been added to the Tudors program. (This was one of the answers to exercise 2-1.) PROLOG must find all the values for the variable y that are solutions to the query condition:

Henry7 is-the-grandfather-of y (F)

There is only one sentence in the data base about this relation, the rule (5) given above. Now, remember that PROLOG forgets the variables used in a rule. It remembers only their positions. When it starts to match a condition with the consequent of the rule it gives the variables of the rule names. It always gives them names that are different from the variable names used in the query condition. Let us suppose it gives the x variable of the rule the name x1, the y variable the name y1, and the z variable the name z1. PROLOG must match the query condition (F) with the consequent of the rule

```
x1 is-the-grandfather-of y1 if x1 is-the-father-of z1 and
                               z1 is-the-parent-of y1
```

Matching is now a little more complicated. To obtain a match, variables of the query condition and variables of the rule may be given values. In this case only variables of the rule are affected. The values x1=Henry7 and y1=y give an exact match. Notice that y1 has a value which is not the name of an individual but the name of a variable in the query. With x1 and y1 given these values the antecedent of the rule becomes the compound condition

```
Henry8 is-the-father-of z1 and z1 is-the-parent-of y
```

The problem of finding all the y values that solve condition (F) has become the task of finding the answers to the auxiliary query

```
Which(y Henry7 is-the-father-of z1 and z1 is-the-parent-of y)        (G)
```

This is solved in the usual way. PROLOG starts by looking for a solution to the condition Henry7 is-the-father-of z1. There is only one solution to this, but immediately this is found, by the match with the fact

```
Henry7 is-the-father-of Henry8
```

PROLOG interrupts its scan of the "is-the-father-of" sentences to find all the solutions to the next condition

```
z1 is-the-parent-of y
```

that are compatible with z1=Henry8. PROLOG has temporarily reduced query (G) to the query

```
Which(y Henry8 is-the-parent-of y)                                   (H)
```

We have another rule defined relation. This time there are two rules, which with renamed variables are:

```
x2 is-a-parent-of y2 if x2 is-the-father-of y2
x2 is-a-parent-of y2 if x2 is-the-mother-of y2.
```

The query condition "Henry8 is-the-parent-of y" matches both rules providing x2=Henry8, y2=y. PROLOG tries these rules one at a time, in the above order. After the successful match with the first rule, PROLOG temporarily replaces (H) by

```
Which(y Henry8 is-the-father-of y)
```

The three solutions of this query become solutions of (G) which are, in turn, solutions of the original query (E). They are printed out. PROLOG returns to the task of answering (H). It uses the second rule for

"is-a-parent-of". This gives rise to the auxiliary query

```
Which(y Henry8 is-the-mother-of y)
```

to which there are no solutions.

Remember (H) was produced when PROLOG found the first solution to the first condition of the query

```
Which(y Henry7 is-the-father-of z1 and z1 is-a-parent-of y)
```

To find more solutions to the query, and hence more solutions to the original query, it returns to the task of solving the condition

```
Henry7 is-the-father-of z1.
```

It continues its scan of the data base sentences for "is-the-father-of". There are no more solutions. PROLOG must now return to the original query

```
Which(y Henry7 is-the-grandfather-of y)
```

to see if there are other sentences in the data base about "is-the-grandfather-of". It has already used the one and only sentence. So the search for solutions stops.

2.3 Recursive descriptions of relations

So far our rule defined relations have been such that they could be dispensed with. Queries using these relations could always be expanded to longer queries that used only the relations of the data base. This is because each rule defined a new relation solely in terms of previously defined relations. There are some relations that cannot be so simply defined. These are relations that can only be described recursively, by definitions that refer back to the relation being defined. For such relations the use of rules is essential. As an example, suppose that our data base describing the Tudor family tree had many generations in it, and that we wanted to query the data base to find all the ancestors of Edward. If we knew that the data base referred to exactly four ancestors of Edward we could find all of them with the query:

```
Which((x1 x2 x3 x4) x1 parent-of-x2 and x2 parent-of x3
                    and x3 parent-of x4 and x4 parent-of Edward)
```

But if we do not know how many ancestors are given in the data base we cannot find all the ancestors with a single query. This is because we cannot know how many "parent-of" conditions will be needed to chain back to the earliest recorded ancestor. To find all the ancestors with a single query, we need to define the relation "is-an-ancestor-of".

If we wanted to explain to someone who his ancestors are we might say:

Your ancestors are your parents and all the ancestors of your parents.

This is a recursive description because the explanation makes use of the concept being explained. If he 'thinks through' the definition it tells him that his ancestors are:

his parents
his grandparents (who are the parent case ancestors of his parents)
his great-grandparents (who are the parent case ancestors of his grandparents)
his great-great-grandparents (who are the parent case ancestors of his great-grandparents),
.
.
and so on until the records run out.

We can express this recursive definition as the pair of PROLOG rules:

```
x is-an-ancestor-of y if x is-a-parent-of y
x is-an-ancestor-of y if z is-a-parent-of y and x is-an-ancestor-of z
```

The declarative reading is quite simply:

x is an ancestor of y if x is a parent of y.
x is an ancestor of y if z is a parent of y and x is an ancestor of z, for some z.

The procedural reading is:

To answer a query of the form x is-an-ancestor-of y
answer the query: x is-a-parent-of y.

To answer a query of the form x is-an-ancestor-of y
answer the query: z is-a-parent-of y and x is-an-ancestor-of z.

Given the task of finding all the ancestors of Edward by a query:

```
Which(x x is-an-ancestor-of Edward)
```

micro-PROLOG will begin by using the first rule to reduce the query to

```
Which(x x is-a-parent-of Edward)
```

When this is answered, and the parents of Edward are found and listed, it will backtrack to use the second rule. This converts the query into the derived query

```
Which(x z is-a-parent-of Edward and x is-an-ancestor-of z)
```

Since the rule defining a parent as a father comes first, the condition "z is-a-parent-of Edward" will be solved by making z the name of the father of Edward who, in the Tudors data base, is Henry8. Given this value for z, we obtain the new query:

```
Which(x x is-an-ancestor-of Henry8)
```

When this has been answered, and all the ancestors have been found, micro-PROLOG backtracks to the second way of finding a parent of Edward. It retrieves his mother Jane. It then finds and lists all her known ancestors.

Separate definition of inverse relations

Logically our two rules defining the ancestor relation also define the inverse relation "is-a-descendant-of". To find the descendants of Henry8 we could use the query


```
Which(y Henry8 is-an-ancestor-of y)
```

micro-PROLOG will again begin by using the first rule to find and list the children of Henry8. It will then backtrack to expand the query using the second rule to get

```
Which(y z is-a-parent-of y and Henry8 is-an-ancestor-of z)
```

The evaluation of this derived query is a very inefficient search for the descendants of the children of Henry8. For in order to try to satisfy the condition "z is-a-parent-of y" it will try each parent-offspring pair in the data base checking each parent to see if it is a descendant of Henry8. This is an example where a separate description of the inverse relation will serve us better as a program for finding descendants.

The problem is to do with the flow of values via the variables of the rule. The rule:

```
x is-an-ancestor-of y if z is-a-parent-of y and x is-an-ancestor-of z
```

gives efficient retrieval if y is given. For then the first condition "z is-a-parent-of y", with y known, has a much smaller set of possible z values to pass on to the "x is-an-ancestor-of z" condition. To get a similar flow for the case when x is given and y is to be found, we should use the given x, find a child z of x, then find all the descendants of z. So optimise the finding of descendants, we should separately define the "is-a-descendant-of" relation by the rules:

```
y is-a-descendant-of x if y is-a-child-of x
y is-a-descendant-of x if z is-a-child-of x and x is-a-descendant-of y
```

These constitute a correct alternative definition of the relation that holds between two people x and y when x is an ancestor of y and y is a descendant of x. For purely pragmatic reasons, we should use these rules for finding descendants and the ancestor rules for finding ancestors. For checking whether two people are in the ancestor/descendant relation either set of rules can be used. The queries:

```
Does(Henry8 is-an-ancestor-of Edward)
Does(Edward is-a-descendant-of Henry8)
```

are logically equivalent. micro-PROLOG does comparable work in answering each query. To answer the first it walks over the family tree beginning at Edward, for the second it begins at Henry8. If the families described in the data base have on average more than two children, the "is-an-ancestor-of form" of the query should be used. Why?

Exercise 2-3

1. Answer the following PROLOG queries, using the Tudor royal family data base:
   a. Which((x is male grandchild of y) x is-a-grandchild-of y & Male(x))
   b. One((x is a wife of Henry8) y is-a-child-of Henry8 & x is-the-mother-of y)
   c. Which(x x is-an-ancestor-of Edward)
   d. Which(x x is-a-descendant-of Elizabeth-of York)
   e. Does(Henry8 is-a-descendant-of Mary)
   f. Which(x x is-a-descendant-of Henry7 and Female(x))

2. Add the "is-an-ancestor-of" and "is-a-descendant-of" rules to your family tree data base. Use PROLOG queries and trace the order in

which answers are received.

3. We have used the built-in predicate LESS. This can also be used to define rules for other relations (as can the other built-in predicates). For instance, to define the relation "lesseq" (which means less than or equal to) we need just two rules:

    x lesseq x

This rule simply states that everything is less than or equal to itself. The other rule is:

    x lesseq y if x LESS y

This rule says that if two numbers (or words) are in the LESS relation then they are also in the lesseq relation.

a. Define the relation "greater-than".
b. Define the relation "greateq" (greater than or equal to).
c. Define the relation "divisible-by".

Notice that because of the restrictions on the use of the arithmetic primitives your rules for these relations can only be used for confirming.

4. Using the books data base, add rules defining the relations:

a. Nineteenth-Century-Author(x) : x has written a book published in the 19th century.
b. Contemporary-Playwright(x) : x has written a play published in the 20th century.

Add rules to express the following information:

c. A book is available from the time it is published.

Express the following questions as PROLOG queries:

d. What books were available in 1899?
e. What works of nineteenth century authors were available in 1980?

3. Lists

3.1 Lists as Individuals

So far we have only seen how to handle facts that referred to single individuals. Sometimes it is more convenient to have a fact that refers to a list of individuals. This is quite common in English. We say:

    John enjoys football, cricket and rugby

which is a fact that relates John to the list (football cricket rugby) of games that he enjoys. We can represent this compound fact in PROLOG by three simple sentences:

    John enjoys football   (1)
    John enjoys cricket
    John enjoys rugby

We can also represent it by a single sentence:

    John enjoys (football cricket rugby)   (2)

in which we collect together the games that John enjoys as a list (football cricket rugby). The query:

    &. Which(x John enjoys x)

used with this single sentence program (2) will produce the response:

    Answer is (football cricket rugby)
    No (more) answers

because the pattern "John enjoys x" matches the data base sentence only when x is this list. The advantage of using lists in place of single individuals is that we often get a more natural and compact representation of information. The disadvantage is that we must sometimes do some work to get at the individuals in a list. With the information about John represented by the three sentences (1) we can directly query the data base about individual games. The query:

    &. Does(John enjoys football)

will return the answer "YES". But for representation (2) the query will get the answer "NO". This is because there is no sentence in the data base that exactly matches the query. To find out if John enjoys football we must be able to get at the components of the list of games (football cricket rugby).

Exercise 3-1

1. You have this PROLOG program:

    (Tom Dick Harry) knows Susie
    Tom knows (Jane Janet Julia)

Answer these PROLOG questions:

a. Does(Tom knows Susie)
b. Which(x x knows Susie)
c. Which(x Tom knows x)

2. You have this PROLOG program:

(Wimbledon Morden Mitcham) part-of Merton
(Hampton Teddington Ham) part-of Richmond
(Surbiton Norbiton) part-of Kingston

Answer these PROLOG questions:

a. Which(x x part-of y)
b. Does(x part-of Kingston)
c. Which(x y part-of x)
d. Does(x part-of Merton and x part-of Richmond)

3. Rewrite the books data base using lists. For example, the sentence:

    Oliver-Twist written-by Charles-Dickens

   should now read:

    (Oliver Twist) written-by (Charles Dickens)

   (This enables us to separate author's surnames from their first names)

3.2 Getting at the members of a list of fixed length

To get at the components of a list we have to elaborate the idea of forms, patterns and pattern-matching introduced earlier. To illustrate these ideas, let us look at a different way of representing information about family relationships which makes use of lists.

Initially we recorded the parent-child information by having separate sentences giving each of the children of each parent. Using lists we can collect together all the information about a particular family in one sentence of the form:

    (father mother) parents-of (all the children of the marriage)

The simple sentences of the data base are now sentences such as:

    (Henry Sally) parents-of (Margaret Bob)
    (Henry Mary) parents-of (Elizabeth Bill Paul)
    (Bill Jane) parents-of (Jim)
    (Paul Jilly) parents-of (John Janet)

The two sentences which have Henry as the father are data for two different marriages. The sentence

    (Bill Jane) parents-of (Jim)

records the only child of the marriage of Bill and Jane in a list with just one name. In this case, we might have expressed this information in the sentence

    (Bill Jane) parents-of Jim

But then our facts about families would not have all been of the same form. In some we would have lists of children, in some just single names. It is important that all sentences about a relation all have a uniform pattern. PROLOG retrieves data by matching sentences with patterns, and patterns are critical when we use lists. So, for uniformity, we have recorded the only child in a list of one name.

The expression "(Jim)" is a list because of the brackets. If we drop the name altogether, writing "()", we have a list of no names: we have an **empty** list. We can use the empty list to record information about families with no children. We can have a sentence such as:

    (Samuel Sarah) parents-of ()

This records the fact that Samuel and Sarah are man and wife, and it tells

us they have no children. (To represent this using our previous notation would have required an auxiliary relation "is-married".)

Suppose that we now want to retrieve the children of Henry. The data giving the children for a family in which Henry is the father is contained in all the sentences of the form:

    (Henry y) parents-of x

So the query is:

    &. Which(x (Henry y) parents-of x)
    Answer is (Margaret Bob)
    Answer is (Elizabeth Bill Paul)
    No (more) answers

Notice that we get the children from the different marriages as different list answers. This is because the query pattern matches two different sentences each of which give x as a list.

Consider the sentence pattern

    (x y) parents-of (x1 x2 x3)

This will match any fact in the data base about a family with three children x1, x2, x3. We can therefore use this to retrieve information about all the three child families.

    &. Which((children x1 x2 x3 father x mother y)
             (x y) parents-of (x1 x2 x3))
    Answer is (children Elizabeth Bill Paul father Henry mother Mary)
    No (more) answers

Here we have used an output pattern to rearrange the retrieved data and to give some documentation. The pattern

    (x y) parents-of z

matches every fact in the database about families. In this pattern x is the father, y is the mother and z the list of children.

We can, therefore, define "father-of-children" and "mother-of-children" relations with the rules:

    x father-of-children z if (x y) parents-of z
    y mother-of-children z if (x y) parents-of z

And a typical query to find the children of Jilly would be:

    &. Which(z Jilly mother-of-children z)
    Answer is (John Janet)
    No (more) answers

We get a list of children because we have defined "mother-of-children" as a relation between an individual and the list of children by a single marriage.

Exercise 3-2

1. Using the notation for the empty list, give a definition of the relation Childless-wife(x).

2. Using the example program above, answer the following PROLOG

questions:

a. Which(x (Bill x) parents-of y)
b. Which((x y) (z x) parents-of (x y))
c. Does((Henry x) parents-of (y z X))
d. Which(x (x y) parents-of z)
e. Which((x father y mother z child X child) (x y) parents-of (z X))
f. Which(x Paul father-of-children x)

3. Using the rewritten books data base, answer the following PROLOG questions:

a. Which(x (Oliver Twist) written-by (Charles x))
b. Does((Great x) type Novel)
c. Which((x y) x written-by (Mark y))
d. Which((x was a great playwright) (Macbeth) written-by x)
e. Which(x (x y) written-by z)

3.3 Getting at the members of a list of unknown length

Using the list representation of family relationships we are still not able to check, with a single query, whether or not someone is some particular child's mother. The trouble is that a single pattern cannot cover all the different size lists of children that we can get back in response to a mother-of-children query. The rules:

y mother-of-child x1 if (x y) parents-of (x1 x2)
y mother-of child x2 if (x y) parents-of (x1 x2)

define the mother-of-child relation for two child families because two child families are recorded by sentences of the form (x y) parents-of (x1 x2). Each rule selects out one of the pair of children (x1 x2). But we also need a rule to cover single child families:

y mother-of-child z if (x y) parents-of (z)

and rules for three, four and even bigger size families.

We can make do with a single rule:

y mother-of-child z if (x y) parents-of Z and z belongs-to Z

if we could define the relation z member-of Z that holds for every individual z that appears in an arbitrary size list of individuals Z.

Heads and Tails

An arbitrary size list is of the form

$$(\underset{\text{head}}{x1}\quad \underbrace{x2\ \ldots\ xn}_{\text{tail}})$$

Let us call the first individual in the list, x1, the **head** of the list. If we take away the head element we are left with a list (x2 ... xn) which we shall call the **tail** of the list. The tail of a list that only contains one element, is the **empty** list ().

One rule about membership of an arbitrary size list is:

The head individual of a list is a member of the list. (3)

Another is:

An individual is a member of a list if it is a member of its tail. (4)

Just like our recursive definition of the ancestor relation these two rules enable us to check whether any individual appears on a list.

To formalise these as PROLOG rules we need to have a pattern that enables us to talk about the head and the tail of a list. This is the pattern (x|y).

We read the pattern as:

(x|y) is a list which is x followed by the list y.

The "|" is the "followed by". Without the "|" the pattern (x y) denotes a list of just two elements.

If PROLOG matches (x|y) against the list (A B C D) it gives x the value A and y to the tail list (B C D). If it matches (x|y) against the list (A) comprising just the element A then x is bound to A and y is bound to the empty list (). Other examples of the use of "|" are:

(x y|z)

This denotes a list of two individuals x y followed by some list z. Since z can be the empty list, this denotes any list of two or more individuals. Matched against the list (A B C D) we get the values x=A, y=B, z=(C D). It fails to match the list (A) because this only has one element.

(x y z|Z)

is a list of three individuals x y z followed by some remainder list Z.

We can describe a list of at least n individuals by having n different variables before the "|". We should always follow the "|" with a variable or another pattern that describes a list. For example, (x1 x2|(x3 x4)) is the list x1 x2 followed by the list of two elements x3 x4. In other words, it denotes the list of four individuals (x1 x2 x3 x4). In this case, there is no point in using the "|". Indeed there is only a point in using "|" when we do not know anything about the structure of the remainder of the list, i.e. when we describe it by a variable that can match any remainder.

Exercise 3-3

1. What values if any, are assigned to the variables when (x y z|Z) is matched against:

a. (A B C D E)
b. (A B C D)
c. (A B C)
d. (A B)
e. (A)
f. ()

2. Lists can have other lists as elements, so show the values given to x and y that arise from matching ((A B)|x) and (y C|y). Hint: ((A B)|x) matches any list that has as its first element the list (A B).

3. Suppose that we had the data base:

(Piccadilly Victoria District Circle Northern) lines-of Underground
(Hackney Lambeth Richmond Kingston) boroughs-in London

Answer these PROLOG questions:

a. Which(x (Piccadilly Victoria | x)lines-of Underground)
b. Does((x Victoria | y) lines-of z)

c. Which(x x boroughs-in London)
d. Which((x y) (x Lambeth y Kingston) boroughs-in z)
e. Does((Hackney | x) boroughs-in London)

Belongs-to

Using the "|" pattern, we can express rules (3) and (4) directly as micro-PROLOG rules:

```
x belongs-to (x|z)                          (5)
x belongs-to (y|z) if x belongs-to z        (6)
```

Let us illustrate how this program works, using the list (A B C D E). If we ask:

```
&. Which(x x belongs-to (A B C D E))
```

we first get the answer

```
Answer is A
```

This is produced because rule (5) matches the pattern (x|z) against the list (A B C D E) making x=A, the head of the list.

The next answer is:

```
Answer is B
```

This is produced using rule (6) and then rule (5). Rule (6) matches (y|z) against (A B C D E) and z becomes the tail list (B C D E). It then reduces the query to

```
Which(x x belongs-to (B C D E))
```

As with the original query this is first answered using rule (5) which produces the answer B. A new application of rule (6) then reduces this to the query

```
Which(x x belongs-to (C D E))
```

The evaluation continues in this way, giving us the next two answers C, D until the query has been reduced to

```
Which(x x belongs-to (E)).
```

A last use of rule (5) prints out the answer E. The last application of rule (6) matches (y|z) against the list (E). For the list (E) the tail list is empty. So z is bound to (), and we get the derived query

```
Which(x x belongs-to ())
```

Since there are no rules for belongs-to and the empty list, this query has no answers and the evaluation terminates. The full answer to the query is therefore:

```
Answer is A
Answer is B
Answer is C
Answer is D
Answer is E
```



```
No (more) answers
```

We can now see who are the individual children of Jilly, using our program for "mother-of-child":

```
Which(x Jilly mother-of-child x)
Answer is John
Answer is Janet
No (more) answers
```

Notice that "mother-of-child" is a rule defined relation that is the same as the fact defined relation "is-the-mother-of" of Chapter 1.

Exercise 3-4

1. You have this PROLOG program:

```
(English Welsh Gaelic) spoken-in United-Kingdom
(English French) spoken-in Canada
```

Answer these PROLOG questions:
a. Which(x x spoken-in Canada)
b. Which(x (x|y) spoken-in z)
c. Which(x y spoken-in United-Kingdom and x belongs-to y)
d. Does (x spoken-in United-Kingdom and y spoken-in Canada and z belongs-to x and z belongs-to y)
e. Using the program and queries above, give a definition of the relation British-language(x) which is defined to be a language spoken both in the United-Kingdom and Canada.
f. Assuming that the languages have been listed in order of importance in each case, give a definition of the relation Minor-language(x) where a minor language of a community is not the most important spoken language.

2. Answer these PROLOG questions:

a. Which(x x belongs-to (R O B E R T) and x belongs-to (B O B))
b. Does(x belongs-to (A L F) and x belongs-to (F R E D))

The spaces between the letters in these queries are important; spaces separate the members of a list. The list (R O B E R T) has six elements, each of which is a single letter. However, the list (ROBERT) has just one element, the word "ROBERT". It has one element because there are no spaces.

If you use the micro-PROLOG system to answer (a) you will notice that you get the answer "B" twice. This is because micro-PROLOG can show that "B" also appears on (B O B) in two ways. In answering the compound query, micro-PROLOG finds each letter in (R O B E R T) as a candidate value for x. For each value it looks for all ways of showing that the found x is also on the list (B O B). Each time it succeeds in doing this, it prints out that value for x. If (R O B E R T) had been given as (R O B B E R T), with the two B's instead of one, "B" would be printed out four times. micro-PROLOG would find it twice, and each time twice confirm that it is also on the list (B O B).

3. Using the program developed in section 3.2, give definitions of

a. x is-a-parent-of-children y
b. x is-a-child-of y

In each case make use of the "belongs-to" relation.

3.4 The Length of a List

A very common list program is the "has-length" program which simply counts the length of a list. Although very simple it has many uses and some surprising properties. There are just two sentences in the "has-length" program, a fact and a rule:

```
() has-length 0
(x|X)has-length z if X has-length y and SUM(y 1 z)
```

The declarative reading of these rules is equally simple:

> The empty list has length zero (as might be expected)
> A non-empty list has length one more than the length of the tail sub-list.

To find the length of "(A B C D)" we use the query

```
Which(x (A B C D) has-length x).
```

We can use the "has-length" program to check that a list has a given length:

```
Does(A B C D) has-length 4)
```

We can use it to find the length:

```
Which(x (S B C D) has-length x)
```

Amazingly, we can also use it to find a list of a given length, and to find all instances of the "has-length" relation. The queries

```
One(x x has-length 4)
```

and

```
One((x y) x has-length y)
```

will both be answered by micro-PROLOG. If you have a computer handy, define "has-length" and try the queries. Stop the evaluation of the first query after it has given you one list of length 4. There is only one micro-PROLOG answer to the query. You can run the second query until you get tired of seeing the answers. It is important that you Add the "has-length" fact before the rule.

Let us now examine the way micro-PROLOG answers these queries. This will explain the answers that we get. We will start by examining the query

```
One((x y) x has-length y)                                          (A)
```

This is the same as a Which query with the option of stopping generation of the answers at any point. For this query, having this option is very necessary. There are an infinite number of answers to

```
Which((x y) x has-length y)
```

micro-PROLOG answers the query (A) by scanning the sentences for "has-length" trying to match the sentence with the query condition "x has-length". The first sentence is

```
() has-length 0
```


So there is a successful match with x=() and y=0. This gives us the first answer

```
Answer is (() 0).
```

If we type "C", micro-PROLOG continues with its scan. The second (and last) sentence for the relation is

```
(x1|X1) has-length z1 if X1 has-length y1 & SUM(y1 1 z1)
```

Notice that we have renamed the variables. Remember that micro-PROLOG always does this when it uses a rule. It uses variables that are different from any that appear in the query being evaluated. There is a match between

```
(x1|X1) has-length z1
```

and

```
x has-length y
```

providing x=(x1|X1) and z1=y. Our original query

```
One((x y) x has-length y)
```

is thus reduced to

```
One((x y) X1 has-length y1 & SUM(y1 1 z1))
                             with x=(x1|X1) and y=z1     z1=y
```

This is the derived query

```
One((x1|X1) y) X1 has-length y1 & SUM(y1 1 y))                    (B)
```

The answer to this query are all the remaining answers to (A).

Now, in answering query (B), the condition "X1 has-length y1" becomes a generator for candidate values of X1 and y1. The y1 values are handed over to "SUM(y1 1 y) which finds the value of y of the answer pattern. We know that the first answer micro-PROLOG will give to

```
X1 has-length y1
```

is

```
X1=() and y1=0
```

obtained by the match with the fact "() has-length 0". The passing on of the value y1=0 gives the value y=1. Hence the first answer to (B) (and so the second answer to (A)) is the value of the answer pattern

```
((x1|X1) y)          with X1=() and y=1.
```

This is

```
Answer is ((x1) 1).
```

We get the list (x1) because (x1|()) is the list that is the element x1 followed by the empty list. That is, it is the one element list (x1). Note that x1 is still a variable. The pattern (x1) is the answer:

all lists of just one element.

If we continue the evaluation of (A), the next answer is obtained when the generator "X1 has-length y1" of (B) produces its second answer. But we know what the second answer to the query condition is, for we have already had it given as the second answer to our original query. It is value for X1 that is a list pattern representing all lists of one element, and the value 1 for y1. The value for X1 will be a list pattern, such as (x2). micro-PROLOG will not generate the value (x1), because x1 already appears in query (B). This pair of values, gives us the next answer to (B). It is

```
((x1|X1) y)          with X1=(x2) and y=2
```

But (x1|(x2)) is the list of two variables (x1 x2). So we get

```
Answer is ((x1 x2) 2).
```

You should now see what the general pattern is. The third answer to (B) is produced by using the second answer, with variables replaced, as the next solution given by the generator "X1 has-length y1". It gives us an answer comprising a list of three variables, paired with the length 3. The evaluation continues, always using the last answer to produce the next answer. Our original query

```
One((x y) x has-length y)                                    (A)
```

has an infinite number of different answers, each answer is a list of different variables paired with its length. The answers are generated in order of increasing length.

Notice the importance of the ordering of the sentences for "has-length". If they had been entered there in the order

```
(x|X) has-length z if X has-length y & SUM(y1 z)
() has-length z
```

there would be no real difference in the way micro-PROLOG answers length checking or length finding queries. But in trying the answer query (A), this ordering will cause micro-PROLOG to enter a bottomless pit.

The reason is that micro-PROLOG always uses the <u>first</u> sentence that matches a query condition for a relation. So in answering (A), it will now use the rule before the fact. It first reduces (A) to

```
One(((x1|X1) y) X1 has-length y1 & SUM(y1 1 y))             (B)
```

It is answering this query, it again encounters an "has-length" condition. It will again use the first sentence for the relation, the rule. This effectively replaces (B) by

```
One((x1|(x2|X2)) y) X2 has-length y2 & SUM(y2 1 y1) & SUM(y1 1 y))
```

This expansion continues, and will continue indefinitely. Each step introduces a new query condition for which the rule is the first matching sentences. micro-PROLOG never has a chance to use the fact "() has-length 0)" which gives the first, crucial answer to the query. The moral here is that the ordering of the rules for a recursively described relation is important if they will be used to find instances of the relation. For such a use, we should make sure the facts, (more generally the non-recursive rules) precede the recursive rules.

Let us now examine the way micro-PROLOG answers the query

```
One(x x has-length 4)                                        (C)
```


3.4 The Length of a List

We assume that the sentence for "has-length" are as originally given, with the fact before the rule.

micro-PROLOG first tries to use the fact

```
() has-length 0
```

to match the query condition

```
x has-length 4.
```

It fails to get a match, since 4 and 0 are different. It can only get an answer by using the rule, which with renamed variables, is

```
(x1|X1) has-length z1 if X1 has-length y1 & SUM(y1 1 y z1)
```

There is a successful match, with the query condition providing x=(x1|X1) and z1=4. micro-PROLOG reduces (C) to

```
One((x1|X1) X1 has-length y1 & SUM(y1 1 4))
```

The condition "X1 has-length y1" now becomes a generator for candidate values for X1 and y1 with the y1 value checked with the SUM(y1 1 4) condition. Now we know that there are an infinite number of solutions to the condition and that the solutions will be generated in order of increasing length. When the solution X1=(x2 x3 x4), y1=3 is generated we get the answer (x1 x2 x3 x4) to query (C).

This is, of course, the only answer. But micro-PROLOG does not know this. It will happily continue generating more and more candidate solutions for the condition "X1 has-length y1" checking if the length is one less than 4. If we let it, after giving us the only answer, micro-PROLOG will enter a bottomless pit.

This is similar to the problem that can arise if we do not choose a judicious ordering for the rules of a recursively defined relation. In this case, the problem is that the ordering of the preconditions of the rule

```
(x|X) has-length z if X has-length y & SUM(y 1 z)
```

is not appropriate for the use in which the length is given and a list of that length is to be found. For this use, we should put the SUM(y 1 z) condition first. Note that we cannot do this for the finding length use. For then we would encounter the problem of trying to find a solution to SUM(y 1 z) with both the arguments y and z unknown. As with ancestor-of/descendant-of, we need a separate definition of the inverse relation, "length-of".

The two sentences,

```
0 length-of ()
y length-of (x|X) if SUM(z 1 y) & y length-of X
```

are a definition of the relation with an ordering of the preconditions of the rule that limits the use to queries in which the length of the list is given. But for that use, it is an efficient, safe program. We can even use it to evaluate the query

```
Which(x 4 length-of x)
Answer is (X Y Z x)
No (more) answers
```

This time, micro-PROLOG stops when it has found the only answer, and tells us there are no more answers. Follow through the evaluation by hand. You will see that there is only one answer because the condition SUM(z 1 y), with y given, only has one answer.

Conclusion

To find the length of a list use the "has-length" relation defined by the rules

```
() has-length 0
(x|X)has-length z if X has-length y and SUM(y 1 z)
```

To find a list of variables of a given length, use the "length-of" relation defined by the rules

```
0 length-of ()
y length-of (x|X) if SUM(z 1 y) & z length-of X
```

To check that a given list has a check length, use either relation.

Do not use either relation when both arguments are unknown. This is because there are infinite number of answers to the condition

```
x has-length y
```

and micro-PROLOG will enter a bottomless pit it tries to answer a Which query in which this condition is used. On the other hand, micro-PROLOG will give an error message when trying to answer

```
y length-of x
```

This is because it will try to evaluate a "SUM" condition with two arguments unknown.

Taking into account these sorts of restrictions on the use of micro-PROLOG programs, particularly programs that embody a recursive definition or use the arithmetic primitives, is part of the pragmatics of programming in the language.

Incidentally, the has-length program has no problem finding the length of a list of variables. The query

```
Which((x y) 4 length-of x & x has-length y)
```

will produce the response

```
Answer is ((X Y Z x) 4)
No (more) answers.
```

Exercise 3-5

1. Use the "has-length" program to define a rule which gives the number of children a mother has, and find out how many children Jilly has.

2. a. Pose the query: Who has five children? (use the "has-length" program in your query.)

   b. Pose the same query, but this time use "length-of".

3. Supposing that we had the following information about sporting teams:
   (Arsenal Chelsea Liverpool Manchester-United) teams Soccer
   (Yankees Astronauts Redsox) teams Baseball

Pose and answer the queries:
   a. Which(x y teams z and y has-length x)
   b. Which(x (Arsenal|y) teams Soccer & y has-length x)
   c. Does(x teams y and x has-length 3)

4. Pose the query

   ```
   One(x 2 belongs-to x)
   ```

   Follow through the evaluation, by hand, so that you understand the answers that you get from micro-PROLOG.

Building a chain of descendants

The "length-of" program can be used to construct a list given a number. Programs that can be used to construct lists are exceedingly useful. We shall deal with them more fully in Chapter 5. We shall complete this section by giving a program that is similar to length-of. It can be used to find a list of intermediary parameters that correct two individuals in a parent-of chair. It is a program that defines the relation

parents connect

```
(x y) have-descendant-chain X: y is a descendant of x and
                               X is the list of intermediary parents.
```

Its definition is:

```
(x y) have-descendant-chain () if x is-a-parent-of y
(x y) have-descendant-chain (z|X) if x is-a-parent-of z and
                                  (z y) have-descendant-chain X
```

This program is a classic example of how the data base handling and the list processing sides of PROLOG cooperate. When used to find the ancestor chain between two individuals, the recursive 'walk' over the parents' data base that is performed is combined with the construction of a list. This list reflects the sequence of steps needed to 'complete' the ancestor links between the pair of individuals.

Exercise 3-6

1. Using the program for have-descendant-chain, pose and answer these questions:
   a. What is the list of descendants between Arthur and Robert?
   b. How many generations are there between Jane and Robert?
   c. Give all the pairs of people separated by one intermediary parent, i.e. the grandparent, grandchild pairs.

   Make use of the following facts:

   ```
   Jane is-a-parent-of Arthur
   Arthur is-a-parent-of Peter
   Mary is-a-parent-of Peter
   Peter is-a-parent-of Robert
   ```

2. Define "is-a-great-grandparent-of" in terms of "has-descendant-chain".

3.5 Answer sets as lists

We shall now look more closely at the relationship between information represented by facts about individuals and the same information represented by facts about lists of individuals. We started the chapter by observing that a lot of facts can often be more compactly represented using lists. For example, in the family relationship program of Chapter 1 instead of having sentences about relations such as "is-the-father-of" between individuals we can have sentences about the relation "parents-of" between a list of the two parents and a list of their children.

These two representations of the family information are essentially duals of each other, we can 'move' between them. We have already seen that we can define the "is-the-father-of" relation in terms of the "parents-of" relation using "belongs-to". The definition is:

```
x is-the-father-of y if (x z) parents-of Y and y belongs-to Y
```

Using "belongs-to" we can always define relations over individuals in terms of relations over lists of individuals. Can we do the reverse construction? The answer is YES. We make use of a primitive relation of micro-PROLOG, the "Is-All" relation.

What "Is-All" does is wrap up the set of all answers to a query as a list. Consider the query:

```
Which(y Henry8 is-the-father-of y)
```

The answer to this query is the set of all the children of Henry8. PROLOG prints them out as:

```
Answer is Mary
Answer is Elizabeth
Answer is Edward
No (more) answers
```

Using "Is-All", we can put all these answers into a list in the order in which they are printed. Thus, the query condition:

```
x Is-All (y Henry8 is-the-father-of y)
```

has one answer. x is given the list (Mary Elizabeth Edward) as its value.

We can therefore use "Is-All" to define the relation "is-the-father-of-children" in terms of the "is-the-father-of" relation. The latter relates a father to a single child, the former relates him to the list of all his children. The rule defining the relation is:

```
x is-the-father-of-children Y if Y Is-All (z x is-the-father-of z) (7)
```

Now we can see how to achieve the full mapping from the separate "is-the-father-of" and "is-the-mother-of" facts to the "parents-of" relation:

```
(x y) parents-of Z if Z Is-All(z x is-the-father-of z and
                                 y is-the-mother-of z)
```

Just like a "Which" query the query component of "Is-All" can have a conjunction of simple conditions.

The "Is-All" program has many useful applications, all stemming from its ability to make available in a list all the answers to a query. A simple example is just to count the number of someone's children as in:

```
x has-no-of-children y if z Is-All(X x is-a-parent-of X)
                        and z has-length y
```

Exercise 3-7

1. Give a query which asks how many male children someone (Peter, say) has.

2. To extend our Tudor royal family data base, we could add information about the Kings and Queens of England:

```
Henry7 family Tudor
Elizabeth1 family Tudor
CharlesI family Stuart
George3 family Hanover
```

Pose and answer the following queries:

a. Give the list of the Tudor Kings of England
b. How many Kings of England have there been?
c. How many Stuart Kings have there been?

3. Give the rules which define the relation: the last member of a list. Hint: an individual is the last member of the list which contains only that individual as member, i.e.

```
x last-of (x)
```

otherwise, it is the last member of the tail of a list.

4. Define the relation "(x y) adjacent-on z" which holds when the pair of elements x and y are next to each other somewhere on the list z. Hint: treat the two cases, x and y the first two elements of the list, x and y not the first two elements, i.e. they are adjacent elements on the tail of the list.

Test out your answers to 3 and 4 on various forms of query. Note: C typed with the control key depressed (the control-C combination) will abort any query evaluation that you think may have got into a bottomless pit.

4. Complex conditions in queries and rules

At the end of the last chapter we introduced the "Is-All" relation. "Is-All" is an example of a *complex condition*; it is a new form of simple sentence. There are two other complex conditions that can appear in queries and rules. They are the "Not" condition and the "For-All" condition. In this chapter we introduce these other conditions and describe "Is-All" more formally.

4.1 Negative conditions

Sometimes the condition that we want the retrieved data to satisfy is more naturally expressed by giving a positive condition that it must satisfy and then giving an extra negative condition that it must not satisfy.

As an example, suppose that we wanted to retrieve all the descendants of Henry8 who do not themselves have any children, or rather, who do not have any children recorded in the data base. What we want are the x's such that

x is-a-descendant-of Henry

can be confirmed, but for which the extra condition

x is-a-parent-of y for some y

cannot be confirmed. In micro-PROLOG we express this negative condition using "Not". We pose the query:

Which(x x is-a-descendant-of Henry8 and Not(x is-a-parent-of y))

Since it is a general property of PROLOG that any query expression can be used as the right hand side of a rule, negated conditions can also be used in rules. Thus, the rule:

```
x childless-descendant-of z if x is-a-descendant-of z and
                               Not(x is-a-parent-of y)
```

generalizes the query and defines the binary relation of being a childless descendant.

Syntax of negative conditions

Syntactically, we have a new type of simple sentence which has the form:

Not(C), C a conjunction of simple sentences

Notice that this means that we can have nested negations, for one or more of the simple sentences of C can be a negative simple sentence. The *declarative reading of a negated condition in a query or rule is:*

It is not the case that C for some y1,..,yk

Here, y1,..,yk are all the variables of C that do not appear elsewhere in the query or rule. They are the local variables of the negative condition. Variables that appear in C which also appear elsewhere are its global variables. The above rule is read:

```
x is a childless descendant of z if x is a descendant of z
                                 & it is not the case that x
                                   is a parent of y for some y.
```


We say, "for some y" because y is a local variable of the negated condition. The x is global because it appears in the other condition of the rule and the consequent of the rule.

Another example of the use of negation is in the query:

Which(x x city-of England & x population-is y & Not(y LESS 10000))

Used with a data base of cities and their populations it will give all the English cities of the data base that have a population greater than or equal to 10000.

Restrictions on use of Not

A negated condition can only be used for checking. It cannot be used for generating candidate values for its global variables. This means that in a query a negative condition must be preceded by a positive condition for each of its global variables. In the evaluation of the query these positive conditions will be used to find values for the variables that the negative condition checks.

The *checking restriction on the use of negation* is reflected in its imperative reading:

to confirm Not(C), check that the query C cannot be confirmed.

In other words, the evaluation of the negated condition Not(C) is the evaluation of the query Does(C) with a "NO" answer interpreted as "YES" and a "YES" answer interpreted as "NO".

Let us see what happens if we ignore the positioning rule for negative conditions. Suppose we posed the query about the childless descendants of Henry8 as:

Which(x Not(x is-a-parent-of y) & x is-a-descendant-of Henry8)

When PROLOG evaluates the query it will now encounter the condition Not(x is-a-parent-of y) with x not yet given a value. The evaluation of the condition reduces to the evaluation of

Does(x is-a-parent-of y)

which will, of course, be confirmed. (We have at least one person who is the parent of someone.) Confirmation of the Does query is failure to confirm the Not(x is-a-parent-of y) condition. So PROLOG will immediately print out

No (more) answers.

This incorrect answer is a consequence of not placing the negative check on x after the positive generator for x which is the condition x is-a-descendant-of Henry8. *For safety PROLOG should give us an error message* when it reaches a negative condition in which there is a global variable which has not been assigned a value. This would stop it evaluating the above query because x is an unbound global variable of the Not condition. PROLOG does not give an error message because to check that each global variable has a value each time it evaluates a negative condition is time consuming. The decision was made to put the responsibility for ensuring that this constraint is always satisfied onto the programmer. He must make sure negative conditions will only be used for checking by a suitable ordering of the query conditions.

Negated equalities

One of the most common uses of negation is the condition Not(x EQ y) which checks that the individuals given as x and y do not have the same name. ("EQ" is a primitive relation of micro-PROLOG. Its definition is the unconditional rule x EQ x. If you prefer to use the symbol "=" instead of "EQ" simply add the rule x=x to your program.)

Suppose that we wanted to define the relation

```
x is-a-brother-of y.
```

We must find some query condition that defines the brother relation. Two individuals x and y are brothers if:

they are male	Male(x) & Male(y)
they are different people	Not(x EQ y)
they have a common parent	z is-a-parent-of x & z is-a-parent-of y

This gives us the rule:

```
x is-a-brother-of y if Male(x) & Male(y) & Not(x EQ y) &
                       z is-a-parent-of x & z is-a-parent-of y
```

The negative condition Not(x EQ y) with global variables x and y comes after the positive conditions Male(x), Male(y) that will be generators for these variables.

<u>Checking vs generating rules</u>

When we use "Not" in a rule we need not always make sure that it is preceded by positive conditions for its global variables. But, if we do not do this, we should make sure that the rule is only used for checking.

As an example, consider the rule:

```
childless(x) if Not(x is-a-parent-of y)
```

This is read:

x is childless if it is not the case that x is a parent of y for some y.

Because the global variable of the negative condition must have a value when the condition is evaluated this rule can only be correctly used for <u>checking</u> that someone is childless. It <u>cannot</u> be used for <u>finding</u> childless people. For generality of use we would need to add an extra condition:

```
childless(x) if person(x) & Not(x is-a-parent-of y)
```

Here person(x) is defined by the two rules:

```
person(x) if Male(x)
person(x) if Female(x)
```

This rule can be used both for checking and generating. When used for checking that someone is childless the condition person(x) is redundant. Thus, if we <u>only</u> use the childless condition as a checking condition, the shorter restricted use rule might be preferred. But to use rules that can only be used as checking rules is to live dangerously. micro-PROLOG does not check that the restriction is adhered to. If you make a mistake, and try to use the rule to generate, you will get incorrect answers.

The rule that has the person(x) condition also has another merit. It

4.1 Negative conditions

makes sure that only <u>people</u> are confirmed as childless. The shorter rule will confirm childless (6), because 6 is something for which no is-a-parent fact can be confirmed.

<u>Not with belongs-to</u>

We can use a negated condition to check that something is not on a list. As an example, the query:

```
Which(x x belongs-to (a cow jumped over the moon) &
                          Not(x belongs-to (a the))
```

will give us all the words in the list (a cow jumped over the moon) which are not one of the articles (a the).

The query:

```
Which(Z  Z Is-All(x x belongs-to (P A L I N D R O M E)
                          & Not(x belongs-to (A E I O U))
```

gives the answer

```
(P L N D R M)
```

which is a list of all the non-vowels in the letters of PALINDROME.

<u>Exercise</u> 4-1

1. Using the built-in Arithmetic relations of micro-PROLOG,
   a. Give a definition of an even number using the PROD relation.
   b. Give a definition of an odd number that makes use of the even number definition. Hint: use the built-in relation "NUM(x)" which tests if x is a number.
   Notice that your programs can only be used for testing the relations they define.

2. Answer the following PROLOG questions:
   a. Which(x x belongs-to (the quick brown fox) and
      Not(x belongs-to (how now brown cow))
   b. Which(x x Is-All(y y belongs-to (F R E D) and
      Not(y belongs-to (D O R I S)))

3. Using the Tudor Royal family database,
   a. Define the relation "a-man-with-no-sons".
   b. Define the relation "a-mother-with-no-daughters".

4. Using the information described in the books database, we will develop a library loan system. Our records of book issues will have the form:

   Issue (Name Title Author Issue-Date Due-Date)

   for instance, the sentence:

   Issue ((John Smith)(Oliver Twist)(Charles Dickens)(4 6 80)(18 6 80))

   says that John Smith borrowed Oliver Twist by C. Dickens, he borrowed it on 4th June 1980, and is supposed to return it by the 18th. Our records of book returns will have the form:

   Return (Name Title Author Return-Date)

   for instance:

Return ((John Smith)(Oliver Twist)(Charles Dickens)(12 6 80))

says that J. Smith returned his book on the 12th of June (before it became overdue)

a. Add this definition to your program:
"A book (title) is overdue if it has been issued, it has not been returned, and the date is after the Due-Date".
b. Give the definition of "after" that you will use.
c. Add this definition to your program:
"Anybody who has an overdue book is banned from the library".

4.2 The Is-All Condition

The Is-All condition is another form of simple sentence. It has the form:

L Is-All (A Q)

The pair (A Q) are an answer-pattern and a query-pattern as in a Which query; L is a variable or a list pattern. The condition is read:

L is a list of all the A's such that Q for some y1,...,yk

Here, y1,...,yk are the local variables of Q, the variables that do not appear in A or in any other simple sentence of the query or rule in which the "Is-All" appears. The global variables of (A Q) are those that do appear in some other simple sentence.

Restrictions on use

As with negative conditions, when the "Is-All" condition is evaluated all the global variables of (A Q) must have values. So in a query we must precede an "Is-All" condition with positive generators for its global variables, and in a rule we must have preceding generators or make sure the rule will only be used to answer queries in which the global variables are given. micro-PROLOG does not check that the global variables have values when it evaluates the "Is-All" condition. It is likely to give incorrect answers in this situation.

Usually, the L argument of the "Is-All" condition will be a variable. The evaluation of the condition then generates a single value for the variable which is the list of all the answers to the query (A Q).

In general, it is not safe to give L as a particular list and use the "Is-All" in a checking mode. This is because the condition only holds when L is identical to the list of answers that would be constructed in the generate use of the condition. Thus, the query:

Does((Tom Dick Peter) Is-All(y Mary is-the-mother-of y))

may fail to be confirmed even though Tom, Dick and Peter are the only answers to the query:

Which(y Mary is-the-mother-of y).

This happens if the evaluation of this query would generate the answers in a different order from that of the list (Tom Dick Peter). In section 4.3 we shall see how we can get around this problem using a relation that checks that two lists have the same elements.

This restriction of the "Is-All" condition is due to the fact that micro-PROLOG knows nothing about sets. It only knows about lists, and

lists are identical if they comprise the same sequence of elements. When we use lists to represent sets, we must do our own testing for equality, and removal of duplicate elements. (The problem of removal of duplicates is dealt with in exercise 5-1(12) of the next chapter).

If the list L is empty, or only contains one element, this problem of exact ordering of the elements does not arise. So "Is-All" can be safely used to check that there are no answers or that some individual is the only answer.

Does(() Is-All(x Tom is-the-father-of x))

checks that Tom has no children. It is equivalent to the query

Does(Not(Tom-is-the-father-of x)).

The query

Does((Bill) Is-All(x Tom is-the-father-of x)

checks that Bill is the only child of Tom.

Finally, the list L can be given as a list of variables.

The query:

Which((x1 x2 x3) (x1 x2 x3) Is-All(y Mary is-the-mother-of y))

checks that there are only three children of Mary, and if there are, gives us their names. The query

Which(x 3 length-of x & x Is-All(y Mary is-the-mother-of y))

is equivalent, and will have the same answer. It uses the relation "length-of" that we discussed and defined in Chapter 3.

Procedural reading

The way an Is-All condition is evaluated is reflected in the alternative procedural reading:

To answer the query L Is-All (A Q)
answer the query Which(A Q)
and check that L is the list of answers in the order they are found.

Notice that this means that any duplicate answers to Which(A Q) appear as duplicates on the list L.

Use of Is-All for constructing lists

The rule:

X intersection-of (Y Z) if X Is-All (x x belongs-to Y & x belongs-to Z)

defines the relation that is satisfied when x is a list of all the individuals that appear on the lists y and z. Because of the restrictions on the use of "Is-All" it can only be used for constructing such an intersection list. Notice that if Y or Z contains a duplicate of a common member this duplication will be repeated on the list X. But X will be without duplicates if Y and Z are without duplicates.

The rule:

X difference-between (Y Z) if X Is-All (y y belongs-to Y &

Not(y belongs-to Z))

defines the relation that holds when X is the list of elements on Y that are not on Z. It can only be used for finding X given Y and Z. The constructed list X will be without duplicates if Y is without duplicates.

Exercise 4-2

1. Using the relation x member-of-either (y z) defined by the two rules:

```
x member-of-either (y z) if x belongs-to y
x member-of-either (y z) if x belongs-to z
```

give a rule for the relation "x union-of (y z)" that can be used for constructing a list x of all the individuals that are members of y or z.

2. Define the "subset-of" relation: x subset-of y holds when all the elements of x also belong to y. (Hint: the difference between x and the intersection of x and y is the empty set.) We will revisit this example later.

3. Define the relation: X set-union-of (Y Z) which is the same as "union-of" except that its use will always give a list X without duplicates if Y and Z are without duplicates. Define it in terms of the "union-of", intersection-of" and "difference-between".

Sometimes we want to check that the answers to a query all satisfy some condition. In the next section we will show how this can be tested directly with a single "For-All" condition. As an exercise in the use of Is-All we show how it can be done using the answer list constructor.

Suppose that we have used the relations over individuals representation of family relations, that we have a set of facts such as

```
Bill  is-the-father-of Roy
Sarah is-the-mother-of Roy
```

giving the mother/father relations. Consider the problem of finding all the men who only have sons.

We can actually pose this query using negation. We can express it:

```
Which(x Male(x) & Not(x is-the-father-of y & Female(y)))          (A)
```

read as:

the x's such that x is Male and it is not the case that
x is the father of a female y, for some y.

We can also express the condition using "Is-All". A male x satisfies the condition if all the answers to the query Which(y x is-the-father-of y) are male. By wrapping up these answers as a list, we can check the condition using the "all-Male" relation defined by:

```
all-Male(())
all-Male((u|x)) if Male(u) & all-Male(x)
```

This is the relation that holds for a list iff it is a list of males. The query can be posed:

```
Which(x Male(x) & Z Is-All(y x is-the-father-of y) & all-Male(Z))   (B)
```


Notice that this query, and query (A) above, are both satisfied by men who have no children at all. This is a correct and strict interpretation of the condition "only have sons". If we wanted to insist that each man had at least one child we could replace the "Male(x)" condition of both query (A) and query (B) by the condition "is-a-father(x)". This is defined by the single rule:

```
is-father(x) if x is-the-father-of y.
```

(A) and (B) are equivalent ways of expressing the same query. There is a third way:

```
Which(x Male(x) & (Male(y)) For-All(y x is-the-father-of y))       (C)
```

This uses the "For-All" condition we are about to describe. It has the effect of testing that all the children of x are male without the need to construct the list of these children. In this respect it is similar to query (A). Notice that in (A), (B) and (C) the global variable x of the complex condition of the each query has a preceding generator, Male(x).

4.3 The For-All condition

A "For-All" condition is a simple sentence of the form:

(C) For-All (A Q)

C is a simple sentence or a conjunction of simple sentences. The (A Q) is a which query expression in which A is the list of variables that appear in "C".

Its declarative reading is:

C is true for all the A's such that Q for some y1,...,yk

The y1,...,yk are the local variables of Q.

The global variable restriction applies. All global variables of Q must be bound before the condition is evaluated, but micro-PROLOG does not check that this constraint is satisfied. If it is not satisfied, you are likely to get the wrong answers to the query or rule in which the condition appears. The moral is, precede it with positive generators for the global variables, or make sure the rule is only used to check a condition in which the variables will be given.

The procedural reading is:

to check the condition (C) For-All (A Q)
  answer the query Which(A Q),
  as each answer A is generated check that C holds,
  if C does not hold for some answer conclude that the "For-All" condition does not hold and abandon the search for answers,
  if C holds for every answer A conclude that the "For-All" condition holds.

Example uses of For-All

(1) The rule:

```
X subset-of Y if (x belongs-to Y) For-All (x x belongs-to X)
```

can be used to check that all the members of a list Y are members of X. The rule:

X same-elements-as Y if X subset-of Y & Y subset-of X

can be used to check that all the members of X are members of Y and vice-versa.

Notice that this defines a set equality with sets represented by lists of their elements. It can also be used to check if some list is just a permutation of the elements of another list of the same length. The relation can be used in conjunction with "Is-All" to check whether some particular set, represented as a list, is the set of answers to some query.

As an example, suppose that we wanted to check that Mary's children were Tom, Dick and Peter. Assuming that information is represented as in the Tudor's data base, we would pose the query:

```
Does(x Is-All(y Mary is-the-mother-of y) &
               x same-elements-as (Tom Dick Peter)).
```

This is the way to get around the restriction on the "Is-All" that we discussed above.

(2) An ordered list is a list such that for all pairs of adjacent elements (x y) the condition x lesseq y holds. This gives us the rule:

```
ordered(X) if (x lesseq y) For-All ((x y) (x y) adjacent-on X)
```

This specification-like rule can be used for checking the ordered condition. The relation "(x y) adjacent-on X" which holds when (x y) are a pair of adjacent elements on a list X can be defined by:

```
(x y) adjacent-on (x y | X)
(x y) adjacent-on (z | X) if (x y) adjacent-on X
```

The definition of the relation was the answer to exercise 3-7(5). The relation lesseq was defined in exercise 2-3(3).

Exercise 4-3

1. Using the relations of the books data base, i.e. "writer", "written-by", "type", "published", define the following relations. Use "For-All".
   (i) Novelist(x): x is a writer whose recorded books are all novels.
   (ii) Modern-author(x): x is a writer whose recorded books are all published in the twentieth century.

2. Use For-All to define:
   (i) Positive-nums(x): x is a list of numbers greater than 0.
   (ii) all-Male(x): x is a list of names of males.

3. Define the relation disjoint(X Y): X and Y are lists with no common element. Define it using:
   (i) Not
   (ii) Is-All
   (iii) For-All
   Any if these programs can be used for testing the relation.

5. List Processing

We have seen that we can access the components of lists and construct new lists out of existing lists by defining relations with lists as arguments. When we query these relations we are processing lists. In this chapter we look at some more list processing relations and their use. We also illustrate the application of list processing to the parsing of sentences expressed as lists of words, an application to which PROLOG is well suited.

5.1 The appends-to relation

We begin by examining a very powerful little list program for the relation "appends-to". This has many uses apart from the 'normal' one of concatenating two lists together; in particular it can be used to find all the ways of splitting a list, to remove an initial or tail segment of a list, even to split a list on a given element.

The condition

```
(x y) appends-to z
```

holds when z is the result of concatenating the list x to the list y.

An example of this is:

```
((A B) (C D E)) appends-to (A B C D E)
```

Before defining it, let us consider an example to illustrate its use. I am trying to remember what I ate for lunch today. It was served in two courses. Each course can be described by a list of its ingredients. Thus

```
(fish chips) served-in first-course
(rhubarb custard) served-in second-course
```

What I ate altogether was the list of things I ate in the first course appended to the list of things I ate in the second course. So

```
Z served-in dinner if x served-in first-course
                    & x served-in second-course
                    & (x y) appends-to Z
```

```
Which(x x served in dinner)
Answer is (fish chips rhubarb custard)
No (more) answers
```

Notice the difference between this answer, which is one list and the answer to:

```
Which(x y) x served-in first-course & y served-in second-course)
Answer is ((fish chip) (rhubarb custard))
No(more) answers
```

The answer to this is a pair of lists. The two lists are not 'glued' together in a single list. This is the rôle of appends-to.

To develop our program for "(x y) appends-to z" we must make statements about the relation that together completely define the relation.

As a rule of thumb, when defining relations over lists, we should pick one of the arguments of the relation and have sentences for different cases for that argument. The cases should together cover all the different types of lists that might appear in that argument of the relation.

For the "(x y) appends-to z" relation, let us pick the first argument x. We will completely define the relation by having a sentence about all instances of the relation when x is the empty list (), and another sentence about all instances of the relation when x is a non-empty list represented by the pattern (x|X).

When x is (), it is always the case that y and z are the same. This is expressed by the unconditional rule

```
(() y) appends-to y                                        (1)
```

which we read as,

for all y, the empty list () and y append to y.

Notice that we do not have to have an explicit condition that says that y and z are the same. We express this implicitly by having the same variable in each argument position.

When x is a non-empty list of the form (x|X) we know that z must also begin with x. So z must be of the form (x|Z) for some Z. We cannot unconditionally state

```
((x|X) y) appends-to (x|Z)
```

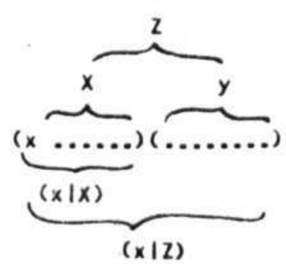
because this does not hold for all X, y and Z. The X, y and Z cannot be arbitrary lists. However, if they are such that

```
(X y) appends-to z
```

then we can be sure that

```
((x|X) y) appends-to (x|z).
```

This is illustrated by the picture:



This gives us the conditional rule

```
((x|X) y) appends-to (x|Z) if (X y) appends-to Z          (2)
```

(1) and (2) are a pair of sentences that together completely define the appends-to relation. They are a logic program for the relation.

__Using appends-to split a list__

Queries to the "appends-to" relation in which x and y are given give back a z that is the concatenation of x and y. To use it to split a list, we give the z and leave x and y as variables.

```
Which((x y)(x y) appends-to (2 3 4))
Answer is (()(2 3 4))
Answer is ((2)(3 4))
Answer is ((2 3)(4))
Answer is ((2 3 4)())
No (more) answers
```


In the answers that we got, two were a pair consisting of the empty list and the original list. To exclude these answers we simply replace x and y by patterns that denote different non empty lists.

```
Which(((x|X) (y|Y)) ((x|X) (y|Y)) appends-to (2 3 4))
Answer is ((2) (3 4))
Answer is ((2 3) (4))
No (more) answers
```

By describing the second list with the pattern (x|Y) we can insist that the split is at a point where the first element of the list recurs.

```
Which(((x|X) (x|Y)) ((x|X) (x|Y)) appends-to (2 4 2 5 1 2 3))
Answer is ((2 4) (2 5 1 2 2))
Answer is ((2 4 2 5 1) (2 3))
No (more) answers
```

Alternatively, we can insist that the second list begins with some particular element, say 3. We do this by denoting it by the pattern (3|Y).

```
Which((x (3|Y)) (x (3|Y)) appends-to (2 3 5 3 1))
Answer is ((2) (3 5 3 1))
Answer is ((2 3 5) (3 1))
No (more) answers
```

This finds all the splittings of the list that start at the given number 3.

As a last example of the use of "appends-to", consider the query

```
Which(x (y (x|X)) appends-to (2 3 4))
```

What will the answers be?

__Exercise__ 5-1

Answer these PROLOG questions:

1. Which(x ((J U M) (B O)) appends-to x)
2. Which((x y) (x y) appends-to (J O H N))
3. Which((x y) (x (R|y)) appends-to (C Y R I L))
4. Which((x y) ((D A M)(S O N)) appends-to x and x has-length y)
5. Try the query
   One((x y z) (x y) appends-to z)).
   Hand evaluate it to the point where you get 4 different answers if you have not got a computer.
6. Give the query that checks that the list (2 3 4 2 3 4) is the result of appending some list to itself and which returns that list.
7. Give the query that returns the second list of all the splittings of the list of words
   (the man closed the door of the house)
   that start with the word "the".
8. Use the "belongs-to" relation to pose a compound query that finds all the second halves of the splittings of
   (Sam threw a ball into the lake)
   that start with one of the words in the list (a the).
9. Using "appends-to" pose the query to find the last element of the list (2 3 4).
10. Consider the relation
    Split-on(y X X1 X2): X1 X2 is a splitting of the list X
    such that X1 is of length y.

(a) Define it using "appends-to" and "has-length".
(b) Give an alternative recursive definition.
Which is the more efficient micro-PROLOG program?

11. Give a recursive definition of the relation
remove-all(x X Y): Y is the list X with all occurrences of x removed.
Hint: treat the three cases
(i) x the empty list
(ii) x a non-empty list that begins with x
(iii) x a non-empty list that begins with a y different from x.

12. Give a recursive definition of the relation
X compacts-to Y: Y is the list X with all but the first occurrence of any duplicated elements of X removed.
Define it using the "remove-all" relation of exercise 11.
Hint: if X is a non-empty list beginning with x then Y must also begin with x but the tail of Y will be a compacted version of the tail of X after all recurrences of x have been removed. Now say this in micro-PROLOG using list patterns and a conditional rule. Don't forget the case when X is empty.
Notice that this relation can be used for removing duplicates from a list of answers given by an Is-All condition. We use a compound query condition of the form:

```
X Is-All(A Q) & X compacts-to Y
```

But note that "compacts-to" is a time consuming operation.

5.2 Rules that use appends-to

(1) The rule:

```
front(x y z) if (y y1) appends-to z & y has-length x
```

defines the relation front(x y z) which holds when y comprises the first x elements of z. It can be used for finding the first x elements of a list as in:

```
Which(x front(3 x (A B C D E F))
Answer is (A B C)
No(more) answers
```

In answering this query the condition "(y y1) appends-to z" of the rule is used to generate candidate splittings of the list (A B C D E F). micro-PROLOG will test every splitting with the "y has-length x" condition.
Notice that we can also define the relation using length-of:

```
front(x y z) if z length-of y & (y y1) appends-to z
```

Used to answer the same query, the condition "z length-of y" will be used to construct a list of three variables (x1 x2 x3) as the value of y that is passed on to "(y y1) appends-to z". The evaluation of the condition then finds values for x1, x2 and x3. In other words, after it has evaluated the first condition of the rule, the query

```
Which(x front(3 x (A B C D E F))
```

is reduced to the evaluation of

```
Which((x1 x2 x3) ((x1 x2 x3) y1) appends-to (A B C D E F))
```

Note the powerful use of answers that are list patterns, and that the evaluation of the query involves generation of different splittings of the given list. It is much more efficient than the evaluation that uses the first definition of "front". The one drawback of the second definition is

that it can only be used if the length of the front list is given. This is because of the restriction on the use of "length-of" that we noted in Chapter 3.

(2) The rules:

```
(x|X) initial-segment-of z if ((x|X) y) appends-to z
(y|Y) back-segment-of z    if (x (y|Y)) appends-to z
```

define the relations suggested by the relation names. Notice the requirement that the initial and back segments be non-empty lists.
We can use these relations to define the relation x segment-of z which holds when x is a non-empty segment of contiguous elements on the list z. Such a list x is an initial segment of a back segment of z.

```
x segment-of z if y back-segment-of z & x initial-segment-of y

Which(x x segment-of (A B C))
Answer is (A)
Answer is (A B)
Answer is (A B C)
Answer is (B)
Answer is (B C)
Answer is (C)
No(more) answers
```

(3) The rules:

```
(x) reverse-of (x)
z reverse-of (x|X) if Y reverse-of X & (Y (x)) appends-to z
```

define the relation z reverse-of x that holds when z is the list x in reverse order. They can be used for checking the relation or finding the reverse of a list with a query in which the second argument is given and the first is to be found.

```
Which(z z reverse-of (A B C D))
Answer is (D C B A)
No(more) answers
```

Why should it not be used with the first argument given and the second to be found? Follow through the evaluation to see what happens in this case.

(4) The rule:

```
delete(x X Y) if (X1 (x|X2)) appends-to X &
                 (X1 X2) appends-to Y
```

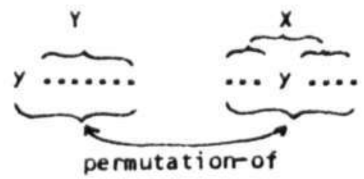
defines the relation which holds when Y is the list X with some single occurrence of x removed.
We can use this relation to give a recursive definition of the relation

```
Y permutation-of X: Y is some reordering of the list X
```

It is defined by the pair of rules:

```
() permutation-of ()
(y|Y) permutation-of X if delete(y X Z)
                        & Y permutation-of Z
```

The second rule tells us that the list (y|Y) is a permutation of the list (x|X) if the first element y appears somewhere on (x|X) and the remainder Y is a permutation of the remainder of (x|X) when y is removed. This diagram illustrates this relationship between (y|Y) and X.



Remember that in Chapter 4 we defined the relation X same-elements-as Y which was true of a pair of lists if every element of X appeared on Y and vice versa. This is equivalent to Y permutation-of X when X and Y have the same length. However, because "same-elements-as" was indirectly defined using "For-All" it can only be used for testing. Our recursive definition of Y permutation-of X can be used for testing or generating. To generate all the permutations of a list we give X and ask for Y.

```
Which(Y Y permutation-of (5 3 7))
Answer is (5 3 7)
Answer is (5 7 3)
Answer is (3 5 7)
Answer is (3 7 5)
Answer is (7 5 3)
Answer is (7 3 5)
No (more) answers
```

To find an ordered permutation we pose the query:

```
One(Y Y permutation-of (5 3 7) & ordered (Y))
Answer is (3 5 7).F
```

Here, "ordered" is the relation defined using "For-All" in Chapter 4.
Finally, we can give a definition of the sort relation

x sorts-to y: y is a sorted version of the list x

It is:

```
x sorts-to y if y permutation-of x & ordered(x)
```

This can be used, somewhat inefficiently, to sort a list with a query condition in which x is given and y is to be found. It sorts the x by generating successive permutations until one is found that is ordered. In the next section we shall give an alternative recursive definition of the sort relation which is a much more efficient PROLOG program.

<u>Exercise 5-2</u>

1. Using the relations defined above, answer:
   a. Which(x front(4 x (J K L M N P Q))
   b. Which(x x segment-of (F R E D A))
   c. Which(x x reverse-of (E R I C))

2. Define the relation "last-of" of exercise 3.7(4) in terms of "appends-to". Notice that this is a non-recursive definition of "last-of" in terms of the recursively defined "appends-to".

3. Define the list membership relation "belongs-to" in terms of "appends-to".

4. The 'power list' of a list is directly analogous to the power set concept in set theory: i.e. the power-list of a list is the list of all sub-lists of the list. Define the relation "x power-list y" which holds when y is the power list of x.
   Try your program on the following query:

   Which(x (A B C D)power-list x)

   (Hint: remember that the empty list is also a sublist, but only once. Don't forget about "Is-All".)

5. Define the relation: palindrome(x) which holds when x is a list that reads the same forwards or backwards. Thus, (M A D A M) is a palindrome list of letters, (1 2 2 1) is a palindrome list of numbers. Define it in terms of "reverse-of". Use your definition to test the above two palindromes.

6. Define the relation "adjacent-on" of exercise 3.7(5) but this time give a non-recursive definition by using "appends-to".

7. Give an alternative recursive definition of the relation delete(x X Y) which was defined above using "appends-to". Hint: treat the two cases:
   (i) the deleted x is the first element of X.
   (ii) the deleted x is not the first element of X.

5.3 <u>Recursive definition of the sort relation</u>

Next, we develop a recursive description of the sort relation between lists that will provide us with a much more efficient sort program than the one defined above using "permutation-of". We start by making one or two simple observations about the relation.

First we know that a singleton list is already sorted, i.e. a list with one element in it is already in the right order. Similarly the empty list is sorted by default. These two facts about the sort relation are expressed by:

```
() sort-is ()          (1)
(x) sort-is (x)        (2)
```

However, most lists are neither empty, nor singleton; so we have to be able to sort these too. One way of dealing with bigger lists is to make them small ones; i.e. use some kind of divide and conquer strategy. This would involve splitting the list (which has at least two elements) into two smaller ones, sorting each of the bits and putting them back together again. This means that we must look for a recursive description of the "sort-is" relation for lists of at least two elements.

<u>Merge sort</u>

The method of splitting that we shall use merely involves dividing the list into two nearly equal halves: i.e. they are within one element of each other in length. We can do this by taking a front segment and a back segment such that when appended together again they make up the original list; making sure at the same time that the lengths are nearly equal.

Let us call this relation split. Thus, split((x1 x2|x) X1 X2) holds when X1 and X2 appends-to (x1 x2|x) and the length of X1 is the length of X2, plus or minus 1.

Now, if X1, X2 are in the split relation to (x1 x2|x), and y1, y2 are sorted versions of X1, X2 respectively, then the sort of (x1 x2|x) is some y which is an order preserving interleaving of y1 and y2. Let us call this relation between y1, y2 and y, merge(y1 y2 y). The following rule gives us a recursive description of the "sort-is" relation that corresponds to this method of sorting:

```
(x1 x2|x) sort-is y if split((x1 x2|x) X1 X2)                    (3)
          & X1 sort-is y1 & X2 sort-is y2 & merge(y1 y2 y)
```

Rule (3) fairly naturally encodes the English statement of sorting using the divide and conquer method. The merge program we shall look at in a moment is clearly the 'guts' of the sort program, it has to be able to take two ordered lists, and merge them into one. This job is easier than sorting a list since we can make use of the knowledge that the two 'input' lists are already ordered.

In defining the "merge" relation we shall need to treat several cases. The first two are when either y1 or y2 is the empty list:

```
merge(() x x)                                                    (4)
merge(x () x)                                                    (5)
```

The remaining case is where both y1 and y2 are non-empty. In this case we have three possibilities: either the first element of each list is equal, the first element of y1 is less than the the first element of y2 or vice-versa.

Notice that it is here that we have to start discussing what it means for an elements of a list to be less than or greater than another element. Up until now we have not actually needed to define what criteria we use to sort lists. We shall just take the built in relation "LESS" to define this. This enables us to compare numbers or constants. It doesn't allow comparison between lists, or between objects of different type.

We could define our own notion of order amongst elements which might allow comparison amongst different types of individual, however for simplicity we shall stick with the LESS test.

Returning to the problem of merging two lists together, having decided that the first element of one is LESS than the first element of the other we put that element as the first element of the merged list. Assuming that we are supposed to be sorting into increasing order, the smaller of the two elements must form the first element of the merged list.

First the rule for when both the first elements of y1 and y2 are identical:

```
merge((x|y1) (x|y2) (x x|y)) if merge(y1 y2 y)                   (6)
```

This rule states that the merge of the two lists with identical first element starts with two of that element, and the tail is got by merging the tail of y1 and y2.

The next rule deals with the case when the first element of y1 is LESS than the first element of y2. In this case the first element of the merged list is the first element of y1. The tail of the merged list is found by merging the tail of y1 and the **whole** of y2:

```
merge((x1|y1) (x2|y2) (x1|y)) if x1 LESS x2                      (7)
                              & merge(y1 (x2|y2) y)
```

In a similar way we get the last rule for merge, which is symmetric to (7):

```
merge((x1|y1) (x2|y2) (x2|y)) if x2 LESS x1                      (8)
                              & merge((x1|y1) y2 y)
```

Finally, we need to define the split relation. We can say that

split(X X1 X2) holds if y1 is approximately half the length of (x1 x2|x) and X1 X2 are a splitting of X such that X1 has y1 elements. This gives us the rule:

```
split(X X1 X2) if X has-length x1 & PROD(2 y1 x1 y2)             (9)
               & split-on(y1 X X1 X2)
```

Here, "split-on" is the relation defined in exercise 5.1(10).

The complete merge-sort program is as follows:

```
() sort-is ()
(x) sort-is (x)
(x1 x2|x) sort-is y if split((x1 x2|x) X1 X2)
          & X1 sort-is y1 & X2 sort-is y2  & merge(y1 y2 y)

merge(() x x)
merge(x () x)
merge((x|y1) (x|y2) (x x|y)) if  merge(y1 y2 y)
merge((x1|y1) (x2|y2) (x1|y)) if x1 LESS x2
                              & merge(y1 (x2|y2) y)
merge((x1|y1) (x2|y2) (x2|y)) if x2 LESS x1
                              & merge((x1|y1) y2 y)

split(X X1 X2) if X has-length x1 & PROD(2 y1 x1 y2)
               & split-on(y1 X X1 X2)

split-on(0 X () X)
split-on(y (x|X) (x|X1) X2) if 0 LESS y & SUM(1 y1 y)
                            & split-on(y1 X X1 X2)
```

And, just to make sure it works, let us try sorting a list:

```
One(x (4 3 6 100 -5 3) sort-is x)
Answer is (-5 3 3 4 6 100).F
```

Quick sort

The same basic strategy for divide and conquer can lead to completely different sort programs if we choose slightly different methods of 'dividing'. For example, in our split, we simply chopped the list into a front and a back half. If instead we had chosen to partition the list in such a way that all the elements of one list were LESS than all the elements in the other we get a quite different recursive description of the sort relation.

The first thing to notice about this scheme for splitting is that when we are merging the two lists back together again we can take advantage of the fact that one list is entirely LESS than the other. In other words each element of one partitioned list (and hence its sorted variety) is LESS than **all** the elements of the other list. This enables us to replace the "merge" part of the sort-is program by a simple "appends-to".

On the other hand the partitioning of the lists is more complicated; it has to do the main work of the sort.

Exercise 5-3

1. Assume that you have some suitable definition of the relation partition(x y z1 z2): y is the first element of list x, each element of x which is LESS than y appears on the list z1, all the other elements of x appear on z2. Give a definition of the sort relation that makes use of "partition". Call the relation "quick-sort".

2. Give the rules for "partition", and verify that your quick sort program gives the same results as the merge sort program. How do they compare for speed?

3. Inefficiency in the merge sort program results from the need to continually recompute the length of a list on each recursive call. This is not necessary since the split relation effectively finds the lengths of the lists X1 and X2 that are recursively sorted. Change the definition of the sort relation so that it is a relation between a pair (x X) and a list Y where Y is the sorted version of X and x is the length of X. You will need to change the recursive rule for "sort-is" and the rule that defines "split". Call the new sort relation "merge-sort", and the new split relation, "merge-split". Don't forget the base cases for "merge-sort". Compare the speed of this program with that for "sort-is" and "quick-sort".

5.4 Parsing sentences expressed as lists of words

One of the more impressive application areas that PROLOG has been used in is the field of natural language understanding. So let us look at a very simple example of this, by developing a PROLOG program which can parse very simple sentences of English while also illustrating some more list processing using "appends-to".

The most fundamental idea behind our program is that we represent an English sentence as a list of words. The various ways that this list of words can be broken up represent the various possible 'parsings' of the sentence. For example the sentence "the boy kicked the ball" is represented by the list:

```
(the boy kicked the ball)
```

By splitting this list up into sub-lists we can see some of the grammatical structure of the sentence:

```
((the boy) (kicked (the ball)))
```

By augmenting the list with labels which describe the various parts of speech:

```
(SENTENCE (NOUN-PHRASE (DETERMINER the) (NOUN boy))
          (VERB-PHRASE (VERB kicked)
                       (NOUN-PHRASE (DETERMINER the) (NOUN ball))))
```

This structure represents the equivalent of the grammatical structure of our sentence, except of course that it is highly simplified: there is no tense to the verb, and there is no representation of plurality in the noun phrases. Still, this kind of grammar is a suitable base for further development.

The program which recognises sentences like this is composed of rules and facts which are organised around the parts of speech found in sentences. For example the rule for "is-sentence" can recognise a sentence, and the rules for "is-noun-phrase" recognise noun phrase. The most simple rule for recognizing sentences is:

```
x is-sentence (S X Y) if
    (x1 x2) appends-to x
    x1 is-noun-phrase X and
    x2 is-verb-phrase Y
```

In other words, if we can split the list of words x into two sub-lists x1 and x2 which form a noun phrase and verb phrase respectively then "x" is

a sentence. The grammatical structure of the sentence is represented by the structure: (S X Y) where "X" and "Y" are the grammatical structures of the noun phrase and verb phrase respectively. (For the sake of brevity we use abbreviations such as "S" to stand for SENTENCE)

Notice that we are using the "appends-to" program to split the sentence into its constituent components. Our simple rule for sentences can only recognise very simple sentences: for example this sentence would not be recognised! One definition of a noun phrase is a determiner followed by a noun, i.e. a word like "the" or "a" followed by a word like "boy":

```
x is-noun-phrase (NP X Y) if
    (x1 x2) appends-to x and
    x1 is-determiner X and
    x2 is-noun-expression Y
```

NP stands for Noun phrase. The program for "is-determiner" only recognises determiners, i.e. it only recognises a list which contains just one word, which is one of the known determiners:

```
(x) is-determiner (DT X) if
    x dictionary DET
```

The program for dictionary represents the vocabulary of the system it says what the type of each word is. Only those words which are in the dictionary are known to the program, if we try to parse a sentence with a word not in the dictionary it will simply fail. The part of the dictionary program concerned with determiners is:

```
the     dictionary    DET
a       dictionary    DET
an      dictionary    DET
```

The simplest kind of noun expression is just a noun. This is expressed by:

```
(x) is-noun-expression (N x) if x dictionary NOUN
```

i.e. a singleton list is a noun expression if the vocabulary has that word down as a noun, and the nouns we know about are:

```
boy     dictionary    NOUN
ball    dictionary    NOUN
girl    dictionary    NOUN
apple   dictionary    NOUN
   etc.
```

Going back to our rule for sentences we have yet to describe what a verb phrase is. A very simple kind of verb phrase is a verb expression followed by a noun phrase, this being the object of the sentence. This rule is expressed by:

```
x is-verb-phrase (VP X Y) if
    (x1 x2) appends-to x and
    x1 is-verb-expression X and
    x2 is-noun-phrase Y
```

By ignoring problems regarding tense we can get a rule for verb expressions which is similar to our noun expressions rule. The simplest form of verb expression is a verb.

```
(x) is-verb-expression (V x) if
     x dictionary VERB
```

and we extend our knowledge of the dictionary with

```
kicked   dictionary   VERB
likes    dictionary   VERB
etc.
```

This more or less completes our first approximation to English syntax. We can now parse some very simple sentences:

```
Which(x (the boy kicked the ball) is-sentence x)
Answer is (S (NP (DT the) (N boy))
          (VP (V kicked) (NP (DT the) (N boy)))))
```

This parse gives only the grammatical structure of the sentence, there is no sense in which the program can be said to understand the sentence. Still, the grammatical structure is probably a lot easier for a semantic analysis program to deal with. An example of what such an analysis might result in could be:

> There is an x, y and z such that x is a (unique) boy and y is a (unique) ball and at a (unique) time z an event occurred. The action associated with z is "kick", the agent X and the object is y.

This is an English description of the meaning of the sentence, however it is beyond the scope of this primer to see how this is arrived at or used. We shall content ourselves with developing our program so that it recognises a slightly richer set of sentences.

One simple extension would be to add adjectival phrases. The adjectival phrase is simply an extension of the noun expression which instead of being just a noun can now also be an adjective followed by a noun expression. Some example noun expressions involving adjectives are:

```
silly boy
sad girl
big fat bouncy ball
etc.
```

This rule must be added to the program for "is-noun-expression":

```
x is-noun-expression (NE X Y) if
     (x1 x2) appends-to x and
     x1 is-adjective X and
     x2 is-noun-expression Y
```

This recursive description allows an arbitrary number of adjectives to precede the noun, and the parse structure returned reflects the adjectives used. Of course we now need to extend the dictionary to include some adjectives:

```
(x) is-adjective (A x) if x dictionary ADJ

     big      dictionary   ADJ
     silly    dictionary   ADJ
     fat      dictionary   ADJ
       etc.
```

We can now parse sentences such as:


```
Which(x (the sad boy likes the bouncy ball) is-sentence x)
Answer is (S (NP (DT the (NE (A sad) (N boy)))
          (VP (V likes)
              (NP (DT the) (NE (A bouncy)
                               (N (ball))))))
```

We can also use the program, somewhat inefficiently, to find all sentences of a given length. A query such as

```
Which((x1 x2 x3 x4 x5 x6) (x1 x2 x3 x4 x5 x6) is-sentence x)
```

will give us all the six word sentences recognised by the program. If you have been following the development of the program on a computer try the query.

We can be more precise. We can insist that x1 is "the" and x5 is "a" with the query:

```
Which((the x2 x3 x4 a x6) (the x2 x3 x4 a x6) is-sentence X)
```

Finally, it can be used, very inefficiently to generate a sentence from a parse structure. The query:

```
One(x (the boy kicked the girl) is-sentence X and
                    x is-sentence X)
```

will parse the given sentence and then convert the parse structure back to the same sentence. Try it! The inefficiency results from the fact that the "appends-to" condition of each grammar rule should appear as the last condition of the rule for the sentence generate use. Placed as it is, it will generate larger and larger lists of variables until one is generated that is long enough to hold the sentence whose parse structure is given. (Remember exercise 5.1(5).)

Exercise 5-4

1. Find the parses of the following sentences (possibly involving an extension of the vocabulary):

   a. the sad boy likes a happy girl
   b. the ball kicked the boy
   c. a lonely man wandered the hills
   d. a piper plays a tune

2. Extend the above program so that it can cope with verb expressions that are verbs preceded by a conjunction of adverbs. The new program should cope with sentences such as:

   a man **slowly and deliberately climbed** the mountain

The extension required is analogous to that which copes with adjectives. Just add a new rule for "is-verb-expression" and give rules and dictionary entries describing adverbs. Use your new grammar to parse the above sentence.
Hint: you could treat an adverb preceded by "and" as an adverb.

6. Imperative primitives of micro-PROLOG

In the preceding chapters we have seen that programs and queries have a dual reading: the descriptive or logical reading and the behavioural reading. We have also seen that some answers which are possible might never be generated. For instance the SUM built-in relation can only be used if at most one of its arguments is a variable, the logically possible query "Which((x y) SUM(x y 10))" is not answered by micro-PROLOG. This restriction is due to the fact that the program for the SUM relation is not written in PROLOG, it is written in machine code and makes use of the arithmetic operations of the machine. Such programs can only handle deterministic calls, query conditions for which there is only one answer binding.

SUM, PROD and LESS are built-in relations which have a logical interpretation even though they are defined by an entirely behavioural program. When we use these relations in a PROLOG program only the logical interpretation is relevant for this gives the complete story of the effect of the machine code program. This is not true of all the built-in relations of micro-PROLOG. There are some relations, defined by machine code programs, for which the logical interpretation does not fully characterise the effect of the program. When we use these relations their logical interpretation is mostly irrelevant. They are used mainly for their non-logical effect. These are the imperative relations of micro-PROLOG.

The imperative relations slightly spoil the logical purity of micro-PROLOG. This is because rules and queries that use them must be read behaviourally to understand their purpose. However, the use of the imperatives can often be isolated. We can use them in PROLOG programs that must be understood behaviourally but which nonetheless define relations that have a completely logical interpretation. These PROLOG programs are analogous to the machine code programs that implement SUM, PROD and LESS. This isolation of the imperatives enables us to give the rest of the PROLOG program a completely declarative reading. It is good PROLOG programming style.

In this chapter we describe the main imperatives of micro-PROLOG and illustrate their use. We also show how their use can be packaged so as to extend the power of micro-PROLOG without destroying its important declarative nature.

6.1 Reading Input

The first imperative we look at is the read-term relation, R. The program for R reads in the next term typed in at the console and returns it as the binding of the variable given as the argument to R. The closest we can get to a logical reading of R is:

R(x) holds iff x is a term.

The behaviour reading of its machine code program is:

To find a fact of the form R(x)
check that x is a variable, read in a term t from the terminal,
return R(t) as the only fact.

The logical interpretation suggests that R can be used to check if something is a term, or to find a term. The behavioural reading tells us it can only be used to find a term and that this term is always the next one to be typed at the terminal. It is the non-logical, entirely behavioural aspect that is crucial to the use of R. We do not use it to generate an arbitrary term, we use it to read terms from the terminal.

An attempt to use R in a checking mode results in an error message.

A call R(YES) will result in a "Control Error" message, as will the call R(x) if x has already been bound to a non-variable. If we want to check that the next term to be typed is some particular term, we use an R(x) call followed by an equality test. Thus,

R(x) and x EQ YES

tests if the next term typed is the constant YES.

Example use

We can use R to delay the input to queries. In a typical query all the input parameters must be entered at 'query time'. A trivial example is in using the "has-length" program of Section 4.4, a typical query being:

```
Which (x (1 2 3 a b c) has-length x)
```

To allow the list to be supplied at 'run-time' we use the query:

```
Which (x R(y) and y has-length x)
```

The attempt to satisfy the R(y) condition of the query causes the system to prompt us for input with the "." character. This is the same "." that's part of the "&." prompt we get at the top level. We then enter the list whose length is to be found:

```
.(1 2 3 a b c)
Answer is 6
No (more) answers
```

The list is bound to y, and the y has-length x condition computes its length.

The declarative reading of the query is:

Find the x's such that x is the length of y for some term y.

The fact that we only get one answer, and that this is the length of the list term typed in response to a prompt, can only be deduced from the behavioural reading of "R". We only get one answer because the "R" program only generates one output binding for any call.

The "R" program will read in any term. It may be a number, a constant, a list or a variable. Any variables read in are immediately converted into internal form: in particular the name of the variable is not remembered. This has its advantages and disadvantages, it is beyond the scope of this primer to go into them.

The rules about entering a sentence over several lines apply to entering a list that is to be read by a R(x) call. The list can be entered over several lines. The system displays a special prompt on each new line until the whole list has been read in. The prompt is a number which is the 'bracket count'. It is the number of right brackets to be entered to complete the list. This corresponds to the current 'depth' at which terms are being read in. This little device makes the problem of entering complex lists very much easier.

Finally, to enter constants which contain special characters we quote the constant with double quotes. Thus, if s is a string of any characters other than the quote sign itself, "s" is a constant. The string s can contain blanks. To include the quote sign, we must use a double quote. Thus, """" in a quoted string is the same as ". For more details on the syntax of terms we refer the reader to the micro-PROLOG Programmer's Reference Manual [McCabe 1981].

6.2 Writing Output

The read term relation is most often used in combination with the write term relation, P. This relation is unusual in that it can have any number of arguments, it is a multi-argument relation. An approximate declarative reading is:

P(t1 t2 ... tn) is true iff t1 .. tn are terms.

The behavioural reading is:

To 'confirm a fact' of the form P(t1 .. tn) display the terms t1 to tn on the console and confirm the fact.

Again, the crucial property is not that it trivially confirms that its arguments are terms but that it displays these terms on the console. It is used for its non-logical side-effect.

Example use

The P imperative can be used to display partial results earlier than would be the case with the querying mechanism. A trivial illustration of this is with the "has-length" program, we can display the length before the query evaluator finishes!

```
&.Which(x (1 2 3 a b c) has-length x and P(x))
6Answer is 6
No (More) answers
```

The declarative reading of the query is:

the x's such that x is the length of (1 2 3 a b c) and x is a term.

But the fact that the single answer appears twice can only be understood if we know about the side-effect of evaluating the condition P(x).

Notice that the result "6" is displayed literally just before the query evaluation responds. The "P" program does not automatically put a blank or new line after the terms it has printed. If this is necessary then we can use the "PP" program. "PP" is the same as "P" with two exceptions: any term printed using "PP" is guaranteed to be 're-readable' using the "R" progrm, and it displays terms, particularly lists, in a more readable form. If "PP" is given no arguments it just generates a new line.

("P" if given no arguments does nothing). Our query above would produce a more respectable output using "PP",

```
&.Which(x(1 2 3 a b c) has-length x and PP(x))
6
Answer is 6
No (more) answers
```

The print imperatives can be used along with the read term program for writing interactive PROLOG programs. An example is the program

```
sum() if PP(Enter a list of numbers) & R(x) & sum-up(x y) &
                                           PP(sum of the list is y)
sum-up(() 0)
sum-up((x|X) z) if sum-up(X z1) & SUM(z1 1 z)
```

An example of its use is:


```
&.Does(sum())
Enter a list of numbers
.(3 5 -2 10 4)
sum of the list is 20
&.
```

The prints are also useful in the early stages of developing a program. The odd print scattered around the rules of a program does not effect its declarative reading but offers useful trace information during an evaluation. Often we can only discover that we need to revise a definition, or change the evaluation order of the preconditions of a rule, by seeing what happens to the bindings of variables when the rule is used.

As an example of the use of print for tracing, consider the rule:

```
x is-noun-phrase (NP x y) if
     (x1 x2 appends-to x
     x1 is-determiner X
     x2 is-noun-expression Y
```

that we give in Chapter 5. Suppose we wanted to trace the attempted use of the rule. We could modify it to:

```
x is-noun-phrase (NP X Y) if
     PP(Rule for noun phrase being used with sentence fragment x) &
     (x1 x2) appends-to x &
     x1 is-determiner X &
     x2 is-noun-expression Y &
     PP(Rule for noun-expression successfully applied to x with
        result (NP X Y))
```

Each time this rule is invoked we get a message that tells us it is being used and gives the sentence fragment to which it is applied. If this fragment is a noun phrase, and the rule is successfully applied, we get a message to this effect which also gives the parse structure that has been produced.

Printing variables

Since variables are converted into an internal form when they are input, and their original names are lost, it is not possible to print them using their original names. The first variable printed by "P" or "PP" is displayed as "X", the next "Y" and so on in the sequence

X, Y ,Z, x, y, z, x1,..

Each time "P" or "PP" is called the sequence is started afresh. This can lead to a situation where two apparently different variables have the same print name:

```
Does(PP(x) and PP(y))
X
X
YES
```

In these two sections we have seen how in micro-PROLOG we can interact with the 'outside world' by reading and writing terms. Although these are essentially behavioural functions with little logical correspondence careful use can greatly extend the power of micro-PROLOG programs.

6.3 Rules that ask for information

We can use the read and print primitives to write special default rules for relations. These are rules, which often are the last rule for a relation, which ask the user to confirm that some fact is true. Being the last rule, they will only be used when all other ways of confirming the fact have been explored and have failed, hence the name default rule.

As an example, let us suppose that we have a data base of family relations and that we cannot be sure that each time someone enters information about the family relations of a new person they remember to add an assertion telling us that they are male, or that they are female. This means that on occasion our sets of facts about the male, female relations may be incomplete. If we have a query that involves confirming that some new person, Percy, is male we may fail to confirm the fact because Male(Percy) was not entered. We can anticipate this by having a special default rule for these relations.

Consider the rule:

```
Male(x) if P(Is x Male? Answer YES or NO) and R(y) and y EQ YES
```

Suppose this rule comes after the "Male" facts. If, during a query evaluation, we try to confirm "Male(Percy)" the facts will be scanned to see if it is given. If it is not given the default rule will be used. The evaluation of the rule causes the message:

```
Is Percy male ? Answer YES or NO
```

to be displayed at the terminal. The name "Percy" is printed because x has been bound to "Percy". The R(y) call then causes the prompt to be displayed and a term to be read in. This is the user response. If it is YES the fact Male(Percy) is confirmed by this rule. If it is NO, the attempt to confirm the fact with this rule fails.

We can have an exactly analogous default rule for the "Female" relation. Notice that these rules are only sensible if the Male, Female facts are only used to check the sex of people, not to generate the names of males and females. They also have the disadvantage that they do not 'remember' any facts that have been confirmed. If the same Male(Percy) condition comes up again, the default rule will again query the user. Of course, their use does serve to remind the user to add this fact after the current query evaluation finishes. He does this with a

```
Add (Male(Percy))
```

command. But what if the rule could anticipate this and do the addition to the data base automatically? This means making a call to "Add" part of the default rule. "Add" can be used in this way, and when it is it has the role of imperative relation. This use of "Add" is the topic of the next section.

6.4 Rule use of Add and Delete

The read and print imperatives are used because of their side-effects on the outside world: they both effect the state of the user terminal. The "Add" and "Delete" imperatives are used because of their side-effect on the internal world of the PROLOG data base. The use of "Add" and "Delete" in rules is an example of a general feature of micro-PROLOG. Commands to the system can be used as relations in rules, and conversely, certain relations can be used as commands. We shall have more to say about the use of relations as commands in the next chapter.

Use of Add

When "Add" is used as a command it is usually immediately followed by the sentence to be added enclosed in brackets. This bracketed sentence is the argument to the "Add" command. When "Add" is used as an imperative relation this bracketed sentence becomes the single argument of the condition which is written:

```
Add((sentence))
```

The declarative reading of "Add" as a relation is:

```
Add(x) holds iff x is a bracketed sentence
```

Its behavioural reading is:

```
To confirm a fact of the form Add(x),
     check that x is a bracketed sentence,
     Add that sentence as a new last sentence for the relation
     that it is about, confirm Add(x).
```

"Add" is used for its side-effect on the data base, not to check that something is a sentence. If its argument is not a sentence, the Add(x) condition will not be confirmed nor will anything be added to the data base.

Let us see what will happen if we change the "male" default rule to:

```
Male(x) if P(Is x Male? Answer YES or NO)
          & R(y) & y = EQ YES
          & Add((Male(x)))
```

When the rule is used in an attempt to confirm "Male(Percy)", the message will be printed as before but if the response is "YES" the fact "Male(Percy)" will be added as a new last sentence about the "Male" relation. That is, it will be added after the default rulle, so we have not avoided the repeated requests to the user about Percy. To avoid this, we can either make sure that the new fact is added at the front of "Male" facts, using the two argument form of "Add" in which the position is specified, or we can separate the facts from the default rule by using an auxiliary relation, Known-Male.

Taking the first alternative, we modify the default rule to:

```
Male(x) if P(Is x Male? Answer YES or NO)
          & R(y) & y EQ YES
          & Add(0 (Male(x)))
```

This two argument form of "Add" corresponds to its command use as in:

```
Add 0 (Male(Percy))
```

We could also write this condition in the infix form that is allowed for binary relations; we could use:

```
0 Add (Male(x))
```

instead of

```
Add(0 (Male(x)))
```

The second approach, which we shall shortly discover is a more general way of coping with data supplied during an evaluation, is to separate the facts from the default rule by storing the facts as assertions about an auxiliary relation "Known-Male". Instead of

```
Male(Bill)   we use   Known-Male(Bill)
Male(Ken)             Known-Male(Ken)
   .                     .
   .                     .
   .                     .
```

This means that the user must enter information about who is male by adding "Known-Male" facts. Then, in queries and rules that need to check if someone is male, we use the old relation, "Male", defined by the two rules:

```
Male(x) if Known-Male(x)
Male(x) if P(Is x Male? Answer YES or NO)
           & R(y) & y EQ YES
           & Add((Known-Male(x)))
```

A request to confirm "Male(Percy)" is answered by trying to use the first rule. This searches the assertions about Known-Male to see if "Known-Male(Percy)" is in the data base. If it is not, the second rule queries the user. If the answer is YES, the fact "Known-Male(Percy)" is added to the data base and the user is not queried about Percy again.

This auxiliary relation solution also enables us to use the facts about "Known-Male" to generate the names of males, something we cannot do if we use a single relation name. The default rule that queries the user is of no use for finding males. If it was invoked with its argument unbound, it would simply print the message:

"Is X Male? Answer YES or NO"

Thus, when we want to generate the names of males we should use the relation "Known-Male", not the relation "Male". A condition, "Known-Male(x)" will be answered by successively giving x the value of the name of each recorded male. When we want to check if someone is male, we use the relation "Male". This includes all the recorded males as well as any that the user can tell us about.

There remains one problem with our two rule program for "Male". We shall deal with this in Section 6.5.

Saying the answers to a query as facts

We can use Add to save all the answers to a query as facts. Instead of the query:

```
Which((x y) x is-the-father-of y & Male(y))
```

we can use:

```
Which((x y) x is-the-father-of y & Male(y) & Add((y son-of-man x)))
```

At the end of the evaluation each answer is recorded is a "son-of-man" fact in the data base. We can see the answers again by "Listing" the relation, and we can use the relation "son-of-man" in subsequent queries.

If we do not want to see the answers to the query immediately, that is we just want to have them recorded, we can use:

```
Does((Add((y son-of-man x))) For-All((x y) x is-the-father-of y &
                                                         Male(y)))
```

This records the results of the query as facts. It is the data base analogue of "Is-All" which records the results in a list.

Use of Delete

The "Delete" command can be used in rules and queries to delete sentences from the data bases. As with "Add" the two forms of "Delete" as an imperative relation correspond closely to the comand forms. If the one argument form is used, the argument in the sentence to delete is enclosed in brackets. An example use is:

```
Does(Delete((Male(Algernon))))
```

This 'query' will delete the sentences "Male(Algernon)" from the program. It is identical to the command:

```
Delete(Male(Algernon))
```

The two argument form of "Delete" puts the program name as the first argument and the index of the clause to delete as the second. Therefore to delete the fourth "Male" sentence we could use the pseudo-query

```
Does(Delete(Male 4))
```

Scratch pad memory

Add and Delete in combination enable us to use the data base as a scratch pad memory. As an example, suppose that we wanted to keep track of the number of times a rule is used during some query evaluation. Suppose that we wanted to record how many times the parsing rule

```
x is-noun-expression (NE X Y) if
     (x1 x2) appends-to x &
     x1 is adjective X &
     x2 is-noun-expression Y
```

is used in parsing some sentence. We need to name the rule in some way. Let us call it "Rule-NE". Before we start to parse the sentence, we can add the fact

```
Uses(Rule-NE 0)
```

to the data base, recording no uses of the rule. We then modify by adding three extra conditions to the end of the rule. This gives us:

```
x is-noun-expression (NE X Y) if
     .
     .
     .
     x2 is-noun-expression Y &
     Delete((Uses(Rule-NE x)) & SUM(x 1 x1) &
     Add((Uses(Rule-NE x1))
```

Each time the rule is successfully applied the old count of its number of uses, recorded by the "Uses" fact, is deleted and a new one, with the count increased by 1, is added.

Notice that if we do this monitoring of the use for several rules the set of "Uses" facts is behaving as a table of information that is being continually updated during the query evaluation.

Another use of the data base as a scratch pad memory is illustrated by the following alternative program to find the sum of a list of numbers.

```
Sum-up(X y) if Add((total(0))) &
               (Update-total-with(x)) For-All(x x belongs-to X)
               & total(y)
```

```
Update-sum-with(x) if Delete((total(y))) & SUM(x y y1)
                   & Add((total(y1)))
```

In this program, the data base is used to keep a running total of the numbers in the list. As each one is retrieved using the (x x belongs-to X) condition, its value is added to the current total.

Variables in sentences

Both the "Add" and "Delete" programs accept arbitrary micro-PROLOG sentences as arguments. The sentences are in fact just lists of constants and variables that satisfy certain syntactic conditions concerning the position of brackets and the connectives "if", "and" and "&". Any unbound variables in these (sentence) lists become variables in the added sentence. As with the negation operator, "Not", this means that the position of the "Add" in a rule is crucial. It must come after any conditions that are intended to give values to the variables before the sentence is added. Thus, an evaluation of the pair of conditions

```
x EQ Algernon & Add((Male(x)))
```

adds the fact, "Male(Algernon)". An evaluation of

```
Add(Male(x)) & x EQ Algernon
```

adds the rule

```
Male(x).
```

This is because x is unbound when the Add is evaluated.

When using "Add" and "Delete" in rules we really must beware! We have to pay great attention to the behavioural reading of the rule. You must be especially careful when using "Delete". Theoretically one can use this in a rule to have the effect of deleting the rule when the rule is used. But if you try to do this micro-PROLOG will get hopelessly confused. It will get into an error state from which it cannot recover.

6.5 Modifying the behaviour of micro-PROLOG

We have seen how various facilities in micro-PROLOG can modify the outside world, and even the internal program. We conclude this chapter on the imperative aspects of micro-PROLOG by seeing how we can modify the behaviour of the evaluator. Essentially this means changing the backtracking behaviour by adding some pseudo conditions to the rules of the program.

There are two imperative relations, both of which have no arguments, which are used to control the evaluator; these are the slash (written as /()) and FAIL(). The latter has a perfectly good declarative reading, if a little trivial.

FAIL() is false! i.e. there is no fact of the form FAIL()

Of itself "FAIL" is not too useful, after all there are no positive results to be had from it, however it is very useful in conjunction with the slash.

The slash has no declarative reading except that it is always true, i.e. it always succeeds. It is used purely for its side-effect, which is to control the backtracking behaviour of the query evaluator.

Slash in queries

The main function of the slash is to eliminate alternative ways of solving queries. For example, recalling the 'Tudors' family relationship data of Chapter 1, consider this query:

```
Which(x x is-a-parent-of Henry8 and /())
```

Declaratively the x's to be retrieved are those that satisfy the condition:

```
x is-a-parent-of Henry and TRUE.
```

But the presence of the logically redundant /() will limit this query so that only one result will be returned:

```
Answer is Henry7
No (more) answers
```

Remember that to evaluate a conjunction of conditions that describe the data to be retrieved the evaluator finds a solution to each component condition one at a time. To find more solutions, it backtracks. It looks for alternative solutions to each condition. But it does this by searching for alternatives to the last condition first. Only when all these are have been exhausted does it look for alternatives to the second to last condition, and so on.

Thus, to find all the x's that satisfy the conjunction of conditons:

```
x is-a-parent-of Henry8 and /()
```

the evaluator first finds one parent, Henry7, and then tests the remaining condition /(). This is always true. But it has the side-effect of preventing the evaluator from backtracking to look for alternative solutions to any conditions that precede it in the query. That is why we get only one answer.

Slash in rules

When used in a rule the slash has a similar effect. When the rule is invoked in order to find a solution to some condition, call it C, the evaluation of the /() prevents backtracking to find alternative solutions to any conditions (of the rule) that precede the /(). Additionally, it prevents the evaluator using any other rule to find a solution to C.

Preventing redundant search

The slash in rules is often useful for preventing redundant search for alternative solutions to a condition. To illustrate this consider again the two rules:

```
Male(x) if Known-Male(x)
Male(x)  if P(Is x Male? Answer YES or NO)
            & R(y) & y= YES
            & Add((Known-Male(x)))
```

In the evaluation of the query:

```
Which(x y) Tom is-the-father-of x & Male(x) & x is-a-parent-of y)
```

these rules will be used to check that the offsprings of Tom found by evaluating the condition: "Tom is-a-father-of x" are male.

They do this by first checking if the given x is a known male and, if that fails, by asking the user. But suppose that the first child of Tom, say Bill, is confirmed as male because the assertion Known-Male(Bill) is in the data base. The evaluator will proceed by finding all solutions to the last condition, x is-a-parent-of y, with x as Bill. When all the off-springs of Bill have been found, it returns to find alternative solutions to the preceding conditions. In this case, the immediately preceding condition is Male(Bill), (Male(x) with x given as "Bill"). The evaluator is not clever enough to realise that there is no point in looking for alternative 'solutions' to a condition that has no variables. It back-tracks to where it finished the proof that Bill was male. Thus, it will continue searching through the remaining "Known-Male" assertions to see if there is a second way of confirming Bill as a known male. When it has looked at all these assertions, it will try to use the second rule. Thus, during the backtracking, the user will be asked, redundantly, to confirm that Bill is male. To prevent this pointless extra activity we add a /() to the first rule. Writing the two rules:

```
Male(x) if Known-Male(x) & /()
Male(x) if P(Is x Male? Answer YES or NO)
         & R(y) & y = YES
         & Add(Known-Male(x))
```

means that as soon as Bill is confirmed as a known male, micro-PROLOG will abandon its scan of sentences about "Known-Male". This is because it will not backtrack to look for other ways of solving the condition Known-Male(x), with x=Bill, of the first rule. Additionally, it will not try to use the second rule to find an alternative way of solving "Male(Bill)".

Slash with FAIL

The main point about combining the slash and FAIL in a rule is to deliberately prevent the successful use of the rule. We shall start by seeing how this combination can be used to define a complement relation, "Not-Male", which is true if its argument is not male. The rules for "Not-Male" are:

```
Not-Male(x) if Male(x) and /() and FAIL()        (1)
Not-Male(x)                                      (2)
```

This program can only be read behaviourally; while there may be a declarative reading of "Not-Male", the rules (5) and (6) bear no relation to this. By tracing the evaluation of two queries on "Not-male" we can see how (1) and (2) behave. First we look at the query:

```
Does(Not-Male(John))                             (A)
```

By using rule (1) this is reduced to the sub-query:

```
Male(John) and /() and FAIL()                    (B)
```

The "Male(John)" condition is satisfied (we shall assume) and (B) is reduced to:

```
/() and FAIL()                                   (C)
```

The slash is executed, which removes any possibility of finding more ways of confirming that "Male(John)" is true. However it __also__ removes from consideration rule (6) from "Not-Male". The (C) query is now:

```
FAIL()                                           (D)
```

According to the definition of "FAIL" this query has no solutions, so the evaluator __backtracks__ looking for an alternative way of solving (C) and

then because the slash has no alternatives retrying (B). Because of the slash's side effects there are no more ways of checking for "Male(John)", so the evaluator backtracks further to (A). Again because rule (2) was eliminated as a way of solving (A) this also has no more possible solutions: hence the query (A) __fails__. This produces the "NO" response to (A). In other words it is __not__ the case that John is not "Male".

Now let us look at a different "Not-Male" query:

```
Does(Not-Male(Jill))                             (E)
```

As before (E) reduces to the query:

```
Male(Jill) and /() and FAIL()                    (F)
```

This time the "Male(Jill)" fact is not confirmed (if the default rule was invoked then the response to "Is Jill male?" question was "NO"). The effect of this is for the evaluator to backtrack as before: this time there is an alternative way of reducing the query (E) using rule (2). The use of this rule succeeds and the (E) query is accordingly confirmed.

The behavioural reading of the Not-Male program given by the two rules (1) and (2) is therefore:

```
To confirm a fact of the form Not-Male(x):
    test if Male(x) can be confirmed
        if it can, fail to confirm Not-Male(x)
        if it cannot, confirm Not-Male(x)
```

Of course, this behaviour is also that of the alternative definition

```
Not-Male(x) if Not(Male(x))
```

that makes use of the micro-PROLOG primitive "Not". This definition also has the merit of an appropriate declarative reading. However, as we shall see in the next chapter, "Not" is itself defined by a micro-PROLOG program that uses "/()" and "FAIL()" is just the way.

Conclusion

We have come down to earth a little in this chapter, and we have seen some very conventional computing techniques presented. If you were a determined BASIC hacker, or Pascal fiend, you could be quite easily seduced into using these facilities indiscriminately. While not taking a totally purist view (after all if we did this chapter would not exist) there are a number of points to note.

Firstly many of the common uses of these facilities have already been 'packaged up' into high level features; features such as Not, For-All, Is-All and so on. It is quite likely that these are more efficiently implemented than could be done by someone with little experience in PROLOG-hacking, so why not make use of them - they cost nothing to use!

Secondly, because of familiarity with conventional computing techniques you might be too lazy to 're-work' your particular algorithm to the higher level logic programming view. This may well result in clumsy PROLOG programs __and__ less efficient ones. The guiding principle should be to restrict use of behavioural primitives to those cases where it is absolutely necessary.

Thirdly, any use of the behaviour of the machine to control the results produced makes understanding programs much harder. In particular the __logic__ of the program can no longer give the complete picture, and this means that program __synthesis__ and __manipulation__ is very difficult.

Finally we note that the imperative features tend to rely very heavily on the underlying 'machine'. It is quite likely that new computer architectures will make the backtracker obsolete; for example computers which can process queries in a parallel way would probably not use a backtracker. This means that when moving to such a new style computer any parts of the program which depend on the backtracker become obsolete, whereas the 'pure' logic program needs no change. In fact it seems increasingly likely that the next generation but one of computers will indeed have a radically different architecture: one which may do many thousand different operations at once in parallel.

7. The internal syntax of micro-PROLOG

The programs and queries that we have seen up until now are all in a special easy to read syntax. There is another internal syntax for programs and queries which has a simpler structure but which is less readable. The internal syntax is, in fact, the only syntax directly understood by micro-PROLOG. Programs written in the **surface** syntax (i.e. the syntax used so far) are converted sentence by sentence into the internal form. Similarly, queries are converted to their internal equivalents before they can be answered. All this is accomplished by a special 'front-end' micro-PROLOG program. This program is called "simple" and is loaded into micro-PROLOG at the start of each session.

In this chapter we give a flavour of the internal syntax, and see some of its expressive power. As might be expected the internal syntax is entirely based on lists; this means that programs in internal syntax are less readable than surface programs, but they are very easy to construct by other micro-PROLOG programs. PROLOG shares with LISP the ability to treat data objects as programs and vice-versa; a property which is very heavily used by the front end program.

We also look at the **module** system of micro-PROLOG. This enables different programs to be 'put together' from several sources, whilst at the same time minimizing name clashes. This is especially necessary when combining programs written by more than one person. The "simple" front end program is implemented as a module.

7.1 Clausal Notation

We adopt a slightly different terminology when talking about programs and queries written in the internal syntax. This helps to avoid confusion when we are discussing the differences between the two forms of syntax. For example, a sentence in surface syntax **is a clause** in internal syntax. So, whenever the front end program accepts an "Add" command the sentence is converted into an equivalent clause and is retained only in the clause form. When the program is listed, using the "List" command, the clause is converted back into sentence form before it is displayed. This means that, for most applications, the programmer does not need to know about the internal syntax.

We shall see shortly that a clause is just a list of **atoms**, an atom being the internal form of the conditions and conclusions of a sentence. The first element, called the **predicate symbol**, is the relation name of the surface level condition and the rest of the list are the arguments of the condition.

Surface form	Internal form
John likes Mary	(likes John Mary)
SUM(1 2 3)	(SUM 1 2 3)

In general, a binary form condition of the form

    arg1 R arg2 becomes the atom (R arg1 arg2),

and the prefix form condition

    P(arg1....argk) becomes the atom (P arg1....argk)

There are no differences between the internal and surface form of arguments of atoms. The general name for the argument of a relation is **term**. Thus, a **term** is a variable, a number, an atom or a list.

Just as an atom is a list so a clause is a list. The first element of the list is the atom corresponding to the head or conclusion of the sentence, and the tail of the list comprises the atoms representing its preconditions. Thus, a single sentence becomes a list of one atom.

Surface form sentence	Internal form clause
John likes Mary	((likes John Mary))
x member-of (x|y)	((member-of x (x|y)))
App(() x x)	((App () x x))

A compound sentence becomes a list of at least two atoms. There are no keywords between the atoms in the clause like the "if" and "and" of the surface form:

```
Surface form sentence              Internal form clause

x is-a-friend-of y if              ((is-a-friend-of x y)
     x likes y and                        (likes x y)
     y likes x                            (likes y x))
x member-of (y|z) if               ((member-of x (y|z))
     x member-of z                        (member-of x z))
App((x|X) Y (x|Z)) if              ((App(x|X) Y (x|Z)) if
     App(X Y Z)                           (App X Y Z))
```

As we can see there is quite a simple correspondence between the internal and surface syntax for sentences. There tend to be more brackets in the internal form and there are no keywords to separate out the various components of the clause. There is a similar correspondence between surface queries and internal **goals.**

While there is no direct equivalent of the "Which" query in the internal form, there is an equivalent of the "Does" query: the "?". This takes as argument a list of atoms which are then evaluated in the same way that the "Does" query evaluates its conjunction of conditions. However the "?" is silent in comparison with the "Does" command; if the query succeeds there is just a new "&." prompt printed, as in:

```
&.?((likes x y))
&.
```

Only if the query fails does the system respond with a "?"

```
&.?((member-of A (B)))
?
&.
```

The "?" query facility at the internal syntax level is the primitive in terms of which all other forms of querying are implemented. Thus, the Does command is just a combination of a translation to internal form a use of "?", and the printing of a suitable response. The Which and One query forms are similarly implemented in terms of "?". Shortly, we shall see how "?" can be used to implement an equivalent to the "Which" command for a query expressed in internal syntax. Let us call this "Wh". An example use of this, corresponding to the "Which" query:

```
&. Which((x y) x likes y & y likes x)
```

would be:

```
&.Wh((x y) (likes x y)(likes y x))
Answer is (John Mary)
Answer is (Mary John)
```



```
No (more) answers
&.
```

Remember the "Wh" command is not built-in to micro-PROLOG. We shall see below how it is defined and how new commands can be added to the system. In the mean time we shall use it as though it were defined.

The "List" command converts clauses back into sentences before they are displayed. If we want them displayed in their internal form, we must use the "LIST" command. (same word, but all letters uppercase). "LIST" command has three forms, two of which we look at now.

To see the entire program use the form "LIST ALL". This corresponds directly with the surface level command "List All", except the program appears in internal form rather than surface form.

```
&.LIST ALL
((is-the-father-of Henry7 Henry8))
     .
     .
     .
     .
((dict is-the-father-of))
((dict is-a-parent-of))
     .
     .
     .
     .
((is-a-parent-of ....)))
```

To list individual programs or a collection of these, the "ALL" keyword is replaced by a list of predicate symbols. For example to list "Likes" and "dict" we would use:

```
&.LIST (Likes dict)
((Likes John Mary))
     .
     .
     .
((dict is-a-parent-of))
     .
     .
     .
&.
```

We shall see the third form of "LIST" when we look at modules below.

We can also directly enter facts and rules written in internal syntax. To do this we simply type in the clause without using the Add command. The Add is the translater to internal syntax which is not needed in this case. Thus to add a new fact about the Likes relation we can type

```
&. ((Likes John Jim))
```

or we can type:

```
&. Add(John Likes Jim)
```

In the second case, Add translates the bracketed sentence (John Likes Jim) into the clause ((Likes John Jim)).

7.2 The Meta-variable

What we have seen of the internal form of micro-PROLOG corresponds quite closely to the surface level; however because of the list based notation of clauses we can achieve greater expressive power. The main source of this is the so-called **meta-variable** where different parts of a clause can be named by variables. Meta-variables are a very powerful tool in micro-PROLOG; with them we can write powerful generic programs. There are many examples of this in the front end program "simple".

The main principle behind the meta-variable is that during the evaluation the meta-variable will be given a value before micro-PROLOG comes to evaluate the part of the clause in which it appears. This value must be such that it is syntactically correct for the part of the clause represented by the meta-variable. The clause is then used as though it had been written with the value in place of the variable. It is called meta-variable because the variable names part of a clause, i.e. part of the program. This is different from the normal use of variables to name unknown individuals that lie in the domain of specified relations.

There are four different ways that the meta-variable can be used in a clause, these arise naturally from the list structure of clauses. These various uses also have parallels in more conventional programming languages, notably Pascal, ALGOL and "C". We will point out these analogies where it is appropriate. Readers not familiar with these languages should ignore these comments.

Meta-variable replacing the predicate symbol

In this first case, the predicate symbol of an atom in a query or the body of a clause is given as a variable. Recall that an atom is a list, the first element of which is the predicate symbol. If this is a variable, the variable must have to bound to a constant predicate symbol before the atom is evaluated. In practice this means that the variable must appear in an earlier atom of the query or clause. The predicate symbol of the head atom of a clause can never be given as a variable, it must always be a constant. (If it was given as a variable micro-PROLOG would not know what relation the clause was about.)

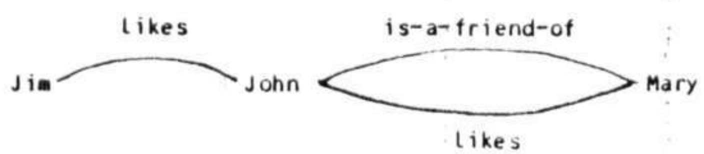
For example, the "simple" front end program maintains an auxiliary program called "dict" which consists of a table of all the predicate symbols for which there are sentences/clauses. We can make use of this dict relation to generate names of relations:

```
Wh(x (dict x)(x John Mary))
Answer is likes
Answer is is-a-friend-of
No(more) answers
```

What this query asks is:

What relationships are known to hold between "John" and "Mary"?

Suppose we view a collection of facts about binary relations as the description of a graph in which the nodes are labelled by the individuals and the arcs by the relation names as in:



This use of the meta-variable enables us to find the names on the arcs between particular nodes, as in the above query. It also enables us to find all the nodes connected to a given node together with the name of the connection:

```
&.Wh((x z) (dict x) (x John z))
Answer is (likes Mary)
Answer is (is-a-friend-of Mary)
Answer is (likes Jim)
No (more) answers
```

The clause

```
((connects x y z) (dict x) (x y z))
```

is a rule that can be used to walk over this graph.

Clearly this is quite a powerful query and it shows off some of the flexibility of PROLOG. However, the meta-variable as predicate symbol has other uses. For example we can emulate the various "MAP" functions of LISP: those functions which apply a function to each element of a list to construct a new list. Let us define the three argument relation "maplist". The first argument is a predicate symbol, which is assumed to be binary. The second argument is the 'input list' and the third is the 'output list'.

In fact (maplist x y z) is true if

y and z have the same length,
 and for each element (call it y') of y,
 and corresponding element of z (call it z')
 (x y' z') is true.

The program for "maplist" (in internal form) is:

```
((maplist x () ()))
((maplist x (y|Y) (z|Z))
  (x y z)
  (maplist x Y Z))
```

Another use of the predicate meta-variable is analogous to the reduction operator in APL. This operator reduces a vector by iteratively applying some binary function over the whole vector; for example reduction of a vector using addition adds up all the elements of the vector. If multiplication were used instead then the result is the product of the elements of the vector.

The analogy in micro-PROLOG would be to reduce a list by iteratively applying a ternary relation to all the elements of the list. For example reducing a list using addition would total up the list. The program for reducing a list is very similar to maplist:

```
((reduce x (y) y))
((reduce x (y1 y2|Y) z)
   (x y1 y2 y3)
   (reduce x (y3|Y) z))
```

A typical call like: "(reduce SUM (1 2 3 4) x)" results in "x" becoming the sum of the list (1 2 3 4).

By using a double call of "reduce" on a list of pairs of numbers, the 'dot' product of two vectors can be defined. The definition of the dot product is left as an exercise to the reader.

This form of meta-variable has an analogy in many conventional programming languages: the passing of procedures as parameters. For example in Pascal it is possible to have a procedure or function name as the parameter of another procedure or function (or even the same one). The 'host' procedure supplies the actual parameters to the 'guest' procedure whose name has been passed. However in Pascal, as in many other similar languages, the name of a procedure is not a 'first class' object it cannot be assigned to a variable or stored in a data structure.

In micro-PROLOG the predicate symbol is such a first class object; it is a constant and as such can be stored, passed around and retrieved with total flexibility.

Exercise 7-1

1. Write a program, in surface form, which takes a pair of lists and returns a list of pairs: each pair coming from successive elements of the two lists. For example:

```
Which(x pair((1 2 3) (a b c) x)
Answer is ((1 a) (2 b) (3 c))
```

2. Use this program, together with "reduce" to write the program "dot" which performs the dot product of a pair of lists of numbers. The dot product is the sum of the pair-wise products of the elements of the lists. I.e. given the two lists (a b c) and (d e f) then the dot product is a*d + b*e + c*f.

3. The meta-variable can be used to implement a very simple arithmetic expression evaluator. These arithmetic expressions have essentially two shapes; either the expression to evaluate is already a number in which case the value of it is itself, or it is a triple in the form "(leftarg operator rightarg)". In this case the value is gotten by evaluating the left and right hand arguments and applying the relation given as the operator to their values. Each operator must therefore be defined as a three argument relation, the final argument being the result of 'applying' the operator.

Write this *program* in internal form (call it "has-val") and test it using the arithmetic operators "+", "-", "*", "md" and "dv". (These are defined within the "simple" front end program.) As an example, the definition of the operator/relation "-" is the clause:

```
((- x y z) (SUM y z x))
```

Remember that to add a clause in internal form you just type the clause when given the "&." prompt. After the return is typed, a new "&." prompt indicates the clause was accepted and entered as a new last clause for the predicate of its first atom.

Test your program with a query such as

```
Which(x ((2 * 3) dv (-3 + 5)) has-val x)
```

Meta-variable as an atom

Apart from simply naming the predicate symbol of a call atom, the whole atom can be named by a variable. This variable must now be bound to a list term that is an atom. This is the most common meta-variable in PROLOG. It is used to implement the Not, For-All and Is-All operators.

A very simple use is in the clause:

```
((Holds x) x)
```

The Holds relation is true of a term if and only if (iff) that term is an atom that is proveable. To define the negation of holds, a relation "NOT" which is true of a term iff that term is an atom that is not proveable, we use the two clauses:


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

When used to try to establish (NOT A), A some atom, the first rule of this program is invoked. It reduces (not A) to A. If A can be proved, the (/) prevents use of the second rule and the (FAIL) ensures failure of the (not A) call. Only if A cannot be established will the second rule be used. But this is exactly the circumstance in which (not A) holds. In internal syntax we can also write the first rule as

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

dropping the brackets from the no-argument predicates "/" and "FAIL".

This definition of "not" restricts x to a single atom. The "not" that we have used so far could handle a conjunction of conditions. In internal syntax a conjunction of conditions is a list of atoms. The following rules define a "Not" that has a list of atoms as its argument:

```
((Not x) (? x) / FAIL)
((Not x))
```

The difference is that here the system provided "?" is used to check if all the atoms on the list x are proveable.

This form of the meta-variable has no obvious counterpart in conventional programming languages (apart from LISP). There is a link with ALGOL 60 and its close counterparts though with the 'call-by-NAME' parameter passing mechanism.

We saw above that the meta-variable as a predicate symbol was close to the procedure name passing mechanism of Pascal: the name of the procedure was passed and the actual arguments are given by the host procedure. In the atom form of meta-variable the whole 'procedure call' is passed, an operation akin to passing an unevaluated expression to a procedure. The time that the expression is evaluated is determined by where the meta-variable appears; this is exactly analogous to call-by-NAME. A value passed by call-by-NAME in ALGOL 60 is actually passed as a special unevaluated expression (called a "thunk" for the technically curious) which then evaluated as the corresponding formal parameter appears in the text.

Meta-variable as the tail of a clause

Another variant of the meta-variable is the meta-variable as the body of a clause. The simplest example of this is the program for the "?" operator:

```
((? x)|x)                                            (1)
```

The variable "x" in (1) must be bound to a list of atoms. The meaning of "?" is quite simple:

(? x) is true if x is a list of atoms which are all proveable

It acts as a 'conjunction operator' which combines a number of atoms into one call. A typical call to "?" might be:

```
(? ((Likes x y)(Male x)(not (Male(y))) ))
```

This program for "?" also defines the command "?" which we saw above was the internal equivalent of "Does". As an evaluation initiating command the above call is written:

? ((likes x y) (Male x) (not (Male y)))

note that the outermost brackets are dropped. Any unary relation can be used as a command in this way. Instead of the call

(R arg)

we can enter the command

R arg

We can use the "?" program to implement the "Wh" command we have been using instead of "Which". Recall that the argument to "Wh" is a list, the head of is the output term and the tail is a query in internal form. We can evaluate this query using the "?", and then print the answer. By using "FAIL" we can force the repeated evaluation of the query to find all the answers. The program for "Wh" is:

```
((Wh (x|y))
    (? y)
    (P Answer is x)
    PP
    FAIL)
((Wh x)
   (PP No (more) answers))
```

This program makes use of the fact that as the query y is evaluated it finds values of variables, some of which are also in x. When the output term x is printed, it is evaluated relative to these solution bindings. So we get the x for one solution of the query y. The "PP" on its own is the internal syntax call to print a new line. Recall that the "PP" built-in program is one of the print term facilities. It always prints a new line after it has printed its arguments, so "PP" with no arguments just prints a new line. In internal syntax the no argument call "(PP)" can be simplified to "PP".

The FAIL is then used to cause backtracking to find the next solution. When there are no more solutions to be found, i.e. the (? y) call ultimately fails, the second clause prints the "No (more) answers" message.

The "Which" command recognised by the simple translater program is just a combination of a cnversion of a compound condition to a list of atoms and a "Wh" evaluation.

There are many other uses for the meta-variables, as the tail of a clause. Below is a program for the "OR" relation which takes two lists of atoms as arguments and is true if either list represents a proveable conjunction:

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

Another use is the definition of the "IF" relation. "IF" has three arguments, an atom which is the conditional test and two 'arms' which are lists of atoms and correspond to the 'then' and 'else' branches. Thus (IF x y z) is proveable if x and y are proveable or if x is not proveable and z is proveable. It is defined by:

```
((IF x y z) x / | y)
((IF x y z) | z)
```

Notice that we have two types of meta-variable in the first clause. The x stands for an atom, the y for the tail of the clause.

Exercise 7-2

Write a program based on the "Wh" program which is analogous to the "One" command of the "simple" front end program. This involves prompting the console after each solution is found. If the response is "C" then use the "FAIL" to force the system to find the next solution, otherwise do nothing.

Meta-variable as the list of arguments

The pattern (x|y) is a list with head x and tail y. When this pattern is used in place of an atom, y is a meta-variable standing for the list of arguments of the atom. This form of meta variable is used when the number of arguments is unknown. Thus, the query:

&. Wh((x|z) (dict x) (x Tom|z))

is the generalisation of the query

&. Wh((x z) (dict x) (x Tom z)),

that we encountered above. The generalisation removes the restriction to binary relations. It gives all the tuples of individuals related to Tom by any relation. This is because the pattern (x Tom|z) denotes an atom of any number of arguments beginning with "Tom".

A meta-variable standing for a list of arguments can appear in the head atom of a clause. The head of a clause can be an atom (C|x) where C is the constant which is the predicate symbol. This use enables us to define relations with a variable number of arguments.

A simple example is the Sum-up relation which has n + 1 arguments: the first is the sum of all the others. It is defined by the single clause:

```
((Sum-up x|y)
   (reduce SUM y ~~0~~ x))
```

This makes use of the reduced relation given in above. A typical call would be (Sum-up x 3 4 5) which binds x to the SUM of the three number arguments.

In practical terms multi-argument relations enable us to drop brackets. We could have defined Sum-up as a binary relatiion between a number and a list of numbers. Its definition would then be

```
             SUM y
((Sum-up x y) (reduce y 0 x))
```

But to use the program we would now have to write the multi-argument call (Sum-up x 3 4 5) as the two argument call (Sum-up x (3 4 5)) in which we wrap-up all but the first argument as a list. In this instance, there is not much advantage to having the multi-argument form, but for other relations with atom arguments it offfers a much more readable syntax.

Earlier we defined a modification of the micro-PROLOG primitive NOT which took a list of atoms as its argument. An example call to the more general, "not" relation, is

(not ((Tall Tom) (Fat Tom)))

The single argument for "not" is the list of atoms

((Tall Tom) (Fat Tom))

It is more convenient to have "not" as a multi-argument relation, able to take any number of atom arguments. We could then write the call

```
(not (Tall Tom) (Fat Tom))
```

Indeed, this is the internal syntax equivalent of the surface syntax

```
Not(Tall(Tom) & Fat(Tom))
```

The rules defining the multi-argument "Not" are:

```
((Not|x) (? x) / FAIL)
((Not|x))
```

In a similar way we can define a multi-argument version of the Holds program which took a single atom as argument. The multi-argument version of Holds which takes a list of atoms is defined by:

```
((Holds | x) | x)
```

A typical call is (Holds (Tall Tom) (Fat Tom)). Compare this with the single atom argument definition given earlier:

```
((Holds x) x)
```

and the definition of "?" which takes a single list of atoms as its argument:

```
((? x) | x)
```

In the multi-argument definition of Holds x is a meta-variable standing for the variable length list of arguments and the variable length body of the clause. In the single atom definition x stands for the single argument and the single atom of the body of the clause. In the definition of ? x is the single list argument which becomes the variable length body of the clause.

<u>Exercise</u> 7-3

1. Define the multi-argument version of One-of.

2. Define a binary relation apply which takes the name of a relation and a list as arguments and it 'applys' the relation to the list. That is,

   (apply R args)

   holds iff args is a tuple of arguments in the relation R.

<u>Using the Meta-variable at the surface level</u>

Generally there is no direct equivalent of the Meta-variable available at the surface level of the system. However, the programs that we have been defining such as "Holds" and "?" <u>are</u> available for use at the surface level.

Using them we can obtain the effect of the meta-variable used in the body of a clause. Thus, the surface form equivalent of the maplist program given above is:


```
maplist(x () ())
maplist(x (y|Y) (z|Z)) if Holds((x y z))
                        & maplist(x Y Z)
```

in which the atom (x y z) of the internal program is replaced by the condition

```
Holds((x y z))
```

Instead of "Holds", we can use "One-of". This is particularly useful when we know there is only one way of solving the condition named by the argument to "One-of". It is particularly useful for test calls. The query

```
Which(x Tom-is-the-father-of x & One-of((Male x)))
```

will be evaluated. Much more efficiently than the one in which we use the condition Male(x) in place of One-of((Male x)). "One-of" causes micro-PROLOG to abandon its search of the "Male" sentences as soon as it has confirmed that some given x is male. If we use Male(x), micro-PROLOG will continue the search to see if it can confirm the condition in another way, something that we know is unnecessary.

7.3 <u>The dictionary and Modules</u>

In this section we look at an important data structure in micro-PROLOG: the dictionary; and how it can be structured into modules. micro-PROLOG is the first PROLOG system which allows large programs to be structured into separate independent sub-programs (called modules) which can then be combined into one.

This facility is very important since it allows program structure to reflect function: related clauses can be grouped together in a module with well defined interfaces to the rest of the program; and modules also allow programs written by different people to be brought together with the minimum anxiety about name clashes etc. A classic example of a module is the "simple" front end program itself.

To fully understand modules it is necessary first to look at the internal dictionary of micro-PROLOG, since modules are essentially a super structure on top of the basic dictionary. The dictionary is quite simply a list of the constants currently in use in the system!

This list is used by the system when it is reading in a term via the "R" and "READ" built-in-programs. When a constant is read in the dictionary list is searched. If the constant is already in the dictionary it is replaced, in the term, by a pointer to its entry in the dictionary. If it is not in the dictionary, it is added, and then replaced by a pointer to the new entry.

This means that constants are represented internally as pointers: this is the so-called **unique reference property** of constants. When two constants are compared during a pattern match it is the pointers that are compared <u>not the print names</u> of the constants.

The only point at which the print name of a constant is important is when it is being read in from the keyboard (or a file), and when it is being printed, otherwise only the reference is important.

The system of modules in micro-PROLOG imposes a hierarchy on this simple dictionary structure. A **module** has four components. Three of these are dictionaries, i.e. lists of constants, and the fourth is a constant: the <u>name</u> of the module. The name of a module is <u>not</u> part of the module.

The three dictionaries associated with the module are the **export** list, the **import** list and the **local** dictionary. Constants appearing in the local dictionary are **private** to the module. Whereas constants appearing in the other two are <u>shared</u> in some way. The dictionaries have no names in common.

The constants appearing in the local dictionaries of two different modules are distinguished even if they have the same print name. This is the most crucial property of any system of modules, it allows programs to be combined even if some of the names in the programs clash.

There are two ways that constants can be 'communicated' across modules, one via the export list, and one via the import list. A constant appearing in the export list or the local list belongs to the module, and one appearing in the import list does not.

Modules can be created, opened, saved, loaded and listed. To list a module the third form of the "LIST" command is used: "LIST module-name". To experiment try: "LIST Simple", and you will get a listing of the "simple" front end program. A module is listed in a particular format, different from the normal listing which consists simply of the clauses in a program.

The first part of a module listing consists of the module name, then the export list and then the import list. The first part of the "Simple" listing is

```
Simple(Add  }
   .        }
   .        }  export list
   .        }
)
(
   .        }
   .        }  import list
   .        }
)
```

After this preliminary the clauses of the module are listed as in the normal listing. The listing of the module is terminated by the symbol "CLMOD". Modules are SAVED on disk in exactly this format. "LOAD" knows about files in this format and correctly re-creates the modules.

At any one time there is one 'current' module. It is to this module that any terms and clauses entered belong. The local dictionary of the current module is the one which is accessed and extended when a new constant is encountered in the input.

You can use in queries and in clauses entered into the current module constants which have been exported by other modules; so long as the exporting module is either a sub-module of the current module, or if the current module explicitly imports the constant. So this would allow, for example, two modules to export to each other different programs.

The current module dictionary is accessible through the built-in program "DICT". This has only one clause whose structure reflects the module: the first argument is a constant which is the name of the current module. The second argument is the export list, the third argument is the import list. The remaining arguments are all constants (of indeterminate number) and they form the local dictionary. To see the "DICT" clause, the second form of "LIST" should be used:

```
&.LIST(DICT)
((DICT & () (...) ........))
```

A further point to note: the top level prompt is actually made up by displaying the current module's name followed by the normal input prompt: ".". The 'root' module has name "&", hence we get usual prompt of "&." at the top level. If the current module were "Simple" then the normal top level prompt becomes "Simple.".

When a module is loaded into the current module, for example when simple is loaded into the root module "&", all its exported constants

become accessible to the current module. Thus, when an exported constant is the name of a relation (e.g. Which, Add, Is-All, in the case of simple), its definition in the loaded module becomes accessible from the current module. Conversely, the descriptions of the imported constants of the module descriptions provided by the current module, become accessible to the loaded module. The current module description of a constant comprises all the clauses about the constant accessible from the current module. This may be clauses about the constant that have been "Added" to the current module, or clauses about the constant that are accessible because it is in the export list of some other module created in, or loaded into, the current module.

Creating modules

There are three built-in programs in micro-PROLOG which manipulate and access modules. They are "CRMOD" which creates a new module, "OPMOD" which opens up an existing module (i.e. makes it the current module), and "CLMOD" which closes a module and 'pops' up to the previously open module. These three merely provide the bare basics of module construction, and as experience with the modules grows these facilities may be extended.

We now see just how to create a module which, like the "simple" front end program, can be used by others with the main concern being the minimization of name clashes. We shall take "Owh" from exercise 7-2 and convert it into a module format, which can then be loaded together with other modules and programs.

Firstly, since the module system is a function of the dictionary the module structure can only be created on input. If the program to be created into module is already in the system then it is too late. So the first step to creating a module may be dumping the program onto a disk file and leaving the micro-PROLOG system!

Assuming that this has been done, we have to determine those symbols which are to be exported, and those to imported, and the name of the module. In this case we export the name of the program which is "Owh". Since the "Owh" program needs to read in a response from the keyboard and compare it against the continue command "C", this symbol will have to be explicitly imported into the module. Otherwise the constant "C" in the "Owh" program remains private to the module. This would make the program very frustrating to use since any "C" typed in response would not be the same "C" as in "Owh", and therefore it would not behave too well. (Only one solution would ever be found.)

Having decided the export/import list we 'evaluate' the following:

```
?((CRMOD Whmod (Owh)(C)))
```

The prompt changes from "&." to "Whmod" to reflect that the new module (whose name is Whmod) is now current. The original "Owh" program is then reLOADed, while the new module is current:

```
Whmod. LOAD wh2
```

Any sub-modules that the program needs should be loaded in at this point. Then the module is finished off using the built-in program "CLMOD".

```
Whmod.?((CLMOD))
&.
```

Now we have created a module in the system, it can be saved on disk by using a variant of the "SAVE" command:

```
?((SAVE WHFILE Whmod))
```

Having saved the file we have created a special module which has this "Wh" program in it. If we ever need to use it we can simply type LOAD WHFILE and it will become available, though we would not normally be able to see it.

To change a module once it has been created we can load it, and then open it with the "OPMOD" command. So

```
LOAD WHFILE
OPMOD Whmod
```

loads and enters the created Whmod module. We can change the clauses in the module, or add new ones. What we cannot do is change the input/export lists of the module. These are fixed when we first issued the "CRMOD" command. The only way these can be changed is by using a text editor on the text of the module in the file WHFILE. If you examine the text, at the operating system level, you will see that it is exactly what is printed when we do a "LIST" of the module within micro-PROLOG. You can edit the text, by adding or deleting to the input, export lists. Notice that the SAVED module is always stored in internal syntax.

Exercise 7-4

Go through the process of creating a module using one of your programs developed from previous chapters. Verify that it still interfaces properly with the "simple" front end program.

Appendix A

Instructions for Running micro-PROLOG

A. Loading

To start at the beginning. You will need a Z-80 based micro-computer using the CP/M operating system (such as Research Machines 380Z, North Star Horizon, Sharp, Tandy, Heathkit/Zenith). In the disk drive (or the A drive if your computer has two or more drives) you will place the system disk, which will include "PROLOG.COM" and "SIMPLE.LOG" among its programs.

Having inserted the disk and reset the computer if necessary, CP/M should be announced on the screen, followed by

```
A>
```

To enter micro-PROLOG, type

```
A> prolog
```

and it should be announced on the screen, in a form such as

```
Micro-PROLOG 2.XX  S/N     xxxx
(c) Logic Programming Associates Ltd. 1981
99999 Bytes Free
```

(There will be variations in what is announced according to the particular version of micro-PROLOG used, and the size of the computer.) The prompt

```
&.
```

will then appear, indicating that the computer is ready. To use the form of micro-PROLOG described in this primer, type

```
&. LOAD simple
```

The prompt

```
&.
```

will appear again when translater program "simple" has been loaded. The computer is now ready to accept programs and queries in the user friendly syntax of this primer.

It is also possible to type, using all lower case letters,

```
A>prolog load simple
```

as a shorter means of entering PROLOG and loading simple.

Note that it is possible under most CP/M's to configure the system disk to automatically start PROLOG, and load simple, when the computer is first switched on. The details of this vary from machine to machine (see local guide to CP/M).

B. Summary of the surface syntax of micro-PROLOG

A Sentence is either a simple sentence,
or a conditional sentence.

A simple sentence is either a binary simple sentence,
or a non-binary simple sentence.

A binary simple sentence is a term followed by a name of relationship followed by another term.

example: John likes Mary

A non-binary simple sentence is a name of relationship followed by a list of terms.

example: SUM(1 2 3)

A term is either a number, e.g. 1 -30
or a constant, e.g. FRED "a man"
or a variable, e.g. x y z10
or a list of terms e.g. () (1 2) (x FRED 3 (5 y))

A conditional sentence is a simple sentence followed by the word "if" followed by a compound condition

example: x is-a-friend-of y if x likes y and y likes x

A compound condition is a simple condition, possibly followed by the word "and" (or "&") followed by a compound condition

A simple condition is either a simple sentence,
or a negated condition,
or a For-All condition,
or an Is-All condition.

A negated condition is the word "Not" followed by a compound condition in parentheses.

example: Female(x) if Person(x) & Not(Male(x))

A **For-All** condition is of the form:

(Compound-condition) For-All (List-of-vars Compound-condition)

example: No-unmarried-children(x) if (Married(y)) For-All ((y) x is-a-parent-of y)

An **Is-All** condition is of the form:

term Is-All (term Compound-condition)

example: x are-children-of y if x Is-All(z y is-a-parent-of z)

C. Commands recognised by the simple translater program

Any of the following commands can be given immediately after micro-PROLOG has displayed its "&." prompt. The simple program must have been loaded. It is the program that interprets the commands and which recognises the surface syntax for micro-PROLOG sentences.

Add

The Add command allows you to add a PROLOG (surface syntax) sentence into the workspace. The format of the command is: Add(sentence). For example,

&. Add(John likes Fred if Fred likes Mary)

adds a new sentence ffor the likes relation into the program.

Added sentences are always added to the end of the program for the appropriate relation. To add into the middle of a program the form

Add n (sentence)

is used, where the number n is the position after which the new sentence is to be added.

For example, to add to the beginning of the "likes" relation use

&. Add 0(Peter likes John)

To add after the third sentence for the likes relation, use

&. Add 3(Peter likes John)

List

The List command displays the program on the screen. To display the whole of your program, type

&. List All

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

&. List likes

Note The CP/M Control-P function can be used to cause the screen listing to be copied on a printer.

Delete

Deletes a sentence from the program. Its usage is illustrated by

&. Delete (John likes Mary)

which deletes the sentence named, or

&. Delete likes 3

which deletes the third likes sentence.

Kill

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

&. Kill likes

Does

The Does command poses a query with a YES/NO answer. For example

&. Does (John likes Mary)
YES

In general, the argument of Does is a bracketed compound condition.

Which

The Which command is used to retrieve names satisfying some query condition. The form of the answer required is specified by the question. For example, to list people who are parents of Edward, use

```
&. Which(x x is-a-parent-of Edward)
   Answer is Henry8
   Answer is Jane
   No (more) answers
```

The general form of the compound is Which(term compound-condition). It gives the value of term for each answer to the compound condition.

<u>One</u>

The One query operates in a similar way to "Which", but prompts after finding each answer.

```
&.One(x x is-a-parent-of Edward)
Answer is Henry8.

                Type "C" to continue
                Type "F" to finish
```

The general form is One(term compound-condition).

<u>Save</u>

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

```
&. Save filename
```

where filename is a file name, with a maximum significant length eight characters. The program will be saved in a CP/M file called filename.log on the currently selected drive. To save on another disc drive a quoted filename such as "B:tudors" must be given. See Reference Manual for details.

<u>Load</u>

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

```
&. Load fred
```

This loads the program in the CP/M file fred.log of the currently selected drive.

<u>Note</u> - File names and relation names must be kept distinct. You cannot load a file whose name is the same as one of your current relation names, nor can you save a program in a file named by a current relation name. For example, if "likes" is a name of relation, then it cannot be used as the name of a file.

(The system objects with a "CONTROL ERROR" if you try to do this.)

<u>Accept</u>

If a lot of data has to be entered, the Accept command can be used as an aid to entering facts about binary relations. It enables a lot of

simple sentences to be added without using the Add command all the time. The Accept command is used as follows:

```
&.Accept likes
likes.(John Mary)
likes.(John Peter)
likes.End
&.
```

The program prompts for each pair with the name of the relation involved. This serves as a useful reminder of what data is to be entered.

<u>External</u>

For larger programs, it is sometimes useful to have parts of it residing on disk rather than it all being in the computer at one time. The "External" command takes an existing relation, or list of relations, and puts them in a special file on the disk.

Thereafter, instead of micro-PROLOG accessing them from the main memory of the computer, it accesses it from disk thus saving space. This accessing of the disk portion of the program is totally transparent to the rest of the program. There is no need for other rules and queries which <u>use</u> the affected parts of the program to know where they are.

The External command is used as in:

```
External file-name relation-name
```

or

```
External file-name (relation1 .. relationk)
```

<u>Note</u> If you use this facility then the simple program **must** be in the computer whenever you use any relations which have been dumped onto disk.

As an example

```
External likesfile likes
```

will dump all the sentences for the "likes" relation into a CP/M file called likesfile.log. A query condition that uses likes is now answered by a search of this file. The only difference between this search and a search through sentences for the "likes" relation held in the main store is that it is a lot slower. By using "External" you trade time for space.

Before using any external relations it is necessary to issue the micro-PROLOG command:

```
&.OPEN file-name
```

where file-name is the file on which the relations are held. When you have finished using the external relations on the file give the command:

```
&.CLOSE file-name
```

In micro-PROLOG only four files can be open at any one time.

<u>Note</u>: If you do not have a version if the "simple" program numbered 2.12c or later you will have to amend your version to support this command. Appendix C gives the complete listing of the "simple" as used in this PRIMER.

D. <u>Some built-in relations of micro-PROLOG</u>

<u>Relation</u>	<u>Meaning</u>	<u>Restrictions on use</u>

SUM(x y z)	z = x + y	At least two arguments must be known at time of evaluation. Any of the arguments can be the unknown.
PROD(x y z)	z = x * y	Same as for SUM
PROD(x y z z1)	z = x * y + z1	z and one of x or y must be known. Other two arguments must be unknown.
LESS(x y)	x < y	Both arguments must be known. Can only be used for testing.
CON(x)	x is a constant, eg. CON(Sam) true.	x must be known, test only.
NUM(x)	x is a number eg. NUM(-56) true.	Same as CON
EQ(x y)	x identical to y	No restrictions. Defined by the rule EQ(x x). In other words a solution is achieved by making the two arguments identical.

Appendix B

Using the keyboard edit facility

For those users who have micro-PROLOG version 2.12 or later there is a more sophisticated way of changing Logic programs. The simple way of changing a sentence is to "Delete" it and then "Add" the correct version in its place. In version 2.12 of micro-PROLOG there is a new command called "Edit" which allows inplace changes to be made to the sentence.

The "Edit" command has the form:

&.Edit relation-name n

This allows the n^{th} sentence of the named relation to be edited. The old version of the sentence is displayed on the screen, surrounded by a single pair of brackets, with the cursor left underneath the opening bracket. Various editing commands can now be used to change the text of the sentence; these commands are described below. Once the sentence has been corrected to your satisfaction type <return> and the original sentence is replaced. The "Edit" command checks that the predicate symbol of the head of the sentence has not been changed, and that the sentence exists at all.

If the predicate symbol of the head was changed then the command responds with the usual "?".

Edit mode commands

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

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

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

i	Insert. Enters input mode (see below). New text inserted before curser position.
<Return>	Exit. Exits the line editor
<space>	Cursor right. Move 'cursor' one character to the right. The character is echoed to the screen. If already at the end of the line then the bell is sounded instead.
<backspace> or <Rubout>	Cursor left. Move the 'cursor' left one character. A backspace is echoed to the screen. If already at the start of the line then the bell is sounded. Note that unlike in input mode the backspace does not delete the char under the cursor.
s <char>	Search. Searches the keyboard buffer from the current position for the <char>. The charac-

ters between the cursor and the target are printed on the screen. If the <char> is not found then the bell is sounded and the cursor is left at the end of the line.

c <char>	Change. Replaces the character under the cursor with <char>
d	Delete. Deletes the character under the cursor. Characters which are deleted are enclosed in "/"s.
k <char>	Kill. Similar to search, except that the characters between the cursor and the target are deleted. As with delete, the deleted characters are enclosed by "/"s.
l	List. Lists the rest of the line and positions the cursor at the beginning of the line.
p	Print. Toggles the print mode; analogous to the Control-P key when in insert mode. (A "p" will toggle a <Control-P> typed in insert mode). Until the next toggle command all text displayed on the screen is printed on the printer.
x	Extend. This is used to extend the line. The rest of the line is displayed and insert mode is entered.
z	Delete, and extend. This cancels the rest of the line from the cursor position and enters input mode. Useful when retyping a whole line.

Input mode

As characters are typed in they are stored in an internal line buffer and are only passed to the system after the carriage return is pressed. The following control keys have special significance in the input mode:

<Backspace> or <Rubout>	will delete the last character typed in
<Control-P>	toggles the device (as in CP/M)
<Return>	Exits the editor
<Escape>	Echos a "$" and enters **edit mode.**
<Control-Q>	Quotes the next key (ignore key function)

Note that <Control-C> does not have the effect of leaving micro-PROLOG and reentering CP/M. This 'feature' was kindly provided by the CP/M line edit facility and is used by many CP/M programs (including earlier versions of micro-PROLOG); however, it can be very irritating if pressing <Control-C> by mistake causes a lot of work to be lost.

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

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

<Control-E>	Physical end of line

General use of the line editor

In the input mode the "ESC" character causes a transfer to the edit mode. This form of entry to the line editor can be used at any time whilst entering a line of text into micro-PROLOG. Thus suppose we are entering a new sentence using the "Add" command and we have typed:

Add(Petr likes P

At that point we realise that we have mispelled "Petr". We could use back-space to erase back to the "t" and start again from that point. Alternatively we can enter the line editor by typing "ESC". The back-space will now take us back to "r" without losing what we have typed after the error. We then use the edit "i" command to enter the missing "e". Another "ESC" brings us back to the editor, and an "x" command will jump to the last "P" and re-enter the input mode. We can now continue entering the sentence.

Appendix C

The Simple Front end program listing

```
Simple
(Add List Kill Delete Does One Which Save Load Accept Edit
 All Not Is-All For-All External)
(End dict C & and if RPRED)
((version 2.12c))
((Add X)
  (NUM X) / (R Y) (Add X Y))
((Add X) /
  (Add 32767 X))
((Add X Y)
  (parse ((Z|x)|y) Y) (declare Z) (ADDCL ((Z|x)|y) X))
((Edit x)
  (dict x) (R y) (NUM y) (CL ((x|x1)|x2) y y)
  (parse ((x|x1)|x2) X) (RFILL X) (R Y)
  (parse ((x|X1)|X2) Y) (ADDCL ((x|X1)|X2) y) (DELCL x y))
((List X)
  (NOT EQ X ALL) / (List-pred X))
((List ALL)
  (CL ((dict x))) (List-pred x) FAIL)
((List ALL))
((Which (X|Y))
  (is-body (?) Z (?|Y)) (Whichex X Z))
((One (X|Y))
  (is-body (?) Z (?|Y)) (Oneex X Z))
((Does X)
  (is-body (?) Y (?|X)) (IF (? Y) ((PP YES)) ((PP NO))))
((Load X)
  (LOAD X))
((Save X)
  (SAVE X))
((Delete (x|y))/
  (parse z (x|y))
  (OR ((DELCL z))((PP No such sentence))))
((Delete X)
  (CON X) (R Y) (IF (DELCL X Y) () ((PP No such sentence))))
((Kill X) (DELCL X 1) (Kill X))
((Kill X)
  (P Program X deleted) PP)
((Accept X)
  (declare X) (Acceptin X))
((External X)
  (CREATE X)
  (R Y)
  (IF (CON Y)
     ((Ext X (Y) 0))
     ((Ext X Y 0)))
  (CLOSE X))
((Ext x () y)/)
((Ext x (y|z) Y)
  (Exx x y Y Y1)
  (ADDCL ((y|y1) (RPRED x Y Y1 (y|y1))))
  (Ext x z Y1))
((Exx x y Y Y1)
  (DELCL ((y|z)|z1))
  (WRITE x ((y|z)|z1))
  (SEEK x Y2)
  (Exx x y Y2 Y1))
```



```
((Exx x y Y Y))
((parse (X|Y) Z)
  (Atom Z X x) (is-body (if) Y x))
((is-body X () ()))
((is-body X (Y|Z) (x|y))
  (Mem x X) (Literal Y y z) (is-body (and &) Z z))
((Literal X x y)
  (Special-Atom X x y)/)
((Literal X x y)
  (Atom x X y))
((Atom (X ()|Y) (X) Y)
  /)
((Atom (X Y Z|x) (Y X Z) x)
  (CON Y)/)
((Atom (X (Y|Z)|x) (X Y|Z) x))
((Special-Atom (Not|x) (Not y|z) z)
  (is-body (?) x (?|y)))
((Special-Atom (Is-All x (y|z)) (x Is-All (y|Z)|Y) Y)
  (is-body (?) z (?|Z)))
((Special-Atom (For-All x (y|z)) (X For-All(y|Z)|Y) Y)
  (is-body (?) x (?|X))
  (is-body (?) z (?|Z)))
((List-pred X)
  (CL ((X|Y)|Z)) (Rev-parse ((X|Y)|Z) x) (P|x) PP FAIL)
((List-pred X))
((Rev-parse (x|y) z)
  (Atom z x z1)
  (Rev-body y z1 "if
       "))
((Rev-body () () x))
((Rev-body (x|y) (z|Z) z)
  (Literal x Z Z1)
  (Rev-body y Z1 "and
       "))
((Oneex X Y)
  (? Y) (P Answer is X) (R Z) (IF (EQ Z C) (FAIL) ()))
((Oneex|X)
  (PP No (More) answers))
((Whichex X Y)
  (? Y) (P Answer is X) PP FAIL)
((Whichex X Y)
  (PP No (more) answers))
((Acceptin X)
  (P X) (R Y)
  (OR ((EQ Y End))
      ((OR ((EQ (Z x) Y) (ADDCL ((X Z x))))
           ((P What is Y ?)PP))
       (Acceptin X))))
((Mem X (X|Y)) /)
((Mem X (Y|Z))
  (Mem X Z))
((declare x)
  (OR ((CL((dict x))))((ADDCL ((dict x))))))
((Not|X)
  (? X) / FAIL)
((Not|X))
((Is-All X (Y|Z))
  (DELCL ((All-num x))) (SUM x 1 y) (ADDCL ((All-num y)))
  (All-find x X Y Z))
((For-All x (y|z))
  (NOT ?((? z) (NOT ? x))))
```

C. The Simple front end program listing

```
((All-find X Y Z x)
  (? x) (ADDCL ((All-list X Z))) FAIL)
((All-find X Y Z x)
  (Collect X Y))
((All-num 0))
((Collect X (Y|Z))
  (DELCL ((All-list X Y)))/
  (Collect X Z))
((Collect X ()))
CLMOD
```

Program held on backing store evaluator

```
"?random?"
(RPRED)
()
((RPRED x y y1 z)
  ((SEEK x y) (READ x z1) (SEEK x y2)
  (OR ((EQ z z1))
      ((LESS y2 y1) (RPRED x y2 y1 z))))
CLMOD
```

The arithmetic expression evaluator

```
Evaluate
(Val-of * + - md dv)
()
((Val-of X Y)
  (Eval X Y () () 0) /)
((Eval X () (Y Z|x) (y z|X1) Y1)
  (Y z y Z1) (Eval X () x (Z1|X1) Z))
((Eval X () () (X) Y))
((Eval X (Y|Z) x y z)
  (NUM Y) (Eval X Z x (Y|y) z))
((Eval X ((Y|Z)|x) y z X1)
  (Eval Y1 (Y|Z) () () 0) (Eval X x y (Y1|z) X1))
((Eval X (Y|Z) x y z)
  (Op Y X1 Y1) (LESS z X1)
  (Eval X Z (Y z|x) y Y1))
((Eval X (Y|Z) (x y|z) (X1 Y1|Z1) x1)
  (Op Y y1 z1) (NOT LESS x1 y1) (x Y1 X1 X2) (Eval X (Y|Z) z (X2|Z1) y))
((+ X Y Z)
  (SUM X Y Z))
(("-" X Y Z)
  (SUM Y Z X))
((* X Y Z)
  (PROD X Y Z))
((dv X Y Z)
  (PROD Y Z X x))
((md X Y Z)
  (PROD Y x X Z))
((Op + 2 1))
((Op - 1 2))
((Op * 3 2))
((Op md 2 3))
((Op dv 2 3))
CLMOD
```


Appendix D

Answers to Exercises

Chapter 1

Exercise 1-1

```
1.a. &.List is-the-mother-of
     Elizabeth-of-York is-the-mother-of Henry8
     Katherine is-the-mother-of Mary
     Anne is-the-mother-of Elizabeth
     Jane is-the-mother-of Edward
     &.Delete is-the-mother-of 2
     &.Add 1(Catherine is-the-mother-of Mary)
     &.List Female
     Female(Elizabeth-of-York)
     Female(Katherine)
     Female(Mary)
     Female(Elizabeth)
     Female(Anne)
     Female(Jane)
     &.Delete Female 2
     &.Add 1(Female(Catherine))
     &.

  b. &.Add 0(Henry7 is-the-father-of Arthur)
     &.Add 0(Male(Arthur))

2.   Washington-DC capital-of USA
     Ottawa capital-of Canada
     London capital-of United-Kingdom
     Paris capital-of France
     Rome capital-of Italy
     Lagos capital-of Nigeria
     USA country-in North-America
     Canada country-in North-America
     United-Kingdom country-in Europe
     France country-in Europe
     Italy country-in Europe
     Nigeria country-in Africa

3.   Tom-Sawyer written-by Mark-Twain
     For-Whom-The-Bell-Tolls written-by Ernest-Hemingway
     Oliver-Twist written-by Charles-Dickens
     Great-Expectations written-by Charles-Dickens
     Romeo-And-Juliet written-by William-Shakespeare
     Death-Of-A-Salesman written-by Arther Miller
     Macbeth written-by William-Shakespeare
     Tom-Sawyer type Novel
     For-Whom-The-Bell-Tolls type Novel
     Romeo-and-Juliet type Play
     Death-Of-A-Salesman type Play
     Oliver-Twist type Novel
     Great-Expectations type Novel
     Macbeth type Play
     writer(Charles-Dickens)
     writer(William-Shakespeare)
     writer(Arther-Miller)
     writer(Mark-Twain)
     writer(Ernest-Hemingway)
```

Exercise 1-2

1.a. NO
b. YES
c. Answer is Henry8
No (more) answers
d. YES
e. Answer is Edward
No (more) answers
f. Answer is (Henry7 Mary)
Answer is (Henry7 Elizabeth)
Answer is (Henry7 Edward)
No (more) answers

2.a. Does(Katherine is-the-mother-of Edward)
b. Which(x x is-the-father-of y)
c. Does(Jane is-the-mother-of x and Henry7 is-the-father-of x)
d. Which(x Henry8 is-the-father-of x and Katherine is-the-mother-of x)

3.a. Does(Rome is-the-capital-of France)
b. Does(Washington-DC is-the-capital-of x and x country-in Europe)
c. Which(x x capital-of y and y country-in Europe)
d. Does(x is-the-capital-of Italy)
e. Which(x y is-the-capital-of x and x country-in North-America)
f. Which (x y country-in x and z capital-of y)

4.a. NO
b. YES
c. Answer is (Romeo-And-Juliet William-Shakespeare)
Answer is (Macbeth William-Shakespeare)
Answer is (Death-Of-A-Salesman Arther-Miller)
No (more) answers
d. Answer is Oliver-Twist
Answer is Great-Expectations
No (more) answers
e. Answer is Mark-Twain
Answer is Ernest-Hemingway
Answer is Charles-Dickens
Answer is Charles-Dickens
Answer is William-Shakespeare
Answer is William-Shakespeare
Answer is Arther-Miller
No (more) answers

Charles-Dickens and William-Shakespeare are both given twice because each is recorded as having written two things. In answering the query

Which(x y written-by x)

micro-PROLOG finds all the sentences of the form "y written-by x" and for each one it finds it gives us the 'x'.

Exercise 1-3

1.a. YES
b. Answer is 22
No (more) answers
c. Answer is 17
No (more) answers
d. YES

e. YES
f. Answer is 63
No (more) answers
g. NO
h. Answer is (3 2)
No (more) answers

2.a. Which(x SUM(9 7 x))
b. Which(x PROD(y 7 65 x))
c. Which(x SUM(29 53 y) and PROD(x 2 y z))
d. Does(PROD (x 5 93))
e. Does(PROD (17 3 x) and x LESS 50)

Exercise 1-4

1.a. Which(x x location (y z) and London location (X Y) and X LESS y)
b. Which(x x location (y z) and Rome location (X Y) and Y LESS z)
c. Does(x country-in Europe and y capital-of x and
y location (z X) and Rome location (Y Z) and
London location (x1 y1) and Y LESS z and z LESS x1)
d. Which(x x country-in Europe and y capital-of x and
y location (z X) and London location(Y Z) and X LESS Z)
e. Which((x y) x country-in y and z capital-of x and
z location (X Y) and Rome location (Z x1) and
X LESS Z and x1 LESS Y)

2.a. Which(x Apple costs y and Wallet contains z and PROD(x y z X))
b. Does(Bread costs x and Cheese costs y and Wallet contains z and
SUM(x y X) and X LESS z)
c. Which(x Wallet contains y and Cheese costs z and Apple costs X and
SUM(z X Y) and SUM(x Y y))
d. Which(x Apple costs y & Bread costs z & PROD(y 5 X) & PROD(z 3 Y)
& SUM(X Y Z) & Wallet contains x1 & SUM(x x1 Z))

3.a. Does(Oliver-Twist published 1850)
b. Which(x x published 1623)
c. Which(x Tom-Sawyer published x)
d. Does(Oliver-Twist published x and Great-Expectations published x)
e. Does(Macbeth published x and Romeo-And-Juliet published y and x LESS y)
f. Which(x x published y & For-Whom-The-Bell-Tolls published z & y LESS z)
g. Does(x published y and y LESS 1600)

Chapter 2

Exercise 2-1

1.a. x is-maternal-grandmother-of y if x is-the-mother-of z and
z is-the-mother-of y
b. x is-a-grandparent-of y if x is-a-parent-of z and z is-a-parent-of y
c. x is-a-grandchild-of y if y is-a-grandparent-of x

2.a. x city-in Europe if x capital-of y and y country-in Europe
b. x North-of London if x location (y z) and London location (X Y) and
X LESS y
c. x West-of y if x location (z X) and y location (Y Z) and Z LESS X)
3.a. x classified-as fiction if x type Novel
x classified-as fiction if x type Play
b. x isa classic if x written-by William-Shakespeare
x isa classic if x written-by Charles-Dickens
c. x is contemporary-literature if x published y and 1900 LESS y

Exercise 2-2

1.a. x is-grandfather-of y if x is-the-father-of z and z is-a-parent-of y
b. x is-grandmother-of y if x is-the-mother-of z and z is-a-parent-of y

2.a. Answer is Henry7
Answer is Henry8
Answer is Henry8
Answer is Henry8
Answer is Elizabeth-of-York
Answer is Katherine
Answer is Jane
Answer is Anne
No (more) answers
b. Answer is Mary.
c. YES
d. Answer is Katherine
Answer is Jane
Answer is Anne
No (more) answers

3.a. Which(x y is-the-father-of Edward and x is-the-mother-of y)
b. Which(x y is-a-grandchild-of Henry7 and x is-the-mother-of y)
c. Does(x is-a-child-of Katherine and Male(x))
d. Which(x y is-a-child-of Henry8 and Male(y) and x is-the-mother-of y)

4.a. Which(x x city-in Europe)
b. Does(x North-of London)
c. Which(x x North-of London and x West-of Rome)

5.a. Which(x x isa classic)
b. Which(x y written-by x and y published z and z LESS 1900)

Exercise 2-3

1.a. Answer is (Edward is male grandchild of Henry7)
Answer is (Edward is male grandchild of Elizabeth-of-York)
No (more) answers
b. Answer is (Katherine is a wife of Henry8).
c. Answer is Henry8
Answer is Jane
Answer is Henry7
Answer is Elizabeth-of-York
No (more) answers
d. Answer is Henry8
Answer is Mary
Answer is Elizabeth
Answer is Edward
No (more) answers
e. NO
f. Answer is Mary
Answer is Elizabeth
No (more) answers

3.a. x greater-than y if y LESS x
b. x greateq y if y lesseq x
c. z divisible-by x if PROD(x y z)

4.a. Nineteenth-Century-Author(x) if y written-by x and y published z and
1800 lesseq z and z LESS 1900

b. Contemporary-Playwright(x) if y written-by x and y type Play and
y published z and 1900 lesseq z
c. x available-at y if x published z and z lesseq y
d. Which(x x available-at 1899)
e. Which(x x written-by y and Nineteenth-Century-Author(y) and
x available-at 1980)

Chapter 3

Exercise 3-1

1.a. NO
b. Answer is (Tom Dick Harry)
No (more) answers
c. Answer is (Jane Janet Julia)
No (more) answers

2.a. Answer is (Wimbledon Morden Mitcham)
Answer is (Hampton Teddington Ham)
Answer is (Surbiton Norbiton)
No (more) answers
b. YES
c. Answer is Merton
Answer is Richmond
Answer is Kingston
No (more) answers
d. NO

3. (Oliver Twist) written-by (Charles Dickens)
(Great Expecations) written-by (Charles Dickens)
(Macbeth) written-by (William Shakespeare)
.
.
.
.

Exercise 3-2

1. Childless-wife(x) if x mother-of-children ()

2.a. Answer is Jane
No (more) answers
b. No (more) answers
c. YES
d. Answer is Henry
Answer is Henry
Answer is Bill
Answer is Paul
No (more) answers
e. Answer is (Henry father Sally mother Margaret child Bob child)
Answer is (Paul father Jilly mother John child Janet child)
No (more) answers
f. Answer is (John Janet)
No (more) answers

3.a. Answer is Dickens
No (more) answers
b. YES
c. Answer is ((Tom Sawyer) Twain)
No (more) answers
d. Answer is ((William Shakespeare) was a great playwright)
No (more) answers

e. Answer is Tom
   Answer is Oliver
   Answer is Great
   No (more) answers

Exercise 3-3

1.a. x=A; y=B; z=C; Z=(D E)
  b. x=A; y=B; z=C; Z=(D)
  c. x=A; y=B; z=C; Z=()
  d. No match
  e. No match
  f. No match

2. x=(C|y); y=(A B) i.e. x=(C A B)

3.a. Answer is (District Circle Northern)
     No (more) answers
  b. YES
  c. Answer is (Hackney Lambeth Richmond Kingston)
     No (more) answers
  d. Answer is (Hackney Richmond)
     No (more) answers
  e. YES

Exercise 3-4

1.a. Answer is (English French)
     No (more) answers
  b. Answer is English
     Answer is English
     No (more) answers
  c. Answer is English
     Answer is Welsh
     Answer is Gaelic
     No (more) answers
  d. YES
  e. British-language(x) if y spoken-in United-Kingdom and
        z spoken-in Canada and x belongs-to y and x belongs-to z
  f. Minor-language(x) if (y|z) spoken-in X and x belongs-to z

2.a. Answer is 0
     Answer is 8
     Answer is 8
     No (more) answers

  b. YES

3.a. x is-a-parent-of-children y if z parents-of y and x belongs-to z
  b. x is-a-child-of y if z parents-of X and x belongs-to X and
        y belongs-to z

Exercise 3-5

1. x mother-of-children-number y if x mother-of children z and
        z has-length y
   Which(x Jilly mother-of-children-number x)

2.a. Which(x y parents-of z and x belongs-to y and z has-length 5)

  b. Which(x 5 length-of X & y parents-of X and x belongs-to y)

3.a. Answer is 4
     Answer is 3
     No (more) answers
  b. Answer is 3
     No (more) answers
  c. YES

4. One(x 2 belongs-to x)
   Answer is (2|X).C
   Answer is (X 2|Y).C
   Answer is (X Y 2|Z).C
   .
   .

Exercise 3-6

1.a. Which(x (Arthur Robert) have-descendant-chain x)
     Answer is (Peter)
  b. Which(x (Jane Robert) have-descendant-chain y & y has-length x)
     Answer is 2
  c. Which((x y) (x y) have-descendant-chain (z))

2. x is-a-great-grandparent-of y if (x y) have-descendant-chain (z1 z2)

Exercise 3-7

1. Which(x y Is-All(z Peter is-a-parent-of z and Male(z)) and
        y has-length x)

2.a. Which(x x Is-All(y y family Tudor))
     Answer is (Henry7 Elizabeth1)
  b. Which(x y Is-All(z z family Z) and y has-length x)
     Answer is 4
  c. Which(x y Is-All(z z family Stuart) and y has-length x)

3. x last-of (x)
   x last-of (y|z) if x last-of z

4. (x y) adjacent-on (x y|z)
   (x y) adjacent-on (z|X) if (x y) adjacent-on X

Chapter 4

Exercise 4-1

1.a. Even(x) if PROD(y 2 x)
  b. Odd(x) if NUM(x) & Not(Even(x))
2.a. Answer is the
     Answer is quick
     Answer is fox
     No (more) answers
  b. Answer is (F E)
     No (more) answers

3.a. a-man-with-no-sons(x) if male(x) & Not(x is-the-father-of y & Male(y))
  b. a-mother-with-no-daughters(x) if x is-the-mother-of y &
        Not(x is-the-mother-of z & Female(z))

4.a. Overdue(x) if Issue(y x z X Y) and Not(Return (y x z Z)) and
        Today date x1 and x1 after Y

b. (x y z) after (X Y Z) if Z LESS z
(x y z) after (X Y Z) if Y LESS y
(x y z) after (X y z) if X LESS x
c. Banned(x) if Issue(x y z X Y) and Overdue(y)

Exercise 4-2

1. x union-of (y z) if x Is-All(X X member-of-either (y z))

2. x subset-of y if z intersection-of (x y)
& z1 difference-between (x z) & z1 EQ ()

3. x set-union-of (Y Z) if X1 intersection-of (Y Z)
& X2 difference-between (Y Z)
& X union-of (X1 X2)

Exercise 4-3

1. (i) Novelist(x) if author(x) & (y type Novel) For-All (y y written-by x)
(ii) Modern-author(x) if author(x) & (1900 lesseq y & y LESS 2000) For-All (y Z written-by x & Z published y)

2. (i) Positive-nums(x) if (0 LESS y) For-All(y y belongs-to x)
(ii) all-Male(x) if (Male(y)) For-All(y y belongs-to x)

3. (i) disjoint(X Y) if Not(x belongs-to X & x belongs-to Y)
(ii) disjoint(X Y) if () Is-All(x x belongs-to X & x belongs-to Y)
(iii) disjoint(X Y) if (Not(x belongs-to X)) For-All(x x belongs-to Y)

Chapter 5

Exercise 5-1

1. Answer is (J U M B O)
No (more) answers
2. Answer is (() (J O H N))
Answer is ((J) (O H N))
Answer is ((J O) (H N))
Answer is ((J O H) (N))
Answer is ((J O H N) ())
No (more) answers
3. Answer is ((C Y) (I L))
No (more) answers
4. Answer is ((D A M S O N) 6)
No (more) answers
5. Answer is (( ) X X). C
Answer is ((X) Y (X|Y)). C
Answer is ((X Y) Z (X Y|Z)). C
Answer is ((X Y Z) x (X Y Z|x)). F
6. Which(x (x x) appends-to (2 3 4 2 3 4))
Answer is (2 3 4)
No(more) answers
7. Which((the|y)) (x (the|y)) appends-to (the man closed the door of the house))
Answer is (the man closed the door of the house)
Answer is (the door of the house)
Answer is (the house)
No(more) answers

8. Which((y|z) y belongs-to(a the) & (x (y|z)) appends-to (Sam threw a ball into the lake))
Answer is (a ball into the lake)
Answer is (the lake)
No(more) answers
9. Which(y (x (y)) appends-to (2 3 4))
Answer is 4
No(more) answers
10.(a) split-on(x X X1 X2) if (X1 X2) append-to X & X1 has-length x
(b) split-on(0 X () X)
split-on(y (x|X) (x|X1) X2) if 0 LESS y & SUM(y1 1 x)
& split-on(y1 X1 X2)

(b) is much more efficient.
11. remove-all(x () ())
remove-all(x (x|X) Y) if remove-all (x X Y)
remove-all(x (y|X) (y|Y)) if Not(x EQ y) & remove-all(x X Y)
12. () compacts-to ()
(x|X) compacts-to (x|Z) if remove-all(x X Y) & Y compacts-to Z

Exercise 5-2

1.a. Answer is (J K L M)
No (more) answers
b. Answer is (F)
Answer is (F R)
Answer is (F R E)
Answer is (F R E D)
Answer is (F R E D A)
Answer is (R)
Answer is (R E)
Answer is (R E D)
Answer is (R E D A)
Answer is (E)
Answer is (E D)
Answer is (E D A)
Answer is (D)
Answer is (D A)
Answer is (A)
No (more) answers
c. Answer is (C I R E)
No (more) answers
2. y last-of z if (x (y)) appends-to z
3. x power-list (()|y) if y Is-All(z z segment-of x)
4. y belongs-to z if (x (y|Y)) appends-to z
5. palindrome(x) if x reverse-of x
6. (x y) adjacent-on Z if (X (x y|X1)) appends-to Z
7. delete(x (x|X) X)
delete(x (y|X) Y) if delete(x X Y)

Exercise 5-3

1. (x1 x2|X) quick-sort y if
partition((x2|X) x1 y1 y2) and
y1 quick-sort Y1 and
y2 quick-sort Y2 and
(Y1 (x1|Y2)) appends-to y

2. partition(() X () ())
partition((x|y) X (x|y1) y2) if
x LESS X and

```
     partition(y X y1 y2)
partition((x|y) X y1 (x|y2)) if
     Not(x LESS X) and
     partition(y X y1 y2)
```

3.
```
(0 ()) merge-sort()
(1 (x) merge-sort(x)
( y X) merge-sort Z if 1 LESS y
                      & merge-split((y X) Y1 Y2)
                      & merge-sort(Y1 Z1)
                      & merge-sort(Y2 Z2)
                  y1
                      & merge(Z1 Z2 Z)
merge-split((y X) (Y1 X1) (y2 X2))
            if PROD(2 y1 y y3) & SUM(y1 y3 y2)
             & split-on(y1 X X1 X2)
```

plus the old rules for "merge" and "split-on".

To sort using this program, we use a query such as

```
One(x (6 (4 3 6 100 -5 3) merge-sort x)
```

in which the length of the list to be sorted is also given.

Exercise 5-4

1. a.
```
(S (NP (DT the)
       (NE (A sad)
           (N boy)))
   (VP (V likes)
       (NP (DE a)
           (NE (A happy) (N girl)))))
```

b.
```
(S (NP (DT the) (N ball))
   (VP (V kicked)
       (NP (DT the) (N boy))))
```

c.
```
(S (NP (DT a) (NE (A lonely) (N man)))
   (VP (V wandered)
       (NP (DT the) (N hills))))
```

d.
```
(S (NP (DT a) (N piper))
       (VP (V plays) (NP (DT a) (N tune))))
```

2. The extension needed is:

```
x is-verb-expression (ADV y z) if
  (x1 x2) appends-to x and
   x1 is-adverb y and
   x2 is-verb-expression z

(and x) is-adverb (AD x) if
   x dictionary ADVERB

(x) is-adverb (AD x) if
   x dictionary ADVERB

slowly       dictionary ADVERB
deliberately dictionary ADVERB
```

Chapter 7

Exercise 7-1

1.
```
pair ( () () () )
pair ( (x|y) (X|Y) ((x X)|Z)) if
     pair (y Y Z)
```

2.
```
dot (x y z) if
     pair (x y Z) and
     reduce (sumprod Z z)
sumprod ((x1 x2) y z) if
     PROD (x1 x2 x3) and
     SUM (x3 y z)
```

3.
```
((has-val x x)
     (NUM x))
((has-val (x y z) Y)
     (has-val x X)
     (has-val z Z)
     (y X Z Y))
```

Exercise 7-2

```
((Owh (x|y))
     (? y)
     (P Answer is x)
     (R z)
     (IF (EQ z C) ((FAIL)) ((PP Finished))))
((Owh x)
     (PP No (more) answers))
```

Exercise 7-3

1.
```
((One-of|x)
     (x /)
```

2.
```
((apply x y)
     (x|y))
```

Bibliography

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., [1979] Predicate Logic as a Computational Formalism. Research Report, Dept. of Computing, Imperial College.

Clark,K.L., McCabe,F., [1979] Control facilities of IC-PROLOG. Expert systems in the Microelectronic Age, (ed D.Michie) Edinburgh Univ.Press.

Clark,K.L., McCabe,F., Gregory,S., [1980] IC-PROLOG Language Features. Research Report, Dept. of Computing, Imperial Coll. To appear in Logic Programming (eds. Clark & Tarnlund), Academic Press, 1982.

Clark,K.L., McCabe,F., [1980] PROLOG : A Language for implementing Expert Systems. DOC, Imperial College. To appear in Machine Intelligence 10, (eds. Hayes & Michie), Ellis Horwood, 1982.

Clocksin,W., Mellish,C., [1981] Programming in PROLOG, Springer-Verlag, New York.

Coelho,H., Cotta,J.C., Pereira,L.M., [1980] How to Solve it with PROLOG. Laboratorio Nacional de Engenhario Civil, Lisbon.

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

Darvas,F., Futo,I., Szeredi,P., [1980] Logic based program for predicting drug interactions. Int.J.Biomedical Computing, Logic Programming Workshop 1980 (ed. S-A Tarnlund).

Deliyanni,A.J., [1976] A Comparative Study of Semantic Networks and Predicate Logic. M.Sc.Thesis, CCD, Imperial College.

Deliyanni,A.J., Kowalski,R.A., [1979] Logic and Semantic Networks. Comm. ACM 22 3 (March 1979) pp184-192.

Ennals,J.R., [1981] Logic as a Computer Language for Children: A One Year Course. DOC 81/6, Imperial College.

Ennals,J.R., [1981] Children Program in PROLOG, DOC 81/8, Imperial College.

Ennals,J.R., [1981] PROLOG can link diverse subjects with logic and fun, (Logic and Computing for Schools), Practical Computing, March 1981.

Hammond,P., [1980] Logic Programming for Expert Systems. M.Sc.Thesis, Imperial College.

Hodges,W., [1977] Logic. Penguin, London.

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

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

Kowalski,R.A., [1978] Logic for Data Description. Logic and Data Bases, Plenum Press, New York.

Kowalski,R.A., [1979] Algorithm = Logic + Control. CACM, August 1979.

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

Kowalski,R.A., [1980] Position Statement. SIGART Newsletter Feb 1980 No.70. Issue on Knowledge Representation (eds Brackman & Smith).

Markusz,Z., [1977] How to design variants of flats using programming language PROLOG, based on mathematical logic. Proc.IFIP.

Mayaramani,S., [1979] A Deductive Database. B.Sc.(Eng), Imperial College.

McCabe,F., [1981] Micro-PROLOG Programmers Reference Manual, Logic Programming Associates.

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

Pereira,L., Pereira,F., Warren,D., [1978] User's guide to DEC system 10 PROLOG. Dept AI University of Edinburgh.

Polya,G., [1946] How to Solve It. Princeton University Press.

Ritchie,K.E., [1977] Query System for Database. B.Sc.(Eng),Imp.Coll.

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. The Mechanization of Deductive Reasoning, Edinburgh Univ.Press.

Robinson,J.A., [1979] The Logical Basis of Programming by Assertion and Query. Expert Systems in the Microelectronic Age (ed. Michie), Edinbrugh University Press.

Roussel,P., [1975] PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

Santane-Toth,E., Szeredi,P., [1980] PROLOG applications in Hungary. Institute of Coordination of Computer Techniques, Budapest. Logic Programming Workshop (ed. S-A Tarnlund)

Sergot,M., [1980] Programming Law: LEGOL as a Logic Programming Language, DOC, Imperial College.

Silva,G., et al (Operating Systems Inc) [1979] A Knowledge-based automated message understanding methodology for an advanced indications system. OSI-R79-006, 14.2.79, Woodland Hills, California.

Skuce,D.R., [1979] An Approach to Defining and Communicating the Conceptual Structure of Data. Report to Systems Development Division of Statistics, Canada. Department of Computer Science, University of Ottawa.

Swinson,P.S.G., [1980] Prescriptive to Descriptive Programming: a way ahead for CAAD. Department of Architecture, University of Edinburgh. Logic Programming Workshop 1980 (ed. S-A Tarnlund).

Warren,D., [1979] PROLOG on the DEC-System 10 in Expert Systems in the Microelectronic Age (ed. D.Michie), Edinburgh University Press.

Warren,D., [1981] Efficient Processing of Interactive Relational Database Queries Expressed in Logic. DAI Research Paper No.156, Univ.of Edinburgh.

Warren,D., Pereira,F.C.N., [1981] An Efficient Easily Adaptable System for Interpreting Natural Language Queries, DAI, University of Edinburgh.