PROLOG TUTORIAL

Before you start

The first thing you should do with any new piece of software is to take a backup. For a full account of this, refer to your OS9 operating system user's guide. It is recommended that you backup the Prolog diskette to a new diskette, then keep the original Prolog diskette in a safe place, hopefully never to be used again. Then backup the first copy, and use the first copy as a master diskette - again, keep it in a safe place and avoid using it except in emergencies. The second copy is your working copy.

The rest of this tutorial assumes that you have a working knowledge of the OS9 operating system. You should be familiar with OS9 directories, the OS9 shell, and you should be able to refer to the OS9 manual to use OS9 utilities.

Getting started

The diskette on which Prolog is supplied contains a 'cmds' directory where the interpreter itself is located. To run the interpreter, make this directory the current execution directory using a command such as:

```
chx /d0/cmds
```

The Prolog interpreter is held in a file called 'prolog'. To run the interpreter, use the command:

```
prolog
```

After a short delay while the interpreter is loaded from disk into memory, you will see a sign-on message showing details of revision level and copyright. After this you will see the prompt 'P>'. This is the Prolog prompt which is displayed when Prolog is awaiting input from the terminal. At this stage you can leave Prolog and return to OS9 by typing:

```
QUIT();
```

You must type QUIT in capitals - Prolog distinguishes between upper and lower case letters (but note that Prolog has been designed so that it can be used with an uppercase-only terminal). The semicolon (;) is necessary to tell Prolog that the command has finished; if omitted, Prolog will just wait for more terminal input. You should now see the 'OS9' prompt again.

Simple relations

The most important notion in Prolog is that of a relation, also known as a predicate. A relation applies to a group of specific items and expresses the fact that there is some general logical relationship between them. The notation for a relation is the name of the relationship followed by a list of specific items in parentheses. For example:

```
father-of(Joe,Sam)
```

is the notation for expressing the fact that Joe is the father of Sam. The general logical relationship expressed is father-of. The word general is important because it implies that this relationship may be true for other sets of data, for instance:

```
father-of(Sam,Joanna)
```

which expresses the fat that Sam is the father of Joanna. The same name (father-of) is used for the relation, but different items appear inside the brackets. The items inside the brackets are referred to as arguments. A relation may have any number of arguments, including zero or one. For example:

```
today-is-thursday()
```

expresses the fact that today is thursday but no arguments are taken because the assertion "today is thursday" can be viewed as a simple fact which does not relationships between objects. As an example of a relation with only one argument:

```
dog(Rover)
```

expresses the fact that Rover is a dog. Relations with one argument are called unary relations. Unary relations usually assert a simple property of their argument, for instance:

```
red(my-house)
```

expresses the fact that my house is red. Relations may take as many arguments as required. For example:

```
meeting(Andrew,Jenny,"10:30",Conference-Room,"Monday, 5th August 1986)
```

expresses that Andrew has a meeting with Jenny at 10:30am in the conference room on Monday the 5th of August, 1986. Another meeting could be:

```
meeting(Jenny,Clive,"4:30pm",Lecture-Room-B,"Tuesday, 7th September 1986")
```

which has an equally transparent meaning.

In Prolog, a relation can be asserted simply by typing it in using the notation illustrated below:

```
colour(my-car,red):-;
```

In the following examples of Prolog dialogues, the P> prompt is shown to indicate that the characters

following are types by the user.

```
P>colour(my-car,red):-;
P>colour(fred's-car,blue):-;
P>colour(joan's-car,green):-;
P>colour(harry's-car,orange):-;
```

Prolog automatically remembers these relations. The information can be retrieved using a query. To find out what colour fred's-car is, use the query:

```
P>colour(fred's-car,?colour);
<?colour="blue">
```

Prolog solves the query and prints the answer inside the angle brackets (<>). The object "?colour" is a Prolog variable. Prolog solves the query by finding what value the variable ?colour must take so that the query matches a known fact. In this case, when the variable ?colour takes the value "blue", the query matches the relation

```
colour(fred's-car,blue)
```

A different type of query can be given:

```
P>colour(?which-car,green);
<?which-car="joan's-car">
```

The query is equivalent to the question "which car is green?". Prolog solves the query using the same method as before, finding the value that the variable must take in order for the query to match a known fact. Prolog queries can be formulated with any number of variables. For example:

```
P>colour(?car,?colour);
<?colour="red",?car="my-car">

<?colour="blue",?car="fred's-car">

<?colour="green",?car="joan's-car">

<?colour="orange",?car="harry's-car">
```

Now Prolog displays four sets of answers - there are, of course, four pairs of values for ?car and ?colour for which the query matches a known fact, and Prolog prints all of them. Now define the following relations:

```
P>make(my-car,Ford):-;
P>make(fred's-car,Vauxhall):-;
P>make(joan's-car,Vauxhall):-;
P>make(harry's-car,Ford):-;
```

To ask the question "which cars are Fords?":

```
P>make(?car,Ford);
<?car="my-car">
```

```
<?car="harry's-car">
```

To ask the question "which car is a blue Vauxhall?":

```
P>colour(?car,blue),make(?car,Vauxhall);
<?car="fred's-car">
```

In this case the query consists of two parts. Only values of ?car for which both parts of the query are true are printed. Queries do not always have answers - consider:

```
P>colour(?car,green),make(?car,Ford);
```

ie "which car is a green Ford?". Prolog prints no answers and responds with another P> prompt - there is no green Ford. Any number of predicates (relations) may be specified in a query. They must be separated by commas (,) and the list must be terminated by a semicolon (;). The only answers printed will be those for which all the separate relations in the query are simultaneously true.

Rules
=====

The programs we have looked at so far consist of simple facts and queries. Prolog allows a fact to be extended by the addition of conditions, making it into a rule. An example of a rule is:

```
father-of(?x,?y):-
      parent-of(?x,?y),
      male(?x);
```

The predicate before the ':-' symbol is the fact that the rule asserts, and the predicates following it are the conditions which must hold for it to be true. Returning briefly to the simple fact:

```
colour(my-car,red):-;
```

Note that this fact is really a rule which has no conditions, that is, it is always true. The semicolon simply serves to mark the end of the list of conditions. In the father-of() predicate, the rule can be read as: "x is the father of y if x is a parent of y and x is male". There is always such an English translation of a Prolog rule of the form "P is true if Q and R and ... and T" where P is the part before the ':-' sign, sometimes called the head of the rule, and Q, R, ..., T are the predicates after the ':-' sign, collectively known as the body of the rule. Suppose we have a few simple rules and facts (sometimes called a rule base):

```
parent-of(Henry,Paul):-;
parent-of(Joan,Paul):-;
parent-of(Paul,Phillip):-;
parent-of(Susan,Phillip):-;
parent-of(Paul,Claire):-;
parent-of(Susan,Claire):-;

male(Henry):-;
male(Paul):-;
male(Phillip):-;

female(Joan):-;
female(Susan):-;
female(Claire):-;
```

Then the goal:

```
father-of(?x,?y);
```

ie. "for which x and y is x the father of y?". The answers given by Prolog are:

```
<?y="Paul",?x="Henry">

<?y="Phillip",?x="Paul">

<?y="Claire",?x="Paul">
```

Note that Paul appears twice as a father. The way that Prolog goes about solving such a query is just an

extension of the way it solves the simpler queries. Given the goal:

```
father-of(?x,?y);
```

Prolog sees that it can match the goal to a known fact in the database (the rule for father-of()) provided that it can solve the compound query resulting from the conditions in the body of the rule. So the goal becomes:

```
parent-of(?x,?y),male(?x);
```

which is easily solved by reference to simple facts in the database. The process of expanding and substituting goals can be carried on to as many levels as necessary - for instance, if either parent-of() or male() were rules with attached conditions, they would have been expanded in turn and solved. To illustrate this, we introduce another rule, ancestor-of(). The definition of this rule is:

```
ancestor-of(?x,?y):-
      parent-of(?x,?y);

ancestor-of(?x,?y):-
      parent-of(?x,?z),
      ancestor-of(?z,?y);
```

This rule needs some explanation as it introduces some tricky concepts. The predicate has two parts. Any number of rules may be used in Prolog to define a predicate. When there are several rules, they are viewed as alternatives. If, as Prolog attempts to solve a query, it is found to be impossible using one of the alternatives, the interpreter discards the work done using that alternative and proceeds to try the next. The alternatives are always tried in the order in which they are found in the database, which is the order in which they are entered. The English translation of the ancestor-of() rule is then: "x is an ancestor of y if x is the parent of y, or x is the ancestor of y if x is the parent of z and z is an ancestor of y". In other words, when translating a predicate into English when it has several alternative rules (or "clauses"), the translation is simply the translation of each separate clause linked to the next by the word "or".

The next tricky aspect of the ancestor-of() rule is the fact that it is defined in terms of itself (ie an ancestor-of() appears in one of the conditions for a rule that defines ancestor-of()). In this case, it is reasonably clear in the English translation what is meant by this "recursive" definition. Prolog will expand the ancestor-of() in the body whenever it comes to it by using the rule base, ie it will substitute in the goal the body of one of the clauses for ancestor-of(). Note that this process will not continue forever because the first clause, which does not contain a recursive reference to ancestor-of() is tried first, and if this succeeds there is no need for Prolog to proceed to the next.

It is essential that the Prolog programmer becomes familiar with the technique of recursion, not only because it is a very powerful technique (and more general than a simple "flat" loop), but also because it is the only method of looping available in Prolog. Consider how you might define a rule to find the successors of someone (think about the English formulation of the problem: "x is a successor of y if x is a child of y, or x is a successor of y if z is a child of y and x is a successor of z").

Lists and trees

So far, data types have been confined to fixed symbols such as 'my-car', 'red' and so on. These symbols are known as constants. Prolog constants share many properties with character string types of other languages. Prolog has one other simple data type, the number. Prolog only deals with whole numbers, or integers, in the range -32768 to 32767. Numbers behave very much like constants, but there are special built-in operations that can be used to perform arithmetic on them. These built-in rules are described under "Predefined Rules". Numbers are always written in decimal, and may be preceded by a minus sign if they are negative (though not by a plus sign).

Prolog has one compound data structure, the list. Lists are built from constants, variables and numbers using the list constructor operator, written as a dot (.). The list constructor is a binary operator. Some examples of lists made using the list constructor are:

```
a.b
1.2
?x.?y
a.3
1.?z
```

The list constructor has the property that it is right-associative, that is to say that:

```
X.Y.Z is taken to mean X.(Y.Z)
```

The list constructor is not commutative, ie:

```
X.Y is not equal to Y.X
```

The list constructor is not associative, ie:

```
(X.Y).Z is not equal to X.(Y.Z)
```

Prolog identifies one constant as being different from all others. This constant is 'nil' which may be thought of as the empty list. Using this convention:

```
nil
```

is a list containing no elements,

```
mat.nil
```

is a list containing the single element "mat",

```
the.mat.nil
```

is a list containing the two elements "the" and "mat",

```
the.cat.sat.on.the.mat.nil
```

is a list containing six elements. Note that lists may be nested to any level, for example:

```
the.(fat.cat.nil).sat.on.the.(green.mat.nil).nil
```

is a list containing six elements, and two of these elements are themselves lists containing two elements each. Note that lists may contain variables, for example:

```
the.cat.sat.?x
```

If the variable x later became bound to the value:

```
on.the.mat.nil
```

then the original list would become:

```
the.cat.sat.on.the.mat.nil
```

Consider now a rule which deals with lists. Recursion is a universal tool in Prolog for dealing with lists, the usual scheme being to deal with one element (the first element, or "head" of the list) then to make a recursive call to deal with the remainder of the list (the part left when the head is deleted, or the "tail"). isin() is a simple rule which tests whether a list contains a particular element, so that:

```
isin(sat,the.cat.sat.on.the.mat.nil)
```

succeeds, whereas:

```
isin(sat,the.cow.jumped.over.the.moon.nil)
```

fails. To set out the problem informally, an element is in a list if it is the head of that list. Alternatively, an element is in a list if it is in the tail of the list. From this summary, it is easy to write down the rules for isin(). The first part gives the rule:

```
isin(?x,?x.?y):-;
```

which simply says that ?x is in the list ?x.?y (because it is the head of the list). The second part of the informal statement suggests the rule:

```
isin(?x,?y.?z):-isin(?x,?z);
```

which says that ?x is in the list ?y.?z if ?x is in the list ?z. Experiment with isin(). When you have checked the examples above, try the following query:

```
P>isin(?x,the.cat.sat.on.the.mat.nil);
```

Prolog will print as its answers for the value of ?x the elements of the list, one at a time. This is because Prolog treats the query as equivalent to the question "for which x is x in the list the.cat.sat.on.the.mat.nil?". There are, of course, six answers, and Prolog finds and prints all of them.

Unification

When Prolog is attempting to solve a query, at any one time, it has a goal predicate, for example:

```
parent-of(?x,Jo)
```

which it is trying to match with the head of a rule. If there is a rule:

```
parent-of(Mary,Jo):-;
```

in the rule base a match is made. Prolog actually tests for a match using a process known as unification. When two terms are unified, any variables appearing in them are given values which will make the terms equal. Prolog tries to unify each of the arguments in the goal predicate with the corresponding term in the head of the rule. For example:

```
?x unifies with Mary giving ?x the value Mary
Jo unifies with Jo but no variables are set
```

Prolog can unify quite complex terms built using the list constructor, for example:

```
?x.(a.?y.(?z.?z)).b
```

unifies with:

```
(i.j.nil).(?u.nil.(k.k)).?v
```

and as a result, the following bindings are made:

```
?x = i.j.nil
?y = nil
?z = k
?u = a
?v = b
```

It is important to note that if a variable is bound to another variable, for instance if ?x is unified with ?y then any later reference to one is the same as a reference to the other, and if one is later given a fixed value then both effectively have that value. For example:

```
P>EQ(?x,?y),EQ(?y,hello.world.nil);
<?y="hello"."world".nil,?x="hello"."world".nil>
```

In this example, the predefined rule EQ() is used, which simply unifies its two arguments.

Further examples of the use of lists

The list data structure is extremely flexible. It is worth noting that the Prolog interpreter itself uses lists extensively - programs are stored as lists of clauses, clauses are stored as lists of predicates, predicates are stored as lists of terms where the first term is the name of the predicate and the tail is a list of the arguments. When you type in a predicate, rule or goal, the input analyser immediately converts it into a list. This is very useful because it allows a different notation for lists which is sometimes more appropriate. For example:

```
a(b,c)
```

is equivalent to the list:

```
a.b.c.nil
```

Such a term is called a prefixed term, and is useful for representing trees. For example, the expression:

```
a+b*c
```

could be represented for the purposes of symbolic processing as:

```
plus(a,times(b,c))
```

Internally this is identical to:

```
plus.a.(times.b.c.nil).nil
```

There is another external representation of lists recognised by Prolog called a tuple. The tuple would normally be used for fixed length lists and avoids the necessity to always terminate the list with 'nil'. Tuples are lists of terms separated by commas and surrounded by angle brackets (<>). For example:

```
<a,b,c>
```

which is equivalent to the list

```
a.b.c.nil
```

Prefixed terms and tuples are recognised on input as a convenience to the programmer but are never output by Prolog's low-level output routines. It is, of course, possible to write Prolog programs which do output prefixed terms and tuples. For example:

```
print-tuple(?x):-
      W("<"),
      print-tuple-args(?x),
      W(">");

print-tuple-args(nil):-;

print-tuple-args(?x.?y):-
      /,
```

```
        WQ(?x),
        WQ(","),
        print-tuple-args(?y);

print-tuple-args(?x.nil):-
        WQ(?x);
```

print-tuple() can be used to print lists in tuple format, eg:

```
P>print-tuple(tom.dick.harry.nil);
<"tom","dick","harry">
```

W() and WQ() are predefined rules which have the effect of printing their arguments. / is another predefined rule which has the effect of cutting out the alternative clauses for the predicate in which it appears.

As an example of the use of prefixed terms and trees, the predicate walk() takes a single argument which is a tree representing an arithmetic expression and prints it out in infix notation:

```
walk(add(?x,?y)):-
        W("("),
        walk(?x),
        W("+"),
        walk(?y),
        W(")");

walk(sub(?x,?y)):-
        W("("),
        walk(?x),
        W("-"),
        walk(?y),
        W(")");

walk(mul(?x,?y)):-
        W("("),
        walk(?x),
        W("*"),
        walk(?y),
        W(")");

walk(div(?x,?y)):-
        W("("),
        walk(?x),
        W("/"),
        walk(?y),
        W(")");

walk(?x):-
        CON(?x),
        W(?x);
```

An internal expression tree such as add(a,mul(b,c)) can be printed out using the goal:

```
P>walk(add(a,mul(b,c)));
(a+(b*c))
```

Expression trees are a very useful way of representing formulae for symbolic processes such as differentiation.

Predefined rules

Prolog contains many predefined rules. These are rules which are always available, and perform useful functions like input and output, manipulation of the rule-base and arithmetic. Although they behave like rules, they are not written in Prolog but translate into calls to machine-code routines. Details and examples of predefined rules are given in the programmer's manual, but a short summary is given here:

Database predicates:

ADDCL(<clause>,<number>) adds a clause to the rule-base.

CL(<variable>,<predicate name>) finds the list of clauses for a predicate.

DB(<variable>) gets a list of the names of all known predicates.

DELCL(<predicate name>,<clause number>) deletes a clause from a predicate.

KILL(<predicate name>) deletes an entire predicate from the database.

ADD(<num1>,<num2>,<variable>) sets <variable> to the sum of <num1> and <num2>.

DIV(<num1>,<num2>,<variable>) sets <variable> to the quotient of <num1> and <num2>.

MUL(<num1>,<num2>,<variable>) sets <variable> to the product of <num1> and <num2>.

SUB(<num1>,<num2>,<variable>) sets <variable> to the difference between <num1> and <num2>.

DEL(<filename>) deletes <filename> from disk.

GETC(<variable>) sets <variable> to the next character read from the input stream.

IN(<filename>) redirects subsequent input from <filename>.

OUT(<filename>) redirects subsequent output to <filename>.

PUTC(<character>) sends a single character to the output stream.

R(<variable>) sets <variable> to the next Prolog term read from the input stream.

W(<term>) writes a single Prolog term to the output stream.

WQ(<term>) writes a single Prolog term to the output stream. Constants are quoted and @nnn notation is used for unprintable characters.

ASC(<character>,<variable>) sets <variable> to the ASCII code for <character>.

CHR(<number>,<variable>) sets <variable> to the character whose ASCII code is <number>.

JOIN(<variable>,<list>) sets <variable> to the constant formed by concatenating all the constants in <list>, which must be terminated by nil.

SPLIT(<variable>,<constant>) sets variable to a list of the characters forming <constant>.

EQ(<term1>,<term2>) unifies <term1> and <term2>.

GE(<con1>,<con2>) succeeds if <con1> is greater than or equal to <con2>.

GT(<con1>,<con2>) succeeds if <con1> is greater than <con2>.

LE(<con1>,<con2>) succeeds if <con1> is less than or equal to <con2>.

LT(<con1>,<con2>) succeeds if <con1> is less than <con2>.

NE(<term1>,<term2>) succeeds if <term1> cannot be unified with <term2>.

/ suppresses all alternatives encountered since the beginning of the current clause.

ASSIGN(<variable>,<term>) sets <variable> to <term>, even if <variable> already had a value.

FAIL() always fails.

QUIT() leaves the Prolog interpreter.

CON(<term>) succeeds if <term> is a constant.

LST(<term>) succeeds if <term> is a list (including nil).

NUM(<term>) succeeds if <term> is a number.

VAR(<term>) succeeds is <term> is an unbound variable.

Advanced topics

Escape sequences and control codes

Escape sequences and other control characters can be inserted in constants using the @ notation. An @ sign in a constant followed by up to three decimal digits is replaced by the character whose ASCII code is given by the digits. As with any constant containing non-alphanumeric characters, constants containing @ characters mus be contained in quotes ("). For example, the sequence ESC A 01 02 (in BASIC, CHR$(27)+"A"+CHR$(1)+CHR$(2)) is written as "@27A@1@2" in Prolog. If the escape sequence for positioning the cursor to row x, column y was ESC A y x, a predicate locate() could be written to position the cursor as follows:

```
locate(?x,?y):-
      W("@27A"),
      PUTC(?y),
      PUTC(?x);
```

Then the goal locate(10,12) would move the cursor to row 10, column 12.

Garbage collection

During the evaluation of a goal, Prolog accumulates a certain amount of "garbage", consisting mostly of variables that have been used but will not be used again. Periodically, Prolog starts a "Garbage Collection" which involves looking ahead through the future execution of the program to see which variable are needed, and recovering the space associated with those that are not needed. Prolog begins garbage collection when the amount of memory acquired exceeds a threshold value. This threshold can be set by the user on the Prolog command line. The command line parameter "-mXXX" or "-mXXXk" is used to set the garbage collection threshold to XXX pages or XXX kilobytes respectively. For example:

```
prolog -m5k
```

starts the Prolog interpreter with a garbage collection threshold of 5 kilobytes. When the amount of acquired memory exceeds this amount, garbage collection begins. If garbage collection fails to bring the memory allocation under the threshold, it will be repeated on every cycle of the interpreter until it does so. Garbage collection is relatively slow, so it is important not to set the threshold too low. Conversely, if the threshold is set higher than the amount of memory available to Prolog, garbage may accumulate until Prolog runs out of memory, without any effort being made to collect it. You may need to experiment to find the most efficient garbage collection threshold for your system (depending on which modules are loaded into memory when Prolog is running).

Database manipulation

Predefined rules are provided for manipulating the relational database. A clause can appear as

an argument to a predicate, so for example, ADDCL() can be used to add a new clause at a particular position in the list of clauses for a predicate, eq:

```
ADDCL(p(?x,?y):-q(?x),r(?y),5)
```

This goal adds the clause shown for predicate p() as clause number 5 (the first clause is numbered 0).

Arithmetic

Predefined rules are provided for addition, subtraction, multiplication, division. For example, to set ?x to the sum of ?y and ?z:

```
ADD(?y,?z,?x)
```

Input and output

Prolog provides two "streams", the input stream and the output stream. Normally the input stream reads from the keyboard and the output stream is displayed on the monitor. The input and output streams correspond to OS9's standard input and standard output. Thus to run a non-interactive Prolog session from a file "prolog.session", and send the output to the printer "/p" command-line redirection can be used:

```
prolog <prolog.session >/p
```

The streams can be redirected during the course of the program using the predefined rules IN() and OUT(). If IN() is used interactively, input reverts to the keyboard when end-of -file is reached. Thus the goal:

```
P>IN("prolog.source");
```

results in the Prolog source program in the file "prolog.source" being loaded into memory.

String handling

String handling in Prolog is accomplished by splitting constants into lists of characters using SPLIT(), manipulating the lists using Prolog's list-processing facilities, then joining the characters back to form a single constant using JOIN(). For example:

```
reverse(?x,?y):-
      SPLIT(?x-list,?x),
      rev(?x-list,nil,?y-list),
      JOIN(?y-list,?y);

rev(nil,?x,?x):-;
```

```
rev(?x.?y,?z,?r):-
      rev(?y,?x.?z,?r);
```

The goal reverse(hello,?x) succeeds by binding ?x to the constant "olleh", ie the characters of the first argument are reversed to form the second argument.