4.900

Prolog Tutorial 3

by Matthew Huntbach

Prolog Tutorial 3

The contents of this manual describe the product, ***WIN-PROLOG***, version 4.500, and are believed correct at time of going to press. They do not embody a commitment on the part of Logic Programming Associates Ltd (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
web: http://www.lpa.co.uk

4 February, 2005

Contents

Artificial Intelligence 1

These notes were written by Matthew Huntbach, Dept of Computer Science, Queen Mary and Westfield College, London, UK E1 4NS. Email: mmh@dcs.qmw.ac.uk.

These notes were edited by Clive Spenser, LPA.

This is a set of notes introducing Prolog as a logic programming language.

It shows how Prolog may be considered as a relational language, a non-deterministic language, and a database language.

It also covers more advanced topics such as trees and meta-level reasoning.

Notes on Prolog

This is a set of notes introducing Prolog as a programming language.

In this set of notes, sections 1, 2 and 3 respectively show how Prolog may be considered as a relational language, a non-deterministic language, and a database language. Section 4 introduces some of the extra features of Prolog which are added on to its base of automated logic. Section 5 covers the subject of search which is important in Artificial Intelligence generally, as is meta-programming which is covered in section 6 along with some other Prolog techniques.

These notes were written by Matthew Huntbach, Dept of Computer Science, Queen Mary and Westfield College, London, UK E1 4NS. Email: mmh@dcs.qmw.ac.uk.

These notes were edited by Clive Spenser, LPA.

Section 1 – Prolog as a Relational Language

Accessing Prolog

Once installed, LPA Prolog is available from the menu under the "Start" button in NT Windows.

Prolog is often seen as an interpreted language. The current program consists of whatever procedures have been loaded. A call to any procedure in the program may be made by typing the call in response to the prompt.

A number of procedures are built-in, that is they exist before any other program is loaded.

One of these is consult, and can be used to load a program from a file. Type:

```
consult(<filename>).
```

in response to the prompt, and the procedures in the named file will be loaded. Note the full-stop at the end. If the filename starts with a capital letter, or contains non-alphanumeric characters, you will have to enclose it within quotes:

```
consult('<filename>').
```

The file will be taken from your current directory, but you may use a full path to access files in other directories.

Or you can use the File/Open option to open the file into the *WIN-*PROLOG IDE and then Run/Compile it when you want to use it.

Or you can use the File/Load option to consult the file directly into memory.

To exit the Prolog system type:

```
halt.
```

Or hit the close box, as you would expect.

Help on any Prolog built-in procedure can be obtained by accessing the *WIN-*PROLOG Help file.

A Relational Language

Prolog is a relational language. That is, a Prolog call may be regarded as expressing a relation between its arguments. This can be contrasted with functional languages where a call may be regarded as a mathematical function. Both functional and relational languages however are considered declarative, that is they break away from the idea of a programming language as a set of instructions to manipulate the memory of a computer (such languages are referred to as imperative), and instead concentrate on providing a precise notation in which computation is a form of reasoning. Prolog originates from attempts to provide a computer-executable form of mathematical logic notation (its name comes from Programming in Logic), thus it and similar languages are often referred to as the logic languages.

To return to the idea of relations, in a functional language we might have the function `square`, which takes an integer and squares it; a call `square(5)` is considered as rewriting to `25`, expressing the functional relation `square(5)→25`. The equivalent in Prolog is a call `square(5,X)`. Here `X` is a Prolog variable. If a correct program for square has been loaded, typing:

```
?- square(5,X).
```

will cause

```
X = 25
```

to be printed, indicating that `X` must have the value `25` for the relation `square(5,X)` to be true. You need to type a carriage return after this to get the prompt for the next call (you will see why later).

A program for `square` consists of just:

```
square(X,Y) :-
  Y is X*X.
```

This says that `X` and `Y` are in the relationship `square` if `Y` is `X*X` evaluated arithmetically. Considering declarative languages in terms of the imperative language like C, a functional language may be considered as programming in terms of functions, which return values but do not have any side-effects. A logic language may be considered as programming in procedures, which do not return values directly (`void` in C terminology) and in which variable parameters are always passed by reference (preceded by `&` in C terminology) though the variables can only be assigned new values within the procedure if they were unassigned when the procedure was called.

In these notes, Prolog will be introduced as a programming language by starting from a restricted subset and gradually building up to a fuller version of the language.

Prolog Procedures

A Prolog procedure for a k-ary relation (i.e. one which takes k arguments), named `proc` takes the form:

```
proc(X1,
   ... Xk) :-
   call1,
   ...,
   calln.
```

where X_1,... X_k are Prolog variables, and $call_1$, ..., $call_n$ Prolog procedure calls. Note that a variable name may be any alphanumeric atom starting with a Capital letter, while a procedure name may be any alphanumeric atom starting with a small letter. The underscore character '_' is treated as an additional capital letter in names. Each of $call_1$, ..., $call_n$ may contain variables from X_1, ... X_k among its arguments, or new variables, or constants.

When a call $proc(arg_1, ... arg_k)$ is made, any occurrence of a variable X_i among the calls is replaced by arg_i. Each of $call_1$, ..., $call_n$ is then executed in turn. As an example, if we have the existing procedure `square`, a procedure to give the difference of two squares may be given by:

```
diff2squares(X,Y,D) :-
   square(X,X2),
   square(Y,Y2),
   D is X2-Y2.
```

Note that the call `is` (expressible in infix form) is used to give arithmetic calculation. Don't mistakenly use = in its place, as that has a different effect – see later in the notes. If the left-hand side of `is` is a variable, and the right-hand side an arithmetic formula, the result is that execution of the `is` call will cause the variable to store the result of evaluating the formula. One very important point to note is that Prolog variables may only be assigned a value once in a procedure body. You could not have:

```
diff2squares(X,Y,A) :-
   square(X,A),
   square(Y,B),
   A is A-B.
```

since here, once `A` has been assigned a value in the call `square(X,A)`, that value cannot be changed by the further call `A is A-B`.

Pattern Matching

In fact, Prolog procedures are rarely given as a single list of calls as in the above example. In general pattern-matching is employed. A Prolog procedure will take the form:

```
proc(pat11,
   ... patk1) :-
   call11,
   ...,
   calln(1)1.

...

proc(pat1m,
   ... patkm) :-
   call1m,
   ...,
   calln(m)m.
```

that is there will be several (in this case m) parts taking the form

```
proc(pat1,
   ... patk) :-
   call1,
   ...,
   calln
```

Each of these is referred to as a clause, since in the declarative reading of the program it may be considered a clause in logic. The proc(pat_1, ... pat_k) is referred to as the head of the clause, the calls $call_1$, ..., $call_n$ the body.

It will be noted that the head contains patterns rather than single variables. The intention is that a call proc(arg_1, ... arg_k) is first matched against proc(pat_{11}, ... pat_{k1}); if the match succeeds, the calls $call_{11}$, ..., $call_{n(1)1}$ are executed with variables having any values obtained in the matching. If the matching does not succeed, the next clause is tried, and the clauses are tried one at a time until one is found where the matching does succeed.

The matching works by matching each arg_i with each pat_i. If pat_i is a single variable, the matching succeeds, with the variable taking the value arg_i in the execution of the calls. If pat_i is a constant value, the matching only succeeds if arg_i is an identical constant. A Prolog constant may be an integer, or an atom enclosed in quotes e.g. 'This is an atom'. The quotes may be omitted if the atom starts with a small letter and contains no non-alphanumeric symbols apart from _. Patterns may be compounds made up of further patterns. For now, we will consider only patterns representing lists.

In Prolog, lists are written enclosed within square brackets, for example:

```
[a,35,[b,2],'Hello world']
```

is the four-element list whose first element is the atom 'a', second the integer 35, third the list [b,2], fourth the atom 'Hello world'. Note that Prolog is an untyped language, thus lists containing such mixed elements are allowed.

The pattern [pat_h | pat_t], where pat_h and pat_t are further patterns, matches with any

argument which is a list whose head matches with pat_h and whose tail matches with pat_t. More generally, the pattern $[pat_1, pat_2, \ldots, pat_n \mid pat_t]$ matches with any list of at least n elements, whose first element matches with pat_1, second with pat_2 and so on till the nth matches with pat_n, and finally the rest of the list matches with pat_t. A pattern in which there is no '|' matches only with a list of an exact length, so $[pat_1,pat_2,\ldots,pat_n]$ matches only with a list of length n whose first element matches with pat_1, second with pat_2 and so on till the nth matches with pat_n.

List Processing Examples

We now have nearly enough Prolog to be able to write simple list-processing programs. To complete this first subset of Prolog, one more system primitive is needed. The call `X = arg`, where `arg` is any Prolog value (i.e. a constant, a variable, or a list), and `X` is any Prolog variable, may be thought of as a form of assignment, assigning the value `arg` to `X`. The symbols = and `is` are distinguished by the fact that `is` causes its right-hand argument to be evaluated as an arithmetic expression, whereas = does not cause any arithmetic to be done. However, since Prolog variables cannot be reassigned values we generally say a variable which has been given a value has been instantiated or bound to that value. In fact Prolog programs rarely use =, it is introduced temporarily here to aid learning the language for those unfamiliar with it; later you will see why it is usually not needed. The following are some examples of simple Prolog programs which process lists:

```
double([],L) :-
   L=[].

double([H|T],L) :-
   double(T,D),
   L=[H,H|D].
```

This example will take a list and duplicate every element. So, for example if the call double([a,b,c],X) is made, computation will terminate with X bound to [a,a,b,b,c,c]. What this is saying is that if the input list is the empty list, [], the output list must also be the empty list. Otherwise, to double the elements of a list, double its tail, and cons the head of the list to the result twice.

This can be put slightly differently: the `double` of an empty list is the empty list, the `double` of a list `[H|T]`is `[H,H|D]` where `D` is the `double` of `T`. Putting it this way is approaching the problem declaratively – it is thinking about the program in logical terms rather than trying to work out the steps taken by the computer to solve it.

The next program shows a slightly more complex example of pattern matching. It is a program to swap pairs of elements in a list, so the call `swappairs([a,b,c,d,e,f],X)` will result in `X` instantiated to `[b,a,d,c,f,e]`. The main recursive clause matches with the first two elements of the input list, and its tail, and there is a separate clause to deal with the empty list. But neither of these deal with the case of lists of one element. Since a call to `swappairs` with a list of length n results in a call to `swappairs` with a list of length n-2, any call with a list of odd length will eventually result in a call with a list of one element. The solution we have chosen will result in the last singleton element being removed altogether:

```
swappairs([],L) :-
   L=[].

swappairs([X],L) :-
   L=[].

swappairs([X,Y|T],L) :-
   swappairs(T,S),
   L=[Y,X|S].
```

Next an example involving two input lists. This will simply append the two lists, so

append([a,b,c],[d,e,f],X) will result in X being instantiated to [a,b,c,d,e,f]. The declarative view of this is that appending the empty list to a list L must be that list L unchanged. Otherwise the append of a list [H|T] to a list L must be H consed to the append of T to L. Note that there is no need for a special clause for the case where the second list is the empty list, this is already covered by the two clauses as given:

```
append([],X,L) :-
   L=X.

append([H|T],X,L) :-
   append(T,X,A),
   L=[H|A].
```

The next example covers a case of lists of lists. It is intended to take a list of lists and return the list of the heads of its input lists. As an example, the call:

```
heads([[a,b,c],[d,e],[f,g,h,i]],X)
```

will result in X being instantiated to [a,d,f].

Note how the pattern matching works with the second clause. The head of the input list will be matched with [H|T], resulting in further matching. The tail of the input list will be matched with the single variable R.

```
heads([],L) :-
   L=[].

heads([[H|T] | R], L) :-
   heads(R,S),
   L=[H|S].
```

This heads example cannot deal with a case where one of the lists within the input list is the empty list, for example heads([[a,b,c],[d,e],[],[f,g,h]],X). Such a call would eventually result in a call to heads with an input list whose first element is the empty list, in the example given heads([[],[f,g,h]],V). This cannot match with the first clause, since the input list is not the empty list, neither will it match with the second clause, since the head of the input list is [] which cannot match with [H|T]. The result is that the call will fail. Failure of a call within a clause will cause that whole clause to fail, thus the call heads([[a,b,c],[d,e],[],[f,g,h]],X) would fail. In Prolog this is indicated by the interpreter responding to the call with no, giving no variable instantiations.

Adding Tests

The subset of Prolog given above is limited by the lack of tests except pattern matching tests. We cannot, for example, perform one operation if some value is greater than another, another operation if it is less. In fact, Prolog allows tests to occur among the calls in the body of a clause. The rule is that if a call matches with the head of a clause, but on executing the body of that clause a failure is encountered, the interpreter will go back and attempt to match the original call with the next clause and, if the matching succeeds, attempt to execute that clause.

As an example, suppose we want to insert a number into a numerically ordered list (in ascending order). If the number is less than or equal to the first item in the list, then we simply cons it to the front of the list, thus inserting 2 into [3,4,5] gives [2,3,4,5].

If the number is greater than the first item of the list, then we should insert it into the tail. We also need to cover the case of inserting a number into the empty list. This gives us the following program for **insert**:

```
insert(X,[],L) :-
   L=[X].
insert(X,[H|T],L) :-
   X=<H,
   L=[X,H|T].

insert(X,[H|T],L) :-
   X>H,
   insert(X,T,I),
   L=[H|I].
```

Here the arithmetic tests either fail or succeed as you might expect.

You might consider that the X>H test in the third clause is unnecessary, since this clause would only be reached in the case where X=<H has failed in the previous clause. Although you would be correct, you will see later why simply leaving it out would result in an incorrect program. For now, the advantage of putting in this redundant test is that it maintains the declarative reading of the program. The third clause says that "the insert of X into the list [H|T], when X>H is [H|I] where I is the insert of X into T". Keeping this declarative approach means that the order in which the clauses are written does not matter, the program could equally well be written:

```
insert(X,[H|T],L) :-
   X>H,
   insert(X,T,I),
   L=[H|I].

insert(X,[],L) :-
   L=[X].

insert(X,[H|T],L) :-
   X=<H,
   L=[X,H|T].
```

Testing for equality can often be done using pattern matching. As an example, if we wish to remove the nth item from a list, we test if n =1. If it is, we simply remove the head of the list (i.e. return the tail), otherwise we decrease n by 1 and go to the tail of the list. Or, putting it declaratively, the **remove** of the 1st item of a list is the tail of the list; the **remove** of the nth item of a list, when n>1, is the head of the list consed onto the **remove** of the n-1th item from the tail of the list. This gives the code:

```
remove(1,[H|T],L) :-
   L=T.

remove(N,[H|T],L) :-
   N>1,
   N1 is N-1,
   remove(N1,T,R),
   L=[H|R].
```

Note that there is no clause for the case where **N<1**. The result is that a call to **remove** would fail when **N<1**.

As a further example of the use of tests in list processing, the following program will take a list, and two bounds, Upper and Lower. Any element of the list which falls outside these bounds is deleted from it. The list is given as the first argument, the upper and lower bounds as the second and third arguments respectively.

For example, deleterange([4,2,5,7,8,1,3],3,7,X) would result in X instantiated to [4,5,7,3].

```
deleterange([],Upper,Lower,L) :-
   L=[].

deleterange([H|T],Upper,Lower,L) :-
   H>Upper,
   deleterange(T,Upper,Lower,L).

deleterange([H|T],Upper,Lower,L) :-
   H<Lower,
   deleterange(T,Upper,Lower,L).

deleterange([H|T],Upper,Lower,L) :-
   H>=Lower,
   H=<Upper,deleterange(T,Upper,Lower,D),
   L=[H|D].
```

Note the two tests in the last clause. Each test must succeed for the clause to succeed.

The tests >, <, >=, =< are used for testing integers. To compare symbols (assuming standard alphabetic order) they must be preceded by @, giving @>, @<, @>=, @=<. The double equals, == can be used as an equality test in either case, \== as an inequality test. Note that the arithmetic tests require simple arithmetic arguments, they will not evaluate expressions. So, for example, you could not have a call X>Y+Z. This would have to be replaced by calls which evaluate Y+Z first using is, i.e. Sum is Y+Z,X>Sum. The arithmetic tests fail if their arguments are not integers or if they are variables which by the time the tests are called have not become bound to integers. In many versions of Prolog if they fail in this way, no error message is printed, it is treated as a normal test failure, which can be the source of subtle errors. In *WIN-PROLOG*, though, a warning message is printed.

In the examples given so far, the only user-defined procedure call in the bodies of clauses is a recursive call. In fact, any call may be used in the bodies of a clause. As an example, the following is a procedure for insertion sort, using the insert procedure given above:

```
isort([],L) :-
   L=[].

isort([H|T],L) :-
   isort(T,S),
   insert(H,S,L).
```

Section 2 – Prolog as a Non-deterministic Language

Prolog's Non-determinism

A deterministic language is one in which there is only one possible output for any given input. A non-deterministic language is one on which for a given input there are several possible outputs. For example, in the following program in the Miranda functional language:

```
select x = x+3, x>5

select x = x*4, x<8
```

if `x` is `7` either branch is applicable, and though the convention is that the first applicable branch textually is used for reduction, an alternative way of viewing it would be to say that the call `select7` could return either `10` or `28` – which one not being determined. Prolog is a non-deterministic language, the above could be written (note the need to introduce an extra variable for output compared with Miranda):

```
either(X,Y) :-
   X>5,
   Y is X+3.

either(X,Y) :-
   X<8,
   Y is X*4.
```

With the execution rules as described before, the first matching clause is chosen for execution, so executing `either(7,Y)` will result in `Y` being instantiated to `10`. However the effect of choosing the alternative match can be given by typing a semicolon ';' in response to the return of an instantiation for `Y`. The alternative instantiation for `Y`, `28`, will then be given. Thus Prolog like Miranda, chooses clauses in textual ordering of matching, but unlike Miranda, whenever a choice is made a record is made of the choice point, and it is possible to backtrack and return to the condition of the program at that choice point, and consider the result of making an alternative choice. As a more complex example, we start with the following program which simply takes a list and adds a character before every element in that list:

```
addch(X,[],L) :-
   L=[].

addch(X,[H|T],L) :-
   addch(X,T,A),
   L=[X,H|A].
```

So, `addch(q,[a,b,c],X)` will result in `X` bound to `[q,a,q,b,q,c]`. Now we give a similar program which is non-deterministic:

```
addeither(X,Y,[],L) :-
   L=[].
```

```
addeither(X,Y,[H|T],L) :-
   addeither(X,Y,T,A),
   L=[X,H|A].

addeither(X,Y,[H|T],L) :-
   addeither(X,Y,T,A),
   L=[Y,H|A].
```

The intention is that at each point in the list, either of the two characters given as arguments may be added. If the program is loaded, and a call is made all the possible outputs can be obtained by repeatedly entering semicolons:

```
?- addeither(p,q,[a,b,c],L).

L = [p,a,p,b,p,c] ;
L = [p,a,p,b,q,c] ;
L = [p,a,q,b,p,c] ;
L = [p,a,q,b,q,c] ;
L = [q,a,p,b,p,c] ;
L = [q,a,p,b,q,c] ;
L = [q,a,q,b,p,c] ;
L = [q,a,q,b,q,c] ;
no
```

The process that is taking place is known as backtracking. The effect is that whenever a non-deterministic choice is made the program can be made to go back and choose the next alternative choice. The final `no` indicates there are no more possible outputs.

To show why the possible outputs are given in that particular order, you should recall that whenever the program is made to backtrack, the last choice made is undone and the next choice made. If there are no more possibilities for the last choice made, the choice before that is undone, and all the possibilities derived from making this alternative second-to-last choice are explored, and so on.

The possible choices may be shown in tree form. In the diagram below each node in the tree represents a choice point. The upper branch represents choosing the second clause, the lower choosing the third; `addeither` has been abbreviated to `ae`. In general, the list of calls may be considered a stack, with rewriting popping the first goal from the stack and pushing on the goals from the body of the clause used for rewriting, with matching as appropriate. Any variable which occurs only in the body of this clause is renamed to prevent clashes of variable names, in the example below the renaming is done through the use of single and double quotes.

```
ae(p,q,[],A"),A'=[p,c|A"],A=[p,b|A'],L=[p,a|A]

        ae(p,q,[c],A'),A=[p,b|A'],L=[p,a|A]

ae(p,q,[],A"),A'=[q,c|A"],A=[p,b|A'],L=[p,a|A]

                ae(p,q,[b,c],A),L=[p,a|A]

ae(p,q,[],A"),A'=[p,c|A"],A=[q,b|A'],L=[p,a|A]
```

```
                                    ae(p,q,[c],A'),A=[q,b|A'],L=[p,a|A]

                          ae(p,q,[],A''),A'=[q,c|A''],A=[q,b|A'],L=[p,a|A]

                                                    ae(p,q,[a,b,c],L)

                          ae(p,q,[],A''),A'=[p,c|A''],A=[p,b|A'],L=[q,a|A]

                                    ae(p,q,[c],A'),A=[p,b|A'],L=[q,a|A]

                          ae(p,q,[],A''),A'=[q,c|A''],A=[p,b|A'],L=[q,a|A]

                                               ae(p,q,[b,c],A),L=[q,a|A]

                          ae(p,q,[],A''),A'=[p,c|A''],A=[q,b|A'],L=[q,a|A]

                                    ae(p,q,[c],A'),A=[q,b|A'],L=[q,a|A]

                          ae(p,q,[],A''),A'=[q,c|A''],A=[q,b|A'],L=[q,a|A]
```

At each case, the goal `ae(p,q,[],A'')` in the leaves of the tree will rewrite to `A''=[]`, and the execution of the assignments will then give the output value for `L`. Backtracking involves moving through this tree in depth-first order.

As another example, the following program is designed to split a list into one of its elements and the rest of the list. Clearly one way of doing this is to return the head and tail of the list. Another way is to split off an element from the tail of the list. This leads to the following program:

```
split([H|T],E,L) :-
   E=H,
   L=T.

split([H|T],E,L) :-
   split(T,E,TL),
   L=[H|TL].
```

Running through a call of `split` with an input list will give all the possible splits of that list.

The `split` procedure may be used in a procedure which will generate all possible permutations of a list:

```
perm([],Perm) :-
   Perm=[].

perm(List,Perm) :-
   split(List,Element,List1),
```

```
    perm(List1,Perm1),
    Perm=[Element|Perm1].
```

Reading this declaratively, the only permutation of the empty list is the empty list. Otherwise a permutation can be obtained by splitting off one of the elements of the list (any one will do), finding a permutation of the remainder of the list, and consing the split off element to this permutation.

The procedural way of looking at it is that `split` will pass a sublist to `perm`. `perm` will then generate each possible permutation of the sublist, consing the split off element to it each time it is made to backtrack. When `perm` has exhausted all possible permutations of the sublist, another backtracking will cause `split` to backtrack, generating another split from which more permutations can be generated.

User-defined Tests

We have already seen the built-in comparison tests, but it is also possible for users to define their own tests. A call fails if there is no clause with whose head it can match or if for every clause whose head it matches one of the calls in the body fails. So tests can be programmed by writing procedures which are intended to fail or succeed rather than instantiate variables.

As a simple example, the following gives a test for membership of a list:

```
member(X,[H|T]) :-
  X==H.

member(X,[H|T]) :-
  member(X,T).
```

Declaratively, an object is a member of a list if it is equal to the head of that list or if it is a member of the tail of the list. Procedurally, a call such as `member(c,[a,b,c,d])` will first match with the head of the first clause, and execute the body of that clause, that is test if the object is equal to the head (in this case if `a==c`). If this succeeds, the membership test is true. Otherwise it will go on to the second clause, resulting in the recursive call. The effect is that the object is tested against each item in the list until it is found in the list, or the empty list is reached. If the empty list is reached, there is no matching clause and the membership test is false.

Having defined `member`, it may now be used by other procedures. For example, the following may be used to find the intersection of two lists. It runs through the first list adding its elements to the output list if they are members of the second:

```
intersect([],L1,L) :-
  L=[].

intersect([H|T],L1,L) :-
  member(H,L1),
  intersect(T,L1,L2),
  L=[H|L2].

intersect([H|T],L1,L) :-
  \+member(H,L1)),
  intersect(T,L1,L).
```

As an example, `intersect([a,b,c,d,e],[u,o,i,e,a],L)` will result in `L` instantiated to

```
[a,e].
```

Note the use of \+ in this case. A call \+C where C is some call succeeds wherever the call C would fail, and vice versa. It can be interpreted as, and in some versions of Prolog is written as, "not C". This interpretation is known as "negation by failure". It should be used with caution, though, for reasons given in section 6.

Prolog offers an alternative way of writing the above:

```
intersect([],L1,L) :-
  L=[].

intersect([H|T],L1,L) :-
  member(H,L1),
  !,
  intersect(T,L1,L2),
  L=[H|L2].

intersect([H|T],L1,L) :-
  intersect(T,L1,L).
```

The '!' is known as the "cut". Any calls occuring in a clause body before a cut may be considered as occurring negated in subsequent clauses. The advantage of the cut is that it is much more efficient. Without it, if `member(H,L1)` failed in the second clause above, it would need to be re-evaluated and found to fail again in the third clause in order that `\+member(H,L1)` is shown to succeed. The disadvantage of the cut is that it loses the pure declarative nature of the program. The ordering of the clauses now affects the meaning of the program, since the third clause only makes sense when it occurs after the second clause with its cut.

The exact definition of the cut is that it cuts off backtracking. We have said that when we attempt to execute a call it will be matched with the heads of the clauses for that call. If it matches, but one of the calls in the body fails (and there are no intermediate choice points in the body calls before the failure) , it will go back and attempt another clause. However this is limited by the cut. If a call matches with the head of a clause and executes the calls in the body until it gets past a cut and then a call after the cut fails (and there are no choice points between the cut and the failure point) there is no more backtracking as there is no backtracking across a cut.

For another example of a user-defined test, below is given a procedure to test whether a list of integers is sorted in ascending order. Declaratively we can say that a list of integers is sorted in ascending order if its tail is sorted and its head is less than or equal to the first element in the tail. Lists of none or one element are also sorted. This translates to the program:

```
sorted([]). sorted([X].

sorted([X,Y|L]) :-
   X>=Y,
   sorted([Y|L]).
```

Note that the third clause could be written with the calls in the body the other way round:

```
sorted([X,Y|L]) :-
   sorted([Y|L]),X>=Y.
```

Although this would not change the meaning of the program, it would make it less efficient. In the first case, a list such as `[5,3,8,4,2,1]` would immediately be found not to be sorted when matched with the third clause and the >= test applied. In the second case, however, a call to `sorted([3,8,4,2,1])` would be made, which would make a further recursive call and so on, and the fact that the list is not sorted would not be discovered until a considerable amount of work had been done.

Generate and Test

We saw previously how a program could be made to backtrack and deliver alternative answers. If in the body of a clause a call is executed, generating some instantiation for a variable, and that variable is then used in a test which fails, the Prolog execution mechanism does not immediately abandon that clause. Rather it causes a backtracking to the call to generate a new instantiation for the variable which is tested again.

This mechanism is used in the following program which sorts a list, using the procedures perm and sorted we covered earlier:

```
slowsort(L,S) :-
   perm(L,S),
   sorted(S).
```

When `slowsort(L,X)` is called, with `L` some list, the call `perm(L,S)` will generate a permutation of `L` and put it in `S`. This permutation is tested to see if it is sorted. If it is not, `perm(L,S)` will backtrack and generate another permutation, and this process will continue until the permutation which happens to be the sort of the input list is generated.

The program is justifiably called "slowsort", since generating permutations at random and then testing each one to see if it is sorted is about the most inefficient algorithm one could use for sorting a list. Nevertheless, it serves as a demonstration of the power of Prolog. The method of generating values and then testing them to see if they fit some condition is known as generate-and-test. It should be used with caution since, as in this example, it tends to lead to inefficient programs.

In general, if we have:

```
proc(Args) :-
   call1,
    ... ,
   calln-1,
   calln,
   test,
   rest.

proc(Args) :-
   body2.
```

if the execution gets as far as **test** and then fails, it will call on $call_n$ to backtrack. If there are no more possible backtracks in $call_n$ it will call on $call_{n-1}$ to backtrack, execute $call_n$ with the new values generated from $call_{n-1}$ and call **test** again. It will keep on calling for backtracks further and further back, and only give up on the first clause and try the second when there are no more possible backtracks in any of $call_1$, ... $call_n$.

Note the way that the cut cuts off backtracking. If we had:

```
proc(Args) :-
   call1,
   ... calln,
   test1,
   !,
   test2,
   rest.

proc(Args) :-
   body2.
```

if **test1** succeeded, the execution mechanism would go past the cut and execute **test2**. If **test2** then failed, there would be no bactracking even if backtracking were available in $call_1$, ... $call_n$, since the cut stops backtracking. There would be no backtracking to the second clause either, the call to **proc** would simply fail.

Backtracking over the goals past the cut is possible though. If we had:

```
proc(Args) :-
   call1,
   ... callk,
   !,
   callk+1,
   ... calln,
   test,
   rest.

proc(Args) :-
   body2.
```

if a call to proc got as far as test, and test failed, any backtracking within $call_{k+1}$, ... $call_n$ would be done if necessary, but no backtracking within $call_1$, ... $call_k$ and the second clause would never be reached.

In general, whenever there is a non-deterministic choice of clauses to be made, Prolog will pick the first choice, but leave a choice point. Whenever a point is reached where there are no choices that can be made, the Prolog system will go back to the state it was in when it last made a non- deterministic choice, unbinding any variables that were bound after that choice, and start again with the next possible choice (except that certain system-defined operations which you will see later on cannot be undone, for example, the input-output operations, and also the database manipulating operations). The system-defined tests such as the comparison operators may be thought of as non-deterministic operations in which there is either one or no choices that may be taken. A cut may be thought of as linking up with the choice point at which the decision to choose the clause containing the cut was made. When the cut is passed, this choice point and all those below it are removed.

In many cases, cuts are put it not to cut off computation, but because there are no further clauses that will match with a call that reaches the cut, and putting the cut in will stop an inefficient and unnecessary attempt to try further clauses, all of which will result in failure. This sort of cut is sometimes described as a "green cut". A green cut is one which does not change the meaning of a program, and is contrasted with a "red cut" which does.

Section 3 – Prolog as a Database Language

Unification

A problem novice Pascal programmers often make is to confuse the assignment operator := with the equality operator =, while in C confusion is caused by = being the assignment operator and == the equality comparison test. In section 2 it was suggested that = is Prolog's assignment operator, and == the equality test, as in C. If, however, you had accidentally used = in the place of ==, you might have noticed that it seemed to function just as well as an equality test. In fact, Prolog's = is more powerful than both. It may be read as an operator which "forces equality", that is it takes two arguments, if they are equal it succeeds, but if one is an unassigned variable it forces it to become equal to the other by assigning it the other argument's value. If both are lists containing uninstantiated variables, if there is a way of assigning values to these variables to make the lists equal, the assignment is done. If the arguments are two separate variables, the variable names are made to be two names for the same variable, so that any subsequent instantiation of one will instantiate the other to the same thing. Only if both arguments have already been assigned values, and these values cannot be made equal, does Prolog's = fail.

Prolog's = corresponds in fact to what is called *unification* in logic. It is a symmetric operation, so it may cause variables on both the left hand and right hand side to become instantiated. To unify two expressions, we use the following algorithm:

unify **a** **b** = if **a** is a variable then

instantiate **a** to **b**; return true else if **b** is a variable then

instantiate **b** to **a**; return true

else if **a** is a list **[ha|ta]** and **b** a list **[hb|tb]**

if unify(**ha**,**hb**) then return unify(**ta**,**tb**)

else return false

else if **a** and **b** are identical then return true else return false;

Note that there is no longer a distinction between patterns and arguments. What we previously called a "pattern" is simply data which contains uninstantiated variables. This means that two values which are both list structures may be unified if their heads and tails may be unified. A variable may occur more than once in an expression. If so, if it is instantiated in one part of the unification, that instantiation carries through to the rest. Another name for an instantiation is a "binding", and we speak of a Prolog variable as "bound" to a particular value by unification.

To give some examples (recall that variables start with Capital letters, constants with small letters):

X=5 unifies, and results in X bound to 5.

`fred=Y` unifies and results in `Y` bound to `fred`.

`X=Y` unifies and results in `X` bound to `Y`, thus, following the unification, `X` and `Y` are two different names for the same variable.

```
5=fred fails to unify (as they are two non-identical constants).

fred=fred unifies (as they are two identical constants).

X=[a,b,c] unifies, with X bound to the list [a,b,c].
```

`[H|T]=[a,b,c]` unifies, with `H` bound to `a` and `T` to `[b,c]`. According to the above algorithm, it works in stages, first breaking down to `H=a,T=[b,c]`.

`[a,b,c]=[H1,H2|T]` unifies with `H1` bound to `a`, `H2` to `b`, and `T` to `[c]`. This works in the following stages: first the unification breaks down into unification of the heads and tails: `a=H1,[b,c]=[H2|T]`. The second unification then breaks down further into its heads and tails: `a=H1,b=H2,[c]=T`.

`[H1,tom|T]=[dick,H2,harry]` unifies, with `H1` bound to `dick`, `H2` to `tom` and `T` to `[harry]`. This is because the unification breaks down to: `H1=dick,tom=H2,T=[harry]`.

`[joe,X,3]=[Y,Z,4]` fails to unify. It breaks down to `joe=Y,X=Z,3=4`, and the last of these unifications fails, so the whole unification fails.

`[X,X|T]=[a,Y,c,d]` unifies, with `X` bound to `a`, `Y` bound to `a` and `T` bound to `[c,d]`. This is because the unification breaks down to `X=a,X=Y,T=[c,d]`. Following the first of these, `X` is bound to `a`. So when `Y` is then bound to `X`, `Y` is bound to `a`.

`[a,X,c,X]=[Y,d,c,Y]` fails. This is because it breaks down to `a=Y,X=d,c=c,X=Y`. The first two of these succeed, resulting in `Y` bound to `a` and `X` to `d`; `c=c` succeeds as both arguments are equal. But `X=Y` fails because of the previous binding of these two variables to different values.

`[[a,b,c],[d,e,f],[g,h,i]]=[H|T]` unifies with `H` bound to `[a,b,c]` and `T` to `[[d,e,f],[g,h,i]]`.

`[[a,b,c],[d,e,f],[g,h,i]]=[[H1|T1],H2|T2]` unifies with `H1` bound to `a`, `T1` to `[b,c]`, `H2` to `[d,e,f]` and `T2` to `[[g,h,i]]`.

`fred=[X]` fails because a constant cannot be bound to a list.

`X=[Y]` succeeds, with `X` bound to `[Y]`.

`X=[q|X]` is a problem case. What it is saying is that `X` is bound to a list whose first element is `q` and whose tail is `X`; so its tail is a list whose first element is `q` and whose tail is a list whose first element is `q` and … . Or to put it another way, `X` is bound to an infinite list of `q`s. Some Prolog systems have what is called an occur check which tests whether a variable is being bound to a structure which includes that variable, thus leading to such infinite structures. In such a case, the unification is failed. Most Prolog systems don't have this check, but the effect of making such a binding is undefined, so it is left to the programmer to avoid it. There have also been experiments with systems which allow such circularity in much the same way as lazy functional languages allow infinite data structures.

A Look Again at Clause-Matching

Previously I described the method by which a call is matched with a clause rather informally. In fact the matching is done through full unification. Given a call of the form proc(Arg_1, ..., Arg_n), attempts are made to unify the call with the heads of the clauses for proc. This means that the specific use of unification to give an output assignment, suggested earlier, is not in fact necessary. The assignment can be done as part of the unification of call with clause head.

As an example, the following program was previously given for insert:

```
insert(X,[],L) :-
   L=[X].

insert(X,[H|T],L) :-
   X=<H,
   L=[X,H|T].

insert(X,[H|T],L) :-
   X>H,
   insert(X,T,I),
   L=[H|I].
```

However, in the place of the explicit unifications, the following could be used:

```
insert(X,[],[X]). insert(X,[H|T],[X,H|T]) :-
   X=<H.

insert(X,[H|T],[H|I]) :-
   X>H,
   insert(X,T,I).
```

So now, as an example, if the call insert(2,[1,3],L) were made, an attempt would first be made to unify insert(2,[1,3],L) with insert(X,[],[X]). This would fail, and insert(2,[1,3],L) would then be unified with insert(X,[H|T],[X,H|T]). This succeeds, with X bound to 2, H to 1, T to [3], and L to [2,1,3]. However, the call X=<H fails with this binding. This failure causes backtracking, and the backtracking causes the binding made to L to be rescinded. insert(2,[1,3],L) is then unified with insert(X,[H|T],[H|I]), causing L to be bound to [1|I], with X bound to 2, H to 1, T to [3]. The test X>H succeeds, and the call insert(X,T,I), bound to insert(2,[3],I) is made. This call results in I bound to [2,3], thus L is bound to [1,2,3]. Now you can see why = rarely occurs directly in Prolog programs, since it is usual to use unification with patterns in the head directly for output as well as input; this makes the clauses more like statements in logic and less like parts of imperative programs where you have to issue an order for output to be given.

Note that each time a clause is used to solve a call, it is assumed the variables in that clause are fresh. In the above example, when the recursive call insert(2,[3],I) was made, and matched with the clause insert(X,[H|T],[X,H|T]) :- X=<H, the variables X, H, T and I in this clause would not be linked with the variables of the same name from the calling clause.

Another point to note is that the first clause for insert lost its body altogether. In the absence of any body, the ":-" is omitted. One way of considering this is that insert(X,[],[X]) is shorthand for insert(X,[],[X]) :- donothing. But another way is

not to think of it in procedural terms at all, and instead to consider it as expressing a relationship. What it is saying is that the `insert` of anything into an empty list is a list containing just that one element.

The Prolog Database

The use of facts in Prolog can be extended to give the effect of a built-in database. A relation may be defined not in procedural terms, as we have emphasized up to now, but simply as a set of facts. It may be accessed by Prolog's unification and backtracking facilities.

As an example, the following is a database of facts representing the courses taught in a Computer Science department:

```
course(prog1,smith,a,[]).

course(prog2,jones,b,[prog1]).

course(arch1,sykes,c,[]).

course(arch2,higgins,a,[arch1]).

course(langs,wilkins,b,[prog2]).

course(systems,jenkins,d,[]).

course(vlsi,evans,c,[systems]).

course(networks,jenkins,a,[]).

course(dismaths,brown,d,[]).

course(logic,robinson,b,[]).

course(computability,brown,c,[dismaths,logic]).

course(compilers,jones,d,[prog2]).

course(ai,jones,a,[prog2,logic]).

course(distsys,wilkins,a,[langs,networks]).

course(cryptography,smith,b,[computability]).
```

Here the first argument is the name of the course, the second the name of the lecturer, the third a code indicating the timeslot of the lectures, and the fourth a list of course names which are prerequisites for the course.

Now, if the facts above are loaded into the system, to find all the courses taught by Jones, I type:

```
course(Name,jones,Time,PreReq).
```

An attempt will be made to unify this (remember `Name`, `Time` and `PreReq` are all variables as they begin with capital letters) with each of the facts in turn. It first unifies with the second one, causing `Name` to be bound to `prog2`, `Time` to `b` and `PreReq` to `[prog1]`. Typing a semicolon will cause the system to backtrack giving the second course taught by Jones: `Name=compilers`, `Time=d`, `PreReq=[prog2]`. Another semicolon will cause it to backtrack and give the third course taught by Jones: `Name=ai`, `Time=a`, `PreReq=[prog2,logic]`.

Similarly, to find all the courses taught at time slot b, I type:

```
course(Name,Lecturer,b,PreReq).
```

It is possible for the initial call to have more than one non-variable argument, for example the following:

```
course(Name,brown,Time,[]).
```

will find all courses taught by Brown which have no prerequisites. The call

```
course(Name,jones,Time,[]).
```

unifies with none of the facts, thus showing there are no courses taught by Jones which have no prerequisites. If it were typed into the system, the system would respond with no.

Thus an alternative way of thinking of Prolog calls is as queries to the database of Prolog facts, which are answered with bindings giving values which satisfy the queries, or no if the query cannot be satisfied.

Queries may be qualified by tests. For example, if the code for member (as given in section 2 of these notes) has been loaded, the following may be used to find those courses taught by Jones for which prog2 is a prerequisite:

```
course(Name,jones,Time,PreReq), member(prog2,PreReq).
```

This will first cause Name to be bound to prog2, Time to b, and PreReq to [prog1], but then member(prog2,[prog1]) fails, so the system backtracks without printing these bindings, and on the backtrack binds Name to compilers, Time to d, and PreReq to [prog2]. This time, member(prog2,[prog2]) succeeds, so these answers will be printed. It is possible to obtain the other course taught by Jones for which prog2 is a prerequisite by forcing a backtrack at this point through typing a semicolon, resulting in Name=ai, Time=a, PreReq=[prog2,logic].

Queries to databases may occur inside programs. For example, suppose we want to find the requirements for a course where a requirement is a course which is needed either directly or indirectly for that course. For example, the course compilers has as its prerequisite the list consisting of just prog2. But prog1 is a prerequisite for prog2. So the requirement for compilers is

```
[prog1,prog2].
```

The following may be used to generate a complete list of requirements:

```
requirement(Course,Req) :-
   course(Course,Lecturer,Time,PreReq),
   requirements(PreReq,Req1),
   append(Req1,PreReq,Req).

requirements([],[]).

requirements([Course|Courses],Req):-
   requirement(Course,Req1),
   requirements(Courses,Req2),
   append(Req1,Req2,Req).
```

where append is as defined previously. Note that there are two separate procedures:

requirement which takes as input a single course name, and requirements which takes a list of course names.

Note that, as a shorthand, the clause for requirement can be written:

```
requirement(Course,Req) :-
   course(Course,_,_,PreReq),
     requirements(PreReq,Req1),
     append(Req1,PreReq,Req).
```

The underscores are "anonymous variables", representing variable arguments which are only used in one place, thus avoiding the necessity of giving each a name. If the Single Variables flag is set, ***WIN-PROLOG*** prints a warning if a variable name is used only in one place in a clause, since this might be an indication that there has been a spelling mistake somewhere, causing two occurrences of a variable to be written differently, which would not give a compiler error, but would result in incorrect behaviour. It is therefore sensible always to use "_" when a variable is intentionally used just once in a clause to avoid this warning and guard against this sort of error.

Note that with Prolog as described so far, we cannot accumulate lists of solutions. If we wanted to find out how many courses Jones taught, we could not do so by obtaining a list of those courses and finding its length, because we cannot obtain such a list using the methods we have seen. The only way we could look at the separate courses taught by Jones was by backtracking, but backtracking by definition redefines values, and we cannot combine the different values we have obtained through different backtrackings. This problem can be overcome by using the all-solutions extensions which are available in Prolog, but we will leave discussion of these to section 6.

Section 4 – Further Prolog Features

Data Structures

We have seen the use of the list data structure, and since Prolog like Lisp is untyped, as in Lisp lists may be used to model every other data structure. However, Prolog allows the use of named structures. A structure is written simply with a structure name (which may be any Prolog atom) and a list of its arguments enclosed within brackets. As an example, if we are writing a program which deals with dates, rather than writing a date as a list of three elements: e.g. `[5,march,1991]`, we could write it as `date(5,march,1991)`. We need to extend the unification algorithm to deal with structures. Two structures unify if and only if their functors (name of the structure) are equal, they both have the same number of arguments, and each argument unifies with its corresponding argument.

So, for example:

`date(5,march,1991)=date(Day,Month,Year)` is a successful unification, resulting in `Day=5`, `Month=march`, `Year=1991` (recall that lower case initial letters indicate constants, upper case variables).

`student(compsci,'John Jones',2,male) = student(Dept,Name,Year)` is not a successful unification, as the two structures have differing numbers of arguments.

`pair(5,X)=pair(4,Y)` fails as it requires the unification of `5` with `4` which fails.

Prolog gives a number of built-in operations for handling structures. The operation `arg` can be used to find the nth argument of a structure, so a call `arg(3,date(4,june,1976),Y)` will cause `Y` to become bound to `1976`, while `arg(2,date(4,june,1976),Y)` would cause `Y` to become bound to `june`. The operation `functor` returns the functor and arity (number of arguments) of a structure, so `functor(date(4,june,1976),F,N)` would cause `F` to become bound to `date` and `N` to `3`.

One way of thinking of a structure is to think of it as a way of writing a list, with the first element of the list becoming the functor (this is how a structure is represented in Lisp). So `date(4,june,1976)` is a way of writing `[date,4,june,1976]`. Prolog allows a conversion between these two representations using an operation written '`=..`', and termed the univ operation. This operation works both ways, so `date(4,june,1976)=..L` will cause `L` to become bound to

`[date,4,june,1976]` while `D=..[date,4,june,1976]` will cause `D` to become bound to

`date(4,june,1976)`.

In general, a Prolog term that is a value to which a Prolog variable may be bound is an integer, a atom, a variable, or a structure made up of Prolog terms (where a list is just a special case of a structure).

An Example: Binary Trees

The program below gives an extended example of a data structure manipulating program. The data structure used is the binary search tree in which a node stores some data, all items stored in nodes in the left subtree are less than the data stored in the root node in some ordering, all items stored in nodes in the right subtree are greater than it in the ordering.

For example, the following is a binary search tree of characters:

In the program, a tree whose left branch is L , whose right branch is R, and whose node stores the data N is represented by the structure **tree**(L,N,R). The empty tree is represented by the constant **empty**. The program gives code to insert an item into a tree, code to delete an item from a tree, code to build up a tree from a list, and code to flatten a tree into a sorted list.

```
insert(N,empty,tree(empty,N,empty)).
insert(N,tree(Left,M,Right),tree(NewLeft,M,Right)) :-
   N@=<M,
   !,
   insert(N,Left,NewLeft).

insert(N,tree(Left,M,Right),tree(Left,M,NewRight)) :-
   insert(N,Right,NewRight).

delete(N,tree(Left,M,Right),T) :-
   N==M,
   !,
   delete_root(Left,Right,T).

delete(N,tree(Left,M,Right),tree(NewLeft,M,Right)) :-
   N@=<M,
   !,
   delete(N,Left,NewLeft). delete(N,tree(Left,M,Right),tree(Left,M,NewRight)) :-
   delete(N,Right,NewRight).

delete_root(Left,empty,Left) :-
   !.

delete_root(empty,Right,Right) :-
   !.

delete_root(Left,Right,tree(NewLeft,N,Right)) :-
   remove_rightmost(Left,NewLeft,N).

remove_rightmost(tree(Left,N,empty),Left,N) :-
   !.

remove_rightmost(tree(Left,N,Right),tree(Left,N,NewRight),M) :-
   remove_rightmost(Right,NewRight,M).

build([],empty).

build([H|T],Tree) :-
   build(T,Tree1),
   insert(H,Tree1,Tree).
```

```
flatten(empty,[]).

flatten(tree(Left,N,Right),L) :-
   flatten(Left,FlatLeft),
   flatten(Right,FlatRight),
   append(FlatLeft,[N|FlatRight],L).
```

The algorithms are straightforward, though the algorithm for deleting an item may need some explanation (you may have seen it in your programming course last year). If the item to be deleted occurs at the root of the tree, it is replaced by the rightmost item in the left subtree. The procedure `remove_rightmost` is used to split a tree into its rightmost element and the tree remaining when this element is removed. As an example, deleting the character s from the tree given previously would cause it to be replaced by q, resulting in the tree:

Note the use of the cut to prevent unwanted backtracking. For example, in the third clause for delete_root, we know that neither Left nor Right will be empty, since if they were one of the cuts in the previous clauses would have been reached, the cut in that clause passed, and the third clause would never be reached.

Defining Infix Operators

You will already know that the arithmetic operators are an example of infix operators. As an example 3+4 takes 3 and 4 and delivers their sum, 7. If + were not an infix operator, we would have to write this +(3,4) . Prolog allows you to define infix structure constructors, in fact the arithmetic operators are already predefined as such. For example, if you were to make the call:

```
|?- X=3+4.
```

you would get back:

```
X=3+4

yes
```

what this actually means is that `X` has been bound to the structure whose functor is +, whose arity is 2, whose first argument is 3 and whose second argument is 4. This can be shown by using the `=..` operator:

```
|?- X=3+4, X=..L. X=3+4

L=[+,3,4]

yes
```

It is necessary to use the arithmetic evaluation operator `is` to actually calculate an arithmetic expression:

```
|?- X is 3+4. X=7

yes
```

The arithmetic operators have their usual precedence, so * 'binds tighter' than +. For example, `3+4*5` is interpreted as `3+(4*5)` rather than `(3+4)*5`. Brackets can be

used to resolve the ambiguity. This is demonstrated below:

```
?- X=3+4*5, X=..L. X = 3+4*5


L = [+,3,4*5]

yes

?- X=(3+4)*5, X=..L. X = (3+4)*5

L = [*,3+4,5]

yes
```

You can define your own infix functors using the built-in command **op**. **op** takes three arguments. The first argument is a precedence which is an integer value – the lower the value the "tighter" the binding. The second argument is either **xfx**, **xfy**, **yfx**, or **yfy** (these will be explained later). The third argument is the functor itself. For example, if you wanted **with** to be an infix operator, this could be done by entering **op(100,xfx,with)**. You could then use, say, **3 with 4** in the place of **with(3,4)**.

Using **xfx** means the infix operator has no associativity, that is **3 with 4 with 5** does not have a valid interpretation. Using **yfx** gives it left associativity, so **3 with 4 with 5** will be interpreted as **(3 with 4) with 5**, or without using infixing **with(with(3,4),5)**. Using **xfy** gives it right associativity, so **3 with 4 with 5** will be interpreted as **3 with (4 with 5)** or **with(3,with(4,5))**.

The arithmetic and comparator operators may be thought of as already declared using, for example:

```
op(700,xfx,'<'). op(500,yfx,'+'). op(400,yfx,'*').
```

Note that if you want to use self-defined infix operators in a program, they can be defined using a directive. This is a call which is a line in a program in a file, but is executed when the file is consulted and the program loaded from it. A directive is given by starting with "**:-**". As an example, if the file **hello.pl** contained:

```
:- write('Defining functor and '),nl.

:- op(50,xfx,and). hello(X,Y,Z) :-
   Z=X and Y.
```

loading it using

```
?- consult('hello.pl').
```

would cause **write('Defining functor and '),nl** and **op(50,xfx,and)** to be executed. So the words "**Defining functor and**" would be printed, and the **op** call would be executed, enabling **and** to be used as an infix operator in the clause defining **hello**.

Input and Output in Prolog

You have already seen the use of `write` with a atom argument to output words in Prolog. In fact `write` with any Prolog term as an argument will cause that term to be written out as it would be displayed in the Prolog system. The zero argument call `nl` simply writes a newline. The call `read(X)` where `X` is a variable will cause a Prolog term to be read in, and `X` to be bound to that term. Prolog will prompt for the input with '`|:`', and it must be finished with a full stop and newline. So the following represents a possible dialogue in Prolog:

```
?- read(T).

|: tree(empty,3,tree(empty,4,empty)).

T = tree(empty,3,tree(empty,4,empty)) ;

no
```

Note backtracking over `read/1` just gives failure, it will not re-read a term. If the argument to read is not a variable, an attempt will be made to unify the term read with the argument, so:

```
?- read(tree(L,N,R)).

|: tree(empty,3,tree(empty,4,empty)). L = empty

N = 3

R = tree(empty,4,empty)
```

Prolog offers alternative input/output operators which work on individual characters rather than Prolog terms. These may be used to provide flexible interfaces to Prolog programs, but they have one awkward problem due to Prolog's untyped nature.

Since a variable is untyped in Prolog, if it is bound to an integer value there is no way of saying whether that value is to be interpreted as an integer or a character. Thus if a character is read into a variable, if the value of that variable is displayed it will be displayed as its ASCII code equivalent. The character input operation is `get`, so `get(X)` will read the next character from the input and bind `X` to its ASCII value. Repeated calls of `get/1` will give a prompt only at the start of a new line. For example, whereas

```
?- read(X).

|: f(3).
```

will cause `X` to be bound to the Prolog term `f(3)`,

```
?- get(X),get(Y)

|: f(x).
```

will cause `X` to be bound to `102` (the ASCII code for '`f`') and `Y` to `40` (the ASCII code for '`(`').

The operation `put` takes an ASCII code value, and prints the character associated with it, so

put(103) will print the single character 'g'.

get0/1 is a version of get/1 which skips past any spaces or non-printing characters, reading the first printable non-blank character.

The built-in operator name is used to convert between lists of integers, each integer being an ASCII code, and strings. The call name(X,[101,102,103]) will cause X to be bound to the string efg. name can be used to convert either way – the call name(abc,X) will cause X to be bound to

```
[97,98,99].
```

Prolog offers some predicates for handling external files so that input and output can be from and to the files rather than the computer screen. The operation

```
see(Filename)
```

switches current input to the file Filename. It is switched back to the screen by the 0-argument

```
seen
```

and there is also seeing(Filename) for finding the name of the current input stream. The corresponding predicates for switching output to some file, switching back to the screen, and testing the name of the current output stream are: tell(Filename), told, and telling(Filename).

Changing the Database

Prolog offers operations which may be used to change the facts stored in the database. As the database is in fact part of the program, these operations should be used with caution, since they give the effect of a program which alters itself, which can clearly be very hard to debug. Some Prolog programmers make fairly extensive use of these database-changing commands, to give various "special effects"; others would argue that making extensive use of them is a sign that one "hasn't yet learnt to think in Prolog".

The command to add a fact is assert(Term) where Term is a structure (or variable bound to a structure). So if in the database example given in section 3 the command

```
assert(course(ai,simpson,c,[prog1]))
```

were executed, it would give the effect of adding a new fact:

```
course(ai,simpson,c,[prog1]).
```

at the end of the facts on courses.

The command retract removes facts from the database. It takes a single argument, and removes the first fact which unifies with that argument. So, given the database of facts on courses in section

3, executing

```
retract(course(C,jenkins,T,P))
```

would cause the fact

```
course(systems,jenkins,d,[]).
```

to be removed, binding variable C to systems, T to d and P to []. Note that retract can be made to backtrack, but this will not have the effect of restoring the removed fact. Instead it will remove the next fact that unifies with the argument. So backtracking on the above call would cause:

```
course(networks,jenkins,a,[]).
```

to be removed as well.

In fact it is possible to assert and retract not just facts but complete rules as well. Thus:

```
assert(( f(X,Y) :-
   g(X,Z),h(Z,Y) )).
```

adds a new rule for f. Note the need for double brackets to make it clear that assert is taking just one argument, the whole rule. If it is necessary to distinguish whether the rule should be added before or after the other rules for f (recalling that since rules are tried in order, this may make a big difference to execution), the commands asserta and assertz exist to do each respectively, there is also retracta and retractz to retract the first matching rule and last matching rule respectively. In practice, of course, it is poor programming practice to do this sort of thing, since the effect is that most difficult thing to understand, a self-modifying program.

Further Control Structures

Prolog contains a few extra control structures which may occur in clauses. Opinions vary as to whether these are helpful in writing Prolog programs, or whether they just introduce unnecessary extra complexity into the language.

We suggested that the comma, "," in clause bodies could be read as a logical "and". Prolog also gives an operator intended to represent logical "or", the semicolon ";". So where

```
p:-
  q1,
  q2,
  ... ,
  qn.
```

is read as "p if q1 and q2 and ... qn"

```
p:-
  q1; q2; ... ; qn.
```

is read as "p if q1 or q2 or ... qn".

In fact the ";" is usually combined with the ",", with the usual precedence that "and" binds tighter than "or". So

```
p:-
  q1,
  q2,
  ...,
  qn ; r1,
  r2,
  ...,
  rm.
```

is read as "p if (q1 and q2 and ... qn) or (r1 and r2 and ... rm)".

The effect is that the qs are executed, with backtracking if necessary but if there is a failure and no backtracking is possible, execution goes on to attempt the rs. It is in fact another way of writing:

```
p:-
  q1,
  q2,
  ...,
  qn.

p:-
  r1,
  r2,
  ...,
  rm.
```

The call

```
(p->q;r)
```

is read as "if p then q else r". It causes p to be attempted, If execution of p fails, then

r is executed, else q is executed. (p->q) is a shorthand for (p->q;fail) where fail is a call which always fails. (p->q;r) is in fact, just another way of writing c where c is defined by the clauses:

```
c:-p,!,q. c:-r.
```

Use of the -> notation means it is not necessary to define a new procedure name for c. It may also be used to avoid the use of the cut in many cases. Which is the "better" Prolog style is a matter of opinion.

The built-in goal fail, which always fails can prove useful in some cases. For example, if we had the previously defined test member to test whether an item is a member of a list, we could define a test notmember which succeeds if its first argument is **not** a member of its second as follows:

```
notmember(X,L) :-
  member(X,L),
  !,
  fail. notmember(X,L).
```

This is similar to the negation by failure we considered earlier.

The fail goal may also be used to give a looping effect. A call to loop with the clause:

```
loop:-p,fail.
```

will cause repeated backtracking over p. Clearly, this is only of use if p is something which has a side-effect such as printing some output. For example, given the previous definition of perm, the following will take a list and print out all its permutations:

```
allperms(L) :-
  perm(L,P),write(P),nl,fail.
```

There is also a built-in goal which always succeeds, true. This may be used in conjunction with the or or if-then-else operators given above. As an example, the call we have just given allperms(L) will always fail, since it keeps on backtracking until it eventually fails. However, the call (allperms(L);true) will always succeed, since when allperms(L) has finished backtracking and failed, it will go on to true which succeeds.

Section 5 – Graphs and Searching in Prolog

Tree traversal in Prolog

Before considering graphs, let us go back to trees in Prolog, as represented in section 4. If you want to pick an item from a tree, you will either pick the node, or (recursively) pick it from the left or right branch:

```
item(tree(_,N,_),N).

item(tree(L,_,_),X) :-
   item(L,X).

item(tree(_,_,R),X) :-
   item(R,X).
```

If you run `item(T,X)` with `T` bound to some tree constructed (maybe using `build`), continually backtracking, you will get the items from the tree in *pre-order* order, that is, the tree is traversed starting at the node, going to the left branch and then to the right branch. *In-order* traversal (go to the left branch, the the node, then the right branch) can be obtained by reordering the clauses:

```
item(tree(L,_,_),X) :-
   item(L,X).

item(tree(_,N,_),N).

item(tree(_,_,R),X) :-
   item(R,X).
```

Post-order traversal puts the clause which looks at the node last.

Tree traversal in this way is always depth-first as it makes use of the depth-first search built-in to Prolog. If you want to traverse the tree in a breadth-first order, it is more complex, since you can't make use of the built-in backtracking of Prolog. In fact you have to use an intermediate structure, a list of trees. Initially, the list contains the whole tree. Traversal involves taking the first tree from the list, looking at its node, and putting the branches at the back of the list:

```
breadthfirst(T,X) :-
   breadthfirst1([T],X).

breadthfirst1([empty|S],X) :-
   breadthfirst1(S,X).

breadthfirst1([tree(_,N,_)|_],N).

breadthfirst1([tree(L,_,R)|S],X) :-
   append(S,[L,R],S1),
   breadthfirst1(S1,X).
```

Note that if the branches are appended to the front of the list, depth-first traversal is

obtained:

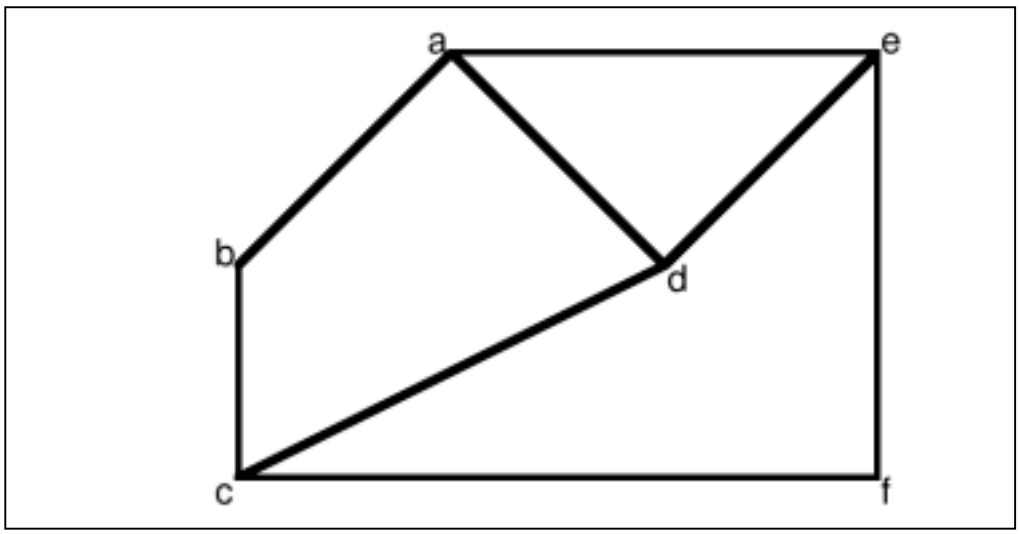
```
depthfirst(T,X) :-
   depthfirst1([T],X).

depthfirst1([empty|S],X) :-
   depthfirst1(S,X).

depthfirst1([tree(_,N,_)|_],N).

depthfirst1([tree(L,_,R)|S],X) :-
   append([L,R],S,S1),
   depthfirst1(S1,X).
```

The difference between this and the simple depth-first tree traversal given first is that here there is an explicit list of trees which operates as a stack (first in, first out) structure, whereas the simple form in fact makes use of Prolog's built-in stack. The list of trees in breadth-first search operates as a queue.

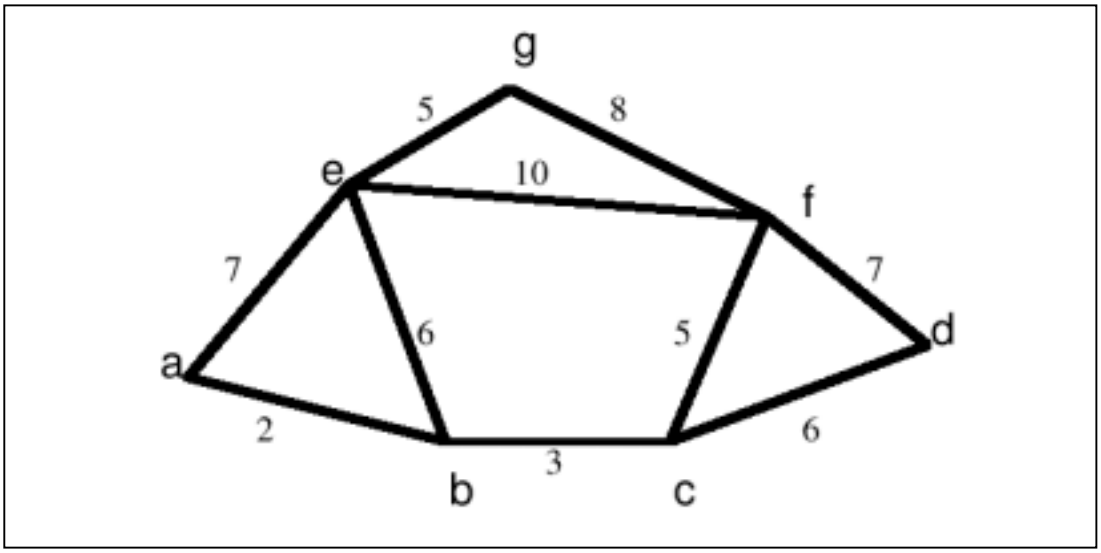
The point of this is to illustrate that while Prolog's built-in search can be useful, caution should be exercised over its use. If there are several ways to find a solution to a problem, Prolog will not necessarily find the best solution first if you rely on its own search mechanism to search over the possibilities, since that search mechanism is depth-first. Many artificial intelligence problems are variants on tree-search, but it is generally the case that a search order guided by heuristics (best- first search) is used rather than simple depth-first search. We shall consider this further with the shortest-path problem in graphs.

Representing Graphs in Prolog

Graphs are used in many applications, and the searching of graphs is of particular importance in Artificial Intelligence. A graph is defined by a set of *nodes* and a set of *edges*, where each edge is a pair of nodes. For example, the graph:



is represented by the set of nodes {a,b,c,d,e,f} and edges {a-b,a-e,a-d,b-c,c-d,c-f,d-e,e-f}.In many cases, there is a cost associated with each edge. For example, the costed graph below:



is represented by the set of nodes {a,b,c,d,e,f,g} and the set of arcs with costs:

{<a-b,2>,<a-e,7>,<b-c,3>,<b-e,6>,<c-d,7>,<c-f,5>,<d-f,7>,<e-f,10>,<e-g,5>,<f-g,8>}. This could, for example, represent a set of ports linked by sea routes, with the costs being the costs of transporting goods across the various sea routes.

One way of representing graphs in Prolog is simply to represent each edge by a fact. So the above costed graph can be represented by the set of Prolog facts:

```
edge(a,b,2).

edge(a,e,7).

edge(b,c,3).

edge(b,e,6).

edge(c,d,6).

edge(c,f,5).

edge(d,f,7).

edge(e,f,10).

edge(e,g,5).

edge(f,g,8).
```

Note that each edge is represented only once, and we have adopted the convention that the first node is the node that comes first alphabetically. The assumption is, then, that the graph is *undirected*. In a *directed* graph, links are one-way, so we cannot assume that if there is a link a-b, that b-a is also a link.

There are various operations we may wish to perform on a graph. We may, for instance wish to find a route between two nodes, and the cost associated with that route (which will be the cost of all the edges making up that route). For example, one path between nodes a and g in the costed graph given above is [a,b,e,g]. Its cost is the cost of the edges which make up this route: 2+6+5=13. The shortest and lowest cost route between a and g is [a,e,g] cost 12. Note that the shortest route (in terms of the number of edges) is not always the lowest cost route (in terms of the cost of the edges that make up the route). For example the shortest route between a and f is [a,e,f], but its cost is 17 compared with a cost of 10 for the longer route [a,b,c,f].

A simple recursive description of the route problem leads to a simple Prolog program. A route of cost C exists between two nodes X and Y if there is an edge X-Y or Y-X of cost C, or if there is some other node Z such that there is an edge between X and Z of cost C_1, and a route between Z and Y of cost C_2, and $C=C_1+C_2$. In the first case the route is just [X,Y] in the second case [X|R] where R is the route between Z and Y.

The Prolog program which this translates into is:

```
route(X,Y,[X,Y],C) :-
   edge(X,Y,C).

route(X,Y,[X,Y],C) :-
   edge(Y,X,C).

route(X,Y,[X|R],C) :-
   edge(X,Z,C1),
   route(Z,Y,R,C2),
   C is C1+C2.
```

```
route(X,Y,[X|R],C) :-
   edge(Z,X,C1),
   route(Z,Y,R,C2),
   C is C1+C2.
```

Note how the call to **edge** in the third and fourth clauses of **route** will give the effect of searching through the set of edges which link with **X** as Prolog backtracks over it. Running this program, however, will prove disappointing. Suppose we want to find a route between **a** and **g**. The least cost route is **[a,e,g]**, but somehow no matter how much we make the program backtrack, it never finds this route:

```
?- route(a,g,R,C). R = [a,b,c,d,f,g]

C = 26 ;

R = [a,b,c,d,f,g,e,g] C = 36 ;

R = [a,b,c,d,f,g,e,f,g]

C = 49 ;

R = [a,b,c,d,f,g,e,f,g,e,g] C = 59 ;

R = [a,b,c,d,f,g,e,f,g,e,f,g] C = 72 ;

R = [a,b,c,d,f,g,e,f,g,e,f,g,e,g] C = 82 ;

R = [a,b,c,d,f,g,e,f,g,e,f,g,e,f,g] C = 95
```

... and so on ...

We have hit one of the problems that can be encountered with Prolog programs, the problem of an infinite number of solutions. If we allow routes in which nodes may be re-visited there are an infinite number of routes between any pair of nodes, and we get caught in a loop which just extends one particular path infinitely.

So we perhaps need to revise our program. To cut out visits to nodes which are already part of the route, we can use the following program:

```
route(Start,Finish,Route,Cost) :-
   routewithout([Start,Finish],Start,Finish,Route,Cost).



routewithout(Used,Start,Finish,[Start,Finish],Cost) :-
   edge(Start,Finish,Cost).

routewithout(Used,Start,Finish,[Start,Finish],Cost) :-
   edge(Finish,Start,Cost).

routewithout(Used,Start,Finish,[Start|Rest],Cost) :-
   edge(Start,Midpoint,Cost1),
   not(member(Midpoint,Used)),
   routewithout([Midpoint|Used],Midpoint,Finish,Rest,Cost2),
   Cost is Cost1+Cost2.

routewithout(Used,Start,Finish,[Start|Rest],Cost) :-
   edge(Midpoint,Start,Cost1),
```

```
    not(member(Midpoint,Used)),
    routewithout([Midpoint|Used],Midpoint,Finish,Rest,Cost2),
    Cost is Cost1+Cost2.
```

Note the variable names have been altered to make the program more understandable. The procedure `routewithout` takes three input parameters, its first is a list of nodes and the second and third individual nodes. It finds a route between the two nodes given as parameters which does not use any of the nodes in the list of nodes, and which does not give any duplication of nodes.

This program will return all possible non-duplicating routes if made to backtrack:

```
?- route(a,g,R,C). R = [a,b,c,d,f,g]

C = 26 ;

R = [a,b,c,d,f,e,g] C = 33 ;

R = [a,b,c,f,g] C = 18 ;

R = [a,b,c,f,e,g] C = 25 ;

R = [a,b,e,g] C = 13 ;

R = [a,b,e,f,g] C = 26 ;

R = [a,e,g] C = 12 ;

R = [a,e,f,g] C = 25 ;

R = [a,e,b,c,d,f,g] C = 37 ;

R = [a,e,b,c,f,g] C = 29 ;

no
```

Clearly this program is still not perfect. It would be nice to have a program which gave us the lowest cost route right away, and did not require us to search through the alternatives manually.

"Dirty" Prolog

There are elegant ways of solving the problem of finding the lowest cost route between two nodes in a graph, which do not use any of Prolog's non-logical features. But a simple way round it is to use "dirty Prolog" – that is to make use of features like **assert** and **retract**. Adding the following clauses to your program will give you a procedure **bestroute** which is like **route** but will return the lowest cost solution as its only solution:

```
bestroute(Start,Finish,Route,Cost) :-
   assert(best([],1000)),
   routewithout([Start,Finish],Start,Finish,Route,Cost),
   compare(Route,Cost),
   fail.

bestroute(Start,Finish,Route,Cost) :-
   best(Route,Cost).



compare(Route,Cost) :-
   best(BestRoute,BestCost),
   change(Route,Cost,BestRoute,BestCost).



change(Route,Cost,BestRoute,BestCost) :-
   BestCost=<Cost,!.

change(Route,Cost,BestRoute,BestCost) :-
   retract(best(BestRoute,BestCost)),
   assert(best(Route,Cost)).
```

This program works by keeping a fact **best(BestRoute,BestCost)** in the database, storing the best route found so far. Initially it is set to a dummy value. The first clause for **bestroute** will keep backtracking over the call to **routewithout**. As it does, it will compare the cost of the latest route found with the cost of the best route found so far, stored in the asserted fact **best(BestRoute,BestCost)**. If the cost of the new solution is lower, this fact is retracted and the new route and cost asserted as the best route found so far. When no more solutions can be found, backtracking causes the second clause for **bestroute** to be attempted, which returns the route last asserted as the best route so far.

This shows a common use of **assert** and **retract** – to store values which would otherwise be lost through backtracking. It also shows the use of **fail** to force backtracking.

It is generally recommended that this style of Prolog programming be avoided if there are other ways round the problem. Nevertheless, while a good programmer knows the rules of style, an expert programmer knows when to break them! In some cases a limited use of "dirty Prolog" may offer the best solution.

Another Graph Representation in Prolog

In search of a cleaner program to solve the shortest path problem, we turn to another way of representing graphs. Instead of representing each edge by a separate fact, the graph is represented by just one structure, a list of edges, with their costs if it is a costed graph. So the graph we have been considering can be represented by the list:

```
[edge(a,b,2),edge(a,e,7),edge(b,c,3),edge(b,e,6),edge(c,d,6),
edge(c,f,5),edge(d,f,7),edge(e,f,10),edge(e,g,5),edge(f,g,8)]
```

The following is an attempt at a path program using this representation of a list:

```
graph([edge(a,b,2),edge(a,e,7),edge(b,c,3),edge(b,e,6),edge(c,d,6),
edge(c,f,5),edge(d,f,7),edge(e,f,10),edge(e,g,5),edge(f,g,8)]).

split(Start,[edge(Start,Next,Cost)|Edges],Next,Cost,Edges).

split(Start,[edge(Next,Start,Cost)|Edges],Next,Cost,Edges).

split(Start,[Edge|Edges],Next,Cost,[Edge|Rest]) :-
   split(Start,Edges,Next,Cost,Rest).

route(Start,Finish,Route,Cost) :-
   graph(Graph),
   routeplan(Start,Finish,Graph,Route,Cost).

routeplan(Start,Finish,Graph,Route,Cost) :-
   split(Start,Graph,Next,Cost1,RestGraph),
   routeusing(Start,Finish,Next,Cost1,RestGraph,Route,Cost).

routeusing(Start,Finish,Finish,Cost,RestGraph,[Start,Finish],Cost) :-
   !.

routeusing(Start,Finish,Next,Cost1,RestGraph,[Start|Rest],Cost) :-
   routeplan(Next,Finish,RestGraph,Rest,Cost2),
   Cost is Cost1+Cost2.
```

The way it works is that `split` non-deterministically splits out an edge from the graph which links with the start node. If this edge links directly with the finish node, a route has been found. Otherwise the node which this edge links to is treated as a mid-point in the route, and a search is made from that node to the finish.

This program doesn't run into the infinity problem, since it passes the list of edges around as a parameter. Each time an edge is split out, the rest of the graph excluding this edge is passed on. Thus no path may use the same edge more than once, and as there are a finite number of edges, there are a finite number of solutions. It will, however, give solutions which revisit a node on a different edge. For example, in the given graph one solution given for a path from node `a` to node `g` is

`[a,b,e,f,c,d,f,g]`. It will not give the lowest-cost solution first, but will reach it after a finite number of other solutions.

As with the previous case, we could use the backtrack with `assert` and `retract` method to find the lowest cost solution. However, a better program for the problem is possible, which is discussed below.

A Breadth-First Solution

We shall start by considering a *breadth-first* search technique. Because Prolog has built-in depth- first search, in order to obtain breadth-first search it is necessary to program in the search yourself. This is similar to the way when we considered tree traversal we had to construct a queue of trees in order to get breadth-first tree traversal. The basis of breadth-first search of the graph is that a *search list* of route-cost pairs is constructed, consisting initially of just the empty route from the Finish node. The program works by taking the first route-cost pair off the search list, constructing all route-cost pairs which extend it by one arc and do not visit an arc that was previously visited in that route-cost pair, and adding those to the back of the search list (so that it operates as a queue, similar to our breadth-first tree traversal). A solution is found when the route-cost pair at the head of the search list has a route with the Start node at its head.

The Prolog code for this is:

```
graph([edge(a,b,2),edge(a,e,7),edge(b,c,3),edge(b,e,6),edge(c,d,6),
edge(c,f,5),edge(d,f,7),edge(e,f,10),edge(e,g,5),edge(f,g,8)]).

route(Start,Finish,Route,Cost) :-
   graph(Graph),
   search(Graph,[route([Finish],0)],Start,Route,Cost).

search(Graph,[route([At|Rest],Cost)|Routes],Start,FinalRoute,FinalCost) :-
   At==Start,
   FinalRoute=[At|Rest],
   FinalCost=Cost.
search(Graph,[route([At|Rest],Cost)|Routes1],Start,FinalRoute,FinalCost) :-
   extensions(Graph,At,Rest,Cost,Routes2),
   append(Routes1,Routes2,Routes),
   search(Graph,Routes,Start,FinalRoute,FinalCost).

extensions([],At,Rest,Cost,[]).
extensions([edge(At,Next,Cost1)|Edges],At,Rest,Cost,Routes) :-
   member(Next,Rest),!,
   extensions(Edges,At,Rest,Cost,Routes).
extensions([edge(At,Next,Cost1)|Edges],At,Rest,Cost,Routes) :-
   !,
   extensions(Edges,At,Rest,Cost,Routes1),
   Cost2 is Cost1+Cost,
   Routes=[route([Next,At|Rest],Cost2)|Routes1].
extensions([edge(Next,At,Cost1)|Edges],At,Rest,Cost,Routes) :-
   member(Next,Rest),!,
   extensions(Edges,At,Rest,Cost,Routes).
extensions([edge(Next,At,Cost1)|Edges],At,Rest,Cost,Routes) :-
   !,
   extensions(Edges,At,Rest,Cost,Routes1),
   Cost2 is Cost1+Cost,
   Routes=[route([Next,At|Rest],Cost2)|Routes1].
extensions([Edge|Edges],At,Rest,Cost,Routes) :-
   extensions(Edges,At,Rest,Cost,Routes).
```

Here the predicate search does the basic search, with its first argument being the graph, its second the search list, its third the Start node, and its fourth and fifth become bound to the route and cost which is discovered. Note its similarity in

structure to breadthfirst1 in the tree traversal example.

The first clause for search gives the case where the head of the first route in the list of routes is the Start node. In this case, we have found a path from the Start to Finish node, so we give it as output. The second clause for search calls extensions to give the list of one-arc extensions to the route at the head of the list of routes. It then appends the two lists of routes, and makes a recursive call, taking the list resulting from the append as input.

The predicate extensions gives the one-arc extensions of a route. Its first argument is the graph, its second argument is the node at the start of the input route, its third argument is the rest of the input route, its fourth argument is the cost of the input route, and its fifth argument becomes bound to the list of extension route-cost pairs.

The first clause of extensions deals with the case where there are no more edges left to try and form an extension. The second and fourth clauses deal with the case where the next edge tested does link up with the head node of the route but it also links up with a node which is already on the route (found using member), so the extension is discounted. The third and fifth clauses are only reached when member fails in the second and fourth clauses (so the cut is never reached in these clauses). They deal with the case where the next edge links with the head node and does not link with any other nodes in the existing route, so it can be used to form an extension. The sixth clause for extensions is only reached when all the other clauses fail. This deals with the case where the next edge on the list of edges does not link up with the head node of the route, so no extension is formed. All of the clauses for extensions except the first have a recursive call to try for further extensions using the rest of the list of edges.

Breadth-first search will return the solutions in order with those having the lowest number of arcs first. This will not always be the cost order, as with the example of the route from a to f, where the lowest cost route is a three arc route, but there is a higher cost two arc route, which is found first in breadth-first search.

Note that if the first two arguments to append in `search` are reversed, making it `append(Routes2,Routes1,Routes)`, the result will be that search is depth-first in a search tree where the predicate `extensions` gives the order of the children of each node. This is similar to the way depth-first tree traversal was obtained using an explicit stack.

A Best-First Solution

Breadth-first search always looks first at the route with the fewest number of actual moves from the initial node. Artificial intelligence programs, however, often use best-first search, which order routes in the search list according to some guess as to which his most likely to lead to a solution. In the route-finding program, a simple ordering would be to put the routes in order of their cost. This means that the first route found which is a complete route from the Start to the Finish node is guaranteed to be the lowest cost route from Start to Finish. It can still find other routes by being made to backtrack, and it will return them in order of cost.

The Prolog code for best-first search is:

```
graph([edge(a,b,2),edge(a,e,7),edge(b,c,3),edge(b,e,6),edge(c,d,6),
edge(c,f,5),edge(d,f,7),edge(e,f,10),edge(e,g,5),edge(f,g,8)]).

route(Start,Finish,Route,Cost) :-
   graph(Graph),
   search(Graph,[route([Finish],0)],Start,Route,Cost).

search(Graph,[route([At|Rest],Cost)|Routes],Start,FinalRoute,FinalCost) :-
   At==Start,
   FinalRoute=[At|Rest],
   FinalCost=Cost.
search(Graph,[route([At|Rest],Cost)|Routes1],Start,FinalRoute,FinalCost) :-
   extensions(Graph,At,Rest,Cost,Routes2),
   mergeroutes(Routes1,Routes2,Routes),
   search(Graph,Routes,Start,FinalRoute,FinalCost).

extensions([],At,Rest,Cost,[]).
extensions([edge(At,Next,Cost1)|Edges],At,Rest,Cost,Routes) :-
   member(Next,Rest),
   !,
   extensions(Edges,At,Rest,Cost,Routes).
extensions([edge(At,Next,Cost1)|Edges],At,Rest,Cost,Routes) :-
   !,
   extensions(Edges,At,Rest,Cost,Routes1),
   Cost2 is Cost1+Cost,
   insertroute(route([Next,At|Rest],Cost2),Routes1,Routes).
extensions([edge(Next,At,Cost1)|Edges],At,Rest,Cost,Routes) :-
   member(Next,Rest),
   !,
   extensions(Edges,At,Rest,Cost,Routes).

extensions([edge(Next,At,Cost1)|Edges],At,Rest,Cost,Routes) :-
   !,
   extensions(Edges,At,Rest,Cost,Routes1),
   Cost2 is Cost1+Cost,
   insertroute(route([Next,At|Rest],Cost2),Routes1,Routes).
extensions([Edge|Edges],At,Rest,Cost,Routes) :-
   extensions(Edges,At,Rest,Cost,Routes).

insertroute(route(Route,Cost),[],[route(Route,Cost)]).

insertroute(route(Route1,Cost1),[route(Route2,Cost2)|Routes],
   [route(Route1,Cost1),route(Route2,Cost2)|Routes]) :-
   Cost1=<Cost2,!.
```

```
insertroute(Route1,[Route2|Routes],[Route2|Routes1]) :-
   insertroute(Route1,Routes,Routes1).

mergeroutes([],Routes,Routes).

mergeroutes(Routes,[],Routes).

mergeroutes([Route1|Routes1],[Route2|Routes2],[Route1|Routes]) :-
   lowercost(Route1,Route2),!,
   mergeroutes(Routes1,[Route2|Routes2],Routes).
mergeroutes([Route1|Routes1],[Route2|Routes2],[Route2|Routes]) :-
   mergeroutes([Route1|Routes1],Routes2,Routes).

lowercost(route(_,Cost1),route(_,Cost2)) :-
   Cost1=<Cost2.
```

To explain this code, note that `search` is the main search procedure, similar to `search` in breadth- first search, and `extensions` is similar to `extensions` in breadth-first search. However, `extensions` here inserts new states into the list it produces in order of cost using a procedure `insertroute`, rather than just putting them at the front as it did in breadth-first search. In the place of `append`, `search` uses `mergeroutes` which merges two ordered lists of routes together, keeping the result ordered.

The first clause for `insertroute` deals with inserting a route/cost pair into an empty list. Its second clause deals with inserting a route/cost pair into a list where the cost of the head is greater than or equal to the cost of the pair being inserted, so insertion is done by putting the pair to be inserted at the head of the list. The third clause is only reached when the less than or equal test in the second clause fails (thus the cut is never reached) and gives the case where the cost of the pair being inserted is greater than the cost of the pair at the head of the list, so to insert the pair, it must be inserted into the tail of the list.

The first two clauses for `mergeroutes` deal with the case where one of the input lists is the empty list, so the other list can be returned as the output. Its third clause deals with the case where the lowest cost pair is found at the head of the first input list, so it becomes the head of the merged list. The fourth clause deals with the case where the lowest cost pair is found at the head of the second input list, again this can be assumed because this clause would only be reached if the cut were never reached in the third clause.

The program is given as an example of a fairly complex Prolog program, though still short enough to fit on a single page. It avoids "dirty Prolog" methods like the use of `assert` and `retract` and forced backtracking through `fail`, and also avoids using the Prolog "or" and "if-then-else" operators. Don't worry if it takes a while to figure out how it works, though do make the effort to try! Further details on search methods may be found in most introductory books on Artificial Intelligence.

Section 6 – Advanced Programming Techniques

Incomplete Data Structures

The logic variable enables a number of powerful programming techniques to be used in logic languages which are not available in functional languages. One of the most common ones is the creation of data structures with "holes", that is containing unbound variables. If we keep a record of the unbound variable and later instantiate it to a value, we are "filling in the hole". An unbound variable which is the same as an unbound variable in a data structure may be thought of as a pointer to that hole in imperative language terminology.

The simplest example of this technique is the so-called difference list. A difference list is a pair consisting of an incomplete list L, and a variable B. L is either the variable B, or a list whose head is bound and whose tail is an incomplete list containing B. One way of thinking of this is as a list with a pointer to its back. Thus, for example, the pair [2,3,4|X]-X is a difference list representing the list [2,3,4]. The advantage of the difference list is that elements may be added to either end in one step. Thus we can define a cons operation, similar to that we would have with ordinary lists, so that consing 1 to the difference list [2,3,4|X]-X gives [1,2,3,4|X]-X: cons(H,L-B,[H|L]-B).

But we can also define an operation which adds elements to the rear by binding the back pointer to a new structure containing the new element and a further unbound variable. So to add 5 to the rear of the difference list [2,3,4|X]-X representing [2,3,4] to get the difference list representing

[2,3,4,5], we bind X to [5|Y], and return the difference list [2,3,4,5|Y]-Y:

```
addrear(R,L-B,L-C) :-
  B=[R|C].
```

or more succintly:

```
addrear(R,L-[R|C],L-C).
```

It is also possible to append two difference lists in one step, whereas appending two ordinary lists means traversing the first one. So the append of the difference list representing [1,2,3] and the difference list representing [4,5,6] is the difference list representing [1,2,3,4,5,6], or the append of [1,2,3|X1]-X1 and [4,5,6|X2]-X2 is [1,2,3,4,5,6|X2]-X2:

```
append(L1-X1,L2-X2,L3-X3) :-
  L3=L1,
  X3=X2,
  X1=L2. or:

append(L1-L2,L2-X2,L1-X2).
```

The name difference list arises because another way of looking at this structure is to say that the structure L-B represents the difference between the list L and the list B.

That is why the infix '-' is generally used as the pairing operator.

All-solutions operators

As mentioned in section 3, it is not possible through just simple backtracking to obtain lists of all possible solutions to some problem. It was suggested in section 5 that the way to save information over backtracking is through the use of `assert` and `retract`. The following, for example, is a procedure which could be combined with the courses database of section 3 so that `all_taught(Name,List)`, with `Name` set to the name of some person will give cause `List` to be set to the list of names of all courses taught by that person:

```
all_taught(_,_) :-
   assert(taught([])),
   fail.

all_taught(Name,_) :-
   course(Course,Name,_,_),
   retract(taught(Courses)),
   assert(taught([Course|Courses])),
   fail.

all_taught(_,Courses) :-
   retract(taught(Courses)).
```

This works by having a temporary fact `taught(Courses)` in the database, which is retracted and then asserted again with `Courses` extended by the next course found by backtracking until no more courses taught by the named person can be found.

Prolog however has built-in predicates which should be used in the place of explicitly modifying the database. These techniques are called second-order logic programming since they talk about sets and their properties rather than individuals. The predicate `bagof` is define so that `bagof(Term,Goal,Instances)` where `Goal` is a Prolog goal, unifies `Instances` with the list of all instances of `Term` for which `Goal` is true. So given the database of section 3, `bagof([Course,Time,PreReq],course(Course,jones,Time,PreReq),List).`

is a call which will cause the variable `List` to be set to the list

```
[[prog2,b,[prog1]],[ai,a,[prog2,logic]],[compilers,d,[prog2]]].
```

Care should be taken with this predicate, since its behaviour when an unbound variable occurs in `Goal` but not in `Term` is not, perhaps, as one might expect. The all-solutions behaviour applies only to those variables which occur in `Term`. Thus, for example:

```
bagof([Course,Name],course(Course,Name,Time,[]),List)
```

might be expected to cause `List` to be set to all courses for which there are no prerequisites. However, the variable `Time` occurs in the goal but not in the term, with the result that the query is interpreted as "for a single possible value of `Time`, set `List` to all `[Course,Name]` pairs for which `course(Course,Name,Time,[])` is true". Thus it will actually first give the list of all `[Course,Name]` pairs for `Time=a`, only, i.e. `[[prog1,smith],[networks,jenkins]]`, then, if made to backtrack, all `[Course,Name]` pairs for `Time=b` and so on. To cause the all-solutions to range over

all values of Time as well, the following notation, using "^" is used: bagof([Course,Name],Time^course(Course,Name,Time,[]),List).

This will cause List to be set to the list of all [Course,Name] pairs for which there are no prerequisites for all possible times. Note that bagof will return the solutions in the order obtained through backtracking. Another procedure, setof, which otherwise works as bagof, sorts the list and removes any duplicated solutions in it.

Negation as failure

It was noted previously that Prolog allows a form of negation, so that \+G, where G is a Prolog goal is interpreted as "not G", succeeding when G fails and vice versa. This should not be interpreted as a normal Prolog goal, however, since it has a different behaviour if G contains unbound variables. In particular, it won't find values for the variables to be bound to to make G false and hence \+G true. As an example, suppose we have:

```
married_student(X) :-
   married(X),
   student(X).

unmarried_student(X) :-
   \+married(X),
   student(X).

student(bill).

student(joe).
student(mary).
married(joe).
married(fred).
married(mary).
```

Then married_student(joe) will succeed, as will unmarried_student(bill). Also married_student(X) will succeed, binding X to joe, and on backtracking to mary. However, unmarried_student(X) will fail, rather than bind X to bill, for the reason that married(X), with X an unbound variable, will always succeed, binding X to joe. Since married(X) succeeds, \+married(X) fails, it does not work by binding X to a value which will cause married(X) to fail. In the above example, if the second clause were written:

```
unmarried_student(X) :-
   student(X),
   \+married(X).
```

the problem would not arise, since the goal student(X) would cause X to be bound before

\+married(X) were tried.

Meta-Interpreters

An interpreter for a language written in that same language is known as a meta-interpreter, and it is easy to write a meta-interpreter for Prolog. The Prolog meta-interpreter makes use of the built-in predicate clause. The call clause(A,B) where A

is a structure causes B to be unified with the body of the first clause whose head unifies with A, when it is backtracked over it will match with further clauses. If A matches with a fact (i.e. clause head with no body), B is unified with the constant true. If the body of the clause has more than one call, B unifies with with the pair (H,T)

(in rounded brackets) where H is the first call in the body and T the rest of the body in similar rounded bracket form if it consists of more than one call.

The following is a Prolog meta-interpreter:

```
solve(true) :-
   !.

solve(X) :-
   system(X),!,X.

solve((H,T)) :-
   !,solve(H),solve(T).

solve(G) :-
   clause(G,B),solve(B).
```

We say that this program is the *meta-level* program, a program which manipulates programs, while the program it manipulates is the *object-level* program. It should be noted that this meta- interpreter cannot cope with Prolog programs containing cuts, since a cut at the object-level cannot be implemented directly by a cut at the meta-level.

The predicate system is not built in, the idea is that it distinguishes system level predicates. It may be added simply by listing all the system predicates with anonymous variables as arguments:

```
system(_ is _).

system(write(_)).

system(_>_).

... etc ...
```

Note the use of the *meta-call* facility in cases where the object-level call is a system call. A single Prolog variable is given as a goal, the intention being that when execution reaches this point the variable will be bound to a structure which is treated as a Prolog call.

The meta-interpreter works by mapping most of the object level of the language into the meta- level. Thus object-level backtracking is implemented directly by meta-level backtracking, object- level variables represented by meta-level variables and so on. The main interest in meta- interpreters is that enhanced varieties may be used in a variety of ways. Thus, for example, the following enhanced meta-interpreter executes a goal given as the first argument to solve, and binds its second argument to a count of the total number of calls made during the execution (ignoring those backtracked over):

```
solve(true,0) :-
   !.
```

```
solve(X,1) :-
   system(X),!,X.

solve((H,T),N) :-
   !,solve(H,N1),solve(T,N2),N is N1+N2.

solve(G,N) :-
   clause(G,B),solve(B,N1),N is N1+1.
```

Another example of the use of meta-interpreters is the one below which is enhanced by a query mechanism where every time a call is successful, the user is asked whether the call has in fact given a result which he believes to be correct. This principle may be used to construct debuggers. If all the calls made due to a particular clause are judged by the user to be correct, but the result of the call which used the clause is judged to be incorrect, then it must be the case that that particular clause is faulty. This form of debugging using a meta-interpreter is known as *declarative debugging*. A simple declarative debugger for wrong solutions in Prolog is:

```
solve(true,none) :-
   !.

solve(X,none) :-
   system(X),!,X.

solve((H,T),Error) :-
   !,solve(H,Error1),continue(Error1,T,Error).

solve(G,Error) :-
   clause(G,B),solve(B,Error1),check(Error1,G,B,Error).

continue(none,B,Error) :-
   !,solve(B,Error).

continue(Error,_,Error).

check(none,G,_,none) :-
   query(G),!.

check(none,G,B,(G:-B)) :-
   !.

check(Error,_,_,Error).

query(G) :-
   write('Is '),write(G),write(' a correct goal? '),
   read(Reply),affirmative(Reply).

affirmative(yes).

affirmative(y).

affirmative(no) :-
   !,fail.

affirmative(n) :-
   !,fail.
```

```
affirmative(_) :-
   write('Please enter yes or no '),
   read(Reply),
   affirmative(Reply).

debug(G) :-
   solve(G,Error),report(Error),nl.

report(none) :-
   !,
   write('No bug found - do you want to backtrack?'),
   read(Reply),\+affirmative(Reply).

report(Clause) :-
   write('Faulty clause instance: '),write(' '),write(Clause),nl.
```

More complex declarative debuggers which ask fewer queries to the user may be defined, but this will suffice as an illustration. If this debugger were loaded into Prolog, together with the faulty insertion sort program:

```
isort([],[]).

isort([H|T],S) :-
   isort(T,S1),insert(H,S1,S).

insert(X,[],[X]).

insert(X,[H|T],[X|T]) :-
   X<H.

insert(X,[H|T],[H|S]) :-
   X>=H,insert(X,T,S).
```

which will result in the call `isort([2,3,1,4],L)` causing `L` to be wrongly bound to `[1,2]` instead of `[1,2,3,4]`, the buggy clause may be found by typing `debug(isort([2,3,1,4],L))`, resulting in the following dialog:

```
Is isort([],[]) a correct goal?
|: y.

Is insert(4,[],[4]) a correct goal?
|: y.
Is isort([4],[4]) a correct goal?
|: y.

Is insert(1,[4],[1]) a correct goal?
|: n.
Faulty clause instance:

insert(1,[4],[1]) :-
   1<4
```

This detects that the clause which can be matched with `insert(1,[4],[1]):-1<4` is faulty. The clause is the second one for insert which should be `insert(X,[H|T],[X,H|T]):-X<H.`

Memoization

Memoization is a technique which involves storing the results of function calls. If a call to the same function with the same arguments is made later, rather than recompute the result a search is made for it in the set of stored previous results. Memoization is applicable to both functional and logic programming; in logic programming the existence of the `assert` call makes it particularly easy to implement. The classic example of a function which is improved in efficiency by memoization is the Fibonacci function. The first Fibonacci number is 1, the second is also 1, and subsequent Fibonacci numbers are generated by the relationship that `fibonacci(N) = fibonacci(N-1)+fibonacci(N-2)`. However, evaluating `fibonacci(N-1)` means itself evaluating `fibonacci(N-2)`, and in general it will be seen that a naive implementation will involve a great deal of re-evaluation. Thus it is efficient to memoize Fibonacci values. This is done in the program below which memoizes them in the form of facts `isfib(N,F)`:

```
isfib(1,1). isfib(0,1).



fib(N,F) :-
   isfib(N,F),!. fib(N,F) :-
   N1 is N-1,
   N2 is N-2,
   fib(N1,F1),
   fib(N2,F2),
   F is F1+F2,
   asserta(isfib(N,F)).
```

Exercises

1) Give Prolog programs for the following problems:

a) double2nd – takes a list and doubles every second character, so

double2nd([a,b,c,d,e,f],X) instantiates X to [a,b,b,c,d,d,e,f,f].

b) delete – takes a character and a list and deletes all occurences of that character from the list so delete(a,[a,b,a,c,b,a,e,c],X) instantiates X to [b,c,b,e,c].

c) allbutlast – takes a list and returns the list consisting of all but its last character, so

allbutlast([a,b,c,d,e],X) instantiates X to [a,b,c,d].

d) nodups - takes a list and returns the list with all duplicated characters deleted, so

nodups([a,b,a,c,b,a,e,c],X) instantiates X to [a,b,c,e].

e) location - takes a list and a character and returns an integer which gives the location in the list where that character is first found, so location([a,b,a,c,b,a,e,c],c,X) instantiates X to 4.

f) changesome - takes a list, an integer N and two characters and returns the list in which the first N occurences of the first character are changed to the second character. So: changesome([a,b,c,a,c,d,b,a,d,e,b,b,c,b],3,b,q,X) instantiates X to

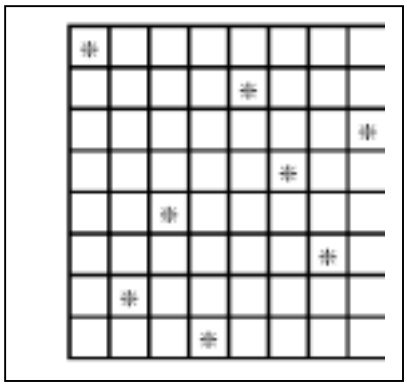
[a,q,c,a,c,d,q,a,d,e,q,b,c,b].

g) half - takes a list and returns the first half of that list, so half([a,b,c,d,e,f,g,h],X) instantiates X to [a,b,c,d].

2 a) Write a program which can be made to backtrack and print out all possible subsets of an input list. You could write two versions – one in which the items in the subsets must be in the same order as in the original list, the other which prints out all possible orderings of all possible subsets.

b) Write a program which takes two lists and test whether the second is a subset of the first. How does this program compare with the one for question 1).

c) The 8-Queens problem is a well-know example in Computer Science. The idea is to place 8 Queens on a chess board such that none attacks any of the others, that is no two queens are in the same column, no two queens are in the same row, and no two queens on the same diagonal. Below is an example setup:



Note that since all the queens must be in a separate row and a separate column, the problem degenerates into finding a permutation of the numbers 1 to 8 such that placing the queen in row 1 in the first column of the permutation, the queen in row 2 in the second column of the permutation, and so on, gives a safe placing. The above example represents the permutation [1,5,8,6,3,7,2,4].

Give a Prolog program to solve the 8-queens problem. You may find it best to generalise the problem to placing N queens on an NxN board. Even a good program will take some time to solve the

8-queens problem, so you should perhaps test your program on the 4 or 5-queens problem until you have it working correctly.

Further Reading

There are a lot of books available on Prolog as a result of the strong interest in the language due to the adoption of logic programming by the Japanese Fifth Generation Project in the 1980s. For the purposes of this course, the most suitable books are those which have a good background in the theory of logic programming.

The first book published specifically on the language Prolog, is still a good one to read, and has been regularly updated in new editions (the latest is the 4th edition):

W.F.Clocksin and C.S.Mellish Programming in Prolog , Springer-Verlag.

Two books which are recommended for an advanced look at Prolog programming are: L.S.Sterling and E.Y.Shapiro The Art of Prolog MIT Press 1986.

R.A.O'Keefe The Craft of Prolog MIT Press 1990.

Another book which is reasonable as a practical guide to Prolog programming is: T.Van Le Techniques of Prolog Programming Wiley 1993.

The books previously recommended by Bratko and by Flach also serve as good introductions to the Prolog language.