Quick Prolog

Dave Robertson

September, 2005

School of Informatics,
University of Edinburgh.

Contents

1 Getting Started

This section, along with the sample program in `ancestors.pl` is designed to give an initial, superficial introduction to basic Prolog programming. The idea is to get you working on the computer from the very first.

1.1 Some Anatomy

Prolog programs consist of clauses. For example, the following is a valid Prolog program containing 3 clauses (1 per line in this example):

```
parent(fred, greta).
parent(greta, henry).
grandparent(X, Z):- parent(X, Y), parent(Y, Z).
```

All Prolog clauses are terms finishing with a full stop symbol.
Prolog clauses are normally of three types:

- facts declare things that are always true.
- rules declare things that are true depending on a given condition.
- questions to find out if a particular goal is true.

Anatomy of a fact:

- `parent(fred, greta).` is a fact.
- `parent` is a predicate name.
- `fred` is the first argument.
- `greta` is the second argument.
- Arguments in a term are separated by commas.
- `parent`, `fred` and `greta` are atoms.

Anatomy of a rule:

- `grandparent(X, Z) :- parent(X, Y), parent(Y, Z).` is a rule.
- Its head is `grandparent(X, Z).`
- Its body is `parent(X, Y), parent(Y, Z).`
- `parent(X, Y)` is the first subgoal of the rule.
- `parent(Y, Z)` is the second subgoal of the rule.
- Subgoals in the body of a rule are separated by commas.
- `X`, `Y` and `Z` are variables.

- Its declarative interpretation can be stated in two different ways:
  - For all `X` and `Z`, `X` is a `grandparent` of `Z` if there exists some `Y` such that `X` is a `parent` of `Y` and `Y` is a `parent` of `Z`.
  - For all `X`, `Y` and `Z`, if `X` is a `parent` of `Y` and `Y` is a `parent` of `Z` then `X` is a `grandparent` of `Z`.
- Its procedural interpretation can be stated as:
  - The goal `grandparent(X, Z)` succeeds with binding `X1` for `X` and and binding `Z1` for `Z` if first, the goal `parent(X, Y)` succeeds with bindings `X1` and `Y1` and then the goal `parent(Y, Z)` succeeds with bindings `Y1` and `Z1`.

Running a Prolog program consists of supplying some question. This also is referred to as giving an initial goal or query. A goal succeeds if it is satisfiable from the set of clauses constituting ther Prolog program. A goal fails if it is unsatisfiable from the set of clauses constituting the Prolog program. For example:

- The query:

  ```
  ?- grandparent(fred, X).
  ```

  is satisfiable from the program at the top of the page, with `X` instantiated to `henry`.
- On the other hand, the query:

  ```
  ?- grandparent(fred, bob).
  ```

  is not satisfiable from the program at the top of the page because `bob` doesn't appear in that set of clauses.

Prolog programs can be recursive. That is, a rule with some predicate as its head can contain a reference to the same predicate in its body. Consider the following program:

```
/* Clause1 */ ancestor(A, B):- parent(A, B).
/* Clause2 */ ancestor(A, B):- parent(P, B), ancestor(A, P).
/* Clause3 */ parent(fred, greta).
/* Clause4 */ parent(greta, henry).
```

Here the second clause of the predicate ancestor is recursive because it has `ancestor(A, B)` as its head and `ancestor(A, P)` as the second subgoal in its body.

The Prolog interpreter can backtrack to find alternative solutions to a given goal. Backtracking can be forced by typing a semi-colon after Prolog finds the first solution to the query. For example, we get the following behaviour when running the program given above:

What you see on the screen (annotations between `**` symbols) is:

```
| ?- ancestor(X, henry).  ** I give a query to Prolog **

     X=greta ;  ** It finds a solution and I type a semi-colon **

     X=fred    ** Another solution is found for the original query **
yes   ** Since I didn't type another semi-colon Prolog reports success **
| ?-  ** and prompts for another query **
```

An explanation of how Prolog obtains this observed behaviour is given by the following annotated sequence of events. Goal1 is the initial query.

```
Goal1:      ancestor(X, henry).
        ** Match head of Clause1 to Goal1 **
Clause1(a): ancestor(X, henry) if (Goal2) parent(X, henry).
        ** Attempt to satisfy Goal2 **
Goal2:      parent(X, henry).
        ** Match to Clause4 to Goal2 **
Clause4(a): parent(greta, henry).
        ** SUCCEED with X = greta **
        ** Asked to find another solution **
        ** so backtrack to look for alternative solution for Goal2 **
        ** No further matches for Goal2 **
        ** so backtrack to look for alternative solution for Goal1 **
        ** Match head of Clause2 to Goal1 **
Clause2(a): ancestor(X, henry) if (Goal3) parent(P, henry)
                              and (Goal4) ancestor(X, P).
        ** Attempt to satisfy Goal3 **
Goal3:      parent(P, henry).
        ** Match Clause4 to Goal3 **
Clause4(b): parent(greta, henry).
        ** so P is now bound to 'greta' **
        ** Attempt to satisfy Goal4 **
Goal4:      ancestor(X, greta).
        ** Match Clause 1 to Goal4 **
Clause1(b): ancestor(X, greta) if (Goal5) parent(X, greta).
        ** Attempt to satisfy Goal2 **
Goal5:      parent(X, greta).
        ** Match to Clause3 to Goal2 **
Clause3(a): parent(fred, greta).
        ** SUCCEED with X = fred **
```

2 Standard Programming Style

You should always document you Prolog programs in order that other people (or you, at some later date) can understand what they are meant to do. There aren't rigid guidelines for this but there is a reasonably well established set of conventions, which are as follows:

- First, put a skeletal description of your predicate, showing the predicate name and its arguments (in order), with the names of the arguments giving an idea of the type of structure expected and a mode declaration at the start of each argument. Mode declarations are either `+` (if the argument is instantiated whenever the predicate is called), `-` (if the argument is uninstantiated when the predicate is called) or `?` (if the argument can be either instantiated or uninstantiated when called).
- Below this put a description in English of the behaviour of the predicate. This often makes use of the argument names from the skeletal description.

Consider, for example, the following documentation for `memberchk/2`: checks if

```
%   memberchk(+Element, +Set)
%   means the same thing, but may only be used to test whether a known
%   Element occurs in a known Set.  In return for this limited use, it
%   is more efficient when it is applicable.

memberchk(Element, [Element|_]) :- !.
memberchk(Element, [_|Rest]) :-
        memberchk(Element, Rest).
```

Note the use of `%` symbols at the start of lines which are comments. When loading a program, Prolog ignores everything from this symbol up to the end of the same line. Larger comments can be enclosed between `\*` and `*\` symbols. In the `memberchk/2` example, `Element` and `Set` are both allocated mode `+` because both need to be instantiated for the predicate to work as intended. Compare this to the definition of `member/2` (below), where either of the two elements may be either instantiated or uninstantiated, as required, and are therefore given `?` as their mode declarations.

```
%   member(?Element, ?Set)
%   is true when Set is a list, and Element occurs in it.  It may be used
%   to test for an element or to enumerate all the elements by backtracking.
%   Indeed, it may be used to generate the Set!

member(Element, [Element|_]).
member(Element, [_|Rest]) :-
        member(Element, Rest).
```

One final point about these examples. Notice that an underscore to represent an anonymous variable is used wherever the name of a variable is unimportant. How do you know that it is unimportant? If you see a clause in which a named variable appears only once then its name doesn't matter since it isn't matching to anything in the clause. For instance the first clause of `member/2` could have been:

```
member(Element, [Element|Rest]).
```

but it isn't important to distinguish `Rest` with a special name, so the convention is to use an underscore. Many Prolog systems will generate warning messages when they find "singleton" variables, such as `Rest` in the example above. It is a good idea to get into the habit of using underscores to avoid these warning messages. That way, if you accidentally introduce a singleton variable (*e.g.* by mistyping the name of a variable) you will always be able to spot it when you load the file and see the singleton warning message.

3 Prolog Terms

All Prolog data structures are called terms. A term is either:

- A constant, which can be either an atom or a number.
- A variable.
- A compound term.

3.1 Atoms

An atom is:

- Anything enclosed in single quotes (e.g. `'Whatever you like'`).
- Any sequence of characters, numbers and/or the underscore character which is preceded by a lower case character (e.g. `this_IS_an_atom`).
- Any continuous sequence of the symbols: `+`, `-`, `*`, `/`, `\`, `^`, `>`, `<`, `=`, `'`, `:`, `.`, `?`, `@`, `#`, `$`, `&`. (e.g. `*******+@`).
- Any of the special atoms: `[]`, `{}`, `;`, `!`.

The Prolog built-in predicate `atom/1` will tell you if a given term is or isn't an atom. It gives the following behaviour:

```
| ?- atom(foo).
yes
| ?- atom(Foo).
no
| ?- atom('Foo').
yes
| ?- atom([]).
yes
```

3.2 Numbers

A number can be either:

- An integer (e.g. `99`).
- A floating point number (e.g. `99.91`).

The built-in predicate `integer/1` succeeds if its argument is an integer. The built-in predicate `real/1` succeeds if its argument is a real number. These give the following behaviour:

```
| ?- integer(9).
yes
| ?- integer(99.9).
no
| ?- real(99.9).
yes
| ?- real(9).
no
| ?- number(9).
yes
| ?- number(99.9).
yes
| ?- number(foo).
no
| ?- number(Foo).
```

3.3 Variables

A variable is:

- Any sequence of characters, numbers and/or the underscore character which is preceded by an upper case character (e.g. `This_IS_a_variable`).

- Any sequence of characters, numbers and/or the underscore character which is preceded by an underscore (e.g. `_this_IS_a_variable`).
- An underscore character by itself (e.g. `_`). The underscore character is referred to as the anonymous variable because it cannot be referenced by its name in other parts of the clause.

The Prolog built-in predicate `var/1` will tell you if a given term is or isn't a variable. It gives the following behaviour:

```
| ?- var(Foo).
     Foo=_125459
yes
| ?- var(_).
yes
| ?- var(foo).
no
```

3.4 Compound Terms

Compound terms allow the representation of data with substructure. A compound term consists of a functor followed by a sequence of one or more subterms called arguments. It sometimes helps to think of a compound term as a tree structure. For example, the term `sentence(nphrase(john),vbphrase(verb(likes),nphrase(mary)))` could be depicted as the structure:

```
       ______sentence______
      |                    |
   nphrase          _____vbphrase_____
      |            |                  |
    john         verb              nphrase
                   |                  |
                 likes               mary
```

where sentence is the principal functor of the term and its arguments are `nphrase(john)` and `vbphrase(verb(likes),nphrase(mary))`. Each argument in this example is also a term with functor name and arguments.

When we want to refer to a particular class of term without giving its precise definition, we give its principal functor and its arity (the number of arguments which it takes). The example above has principal functor sentence and arity 2. The shorthand notation for this is `sentence/2`.

The predicate `functor(Term, F, A)` succeeds if `Term` has functor `F` and arity `A`. Its behaviour is as follows:

```
| ?- functor(sentence(nphrase(john),vbphrase(verb(likes),nphrase(mary))),
|:          Functor, Arity).
     Functor=sentence
     Arity=2
yes
| ?- functor(foo, Functor, Arity).
     Functor=foo
     Arity=0
yes
```

3.5 Lists

Lists are a special class of compound term. They are represented in Prolog as either:

- The atom `[]`, representing the empty list.
- A compound term with functor `'.'` and two arguments representing respectively the head and tail of the list. The tail argument must, itself be a list (possibly empty).

For example the list containing the sequence of elements `a`, `b` and `c` is represented by the term `.(a,.(b,.(c,[])))`. Since this is hard to read, you can use an alternative notation. Simply write the list as a sequence of elements separated by commas and enclosed in square brackets. The previous example could we rewritten using this notation as `[a,b,c]`.

You can confirm this by giving the following goals to Prolog:

```
| ?- functor(.(a,.(b,.(c,[]))), Functor, Arity).
     Functor=(.)
     Arity=2
yes
| ?- functor([a,b,c], Functor, Arity).
     Functor=(.)
     Arity=2
yes
| ?- .(a,.(b,.(c,[]))) = [a,b,c].
yes
```

3.6 Operators

Operators in Prolog are simply a notational convenience. We could represent the English statement "2 plus 3" in Prolog as `+(2, 3)` but this isn't the way we are accustomed to seeing the expression. A more familiar notation involves making the `+` an INFIX operator so that we can write `2 + 3`. When an operator is defined it is also allocated two descriptive attributes:

- Its precedence. This is a number between 1 and 1200 which allows the Prolog system to decide the principal functor of any nested term. For example, the precedence of `+` is 500 while the precedence of `*` (the multiplication operator) is 400. The rule is that in a nested term the operator with highest precedence value is the principal functor. For instance, the term `2 + 3 * 5` has principal functor `+`. If we were to write it in standard Prolog syntax this term would look like this: `+(2,*(3,5))`. NOTE that terms representing numerical expressions are NOT automatically evaluated, like in a functional language. There is a special built–in predicate for doing this, which we shall meet later on.
- Its type. This defines whether the operator is:
  - Prefix, in which case it is allocated type `fx` or `fy`.
  - Postfix, in which case it is allocated type `xf` or `yf`.
  - Infix, in which case it is allocated type `xfx`, `xfy` or `yfx`.

  Some operators may have more than one type. For example the `+` operator is both prefix (e.g. `+ 2`) and infix (e.g. `2 + 3`).

You can check what the current operators in your Prolog system are by using the `current_op/3` built-in predicate. You can also define your own operators using the built–in `op/3` predicate but we won't worry about this yet. To display a term containing operators in standard Prolog syntax, use the built-in predicate `display/1`.

```
| ?- current_op(Precedence, Type, +).

     Precedence=500
     Type=fx ;

     Precedence=500
     Type=yfx
yes
| ?- current_op(Precedence, Type, *).

     Precedence=400
     Type=yfx
yes
| ?- display(2 + 3 + 4).
+(+(2,3),4)
yes
```

4 Unification

The way in which Prolog matches two terms is called unification. The idea is similar to that of unification in logic: we have two terms and we want to see if they can be made to represent the same structure. For example, we might have in our database the single Polog clause:

```
parent(alan, clive).
```

and give the query:

```
|?- parent(X,Y).
```

We would expect `X` to be instantiated to `alan` and `Y` to be instantiated to `clive` when the query succeeds. We would say that the term `parent(X,Y)` unifies with the term `parent(alan, clive)` with `X` bound to `alan` and `Y` bound to `clive`. The unification algorithm in Prolog is roughly this:

Definition 1 *Given two terms T_1 and T_2 which are to be unified:*

- *If T_1 and T_2 are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.*
- *If T_1 is a variable then instantiate T_1 to T_2.*
- *Otherwise, If T_2 is a variable then instantiate T_2 to T_1.*
- *Otherwise, if T_1 and T_2 are complex terms with the same arity (number of arguments), find the principal functor F_1 of T_1 and principal functor F_2 of T_2. If these are the same, then take the ordered set of arguments of $\langle A_1, \cdots, A_N \rangle$ of T_1 and the ordered set of arguments $\langle B_1, ..., B_N \rangle$ of T_2. For each pair of arguments A_M and B_M from the same position in the term, A_M must unify with B_M.*
- *Otherwise fail.*

For example: applying this procedure to unify `foo(a,X)` with `foo(Y,b)` we get:

- `foo(a,X)` and `foo(Y,b)` are complex terms with the same arity (2).
- The principal functor of both terms is `foo`.
- The arguments (in order) of `foo(a,X)` are `a` and `X`.
- The arguments (in order) of `foo(Y,b)` are `Y` and `b`.
- So `a` and `Y` must unify , and `X` and `b` must unify.
  - `Y` is a variable so we instantiate `Y` to `a`.

  - X is a variable so we instantiate X to b.

- The resulting term, after unification is foo(a,b).

The built in Prolog operator '=' can be used to unify two terms. Below are some examples of its use. Annotations are between ** symbols.

```
| ?- a = a.        ** Two identical atoms unify **
yes
| ?- a = b.        ** Atoms don't unify if they aren't identical **
no
| ?- X = a.        ** Unification instantiates a variable to an atom **
     X=a
yes
| ?- X = Y.        ** Unification binds two differently named variables **
     X=_125451     ** to a single, unique variable name **
     Y=_125451
yes
| ?- foo(a,b) = foo(a,b).       ** Two identical complex terms unify **
yes
| ?- foo(a,b) = foo(X,Y).       ** Two complex terms unify if they are **
     X=a                        ** of the same arity, have the same principal**
     Y=b                        ** functor and their arguments unify **
yes
| ?- foo(a,Y) = foo(X,b).       ** Instantiation of variables may occur **
     Y=b                        ** in either of the terms to be unified **
     X=a
yes
| ?- foo(a,b) = foo(X,X).       ** In this case there is no unification **
no                              ** because foo(X,X)  must have the same **
                                ** 1st and 2nd arguments **
| ?- 2*3+4 = X+Y.       ** The term 2*3+4 has principal functor + **
     X=2*3              ** and therefore unifies with X+Y with X instantiated**
     Y=4                ** to 2*3 and Y instantiated to 4 **
yes
| ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like other terms **
     X=a
     Y=b
     Z=c
yes
| ?- [a,b,c] = [X|Y].   ** Unification using the '|' symbol  can be used **
```

```
     X=a                ** to find the head element, X, and tail list, Y, **
     Y=[b,c]            ** of a list **
yes
| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't have to be **
     X=a                ** restricted to finding the first head element **
     Y=b                ** In this case we find the 1st and 2nd elements **
     Z=[c]              ** (X and Y) and then the tail list (Z) **
yes
| ?- [a,b,c] = [X,Y,Z|T].       ** This is a similar example but here **
     X=a                        ** the first 3 elements are unified with **
     Y=b                        ** variables X, Y and Z, leaving the **
     Z=c                        ** tail, T, as an empty list [] **
     T=[]
yes
| ?- [a,b,c] = [a|[b|[c|[]]]].  ** Prolog is quite happy to unify these **
yes                             ** because they are just notational **
                                ** variants of the same Prolog term **
```

5 Expressing disjunctive subgoals

You have already seen how to represent alternative ways of satisfying a particular goal by giving more than one clause which could match that goal. For example, we could write a program to decide whether or not any number was between two other numbers using the following two clauses:

```
between(N, N1, N2):-
    N > N1,
    N < N2.
between(N, N1, N2):-
    N < N1,
    N > N2.
```

This program states that `N` is between two numbers `N1` and `N2` if *either* `N` is greater than `N1` and `N` is less than `N2` *or* `N` is less than `N1` or `N` is greater than `N2`. This could be stated more succinctly using the special disjunction operator, `';'`, which is often interpreted in English as ”or”.

```
between(N, N1, N2):-
    (N > N1,
     N < N2) ;
    (N < N1,
     N > N2).
```

The brackets round the conjunction of inequalities on each side of the disjunction operator aren't really necessary. I have included them to emphasise the nested structure of the term. This new definition of `between/3` using the `';'` operator is identical in meaning to the original definition using two separate clauses.

6 Evaluating numerical expressions

To Prolog a numerical expression is just a structure which can be manipulated just like any other term. So the expression `2 + 3 * 4` can be "unpacked" like any other term. For example, the goal:

```
|?- 2+3*4 = X+Y.
```

Unpacks `2+3*4` into subterms `X` (instantiated to `2`) and `Y` (instantiated to `3*4`). If you want to evaluate a numerical expression, such as the one given above, then you must use the built in operator: `is`. This is an infix operator which takes on its right side a numerical expression to be evaluated and the result of evaluating this expression is the number on its left side. For example, to find the result of evaluating the expression `2+3*4` we give the goal:

```
|?- X is 2+3*4.
    X=14
yes
```

We don't have to put a variable on the left of the `is` operator. We could use it to test that a particular number is the result of evaluating a given numerical expression:

```
|?- 14 is 2+3*4.
yes
```

Only certain special operators are allowed in evaluable expressions (i.e. terms which appear on the right side of `is` goals). These include the normal arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). Some dialects of Prolog also allow operators such as `sin/1`, `cos/1`, *etc.*

Just to emphasise that numerical expressions can be picked up from the database and passed around like any other expression, only being evaluated when you want it to, consider the following program: I represent data expressions using the following two facts:

```
data(2).
data(3*4).
```

I then give a rule which will calculate the sum of any pair of data expressions contained in the database:

```
calc(N):-
    data(N1),
    data(N2),
    N is N1 + N2.
```

I can then find the sums of all combinations of data expressions by giving the goal `calc(N)` and forcing backtracking for alternative solutions using the '`;`' operator:

```
| ?- calc(N).
     N=4 ;
     N=14 ;
     N=14 ;
     N=24
yes
```

7 Negation

You may want to make the success of a particular goal contingent on the failure of some subgoal. This can be done using the built in `\+` operator. For example, you may remember that in the `ancestors.pl` file there is a predicate `possible_to_marry(A,B)` which states that "It is possible for A to marry B if A is male and B is female and A is not related to B". The code for this is:

```
    possible_to_marry(A, B):-
        male(A),
        female(B),
        \+ related(A, B).
```

Note that `\+` is a prefix operator which, in this case, takes `related(A,B)` as its argument. An important point to note about negation in Prolog is that it doesn't mean that the goal you have negated is proved false in the logical sense. It does mean that the goal can not be established from the current set of axioms in the database. The implicit assumption here is that the set of axioms in the current database are a complete description of every aspect of the problem. This is often referred to as the closed world assumption. This rather liberal interpretation of negation can be used to express defaults. For example we might have a problem in which we could identify all females by name, using the predicate `female/1` to denote that someone is female:

```
female(alice).
female(betty).
    ...etc.
```

and, given that we have recorded the names of all females, we might want to assume that anyone who wasn't a known female was male. We could do this using the rule:

```
male(X):-
    \+ female(X).
```

Now the goal `|?- male(alice).` will fail because `female(alice)` is true and so `\+ female(alice)` must fail. Conversely, the goal `|?- male(fred).` will succeed because there is no way to establish `female(fred)` and so `\+female(fred)` succeeds. However, you have to be careful when exploiting the closed world assumption in this way because someone may subsequently run your program without being aware of this limitation. For example, I happen to know that `'minnie mouse'` is female but the program in our example will happily allow the goal `|?- male('minnie mouse').` to succeed.

8 Cuts

Prolog is good at backtracking but uncontrolled backtracking can be a nuisance. The cut symbol (written as `!`) is a special subgoal used to provide a limited form of control over backtracking, by committing the Prolog interpreter to a particular portion of the potential search space. An example will help start to explain how this works. The code for this example can be found in the file `cuts.pl` and further discussion appears in [Sterling & Shapiro 86].

We start with the following definition of `minimum/3`:

```
% minimum(+Number1, +Number2, ?Minimum).
% Succeeds if Minimum is the minimum value of Number1 and Number2.

minimum(X, Y, X):-
    X =< Y.
minimum(X, Y, Y):-
    X > Y.
```

The potential search tree for the goal `minimum(1,2,X)` is:

```
                 minimum(1, 2, X)
                  /           \
                 /             \
        minimum(1, 2, 1)    minimum(1, 2, 2)
             /                   \
            /                     \
           /                       \
        1 =< 2                    1 > 2
```

In other words, the Prolog interpreter, in trying to satisfy `minimum(1,2,X)` would first test whether `1 =< 2`, which succeeds giving the result `minimum(1,2,1)`. However, if forced to backtrack it would also try the second clause and try to establish `1 > 2`, which will fail. It would be useful to have some way of telling the Prolog interpreter, once it had established `1 =< 2` that there is no point in looking for any other ways of finding a solution. This information can be given by putting a cut after the first subgoal of the first clause, as shown in `gc_minimum/3`.

```
%  gc_minimum(+Number1, +Number2, ?Minimum).
%Identical to minimum/3 but with a cut in the first clause.

gc_minimum(X, Y, X):-
    X =< Y, !.
gc_minimum(X, Y, Y):-
    X > Y.
```

The potential search tree for the goal `gc_minimum(1,2,X)` is:

```
                 gc_minimum(1, 2, X)
                  /           \
                 /             *<--- This branch is pruned from the tree
                /               \    once the ! has been reached.
   gc_minimum(1, 2, 1)   gc_minimum(1, 2, 2)
              /                   \
             /                     \
            /                       \
         1 =< 2, !                 1 > 2
          /
         /
        !
```

Notice that this doesn't alter the set of results which can be obtained from running the program. All it does is pre–empt search which would be fruitless. Having put the cut into `gc_minimum/3` we might go one step further and remove the `X > Y` test in the second subgoal. The argument for doing this might be that if we know that the cut in the first clause will prevent the second clause ever being tried if `X =< Y` then the only condition under which the second clause is reached is when `X > Y`, so we don't need to make this test explicit. By making this change we get `rc_minimum/3` as shown below:

```
% rc_minimum(+Number1, +Number2, ?Minimum).
% Like to gc_minimum/3 but with the check in the second clause
% left out, because the programmer thinks it isn't needed (given
% that Y should be the chosen minimum if X isn't).
```

```
rc_minimum(X, Y, X):-
    X =< Y, !.
rc_minimum(_, Y, Y).
```

The potential search tree for the goal `rc_minimum(1,2,X)` is:

```
                 rc_minimum(1, 2, X)
                  /           \
                 /             *<--- This branch is pruned from the tree
                /               \    once the ! has been reached.
   rc_ minimum(1, 2, 1)   rc_minimum(1, 2, 2)
              /
             /
            /
         1 =< 2, !
          /
         /
        !
```

<u>But</u> there is a major flaw in this reasoning, as shown by calling the goal `minimum(1, 2, 2)`, which will (erroneously) succeed, since the first clause doesn't match and the cut is therefore never reached. Thus the intended meaning of `rc_minimum/3` is different from that of `minimum/3`. Some people refer to cuts which don't alter the meaning of the program as "green cuts" and cuts which do alter the meaning of the program as "red cuts".

8.1 Using Cuts to Specify Defaults

Cuts are often used to give a range of behaviours of the program under special circumstances, with a "catch–all" default definition at the end. Consider, ofr example, the following program, which is discussed in [Sterling & Shapiro 86]. We want to define a predicate which will be able to determine a pension for some person (`X`), given certain characteristics of `X`. The default is `nothing` but, before that we put a series of tests to see if `X` qualifies for any pension. The code we use is:

```
pension(X, invalid_pension):-
    invalid(X), !.
pension(X, old_age_pension):-
    over_65(X),
    paid_up(X), !.
pension(X, supplementary_benefit):-
    over_65(X), !.
pension(_, nothing).    % This is the default if none of the above succeed.
```

Let's also define some characteristics for a sample of people.

```
invalid(fred).

over_65(fred).
over_65(joe).
over_65(jim).

paid_up(fred).
paid_up(joe).
```

This works fine for queries such as `pension(fred,P)`, which would instantiate `P` to `invalid` and return no further results. However, there are ways of forcing the program to give incorrect recommendations. In particular, the query `pension(fred,nothing)` would succeed. How can this be fixed? One way is to make it explicit that the default of `nothing` only applies when there is no possible pension. In the code below, this is achieved by inventing a new predicate, `entitlement/2` which makes this distinction.

```
entitlement(X, Y):-
    possible_pension(X, Y).
entitlement(X, nothing):-
    \+ possible_pension(X, _).

possible_pension(X, invalid_pension):-
    invalid(X).
possible_pension(X, old_age_pension):-
    over_65(X),
    paid_up(X).
possible_pension(X, supplementary_benefit):-
    over_65(X).
```

8.2 Just One More Thing About Cuts

At the start of the section on cuts, I said that they pruned the search space by telling the interpreter, once it has hit a cut, to forget about any other definitions for the predicate in which the cut appears. The cut also has another effect: it tells the interpreter not to look for any new solutions to any subgoals appearing in the body of the clause before the cut. The best way to explain this is by using an example, which you can find in `morecuts.pl`.

Let's start with some test data:

```
baz(1, 2).
baz(2, 3).
baz(3, 4).

bar(3).
bar(4).

eek(3, a).
eek(3, b).
eek(4, c).
```

Now we define a predicate, `foo/2`, as follows:

```
foo(A, B):-
        baz(A, B),
        bar(B), !.
```

The cut at the end of this clause will commit to a single solution for `baz(A, B)` and for `bar(B)`, so the behaviour of this program is:

```
| ?- foo(A, B).
     A=2
     B=3 ;
no
```

Let's now invent a new predicate, `foo1/3`:

```
foo1(A, B, C):-
        baz(A, B),
        bar(B), !,
        eek(B, C).
```

This has an extra subgoal after the cut, so the subgoals before the cut will be "frozen", as before, but the interpreter is free to generate new solutions for the subgoal after the cut. Thus the behaviour is:

```
| ?- foo1(A, B, C).
     A=2
     B=3
     C=a ;
     A=2
     B=3
     C=b ;
no
```

9 Some Common Techniques

To become a skilled Prolog programmer requires, above all else, **practice**. However, there are a few standard ways of constructing a program which give some overall structure to the task. Some of these are described below:

Technique 1 *Recursion, decomposing a structure. The key idea is to separate base cases from recursive cases, then make sure that the recursive subgoals all tend towards one or more of the base cases. The general form of this technique is:*

$P(BaseCase)$.
$P(Structure)$:–
        $P(SmallerStructure)$.

For example:

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

Technique 2 *Recursion, building up a structure in recursive subgoals. Typically two arguments are used: the first storing the accumulated result and the second being used to "pass back" the final state of the accumulator.*

$P(Final, Final)$.
$P(Sofar, Final) : -$
        $Update(Sofar, NewSofar)$,
        $P(NewSofar, Final)$.

For example:

```
reverse([], Final, Final).
reverse([H|T], Sofar, Final):-
     reverse(T, [H|Sofar], Final).
```

Technique 3 *Recursion, building up a structure in the head of each clause. This normally involves just one extra argument which contains one or more variables to be instantiated from successful recursive subgoals.*

$P(BaseCase)$.
$P(Final)$:–
        $P(Partial)$,
        $Update(Partial, Final)$.

For example:

```
numbers_in_term(A, [A]):- number(A).
numbers_in_term(A + B, Final):-
    numbers_in_term(A, PA),
    numbers_in_term(A, PB),
    append(PA, PB, Final).
```

Technique 4 *Committment by cases. If the problem is one which involves a number of mutually exclusive cases then these can be tackled using a sequence of clauses with cuts placed at the points of committment.*

$P(Term)$:–
    $Test1(Term), !,$
    $P1(Term).$
$\vdots$
$P(Term)$:–
    $TestN(Term), !,$
    $PN(Term).$

For example:

```
has_type(X, variable):-
    var(X), !.
has_type(X, atom):-
    atom(X), !.
has_type(X, number):-
    number(X), !.
has_type(X, structure):-
    nonvar(X),
    functor(X, _, N), N > 0.
```

Technique 5 *Failure driven loops to force all solutions. This is often used in combination with printing routines.*

P :–
    $Goal,$
    $fail.$
$P.$

For example:

```
all_solutions(Goal):-
    Goal,
    write(Goal), nl,
    fail.
all_solutions(_).
```

Technique 6 *Repeat loops to perform some task until a terminating condition is obtained. These are often used as replacements for recursive predicates in cases where a simple cycle of repeating and testing is required.*

```
P :-
        repeat,
        Goal,
        Test, !.
```

For example:

```
read_until(Prompt, Test):-
    repeat,
    write(Prompt), read(Answer),
    ((Test, !) ; fail).
```

Technique 7 *Cut and fail to ensure no solutions on particular cases.*

```
P :-
        UndesirableCase,
        !, fail.
⟨ OTHER DEFINITIONS OF P. ⟩
```

For example:

```
no_numbers([H|_]):-
    number(H),
    !, fail.
no_numbers([_|T]):-
    no_numbers(T).
no_numbers([]).
```

Technique 8 *Double negation to apply a test without binding any variables in the terms used during the test.*

```
P(Term) :-
        \+ \+ P1(Term).
```

For example:

```
unifiable(Term1, Term2):-
    \+ \+ Term1 = Term2.
```

10 Efficiency Considerations

The quality of a program depends on 3 main metrics:

1. How easy it is to read and understand.
2. The speed with which it performs a given task.
3. The amount of space it needs to perform the task.

Metric 1 is often the most important for beginners but 2 and 3 are always hanging around in the background and need to be considered explicitly. It isn't easy to lay down precise guidelines for 1, 2 or 3 because they often interact – a program which is easy to read may not be maximally fast to run – so it is often necessary to strike a balance. Where this balance lies will depend on the demands of the application for which the program is required. The following are a few useful heuristics (for more detailed discussion of this see [Covington 91]):

Useful tip 1 *Be careful how you order your clauses. The order of clauses and/or subgoals in a Prolog program with a declarative interpretation doesn't affect its declarative meaning but often affects the way the program is executed. For example, the ancestor relation from the program in file* `ancestors.pl` *was defined as follows:*

```
ancestor(A, B):-
    parent(A, B).
ancestor(A, B):-
    parent(P, B),
    ancestor(A, P).
```

By switching the ordering of the clauses and/or subgoals in this definition we can obtain 3 different ways of defining the ancestor relation:

- `ancestor1` *has the two clauses switched around.*
- `ancestor2` *has the order of subgoals switched around.*
- `ancestor3` *has the order of both clauses and subgoals switched around.*

This provides the definitions shown below (you can find these in the file `declar_v_proc.pl`*):*

```
ancestor1(A, B):-
    parent(P, B),
    ancestor1(A, P).
ancestor1(A, B):-
    parent(A, B).

ancestor2(A, B):-
    parent(A, B).
ancestor2(A, B):-
    ancestor2(A, P),
    parent(P, B).

ancestor3(A, B):-
    ancestor3(A, P),
    parent(P, B).
ancestor3(A, B):-
    parent(A, B).
```

All of these variants of `ancestor/2` *have the same declarative meaning; the order of clauses or subgoals doesn't affect this. However, the way they are executed by the Prolog interpreter is very different, giving a different external behaviour of the program. Consider the following transcript of a Prolog session, in which I first consult the* `ancestors.pl` *file and then the* `declar_v_proc.pl` *file. I then give the goal to find some ancestor of* `dave` *using each of the definitions in turn :*

```
| ?- ['declar_v_proc.pl'].

ancestors.pl consulted:   3956 bytes      6.52 seconds

declar_v_proc.pl reconsulted:   1180 bytes      2.28 seconds

yes
| ?- ancestor(X, dave).

     X=clive ;

     X=clarissa ;

     X=alan ;

     X=andrea ;
     X=bruce ;

     X=betty
yes
| ?- ancestor1(X, dave).

     X=alan ;

     X=andrea ;

     X=bruce ;

     X=betty ;

     X=clive ;

     X=clarissa
yes
| ?- ancestor2(X, dave).

     X=clive ;

     X=clarissa ;

     X=alan ;

     X=andrea ;

     X=bruce ;

     X=betty
yes
| ?- ancestor3(X, dave).
Resource error: insufficient memory
```

Although `ancestor1` *and* `ancestor2` *return the same values for* `X`*, these values are returned in a different order. This is because in* `ancestor1` *the recursive clause appears before the base case, so the interpreter always tries to find a solution "the long way" before considering the shorter alternative. In* `ancestor3`*, the combination of putting the recursive case first and putting the* *recursive call to* `ancestor3` *before the step to parent causes the interpreter immediately to descend through an endless chain of subgoals until it eventually runs out of memory and drops dead.*

The moral of the story is : Your Prolog program may have a nice declarative interpretation but if you want it to execute you must often consider its procedural aspects.

Useful tip 2 *Be careful how you call your predicates. In the* `colour_countries/1` *example in* `mapcolour.pl` *the order in which countries are coloured makes a big difference to the speed of solution. You can test this by changing the order of clauses for* `ngb/2`*, which gives the neighbouring countries. The general principle is to select the (potentially) most difficult countries and colour them first. The ones likely to be most difficult are the ones with most neighbours. (See [Bratko 86] for a detailed discussion of efficiency for the map colouring problem).*

Useful tip 3 *Take advantage of tail recursion optimisation. This is a technique built into many Prolog systems which allows the space allocated to the head of a clause to be used for the last subgoal, provided that there is no chance of further results on backtracking by the time the last subgoal gets called. Consider, for example, the* `append/3` *predicate:*

```
append([H|T], L1, [H|L2]):-
    append(T, L1, L2).
append([], L, L).
```

For this predicate, if we are trying to satisfy a goal such as:

```
| ?- append([a,b], [c,d], X).
```

and have managed to match on the head of the first clause and are at the point of calling the last (and only) subgoal of that clause then we know that there would be no other ways of finding a solution (since there are no other subgoals within the clause and the 2nd clause wouldn't match to the goal). If the Prolog interpreter is smart enough to spot this, it can get rid of any choice points for the current goal (`append([a,b], [c,d], X)`*) and replace it in memory with the outstanding subgoal (*`append([b], [c,d], X)`*). This means that a solution to the* `append/3` *goal can be found using constant memory space, rather than requiring space which increases linearly with the depth of recursion. Of course, this all depends on the interpreter being able to recognise when tail recursion optimisation can be applied - a task which isn't always easy. The* `append/3` *example, above, is comparatively simple but even it isn't straightforward. Suppose that the goal had been:*

```
| ?- append(X, Y, [a,b]).
```

Then tail recursion optimisation wouldn't apply because the program isn't deterministic at the point where the subgoal of the 1st clause gets called. Some Prolog systems have fancy indexing systems which allow them to detect determinism quite effectively. However, there is one way in which you can make it clear to the Prolog interpreter that no choice points remain at the point where the last subgoal of a clause is about to be called....add a cut. For example, suppose we have the following program which is designed to gobble up space on the machine:

```
gobble_space:-
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    statistics(local_stack, L), write(L), nl,
    gobble_space.
gobble_space.

gobble.
gobble:-
    gobble_space.
```

The behaviour of this predicate is as follows:

```
| ?-  gobble_space.
[380,16000]
[652,15728]
......SOME TIME LATER.....
[4190812,2468]
[4191084,2196]
Resource error: insufficient memory
```

i.e. It eats up a lot of space and eventually exceeds the limits of the system. Now let's try the same program but with a cut in the first clause of `gobble_space/0` *to permit tail recursion optimisation.*

```
gobble_space:-
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    statistics(local_stack, L), write(L), nl, !,
    gobble_space.
gobble_space.
```

The behaviour of this predicate is:

```
| ?- gobble_space.
[380,16000]
[380,16000]
......SOME TIME LATER.....
[380,16000]
[380,16000]
```

i.e. It doesn't eat up space and will continue to run indefinitely (or at least for a very long time).

Useful tip 4 *Use cuts to remove choice points.*

Useful tip 5 *But don't use cuts indiscriminately. Here is an example taken from [O'Keefe 90]. The plan is to define a predicate,* `match1/2`*, which will take as its first argument a pattern for a sequence (with some "holes" in it) and take as its second argument a complete sequence of symbols. The predicate should succeed if the pattern can be matched to the sequence. For example:*

```
| ?- match1([i,X,my,Y], [i,like,my,lecturing,job]).
X = [like],
Y = [lecturing,job]
```

One way to define it is as follows:

```
match1([], []).
match1([H|T], [F|R]):-
    nonvar(H), !,
    H = F,
    match1(T, R).
match1([V|T], [F|R]):-
    match1(T, R),
    V = [F].
match1([V|T], [F|R]):-
    match1([New|T], R),
    V = [F|New].
```

This works, but is not as fast as the following definition, named `match2/2`*, which has no cuts but distinguishes the type of symbol in the pattern list using* `w(X)`*, if* `X` *is a word, and* `s(X)`*, if* `X` *is a segment of the sequence.*

```
match2([], []).
match2([H|T], [F|R]):-
    match2(H, T, F, R).

match2(w(Word), Items, Word, Words):-
    match2(Items, Words).
match2(s([Word|Seg]), Items, Word, Words0):-
    append(Seg, Words1, Words0),
    match2(Items, Words1).
```

The behaviour of this predicate is as follows:

```
| ?- match2([w(i),s(X),w(my),s(Y)], [i,like,my,lecturing,job]).
X = [like],
Y = [lecturing,job]
```

Useful tip 6 *For some tasks you can save space by using failure driven loops. For example, suppose we want to write a predicate,* `read_file/1`*, which will read each clause,* `C`*, from a given file and assert them into the database as a series of facts of the form* `cl(C)`*. The first part of this predicate takes care of opening the file; calling the subgoal,* `read_all_clauses`*, which does the reading and asserting; and closing the file at the end.*

```
read_file(File):-
    open(File, read, Stream),
    set_input(Stream),
    read_all_clauses,
    close(Stream).
```

We could write a recursive definition of `read_all_clauses/0`*:*

```
read_all_clauses:-
    read(Clause),
    ((Clause = end_of_file, !) ;
     (assert(cl(Clause)), read_all_clauses)).
```

But a failure driven version will be more efficient:

```
read_all_clauses:-
    repeat,
    read(Clause),
    ((Clause = end_of_file, !) ;
     (assert(cl(Clause)), fail)).
```

Useful tip 7 *Don't duplicate effort. If you are going to need to do the same, involved calculation a number of times then it is probably worth storing the result of the first calculation for later use.*

11 Definite Clause Grammars

DCGs are a special notation provided in most Prolog systems which provide you with a convenient way of defining grammar rules. The general form of each DCG clause is as follows:

```
Head --> Body
```

meaning “A possible form for Head is Body”. `Head` and `Body` are Prolog terms and `‘-->’` is an infix operator. Examples of DCGs for specific tasks are given in file `dcg.pl`.

The most common use of DCGs is to parse (or generate) some list of symbols according to valid “rules of grammar” defined by the DCG clauses. DCGs can be used for purposes other than parsing (there is an example of such a use in `dcg.pl`) but for the purposes of this explanation we will stick with the standard parsing application.

Agree to call the symbols (usually words) in the parsed sentence ”terminal symbols”. Also agree to call symbols which represent higher level language constructs ”non-terminal symbols”.

- A non-terminal symbol may be any Prolog term, other than a variable or number.
- A terminal symbol may be any Prolog term.
- DCG heads consist of a non-terminal symbol, optionally followed by a sequence of terminal symbols.
- DCG bodies may contain any sequence of terminal or non-terminal symbols, separated by the `’,’` (conjunction) and/or `’;’` (disjunction) operators.
- In the body of a DCG clause, terminal symbols are distinguished from non-terminal symbols by being enclosed within a Prolog list: `[T1,T2,...TN]` An empty sequence of terminal symbols is represented by the empty list `[]`.
- Extra conditions, in the form of Prolog procedure calls, may be included in the body of DCG clause. These are distinguished by being enclosed in curly brackets: `{P1,P2,...PN}`
- The cut symbol can be included in the body of a DCG clause and doesn’t have to be enclosed within curly brackets.

11.1 How do DCGs relate to standard Prolog clauses ?

DCGs are really just a convenient abbreviation for ordinary Prolog clauses. Each DCG clause is translated into a standard Prolog clause as it is loaded. The procedural interpretation of a grammar rule is that it takes an input list of symbols, analyses some initial portion of that list and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output lists are not written explicitly in a grammar rule but are added when the rule is translated into an ordinary Prolog clause. For example, the DCG rule:

```
p(X, Y) --> q(X), r(X, Y), s(Y).
```

translates into:

```
p(X, Y, Input, Output):-
    q(X, Input, Out1),
    r(X, Y, Out1, Out2),
    s(Y, Out2, Output).
```

Terminal symbols are translated using the built in predicate :

```
'C'(Input, Symbol, Remainder)
```

which nips the given Symbol from the Input list, leaving the Remainder of the list of terminal symbols. It is defined by the single clause:

```
'C'([X|Remainder], X, Remainder).
```

This predicate isn't normally accessed by the programmer, hence the obscure name. This gives us the following translation for a DCG rule containing both terminal and non-terminal symbols:

```
p(X) --> [go,to], q(X), [stop].
```

is translated into:

```
p(X, Input, Output):-
    'C'(Input, go, Out1),
    'C'(Out1, to, Out2),
    q(X, Out2, Out3),
    'C'(Out3, stop, Output).
```

Extra conditions expressed as procedure calls just appear as themselves, without any extra translation, so:

```
p(X) --> [X], {integer(X), X > 0}, q(X).
```

is translated into:

```
p(X, Input, Output):-
    'C'(Onput, X, Out1),
    integer(X),
    X > 0,
    q(X, Out1, Output).
```

11.2 Finding out for yourself

If you want to experiment with DCGs here are a few things you should know:

There are two ways to call a procedure which you have defined using DCGs:

- Using the built in predicate `phrase/2`. This takes as its first argument the highest level DCG rule head to apply (think of this as being like a top level Prolog goal) and as its second argument the list of terminal symbols which can be parsed using the group of DCG clauses referenced by the term in the first argument. See file `dcg.pl` for examples of how this can be used.
- By just calling the translated clauses directly (see details of translation above).

If you want to see how the Prolog system will translate your DCGs into ordinary clauses there are two ways in which you can do this:

- Call the built in predicate `expand_term/2` with the first argument containing a DCG clause. It will return the expanded translation as its second argument. For example:

```
| ?- expand_term(a --> b, X).
     X = a(_124,_133):-b(_124,_133)
```

- Load a DCG from a file and then call the built in predicate `listing/0` which will list out all the loaded clauses in translated form.

References

[Bratko 86] I. Bratko. *Prolog Programming for Artificial Intelligence (2nd edition).* Addison Wesley, 1986. ISBN 0-201-41606-9.

[Covington 91] M.A. Covington. Efficient prolog: a practical tutorial. *Artificial Intelligence Review*, 5:273–287, 1991.

[O'Keefe 90] R. O'Keefe. *The Craft of Prolog.* MIT Press, 1990. ISBN 0-262-15039-5.

[Sterling & Shapiro 86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986. ISBN 0-262-69105-1.