Prolog

Sup A2
Loiseleur Michel and Vigier Nicolas

February 15, 2001

Contents

P rolog was invented in the early seventies at the University of Marseille. Prolog stands for PROgramming in LOGic. It is a logic language that is particularly used by programs that use non-numeric objects. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols is a common task. Prolog differs from the most common programmings languages because it is declarativre language. Traditional programming languages are said to be procedural. This means that the programmer specify how to solve a problem. In declarative languages the programmers only give the problem and the language find himself how to solve the problem.

Although it can be, and often is, used by itself, Prolog complements traditional languages when used together in the same application.

1 Presentation of Prolog

1.1 What is a Prolog program

Programming in Prolog is very different from programming in a traditional procedural language like Pascal. In Prolog you don't say how the program will work.
Prolog can be separated in two parts :

1.1.1 The Program

The program, sometimes called Database is a texte file (*.pl) that contain the facts and rules that will be used by the user of the program. It contains all the relations that make this program.

1.1.2 The Query

When you launch a program you are in query mode. This mode is represented by the sign `? -` at the begining of the line. In query mode you ask questions about relations described in the program.

1.2 Loading a program

First you have to launch your Prolog compiler, for this report we used the SWI-Prolog wich is a freeware (if you want to know more about SWI-Prolog go to the page 24). When Prolog is launched the `?-` should appear meaning you are in query mode. The manier to launch a program depends of your compiler. For SWI-Prolog you can load a program by typing the cammand `[file].` when the file of your program is `file.pl`. If you compiler is not SWI-Prolog you can also try the command `reconsult(file).` When you have done this you can use all the facts and rules that are contained in the program. Now let's begin to see what is a fact...

2 The Facts

2.1 Simple facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. For example we can represent the fact that it is sunny by writing the program :

```
sunny.
```

We can now ask a query of Prolog by asking

```
?- sunny.
```

?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.
Facts have some simple rules of syntax. Facts should always begin with a lowercase letter and end with a full stop. The facts themselves can consist of any letter or number combination, as well as the underscore `_` character. However, names containing the characters `-,+,*,/,` or other mathematical operators should be avoided.

2.2 Facts with arguments

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation(<argument1>,<argument2>,....,<argumentN> ).
```

The arguments can be any legal Prolog term. The basic Prolog terms are an integer, an atom, a variable or a structure. Various Prolog implementation enhance this basic list with other data types, such as floating point numbers, or strings.
Exemple :

```
likes(john,mary).
```

In the above fact john and mary are two atomes. Atoms are usally made from letters and digits with lowercase characters. The underscore (_) can also be used to separe 2 words but is not allowed as the first charactere. Atoms can also be legally made from symbols. The followings are legal atoms :

```
hello
zz42
two_words
====>
```

The followings are not legal atoms :

```
Hello
4hello
_Hello
two words
two-words
```

You can use single quotes to make any character combination a legal atom.

```
'two words'
'UpperCase'
'12444'
'etc...'
```

The fact likes(john,mary). say that there is a relation between john and mary. It can be read as either john likes mary or mary likes john. This reversibility can be very useful to the programmer, however it can also be a source of mistakes. You have to be clear on how you intend to interpret the relation.
The number of arguments is the arity of the predicate. A predicate with 2 arguments will be called by predicate_name/2. You can have differents predicats with the same name if they have a different arity.

2.3 How to query

Once you have entered the facts in a program you can ask prolog about it. An exemple programm can be :

```
eats(fred,oranges).     /* ``Fred eats oranges'' */

eats(tony,apple).       /* ``Tony eats apple''   */

eats(john,apple).       /* ``John eats apple''   */
```

If we now ask some queries we would get the followings things :

```
?- eats(fred,oranges).
/* does this match anything in the database? */

yes
/* yes, that matchs the first clause */

?- eats(john,apple).

yes

?- eats(mike,apple).

no
/* there is no relation between mike and apple */
```

3 Variables and Unification

3.1 Simples unifications

How can we ask something like “what does Fred eat ?” If we have the following program :

```
eats(fred,oranges).
```

How do we ask what fred eats ? We could write something like this :

```
?- eats(fred,what).
```

But Prolog will say no. The reason is that Prolog can’t find the relation eats(fred,what) in hi database. In this case we have to use a variable wich will be unified to match a relation given in the program. This process is known as unification.
Variables are distinguished from atoms by starting with a capital letter. Here are some exemples of variables :

```
X           /* a single capital letter                    */

VaRiAbLe    /* a word beginning with an upper case letter */

Two_words   /* two words separated with an underscore     */
```

Now that we know how to use a variables, we can ask the same question as before using the variable What instead of an atom.

```
?- eats(fred,What)

What=oranges

yes
```

In this case Prolog try to unifie the variable with an atom. “What=oranges” means that the query is successfull when What is unified with oranges.
With the same program if we ask :

```
?- eats(Who,oranges).
```

In this example we ask who eats oranges. To this query Prolog should answer :

```
?- eats(Who,oranges).

Who=fred

yes
```

Now if we ask :

```
?- eats(Who,apple).

no
```

Prolog answer no because he can't find in his database any atom than can match with this relation.
Now if we only want to know if something is eated by anyone and we don't care about that person we can use the underscore ('_'). The '_' can be used like any variable.
For example if we ask eats(fred,_) the result will be :

```
?- eats(fred,_).

yes
```

The result will be yes because Prolog can find a relation of eat between fred and something. But Prolog will not tell use the value of '_'.
Now if we have the following program :

```
eats(fred,apple).

eats(fred,oranges).
```

Now if we ask :

```
?- eats(fred,What).
```

The first answer will be "What=apple" because that is the unification that match the first relation of eats with fred in the database. Then prolog will be waiting for you to press a key. If you press enter Prolog will be ready for a new query. In most implemetation if you press the key ";" then Prolog will try to find if there is any other successful unification. Prolog will give the second result "What=orange" because this the second one in the program. If you press again the key ";" then Prolog will try to find a third unification. The result will be "no" because he is not able to find any other successful unification.

```
?- eats(fred,What).

What=apple ;

What=oranges ;

no
```

3.2 Variables unification example

Consider this exemple of program that could be used by a library :

```
book(1,title1,author1);

book(2,title2,author1);

book(3,title3,author2);

book(4,title4,author3);
```

Now if we want to know if we have a book from the author2 we can ask :

```
?- book(_,_,author2).

yes
```

If we want to know which book from the author1 we have :

```
?- book(_,X,author1).

X=title1 ;

X=title2 ;
```

4 Rules

4.1 Rules

Consider the following sentence : 'All men are mortal' We can express this thing in Prolog by :

```
mortal(X) :- human(X).
```

The clause can be read as 'X is mortal if X is human'.
To continue with this example, let us define the fact that Socrate is a human. Our program will be :

```
mortal(X) :- human(X).

human(socrate).
```

Now if we ask to prolog :

```
?- mortal(socrate).
```

Prolog will respond :

```
yes
```

In order to solve the query -? mortal(socrates). Prolog will use the rule we have given. It says that in order to prove that someone is mortal we can prove that he is human. So from the goal mortal(socrate) Prolog generate the subgoal human(socrate).
We can still use variables. For example we might want to know who is mortal :

```
?- mortal(X).
```

Then Prolog should respond :

```
P=socrate
```

This means that Prolog was able to succed the goal by unifying the variable X to socrates. Again this was done by using the subgoal human(X).
Sometimes we may wish to specify alternatives ways to provre something. We can do this by using differents rules ands facts with the same name. For exeample, we can represent the sentence 'Something is fun if it is a PC running UNIX or am old amiga or an ice cream' with the following program :

```
fun(X) :-            /* something is fun if       */

          pc(X),    /* it is a pc and             */

          unix(X).    /* it is running unix       */


fun(X) :-            /* or it is fun if           */

          old(X),    /* it is old and             */

          amiga(X).  /* it is an amiga            */

fun(ice_cream).      /* the ice_crean is also fun */
```

This program says that there are three ways to know if an object is fun or not. Like for pure facts, Prolog will start from the first clause (a clause can be a rule or a fact) of fun and try it. If that does not succed Prolog will try the next clause. If there is no more clauses then it fails and Prolog responds ' no '. We can also see in this example that the 'and' is represented by a ',' and the 'or' by differents clause. If needed the 'or' can also be represebted by ';'. In the previous examples when we was asking eats(fred,What) and pressing the key ';' to see the following results we was in fact asking 'or'.
All identically-named variables in a rule (for example X in the last rule we've seem) are of course considered as one unic variable and must have the same instantiation for each solution in a particular query. Identical variables names in differents rules are considerated as differents variables and are totally independent, this is the same as if we had used differents names.
The following program :

```
fun(X) :-
          pc(X),
          unix(X).

fun(X) :-
          old(X),
          amiga(X).
```

Will be seen by Prolog as :

```
fun(X_1) :-
            pc(X_1),
            unix(X_1).

fun(X_2) :-
            old(X_2),
            amiga(X_2).
```

4.2 How to add a rule with a program

It is possible to add new rules or facts with the instruction Assert(fact1) which will add the fact called fact1. The predicates added with this command are considereted like any other in the source of the program.

4.2.1 The instructions

The usefull instructions are :

assert(c). Add the rule c in the database.

retract(c). Remove the c from the database.

asserta(c). Add c at the beginning of the database.

assertz(c). Add c at the end of the database.

4.2.2 Example

```
?- sunny.
no.

?- assert(sunny).
yes.

?- sunny.
yes

?- retract(sunny).
yes.

?- sunny.
no.
```

5 Backtracking

5.1 Fail

The fail/1 predicate is provided by Prolog. When it is called, it causes the failure of the rule. And this will be for ever, nothing can change the statement of this predicate. You can ask you what is its utility. We can associate it to the Cut/1 predicate, described in the next subsection in this report.it allow us to include the negation in a rule.A typical use of fail is a negation of a predicate.We can resume the fail with this sheme :

```
goal(X) :- failure(X),!,fail.
goal(X).
```

failure(X) are the conditions that make goal(X) fail.

5.2 Cut

Up to this point, we have worked with Prolog's backtracking. We have seen how to use the backtracking to write compact predicates. Sometimes it is desirable to selectively turn off backtracking. Prolog provides a predicate that performs this function. It is called the cut/1, represented by an exclamation point (!).

The cut/1 effectively tells Prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives. We will first examine the effects of the cut/1 and then look at some practical reasons to use it . When the cut/1 is encountered, it re-routes backtracking. It short-circuits backtracking in the goals to its left on its level, and in the level above, which contained the cut/1. That is, both the parent goal and the goals of the particular rule being execut/1ed are affected by the cut/1. The effect is undone if a new route is taken into the parent goal.

We will write some simple predicates that illustrate the behavior of the cut/1, first adding some data to backtrack over.

```
data(pentiumIII).
data(athlon).
```

Here is the first test case. It has no cut/1 and will be used for comparison purposes.

```
compare_cut_1(X) :-
   data(X).
compare_cut_1('last chip').
```

This is the control case, which exhibits the normal behavior.

```
?- compare_cut_1(X), write(X), nl, fail.
pentiumIII
athlon
last chip
no
```

Next, we put a cut/1 at the end of the first clause.

```
compare_cut_2(X) :-
   data(X),
   !.
compare_cut_2('last chip').
```

Note that it stops backtracking through both the data subgoal (left), and the compare_cut_2/1 parent (above).

```
?- compare_cut_2(X), write(X), nl, fail.
pentiumIII
no
```

Next we put a cut/1 in the middle of two subgoals.

```
compare_cut_3(X,Y) :-
   data(X),
   !,
   data(Y).
compare_cut_3('last chip').
```

Note that the cut inhibits backtracking in the parent compare_cut_3/2 and in the goals to the left of (before) the cut (first data). The second data/1 to the right of (after) the cut is still free to backtrack.

```
?- compare_cut_3(X,Y), write(X-Y), nl, fail.
pentiumIII - pentiumIII
pentiumIII - athlon
no
```

Performance is the main reason to use the cut/1. This separates the logical purists from the pragmatists. Various arguments can also be made as to its effect on code readability and maintainability. It is often called the 'goto' of logic programming. You will most often use the cut/1 when you know that at a certain point in a given predicate, Prolog has either found the only answer, or if it hasn't, there is no answer. In this case you insert a cut/1 in the predicate at that point. Similarly, you will use it when you want to force a predicate to fail in a certain situation, and you don't want it to look any further.

5.3 Not

For logical puritey, it is always possible to rewrite the predicates without the cut/1. This is done with the built-in predicate not/1. Some claim this provides for clearer code, but often the explicit and liberal use of 'not' clutters up the code, rather than clarifying it. When using the cut/1, the order of the rules become important. Let's try to rewrite compare_cut_2/1 with the not/1 predicate.

```
not_2(X) :- X = pentiumIII.
not_2(X) :- not(X = pentiumIII).
```

You can now see the difference between not/1 and cut/1, but the code is here to simple to see really the difference of clarity.Try with compare_cut_3 :

```
not_3(X,Y) :- X = pentiumIII,data(Y).
not_3(X,Y) :- not(X = pentiumIII),not(Y).
```

Now, you can imagine what will be the difference between the algorithms based on the cut/1 and those on the not/1. The result is the same, it's just a way of thinking.
it is interesting to note that not/1 is defined using the cut/1. It also uses call/1, another built-in predicate that calls a predicate.

```
not(X) :- call(X), !, fail.
not(X).
```

6 Recursion

6.1 What is recursion

The recursion in any language is a function that can call itself until the goal has been succeed.
In Prolog, recursion appears when a predicate contain a goal that refers to itself. As we have seen in the earlier chapters when a rule is called Prolog create a new query with new variables. So it make no difference wether a rule calls another rule or call itself.
In Prolog and in any language, a recursive definition always has at least two parts. A first fact that act like a stopping condition and a rule that call itsefl simplified. At each level the first fact is checked. If the fact is true then the recursion ends. If not the recursion continue.
A recursive rule must never call itself with the sames arguments ! If that happens then the program will never end.

6.2 Examples of Recursion

6.2.1 Example of the ancestors

```
parent(john,paul).   /* paul is john's parent    */

parent(paul,tom).    /* tom is paul's parent     */

parent(tom,mary).    /* mary is tom's parent     */

ancestor(X,Y) :- parent(X,Y).
/* If Y is a parent of X, then Y is an ancestor of X */

ancestor(X,Y) :- parent(X,Z),

                 ancestor(Z,Y).
/* if Y is an ancestor of Z and Z is a parent of X,
then Y is an ancestor of X */
```

This program finds ancestors using the database at the begin of the program using the recursive rule of ancestor. If now we ask to Prolog :

```
?- ancestor(john,tom).
```

Prolog will first look in the program what he can fint about ancestor. He'll try the goal parent(john,tom). and it will fail. It will now try the second clause of ancestor. The new query is parent(john,Z). Prolog will find Z=paul and try ancestor(paul,tom) wich will conduct to check parent(paul,tom). This is successfull. As a result the goal ancestor(paul,tom) succeeds. Then ancestor(john,tom) can succeed.
The execution would be :

```
CALL ancestor(john,tom).
     CALL parent(john,tom).
     FAIL parent(john,tom).
```

```
    CALL parent(john,Z).
         TRY Z=paul
         CALL ancestor(paul,tom).
              CALL parent(paul,tom).
              SUCCEEDS parent(paul,tom).
         SUCCEEDS ancestor(paul,tom).
    SUCCEEDS with Z=paul
SUCCEEDS ancestor(johnmtom).
```

6.2.2 Factorial

The best way in Prolog to calculate a factorial is to do it recursivelly. Here is an example of how it can be done :

```
factoriel(0,1).
factoriel(X,Y) :-
                 X1 is X - 1,
                 factoriel(X1,Z),
                 Y is Z*X,!.
```

Note that in this case the cut is not necessary, it means that there is only one solution. The message "Out of local stack" may appear if you press ';' after the first solution. Now if you enter :

```
?- factoriel(5,X).
X = 120
Yes
```

In this example Prolog will try to calculate 5! then 4! then 3! until it has to calculate 0!, then the result will be 0!*1*2*3*4*5. Prolog will respond 120.

7 Lists

7.1 What is a list in Prolog

Lists are powerful data structures for holding and manipulating groups of things. In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types,including structures and other lists.Syntactically, a list is denoted by square brackets with the terms separated by commas. For example, a list of alcohol is represented as

```
[Tequila,Whisky,Vodka,Martini,Muscat,Malibu,Soho,Epita]
```

This gives us an alternative way of representing the locations of things. Rather than having separate location predicates for each thing, we can have one location predicate per container, with a list of things in the container.

```
list_where([Tequila,Whisky,Vodka], bathroom).
list_where([Martini,Muscat], kitchen).
list_where([Malibu,Soho], under_the_bed).
list_where([Epita], everywhere).
```

The empty list is represented by a set of empty brackets []. This is equivalent to the nil in other programming language.For our example in this section, it can describe the lack of things in a place :

```
list_where([], cave).
```

The Unification works on lists just as it works on other data structure of SWI Prolog. With that, we now can ask questions about lists to prolog:

```
?- [_,_,X] = [lesson, work, sleeping].
X = sleeping

?- list_where(X, under_the_bed).
X = [Malibu,Soho]
```

Notice that there is a impractical method of getteing a list of elements in the first example, because of Prolog won't unify unless both list have the same number of elements. At last, the special notation for list structures.

```
[X | Y]
```

This structure is unified with a list, X is bound to the first element of the list, called the head. Y is bound to the list of remaining elements, called the tail.Note that the tail is considered as a list for Prolog and the empty list does not unify with the standart list syntax because it has no head. Here is an example :

```
?- [X|Y] = [a, b, c, d, e].
X = a
Y = [b, c, d, e]

?- [X|Y] = [].
no
```

The empty list does not unify with the standard list syntax because it has no head.

```
?- [X|Y] = [].
no
```

This failure is important, because it is often used to test for the boundary condition in a recursive routine. That is, as long as there are elements in the list, a unification with the [X—Y] pattern will succeed. When there are no elements in the list, that unification fails, indicating that the boundary condition applies. We can specify more than just the first element before the bar (—). In fact, the only rule is that what follows it should be a list.

```
?- [First, Second | Q] = [water,gin,tequila,whisky].
First = water
Second = gin
Q = [tequila,whisky]
```

We have said a list is a special kind of structure. In a sense it is, but in another sense it is just like any other Prolog term. The last example gives us some insight into the true nature of the list. It is really an ordinary two-argument predicate. The first argument is the head and the second is the tail. This predicate is represented by a period(.). To see this notation, we use the built-in predicate display, which writes lits in using this syntax.

```
?- X = [T|Q], display(X).
.(_01,_02)
```

From this examples it should be clear why there is a different syntax for lists. The easier syntax makes for easier reading, but sometimes obscures the behavior of the predicate. It helps to keep this "real" structure of lists in mind when working with predicates that manipulate lists.

7.2 How to manipulate list

For lists to be useful,there must be easy way to access,add, and delete list elements.Moreover, we should not have to concern ourselves about the number of list items, or their order.

Two Prolog features enable us to accomplish this easy access. One is a special notation that allows reference to the first element of a list and the list of remaining elements, and the other is recursion.

These two features are very useful for coding list utility predicates, such as member, which finds members of a list, and append, which joins two lists together. List predicates all follow a similar strategy–try something with the first element of a list, then recursively repeat the process on the rest of the list.

The first one we will look at is member. As with most recursive predicates, we will start with the boundary condition, or the simple case. An element is a member of a list if it is the head of the list.

```
member(T,[T|Q]).
```

This clause also illustrates how a fact with variable arguments acts as a rule. The second clause of member is the recursive rule. It says an element is a member of a list if it is a member of the tail of the list.

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

As with many Prolog predicates, member can be used in multiple ways. If the first argument is a variable, member will, on backtracking, generate all of the terms in a given list.

```
?- membre(X, [muscat, soho, martini]).
X = muscat;
X = soho;
X = martini;
no
```

Another very useful list predicate builds lists from other lists or alternatively splits lists into separate pieces. This predicate is usually called append. In this predicate the second argument is appended to the first argument to yield the third argument. For example

```
?- append([a,b,c],[d,e,f],X).
X = [a,b,c,d,e,f]
```

It is a little more difficult to follow, since the basic strategy of working from the head of the list does not fit nicely with the problem of adding something to the end of a list. append solves this problem by reducing the first list recursively. The boundary condition states that if a list X is appended to the empty list, the resulting list is also X.

```
append([],X,X).
```

The recursive condition states that if list X is appended to list [T—Q1], then the head of the new list is also H, and the tail of the new list is the result of appending X to the tail of the first list.

```
append([T|Q1],X,[T|Q2]) :- append(Q1,X,Q2).
```

Real Prolog magic is at work here, which the trace alone does not reveal. At each level, new variable bindings are built, that are unified with the variables of the previous level. Specifically, the third argument in the recursive call to append is the tail of the third argument in the head of the clause. These variable relationships are included at each step.

8 Others Elements of Prolog

8.1 Operators

For the arithmetic operators, prolog has already a list of predefined predicates. These are :

```
=,is,<,>,=<,>=,==,=:=,/,*,+,-,mod,div
```

In Prolog, the calculations don't write like we have learned.They are written as a binary tree.That is to say that :

`y*5+10*x are written in Prolog : +(*(y,5),*(10,x)).`

Because we don't have learned to write like this, Prolog accept our way of writing calculations.Nevertheless, we have to say to him the rules of priority on the operators.For instance, we have to say to Prolog that * is prior on +.We have also to indicate to Prolog the prior calculations with "()".Prolog allows the programmer to define his own operators with the relations next :

```
Op(P,xfy,name).
```

Where P is the priority of the operators(between 1 and 1200), xfy indicates if the operator is infix(xfx,xfy,yfx) or postfix(fx,fy).The name is, of course, the name of the operator.Note that the prior operator has the lowest priority. Prolog has already these predefined operators :

```
        Op(1200,xfx,':-').
Op(1200,fx,[:-,?-]).
Op(1100,xfy,';').
Op(1000,xfy,',').
Op(700,xfx,[=,is,<,>,=<,>=,==,=:=]).
Op(500,yfx,[+.-]).
Op(500,fx,[+,-,not]).
Op(400,yfx,[*,/,div]).
Op(300,xfx,mod).
```

8.2 Arithmetic

Prolog use the infix operator "is" to give the result of an arithmetic operation to a variable.

```
X is 3 + 4.
```

Prolog responds

```
X = 7
yes
```

When the goal operator "is" is used the variables on the right must have been unified to numbers before. Prolog is not oriented calculations, so, after a certain point, he approximates the calculations and he don't answer the exact number but his approximation Fo example :

```
?- X is 1000 + .0001
X = 1000
yes
```

9 Input and Output

9.1 Input Output commands

At this time we have seen how we can communicate with prolog using the keyboard and the screen. We will now see how we can communicate with prolog using files.

Prolog can read and write in a file. The file where Prolog read is called input and the file where Prolog write is called output. When you run Prolog the default output is your screen (the shell) and the input is your keyboard. If you want to use files for thatr you have to tell it to Prolog using the commands listed in the appendix B on page 25.

9.2 Read and Write

Sometimes a program will nedd to to read a term from a file or the keybord and write a term on the screen or in a file. In this case the goals write and read can be used.

read(X). Read the term from the active input and unifie X with it.

write(Y). Write the term Y on the active output.

9.3 Examples

9.3.1 Calculating the cube of an integer

If we have the following program :

```
cube(C,N) :- C is N * N * N
```

If you ask something like `cube(X,3).` then Prolog would respond `X=9`. Suppose that we want to ask for other values than 3, we could write this program like this :

```
cube :-
        read(X), calc(X).  /* read X then query calc(X). */

calc(stop) :- !.           /* if X = stop then it ends   */

calc(X) :- C is X * X * X, write(C),cube.
                          /* calculate C and write it then ask again cube. */
```

Now if we ask Prolog cube, Prolog will ask use a value and give us the result until we write `stop`.

9.3.2 Treating the terms of a file

If we wanted to treat each term of a file, we could use this query :

```
?- see(filename), treatfile.\index{see(filename)}
```

using the following program :

```
treatfile :- read(Term), treat(Term).

treat( end_of_file ) :- !.

treat(Term) :- treatterm(Term),treatfile.

treatterm :- [the treatment you want to do to each term.]
```

9.3.3 ASCII characters

It is of course possible to use ASCII code in Prolog. For example if you type :

`\index{ASCII}\index{put()}`

```
?- put(65), put(66), put(67).
```

Prolog will write `ABC`.

9.3.4 Using Another Program

It is possible to load another Prolog program using a program. Two predicates have been made for this : `consult(program1)` to add in the database all the predicates contained in program1 amd `reconsult(program2)` to reload the predicates contained in program2.

10 SWI-Prolog

10.1 What is SWI-Prolog

If you want to use Prolog you need a compiler. There are many compiler downloadables on internet. For this report we used SWI-Prolog. SWI-Prolog is free Prolog compiler licenced under the GPL, targeting primaly the research and education.

10.2 Author

SWI-Prolog is designed and implemented by Jan Wielemaker from the University of Amsterdam, department of Social Science Informatics (SWI).

10.3 Platforms

SWI-Prolog is written in ANSI C and should configure for most Unix machines having an ANSI C Compiler.
Ports have been made for :

1. MS Win32
2. MS Win3.1
3. MS DOS
4. OS/2

10.4 FTP Sites

The main ftp-site to download SWI-Prolog is :
    ftp://swi.psy.uva.nl/pub/SWI-Prolog/
Latest version of the sources :
    ftp://swi.psy.uva.nl/pub/SWI-Prolog/pl.tar.gz
Latest binary for use with with Win32 :
    ftp://swi.psy.uva.nl/pub/SWI-Prolog/w32pl.exe
You can obtain more informations about SWI-Prolog by visiting the site :
    http://www.swi.psy.uva.nl/projects/SWI-Prolog/

A Meta logical Constructs

var(X) True if X is a variable.

nonvar True if X is not a variable.

atom(X) True if X is an atom.

integer(X) True if X is an integer.

atomic(X) True if X is an atom or a number.

float(X) True if X is a floiting point number.

functor(Term,Name,Arity) True if the arity of Term is Arity and his name Name.

B Input and Output

B.1 input

To read in a file in Prolog the following commands are used :

see(Filename). Open for output the file Filename.

seeing(File). When File is a variable, File is unified with the name of the input file.

get0(C). Reads one character from the input and unifies C to it ASCII code.

get(C). Reads non blank chars.

seen. Close the current input file.

consult(File). (also [File]. in some Prologs) When File is a filemame, loads the Prolog code contained in file in the database.

reconsult(file). Reload the the Prolog code contained in file in the database.

B.2 Output

To write in a file the following commands are used :

tell(Fillename) Open the file for output.

telling(File). File is unified to the name of the output file.

put(C). When C an ACSII code Writes the character given by C.

told. Close the current output file.

C Some others usefull predicates

C.1 True

The goal True/0 always succeeds.

```
father(jim,fred).
```

is in fact equivalent to :

```
father=(jim,fred) :- true.
```

C.2 repeat

The goal repeat/0 is used to do a loop. The predicate in witch it is used is repated until every goal succeeds. Example :

```
test :- repeat,
        write('Please enter a number'),
        read(X),
        (X=:=42).
```

In this example the program will ask you to enter a number until you enter 42.

C.3 Call

This predicate is used to launch another predicat. For example `call(p)` will run p.
Example :

```
call(write((``Hello World!'',nl))).
```

Note that we can't write `call(write('Hello World'),nl).` because the predicate call/2 is not avaible in every Prolog.

C.4 Setof

setof/3 can be usefull if you want to find all the solutions of a predicate. For example if we have this database :

```
knows(jim,fred).

knows(alf,bert).
```

If we want to find all the solutions of `knows(X,Y).` we can enter :

```
setof([X,Y],knows(X,Y),Z).
Z = [[jim,fred],[alf,bert]]
```

C.5 bagof

Bagof/3 is simmillar to setof/3. The difference is that bageof/3 leaves all repeated solutions while setof/3 removes them.

D Comparison Operators

D.1 Arithmetic Comparison Operators

The arithmetic comparison operators are :

`X<Y`
: `True if X is less than Y.`

`X=<Y`
: `True if X is less than or equal to Y.`

`X>Y`
: `True if X is greater than Y.`

`X>=`
: `True if X is greater than or equal to Y.`

`X=:=Y`
: `True if X is equal to Y.`

`X=/=Y`
: `True if the values of X and Y are not equal`

Unlike unification theses operators cannot be used to give values to a variable. The can only be evaluated when every term on each side have been instantiated.

D.2 Term Comparison

There is an order on the Prolog terms. The operators of comparison are :

`X@<Y`
: `The term X is less than Y`

`Y@=<Y`
: `The term X is less than or equal to Y`

`X@>Y`
: `The term X is greater than Y`

`X@>=Y`
: `The term X is greater or equal to Y`

The term order from the lowest to the highest is : 1. Variables. 2. Floating point numbers. 3. Integers. 4. atoms in the alphabetical order.

References

[1] Ivan Bratko. *PROLOG Programming for Artificial Intelligence.* Paperback, August 1990.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 1987.

[3] William F. Clocksin. *Clause and Effect : Prolog Programming for the Working Programmer.* Paperback, november 1997.

[4] H. Coelho and J.C. Cotta. *Prolog by Example.* Springer Verlag, 1998.

[5] Daniel E. Cooke. *Logic: The Basis for Understanding Prolog.* Norwood, 1994.

[6] Clive Matthews. *An Introduction to Natural Language Processing Through Prolog.* Paperback, August 1998.

[7] Dennis Merritt. Adventure in prolog, 1990.

[8] P. Saint-Dizier. *An Introduction to Programming in Prolog.* Springer Verlag, 1990.

[9] Leon Sterling. *The Practice of Prolog.* MIT Press, 1990.

[10] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* Hardcover, April 1994.

[]

Index