Chapter 2: Syntax and Meaning of Prolog Program

2.1 Data objects

There are two types of Data objects.

1. Simple Objects
2. Structures

Simple Objects are of two types:

1. Constants
2. Variables

Constants are of two types:

1. atoms
2. numbers

The Prolog system reiognizes the type of an object in the program by its syntactic form. variables start with upper-case letters whereas atoms start with lower-case letters.

Atoms can be constructed in three ways:

1. Strings of letters, digits and the underscore character, '-', starting with a lower-case letter. ex. anna, x_25AB
2. Strings of special characters. ex. <-->, ==>,..., .:. , ::=
3. Strings of characters enclosed in single quotes. This is useful if we want,for example, to have an atom that starts with a capital letter. Ex. 'Ankit', 'Ankit Gupta','Ankit_Gupta'

Numbers used in Prolog include integer numbers and real numbers

Not all integer numbers can be represented in a computer, therefore the range of integers is limited to an interval between some smallest and some largest number permitted by a particular Prolog implementation. Normally the range allowed by an implementation is at least between - 16383 and 16383, and often it is considerably wider

Real numbers are not used very much in typical Prolog programming. The reason for this is that Prolog is primarily a language for symbolic, non-numeric computation, as opposed to number crunching oriented languages such as Fortran. In symbolic computation, integers are often used, for example, to count the number of items in a list; but there is little need for real numbers

In general, we want to keep the meaning of programs as neat as possible. The introduction of real numbers somewhat impairs this neatness because of numerical errors that arise due to rounding when doing arithmetic. For example, the evaluation of the expression 10000+0.0001 -10000 may result in 0 instead of the correct result 0.0001

2.1.2 Variables

Variables are strings of letters, digits and underscore characters. They start with an upper-case letter or an underscore character

When a variable appears in a clause once only, we do not have to invent a name for it. We can use the so-called'anonymous'variable, which is written as a single underscore character. For example, let us consider the following rule:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

```
hasachild( X) :- parent( X, Y).
```

This rule says: for all X, X has a child if X is a parent of some Y.

We are defining the property hasachild which, as it is meant here, does not depend on the nurni of the child. Thus, this is a proper place in which to use an anonymous variable. The clause above can thus be rewritten:

```
hasachild( X) :- parent( X, _).
```

Eachtime a single underscore character occurs in a clause it represents a new anonymous variable. For example, we can say that there is somebody who has a child if there are two objects such that one is a parent of the other:

```
somebody_has_child :- parent( _, _).
```

This is equivalent to:

```
somebody_has_child :- parent( X, Y).
```

But this is, of course, quite different from:

```
somebody_has_child :- parent( X, X).
```

If the anonymous variable appears in a question clause then its value is not output when Prolog answers the question. If we are interested in people who have children, but not in the names of the children, then we can simply ask:

```
```

```
1 parent( pam, bob).
2 parent( tom, bob).
3 parent( tom, liz).
4 parent( bob, ann).
5 parent( bob, pat).
6 parent( pat, jim).
```

```
?- parent( X,_)
```

The lexical scope of variable names is one clause. This means that, for example, if the name X15 occurs in two clauses, then it signifies two different variables. But each occurrence of X15 within the same clause means the same variable, The situation is different for constants: the same atom always means the same object in any clause - that is, throughout the whole program.

2.1.3 Structures

Structured objects (or simpl y structures) are objects that have several components. The components themselves can, in turn, be structures. For example,the date can be viewed as a structure with three components: day, month, year. Although composed of several components, structures are treated in the program as single objects. In order to combine the components into a single object we have to choose afunctor. A suitable functor for our example is date. Then the date 1st May 1,983 can be written as:

```
date( 1, may, 1983)
```

here, "date" is a functor and 1,may,1983 are the arguments

All the components in this example are constants (two integers and one atom). Components can also be variables or other structures

This method for data structuring is simple and powerful. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation. Syntactically, all data objects in Prolog are terms. For example,

may and `date( 1, may, 1983)` are terms

All structured objects can be pictured as trees (see Figure2,.2 for an example). The root of the tree is the functor, and the offsprings of the root are the components. If a component is also a structure then it is a subtree of the tree that corresponds to the whole structured object

Our next example will show how structures can be used to represent some simple geometric objects (see Figure2.3). A point in two-dimensional space is defined by its two coordinates; a line segment is defined by two points; and a triangle can be defined by three points. Let us choose the following functors:

point for points,

seg for line segments, and

triangle for triangles.

Then the objects in Figure 2.3 can be represented by the following Prolog terms:

```
point for points,
seg for line segments, and
triangle for triangles
```

```
P1 = point(1,1).
P2 = point(2,3).
S = seg( P1, P2) = seg( point(1,1), point(2,3)).
T  = triangle(point(4,2), point(6,4), point(7,1)).
```

In general, the functor at the root of the tree is called the principal functor of. the term.

We can, however, use the same name, point, for points in both two and three dimensions, and write for example: `point( X1, Y1)` and `point( X,Y, Z)`

If the same name appears in the program in two different roles, as is the case for point above, the Prolog system will recognize the difference by the number of arguments, and will interpret this name as two functors: one of them with two arguments and the other one with three arguments. This is so because each functor is defined by two things:

(1) the name, whose syntax is that of atoms;

(2) the arity - that is, the number of arguments.

Figure 2.5 shows the tree structure that corresponds to the arithmetic expression

(a+b)*(c-5)

According to the syntax of terrns introduced so far this can be written, using the symbols '*', '*' and '-' as functors, as follows:

*(+(a, b), -(c, 5))

(make a tree where root is * and its left child is + and right is -)

This is of course a legal Prolog term; but this is not the form that we would normally like to have. We would normally prefer the usual, infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols '*' ,'*' and '-' are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

2.2 Matching

The most important operation on terms is matching. Matching alone can produce some interesting computation. Given two terms, we say that they match if.:

(1) they are identical, or

(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

For example, the terms `date( D, M, 1983)` and `date( D1, may, y1)` match. one instantiation that makes both terms identical is:

. D is instantiated to D1

. M is instantiated to may

. Y1 is instantiated to 1983

on the other hand, the terms `date( D, M, l9s3)` and `date( Dl, Ml, 1444)` do not match, nor do the terms `date( X, y, Z)` and `point( X, y, Z)`.

Matching is a process that takes as input two terms and checks whether they match. If the terms do not match we say that this process/ails. If they do match then the process succeeds and it also instantiates the variables in both terms to such values that the terms become identical.

Let us consider again the matching of the two dates. The request for this operation can be communicated to the Prolog system by the following question, using the operator '=':

```
?- date( D, M, 1983) = date( D1, may, Y1)
```

```
Singleton variables: [P1]
No permission to modify static procedure `(=)/2'
Singleton variables: [P2]
No permission to modify static procedure `(=)/2'
Syntax error: Operator priority clash
Singleton variables: [T]
No permission to modify static procedure `(=)/2'
D = D1,
M = may,
Y1 = 1983
```

```
?- date( D, M, 1983) = date(D1, maY, Y1).
```

```
Syntax error: Operator expected
Syntax error: Operator expected
Syntax error: Operator expected
Syntax error: Operator expected
D = D1,
M = maY,
Y1 = 1983
```

The general rules to decide whether two terms, S and T, match are as follows:

```
1 vertical(seg( point(X,Y), point(X,Y1))).
```

Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]

```
2 horizontal( seg( point(X,Y), pornt(X1,Y) )).
```

Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]
Singleton variables: [X,Xl]

```
?- vertical( seg( point(1,1), point(1,2) ) ).
```

Singleton variables: [Y,Y1]
Singleton variables: [X,Xl]
true *1*

```
?- vertical( seg( point(1,1), point(2,Y) ) ).
```

Singleton variables: [Y,Y1]
Singleton variables: [X,Xl]
false

```
1 horizontal( seg( point(X,Y), pornt(X1,Y) )).
```

Singleton variables: [X,X1]
Singleton variables: [X,X1]

```
?- horizontal( seg( point(1,1), point(2,Y) ) )
```

Singleton variables: [X,X1]
false

```
1 vertical(seg( point(X,Y), point(X,Y1))).
```
Singleton variables: [Y,Y1]
Singleton variables: [Y,Y1]

```
?- vertical( seg( point(2,3), P) )
```
Singleton variables: [Y,Y1]

P = *point*(2,_1736)

Here _1736 is a variable that has not been instantiated, It is, of course, a legal variable name that the system has constructed during the execution

```
1 horizontal( seg( point(X,Y), pornt(X1,Y) )).
```
Singleton variables: [X,X1]
```
2 vertical(seg( point(X,Y), point(X,Y1))).
```
Singleton variables: [Y,Y1]

```
?- vertical( S), horizontal( S).
```
Singleton variables: [X,X1]
Singleton variables: [Y,Y1]

false

S : `seg( point(X,Y), point(X,Y) )`

This answer by Prolog says: Yes, any segmenthat is degenerated to a point has the property of being vertical and horizontal at the same time. The answer was, again, derived simply by matching. As before, some internally generated names may appear in the answer, instead of the variable names X and Y.

2.3 Declarative meaning of Prolog programs

We have already seen in Chapter 1 that Prolog programs can be understood in two ways: declaratively and procedurally

e. To precisely define the declarative meaning we need to introduce the concept of instance of. a clause. An instance of a clause C is the clause C with each of its variables substituted by some term. A varianr of a clause C is such an instance of the clause C where each variable is substituted by another variable. For example, consider the clause

```
1 hasachild( X) :- parent( X, Y).
```

Two variants of this clause are:

```
1 hasachild( A) :- parent( A, B).
2 hasachild( Xl) :- parent( Xl, X2)
```

Instances of this clause are:

```
1 hasachild( peter) :- parent( peter, Z).
2 hasachild( barry) :- parent( barry, small(caroline) ).
```

disjuction of goals:

P:- Q;R

is read: P is true if Q is true or R is true. The meaning of this clause is thus the same as the meaning of the following two clauses together:

P:-Q. P:-R.

2.4 Procedural meaning

The procedural meaning specifies howProlog answers questions. To answer a question means to try to satisfy a list of goals. They can be satisfied if the variables that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To 'execute goals' means: try to satisfy them.

```
1 big( bear). % Clause 1
2 big( elephant). % Clause 2
3 small( cat). % Clause 3
4 brown( bear). % Clause 4
5 black( cat). % Clause 5
6 gray( elephant). % Clause 6
7 dark( Z) :- black( Z). % Clause 7:Anything black is dark
8 dark( Z) :- brown( Z). % Clause 7: Anything brown is dark
```

```
?- dark( X), big( X) %  Who is dark and big?
```

X = bear

```
1 p:-p.
```

```
?- p
```

```
** Execution aborted **
```

Using the clause above, the goal p is replaced by the same goal p; this will be in turn replaced by p, etc. In such a case Prolog will enter an infinite loop not noticing that no progress is being made.

the order of goals and clauses does matter. Furthermore, there are programs that are declaratively correct. but do not work in practice

Prolog's syntax is that of the first-order predicate logic formulas written in the so-called clause form (a form in which quantifiers are not explicitly written), and further restricted to Horn clauses only (clauses that have at most one positive literal). Clocksin and Mellish (1981) give a Prolog program that transforms a first-order predicate calculus formula into the clause form' The procedural meaning of Prolog is based on the resolution principle fot mechanical theorem proving introduced by Robinson in his classical paper (1965). Prolog uses a special strategy for resolution theorem proving called SLD. An introduction to the first-order predicate calculus and resolution-based theorem proving can be found in Nilsson 1981. Mathematical questions regarding the properties of Prolog's procedural meaning with respecto logic are analyzed by Lloyd (1984)

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

predecessor( X, Z) :-parent( X, Z).
predecessor( X, Z) :-
    parent( X, Y),
    predecessor( Y, Z).
```

Let us analyze some variations of this program. All the variations will clearly have the same declarative meaning, but not the same procedural meaning.

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

predecessor( X, Z) :-parent( X, Z).
predecessor( X, Z) :-
    parent( X, Y),
    predecessor( Y, Z).
% Four versions of the predecessor program
% The original version
pred1( X, Z) :-parent( X, Z).
pred1( X, Z) :-
    parent( X, Y),
    pred1( Y, Z).
%variation a: swap clauses of the original version
pred2( X, Z) :-
    parent( X, Y),
```

```
21      pred2( Y, Z).
22  pred2( X, Z) :-parent( X, Z).
23  % Variation b: swap goals in second clause of the original version
24  pred3( X, Z) :-parent( X, Z).
25  pred3( X, Z) :-
26      pred3( X, Y),
27      parent( Y, Z).
28  % Variation c: swap goals and clauses of the original version
```

```
?- pred1( tom, pat).
```

true *1*
false

```
?- pred2( tom, pat).
```

true *1*

Next | 10 | 100 | 1,000 | Stop

```
?- pred3( tom, pat)
```

true *1*

Next | 10 | 100 | 1,000 | Stop

```
?- pred4( tom, pat).
```

```
Stack limit (0.2Gb) exceeded
  Stack sizes: local: 0.2Gb, global: 11.4Mb, trail: 1Kb
  Stack depth: 1,487,674, last-call: 0%, Choice points: 1,487,658
  Probable infinite recursion (cycle):
    [1,487,674] pred4(tom, _1730)
    [1,487,673] pred4(tom, _1756)
```