

Chapter 2

More about Prolog

In this chapter, we dig a bit deeper into the Prolog language. We start with the basic objects that can be used as the arguments of a relation.

Any object is a string of basic symbols, such as characters (both upper case and lower case); digits, and some other special symbols.

Atoms, i.e. constant objects, can be either string of letters, digits, and ‘_’, starting with a lower case letter, such as “anna”, “and”, “the”, “king”, and “colt_45”, etc.; or string of characters enclosed in single quote, such as “ ’Tom’ ”; or strings of special symbols, such as “<-->”.

How about numbers?

Numbers include integers, such as 1, or -7; and real numbers, such as 3.14, or -0.018.

Real numbers are not heavily used in Prolog, since it is not what Prolog is used for. But, integers are often used, in, e.g., counting how many objects are there in a list.

Bad, bad, really bad

Variables are strings of letters, digits, and ‘_’, starting with an upper case letter. For example, “X”, “Result”, and “_23”, etc..

When a variable occurs in a clause only, we often don’t need to give it a name. For example, in

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

its meaning has nothing to do with the name of the second variable in `parent`. Thus, we can simply say

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

More about variables

The variable ‘_’ in the last clause is properly called an *anonymous* variable. The following clause

```
some_has_child:-parent(_,_)
```

is equivalent to

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

but, not equivalent to

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

If an anonymous variable occurs in a query, its value will not be output. For example, if we only care about who has children, but not the names of those children, we can say

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

Finally, the *lexical scope* of a variable is within the clause when it occurs.

Structures

This type is really like the `struct` type in C as we know of in the System programming course. It refers to objects that have several components. For example, a date object is a structure with three pieces: day, month, and year.

To combine those pieces together, we have to use a *functor*. For example, the date of September 1, 2009 can be expressed as

```
date(1, september, 2009)
```

while,

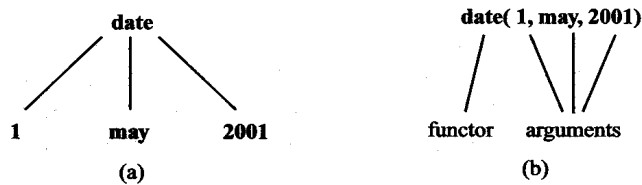
```
date(Day, may, 2009)
```

represents some day in May of 2009.

Technically speaking, any object used in a Prolog program is a *term*. We will further explain this term later.

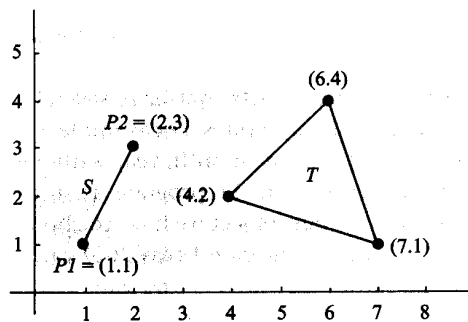
Graphical representation

All structured objects can be represented as a tree. The root of such a tree is the functor, and the components are the offsprings.



We can also use structured objects to represent geometric objects.

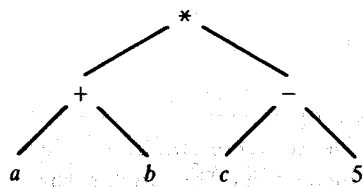
For example, `triangle(point(4,2),point(6,4),point(7,1))`.



It is pretty general

We can also use structured objects to represent arithmetic expressions, and even electric circuits. This is the Prolog's way for us to represent certain knowledge, which can then be presented to its inference engine to figure out things.

Below represents $(a + b) * (c - 5)$.



Homework: Exercises 2.1 and 2.2.

Matching

We just saw how to use terms to represent objects. If we can recall, a crucial operation in deriving a goal is to find applicable rules for that goal.

Technically speaking, the head of such an applicable rule has to be *matched* to that goal. We will explain how Prolog does it here.

Given two terms, we say they match with each other if 1) they are identical, or 2) the variables occurring in them can be instantiated in such a way that once the instantiation is applied, these two terms become identical.

(Most general) matching

For example, the two terms `date(D,M,2001)` and `date(D1,may,Y1)` match under the instantiation $D=D1$, $M=may$, and $Y1=2001$. Another instantiation is $D=14$, $D1=14$, $M=may$, and $Y1=2001$.

The latter instantiation is *less general* compared with the former, since it constraints the values more specific than necessary.

Prolog always tries to get the *most general instantiation* when it tries to match two terms.

This allows the greatest freedom for the involved variables, just in case, further matching is required down the road.

An example

Given the following goal:

```
date(D,M,2001)=date(D1,may,Y1) , date(D,M,2001)
=date(15,M,Y).
```

If we use the less general one for the first goal, we would not succeed to satisfy both. But, if we adopt the most general one for the first goal, i.e., $D=D1$, $M=may$, $Y1=2001$, then when satisfying the second goal, it will be further restricted to $D=15$, $D1=15$, $M=may$, $Y1=2001$, $Y=2001$.

The terms `date(D,M,2001)` and `date(D1,may,1786)` don't match; nor do the terms `date(D,M,2001)` and `point(1,1)`

The general matching process

Below is the general rule to decide if two terms, S and T , match.

1. If S and T are constant, then they match iff they are same.
2. If one of them is a variable, then they match, and the variable is instantiated to the other.
3. If both of them are structures, then they match only if a) They have the same functor, and b) all the corresponding components match. In this case, the resulting instantiation is determined by the matching of the components.

For example, `triangle(point(1,1),A,point(2,3))` matches with `triangle(X,point(4,Y),point(2,Z))`.

Homework: Read the rest of §2.2, then complete Exercises 2.3–2.5.

What does that mean?

A Prolog program can be understood in two ways: declarative, and procedural.

More specifically, given a clause

$P:-Q, R$

declaratively, it means that P is true if both Q and R are true. In other words, P (logically) follows from Q and R .

Procedurally, it tells us that to solve problem P , first solve problem Q , then solve problem R .

Notice that, in the procedural reading, an *order* of processing is also mentioned, besides the relationship between relations.

What does that mean, declaratively?

Given a program, P , and a goal, G , G is true in P , or logically follows from P , iff for some instantiation Θ and a clause C in P such that the head of $C \circ \Theta$ is identical to $G \circ \Theta$, and all the goals in the body of $C \circ \Theta$ are true in P .

More generally, a query, namely, a list of goals, is true in a program, if all those goals are true in the program, for the same instantiation of all the variables occurring in that question.

Homework: Read the rest of §2.3, then complete Exercises 2.6 and 2.7.

What does that mean, procedurally?

This basically means that given a program, and a query, how does Prolog answer the question, in other words, how does Prolog try to satisfy all the goals. This is done by a process, called *execute*.

Given a list of goals, $G = G_1, G_2, \dots, G_m$, the execute process does the following:

1. If the goal list is empty, terminate with *success*.
2. Otherwise, scanning through the program, from top to bottom, until it finds the first(next) clause, C , such that the head of C matches with G_1 under certain instantiation. If no such C exists, terminate with *failure*.

3. If there is such a C . Let it be $H:-B_1, \dots, B_n$, and rename the variables in B_1, \dots, B_n , to obtain C' , a *variant* of C , such that C' and G have no variables in common.

Let C' be $H':-B'_1, \dots, B'_n$, match G_1 and H' , and let the associated instantiation be Θ .

In the goal list, replacing G_1 with $B'_1\Theta, \dots, B'_n\Theta$, and replacing $G_i, i \in [2, m]$, with $G_i\Theta$, where for any goal G' , $G'\Theta$ means to apply Θ to G' .

4. Now, recursively apply this process to the new goal list as just obtained. If this process fails, going back to step 2.

An example

Given the following program:

1. `big(bear).`
2. `big(elephant).`
3. `small(cat).`
4. `brown(bear).`
5. `black(cat).`
6. `grey(elephant).`
7. `dark(Z):-black(Z).`
8. `dark(Z):-brown(Z).`

and the query:

```
?-dark(X), big(X).
```

Let's trace through the execute process.

1. Initial goal list: `dark(X), big(X)`.
2. scan the program and finds C7, and replace the first goal with the instantiated body of C7, with the matching instantiation being $Z = X$, to derive a new goal list `black(X), big(X)`.
3. Scan the program and find C5, and derive the new goal list of `big(cat)`.
4. Scan the program and could not find anything that matches `big(cat)`.

5. Backtrack to Step 2 and finds C8, and replace the first goal with the instantiated body of C8, again with the matching instantiation being $Z = X$, to derive a new goal list `brown(X)`, `big(X)`.
6. Scan the program and find C4, and derive the new goal of `big(bear)`, with the instantiation being $X = cat$.
7. Scan the program and find C1, and derive a new empty goal.
8. Terminate with success, together with $X=bear$.

Homework: Study Figure 2-11, and then complete Exercise 2.9.

The trace is automatic...

SWI Prolog provides a *trace* feature.

```
12 ?- trace.
```

Yes

```
[trace] 12 ?- dark(X), big(X).  
  Call: (8) dark(_G464) ? creep  
  Call: (9) black(_G464) ? creep  
  Exit: (9) black(cat) ? creep  
  Exit: (8) dark(cat) ? creep  
  Call: (8) big(cat) ? creep  
  Fail: (8) big(cat) ? creep  
  Redo: (8) dark(_G464) ? creep  
  Call: (9) brown(_G464) ? creep  
  Exit: (9) brown(bear) ? creep  
  Exit: (8) dark(bear) ? creep  
  Call: (8) big(bear) ? creep  
  Exit: (8) big(bear) ? creep
```

X = bear

Yes

Monkey and banana

A hungry monkey stands by the door. In the middle of the room, a banana is hanging from the ceiling. Although she can walk there, she is not tall enough to get the banana. But, she sees a box at the window of the room, which she can push to anywhere.



Can she get it?

Assume the monkey can walk, grasp, push, and climb onto that box, can she get the banana?

We use a relation, `state`, to represent the position of the Monkey, that of the box and whether she has got the banana or not; and use another one, `move`, to represent the transition between states, caused by a certain action.

For example, we can have the following transition:

```
move(state(middle,onbox,middle,hasnot),  
      grasp,  
      state(middle,onbox,middle,has)).
```

The program

```
move(state(middle, onbox, middle, hasnot),
      grasp,
      state(middle, onbox, middle, has) ).
move(state(P, onfloor, P, H),
      climb,
      state(P, onbox, P, H) ).
move(state(P1, onfloor, P1, H),
      push( P1, P2),
      state(P2, onfloor, P2, H) ).
move(state(P1, onfloor, B, H),
      walk( P1, P2),
      state(P2, onfloor, B, H) ).
canget(state(_, _, _, has) ).
canget(State1) :-
    move(State1, Move, State2),
    canget(State2).
```

Let's try it out with the following query

```
?-canget(state(atdoor, onfloor, atwindow, hasnot)).
```

Big deal?

Given the above query, there are four facts the next step could match.

Since the order in which those four facts will be tried is to start with the first one and come downwards, the order of those two facts are really important.

Question: What happens if we change the order of the first four clauses? Particularly, what happens if the one for *walk* is put as the first one?

Answer: It will get stuck with an “out of stack” message, since the transition will not make it closer to the end.

The general approach is to put something more specific first, and more general stuff later.

Ordering of clauses and goals

As another example, let's see what happens if we exchange the order of the two clauses and that of the two goals in the recursive case of the *predecessor* program. There are four combinations for such changes.

```
pred1(X, Z):-parent(X, Z).
```

```
pred1(X, Z):-parent(X, Y), pred1(Y, Z).
```

```
pred2(X, Z):-parent(X, Y), pred2(Y, Z).
```

```
pred2(X, Z):-parent(X, Z).
```

```
pred3(X, Z):-parent(X, Z).
```

```
pred3(X, Z):-pred3(X, Y), parent(Y, Z).
```

```
pred4(X, Z):-pred4(X, Y), parent(Y, Z).
```

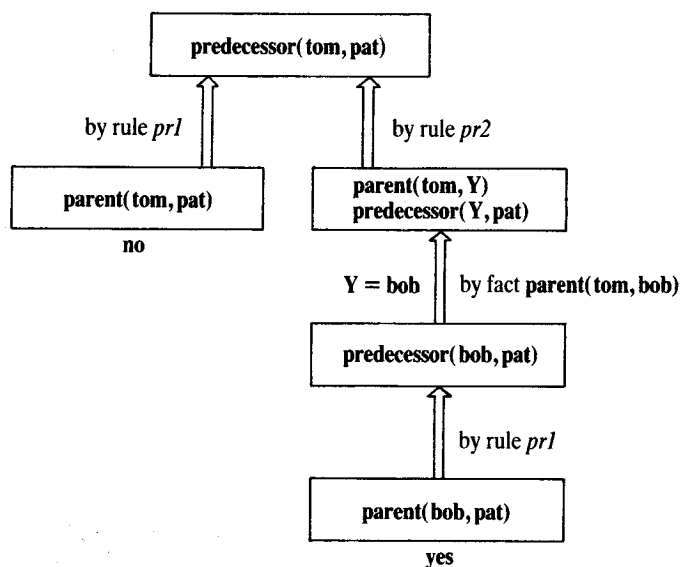
```
pred4(X, Z):-parent(X, Z).
```

What happens?

If we pose the following query

?-pred1(tom, pat).

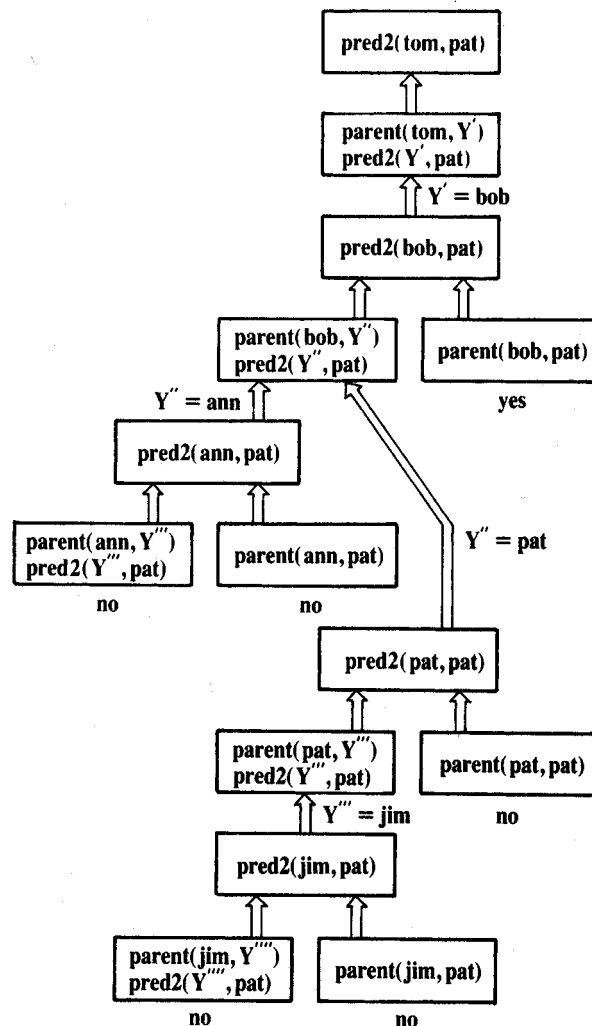
we will get an answer of “yes”. The following picture shows the process.



Why it takes so long?

The following picture shows the process for the following query

?-pred2(tom, pat).

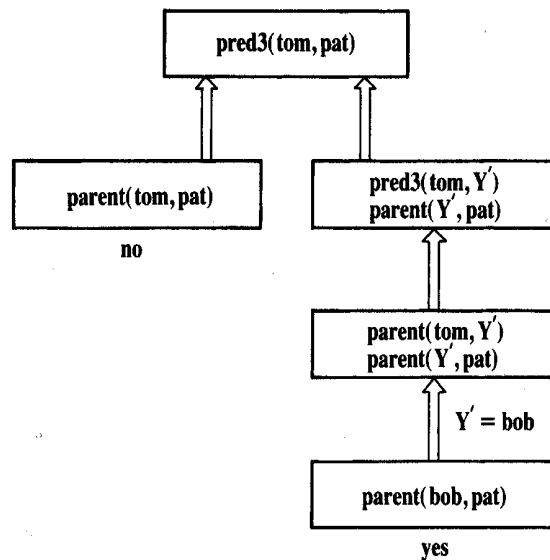


Why it takes so fast?

If we pose the following query

?-pred3(tom, pat).

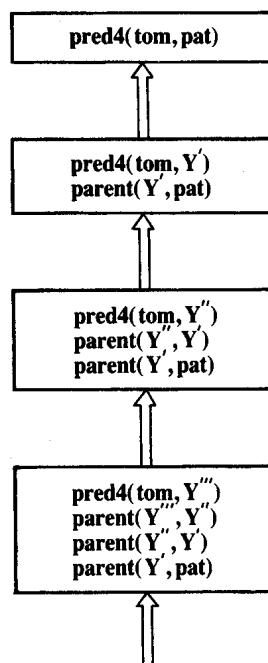
we will get an answer of “yes” very quickly.
The following picture shows the process.



Why nothing happens?

Nothing comes out for the following query

?-pred4(tom, pat).



A general approach is to work out a declarative program first, then test it to see if it works procedurally.

Homework: Look through §2.7 and think about Exercise 2.10.