

Chapter 7

More Goodies

Prolog does come with a bunch of built-in predicates, which allow us to do many things that are otherwise difficult, or even impossible, to do.

They allow us to test the type of atoms, take things apart, or combine them. Another set of such predicates allow us to add things into a Prolog program as a database, or delete things from it.

They, by and large, depend on various implementations, but the one presented here do come with many stuff.

Testing the types

Terms can be of various types. For example, if a term is a variable, then it can be instantiated at some point. When it is instantiated, its current value can be an atom, a structure, etc..

The types of terms are important to us. For example, before we carry out such a goal as `Z is X+Y`, we want to make sure that the types of both `X` and `Y` are indeed instantiated to integers, which can be done as follows:

```
...,number(X),number(Y),Z is X+Y,...
```

If either test fails, the calculation will not be done.

Some of those predicates

Besides `number(X)`, we have `var(X)`, `nonvar(X)`, `atom(X)`, `integer(X)`, `float(X)`, `atomic(X)`, `compound(X)`. The difference between `atom(X)` and `atomic(X)` is that the former is true iff `X` is an atom; while the latter is true if `X` is either a number or an atom. Below are some examples.

```
?-var(Z), Z=2.
```

```
Z=2
```

```
?-Z=2, var(Z).
```

```
no
```

```
?-integer(Z), Z=2.
```

```
no
```

```
?-Z=2, integer(Z), nonvar(Z).
```

```
Z=2
```

```
?-atom(3.14).
```

```
no
```

```
?-atomic(3.14).
```

```
yes
```

Another example

Let's write up a procedure, `count(A,L,N)`, which counts how many times an atom `A` occurs in `L`.

```
count(_, [], 0).  
count(A, [A|L], N) :- !, count(A, L, N1), N is N1+1.  
count(A, [_|L], N) :- count(A, L, N).
```

Now, a few executions.

```
?-count(a, [a,b,a,a], N).  
N=3.  
?-count(a, [a,b,X,Y], Na).  
Na=3.  
?-L=[a,b,X,Y], count(a,L,Na), count(b,L,Nb).
```

```
L = [a, b, a, a],  
X = a,  
Y = a,  
Na = 3,  
Nb = 1
```

What do we really want?

In the last example, both `X` and `Y` are instantiated to 'a', which leads to `Na=3`. Obviously, what we really want is to count the number of real occurrences of `atom`, but not the number of terms that match with the `atom`. Thus, the following modification.

```
count(_, [], 0).
count(A, [B|L], N) :- atom(B), A=B,!,
    count(A,L,N1), N is N1+1.
count(A, [_|L], N) :- count(A,L,N).
```

Notice, in `A=B`, we mean `A` matches with `B`, i.e., there is a substitution that makes `A` and `B` identical.

Let's test them out.

A tough one

Below is a (popular?) puzzle:

$$\begin{array}{r} DONALD \\ + \underline{GERALD} \\ ROBERT \end{array}$$

The problem is to *find* proper assignment of *different* digits to the letters, A, B, \dots, T , so that the equation holds.

Note that any digit can only be used for exactly one letter.

We will write up a procedure $\text{sum}(N1, N2, N3)$, where $N1$, $N2$ and $N3$ represent the three six-digit numbers of the above crypt-arithmetic puzzle.

The relation is true iff there is an assignment of digits to letters such that the equality holds.

The first step is always to find out how to represent the problem. It is natural to represent the numbers in lists. Thus, we have the following:

$$[D, 0, N, A, L, D] + [G, E, R, A, L, D] = [R, 0, B, E, R, T] .$$

Thus, the goal can be the following:

$$?-sum([D, 0, N, A, L, D], [G, E, R, A, L, D], [R, 0, B, E, R, T]) .$$

To define the `sum` relation, we have to implement the rules for carrying out addition in decimal system: It is done digit by digit, starting from right, going to the left, always taking into account the carry digit generated along the way.

For our purpose, we also keep a list of not-yet-used digits.

An accessory function

We define the following accessory relation,

`sum1(N1,N2,N,C1,C,Digits1,Digits),`

where `C1` is the carry from the right (before adding up `N1` and `N2` to `N`); `C` is the carry generated from adding `N1` and `N2`); `Digits1` refers to the set of digits available before adding `N1` and `N2`); and `Digits` refers to the leftover, i.e., those not used for the letters as shown in `N1`, `N2` and `N3`.

An example

For the following goal:

?-sum1([H,E],[6,E],[U,S],1,C,[1,3,4,7,8,9],D).

H=8

E=3

S=7

U=4

C=1

D=[1,9]

It calculates the following:

$$\begin{array}{r} HE \\ +_1 \underline{6E_1} \\ 47 \end{array}$$

Note D collects 1 and 9, the only ones that have not been used for the letters in N1, N2 and N.

The base cases

We can now define the original query, where we start with all the 10 digits.

```
sum(N1,N2,N):-  
  sum1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_)
```

The trivial case is the following:

```
sum1([],[],[],C,C,Digs,Digs).
```

where we have nothing to add which does not change the carry, nor the available digits.

The general case

Let the three numbers be $[D1|N1]$, $[D2|N2]$, and $[D|N]$. Then two things must happen:

1) $N1$, $N2$ and N must satisfy the `sum1` relation, producing $C2$ as the carry to the left, and leaving some unused digits for the first digits, `Digs2`.

2) The first digits must satisfy the decimal addition rule, $C2$, $D1$ and $D2$ are added giving D , and a carry C is generated to the left.

Thus, the code for this case is the following:

```
sum1([D1|N1],[D2|N2],[D|N],C1,C,Digs1,Digs):-  
    sum1(N1,N2,N,C1,C2,Digs1,Digs2),  
    digitsum(D1,D2,C2,D,C,Digs2,Digs).
```

The only thing left...

is to check if any of D1, D2 and D is instantiated with a digit; and if it is not, some unused digits have to be assigned to them, and also deleted from the unused digit set. This latter part is done in the `del_var` relation.

```
del_var(A,L,L):-  
    nonvar(A),!. %Already instantiated  
del_var(A,[A|L],L).  
del_var(A,[B|L],[B|L1]):-  
    del_var(A,L,L1).
```

The complete program is given in the next slide. The way to solve the aforementioned puzzle is

```
?-puzzle1(N1,N2,N3),sum(N1,N2,N3).
```

Homework: Exercise 7.1.

The whole thing

```
sum(N1,N2,N) :- sum1( N1, N2, N, 0, 0,
                    [0,1,2,3,4,5,6,7,8,9],_).
```

```
sum1([], [], [], C, C, Digits, Digits).
```

```
sum1([D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs) :-
    sum1(N1, N2, N, C1, C2, Digs1, Digs2),
    digitsum(D1, D2, C2, D, C, Digs2, Digs).
```

```
digitsum(D1, D2, C1, D, C, Digs1, Digs) :-
    del_var(D1, Digs1, Digs2),
    del_var(D2, Digs2, Digs3),
    del_var(D, Digs3, Digs),
    S is D1 + D2 + C1,
    D is S mod 10,
    C is S // 10.
```

```
puzzle1([D,O,N,A,L,D], [G,E,R,A,L,D],
        [R,O,B,E,R,T]).
```

(De)assembler

There are three built-ins that will decompose terms and construct new terms: `functor`, `arg` and `=...`

The goal `Term=..L` is true if `L` is a list that contains the principal functor of `Term`, followed by its arguments.

For example, `..` can be used to switch back and forth between a list of components and a structured term.

```
?-f(a,b)=..L.
```

```
L=[f,a,b]
```

```
?-Z=..[p,X,f(X,Y)].
```

```
Z=p(X,f(X,Y))
```

An application

Let's consider a program that manipulates geometric figures, such as squares, rectangles, triangles, etc.. These figures can be represented as terms whose functor indicates the type, and the arguments specify the size. We can certainly apply various operations on them. One of them is enlargement,

```
enlarge(Fig, Factor, Fig1).
```

One implementation can be the following:

```
enlarge(square(A), F, square(A1)):-
```

```
  A1 is F*A.
```

```
enlarge(circle(R),F,circle(R1)):-
```

```
  R1 is F*R.
```

```
enlarge(rectangle(A,B),F,rectangle(A1,B1)):-
```

```
  A1 is F*A,B1 is F*B.
```

A better way

When there are many different figures, we have to write many different procedures. But, essentially what we will do is to simply multiply all the arguments by a factor, then redraw the figure. This can be done easily using the `=..` procedure.

```
enlarge(Fig,F,Fig1):-  
  Fig=..[Type|Pars],  
  multiplylist(Pars,F,Pars1),  
  Fig1=..[Type|Pars1].  
  
multiplylist([],_,[]),  
multiplylist([X|L],F,[X1|L1]):-  
  X1 is F*X, multiplylist(L,F,L1).
```

Homework: Exercise 7.4.

The other two

Sometimes, we can also use functor and arguments to (de)assemble terms.

The relation $\text{functor}(\text{Term}, F, N)$ is true iff F is the principal functor of Term and N is the arity (the number of arguments) of F .

The relation $\text{arg}(N, \text{Term}, A)$ is true iff A is the n^{th} argument in Term , assuming that all the arguments are numbered from left to right, starting with 1.

?-functor(t(f(X,Y),X,t),Fun,Arity).

Fun=t

Arity=3

?-arg(2,f(X,t(a),t(b)),Y).

Y=t(a).

?-functor(D,date,3), arg(1,D,2),

arg(2,D,sept), arg(3,D,2001).

D=date(2,sept,2001).

What do you mean by equality?

There are at least three equality in Prolog. 1) $X=Y$, if they match. 2) X is E , if X matches the values of the expression E . 3) $E1::=E2$, if the values of expressions $E1$ and $E2$ are equal. Its opposite is $E1\neq E2$.

Moreover, '==' checks the *literal identity*. Its opposite is \neq . For example,

```
?-f(a,b)==f(a,b).
```

```
yes
```

```
?-f(a,b)==f(a,X).
```

```
no
```

More comparisons

We use ' $<$ ' to compare things numerically, e.g., " $x+2 < 5$ ". We also have " $@<$ ' to compare two terms for their lexicographical orderings, e.g.,

?-paul@<peter.

yes

?-f(2)@<f(3).

yes.

?-g(2)\$>=f(3).

no

You have got the idea, haven't you?

Database manipulation

Question: What is a database, from the RDB's point of view?

Answer: A database is a specification of a set of relations.

Thus, a Prolog program is nothing else but a database. It explicitly specifies a bunch of facts, and implicitly specifies a bunch of rules.

Prolog provides several built-ins to deal with this aspect of programs.

A goal `assert(C)` always succeeds, and as a side effect, it adds the clause `C` into the current program as a database. Its opposite is `retract(C)`, which deletes the clause from the program.

The following session demonstrates how these two predicates work.

```
?-crisis.  
no  
?-assert(crisis).  
yes  
?-crisis.  
yes  
?-retract(crisis).  
yes  
?-crisis.  
no
```

Clauses asserted this way will be part of the program. For example, if we have the following program running.

```
nice:-sunshine,not raining.  
funny:-sunshine,raining.  
disgusting:-raining,fog.  
raining.  
fog.
```

Now, we can update the program as follows:

```
?-nice.  
no  
?-disgusting.  
yes  
?-retract(fog).  
yes  
?-disgusting.  
no  
?-assert(sunshine).  
yes  
?-funny.  
yes  
?-retract(raining).  
yes  
?-nice.  
yes
```

Thus, Prolog can work in an interactive way with users.

A couple of points

1. There are two variants of `assert(C)`, `asserta(C)`, and `assertz(C)`. The former adds it at the beginning, and the latter adds it at the end.

Question: Why is it important?

2. When we read in a program, using `consult(File)`, it adds each and every clause at the end.

3. One usage of `asserta(C)` is to add a solution just found at the beginning of the program, so that later on, when searching for other answers, it can be found. It is called *memorization*, or *caching*.

Homework: Using the idea of caching to write a program for calculating `Fibonacci(50)`, and compare the speed with another version which does not use this technique. Explain the speed difference.

Get them all!

When there are more than one answers for a goal, Prolog will go through a backtracking process to get them one by one. When the new answer shows up, the old one is gone. Sometimes, we do need to collect all the answers.

The goal `bagof(X,P,L)` will produce the list `L` of all the objects `X` that satisfy a goal `P`. (Think about such problems as topological sort, permutation, eight-queens, etc..)

For example, let's have the following facts in our program,

```
age(peter,7).  
age(ann,5).  
age(pat,8).  
age(tom,5).
```

and play it out.

```
?-bagof(Child,age(Child,5), List).  
List=[ann,tom]  
?-bagof(Child,age(Child,Age), List).  
Age=7  
List=[peter];  
Age=5  
List=[ann,tom];  
Age=8  
List=[pat];  
no
```

The predicate `setof(X,P,L)` is similar, except that no duplicates will be allowed for the objects.

The predicate `findall(X,P,L)` differs from `bagof` in that all the objects `X` are collected regardless of different solutions of variable in `P` but not in `X`. So, it is really a mixed bag of everything that satisfies `P`.

```
?-findall(Child,age(Child,Age),List).  
List=[peter,ann,pat,tom]
```

In the above, although all the four objects satisfy the goal, their values for the variable `Age` do differ.

Homework: Exercises 7.8.