

Chapter 8

How to Program in Prolog?

In programming with Prolog, we really have to think “big” and declaratively. The technique of *divide and conquer*, thus recursion, is particularly appropriate with Prolog.

Again, when using recursion, we have to split the problem into two cases: 1) The trivial one, such as an empty list, the number 0, etc.. 2) In general, we try to construct a solution for the general case, based on solutions for simpler, or smaller, cases.

Let’s write `maplist(List,F,NewList)` to transform `List` with `F`, a certain function, into a new list `NewList`.

We saw `multiplylist(List,F,NewList)`, a special case of `maplist(List,F,NewList)`, in the previous chapter.

1. The base case: When `List` is empty, so is `NewList`.

2. In the general case, if `List=[X|Tail]`, we apply `F` to `X` to obtain `NewX`; then *recursively* apply `F` to `Tail` to get `NewTail`. The solution for the original problem, i.e., `[X|Tail]`, must be `[NewX|NewTail]`. Thus,

```
maplist([],_,[]).
maplist([X|Tail],F,[NewX|NewTail]):-
    G=..[F,X,NewX],
    call(G),
    maplist(Tail,F,NewTail).
```

Here, `call(G)` executes a goal `G`. For example,

```
square(X,Y):-Y is X*X.
?-maplist([2,3,5],square,Squares).
Squares=[4,9,25]
```

How come?

```
[trace] 6 ?- maplist1([2,5],square,Squares).
  Call: (7) maplist1([2, 5], square, _G556) ? creep
  Call: (8) _L175=..[square, 2, _G627] ? creep
  Exit: (8) square(2, _G627)=..[square, 2, _G627] ? creep
^ Call: (8) call(square(2, _G627)) ? creep
  Call: (9) square(2, _G627) ? creep
^ Call: (10) _G627 is 2*2 ? creep
^ Exit: (10) 4 is 2*2 ? creep
  Exit: (9) square(2, 4) ? creep
^ Exit: (8) call(square(2, 4)) ? creep
  Call: (8) maplist1([5], square, _G628) ? creep
  Call: (9) _L198=..[square, 5, _G645] ? creep
  Exit: (9) square(5, _G645)=..[square, 5, _G645] ? creep
^ Call: (9) call(square(5, _G645)) ? creep
  Call: (10) square(5, _G645) ? creep
^ Call: (11) _G645 is 5*5 ? creep
^ Exit: (11) 25 is 5*5 ? creep
  Exit: (10) square(5, 25) ? creep
^ Exit: (9) call(square(5, 25)) ? creep
  Call: (9) maplist1([], square, _G646) ? creep
  Exit: (9) maplist1([], square, []) ? creep
  Exit: (8) maplist1([5], square, [25]) ? creep
  Exit: (7) maplist1([2, 5], square, [4, 25]) ? creep
```

Squares = [4, 25]

Yes

Generalization...

is another good idea to design a Prolog program. Sometimes, instead of sticking to the original program, it is actually easier to solve a more general program. Once this general one is done, the original one becomes a special case.

For example, for the eight queens problem, if we call the original solution `eightqueen(Pos)`, which is true if `Pos` is a state with eight non-attacking queens. As we saw, a good idea is to generalize this problem to $N(\geq 0)$ queens.

The reason for the generalization is, of course, to enable us to....

... think recursively

1. As we saw, the base case $N = 0$ is trivial.
2. For the general case, we have to do two things: 1) Assume that we have found a solution for $N - 1$ queens; or rather for a list of $N - 1$ queens; and 2) add in yet another queen so that it will not attack any of the existing queens. This gives us a solution to the N queen version of the problem.

Once the general version is correctly defined, the solution for the original one is immediate: We just apply the program to a list of 8 queens.

This not only provides a solution to the original problem, but also finds a general solution for any N .

How many words is a picture worth?

We have seen in so many places that a picture really helps. It is also useful in programming with Prolog.

1. All the objects in Prolog are represented in a hierarchical form. They can be naturally represented with trees.
2. All the relations in Prolog can also be so represented, with the principal functor as the root of the tree, and all the arguments are subtrees.
3. Once a picture is done, we know more or less the declarative meaning of the problem, which can then be put into a program. (Think about the concatenation problem, the membership problem, etc..)

Dr. Shen, it doesn't work!

It is much easier to debug smaller pieces. This is also true with Prolog. Thus, the general principle is to work with smaller units first. Once they can be trusted, then test the bigger pieces.

Prolog is an interactive language, so any part of the program can be confronted with a direct question. Also, any implementation of Prolog usually comes with debugging tools. Thus, when you have an error or so, you don't need to knock my door.(:-))

The basis of any Prolog debugging is *tracing*: i.e., displaying information about the satisfaction of a goal.

Tracing

With tracing, we can get such information as:

- 1) The *entry info*, i.e., `Call`: The predicate name and the values of the arguments when the goal is called;
- 2) *exit info*.: In the case of a success, the values of arguments for the satisfied goal; otherwise, an indication of a failure;
- 3) *re-entry info*, i.e., `Redo`: We get into a backtrack for the same goal, in case of failure.

Once the trace is turned on, every bit of details will be spit out. If we no longer want it, we can assert `notrace`.

If we only want to trace one predicate `P`, we can use `spy(P)`. The trace will be stopped if we use `nospy(P)`.

Difference list

When we wrote a procedure for the concatenation of two lists, we did the following:

```
conc([],L,L).  
conc([X|L1],L2,[X|L]);-  
  conc(L1,L2,L).
```

This is fine, but it is slow, when the first list is long, e.g., for the goal `?-conc([a,b,c],[d,e],L)`, we have to go through the following:

```
?-conc([a,b,c],[d,e],L)  
?-conc([b,c],[d,e],L'), where L=[a|L']  
?-conc([c],[d,e],L''), where L'=[b|L'']  
?-conc([], [d,e],L'''), where L'''=[c|L''']
```

```
true, where L'''=[d,e]
```

Why is this boat slow?

The reason for the above inefficiency is that without knowing where the list ends, we have to go through the whole list to find it out.

To overcome this sort of problems, we can use a different way to represent lists, *difference list*, in which a list is represented in a pair. For example, the list [a,b,c] can be represented as the difference of two lists: $L1=[a,b,c,d,e]$ and $L2=[d,e]$.

More generally, [a,b,c] can be represented as $[a,b,c|T]-T$, where T can be anything and '-' is an infix operator.

Now, the concatenation is simply the follows:

$\text{concat}(A1-Z1, Z1-Z2, A1-Z2)$.

Now what?

Given the following goal,

$?-concat([a, b, c|T1]-T1, [d, e|T2]-T2, L).$

we have that

$A1/[a, b, c|T1], Z1/T1/[d, e|T2], Z2/T2.$

Thus,

$L/A1 - Z2/[a, b, c|[d, e|T2] - Z2.$

In other words,

$L/[a, b, c, d, e|T2] - T2.$

Homework: 8.2.

Tail recursiveness

Behind any recursive function, there is a stack, which takes space. Thus, if there are too many recursive calls within the program, the efficiency must be low.

Tail recursiveness is a special kind of recursion, in the sense that it uses little extra space. More specifically, only the last goal of its last clause is a recursive call; and the goals preceding the recursive call must be deterministic, which can be forced by using a *you-know-what*.

In a tail recursion, since the goal is the last one, it does not need to come back, thus no need to keep information for the return address.

An example

Assume that we want to add a list of numbers.
Below is the first implementation.

```
sumlist([],0).
sumlist([First|Rest],Sum):-
    sumlist(Rest,Sum1),
    Sum is First+Sum1.
```

This will cause many recursive calls (How many?)
Let's convert it into a tail recursion in an iterative call style.

```
sumlist(List,Sum):-
    sumlist1(List,0,Sum).

sumlist1([],Sum,Sum).
sumlist1([First|Rest],PSum,TSum):-
    NPSum is First+PSum,
    sumlist1(Rest,NPSum,TSum).
```

We love *it* so much...

that we want to use it again.

Question: What is *it*?

Answer: The usage of PSum, for a partial sum.

Let's write a relation `reverse(L,RL)` that reverses a list.

```
reverse(L,RL):-  
    reverse(L,[],RL).  
  
reverse([],RL,RL).  
reverse([X|Rest],PR,TR):-  
    reverse(Rest,[X|PR],TR).
```

Homework: 8.1, 8.5.

How did it go?

```
[trace] 17 ?- reverse([a, b, c], L).  
  Call: (7) reverse([a, b, c], _G510) ? creep  
  Call: (8) reverse([a, b, c], [], _G510) ? creep  
  Call: (9) reverse([b, c], [a], _G510) ? creep  
  Call: (10) reverse([c], [b, a], _G510) ? creep  
  Call: (11) reverse([], [c, b, a], _G510) ? creep  
  Exit: (11) reverse([], [c, b, a], [c, b, a]) ? creep  
  Exit: (10) reverse([c], [b, a], [c, b, a]) ? creep  
  Exit: (9) reverse([b, c], [a], [c, b, a]) ? creep  
  Exit: (8) reverse([a, b, c], [], [c, b, a]) ? creep  
  Exit: (7) reverse([a, b, c], [c, b, a]) ? creep
```

L = [c, b, a]

It is clear that there is no need to go back to where a recursive call is made, since it is always the last one.

Another way...

Below is another program for reverse, which is not done tail recursively.... Here `combine(FirstList, SecondList, Combo)` simply

```
reverse([], []).
reverse([H|T], TR):-reverse(T, T1),
                    conc(T1, [H], TR).
```

Since the recursive call is not the last one, we have to remember where to come back when this call is done.

Thus, we have to put in a stack to do the return address management.

How did it go?

```
[trace] 20 ?- reverse([a, b, c], L).  
  Call: (7) reverse([a, b, c], _G510) ? creep  
  Call: (8) reverse([b, c], _L172) ? creep  
  Call: (9) reverse([c], _L192) ? creep  
  Call: (10) reverse([], _L212) ? creep  
  Exit: (10) reverse([], []) ? creep  
  Call: (10) conc([], [c], _L192) ? creep  
  Exit: (10) conc([], [c], [c]) ? creep  
  Exit: (9) reverse([c], [c]) ? creep  
  Call: (9) conc([c], [b], _L172) ? creep  
  Call: (10) conc([], [b], _G590) ? creep  
  Exit: (10) conc([], [b], [b]) ? creep  
  Exit: (9) conc([c], [b], [c, b]) ? creep  
  Exit: (8) reverse([b, c], [c, b]) ? creep  
  Call: (8) conc([c, b], [a], _G510) ? creep  
  Call: (9) conc([b], [a], _G596) ? creep  
  Call: (10) conc([], [a], _G599) ? creep  
  Exit: (10) conc([], [a], [a]) ? creep  
  Exit: (9) conc([b], [a], [b, a]) ? creep  
  Exit: (8) conc([c, b], [a], [c, b, a]) ? creep  
  Exit: (7) reverse([a, b, c], [c, b, a]) ? creep
```

L = [c, b, a]

Notice we now have the sandwiched calls and exits.