Lecture 4

Recursion

Contents

- Recursion in general
- Simple examples
- The need to move closer to a solution
- List processing predicates

Recursion in general

- Recursion is the idea of defining something in terms of itself.
- It allows us to define algorithms very clearly and elegantly.
- Recursion is very common in Prolog, for two main purposes: scanning structures (e.g. lists), and general repetition.
- It can be difficult to grasp at first.

The parts of a recursive definition

A recursive definition, whether in Prolog or some other language (including English) needs two things:

Base case(s). There must be a statement of how the concept (predicate, procedure, etc) is defined for some very simple item (e.g. 0, or 1, or the empty list).

Recursive step. The main part of the definition shows how its argument can be decomposed into simpler value(s)/structure(s), and how the concept decomposes accordingly.

The recursive step “moves closer to” the base case(s). Applying the recursive step a suitable number of times should cause the definition to connect to the base case definition.

Example : factorial

The mathematical function $factorial$ is defined, for any positive integer N, as the product of all the numbers from N down to 1:

$$N \times (N-1) \times \ldots 2 \times 1$$

For the special case of 0, $factorial(0)$ is defined to be 1. This can be stated recursively:

(Base case) $factorial(0) = 1$
(Recursive step) For $N > 0$,
$factorial(N) = N \times factorial(N-1)$.

In Prolog:

```
factorial(0, 1).     % Base case

factorial(Number, Result) :-
                     % Recursive step
     Number > 0,
     NewNumber is Number - 1,
     factorial(NewNumber, NewResult),
     Result is Number * NewResult.
```

Factorial in action

(indentation added)

```
| ?- factorial(3, Result).
 + 1  1  Call: factorial(3,_199) ?
     2  2  Call: 3>0 ?
     2  2  Exit: 3>0 ?
     3  2  Call: _467 is 3-1 ?
     3  2  Exit: 2 is 3-1 ?
 +   4  2  Call: factorial(2,_459) ?
        5  3  Call: 2>0 ?
        5  3  Exit: 2>0 ?
        6  3  Call: _1474 is 2-1 ?
        6  3  Exit: 1 is 2-1 ?
      + 7  3  Call: factorial(1,_1466) ?
           8  4  Call: 1>0 ?
           8  4  Exit: 1>0 ?
           9  4  Call: _2481 is 1-1 ?
           9  4  Exit: 0 is 1-1 ?
 +         10  4  Call: factorial(0,_2473) ?
 +         10  4  Exit: factorial(0,1) ?
           11  4  Call: _1466 is 1*1 ?
           11  4  Exit: 1 is 1*1 ?
 +      7  3  Exit: factorial(1,1) ?
       12  3  Call: _459 is 2*1 ?
       12  3  Exit: 2 is 2*1 ?
 +   4  2  Exit: factorial(2,2) ?
    13  2  Call: _199 is 3*2 ?
    13  2  Exit: 6 is 3*2 ?
+ 1  1  Exit: factorial(3,6) ?

Result = 6 ?
```

Lists

A Prolog list is a naturally recursive structure.
Scanning a list is usually (and most easily) done recursively.
For example, there is a predicate `length/2` which computes (or checks) how many items are in a list:

```
?- length([a, b, c], P).

P = 3 ?

yes

?- length([a, b, c, d], 4).

yes
```

If we did not have this, we could write our own, recursively:

```
ourlength([ ], 0).    % Base case
ourlength([_|Rest], Size) :-
                        % Recursive step
       ourlength(Rest, RestSize),
       Size is RestSize + 1.
```

OURLENGTH in action

```
| ?- ourlength([a,b,c], Answer).
 + 1  1  Call: ourlength([a,b,c],_215) ?
     + 2  2  Call: ourlength([b,c],_484) ?
         + 3  3  Call: ourlength([c],_692) ?
           + 4  4  Call: ourlength([],_899) ?
           + 4  4  Exit: ourlength([],0) ?
             5  4  Call: _692 is 0+1 ?
             5  4  Exit: 1 is 0+1 ?
         + 3  3  Exit: ourlength([c],1) ?
           6  3  Call: _484 is 1+1 ?
           6  3  Exit: 2 is 1+1 ?
     + 2  2  Exit: ourlength([b,c],2) ?
       7  2  Call: _215 is 2+1 ?
       7  2  Exit: 3 is 2+1 ?
 + 1  1  Exit: ourlength([a,b,c],3) ?

Answer = 3 ?

yes
```

Some points to note

- Without the base case, the definitions would not be fully specified. In the Prolog versions, lack of a base case would lead to *non-termination* (infinite looping).
- In Prolog definitions, the base case should be placed **before** the recursive step, because a goal which matches the base case will sometimes also be capable of matching the recursive step. (In our `factorial`, the `N > 0` test prevents this, and in `ourlength`, the empty list can't unify with `[_|Rest]`.)
- There might be more than one base case and/or recursive step.

The need to move closer to a solution

- If the recursive case doesn't move closer to a solution, then our program might not terminate:

```
parent(jane, john).
parent(john, paul).

ancestor(Older, Younger) :-
        ancestor(Older, _Middle),
        parent(_Middle, Younger).

ancestor(Older, Younger) :-
        parent(Older, Younger).

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

- Gets stuck in an infinite loop:

```
1 Call:  ancestor(jane,john) ?
2 Call:  ancestor(jane,_603) ?
3 Call:  ancestor(jane,_964) ?
4 Call:  ancestor(jane,_1325) ?
5 Call:  ancestor(jane,_1686) ?
6 Call:  ancestor(jane,_2047) ?
7 Call:  ancestor(jane,_2408) ?
8 Call:  ancestor(jane,_2769) ?  ...
```

Moving closer to a solution (1)

- An improvement is to put the base case *first*:

```
ancestor(Older, Younger) :-
        parent(Older, Younger).

ancestor(Older, Younger) :-
        ancestor(Older, _Middle),
        parent(_Middle, Younger).

?- ancestor(jane, john).
yes
```

- But this still isn’t quite right:

```
?- ancestor(jane, Who).
Who = john ?  ;
Who = paul ?  ;
(INFINITE LOOP HERE!)
```

- Because if the recursive clause is called with `ancestor(jane, Who)`, its first sub-goal is: `ancestor(jane, _Middle)`, and the first sub-goal of *this* is: `ancestor(jane, _Middle1)` and we don’t move towards a solution.

Moving closer to a solution (2)

- This is better:

```
ancestor(Older, Younger) :-
       parent(Older, Younger).

ancestor(Older, Younger) :-
       parent(_Middle, Younger),
       ancestor(Older, _Middle).

?- ancestor(jane, john).
yes

?- ancestor(jane, Who).
Who = john ?  ;
Who = paul ?  ;
no
```

- Now, the recursive clause must move us towards a solution, even if it's called with uninstantiated arguments:

```
?- ancestor(Who, WhoElse).
Who = jane, WhoElse = john ?  ;
Who = john, WhoElse = paul ?  ;
Who = jane, WhoElse = paul ?  ;
no
```

Left recursion

A clause is **left recursive** if the predicate being defined (in the head) appears as the first term (leftmost item) in the body.

```
ancestor(Older, Younger) :-
         ancestor(Older, _Middle),
         parent(_Middle, Younger).
```

This can cause an indefinite loop (non-termination) if the goal at the start of the body does not move the computation nearer to a solution.

More list processing

- Often, list processing involves creating a new list out of an old one (or old ones).
- For example, to append two lists together:

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

append( [H|Tail1], List, [H|Tail2] ) :-
       append( Tail1, List, Tail2 ).

?- append([1, 2, 3], [4, 5, 6], X).
X = [1, 2, 3, 4, 5, 6]
yes

?- append(X, Y, [a, b, c]).
X = [], Y = [a,b,c] ?  ;
X = [a], Y = [b,c] ?  ;
X = [a,b], Y = [c] ?  ;
X = [a,b,c], Y = [] ?  ;
no
```

Notice that this is left-recursive, but is not a problem, as the values `Tail1, Tail2` are nearer to the base case than `[H|Tail1],[H|Tail2]`.

List processing predicates (2)

- To reverse a list using `append/3`:

```
reverse( [], [] ).

reverse( [Head|Tail], Answer ) :-
      reverse( Tail, RevTail ),
      append( RevTail, [Head], Answer ).
```

- There is a cleverer way to reverse a list, which uses a mechanism called an *accumulator*, rather than `append/3`. We'll look at that in a later lecture.

Summary

- Recursion is a general way of stating a definition.
- Recursion is widely used in Prolog.
- Recursion uses a base case and a recursive step.
- Care must be taken in organising these.
- Lists are recursively defined structures, so recursive procedures are well-suited to scanning them.