Tips on Writing Recursive Procedures in Prolog

The Basics - Base Case and Recursive Case(s)

When writing a recursive procedure in Prolog, there will always be at least two rules to code: at least one rule for the *base case*, or non-recursive case, and at least one rule for the *recursive case*.

The Base Case

Typically the base-case rule(s) will deal with the smallest possible example(s) of the problem that you are trying to solve - a list with no members, or just one member, for example. If you are working with a tree structure, the base case might deal with an `empty` tree, or a tree with just one node in it (like `tree(empty, a, empty)`).

The Recursive Case

To write the recursive case rule(s), you need to think of how the current case of the problem could be solved assuming you had already solved it for all smaller cases. For example, if you were adding a list of 10 numbers, and you had a way of summing the last 9 numbers in the list, then you would do so, then add on the first number on the list. (It might seem more natural to add up the *first* 9 numbers and then add the last number to the subtotal, but in Prolog it is easy to access the first item in the list, but not the last.)

Example 1: adding up a list of numbers

Here's an example of how to apply this to adding up a list of numbers. The comments beginning %% would not normally appear - they are there this time to help you match the scheme described in the previous paragraphs to the code.

```
% addup(List, Sum)
% Binds Sum to the sum of the numbers in the list.
% Assumes that each member of the list really is a number.
% The List must be instantiated at the time addup is called.
% Example of use:
% ?- addup([1,2,3,4], X).
% X = 10

%% base case
addup([], 0). % sum of the empty list of numbers is zero

%% recursive case: if the base-case rule does not match, this one must:
addup([FirstNumber | RestOfList], Total) :-
        addup(RestOfList, TotalOfRest),   % add up the numbers in RestOfList
        Total is FirstNumber + TotalOfRest.
```

So the recursive call `addup(RestOfList, TotalOfRest)`, two lines above, adds up all the items in the list except the first (`FirstNumber`), and binds the sum to `TotalOfRest`, and then the line after the recursive call, `Total is FirstNumber + TotalOfRest`, binds `Total` to the sum of the `FirstNumber` and the number `TotalOfRest` returned by the recursive call.

In one sense, that's all you need to know about the operation of `addup`. However, you may wonder how it all works when the first recursive call results in a second recursive call, and so on, and how the Prolog interpreter keeps track of all the different `TotalOfRest` instances, with recursive calls nested inside one another. You can get an idea of this by using Prolog's `trace` facility to add up the numbers in the list `[3, 5, 7]`. The material in bold is what the user types:

```
?- trace.
true.
[trace]  ?- addup([3, 5, 7], Total).
   Call: (7) addup([3, 5, 7], _G322)
   Call: (8) addup([5, 7], _L177)
   Call: (9) addup([7], _L197)
   Call: (10) addup([], _L217)
   Exit: (10) addup([], 0)
^  Call: (10) _L197 is 7+0
^  Exit: (10) 7 is 7+0
   Exit: (9) addup([7], 7)
^  Call: (9) _L177 is 5+7
^  Exit: (9) 12 is 5+7
   Exit: (8) addup([5, 7], 12)
^  Call: (8) _G322 is 3+12
^  Exit: (8) 15 is 3+12
   Exit: (7) addup([3, 5, 7], 15)
Total = 15.

[debug]  ?- notrace.
true.
```

Note that Prolog actually stops at the end of each line, prints a `?` prompt, and you press Return to get the next line of output. This has been edited out of the transcript above to reduce the clutter.

Note that Prolog replaces `Total` with `_G322` in the first line of tracing, labelled `Call: (7)`.

In the next line, labelled `Call: (8)`, it is executing the recursive rule for `addup`.

For each recursive call to `addup`, Prolog replaces the name `TotalOfRest` with a new variable name, guaranteed to be different from any other variable name currently in use.

So in the first recursive call: `addup([3, 5, 7], _L177)`, Prolog replaces `TotalOfRest` with `_L177`: that is, Prolog uses the name `_L177` for the instance of `TotalOfRest` in the first recursive call to `addup`. It's as though the recursive rule being executed has been replaced, just for this call, by:

```
addup([FirstNumber | RestOfList], Total) :-
        addup(RestOfList, _L177),   % add up the numbers in RestOfList
        Total is FirstNumber + _L177.
```

Similarly, in the second recursive call, `addup([5, 7], _L197)`, Prolog is using the name `_L197` for the instance of `TotalOfRest` in this nested recursive call to `addup`, and in the third recursive call, `addup([7], _L217)`, Prolog uses the name `_L217` for the instance of `TotalOfRest` in this doubly-nested recursive call to `addup`.

There is no fourth recursive call to `addup`, as the next goal, `addup([], TotalOfRest)`, triggers the base case rule.

Further down the trace, as the recursive calls complete and the recursion "unwinds", you can see Prolog *implicitly* binding `_L217` to `0` (`addup([], 0)` - using the base case rule) and *explicitly* binding `_L197` to the value of `7+0`, and binding `_L177` to the value of `5+7` = `12`, and finally binding `_G322` to the value of `3+12`.

To find out more about the trace facility, and/or to see another example of tracing recursive calls, check out the Prolog Dictionary entry on tracing.

Example 2: finding the last item of a list

This example shows how to find the last item in a list:

```
% lastitem(List, Last)
% Binds Last to the item at the end of List.
% List must be instantiated at the time of call, and must not be empty.
% Example of use:
% ?- lastitem([a,b,c,d], X).
% X = d

%% base case: list with just one item.
lastitem([OnlyOne], OnlyOne).

%% recursive case: ignore first item, seek last item of rest of list
lastitem([First | Rest], Last) :-
        lastitem(Rest, Last).
```

Example 3: Squaring each item in a list of numbers

This example shows how to build a result that is a list, i.e. we are transforming a list to produce a new list.

This time we'll start with a version with bugs (= errors) in it and also a stylistic error, and then we'll correct the errors. With each version of the code, you should look carefully at it before reading on, and try to spot the bug (or stylistic error). Spotting the bugs and stylistic errors is something you will have to do for yourself when you write your own programs, so start on it now!

Every programmer makes errors like these at some point. My aim in working through several different versions of this procedure is to give you an idea of how Prolog will respond to programmer errors. Prolog's messages are part of the information that the programmer can use to figure out what is wrong with their program.

NB: `square_1`, `square_2`, `square_3`, and `square_4` **are all wrong**. Only `square_5` is correct.

For the sake of brevity in this exposition, I've left out the comments in all but the final version. In practice you would develop the comments as you wrote the code. However, remember that just because you say something is true about the code, in your comments, doesn't mean it is. And don't forget to review and if necessary correct your comments when you find and fix a bug.

Different dialects of Prolog may produce different error messages: the messages below come from SWI-Prolog.

```
square_1([First | Rest], [First * First | SquareRest]) :-
        square_1(Rest, SquareRest).
```

Try it out:

```
?- square_1([1, 3, 5], Squares).
false.
```

Oops - we left out the base case. If you turn on tracing before you execute the query above, you will be able to see 1 * 1, 3 * 3, and 5 * 5 being produced, but because there is no base case, Prolog cannot complete the query.

```
square_2([], []).
square_2([First | Rest], [First * First | SquareRest]) :-
        square_2(Rest, SquareRest).
```

Try it out again:

```
?- square_2([1, 3, 5], Squares).
Squares = [1*1, 3*3, 5*5].
```

That's better, but why didn't it work out that 1 * 1 = 1, 3 * 3 = 9, etc.?
Answer: because we didn't ask it to. In Prolog, in most cases, evaluation of an arithmetic expression must be explicitly called for using the built-in predicate `is`.

```
square_3([], []).
square_3([First | Rest], [Square | SquareRest]) :-
        FirstSquared is First * First,
        square_3(Rest, SquareRest),
        Square = FirstSquared.
```

Try it:

```
?- square_3([1, 3, 5], Squares).
Squares = [1, 9, 25].
```

It works. But it contains a stylistic problem. The final goal `Square = FirstSquared` is redundant - it can be done better by simply writing `FirstSquared` instead of `Square` in the head of the rule:

```
square_4([], []).
square_4([First | Rest], [Firstsquared | SquareRest]) :-
        FirstSquared is First * First,
        square_4(Rest, SquareRest).
```

Always try it out after any change:

```
?- square_4([1, 3, 5], Squares)
Squares = [_G263, _G269, _G275].
```

Why doesn't it work any more?
Answer: We mistyped `FirstSquared` as `Firstsquared` in the head of the rule. Prolog cares desperately about upper and lower case letters, and the "s" of Squared is lower case in the head of the rule, but upper case in the first goal of the body. This can lead to a range of mystifying error messages - the type above, with one or more unresolved `_G…` variables, is one possibility.

When you re-loaded your code into Prolog, you would have received a warning message too (sometimes the warning is hard to notice because of Prolog's welcome message). The warning message is about "singleton variables" - that is, one or more variables that are only used once. In this case, the singleton variables were `Firstsquared` and `FirstSquared`. This could help you notice the problem with the capitalisation of the two instances of this variable.

```
%% base case
% nothing to square; result is empty list.
square_5([], []).
%% recursive case
% square the First item, recursively square the Rest.
square_5([First | Rest], [FirstSquared | SquareRest]) :-
        FirstSquared is First * First,
        square_5(Rest, SquareRest).
```

Always try it out - even if you just added comments, in case you forgot to put a % sign at the front of a comment line.

```
?- sqare_5([1, 3, 5], Squares).
Correct to: "square_5([1, 3, 5], Squares)"? yes
Squares = [1, 9, 25].
```

We mis-typed `square_5` as `sqare_5` in the *query*. Fortunately, SWI-Prolog has a feature called DWIM (Do What I Mean) which notices that there is no procedure called `sqare_5` and attempts spelling correction - in this case successfully. It suggests `square_5`, we agree by typing "y", and then Prolog prints the answer out.

Bill Wilson's contact info

UNSW's CRICOS Provider No. is 00098G
Last updated: 12/19/2010 09:46:09