1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Appendix B: Basics of recursion

Recursion is a valuable computer programming technique. It's a way to get a computer to do something over and over again. Recursion is tricky to understand at first, but once you get comfortable with it, it's easier and safer to program with recursion than with iterative techniques. We especially emphasize recursion in this book because Prolog has a natural affinity for it.

Recursion is a named piece of a program referring to itself. Why do this? Typically, to describe a task or job in the simplest possible way. A good way to explain a complicated job is to break it into components, and that's simplest when those components are the same sort of job. For this to work, we need to guarantee two things: first, that the new jobs are simpler than the old (for otherwise we wouldn't progress); and second, that by decomposition we'll eventually reach "easy" jobs that we can solve quickly without further decomposition. The easy jobs are called the *basis* of a recursion, and the decomposition into similar subjobs is called the *induction* of a recursion. (Recursion is closely related to a mathematical proof technique called *mathematical induction*, and that's where the terms come from.)

Here's an example (see Figure B-1). Suppose we have employee salaries in our computer, and we want to add 1000 to every one of them. Consider the easiest possible case (basis case) first. That's when there's only one salary, so we just add 1000 to it and store it. But suppose we have 732 salaries. One way to decompose the job (shown in the top diagram in Figure B-1) would be to add 1000 to the first salary, then establish a new job to add 1000 to the other 731 numbers. That new job isn't much simpler, but 731 *is* less than 732 so it is simpler. There are other ways to decompose the job too. For instance (as shown in the bottom diagram in Figure B-1), we can divide the job into equal halves: subjobs of adding 1000 to the first 366 items and adding 1000 to the last 366 items. Each subjob is the same sort of job as the original. Note that successful recursion just requires that the new jobs be smaller than the original job; it doesn't matter how many new jobs there are.

A recursion can have more than one basis step. For the previous example, a second basis step could handle two-item lists, so we wouldn't have to always decompose jobs to single-item lists eventually. In Prolog, basis conditions can be either facts (see Chapter 2) or rules (see Chapter 4), but induction conditions are always rules. Note that the first of the two decomposition methods in the last paragraph (binary decomposition in which one part is size 1) is the most common kind of recursive decomposition, because it's easy to implement despite the apparent inefficiency of its extreme uneveness of decomposition.

If you are having trouble understanding recursion, you should study some simple recursive programs and try to apply their form to new situations. The simplest form of recursion is *tail recursion*, in which there is only one induction (recursion) case, an extreme uneven decomposition, and its recursive call occurs last in the program. The first of the two ways to update 732 salaries is a natural application for tail recursion. Tail recursion is the commonest form of recursion, and has the advantage that it is easy to convert to iteration: the recursive call is like a branch or "goto" at the end of a loop. Recursive programs tend to be a little slower than their equivalent iterative counterparts, so that is one reason to convert them (though the recursive versions tend to be easier to understand). However, any recursion with more than one recursive (self-) reference is hard to convert to iteration, like the second (even-split) way of updating 732 salaries, or the means-ends analysis program in Chapter 11.

Recursion can define functions (in the mathematical sense) as well as subroutines. For instance, suppose the

job is to compute the sum of 732 employee salaries (see Figure B-2). The basis case is when there is only one number, and the sum is that number. The induction case is when there are N numbers, N > 1; let's use tail recursion for it. That means we recursively decompose this problem into a job to find the sum of all but the first of numbers, and add to first number this total. So each level of recursion returns a value to the next highest level.

Chapters 4 and 5 introduce recursive programs in Prolog. One word of advice: students often have trouble with recursion because they worry too much about what the program is does during execution. You can usually follow the first recursive call, and maybe the second, but after that it gets confusing because it's hard to see where to "go back to" when one recursive call concludes. Human brains just aren't as well equipped as computers for this sort of reasoning. So stick to the *declarative* or *logical-truth* description of recursion used in this Appendix. Then if your recursive program doesn't work right, use the following checklist:

1. Do the cases meant to be handled by basis conditions work right?

2. Do the basis conditions cover *all* the easy cases that might occur through recursive decomposition?

3. Does the recursive decomposition really decompose the problem into *simpler* subproblems? (How many doesn't matter.)

4. If the recursive program has arguments, are the arguments in the recursive calls correctly different from the arguments at the top of the program?

Usually one of these things will be at fault if a recursive program is not working.

Go to book index