Mastering recursive programming

Learning recursion yields maintainable, consistent, provably correct code

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: Recursion is a tool not often used by imperative language developers, because it is thought to be slow and to waste space, but as the author demonstrates, there are several techniques that can be used to minimize or eliminate these problems. He introduces the concept of recursion and tackle recursive programming patterns, examining how they can be used to write provably correct programs. Examples are in Scheme and C.

Tags for this article: programming, recursion

Tag this!
Update My dW interests (Log in | What's this?) Skip to help for Update My dW interests

Date: 16 Jun 2005
Level: Intermediate

Comments: 0 (View | Add comment - Sign in)

★★★★☆ Average rating (219 votes)
Rate this article

For new computer science students, the concept of recursive programming is often difficult. Recursive thinking is difficult because it almost seems like circular reasoning. It's also not an intuitive process; when we give instructions to other people, we rarely direct them recursively.

For those of you who are new to computer programming, here's a simple definition of recursion: Recursion occurs when a function calls itself directly or indirectly.

A classic example of recursion

The classic example of recursive programming involves computing factorials. The factorial of a number is computed as that number times all of the numbers below it up to and including 1. For example, `factorial(5)` is the same as `5*4*3*2*1`, and `factorial(3)` is `3*2*1`.

An interesting property of a factorial is that the factorial of a number is equal to the starting number multiplied by the factorial of the number immediately below it. For example, `factorial(5)` is the same as `5 * factorial(4)`. You could almost write the factorial function simply as this:

Listing 1. First try at factorial function

```
int factorial(int n)
{
   return n * factorial(n - 1);
}
```

The problem with this function, however, is that it would run forever because there is no place where it stops. The function would continually call `factorial`. There is nothing to stop it when it hits zero, so it would continue calling factorial on zero and the negative numbers. Therefore, our function needs a condition to tell it when to stop.

Since factorials of numbers less than 1 don't make any sense, we stop at the number 1 and return the factorial of 1 (which is 1). Therefore, the real factorial function will look like this:

Listing 2. Actual factorial function

```
int factorial(int n)
{
   if(n == 1)
   {
      return 1;
   }
   else
   {
      return n * factorial(n - 1);
   }
}
```

As you can see, as long as the initial value is above zero, this function will terminate. The stopping point is called the *base case*. A base case is the bottom point of a recursive program where the operation is so trivial as to be able to return an answer directly. All recursive programs must have at least one base case and must guarantee that they will hit one eventually; otherwise the program would run forever or until the program ran out of memory or stack space.

Back to top

Basic steps of recursive programs

Every recursive program follows the same basic sequence of steps:

1. Initialize the algorithm. Recursive programs often need a seed value to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is nonrecursive but that sets up the seed values for the recursive calculation.
2. Check to see whether the current value(s) being processed match the base case. If so, process and return the value.
3. Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
4. Run the algorithm on the sub-problem.

5. Combine the results in the formulation of the answer.
6. Return the results.

Using an inductive definition

Sometimes when writing recursive programs, finding the simpler sub-problem can be tricky. Dealing with *inductively-defined data sets*, however, makes finding the sub-problem considerably easier. An inductively-defined data set is a data structure defined in terms of itself -- this is called an *inductive definition*.

For example, linked lists are defined in terms of themselves. A linked list consists of a node structure that contains two members: the data it is holding and a pointer to another node structure (or NULL, to terminate the list). Because the node structure contains a pointer to a node structure within it, it is said to be defined inductively.

With inductive data, it is fairly easy to write recursive procedures. Notice how like our recursive programs, the definition of a linked list also contains a base case -- in this case, the NULL pointer. Since a NULL pointer terminates a list, we can also use the NULL pointer condition as a base case for many of our recursive functions on linked lists.

Linked list example

Let's look at a few examples of recursive functions on linked lists. Suppose we have a list of numbers, and we want to sum them. Let's go through each step of the recursive sequence and identify how it applies to to our summation function:

1. Initialize the algorithm. This algorithm's seed value is the first node to process and is passed as a parameter to the function.
2. Check for the base case. The program needs to check and see if the current node is the NULL list. If so, we return zero because the sum of all members of an empty list is zero.
3. Redefine the answer in terms of a simpler sub-problem. We can define the answer as the sum of the rest of the list plus the contents of the current node. To determine the sum of the rest of the list, we call this function again with the next node.
4. Combine the results. After the recursive call completes, we add the value of the current node to the results of the recursive call.

Here is the pseudo-code and the real code for the function:

Listing 3. Pseudo-code for the sum_list program

```
function sum_list(list l)
   is l null?
      yes - the sum of an empty list is 0 - return that
   data = head of list l
   rest_of_list = rest of list l
   the sum of the list is:
      data + sum_list(rest_of_list)
```

The pseudo-code for this program almost identically matches its Scheme implementation.

Listing 4. Scheme code for the sum_list program

```
 (define sum-list (lambda (l)
    (if (null? l)
       0
       (let (
             (data (car l))
             (rest-of-list (cdr l)))
          (+ data (sum-list rest-of-list))))))
```

For this easy example, the C version is just as simple.

Listing 5. C code for the sum_list program

```
int sum_list(struct list_node *l)
{
   if(l == NULL)
      return 0;
    return l.data + sum_list(l.next);
}
```

You may be thinking that you know how write this program to perform faster or better without recursion. We will get to the speed and space issues of recursion later on. In the meantime, let's continue our discussion of recursing of inductive data sets.

Suppose we have a list of strings and want to see whether a certain string is contained in that list. The way to break this down into a simpler problem is to look again at the individual nodes.

The sub-problem is this: "Is the search string the same as the one in *this node*?" If so, you have your solution; if not, you are one step closer. What's the base case? There are two:

- If the current node has the string, that's a base case (returning "true").
- If the list is empty, then that's a base case (returning "false").

This program won't always hit the first base case because it won't always have the string being searched for. However, we can be certain that if the program doesn't hit the first base case it will at least hit the second one when it gets to the end of the list.

Listing 6. Scheme code for determining if a given list contains a given string

```
(define is-in-list
    (lambda   (the-list the-string)
       ;;Check for base case of "list empty"
       (if (null? the-list)
          #f
          ;;Check for base case of "found item"
```

```
         (if (equal? the-string (car the-list))
            #t
            ;;Run the algorithm on a smaller problem
            (is-in-list (cdr the-list) the-string)))))
```

This recursive function works fine, but it has one main shortcoming -- every iteration of the recursion will be passing the *same value* for `the-string`. Passing the extra parameter can increase the overhead of the function call.

However, we can set up a closure at the beginning of the function to keep the string from having to be passed on each call:

Listing 7. Scheme program for finding a string using a closure

```
(define is-in-list2
   (lambda (the-list the-string)
      (letrec
        (
             (recurse (lambda (internal-list)
                  (if (null? internal-list)
                     #f
                     (if (equal? the-string (car internal-list))
                        #t
                        (recurse (cdr internal-list)))))))
         (recurse the-list))))
```

This version of the program is a little harder to follow. It defines a closure called `recurse` that can be called with only one parameter rather than two. (For more information on closures, see Resources.) We don't need to pass in `the-string` to `recurse` because it is already in the parent environment and does not change from call to call. Because `recurse` is defined *within* the `is-in-list2` function, it can see all of the currently defined variables, so they don't need to be re-passed. This shaves off one variable being passed at each iteration.

Using a closure instead of passing the parameter doesn't make a lot of difference in this trivial example, but it can save a lot of typing, a lot of errors, and a lot of overhead involved in passing variables in more complex functions.

The way of making recursive closures used in this example is a bit tedious. This same pattern of creating a recursive closure using `letrec` and then calling it with an initial seed value occurs over and over again in recursive programming.

In order to make programming recursive patterns easier, Scheme contains a shortcut called the *named let*. This construct looks a lot like a `let` except that the whole block is given a name so that it can be called as a recursive closure. The parameters of the function built with the named `let` are defined like the variables in a regular `let`; the initial seed values are set the same way initial variable values are set in a normal `let`. From there, each successive recursive call uses the parameters as new values.

Named `let`'s are fairly confusing to talk about, so take a look at the following code and compare it with the code in Listing 7.

Listing 8. Named let example

```
(define is-in-list2
    (lambda (the-list the-string)
       ;;Named Let
       ;;This let block defines a function called "recurse" that is the
      ;;body of this let.  The function's parameters are the same as
       ;;the variables listed in the let.
       (let recurse
          ;;internal-list is the first and only parameter.  The
          ;;first time through the block it will be primed with
          ;;"the-list" and subsequent calls to "recurse" will
          ;;give it whatever value is passed to "recurse"
          ( (internal-list the-list) )

          ;;Body of function/named let block
          (if (null? internal-list)
             #f
             (if (equal? the-string (car internal-list))
                #t
                ;;Call recursive function with the
                ;;rest of the list
                (recurse (cdr internal-list)))))))
```

The named `let` cuts down considerably on the amount of typing and mistakes made when writing recursive functions. If you are still having trouble with the concept of named `let`s, I suggest that you thoroughly compare every line in the above two programs (as well as look at some of the documents in the Resources section of this article).

Our next example of a recursive function on lists will be a little more complicated. It will check to see whether or not a list is in ascending order. If the list is in ascending order, the function will return `#t`; otherwise, it will return `#f`. This program will be a little different because in addition to having to examine the current value, we will also have to remember the last value processed.

The first item on the list will have to be processed differently than the other items because it won't have any items preceding it. For the remaining items, we will need to pass the previously examined data item in the function call. The function looks like this:

Listing 9. Scheme program to determine whether a list is in ascending order

```
(define is-ascending
    (lambda (the-list)
       ;;First, Initialize the algorithm.  To do this we
       ;;need to get the first value, if it exists, and
       ;;use it as a seed to the recursive function
       (if (null? the-list)
          #t
          (let is-ascending-recurse
             (
                 (previous-item (car the-list))
                (remaining-items (cdr the-list))
             )
             ;;Base case #1 - end of list
             (if (null? remaining-items)
                #t
                (if (< previous-item (car remaining-items))
                   ;;Recursive case, check the rest of the list
                   (is-ascending-recurse (car remaining-items) (cdr
remaining-items))
                   ;;Base case #2 - not in ascending order
                   #f))))))
```

This program begins by first checking a boundary condition -- whether or not the list is empty. An empty list is considered ascending. The program then seeds the recursive function with the first item on the list and the remaining list.

Next, the base case is checked. The only way to get to the end of the list is if everything so far has been in order, so if the list is empty, the list is in ascending order. Otherwise, we check the current item.

If the current item is in ascending order, we then have only a subset of the problem left to solve -- whether or not the rest of the list is in ascending order. So we recurse with the rest of the list and try it again.

Notice in this function how we maintained state through function calls by passing the program forward. Previously we had just passed the remainder of the list each time. In this function though, we needed to know a little bit more about the state of the computation. The result of the present computation depended on the partial results before it, so in each successive recursive call, we pass those results forward. This is a common pattern for more complex recursive procedures.

Back to top

Writing provably correct programs

Bugs are a part of the daily life of every programmer because even the smallest loops and the tiniest function calls can have bugs in them. And while most programmers can examine code and test code for bugs, they do not know how to prove that their programs will perform the way they think they will. With this in mind, we are going to examine some of the common sources of bugs and then demonstrate how to make programs which are correct and can be proven so.

Bug source: State changes

One of the primary sources of bugs occurs when variables change states. You might think that the programmer would be keenly aware of exactly how and when a variable changes state. This is sometimes true in simple loops, but usually not in complex ones. Usually within loops, there are several ways that a given variable can change state.

For example, if you have a complicated `if` statement, some branches may modify one variable while others modify other variables. On top of that, the order is usually important but it is difficult to be absolutely sure that the sequence coded is the correct order for all cases. Often, fixing one bug for one case will introduce other bugs in other cases because of these sequencing issues.

In order to prevent these kinds of errors, a developer needs to be able to:

- Tell by sight how each variable received its present value.
- Be certain that no variable is performing double-duty. (Many programmers often use the same variable to store two related but slightly different values.)
- Be certain that all variables hit the state they are supposed to be in when the loop restarts. (A common programming error is failure to set new values for loop variables in corner cases that are rarely used and tested.)

To accomplish these objectives, we need to make only one rule in our programming: *Assign a value to a variable only once and NEVER MODIFY IT!*

What? (You say increduluously!) This rule is blasphemy for many who have been raised on imperative, procedural, and object-oriented programming -- variable assignment and modification are at the core of these programming techniques! Still, state changes are consistently one of the chief causes for programming errors for imperative programmers.

So how does a person program without modifying variables? Let's look at several situations in which variables are often modified and see if we can get by without doing so:

- Reusing a variable.
- Conditional modification of a variable.
- Loop variables.

Let's examine the first case, reusing a variable. Often, a variable is reused for different, but similar, purposes. For example, sometimes if part of a loop needs an index to the current position in the first half of a loop and the index immediately before or after for the rest of the loop, many programmers use the same variable for both cases, just incrementing it in the middle. This can easily cause the programmer to confuse the two uses as the program is modified. To prevent this problem, the best solution is to create two separate variables and just derive the second from the first the same way you would do so if you were just writing to the same variable.

The second case, the conditional modification of a variable, is a subset of the variable reuse problem except that sometimes we will keep our existing value and sometimes we will want a new value. Again, the best thing is to create a new variable. In most languages, we can use the tertiary operator `? :` to set the value of the new variable. For example, if we wanted to give our new variable a new value, as long as it's not greater than `some_value`, we could write `int new_variable = old_variable > some_value ? old variable : new_value;`.

(We'll discuss loop variables later in the article.)

Once we have rid ourselves of all variable state changes, we can know that when we first define our variable, the definition of our variable will hold for as long as the function lasts. This makes sequencing orders of operations much easier, especially when modifying existing code. You don't have to worry about what sequence a variable might have been modified in or what assumptions were being made about its state at each juncture.

When a variable cannot change state, the full definition of how it is derived is illustrated when and where it is declared! You never have to go searching through code to find the incorrect or misordered state change again!

What about loop variables?

Now, the question is how to do loops without assignment? The answer lies in *recursive functions*. Take a look at the properties of loops and see how they compare with those of recursive functions in Table 1.

Table 1. Comparing loops with recursive functions

Properties	**Loops**	**Recursive functions**
Repetition	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by either finishing the block of code or issuing a `continue` command.	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by calling themselves.
Terminating conditions	In order to guarantee that it will terminate, a loop must have one or more conditions that cause it to terminate and it must be guaranteed at some point to hit one of these conditions.	In order to guarantee that it will terminate, a recursive function requires a base case that causes the function to stop recursing.
State	Current state is updated as the loop	Current state is passed as

	progresses.	parameters.

As you can see, recursive functions and loops have quite a bit in common. In fact, loops and recursive functions can be considered interchangeable. The difference is that with recursive functions, you rarely have to modify any variable -- you just pass the new values as parameters to the next function call. This allows you to keep all of the benefits of not having an updateable variable while still having repetitive, stateful behavior.

Converting a common loop to a recursive function

Let's take a look at a common loop for printing reports and see how it can convert into a recursive function.

- This loop will print out the page number and page headers at each page break.
- We will assume that the report lines are grouped by some numeric criteria and we will pretend there is some total we are keeping track of for these groups.
- At the end of each grouping, we will print out the totals for that group.

For demonstration purposes, we've left out all of the subordinate functions, assuming that they exist and that they perform as expected. Here is the code for our report printer:

Listing 10. Report-printing program using a normal loop

```
void print_report(struct report_line *report_lines, int num_lines)
{
   int num_lines_this_page = 0;
   int page_number = 1;
    int current_line; /* iterates through the lines */
    int current_group = 0; /* tells which grouping we are in */
    int previous_group = 0; /* tells which grouping was here on the last loop
*/
    int group_total = 0; /* saves totals for printout at the end of the
grouping */

     print_headings(page_number);

     for(current_line = 0; current_line < num_lines; current_line++)
    {
       num_lines_this_page++;
        if(num_lines_this_page == LINES_PER_PAGE)
       {
          page_number++;
          page_break();
          print_headings(page_number);
       }

       current_group = get_group(report_lines[current_line]);
       if(current_group != previous_group)
       {
         print_totals_for_group(group_total);
          group_total = 0;
       }

        print_line(report_lines[current_line]);

       group_total += get_line_amount(report_lines[current_line]);
    }
}
```

Several bugs have been intentionally left in the program. See if you can spot them.

Because we are continually modifying state variables, it is difficult to see whether or not at any given moment they are correct. Here is the same program done recursively:

Listing 11. Report-printing program using recursion

```
void print_report(struct report_line *report_lines, int num_lines)
{
    int num_lines_this_page = 0;
    int page_number = 1;
    int current_line; /* iterates through the lines */
    int current_group = 0; /* tells which grouping we are in */
    int previous_group = 0; /* tells which grouping was here on the last loop
*/
    int group_total = 0; /* saves totals for printout at the end of the
grouping */

      /* initialize */
   print_headings(page_number);

     /* Seed the values */
    print_report_i(report_lines, 0, 1, 1, 0, 0, num_lines);
}

void print_report_i(struct report_line *report_lines, /* our structure */
    int current_line, /* current index into structure */
    int num_lines_this_page, /* number of lines we've filled this page */
    int page_number,
     int previous_group, /* used to know when to print totals */
    int group_total, /* current aggregated total */
    int num_lines) /* the total number of lines in the structure */
{
    if(current_line == num_lines)
    {
       return;
    }
    else
    {
       if(num_lines_this_page == LINES_PER_PAGE)
       {
          page_break();
          print_headings(page_number + 1);
          print_report_i(
             report_lines,
              current_line,
              1,
              page_number + 1,
              previous_group,
              group_total,
              num_lines);
       }
       else
       {
          int current_group = get_group(report_lines[current_line]);
          if(current_group != previous_group && previous_group != 0)
          {
             print_totals_for_group(group_total);
             print_report_i(
                report_lines,
                 current_line,
                 num_lines_this_page + 1,
                 page_number,
                 current_group,
                 0,
                 num_lines);
          }
          else
          {
             print_line(report_lines[current_line]);
             print_report_i(
                report_lines,
                 current_line + 1,
                 num_lines_this_page + 1,
                 page_number,
                 current_group,
                 group_total + get_line_amount(report_lines[current_line]),
                 num_lines);
          }
       }
    }
}
```

Notice that there is never a time when the numbers we are using are not self-consistent. Almost anytime you have multiple states changing, you will have several lines during the state change at which the program will not have self-consistent numbers. If you then add a line to your program in the middle of such state changes you'll get major difficulties if your conception of the states of the variables do not match what is really happening. After several such modifications, it is likely that subtle bugs will be introduced because of sequencing and state issues. In this program, all state changes are brought about by re-running the recursive function with completely self-consistent data.

Proofs for recursive report-printing program

Because you never change the states of your variables, proving your program is much easier. Let's look at a few proofs for properties of the report-printing program from Listing 11.

As a reminder for those of you who have not done program proving since college (or perhaps never at all), when doing program proofs you are essentially looking for a property of a program (usually designated P) and proving that the property holds true. This is done using

- *axioms* which are assumed truths, and
- *theorems* which are statements about the program inferred from the axioms.

The goal is to link together axioms and theorems in such as way as to prove property P true. If a program has more than one feature, each is usually proved independently. Since this program has several features, we will show short proofs for a few of them.

Since we are doing an informal proof, I will not name the axioms we are using nor will I attempt to prove the intermediate theorems used to make the proof work. Hopefully they will be obvious enough that proofs of them will be unnecessary.

In the proofs, I will refer to the three recursion points of the program as R1, R2, and R3, respectively. All programs will carry the implicit assumption that `report_lines` is a valid pointer and that `num_lines` accurately reflects the number of lines represented by `report_lines`

In the examples I'll attempt to prove that

- The program will terminate for any given set of lines.
- Page breaks occur after LINES_PER_PAGE lines.
- Every report item line is printed exactly once.

Proof that the program will terminate

This proof will verify that for any given set of lines, the program will terminate. This proof will use a common technique for proofs in recursive programs called an *inductive proof.*

An inductive proof consists of two parts. First, you need to prove that property P holds true for a given set of parameters. Then you prove an induction that says if P holds true for a value of X, then it must hold true for a value of X + 1 (or X - 1 or any sort of stepwise treatment). This way you can prove property P for all numbers sequenced starting with the one you prove for.

In this program, we're going to prove that `print_report_i` terminates for `current_line == num_lines` and then show that if `print_report_i` terminates for a given `current_line`, it will also terminate for `current_line - 1`, assuming `current_line > 0`.

Proof 1. Verifying that for any given set of lines, the program will terminate

Assumptions We will assume that `num_lines >= current_line` and `LINES_PER_PAGE > 1`.
Base case proof

By inspection, we can see that the program immediately terminates when `current_line == num_lines`.
Inductive step proof In each iteration of the program, `current_line` either increments by 1 (R3) or stays the same (R1 and R2). R2 will only occur when the current value of `current_line` is different than the previous value of `current_line` because `current_group` and `previous_group` are directly derived from it. R1 can only occur by changes in `num_lines_this_page` which can only result from R2 and R3. Since R2 can only occur on the basis of R3 and R1 can only occur on the basis of R2 and R3, we can conclude that `current_line` must increase and can only increase monotonically. Therefore, if some value of `current_line` terminates, then all values before `current_line` will terminate.

We have now proven that given our assumptions, `print_report_i` will terminate.

Proof that page breaks occur after LINES_PER_PAGE lines

This program keeps track of where to do page breaks, therefore it is worthwhile to prove that the page-breaking mechanism works. As I mentioned before, proofs use axioms and theorems to make their case. I'll develop two theorems here to show the proof. If the conditions of the theorems are shown to be true, then we can use the theorem to establish the truth of the theorem's result for our program.

Proof 2. Page breaks occur after LINES_PER_PAGE lines

Assumptions The current page already has a page header printed on the first line.
Theorem 1 If `num_lines_this_page` is set to the correct starting value (condition 1), `num_lines_per_page` increases by 1 for every line printed (condition 2), and `num_lines_per_page` is reset after a page break (condition 3), then `num_lines_this_page` accurately reflects the number of lines printed on the page.

Theorem 2
If `num_lines_this_page` accurately reflects the number of lines printed (condition 1) and a page break is performed every time `num_lines_this_page == LINES_PER_PAGE` (condition 2), then we know that our program will do a page break after printing `LINES_PER_PAGE` lines.

Proof
We are assuming condition 1 of Theorem 1. This would be obvious from inspection anyway if we assume `print_report_i` was called from `print_report`.

Condition 2 can be determined by verifying that each procedure which prints a line corresponds to an increase of `num_lines_this_page`. Line printing is done

1. When printing group totals,
2. When printing individual report lines, and
3. When printing page headings.

By inspection, line-printing conditions 1 and 2 increase `num_lines_this_page` by 1, and line-printing condition 3 resets `num_lines_this_page` to the appropriate value after a page break/heading print combination (general condition 3). The requirements for Theorem 1 have been met, so we have proved that the program will do a page break after printing `LINES_PER_PAGE` lines.

Proof that every report item line is printed exactly once

We need to verify that the program always prints every line of the report and never skips a line. We could show using an inductive proof that if `print_report_i` prints exactly one line for `current_line == X`, it will also either print exactly one line or terminate on `current_line == X + 1`. In addition, since we have both a starting and a terminating condition, we would have to prove both of them correct, therefore we would have to prove the base case that `print_report_i` works when `current_line == 0` and that it will *only* terminate when `current_line == num_lines`.

However, in this case we can simplify things quite a bit and just show a direct proof by leveraging our first proof. Our first proof shows that starting with a given number will give termination at the proper point. We can show by inspection that the algorithm proceeds sequentially and the proof is already halfway there.

Proof 3. Every report item line is printed exactly once

Assumptions Because we are using Proof 1, this proof rests on Proof 1's assumptions. We will also assume that the first invocation of `print_report_i` was from `print_report`, which means that `current_line` starts at 0.
Theorem 1 If `current_line` is only incremented after a `print_line` (condition 1) call and `print_line` is only called before `current_line` is incremented (condition 2), then for every number that `current_line` passes through a single line will be printed.
Theorem 2 If theorem 1 is true (condition 1), and `current_line` passes through every number from 0 to `num_lines - 1` (condition 2), and terminates when `current_line == num_lines` (condition 3), then every report item line is printed exactly once.
Proof Conditions 1 and 2 of Theorem 1 are true by inspection. R3 is the only place where `current_line` increases and it occurs immediately after the only invocation of `print_line`. Therefore, theorem 1 is proven and so is condition 1 of theorem 2. Conditions 2 and 3 can be proven by induction and in fact is just a rehash of the first proof of termination. We can take our proof of termination to prove conclusively condition 3. Condition 2 is true on the basis of that proof and the assumption that `current_line` starts at 0. Therefore, we have proven that every line of the report is printed exactly once.

Proofs and recursive programming

These are just some of the proofs that we could do for the program. They can be done much more rigorously, but many of us chose programming instead of mathematics because we can't stand the tedium of mathematics nor its notation.

Using recursion tremendously simplifies the verification of programs. It's not that program proofs cannot be done with imperative programs, but that the number of state changes that occur make them unwieldy. With recursive programs that recurse instead of change state, the number of occasions of state change is small and the program variables maintain self-consistency by setting all of the recursion variables at once. This does not completely prevent logical errors, but it does eliminate numerous classes of them. This method of programming using only recursion for state changes and repetition is usually termed *functional programming*.

Back to top

Tail-recursive functions

So I've showed you how loops and recursive functions are related and how you can convert loops into recursive, non-state-changing functions to achieve a result that is more maintainable and provably correct than the original programming.

However, one concern people have with the use of recursive functions is the growth of stack space. Indeed, some classes of recursive functions will grow the stack space linearly with the number of times they are called -- there is one class of function though, *tail-recursive* functions, in which stack size remains constant no matter how deep the recursion is.

Tail recursion

When we converted our loop to a recursive function, the recursive call was the last thing that the function did. If you evaluate `print_report_i`, you will see that there is nothing further that happens in the function after the recursive call.

It is exhibiting a loop-like behavior. When loops hit the end of the loop or if it issues a `continue`, then that is the last thing it will do in that block of code. Likewise, when `print_report_i`recurses, there is nothing left that it does after the point of recursion.

A function call (recursive or not) that is the last thing a function does is called a *tail-call*. Recursion using tail-calls is called *tail-recursion*. Let's look at some example function calls to see exactly what is meant by a tail-call:

Listing 12. Tail-calls and non-tail-calls

```
int test1()
{
    int a = 3;
    test1(); /* recursive, but not a tail call.  We continue */
             /* processing in the function after it returns. */
    a = a + 4;
    return a;
}

int test2()
{
    int q = 4;
    q = q + 5;
    return q + test1(); /* test1() is not in tail position.
                         * There is still more work to be
                         * done after test1() returns (like
                         * adding q to the result
                         */
}

int test3()
{
    int b = 5;
     b = b + 2;
     return test1();  /* This is a tail-call.  The return value
                      * of test1() is used as the return value
                      * for this function.
                      */
}

int test4()
{
    test3(); /* not in tail position */
    test3(); /* not in tail position */
    return test3(); /* in tail position */
}
```

As you can see in order for the call to be a true tail-call, *no other operation* can be performed on the result of the tail-called function before it is passed back.

Notice that since there is nothing left to do in the function, the actual stack frame for the function is not needed either. The only issue is that many programming languages and compilers don't know how to get rid of unused stack frames. If we could find a way to remove these unneeded stack frames, our tail-recursive functions would run in a constant stack size.

Tail-call optimization

The idea of removing stack frames after tail-calls is called *tail-call optimization*.

So what is the optimization? We can answer that question by asking other questions:

- After the function in tail position is called which of our local variables will be in use? None.
- What processing will be done to the return value? None.
- Which parameters passed to the function will be used? None.

It seems that once control is passed to the tail-called function, nothing in the stack is useful anymore. The function's stack frame, while it still takes up space, is actually useless at this point, therefore the tail-call optimization is to *overwrite* the current stack frame with the next one when making a function call in tail position while keeping the original return address.

Essentially what we are doing is surgery on the stack. The activation record isn't needed anymore, so we are going to cut it out and redirect the tail-called function back to the function that called us. This means that we have to manually rewrite the stack to fake a return address so that the tail-called function will return directly to our parent.

For those who actually like to mess with the low-level stuff, here is an assembly language template for an optimized tail-call:

Listing 13. Assembly language template for tail-calls

```
;;Unoptimized tail-call
my_function:
    ...
    ...
     ;PUSH ARGUMENTS FOR the_function HERE
     call the_function

     ;results are already in %eax so we can just return
    movl %ebp, %esp
    popl %ebp
    ret

;;Optimized tail-call optimized_function:
    ...
    ...
     ;save the old return address
    movl 4(%ebp), %eax

     ;save old %ebp
    movl (%ebp), %ecx

      ;Clear stack activation record (assuming no unknowns like
     ;variable-size argument lists)
    addl $(SIZE_OF_PARAMETERS + 8), %ebp ;(8 is old %ebp + return address))

     ;restore the stack to where it was before the function call
    movl %ebp, %esp

     ;Push arguments onto the stack here
     ;push return address
    pushl %eax

     ;set ebp to old ebp
    movl %ecx, %ebp

     ;Execute the function
     jmp the_function
```

As you can see, tail-calls take a few more instructions, but they can save quite a bit of memory. There are a few restrictions for using them:

- The calling function must not depend on the parameter list still being on the stack when your function returns to it.
- The calling function must not care where the stack pointer is currently pointing. (Of course, it can assume that it is past its local variables.) This means that you cannot compile using `-fomit-frame-pointer` and that any registers that you save on the stack should be done in reference to `%ebp` instead of `%esp`.

- There can be no variable-length argument lists.

When a function calls itself in a tail-call, the method is even easier. We simply move the new values for the parameters on top of the old ones and do a jump to the point in the function right after local variables are saved on the stack. Because we are just jumping into the same function, the return address and old `%ebp` will be the same and the stack size won't change. Therefore, the only thing we need to do before the jump is replace the old parameters with the new ones.

So, for the price of at most a few instructions, your program can have the provability of a functional program and the speed and memory characteristics of an imperative one. The only problem is that right now very few compilers implement tail-call optimizations. Scheme implementations are required to implement the optimization and many other functional language implementations do so, too. Note, however, that because functional languages sometimes use the stack much differently than imperative languages (or do not use the stack at all), their methods of implementing tail-call optimizations can be quite different.

Recent versions of GCC also include some tail-recursion optimizations under limited circumstances. For example, the `print_report_i` function described earlier compiled with tail-call optimization using -O2 on GCC 3.4 and therefore runs with a stack-size that is constant, not growing linearly.

Back to top

Conclusion

Recursion is a great art, enabling programs for which it is easy to verify correctness without sacrificing performance, but it requires the programmer to look at programming in a new light. Imperative programming is often a more natural and intuitive starting place for new programmers which is why most programming introductions focus on imperative languages and methods. But as programs become more complex, recursive programming gives the programmer a better way of organizing code in a way that is both maintainable and logically consistent.

Resources

- A great intro to programming using functional programming, including many recursive techniques, is *How to Design Programs* (MIT Press, 2001).

- A more difficult introduction, but one which goes into much more depth, is *Structure and Interpretation of Computer Programs* (MIT Press, 1996).

- Understanding the issues of recursive programs with respect to the stack require some knowledge of how assembly language works; a good source is *Programming from the Ground Up* (Bartlett Press, 2004).

- This site offers more examples of recursion in action, including a tutorial, with each chapter progressing on certain topics, and a section consisting of a mix of several specific problems solved.

- For those of you who haven't done proofs in a while or at all, here is a good introduction to proof-writing.

- If you need another explanation of proof by induction, check out this tutorial on mathematical induction.

- You probably didn't know that proofs have patterns, too.

- Higher order functions (developerWorks, March 2005) takes a look at closures and other Scheme-related functions.

- XML Matters: Investigating SXML and SSAX (developerWorks, October 2003) examines using Scheme recursive features to manipulate XML.

- The Charming Python: Functional programming in Python series (developerWorks, March 2001) demonstrates that although programmers may think of Python as a procedural and object-oriented language, it actually contains everything you need for a completely functional approach to programming.

- Functional programming in the Java language (developerWorks, July 2004) shows you how to use functional programming constructs such as closures and higher order functions to write well-structured, modular code in the Java language.

- The road to better programming: Chapter 4 (developerWorks, January 2002) introduces the concept of functional programming as it relates to Perl.

- Beginning Haskell (developerWorks, September 2001) offers a gentle introduction to functional programming (with a focus on Haskell 98) for imperative-language developers.

- Use recursion effectively in XSL (developerWorks, October 2002) demonstrates that using XSL transformations effectively and efficiently requires understanding how to use XSL as a functional language which means understanding recursion.

- Find more resources for Linux developers in the developerWorks Linux zone.

- Get involved in the developerWorks community by participating in developerWorks blogs.

- Browse for books on these and other technical topics.

- Innovate your next Linux development project with IBM trial software, available for download directly from developerWorks.

About the author

Jonathan Bartlett is the author of the book *Programming from the Ground Up*, an introduction to programming using Linux assembly language. He is the lead developer at New Media Worx, developing Web, video, kiosk, and desktop applications for clients. Contact Jonathan at johnnyb@eskimo.com.