Roundabout

A Pattern Language for Recursive Programming

Eugene Wallingford
wallingf@cs.uni.edu

Introduction

Roundabout describes a method for writing recursive programs. Recursion is a powerful technique for writing code that follows the structure of the data it processes. By following the data's structure, recursion directly supports inductive reasoning about the program's behavior and correctness. Perhaps more importantly, it can also lead to code that is easier to write, modify, and read. This language guides the programmer through the development of recursive programs in the face of a number of common forces.

Most of the ideas contained in these patterns apply to the use of recursion in any context or language. However, I have been collecting and refining these ideas while teaching functional programming in Scheme to university undergraduates. This fact may account for several features of this version of Roundabout:

- a bias toward a functional style of programming,
- an assumption that students are familiar with inductive data definitions and BNF notation, and
- a focus on the use of recursion when one's data definition suggests it, rather than when one's algorithm suggests it.

One of my long-term goals is to present Roundabout in a way that makes its ideas available to the broadest audience possible.

These patterns may seem a bit elementary to many of you, but I have become quite interested in how we can use patterns to introduce novices to a particular domain. All of the complex pattern languages in the world are of little use if there is no entry point for new programmers. Indeed, the creation of elementary pattern languages supports the idea of piecemeal growth. It does so by allowing novice programmers to evolve toward expertise, over time developing code that is increasingly more habitable. While working on this language, I was struck by how these simple patterns work together to implement a concept that most of my students initially find so complex.

The Patterns

Roundabout consists of seven patterns:

1. Structural Recursion
2. Interface Procedure
3. Mutual Recursion
4. Accumulator Variable
5. Syntax Procedure
6. Local Procedure
7. Program Derivation

Pattern 1 is the entry point to the language. Patterns 2-5 expand on the first and document the primary techniques used to structure recursive programs. Patterns 6-7 describe how to improve the programs generated by the first five patterns in the face of other forces such as run-time efficiency or name-space clutter.

1. Structural Recursion

You are writing Simply Understood Code (Gabriel).

How do you write a procedure that operates on a data element whose structure is specified inductively?

Sometimes, what seems simplest at first is to write a loop to process the components of the data. Support for loops is built into most languages, and iteration often matches how you have been taught to think about programs. But formal reasoning about a loop's behavior is often unintuitive. You reason inductively about inductively-specified data, and you would like to be able to reason in a similar fashion about your programs. Furthermore, whenever the structure of the data changes, you would like to be able to quickly map those changes onto the structure of your code.

Therefore, write a recursive procedure. Give the procedure an Intention Revealing Name (Beck). Have one case for each arm of the data specification. Make the procedure a Composed Procedure (Beck) if any of its cases is complex.

For example, suppose that you have specified a list of symbols inductively as:

```
<list-of-symbols> ::= ()
                    | (<symbol> . <list-of-symbols>)
```

To compute the length of a list of symbols, you would write:

```
(define list-length
   (lambda (lon)
      (if (null? lon)
          0                          ;; base case
          (+ 1 (length (cdr lon)))   ;; recursive case
          )
   ))
```

If a recursive case requires an extra argument, turn your procedure into an Interface Procedure (2). If one of your cases operates on another inductively-defined data element, you may need to use Mutual Recursion (3). If several procedures collaborate to produce the answer, consider using an Accumulator Variable (4). If one of your cases deals with a data element that has several distinct parts, use a Syntax Procedure (5).

2. Interface Procedure

You are using Structural Recursion(1). Perhaps you have added an Accumulator Variable (4) to your code in order support the recursive call.

What should you do when a recursive call requires an extra argument?

Sometimes, in order to handle a recursive case, you need to pass an extra argument to the procedure. Consider the procedure `annotate`, which takes a list of symbols as an argument, for instance,

```
(a b c)
```

and returns a list with each symbol annotated with its position in the list:

```
((a 1) (b 2) (c 3))
```

Following Structural Recursion (1), your first version of `annotate` might look something like this:

```
(define annotate
   (lambda (los)
      (if (null? los)
          '()
          (cons (list (car los) position)
                (annotate (cdr los)))
      )
   ))
```

But where does the position of the next item in the list come from? You could pass the current position down to each recursive call:

```
(define annotate
   (lambda (los position)
      (if (null? los)
          '()
          (cons (list (car los) position)
                (annotate (cdr los) (+ position 1)))
      )
   ))
```

The extra argument makes writing the code easier. However, adding the argument modifies the procedure's interface. Many programming languages allow for default parameters, though, so the user of annotate would not have to provide an argument for position on the initial call. But the presence of the default parameter in the code exposes an implementation detail. You would like to hide this feature, so that potential users of the procedure won't have to concern themselves with the parameter in anyway.

Therefore, create two procedures. The first serves as the public interface to the computation and require only arguments necessary to "ask the question." It immediately passes these arguments plus any other necessary arguments to the second, a helper procedure that actually performs the computation. Give the helper an Intention Revealing Name (Beck). I like to append a phrase such as `-helper` to the public procedure's name.

Using this pattern, `annotate` becomes:

```
(define annotate
   (lambda (los)
      (annotate-helper los 1)    ;; 2nd parameter next position
   ))

(define annotate-helper
   (lambda (los position)
      (if (null? los)
          '()
          (cons (list (car los) position)
                (annotate-helper (cdr los) (+ position 1)))
          )
   ))
```

If use of the helper procedure results in undesirable consequences, try using a Local Procedure (6) or Program Derivation (7) to eliminate it.

3. Mutual Recursion

You are using Structural Recursion (1).

What should you do when the data that you are recursing on is defined in terms of another inductively-specified type?

Consider the procedure `replace`, which operates on s-lists:

```
<s-list> ::= ()
           |  (<symbol-expression> . <s-list>)
```

Symbol expressions are defined as

```
<symbol-expression> ::= <symbol>
                      |  <s-list>
```

`replace` takes three arguments: an object symbol, a target symbol, and an s-list. It returns an s-list identical in all respects to the original except that every occurrence of the target symbol has been replaced with the object symbol. For example,

```
: (replace 'a 'b '((a b) (((b g r) (f r)) c (d e)) b))
((a a) (((a g r) (f r)) c (d e)) a))
```

Using Structural Recursion (1), you might produce:

```
(define replace
   (lambda (new old slist)
      (cond ((null? slist) '())
            ((symbol? (car slist))
                (if (eq? (car slist) old)
                    (cons new (replace new old (cdr slist)))
                    (cons (car slist)
                          (replace new old (cdr slist)))
                    ))
            (else
               (cons (replace new old (car slist))
                     (replace new old (cdr slist)))))
  ))
```

Your procedure works, but it is not really true to the structure suggested by the BNF. `replace` uses the BNF to organize the computation, but the structure of the resulting program doesn't mimic the structure of the BNF. The data definition has two components, one for s-lists and one for symbol expressions. These components are mutually inductive, that is, defined in terms of one another. Should your code have this structure, too?

You would like to make your code as straightforward as possible, with as few side trips as possible. Creating a single procedure to compute the result achieves this goal, since it focuses the reader's attention on a single piece of code. But you also want to be faithful to the structure of your data, since it simplifies the individual pieces of code and makes later changes to data definitions easier to incorporate. Having multiple procedures that interrelate must be handled with care, however, since you want your reader to be comfortable following the computation.

Therefore, use Structural Recursion (1) on both data definitions. Each procedure will invoke the other at the corresponding point in the code. Give the helper procedure a name that indicates the data type on which it operates.

To apply this pattern to `replace`, write two procedures. The first operates on s-lists and is named `replace`. To write the arm of `replace` that handles symbol expressions, assume that the second — named `replace-symbol-expr` — already exists.

```
(define replace
   (lambda (new old slist)
      (if (null? slist)
          '()
          (cons (replace-symbol-expr new old (car slist))
                (replace             new old (cdr slist)))
          )
   ))
```

Now, write `replace-symbol-expr`, using replace in the arm that handles s-lists:

```
(define replace-symbol-expr
   (lambda (new old sym-expr)
      (if (symbol? sym-expr)
          (if (eq? sym-expr old)
              new
              sym-expr)
          (replace new old sym-expr)
          )
   ))
```

If use of the helper procedure results in undesirable consequences, try using a Local Procedure (6) or Program Derivation (7) to eliminate it.

4. Accumulator Variable

You are using Structural Recursion (1). Perhaps you are using Mutual Recursion (3) to process interdependent data types.

How do you handle a situation in which multiple procedures must collaborate to compute a result?

You want your code's structure to reflect the structure of the data it is processing. But following the data's structure slavishly may cause some undesired effects. Consider the procedure `flatten`, which operates on s-lists, as defined in Mutual Recursion (3). `flatten` takes an s-list as an argument and returns a list of all symbols in the argument, in the order that they occur in the s-list. For example,

```
: (flatten '((a b) (((b g r) (f r)) c (d e)) b))
(a b b g r f r c d e b)
```

Using Mutual Recursion (3), you might produce:

```
(define flatten
   (lambda (slist)
      (if (null? slist)
          '()
          (append (flatten-sym-expr (car slist))
                  (flatten (cdr slist)))
          )
   ))

(define flatten-sym-expr
   (lambda (sym-expr)
      (if (symbol? sym-expr)
          (list sym-expr)
          (flatten sym-expr)
          )
   ))
```

The problem is that `append` is an expensive operation that results in O(n^2) run-time performance for `flatten`.

So, the straightforwardness of your program structure comes at a significant performance cost. You would like to preserve the basic structure of the procedure, and yet retain control over how the answer is assembled.

Therefore, use an accumulator variable. Give this variable a Role Suggesting Temporary Variable Name (Beck). Make it a parameter on every procedure that plays a role in creating the answer. When a procedure is computing its answer, it "adds" to the accumulator variable in some fashion and returns the result. Make the top-level procedure an Interface Procedure (2) that calls a helper with the initial value for the accumulator.

To apply this pattern to `flatten`, start by creating an Interface Procedure (2) that passes an empty accumulator to its helper:

```
(define flatten
   (lambda (slist)
      (flatten-helper slist '())
   ))
```

Then define the helper procedure that operates on the s-list and the accumulator:

```
(define flatten-helper
   (lambda (slist symbols-so-far)
      (if (null? slist)
          symbols-so-far
          (flatten-sym-expr-helper
                  (car slist)
                  (flatten-helper (cdr slist) symbols-so-far))
          )
   ))
```

Notice how you use the answer from `flatten-helper`, which is a list of all the symbols in the `cdr` of `slist`: You pass it as the second argument to `flatten-sym-expr-helper`, which processes the symbol expression at the front of the s-list. This is where the accumulation takes place. First, `flatten-helper` solves the problem for the `cdr` of the list. Its answer is passed as the accumulator to `flatten-sym-expr-helper`, which then `cons`es symbols from the car into the accumulator. The `cons` operation is much more efficient than the `append` used in our first solution.

The definition of `flatten-sym-expr-helper` follows the mutual recursion pattern, with the addition of the accumulator:

```
(define flatten-sym-expr-helper
   (lambda (sym-expr symbols-so-far)
      (if (symbol? sym-expr)
          (cons sym-expr symbols-so-far)         ;; accumulate
          (flatten-helper sym-expr symbols-so-far)
          )
   ))
```

This pattern is similar to Collecting Parameter (Beck).

If use of the helper procedure results in undesirable consequences, try using a Local Procedure (6) or Program Derivation (7) to eliminate it.

5. Syntax Procedure

You are using Structural Recursion (1). The data that you are processing has one or more well-defined components that are used in the computation.

How do you eliminate structure-accessing code from the procedure while still accessing the structure?

Consider the procedure `path`, which operates on binary search trees of the form

```
<bst> ::= ()
       |   (<number> <bst> <bst>)
```

`path` takes a target number and a binary search tree as arguments. It returns a list of directions (either left or right) for finding the number in the tree. For example,

```
: (define my-tree '(14 (7 () (12 () ()))
                       (26 (20 (17 () ())
                               ())
                           (31 () ()))))
my-tree

: (path 17 my-tree)
(right left left)
```

Implementing `path` directly in Scheme requires a number of calls to `car` and `cdr` as the tree structure is traversed. These calls obscure the abstract structure of the tree, wherein internal nodes consist of a number, a left subtree, and a right subtree.

```
(define path
    (lambda (n bst)
      (cond ((null? bst)
                 (error "path: number not found!"))
            ((< n (car bst))
                 (cons 'left  (path n (car (cdr bst)))))
            ((> n (car bst))
                 (cons 'right (path n (car (cdr (cdr bst))))))
            (else
                 '())                      ;; n is here!
            )
     ))
```

Access code makes the procedure harder to read and hides the abstract structure of the data. Using helper procedures to access the structure require an extra level of procedure calls, resulting in less efficient run-time performance. But the helper procedures communicate the structure of the data more clearly and isolate details of the structure implementation away from the client code.

Therefore, create one syntax procedure for each component in the structure. Give each an Intention Revealing Name (Beck) to indicate that the procedure retrieves the component from the structure. In Scheme, a common form is

```
Structure->Component
```

Whenever you need a data component in your code, call the appropriate accessor.

To apply this pattern to `path`, start by defining several syntax procedures over the binary search tree data type:

```
(define empty-tree?
   (lambda (bst)
      (null? bst)))

(define node
   (lambda (bst)
      (car bst)))

(define left-subtree
   (lambda (bst)
      (car (cdr bst))))

(define right-subtree
   (lambda (bst)
      (car (cdr (cdr bst)))))
```

Then write `path` in terms of the syntax procedures:

```
(define path
   (lambda (n bst)
      (cond ((empty-tree? bst)
                 (error "path: number not found!"))
            ((< n (node bst))
                 (cons 'left  (path n (left-subtree bst))))
            ((> n (node bst))
                 (cons 'right (path n (right-subtree bst))))
            (else
                 '() )                        ;; n is here!
            )
   ))
```

Syntax procedures are not specific to recursion, but they are especially useful when writing recursive programs. When a computation is spread across a number of different procedures, as is often the case when following these patterns, the reader can become lost rather quickly. Judicious use of syntax procedures to denote the abstract structure of data offers the reader a common focus across cooperating procedures.

If you need to make several calls to the same accessor within one procedure, use a Caching Temporary Variable (Beck) to hold the result from a call and use that variable in place of procedure calls.

If use of the helper procedure results in undesirable consequences, try using a Local Procedure (6) to eliminate it.

6. Local Procedure

You are using Structural Recursion (1) and have created one or more helper procedures.

How do you eliminate a helper whose only useful purpose is to assist another procedure, yet still benefit from the abstraction that it provides?

Oftentimes, the helper procedures produced by Interface Procedure (2) and Accumulator Variable (4), in particular, are ones that no other piece of code can use. They are specialists created to support the recursive technique being applied. No other procedure is likely to use them in any context. Consider, for example, `annotate-helper` in Interface Procedure (2) above.

On the one hand, the procedure is necessary in order to realize the abstraction benefits of the recursive technique. You cannot make the code more efficient in terms of procedure calls without eliminating the helper entirely. Plus, since the helper has an Intention Revealing Name (Beck), its presence may help the reader follow your code better. But if the helper is never used by any other it procedure clutters the global name space and adds overhead to system execution.

Therefore, make the procedure local to its client. Remove any of its parameters that are not modified by the procedure on its recursive calls, instead referring directly to the argument passed by the enclosing context.

Applied to `annotate`, this pattern results in a single procedure:

```
(define annotate
   (letrec ((annotate-helper             ;; local helper
              (lambda (los position)
                 (if (null? los)
                     '()
                     (cons (list (car los) position)
                           (annotate-helper (cdr los)
                                            (+ position 1))))
             )))
      (lambda (los)
         (annotate-helper los 1)
      )))
```

Local Procedure can be applied to the result of any of the recursive structuring patterns. The main question to answer before applying it is, will any other procedure need this helper?

[What is the "exit context" for this pattern?]

7. Program Derivation

You are using Structural Recursion (1) and have created one or more helper procedures.

How do you get rid of a helper that was useful in the course of developing the code but which is unnecessary to the final product?

Your `replace` procedure, produced by Mutual Recursion (3), was easy to create because you followed the structure of the two inductive data definitions directly. It should be easy for readers of your code to understand for the same reason. But that ease comes at the expense of extra procedure calls. `replace` now makes two calls for each item in an s-list, instead of just one — even in situations where one call will suffice.

On the one hand, using two procedures enhances the ease of developing and reading the code. On the other hand, you can make your code more efficient by requiring fewer procedure calls. The increase in performance can be considerable. Eliminating the helper also unclutters the global name space.

Therefore, replace the procedure call with an application of its body to the arguments. Simplify the expression by substituting the arguments for parameters throughout the procedure body.

Applied to `replace`, this pattern results in a single procedure. Start by substituting an application of `replace-symbol-expr`'s lambda for the procedure call:

```
(define replace
   (lambda (new old slist)
      (if (null? slist)
          '()
          (cons ((lambda (new old sym-expr)
                   (if (symbol? sym-expr)
                       (if (eq? sym-expr old) new se)
                       (replace new old sym-expr)))
                  new old (car slist))
                (replace             new old (cdr slist)))
          )
   ))
```

Then replace the application of the lambda with the body of the lambda, substituting `new` for `new`, `old` for `old`, and `(car slist)` for `sym-expr`:

```
(define replace
   (lambda (new old slist)
      (if (null? slist)
          '()
          (cons (if (symbol? (car slist))
                       (if (eq? (car slist) old)
                           new
                           (car slist))
                       (replace new old (car slist)))
                (replace new old (cdr slist))))
    ))
```

The result is a single procedure that behaves exactly like the original two procedures. Note that this derived procedure is not the same as the original `replace` procedure from the motivating problem in Mutual Recursion (3) above. That procedure was hard to read and modify in part because the expression

```
(replace new old (cdr slist))
```

was repeated three times in the procedure's body. The derived procedure is nearly as readable as the two-procedure version, since you have collected the repeated subterm into a single instance. Of course, you cannot do this in a language whose if is a statement that does not return a value.

Generally, you do not want to apply Program Derivation to helpers generated by the Syntax Procedure (5) pattern, since there is no benefit to a derived procedure in which syntax procedures have been eliminated. Otherwise, this pattern can be applied to the result of any of the recursive structuring patterns. The main question to answer before applying it is, does the increased efficiency achieved more than offset the benefits of following the inductive data definition (chiefly, readability and ease of modification)?

[What is the "exit context" for this pattern?]

External Patterns

The following patterns are not a part of Roundabout but are referred to by one or more of its patterns. Links to on-line versions of the patterns are provided where available.

Simply Understood Code (Gabriel)

- Problem: People need to be comfortable reading a piece of code before they feel confident that they understand it and can modify it.
- Solution: "Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about."

Intention Revealing Name (Called "Intention Revealing Selector" by Beck)

- Problem: How do you name a procedure?
- Solution: Name procedures for what they accomplish.

Composed Procedure (Called "Composed Method" by Beck)

- Problem: How should you divide a program into components?
- Solution: "Divide your program into [procedures] that do one identifiable task. Keep all of the operations in a [procedure] at the same level of abstraction. This will naturally result in programs with many small [procedures], each a few lines long."

Collecting Parameter (Beck)

- Problem: How do you return a solution that is the result of collaboration among several procedures?
- Solution: Add a parameter to each of the procedures. Use this parameter as a means of passing the current state of the computation among the collaborators.

Role Suggesting Temporary Variable Name (Beck)

- Problem: What do you call a temporary variable?
- Solution: "Name temporary variables for the role they play in the computation. Use variable naming as an opportunity to communicate valuable tactical information to future readers."

Caching Temporary Variable (Beck)

- Problem: How do you improve the performance of a method, in the face of repeated expression evaluations?
- Solution: "Set a temporary variable to the value of the expression as soon as it is valid. Use the variable instead of the expression in the remainder of the method."

Acknowledgements

I thank shepherd Ian Chai for his helpful comments, which have helped me to improve Roundabout. I also thank Phillip J. Windley, whose lecture notes were the source of many of the code examples I use in Roundabout.

References

1. Kent Beck, **Smalltalk Best Practice Patterns**, Prentice Hall, New York, 1997.

   Role Suggesting Temporary Variable Name and Caching Temporary Variable are available on the web at:

   `http://c2.com/ppr/temps.html`

2. Richard Gabriel, "Simply Understood Code," quoted by Jim Coplien in "The Column Without a Name: Software Development as Science, Art, and Engineering," **C++ Report**, July/August 1995, pp. 14-19.

   Simply Understood Code is available on the web at:

   `http://c2.com/cgi/wiki?SimplyUnderstoodCode`

____