Functional Programming Patterns and Their Role in Instruction

Eugene Wallingford
Department of Computer Science
University of Northern Iowa
Cedar Falls, Iowa 50614-0507
wallingf@cs.uni.edu

Functional and Declarative Programming in Education Workshop
International Conference on Functional Programming
October 7, 2002

Outline

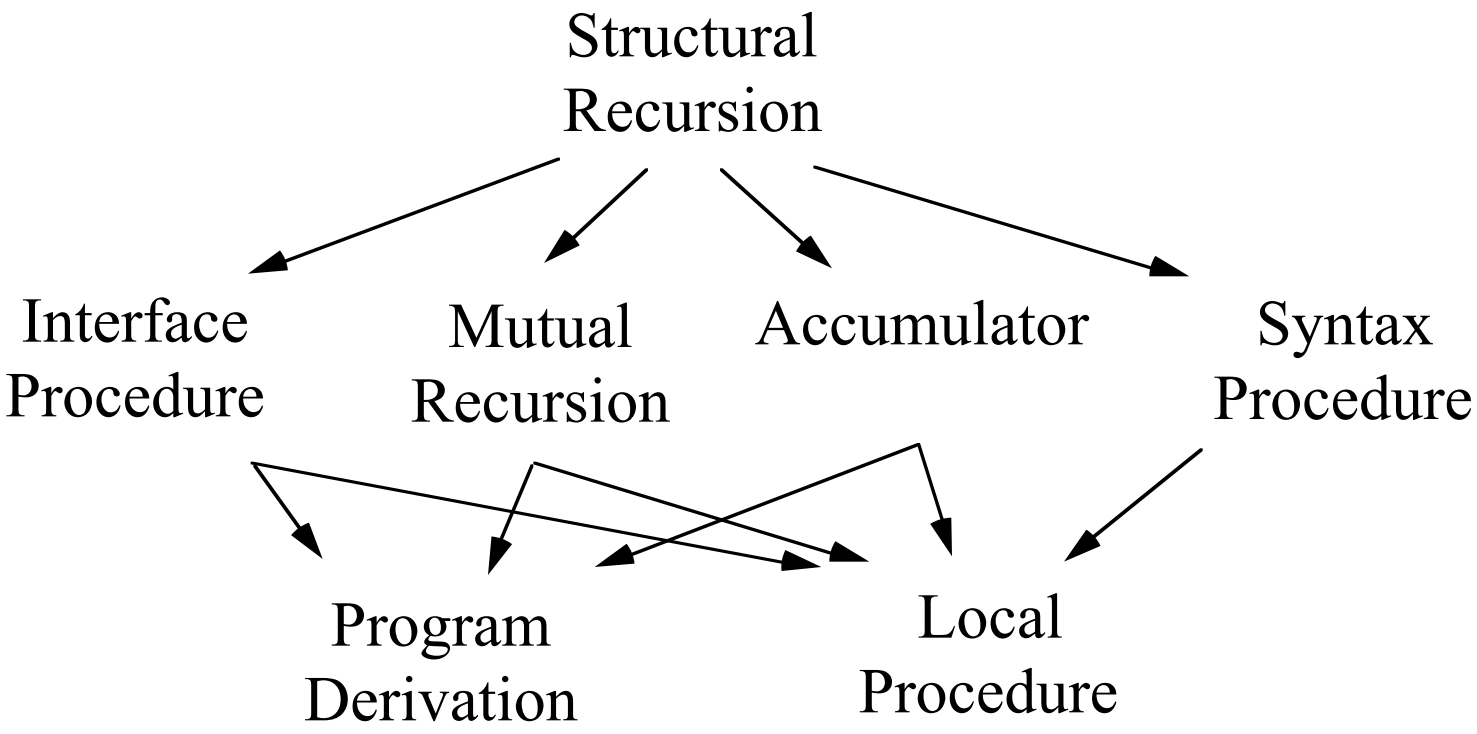
1. A Pattern Language for Recursive Procedures
2. Using Patterns in Instruction
3. The Value of Patterns
4. Future Directions

Patterns and Pattern Languages

A *pattern* is a recurring structure, shaped by the demands of the environment in which it occurs.

A *pattern language* documents how to build something. It is a collection of patterns that closely guides the creation of a meaningful whole.

Roundabout


Structural
Recursion
Interface
Procedure
Mutual
Recursion
Accumulator
Syntax
Procedure
Program
Derivation
Local
Procedure

Thumbnail: Structural Recursion

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

Write a recursive procedure with the same shape as the data type, with one case for each arm of the data specification. Give the procedure an Intention Revealing Name and each of its parameters a Type-Suggesting Parameter Name.

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, 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 Syntax Procedures (5).

Mutual Recursion

You are using Structural Recursion (1).

What should you do when the data 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))
```

Mutual Recursion

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, because 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, because 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, because you want your reader to be comfortable following the computation.

Mutual Recursion

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

To apply this pattern to `replace`, write two procedures. The first, replace, operates on s-lists. To write the arm of replace that handles symbol expressions, assume that the second, `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))))
```

Mutual Recursion (3) relies on multiple procedures making calls to one another. If this results in a program that is too inefficient, try using Program Derivation (7) to retain most of the benefits of the design while achieving better performance. Also, the extra procedures may clutter the global name space of the program or otherwise make code management more difficult. If that is the case, try using a Local Procedure (6) .

Using Roundabout in Instruction

To read as source material:

- by students
- by faculty new to functional programming

To organize lectures, for example:

- a failure-driven approach
- a before-and-after approach

To provide concrete guidance in evaluating designs.

Why Write Patterns?

Patterns offer a **vocabulary** for talking about design.

Pattern languages offer a **process** for constructing solutions.

The pattern form encourages **completeness**:

"Why should I do this?" and "What do I do now?"

Future Directions

Write small pattern languages for other functional programming topics, such as the uses of higher-order procedures, currying, and closures.

Incorporate elements of related projects, such as Refactoring Functional Programs and How To Design Programs.

Extend efforts to industrial-strength functional programming.