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

Abstract

Functional programming is a powerful style in which to write programs. However, students and faculty alike often have a hard time appreciating its beauty and learning to exploit its power. These difficulties arise in part because the functional style requires programmers to think at a higher level of abstraction, and in part because functional programs differ in fundamental ways from programs written in more common styles. A pattern language of functional programs can provide programmers with concrete guidance for writing functional programs and at the same time offer a deeper appreciation of functional style. Used effectively, such pattern languages will help functional programming educators reach a broader audience of faculty, students, and professionals. The result will be better programmers and better programs.

1 Introduction

Functional programming is a powerful style in which to write programs. Yet many students never fully appreciate the style, because they find it so foreign to their other programming experience. After programming in an imperative style, where state and state changes are central, the techniques and idioms of the functional style can feel restrictive or uncomfortable. The vocabulary of the style does not mesh with how they think about problems and programs. Further, many of the key ideas in functional programming involve abstractions beyond what other styles allow, and often students do not fully understand the reasons behind them or trade-offs entailed by their use.

Complicating matters, many university faculty do not have strong backgrounds in functional programming and so have a difficult time helping students to appreciate its beauty and power. Often these faculty have studied functional programming only for a university course or two as undergraduates themselves, yet find themselves teaching a course in which functional programming is a topic. Without experience working in the style over an extended period of time or on programs larger than toy assignments, they have never had the opportunity to develop a sense of what func-tional programming is like. Such faculty would like to convey the beauty of functional programming to their students but often find themselves teaching only the surface features of the style.

Surface features, though, do not provide enough insight for students or faculty to learn to read, let alone to write, programs in a functional style. Equipped only with the

vocabulary of a programming language and the basic concepts of a programming style, novices have difficulty recognizing higher-level structures in the programs they read. They face a similar problem when trying to write programs, as they tend to work at a low level of abstraction. What an expert programmer sees can seem incomprehensible to them. If the expert's knowledge were more readily available to novices, in a form that they could use to read and write programs, then perhaps they could develop their own expertise more readily.

1.1 Patterns as an Approach

Over the last decade, many software professionals have begun to explore the use of *patterns* as a means for recording and sharing expert knowledge for building software. In its simplest form, a pattern is a three-part rule that expresses a relation among

- a programming context,
- the set of competing concerns, or *forces*, that occurs repeatedly in that context,
- and a stereotypical software configuration that resolves these forces in a favorable way.

The context denotes the current state of a system, in particular the components already present and perhaps global constraints on the result. The set of forces constitutes a problem to be solved, and the stereotypical configuration is the solution that expert programmers use to solve it. In practice, patterns usually also explain why this solution suffices and describe how to implement the solution in code.

The value of a pattern lies largely in its ability to explain. Writing a pattern requires the author to state explicitly the context in which a technique applies and to state explicitly the design concerns and trade-offs involved in implementing a solution. These elements of a pattern provide significant benefits to the reader, who can not only study the technique that defines a solution but also explore when and how to use it. The author of a pattern can also benefit in a similar way, as the pattern form encourages a level of explicitness not always present in other written forms.

A *pattern language* is a collection of patterns that can be used together to create a larger program. In a pattern language, the context of each pattern refers to other patterns in the language. As such, the language guides the user through the process of building a complete solution to a more complex problem. This knowledge is prescriptive in that it specifies a partial order on the use of the patterns, which is helpful to novice programmers as they begin to implement more complex programs. But the patterns also contain information about context and design trade-offs, which enables the user to know when and how to make exceptions. In a similar way, a pattern language can also help its user to understand existing programs by describing the higher-level structures typically present in design and code.

Software patterns began as an industry phenomenon, an attempt to document bits of working knowledge that go beyond what developers learned in their academic study. They have become a standard educational device in industry. When computer science departments began to teach object-oriented (OO) design and programming in the 1990s, many academics used standard industry texts such as *Design Patterns* [6] as an important part of their preparation. Based in part on such experiences, faculty have incorporated patterns into their OO programming courses and pedagogy [e.g., 14, 15].

Given their utility in industry and in OO instruction, patterns offer a promising approach to help students and faculty learn to write functional programs. Such patterns will document the stereotypical techniques and program structures used by functional programmers, and pattern languages will document the process of using functional programming patterns in the construction of larger programs—ideally, complete programs that solve problems of real interest.

1.2 About This Paper

This paper recounts some of my work attempting to document patterns of functional programs and to use these small pattern languages to teach elements of functional programming. The next section describes one such pattern language, Roundabout, which documents patterns of recursive programs. Section 3 presents some of the ways I use Roundabout in my course and offers some qualitative evaluation of the results. Section 4 discusses related work involving patterns and other efforts to capture and teach the functional style. Finally, I close by suggesting future work on pattern languages of functional programs.

Note that all of my experiences with functional programming patterns to date involve teaching Scheme and Common Lisp, and the patterns described here reflect a Scheme programming style. The details of many of these patterns are language-specific, but the idea of functional programming patterns and their use in teaching extends beyond any particular language.

2 Functional Programming Patterns

I have written two small pattern languages for use in courses. Roundabout, a pattern language for recursive programs over inductively-defined types, is the more complete and more course-tested of the two. It comprises seven patterns often present in such recursive programs.

Structural Recursion is the entry point to the language, specifying that the shape of a procedure should match the shape of the inductive type definition that it processes. Patterns in the second layer expand on the first and document primary techniques used to structure recursive programs, in particular the creation of particular kinds of helper procedures. Finally, patterns in the third layer describe how to improve the programs generated by the first five in the face of other forces such as run-time efficiency or name-space clutter.

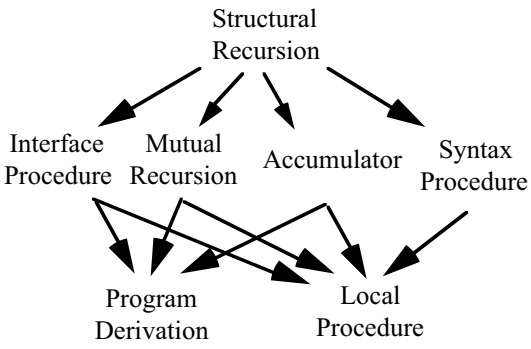

Fig. 1: The patterns of Roundabout

The patterns of Roundabout collaborate to create complete solutions to problems involving inductively-defined types. One way they do this is through the contexts of the

patterns, which record the relationships between patterns explicitly. Figure 1 shows a map of these relationships. Each link indicates that the first pattern appears in the context of the second. This means that, after applying the first pattern, you may want to apply the second pattern to address some forces in the solution unresolved by the first.

2.1 The Anatomy of a Pattern

Each pattern in Roundabout consists of six sections. This section considers each, using the Accumulator pattern as an example. The Mutual Recursion pattern, given in Appendix 1, provides a second example.

Context: a statement of the patterns and global factors that must be present before this pattern is relevant. This is a precondition on the pattern's applicability.

- Accumulator presumes that the program uses Structural Recursion. Because Mutual Recursion introduces a second procedure, which is central to Accumulator's intent, its context mentions Mutual Recursion specifically.

Problem: a concise statement of the specific issue addressed by this pattern. The problem solved by a pattern embodies its intent.

- Accumulator addresses a problem that sometimes occurs when multiple procedures collaborate: the natural flow of a recursive program can lead to unnecessarily inefficient code.

Forces: a discussion of the competing constraints that face programmer trying to solve the problem. This discussion motivates the pattern's use by showing how other solutions do not adequately solve the problem. Often, this section introduces a sample problem to make the discussion of the pattern more concrete.

- The text of the Accumulator pattern uses the example of procedure that flattens nested lists. A straightforward use of Mutual Recursion results in a helper procedure that operates on the nested lists, but the primary procedure must append the helper's results onto the results of the recursive call. This solution is $O(n^2)$ in the size of the data. This behavior is unacceptable for all but the simplest inputs. The naturalness and readability of the two-procedure solution conflicts with the inefficiency of the resulting code.

Solution: a description of the pattern's structure in a program, along with a description of how to implement it. If this pattern has a complex structure, the solution may describe the components of the structure and their implementation as well.

- The programmer can resolve the competing forces in the flattening procedure by introducing an accumulator variable to hold the result of the computation as it is built. Both of the mutually-recursive procedures receive the accumulator as an argument, so each can add to the solution when-ever it desires while controlling the flow of the computation independently. Implementing an accumulator requires that the top-level procedure become an Interface Procedure that passes the initial value of the accumulator variable to a helper procedure that does the computation.

Problem Resolved: a demonstration of how the pattern's structure solves the problem and resolves many or all of the forces at play. Sometimes, this section explicitly discusses the forces and how the pattern addresses them; other times, it only shows the pattern implemented for the sample problem introduced in the forces section.

- This section of Accumulator shows that it solves the flattening problem in a way that uses mutual recursion and computes its answer in O(n) time. The pattern's text might be improved by adding a more explicit discussion of how the pattern resolves the forces.

Resulting Context: a statement of the patterns that may be used to address any constraints that affect the resulting program structure. These constraints may be forces left unresolved by this pattern or new forces introduced by implementing the structure.

- Finally, an Accumulator may leave global forces out of balance. Due to its reliance on multiple procedures that call each other, the program may be inefficient. The Program Derivation pattern may be able to retain most of the benefits of the design while achieving better performance. Also, the extra procedure may clutter the name space of the system or otherwise make code management more difficult, which can be resolved in part by use of the Local Procedure pattern.

The pattern form describes both a structure and how and when to build it. As a pattern writer, I have had to pay careful attention to why I use a given structure in a particular program, why I choose to implement it at that point in the process, and why it is better than some other in that situation. These answers can help my students, who are new to functional programming, to learn the vocabulary of the style and to think critically about the writing of programs.

2.2 Writing a Program with Roundabout

Taken as a whole, Roundabout can guide a programmer in the creation of a complete solution to a given problem. Sometimes, the resulting program will be different from any program a student would have built on his or her own.

Consider the task of writing the procedure `replace`, which replaces all occurrences of one symbol with another in an s-list. (See Appendix 1 for a more detailed coverage of the early part of this example.) Because the s-list is defined inductively, the pattern *Structural Recursion* applies. This pattern says to build a procedure with one arm for each type of s-list. This results in a procedure of this form:

```
(define replace
  (lambda (new old slist)
    (if (null? slist)
        '()
        ; handle compound case )))
```

The case for handling a non-empty s-list involves two choices of its own: Is the first item in the list a symbol? And, if so, is it the symbol to be replaced? This requires a nested `if` expression of its own:

```
(if (symbol? (car slist))
   ; handle two possible cases
   ; replace in both parts )
```

But the s-list data type consists of two mutually inductive data definitions, one for s-lists and one for symbol expressions. The resulting context of the Structural Recursion pattern tells us that such data types should be processed using the specialized *Mutual Recursion* pattern. This pattern applies Structural Recursion to both data definitions, with the resulting procedures making calls to one another in the appropriate locations.

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

(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))))
```

The resulting procedures are truer to the data definition than a one-procedure solution might be, and they do not repeat code in the way a naive application of Structural Recursion would have.

The use of mutually recursive procedures interjects new forces into the problem. For inputs with deeply nested lists, many of the mutually recursive calls will result in immediate calls back to the calling procedure. If this sort of inefficiency is unacceptable, then Roundabout suggests the *Program Derivation* pattern as a potential technique for resolving the forces. Program derivation is a technique for using functional programming's substitution model to generate code.

Applied to the two procedures above, this technique gives the following solution:

```
(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))))))
```

This result can be quite surprising to novices. The final solution consists of a single procedure, which is often their goal, yet it is unlike any procedure that they would have

written from scratch themselves. They see how to use Scheme's tools to create a solution with little or no repeated code, much as an expert programmer might. Students are gratified that they can approach such expert results by using an almost mechanical technique, which further motivates their use of the pattern language.

The pattern language form makes explicit the relationships among the structures that compose a whole. These explicit connections guide students in the process of developing a program while leaving room for them to make trade-offs in the application of the patterns. Through experience, the programmer will internalize the pattern language and begin to develop instincts for the programming style.

3 Teaching with Patterns

I have used Roundabout in three different courses. I originally wrote the language as I learned how to teach functional programming as a major component of my department's Programming Languages and Paradigms course, and it now serves as the basis for a three-week unit on recursive programming. Later, I made Roundabout supplemental reading for a 1-credit Lisp programming lab in our Artificial Intelligence course. Finally, I have used some of Roundabout's patterns when teaching a C++-based Data Structures course. Roundabout affects my teaching in several ways.

Lecture organization. Roundabout strongly affects how I organize course sessions. The pattern form itself present ideas in a good way for to students. A pattern begins with a problem, explores the forces that drive the solution, and then describes a solution that balances the forces. The discussion of the forces can explore other candidate solutions and why they are not good enough, which helps motivate the technique that ultimately solves the problem. Presentation of the solution can also explore why the solution works and what new forces it introduces. This discussion helps students discern when to use a technique and when to look for a better solution.

Another approach gives students a "before and after" picture: show them a program that solves a program in a naive way, perhaps using the techniques that they know at that point, and then show a refactored solution that employs a new pattern. This technique works better for a larger problem and a larger pattern, as that yields a starker contrast between solutions. This approach can motivate students to study the pattern carefully, once they have seen the kind of difference they can make.

Using a pattern language can also help the instructor design of several sessions that deal with the patterns working together. The map of Roundabout given in Figure 1 gives a natural partial order on the presentation of topics, and the contexts and resulting contexts of the patterns offer connective tissue for the transitions between patterns.

Assignment creation. Using Roundabout affects how I select homework and exam problems by encouraging the use of problems that (1) explore each pattern, (2) cause the student to choose among techniques based on the forces in the problem, and (3) cause the student to apply several patterns to create of a single solution. While these seem to be natural goals for good assignments, and textbook authors often cover all three, I find that the pattern language helps me to design assignments with more complete coverage.

Design evaluation. A pattern language can serve as a useful tool in the task of evaluating student designs, for the purposes of giving assistance, feedback, and grades [13]. The patterns in the language constitute a vocabulary for describing designs, and the language gives rules for generating good designs. And, most important from the perspective of evaluation, the pattern contexts give an explicit standard with which to evaluate designs. By specifying more completely when each pattern applies and what forces it addresses, the pattern language guides students in applying the techniques they learn, and it gives instructors a concrete reference for critiquing student work.

3.1 Student Perceptions

I can offer some qualitative evaluation of the effects that teaching with this pattern language has had on my students' performance and perceptions. I have taught our programming languages course five times using Roundabout. At the end of each semester, students evaluate the course in small groups during the last session of the semester, and each small group then reports its results to the class as a whole. These public evaluations typically list 'recursive programming' as one of the three most valuable items learned during the semester. This seems significant because by this time students have studied and used recursion in at least two prior courses. Their new confidence could result simply from growing experience, but students report that Roundabout has helped them learn how to use recursion more comfortably.

Students also complete an anonymous course evaluation, on which they may comment on any part of the course. Most students do not comment but, when they do, approximately one-fourth mention Roundabout, always favorably.

Student scores on recursive programming exercises have improved by an average of 1.5 points out of 20 since I began to use this pattern language in class. This may only be a result of the fact that I now do a better job teaching recursive programming than I did before, but that is one of the central reasons to use the pattern form: it encouraged and sometimes forced me to understand and explain why and when to use the techniques that I was teaching.

This improvement seems to occur whether I discuss the patterns explicitly in class or use them only as a way to structure the content of the course. The one semester that I did not see such an improvement was last spring, when I stated directly that use of the patterns was "optional": the patterns demonstrate good technique, but students were responsible only for their final solutions and not for their technique. I was surprised to find that many students chose not to use them, instead relying on their past experience with recursion. On average, student scores on recursive programming exercises fell back near previous levels. This indicates that students require stronger encouragement to try the techniques in the first place.

However, this encouragement must focus on the content of the patterns and not the patterns *qua* patterns. One of the possible drawbacks of using a pattern language to teach is that instructors or students focus on pattern names and "learning the patterns", and thus lose sight of the fact that patterns simply document good style and technique for solving a class of problems. Learning how to build good solutions is the goal, not memorizing patterns.

4 Related Work

We can view the use of pattern languages in the teaching of functional programming in relationship to work in the functional programming community, the patterns community, and in the broader computer science education community.

4.1 Functional Programming Community

Several groups are doing work to help us improve how we teach functional programming. Among them are at least two in a similar spirit as my work with patterns. One approach, exemplified in the introductory text *How to Design Programs* [4], focuses on the design process that programmers use to *create* programs. Another approach focuses on how programmers *evolve* their programs through refactoring [10].

Pattern languages complement these approaches by defining both a vocabulary and a process. Individual patterns can serve as components in a program being built using another design process, because they specify program structures and when and how to build them. The pattern language can be used to augment the design process, either by giving specific examples of design steps or by filling in gaps in the process. Further, it can also be used in deciding when and how to refactor, because each pattern specifies when it applies and how it resolves the forces at play. Refactoring often involves replacing one pattern with another from the same pattern language that generated the program. (This idea has begun to attract research interest in the broader refactoring community.)

4.2 Software Patterns Community

One of the goals of software patterns is to communicate expertise, and so most pattern languages aim to teach. For historical reasons, the bulk of the current patterns literature deals with OO programming, telecommunication systems, and organizations. Recent book-length efforts have extended pattern approaches to interaction design and the design of small-memory systems. However, several papers document patterns of functional and declarative programming.

In addition to Roundabout, I have also written Envoy, a pattern language for programs that maintain state using closures [11]. Ferguson and Deugo discuss stereotypical ways to build complex control structures using `call-with-current-continuation` in Scheme [5]. Hanmer describes techniques for writing efficient logic programs in Prolog [7].

Some patterns work bridges the OO and functional programming communities. Sandu and Deugo [9] consider how OO programmers use anonymous behaviors to configure higher-order constructs or to defer specifying a computation until a later time. Kuehne [8] shows how common patterns of functional programming can generate desirable OO designs.

4.3 Computer Science Education Community

OO design patterns have become a standard part of the computer science curriculum, and *Design Patterns* [6] has become a standard text for teaching OO program-ming courses after the first year curriculum. Many educators have also developed materials based on these patterns for teaching first-year courses [14, 15].

A small group of computer science educators have documented so-called *elementary patterns* for teaching object-oriented and procedural programming. These patterns seek to capture the low-level design and programming knowledge that underlies more advanced patterns, the sort of knowledge that every expert programmer has and uses subconsciously. This work includes patterns of selection [2], loops [1], and substitutability and delegation [3]. In many ways, my work with Roundabout explores patterns of functional programming at the same level, appropriate for relative novices with the style.

5 Conclusion

This paper describes some of my work documenting patterns of functional programs and using these small pattern languages to teach elements of functional programming. In this approach, patterns serve as a vocabulary of design elements; the pattern language itself documents both relationships among patterns and a process for generating stylistically pleasing programs that meet functional goals. The pattern form encourages the author to explain carefully when each design element works best, and why, and such explanation can help students better understand how to use the techniques they learn. These benefits also accrue to instructors who read and write patterns, perhaps especially to those who write them.

To follow this approach more widely, programmers and educators need to build a more complete literature of pattern languages of functional programs. The existing literature fills only a few small niches. The set of topics yet to describe is rich and varied, from the uses of techniques such as higher-order procedures and currying to more advanced topics such as monads. Another fruitful avenue is to study the targets of typical refactorings of functional programs.

More work also needs to be done to explore and document ways to use patterns effectively in teaching. The value of such work can hardly be overestimated. Though faculty generally design a curriculum with their students in mind, often it most benefits other faculty who will teach it. Teachers need to learn both the content of the approach as well as how to present it effectively. The largest first-order effect of design patterns on OO programming instruction has been on the instructors, and this would likely hold for functional programming instruction, too.

The value in documenting pattern languages of functional programs extends beyond its potential effects on university teaching. Pattern languages share expert programming knowledge with a larger audience. Design patterns have played a central role in the transfer of OO technology within industry. Documenting pattern languages of functional programs may help to reveal the beauty and power of functional programming to a wider audience of programmers.

References

[1] Owen Astrachan and Eugene Wallingford, *Loop Patterns*, Proc. Fifth Pattern Languages of Programs Conference, Allerton Park, Illinois. 1998.

[2] Joe Bergin, *Selection Patterns*, Proc. Fourth European Pattern Languages of Programs Conference, Bad Irsee, Germany, 1999.

[3] Dwight Deugo, *Foundation Patterns*, Proc. Fifth Pattern Languages of Programs Conference, Allerton Park, Illinois. 1998.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*, MIT Press, Cambridge, Massachusetts, 2001.

[5] Darrell Ferguson and Dwight Deugo, *Call with Current Continuation Patterns*, Proc. Eighth Pattern Languages of Programs Conference, Allerton Park, Illinois, 2001.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison Wesley, New York, 1995.

[7] Robert Hanmer, *Patterns of Efficient Prolog Programs*. Proc. Third Pattern Languages of Programs Conference, Allerton Park, Illinois, 1996.

[8] Thomas Kuehne, *A Functional Pattern System for Object-Oriented Design*, Verlag Dr. Kovac, Hamburg, Germany, 1999.

[9] Dorin Sandu and Dwight Deugo, *The Lambda Pattern*, Proc. Sixth Pattern Languages of Programs Conference, Allerton Park, Illinois, 1999.

[10] Simon Thompson and Claus Reinke, *Refactoring Functional Programs*, Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001.

[11] Eugene Wallingford, *Envoy: A Pattern Language for Maintaining State in a Functional Program*, Proc. Sixth Pattern Languages of Programs Conference, Allerton Park, Illinois, 1999.

[12] Eugene Wallingford, *Roundabout: A Pattern Language for Recursive Programming*. Proc. Fourth Pattern Languages of Programs Conference, Allerton Park, Illinois, 1997.

[13] Eugene Wallingford, *Using a Pattern Language to Evaluate Design*, OOPSLA Workshop on Evaluating Object-Oriented Design, Vancouver, British Columbia, 1998.

[14] Michael Wick, Kaleidoscope: Using Design Patterns in CS1, *SIGCSE Bulletin* 33(1):258-262, 2001.

[15] Stephen Wong and Dung Nguyen, Design Patterns for Games, *SIGCSE Bulletin* 34(1):126-130, 2002.

Appendix 1: The *Mutual Recursion* pattern from Roundabout [12]

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))
```

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.

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, `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) .