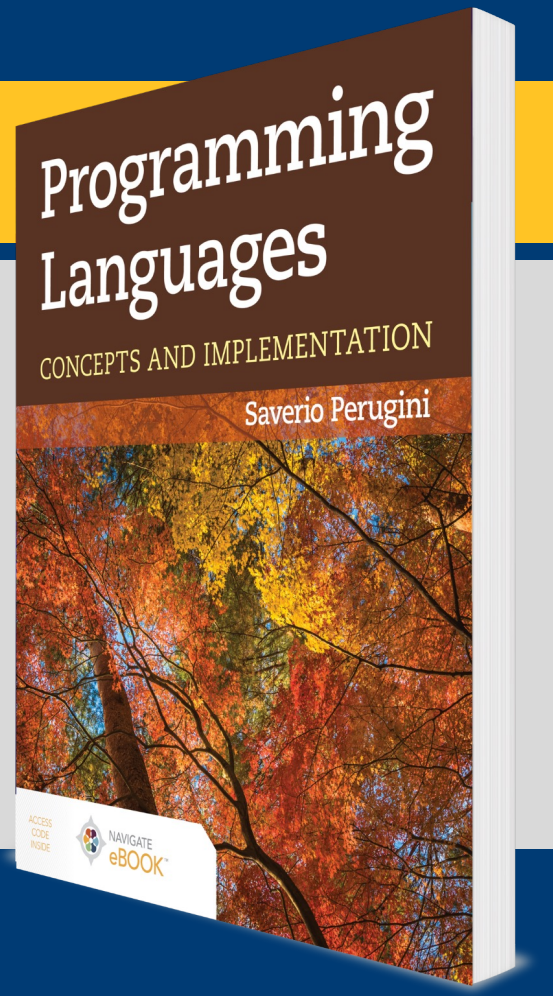
CHAPTER 5

Functional Programming in Scheme



Chapter 5: Functional Programming in Scheme

[L]earning Lisp will teach you more than just a new language—it will teach you new and more powerful ways of thinking about programs.

— Paul Graham

A programming language that doesn't change how you think about programming isn't worth knowing.

-- Alan Perlis

Functional programs operate by returning values rather than modifying variables—which is how imperative programs work.

5.1 Chapter Objectives

- Foster a recursive-thought process toward program design and implementation.
- Understand the fundamental tenets of functional programming for practical purposes.
- Explore techniques to improve the efficiency of functional programs.
- Demonstrate by example the ease with which data structures and programming abstractions are constructed in functional programming.
- Establish an understanding of programming in Scheme.
  - Your text uses Scheme; we're using Racket for this course. What's the difference?
  - Very little. Scheme code will usually run without modification as Racket code.
  - Racket has some additional features added.

5.2.1 Hallmarks of Functional Programming

- Functions are first-class entities.
- Often use higher-order functions (functions that return functions) to build abstraction.
- Recursion, rather than iteration, is the primary mechanism for repetition.
- No or little provision for side effects
- Typically use automatic garbage collection
- Usually do not involve direct manipulation of pointers by the programmer
- Historically, considered languages for artificial intelligence, but this is no longer the case

5.2.2 Lambda Calculus

<*expression*> ::= <*identifier*>
(a symbol)

<*expression*> ::= `(lambda (`<*identifier*>`)` <*expression*>`)`
(a function definition)

<*expression*> ::= `(`<*expression*> <*expression*>`)`
(a function application)

5.3 Lisp

- Ideally situated between formal mathematics and natural language (Friedman and Felleisen 1996b, Preface)
- “If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases” (Friedman and Felleisen 1996b, Afterword).
- It is a metalanguage: a language for creating languages (Sussman, Steele, and Gabriel 1993).
- Lisp is the second oldest programming language. (Which is the oldest?)
- The Lisp interpreter operates as a simple interactive *read-eval-print loop* (REPL; sometimes called an *interactive toplevel*).

5.3.2 Lists in Lisp (1 of 2)

Lisp syntax (programs or data) is made up of S-expressions (i.e., symbolic expressions).

```
<symbol-expr>          ::=  <symbol>
<symbol-expr>          ::=  <s-list>
<s-list>               ::=  ()
<s-list>               ::=  (<list-of-symbol-expr>)
<list-of-symbol-expr>  ::=  <symbol-expr>
<list-of-symbol-expr>  ::=  <symbol-expr> <list-of-symbol-expr>
```

5.3.2 Lists in Lisp (2 of 2)

- Examples of Lisp lists, which are also S-expressions:

```
(1 2 3)
(x y z)
(1 (2 3))
((x) y z)
```

- More examples of S-expressions:

```
(1 2 3)
(x 1 y 2 3 z)
((((Nothing))) ((will) (()()) (come ()) (of nothing)))
```

Formal Parameters Vis-à-Vis Actual Parameters

- *Formal parameters* (also known as *bound variables* or simply *parameters*) are used in the declaration and definition of a function.
- *Actual parameters* (or *arguments*) are passed to a function in an invocation of a function.

Review of Recursion

1. Identify the smallest instance of the problem—the base case—and solve the problem for that case only.
2. Assume you already have a solution to the penultimate (in size) instance of the problem named `n - 1`. Do not try to solve the problem for that instance. Remember, you are assuming it is already solved for that instance. Now given the solution for this `n - 1` case, extend that solution for the case `n`. This extension is much easier to conceive than an original solution to the problem for the `n - 1` or `n` cases.

See *Design Guideline 1: General Pattern of Recursion* in Table 5.7

Figure 5.1 List Box Representation of a Cons Cell

- The `cdr` is a pointer to the tail of the list as a list.
- The `car` is a pointer to the head of the list as an atom or a list.
- In Racket, there are equivalent functions (first L) and (rest L)

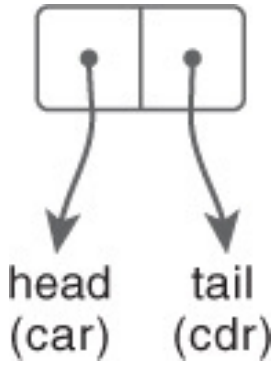


Figure 5.2 `'(a b) = '(a . (b))`

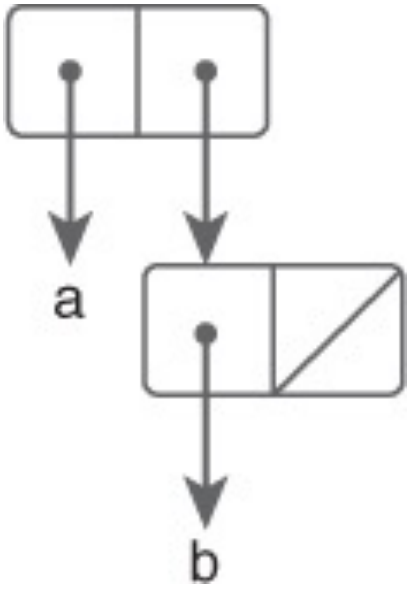


Figure 5.3 `'(a b c) = '(a . (b c)) = '(a . (b . (c)))`

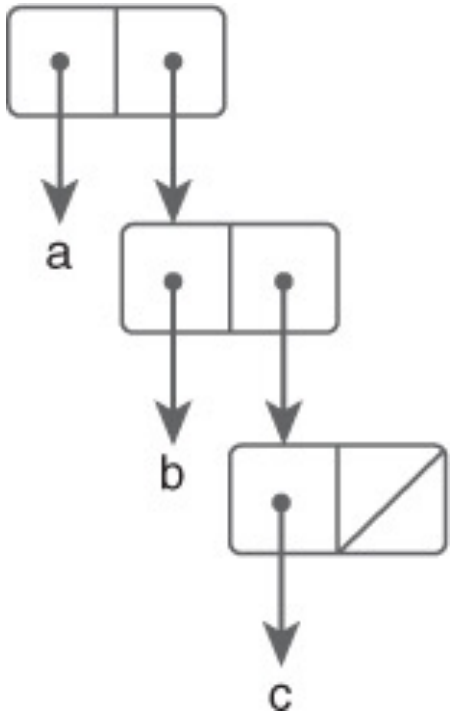


Figure 5.4 '(a . b)

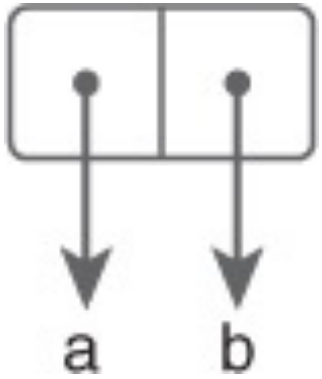


Figure 5.5

`'((a) (b) ((c))) = ' ((a) . ((b) ((c)))) = ' ((a) . ((b) . (((c)))))`

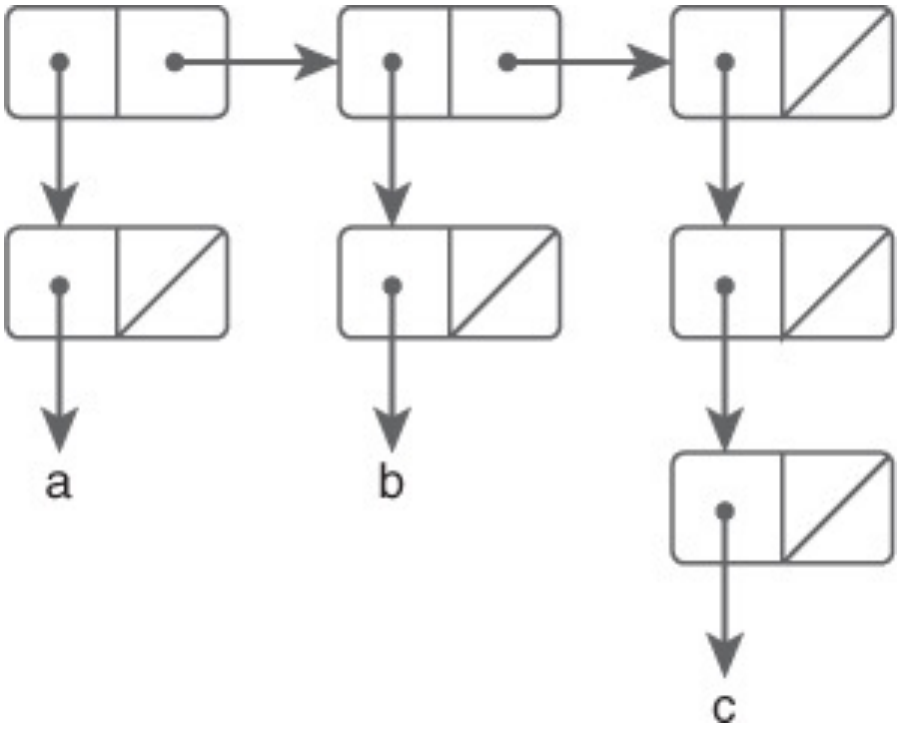


Figure 5.6 '(((a) b) c)

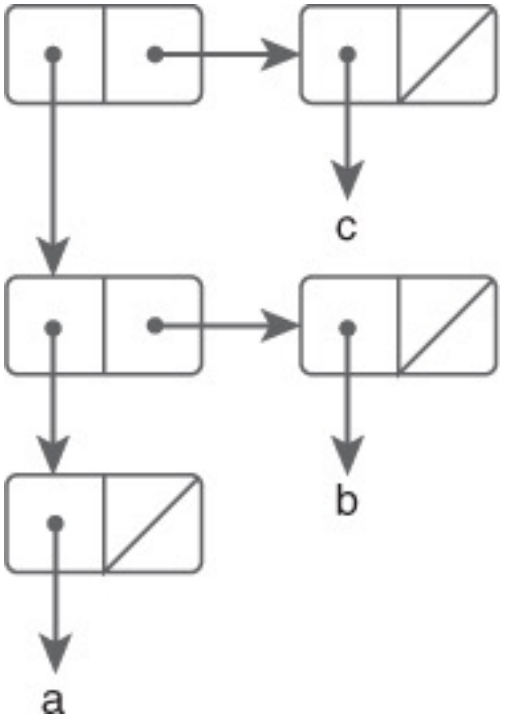


Figure 5.7

`'((a b) c) = ' (((a) b) . (c)) = ' (((a) . (b)) . (c))`

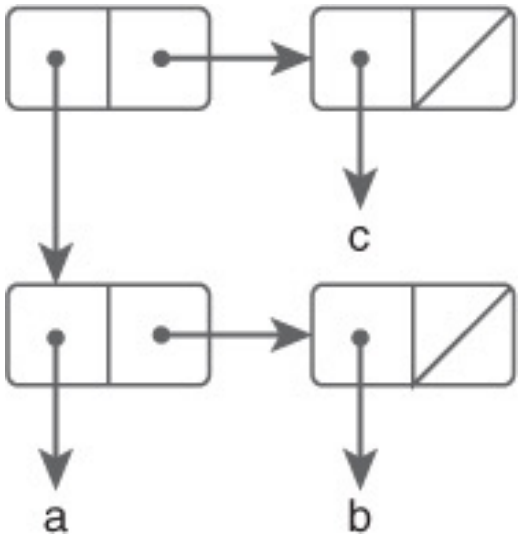
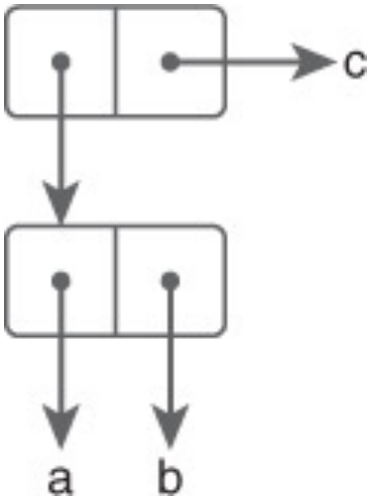


Figure 5.8 `'((a . b) . c)`



5.6.1 A List `length1` Function

- `length`: given a list, returns the length of the list:

```
(define length1
   (lambda (l)
      (cond
         ((null? l) 0)
         (else (+ 1 (length1 (cdr l)))))))
```

- Built-in Scheme predicate `null?` returns `#t` if its argument is an empty list and `#f` otherwise.
- Editorial note: Any length function requires linear time (each item must be counted). This version also requires linear space.

5.6.2 Run-Time Complexity: `append` and `reverse`
(1 of 4)

- Built-in Scheme function `append`:

```
1  (define append1
2     (lambda (x y)
3        (cond
4           ((null? x) y)
5           (else (cons (car x) (append1 (cdr x) y))))))
```

- The run-time complexity of `append` is linear [or *O(n)*] in the size of the first list.

5.6.2 Run-Time Complexity: `append` and `reverse`
(2 of 4)

- While the running time of `append` is not constant like that of `cons`, it is also not polynomial [e.g., $O(n^2)$].
- However, the effect of the less efficient `append` function is *compounded* in functions that use `append` where the use of `cons` would otherwise suffice.
- The `reverse` function accepts a list and returns the list reversed:

```
(define reverse1
   (lambda (l)
      (cond
         ((null? l) '())
         (else (append (reverse1 (cdr l)) (cons (car l) '()))))))
```

- Scheme uses *pass-by-value* parameter passing.
- Notice that `reverse1` can reverse a list of employee records or pixels, or reverse a list involving a combination of multiple types.
- It can even reverse a list of lists.

5.6.2 Run-Time Complexity: `append` and `reverse`
(3 of 4)

- This expansion illustrates how, in reversing the list `(a b c)`, the expression in the `else` clause is expanded:

```
1   (append (reverse1 '(b c)) (cons a '()))
2   (append (reverse1 '(b c)) '(a))
3   (append (append (reverse1 '(c)) (cons b '())) '(a))
4   (append (append (reverse1 '(c)) '(b)) '(a))
5   ;; base case
6   (append (append (append (reverse1 '()) (cons c '())) '(b)) '(a))
7   (append (append (append '() '(c)) '(b)) '(a))
8   (append (append '(c) '(b)) '(a))
9   (append '(c b) '(a))
10  (append '(c b a))
```

- As this expansion illustrates, reversing a list of *n* items requires *n*-1 calls to append.
- The running time of `append` is linear, *O(n)*.

5.6.2 Run-Time Complexity: `append` and `reverse`
(4 of 4)

- But the run-time complexity of this definition of `reverse1` is $O(n^2)$, which is unsettling.
- Intuitively, to reverse a list, we need pass through it only once; thus, the upper bound on the running time should be no worse than $O(n)$.
- The difference in running time between `cons` and `append` is *magnified* when `append` is employed in a function like `reverse1`, where `cons` would suffice.
- *Design Guideline 3: Efficient List Construction:* never use `append` where `cons` will suffice

5.6.3 The Difference Lists Technique (1 of 2)

- Without side effects, which are contrary to the spirit of functional programming, the only ways for successive calls to a recursive function to share and communicate data is through return values (as is the case in the `reverse1` function) or parameters.
- The *difference lists technique* involves using an additional parameter that represents the solution (e.g., the reversed list) computed thus far.

```
1   (define reverse1
2     (lambda (l)
3       (cond
4         ((null? l) '())
5         (else (rev l '())))))
6
7   (define rev
8     (lambda (l rl)
9       (cond
10        ((null? l) rl)
11        (else (rev (cdr l) (cons (car l) rl))))))
```

5.6.3 The Difference Lists Technique (2 of 2)

- Conducting a similar run-time analysis of this version of `reverse1` as we did with the prior version, we see:

```
(reverse1 '(a b c))
(rev '(a b c) '())
(rev '(b c) (cons (car '(a b c)) '()))
(rev '(b c) (cons 'a '()))
(rev '(b c) '(a))
(rev '(c) (cons (car '(b c)) '(a)))
(rev '(c) (cons 'b '(a)))
(rev '(c) '(b a))
(rev '() (cons (car '(c)) '(b a)))
(rev '() (cons 'c '(b a)))
;; base case
(rev '() '(c b a))
(c b a)
```

- Now the running time of the function is linear [i.e., $O(n)$] in the size of the list to be reversed.

Outline

- 5.1 Chapter Objectives
- 5.2 Introduction to Functional Programming
- 5.3 Lisp
- 5.4 Scheme
- 5.5 `cons` Cells: Building Blocks of Dynamic Memory Structures
- 5.6 Functions on Lists
- **5.7 Constructing Additional Data Structures**
- **5.8 Scheme Predicates as Recursive-Descent Parsers**
- **5.9 Local Binding: `let`, `let*`, and `letrec`**
- **5.10 Advanced Techniques**
- 5.11 Languages and Software Engineering
- 5.12 Layers of Functional Programming
- 5.13 Concurrency
- 5.15 Thematic Takeaways

5.7.1 A Binary Tree Abstraction (1 of 3)

Consider the following BNF specification of a binary tree:

```
<bintree>  ::=  number
<bintree>  ::=  (<symbol> <bintree> <bintree>)
```

The following sentences in the language defined by this grammar represent binary trees:

```
111
32
(opus 111 32)
(sonata 1820 (opus 111 32))
(Beethoven (sonata 32 (opus 110 31)) (sonata 33 (opus 111 32)))
```

5.7.1 A Binary Tree Abstraction (2 of 3)

The following function accepts a binary tree as an argument and returns the number of internal and leaf nodes in the tree:

```
1  (define bintree-size
2     (lambda (s)
3        (cond
4           ((number? s) 1)
5           (else (+ (bintree-size (car (cdr s)))
6                    (bintree-size (car (cdr (cdr s))))
7                    1))))) ; count self
```

Table 5.1 Examples of Shortening `car-cdr` Call Chains with Syntactic Sugar

```
(car (cdr (cdr (cdr '(a b c d e f))))) = (cadddr '(a b c d e f)) = d
(car (car (car '(((a b)))))) = (caaar '(((a b)))) = a
(car (cdr (car (cdr '(a (b c) d e))))) = (cadadr '(a (b c) d e)) = c
(cdr (car (cdr (car '((a (b c d)) e f))))) = (cdadar '((a (b c d)) e f)) = (c d)
```

- Note that Racket has functions `(first L)`, `(second L)`, `(third L)`, etc., through `(tenth L)`.
- It also has `(take L n)`, to return the first n elements of L, and `(drop L n)`, which returns list L with the first n items removed.
- In general, `(list-ref L pos)` returns the item at position pos.
- And `(list-tail L pos)` returns the remainder of the list after the first pos elements

5.7.1 A Binary Tree Abstraction (3 of 3)

Thus, we can rewrite `bintree-size` as follows:

```
(define bintree-size
   (lambda (s)
      (cond
         ((number? s) 1)
         (else (+ (bintree-size (cadr s))
                  (bintree-size (caddr s))
                  1)))))
```

Exercise: Rewrite using Racket idiom (first, second, etc)

5.8.1 atom?, list-of-atoms?, and list-of-numbers?
(1 of 4)

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))

;;; first version
(define list-of-atoms?
    (lambda (lst)
      (cond
         ((null? lst) #t)
         ((atom? (car lst)) (list-of-atoms? (cdr lst)))
         (else #f))))

;;; second version
(define list-of-atoms?
  (lambda (lst)
    (cond
      ((null? lst) #t)
      (else (and (atom? (car lst))
                 (list-of-atoms? (cdr lst)))))))
```

5.8.1 atom?, list-of-atoms?, and list-of-numbers?
(2 of 4)

```
;;; final version
  (define list-of-atoms?
    (lambda (lst)
      (or (null? lst)
          (and (pair? lst)
               (atom? (car lst))
               (list-of-atoms? (cdr lst))))))

;;; same structure
  (define list-of-numbers?
    (lambda (lst)
      (or (null? lst)
          (and (pair? lst)
               (number? (car lst))
               (list-of-numbers? (cdr lst))))))
```

5.8.1 atom?, list-of-atoms?, and list-of-numbers?
(3 of 4)

```
;;; let parameterize the predicate
(define list-of
  (lambda (predicate lst)
    (or (null? lst)
        (and (pair? lst)
             (predicate (car lst))
             (list-of predicate (cdr lst))))))
```

```
> (list-of atom? '(a b c d))
#t
> (list-of atom? '(1 2 3 4))
#t
> (list-of atom? '((a b) c d))
#f
> (list-of atom? 'abcd)
#f
> (list-of number? '(1 2 3 4))
#t
> (list-of number? '(a b c d))
#f
> (list-of number? '((1 2) 3 4))
#f
```

5.8.1 atom?, list-of-atoms?, and list-of-numbers?
(4 of 4)

```
;;; updated version
 (define list-of
   (lambda (predicate)
     (lambda (lst)
       (or (null? lst)
           (and (pair? lst)
                (predicate (car lst))
                ((list-of predicate) (cdr lst)))))))
(define list-of-atoms? (list-of atom?))
(define list-of-numbers? (list-of number?))
```

```
(define list-of
  (lambda (predicate)
    (letrec ((list-of-helper
              (lambda (lst)
                (or (null? lst)
                    (and (pair? lst)
                         (predicate (car lst))
                         (list-of-helper (cdr lst)))))))
      list-of-helper)))
```

5.9.1 The `let` Expression

- Local binding through the `let` construct:

```
> (let ((a 1) (b 2))
>   (+ a b))
3
```

- Bindings are created in *parallel* in the list of lists immediately following `let`
  - [e.g., `((a 1) (b 2))`]

and are only bound during the evaluation of the second S-expression

  - [e.g., `(+ a b)`].

- Use of `let` does not violate the spirit of functional programming for two reasons:
  - (1) `let` creates bindings, not assignments
  - (2) `let` is syntactic sugar used to improve the readability of a program
- Any `let` expression can be rewritten as an equivalent `lambda` expression.

The `let*` Expression

- We can produce *sequential* evaluation of the bindings by nesting `let`s:

```
> (let ((a 1))
>   (let ((b (+ a 1)))
>     (+ a b)))
3
```

- A `let*` expression is syntactic sugar for this idiom, in which bindings are evaluated in sequence:

```
> (let* ((a 1) (b (+ a 1)))
>   (+ a b))
3
```

Table 5.2 Binding Approaches Used in `let` and `let*` Expressions

`let`	bindings are added to the environment in	*parallel*.
`let*`	bindings are added to the environment in	*sequence*.

5.9.2 The `letrec` Expression

Use the `letrec` expression to make bindings visible *while* they are being created:

```
> (letrec ((length1 (lambda (l)
>   (cond
>     ((null? l) 0)
>     (else (+ 1 (length1 (cdr l)))))))))
>   (length1 '(a b c d)))
4
```

5.9.3 Using `let` and `letrec` to Define a Local Function
(1 of 2)

Design Guideline 5: Nest Local Functions

```
(define reverse1
  (letrec ((rev
            (lambda (lst rl)
              (cond
                ((null? lst) rl)
                (else (rev (cdr lst) (cons (car lst) rl)))))))
    (lambda (l)
      (cond
        ((null? l) '())
        (else (rev l '()))))))
```

Recursion from First Principles

- Both `let*` and `letrec` are syntactic sugar for `let`.
- Both `let*` and `letrec` are syntactic sugar for `lambda` (through `let`).
- Reduce the preceding `letrec` expression for `length1` to a `let` expression.
- Functions only know about what is passed to them, and what is in their local environment.
- We need `length1` to know about itself—so it can call itself recursively.
- Thus, we pass `length1` to `length1` itself!

```
> (let ((length1 (lambda (fun_length l)
>    (cond
>       ((null? l) 0)
>       (else (+ 1 (fun_length fun_length (cdr l))))))))
>   (length1 length1 '(a b c d)))
4
```

5.9.3 Using `let` and `letrec` to Define a Local Function
(2 of 2)

- Any function accepting more than one argument can be rewritten as an expression in λ-calculus by nesting λ-expressions.

- The function definition and invocation

```
> (lambda (a b)
     (+ a b))
#<procedure>

> ((lambda (a b)
     (+ a b)) 1 2)
3
```

can be rewritten as:

```
> (lambda (a)
     (lambda (b)
        (+ a b)))
#<procedure>

> ((lambda (a)
     ((lambda (b)
        (+ a b)) 2)) 1)
3
```

Figure 5.9 Graphical Depiction of the Foundational Nature of `lambda`

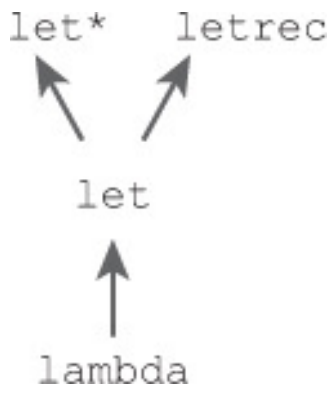


Table 5.3 Reducing `let` to `lambda` (All rows of each column are semantically equivalent.)

General Pattern	Instance of Pattern
(**let** ((sym1 val1) (sym2 val2) ··· (symn valn)) body)	(**let** ((a 1) (b 2)) (+ a b))
((lambda (sym1 sym2 ··· symn) body) val1 val2 ··· valn)	((lambda (a b) (+ a b)) 1 2)
(**let** ((sym1 val1)) (**let** ((sym2 val2)) ··· (**let** ((symn valn)) body)))	(**let** ((a 1)) (**let** ((b 2)) (+ a b))))
((lambda (sym1) ((lambda (sym2) ((lambda (···) ((lambda (symn) body) valn)) ···)) val2)) val1)	((lambda (a) ((lambda (b) (+ a b)) 2)) 1)

Table 5.4 Reducing `let*` to `lambda` (All rows of each column are semantically equivalent.)

General Pattern	Instance of Pattern
`(let* ((sym1 val1) (sym2 val2) ··· (symn valn))` `  body)`	`(let* ((a 1) (b (+ a 1)))` `  (+ a b))`

`(let ((sym1 val1))`
`  (let ((sym2 val2))`
`  ...`
`  (let ((symn valn))`
`  body)))`
	`(let ((a 1))`
	`  (let (b (+ a 1))`
	`  (+ a b)))`

`((lambda (sym1)`
`  ((lambda (sym2)`
`  ((lambda (···)`
`  ((lambda (symn)`
`  body) valn)) ···)) val2)) val1)`
	`((lambda (a)`
	`  ((lambda (b)`
	`  (+ a b)) (+ a 1))) 1)`

Table 5.5 Reducing `letrec` to `lambda` (All rows of each column are semantically equivalent.)

General Pattern	Instance of Pattern
(**let** ((f (lambda (sym1 sym2 ⋯ symn) ⋯(f val1 val2 ⋯ valn)⋯))) (f val1 val2 ⋯ valn))	(**let** ((length1 (lambda (l) (**cond** ((**null**? l) 0) (else (+ 1 (length1 (cdr l)))))))) (length1 '(a b c d)))
(letrec ((f (lambda (sym1 sym2 ⋯ symn) ⋯(f val11 val12 ⋯ valnm)⋯))) (f val1 val2 ⋯ valn))	(letrec ((length1 (lambda (l) (**cond** ((**null**? l) 0) (else (+ 1 (length1 (cdr l)))))))) (length1 '(a b c d)))
(**let** ((f (lambda (copy_of_f copy_of_f sym1 sym2 ⋯ symn) ⋯(copy_of_f copy_of_f val1 val2 ⋯ valn)⋯))) (f f val1 val2 ⋯ valn))	(**let** ((length1 (lambda (copy_of_length copy_of_length l) (**cond** ((**null**? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l)))))))) (length1 length1 '(a b c d)))
(**let** ((f (lambda (copy_of_f copy_of_f sym1 sym2 ⋯ symn) ⋯(copy_of_f copy_of_f val1 val2 ⋯ valn)⋯))) (f f val1 val2 ⋯ valn))	(**let** ((length1 (lambda (copy_of_f copy_of_f l) (**cond** ((**null**? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l)))))))) (length1 length1 l))
((lambda (f) (f f val1 val2 ⋯ valn)) (lambda (copy_of_f copy_of_f sym1 sym2 ⋯ symn) ⋯(copy_of_f copy_of_f val1 val2 ⋯ valn)⋯))	((lambda (length1) (length1 length1 l)) (lambda (copy_of_f copy_of_f l) (**cond** ((**null**? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l)))))))

Table 5.6 Semantics of `let`, `let*`, and `letrec`

	General Pattern	Instance of Pattern

`let`
 	
(parallel)
	`(let ((sym1 val1) (sym2 val2) ··· (symn valn))`
	`  sym1 and sym2 are only visible here in body)`
		`(let ((a 1) (b 2))`
		`  ; a and b are only visible here`
		`  (+ a b))`

`let*`
 	
(sequential)
	`; sym1 is visible here and beyond`
	`; sym2 is visible here and beyond`
	`(let* ((sym1 val1) (sym2 sym1) ··· (symn sym2))`
	`  sym1 sym2 ··· symn are visible here in body)`
		`; a is visible here and beyond`
		`(let* ((a 1) (b (+ a 1)))`
		`  ; a and b are visible here in body`
		`  (+ a b))`

`letrec`
 	
(recursive)
	`; f is visible here and in body`
	`(letrec ((f (lambda (sym1 sym2 ··· symn)`
	`  ··· (f val11 val12 ··· valnm) ···)))`
	`  (f val1 val2 ··· valn))`
		`; length1 is visible here and in body`
		`(letrec ((length1 (lambda (l)`
		`  (cond`
		`  ((null? l) 0)`
		`  (else (+ 1 (length1 (cdr l))))))))`
		`  (length1 '(a b c d)))`

5.10.1 More List Functions (`remove_first`)

The function `remove_first` removes the first occurrence of an atom `a` from a list of atoms `lat`:

```
1  (define remove_first
2    (lambda (a lat)
3      (cond
4        ((null? lat) '())
5        ((eqv? a (car lat)) (cdr lat))
6        (else (cons (car l) (remove_first a (cdr lat)))))))
```

Note typo: Last line should be (else (cons (car **lat**) (remove_first a (cdr lat)))))))

5.10.1 More List Functions (`remove_all`)

`remove_all` extends `remove_first` by removing *all* occurrences of an atom `a` from a list of atoms `lat` by returning `(remove_all (cdr lat))` in line 5 rather than `(cdr lat)`:

```
(define remove_all
  (lambda (a lat)
    (cond
      ((null? lat) '())
      ((eqv? a (car lat)) (remove_all a (cdr lat)))
      (else (cons (car lat) (remove_all a (cdr lat)))))))
```

5.10.1 More List Functions (`remove_all*`)

- Extend `remove_all` so that it removes *all* occurrences of an atom `a` from any S-list, not just a list of atoms.
- Using the third pattern in *Design Guideline 2: Specific Patterns of Recursion* results in:

```
1  (define remove_all*
2    (lambda (a l)
3      (cond
4        ((null? l) '())
5        ((atom? (car l))
6         (cond
7           ((eqv? a (car l)) (remove_all* a (cdr l)))
8           (else (cons (car l) (remove_all* a (cdr l))))))
9        (else (cons (remove_all* a (car l))
10                   (remove_all* a (cdr l)))))))
```

5.10.2 Eliminating Expression Recomputation

- Functional programs usually run more slowly than imperative programs because
  - (1) they are typically interpreted;
  - (2) they use recursion which is slower than iteration due to the overhead of the run-time stack; and
  - (3) the pass-by-value parameter-passing mechanism is inefficient.
- Barring interpretation and recursion, recomputing expressions only makes the program slower.
- *Design Guideline 4: Name Recomputed Subexpressions*

New Version of `remove_all*`

```
1   (define remove_all*
2     (lambda (a l)
3       (cond
4         ((null? l) '())
5         (else (let ((head (car l)))
6                 (cond
7                   ((atom? head)
8                    (cond
9                      ((eqv? a head) (remove_all* a (cdr l)))
10                     (else (cons head (remove_all* a (cdr l))))))
11                  (else (cons (remove_all* a head)
12                              (remove_all* a (cdr l)))))))))
```

Notice that binding the result of the evaluation of the expression `(cdr l)` to the mnemonic tail, while improving readability, does not actually improve performance. While the expression `(cdr l)` appears more than once in this definition (lines 9, 10, and 12), it is computed only once per function invocation.

Newer Version of `remove_all*`

```
1  (define remove_all*
2    (lambda (a l)
3      (letrec ((remove_all_helper*
4                (lambda (l)
5                  (cond
6                    ((null? l) '())
7                    (else (let ((head (car l)))
8                            (cond
9                              ((atom? head)
10                              (cond
11                               ((eqv? a head)
12                                  (remove_all_helper* (cdr l)))
13                               (else
14                                  (cons head
15                                     (remove_all_helper*
16                                        (cdr l))))))
17                             (else
18                                (cons
19                                   (remove_all_helper* head)
20                                   (remove_all_helper*
21                                      (cdr l))))))))))
22       (remove_all_helper* l))))
```

- Passing an argument with the same value across multiple recursive calls is inefficient and unnecessary.
- Every time `remove_all*` is called, it is passed the atom `a`, which never changes.
- *Design Guideline 6: Factor out Constant Parameters*
  - This version of `remove_all*` works because within the scope of `remove_all*` (lines 3–22), the parameter `a` is visible.
  - We can think of it as global just within that block of code.
  - Since it is visible in that range, it need not be passed to any function defined (either with a `let`, `let*`, or `letrec` expression) in that block, since any function defined within that scope already has access to it.
  - We defined a nested function `remove_all_helper*` that accepts only a list `l` as an argument.
  - The parameter `a` is not passed to `remove_all_helper*` in the calls to it on lines 12, 15, and 18–20 (only a smaller list is passed), even though within the body of `remove_all_helper*` the parameter a (from the function `remove_all*)` is referenced.

5.10.3 Avoiding Repassing Constant Arguments Across Recursive Calls (1 of 2)

```
1  ;; style used to define remove_all*
2  (lambda (a)
3
4     ;; body of lambda expression
5     (letrec ((f (lambda (<parameter list>) ...)) ...)
6
7        ;; body of letrec expression
8        ;; parameter a is accessible here
9
10          ;; call to f
11       ... (f ...) ...))
12
13 ;; style used to define reverse1
14 (letrec ((f (lambda (<parameter list>)
15
16                ;; parameter a is not accessible here
17
18                   ;; call to f
19                ... (f ...) ...)))
20
21    ;; body of letrec expression
22    (lambda (a)
23
24       ;; body of lambda expression
25       ;; parameter a is accessible here
26
27          ;; call to f
28       ... (f ...) ...))
```

- If the nested function f must access one or more of the parameters (i.e., *Design Guideline 6*), which is the case with `remove_all*`, then the style illustrated in lines 1–11 must be used.

- Conversely, if one or more of the parameters to the outer function should be hidden from the nested function, which is the case with `reverse1`, then the style used on lines 13–28 must be used.

5.10.3 Avoiding Repassing Constant Arguments Across Recursive Calls (2 of 2)

Consider the following two `letrec` expressions, both of which yield the same result:

```
1  > (letrec ((length1 (lambda (l)
2  >                    (cond
3  >                      ((null? l) 0)
4  >                      (else (+ 1 (length1 (cdr l))))))))
5  >   (length1 '(a b c d e)))
6  5
7
8  > ((letrec ((length1 (lambda (l)
9  >                     (cond
10 >                       ((null? l) 0)
11 >                       (else (+ 1 (length1 (cdr l))))))))
12 >    length1) '(1 2 3 4 5))
13 5
```

- They are functionally equivalent.
- The first expression (lines 1–5) calls the local function `length1` in the body of the `letrec` (line 5).
- The second expression (lines 8–12) first returns the local function `length1` in the body of the `letrec` (line 12) and then calls it—notice the double parentheses to the left of `letrec` on line 8.
- The first expression uses binding to invoke the function `length1`.
- The second uses binding to return the function `length1`.

Outline

- 5.1 Chapter Objectives
- 5.2 Introduction to Functional Programming
- 5.3 Lisp
- 5.4 Scheme
- 5.5 `cons` Cells: Building Blocks of Dynamic Memory Structures
- 5.6 Functions on Lists
- 5.7 Constructing Additional Data Structures
- 5.8 Scheme Predicates as Recursive-Descent Parsers
- 5.9 Local Binding: `let`, `let*`, and `letrec`
- 5.10 Advanced Techniques
- **5.11 Languages and Software Engineering**
- **5.12 Layers of Functional Programming**
- **5.13 Concurrency**
- **5.15 Thematic Takeaways**

5.11 Languages and Software Engineering

- Programming languages that support
  - the construction of abstractions, and
  - ease of program modification

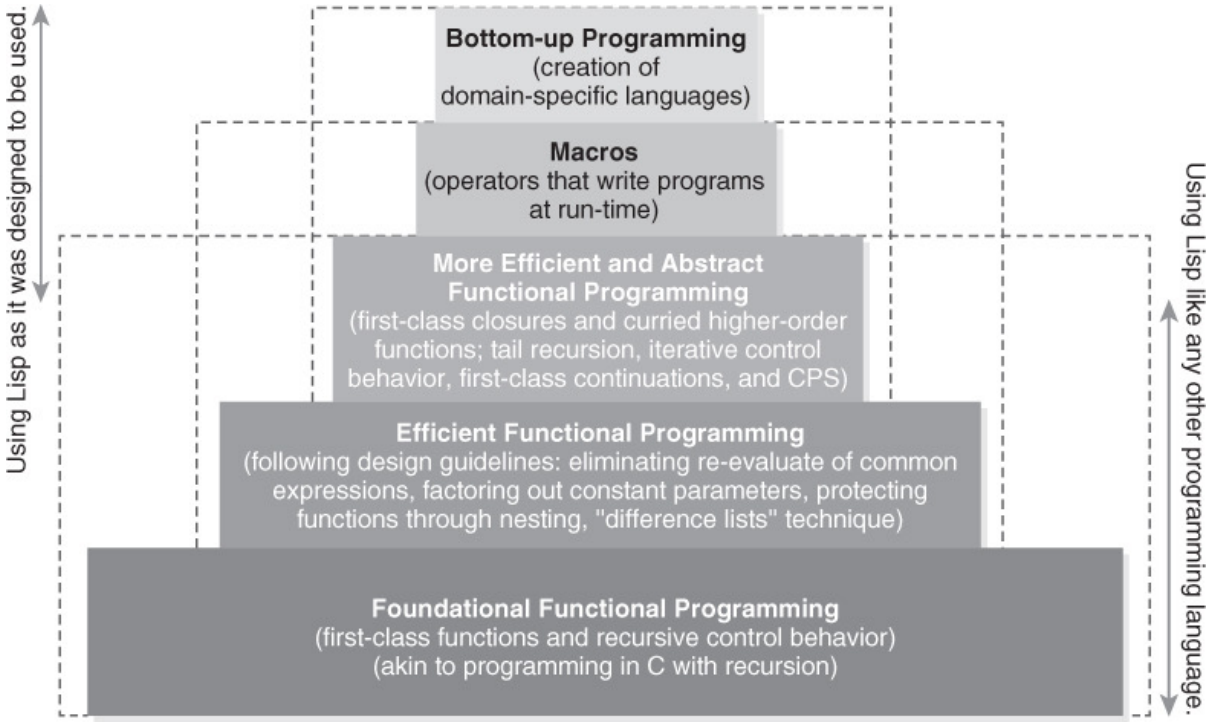
also support

  - ongoing development of a malleable program design, and
  - the evolution of a prototype into product.

5.11.2 Language Flexibility Supports Program Modification

- A functional style of programming in a flexible language supports ease of program modification.
- We not only organically constructed the functions and programs here, but also refined them repeatedly with ease.
- The interactive *read-eval-print* loop used in interpreted languages fosters rapid program development, modification, testing, and debugging.
- In contrast, programming in a compiled language such as C++ involves the use of a *program-compile-debug-recompile* loop.
- The ability to make more global changes to a program easily is especially important—evolving specifications is a reality.
- A language should facilitate, and not handicap, an (inevitable) evolving design and redesign.

Figure 5.10 Layers of Functional Programming



5.15 Thematic Takeaways

- Functional programming unites beauty with utility.
- The λ-calculus, and the three grammar rules that constitute it, are sufficiently powerful.
- An important theme in a course on data structures and algorithms is that data structures and algorithms are natural reflections of each other.
- Powerful programming abstractions can be constructed in a few lines of Scheme code.
- Recursion can be built into any programming language with support for first-class anonymous functions.
- Speed of development is now a more important criterion in the creation of software than it has been historically

Table 5.7 Functional Programming Design Guidelines

1. **General Pattern of Recursion.** Solve the problem for the smallest instance of the problem (called the *base case*; e.g., $n = 0$ for $n!$, which is $n^0 = 1$). Assume the penultimate [i.e., $(n-1)$th, e.g., $(n-1)!$] instance of the problem is solved and demonstrate how you can extend that solution to the nth instance of the problem [e.g., multiply it by n; i.e., $n*(n-1)!$].
2. **Specific Patterns of Recursion.** When recurring on a list of atoms, `lat`, the base case is an empty list [i.e., `(null? lat)`] and the recursive step is handled in the `else` clause. Similarly, when recurring on a number, `n`, the base case is, typically, $n = 0$ [i.e., `(zero? n)`] and the recursive step is handled in the `else` clause.

   When recurring on a list of S-expressions, `l`, the base case is an empty list [i.e., `(null? l)`] and the recursive step involves two cases: (1) where the `car` of the list is an atom [i.e., `(atom? (car l))`] and (2) where the `car` of the list is itself a list (handled in the `else` clause, or vice versa).
3. **Efficient List Construction.** Use `cons` to build lists.
4. **Name Recomputed Subexpressions.** Use `(let (···) ···)` to name the values of repeated expressions in a function definition if they may be evaluated more than once for one and the same use of the function. Moreover, use `(let (···) ···)` to name the values of the expressions in the body of the `let` that are reevaluated every time a function is used.
5. **Nest Local Functions.** Use `(letrec (···) ···)` to hide and protect recursive functions and `(let (···) ···)` or `(let* (···) ···)` to hide and protect non-recursive functions. Nest a `lambda` expression within a `letrec` (or `let` or `let*`) expression:

```
(define f
   (letrec ((g (lambda (···) ···))) ; or let or let*
      (lambda (···) ···)))
```

6. **Factor out Constant Parameters.** Use `letrec` to factor out parameters whose arguments are constant (i.e., never change) across successive recursive applications. Nest a `letrec` (or `let` or `let*`) expression within a `lambda` expression:

```
(define member1
   (lambda (a lat)
      (letrec ((M (lambda (lat) ...)))
         (M lat))))
```

7. **Difference Lists Technique.** Use an additional argument representing the return value of the function that is built up across the successive recursive applications of the function when that information would otherwise be lost across successive recursive calls.
8. **Correctness First, Simplification Second.** Simplify a function or program, by nesting functions, naming recomputed values, and factoring out constant arguments, only after the function or program is thoroughly tested and correct.

Additional Functional Programming Design Guidelines

- Scott Wlaschin suggests the following functional design principles:
  - **Composition Everywhere!** Use functions as building blocks, using the output of one function as input to the next. Design them so you can snap them together like Legos.
  - **Strive for Totality!** Your function should do something 'sensible' for any valid input. If your numeric function crashes if passed a zero, either modify it to do something sensible (use Maybe or Either, perhaps), or document that the parameter must be nonzero—it's up to the caller to check that it's valid.
  - **Don't Repeat Yourself!** If different functions are doing almost-but-not-quite the same thing, factor out the common part, and use subfunctions or helper functions as needed.
  - **Parameterize the things!** Generalize your functions by adding parameters. A sorting function that can take in any comparison method is more useful than one locked in to using <.
  - **The Hollywood Principle (Don't call us, we'll call you).** Provide callback functions so you can use partial application to customize a generic function.