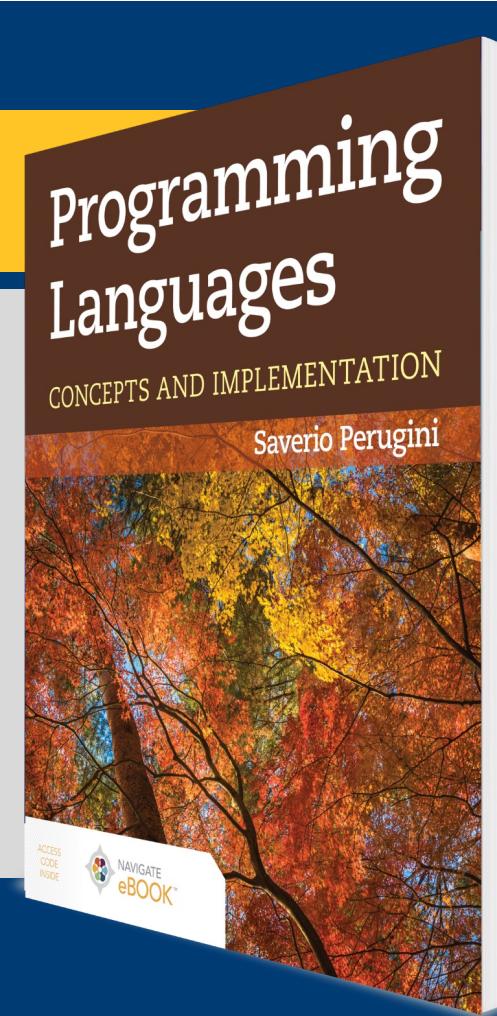


CHAPTER 5

Functional Programming in Scheme



Copyright © 2023 by Jones & Bartlett Learning, LLC an Ascend Learning Company. www.jblearning.com.

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

$\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle$
(a symbol)

$\langle \text{expression} \rangle ::= (\lambda \langle \text{identifier} \rangle \langle \text{expression} \rangle)$
(a function definition)

$\langle \text{expression} \rangle ::= (\langle \text{expression} \rangle \langle \text{expression} \rangle)$
(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`)

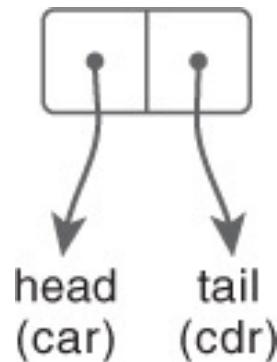


Figure 5.2' $(a \ b) = ' (a . (b))$

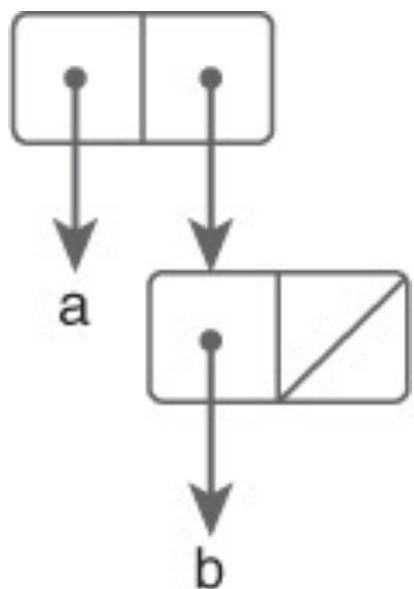


Figure 5.3 ' $(a \ b \ c)$ = ' $(a \ . \ (b \ c))$ =
' $(a \ . \ (b \ . \ (c))$)'

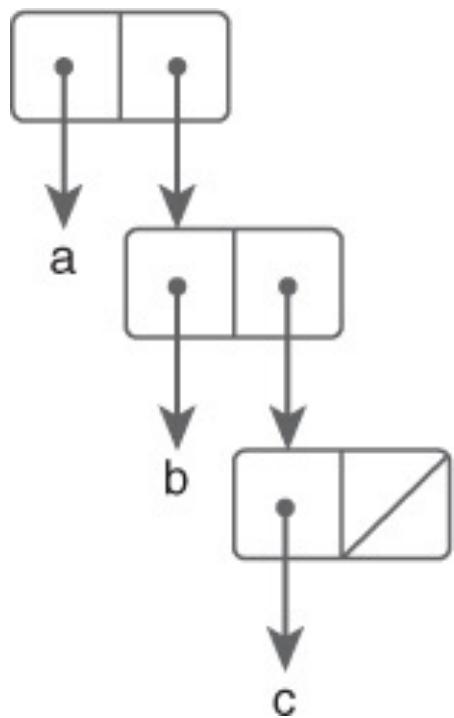


Figure 5.4 ' (a . b)

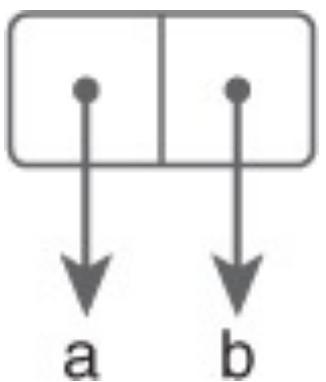


Figure 5.5

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

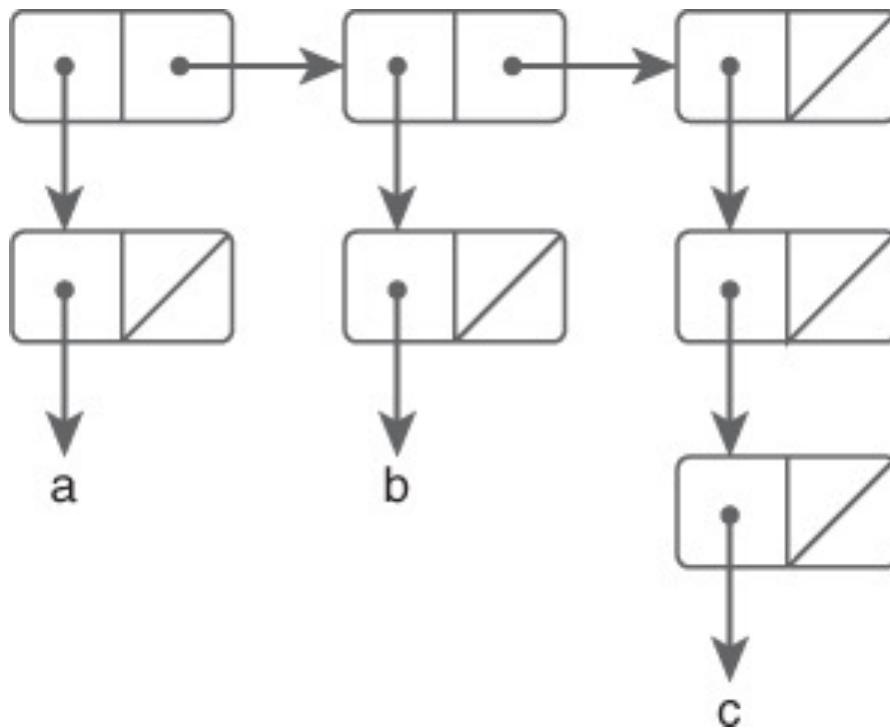


Figure 5.6 ' (((a) b) c)

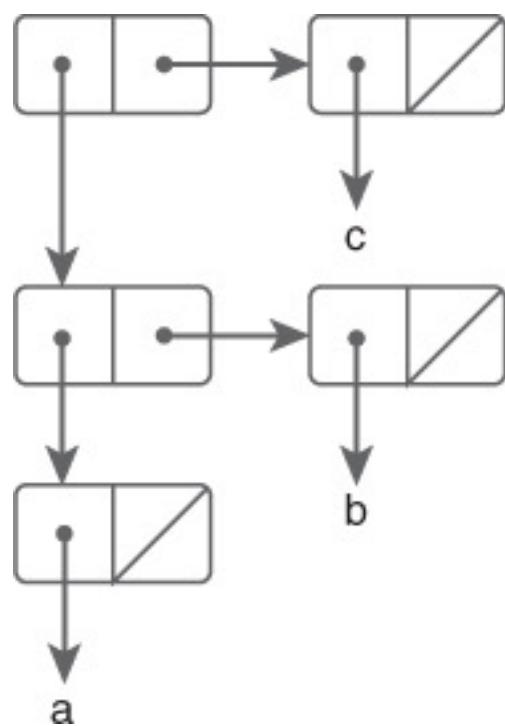


Figure 5.7

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

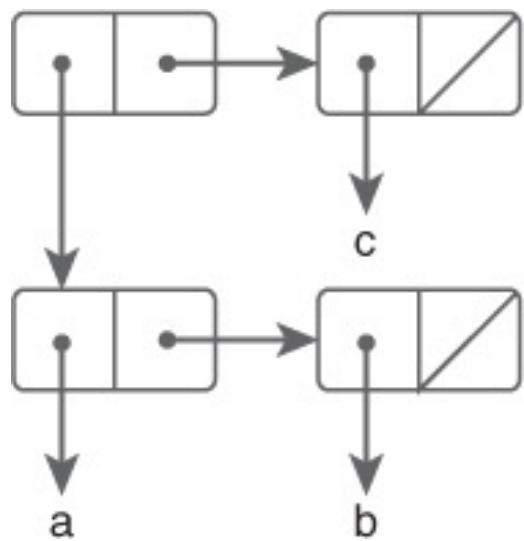
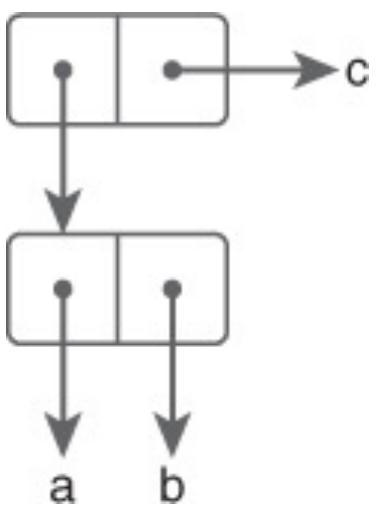


Figure 5.8 ' $((a \ . \ b) \ . \ c)$



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:
 - 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.

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

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 (reversel '(b c)) (cons a '()))
2  (append (reversel '(b c)) '(a))
3  (append (append (reversel '(c)) (cons b '())) '(a))
4  (append (append (reversel '(c)) '(b)) '(a))
5  ;; base case
6  (append (append (append (reversel '()) (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 `lets`:

```
> (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

can be rewritten as:

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

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



```
> ((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

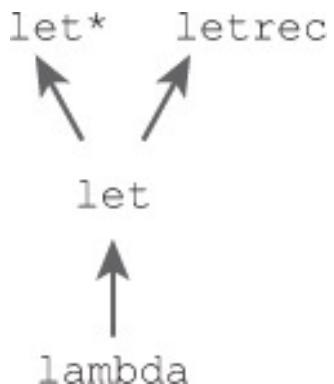


Table 5.3 Reducing let to lambda
(All rows of each column are semantically equivalent.)

General Pattern	Instance of Pattern
<code>(let ((sym1 val1) (sym2 val2) ... (symn valn)) body)</code>	<code>(let ((a 1) (b 2)) (+ a b))</code>
<code>((lambda (sym1 sym2 ... symn) body) val1 val2 ... valn)</code>	<code>((lambda (a b) (+ a b)) 1 2)</code>
<code>(let ((sym1 val1)) (let ((sym2 val2)) ... (let ((symn valn)) body)))</code>	<code>(let ((a 1)) (let ((b 2)) (+ a b))))</code>
<code>((lambda (sym1) ((lambda (sym2) ((lambda (...) ((lambda (symn) body) valn)) ...)) val2)) val1)</code>	<code>((lambda (a) ((lambda (b) (+ a b)) 2)) 1)</code>

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

General Pattern	Instance of Pattern
<code>(let* ((sym1 val1) (sym2 val2) ... (symn valn)) body)</code>	<code>(let* ((a 1) (b (+ a 1))) (+ a b))</code>
<code>(let ((sym1 val1)) (let ((sym2 val2)) ... (let ((symn valn)) body)))</code>	<code>(let ((a 1)) (let (b (+ a 1)) (+ a b)))</code>
<code>((lambda (sym1) ((lambda (sym2) ((lambda (...) ((lambda (symn) body) valn)) ...)) val2)) val1)</code>	<code>((lambda (a) ((lambda (b) (+ a b)) (+ a 1))) 1)</code>

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

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

Table 5.6 Semantics of let, let*, and letrec

	General Pattern	Instance of Pattern
let (parallel)	<pre>(let ((sym1 val1) (sym2 val2) ... (symn valn)) sym1 and sym2 are only visible here in body)</pre>	<pre>(let ((a 1) (b 2)) ; a and b are only visible here (+ a b))</pre>
let* (sequential)	<p style="text-align: center;">; sym1 is visible here and beyond ; sym2 is visible here and beyond</p> <pre>(let* ((sym1 val1) (sym2 sym1) ... (symn sym2)) sym1 sym2 ... symn are visible here in body)</pre>	<p style="text-align: center;">; a is visible here and beyond ; a and b are visible here in body</p> <pre>(let* ((a 1) (b (+ a 1))) ; a and b are visible here in body (+ a b))</pre>
letrec (recursive)	<p style="text-align: center;">; f is visible here and in body</p> <pre>(letrec ((f (lambda (sym1 sym2 ... symn) ... (f val1 val2 ... valn) ...))) (f val1 val2 ... valn))</pre>	<p style="text-align: center;">; length1 is visible here and in body</p> <pre>(letrec ((length1 (lambda (l) (cond ((null? l) 0) (else (+ 1 (length1 (cdr l)))))))) (length1 '(a b c d)))</pre>

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 lat) (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

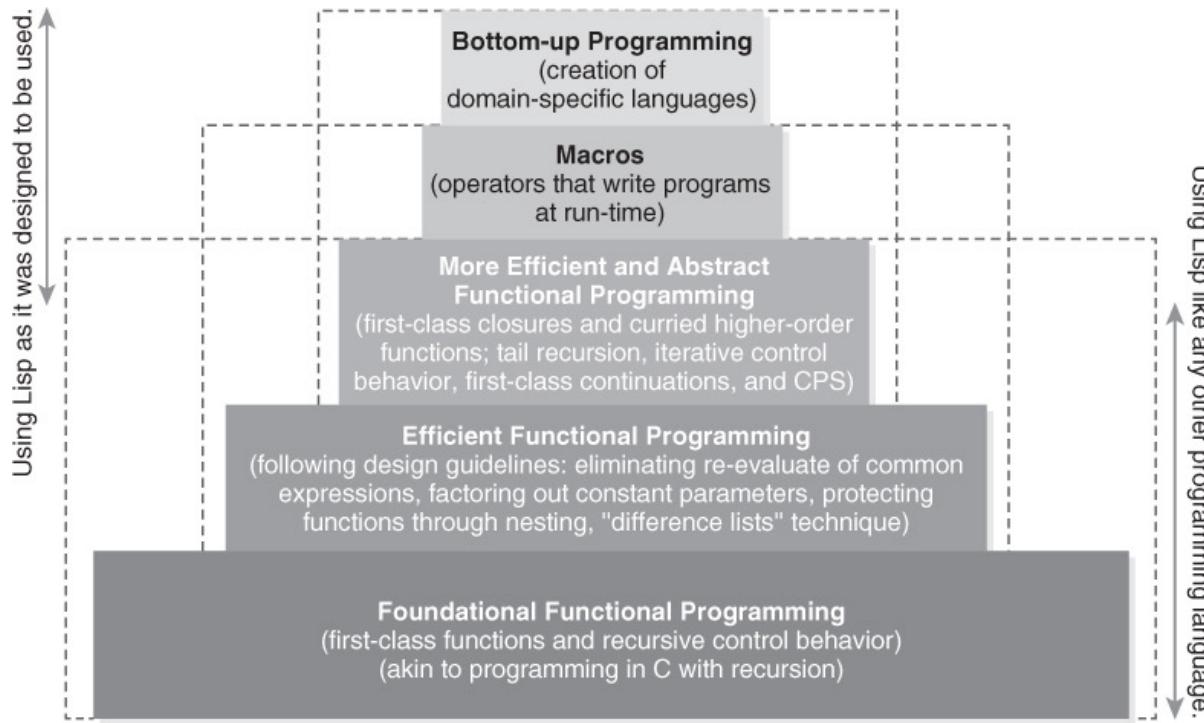
also support

- 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 n th 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:

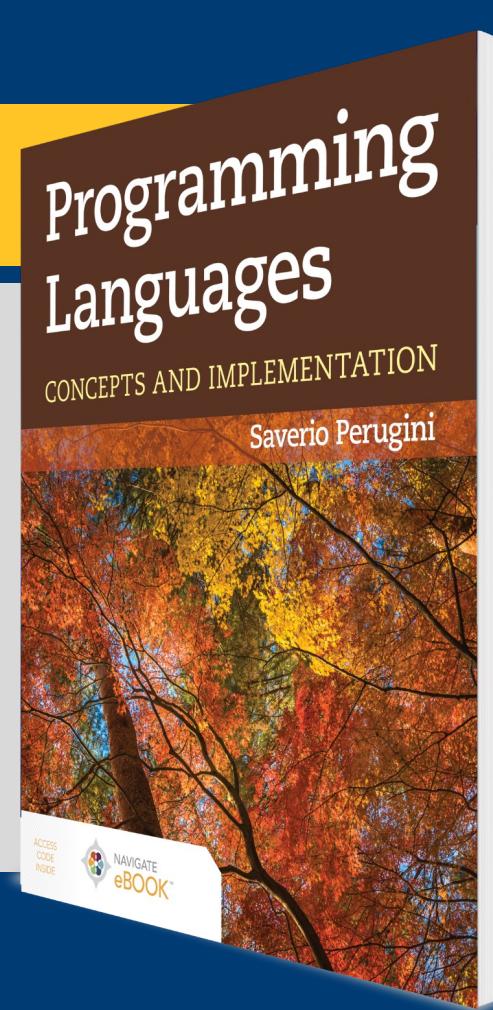
```
(define memberl
  (lambda (a lat)
    (letrec ((M (lambda (lat) ...)))
      (M lat))))
```
7. **Difference Lists Technique.** Use an additional argument representing the return value of the function that is built up across the successive recursive applications of the function when that information 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.

CHAPTER 6

Binding and Scope



Copyright © 2023 by Jones & Bartlett Learning, LLC an Ascend Learning Company. www.jblearning.com.

Chapter 6: Binding and Scope

A rose by any other name would smell as sweet.

— William Shakespeare

6.1 Chapter Objectives

- Describe *first-class closures*.
- Understand the meaning of the adjectives *static* and *dynamic* in the context of programming languages.
- Discuss *scope* as a type of binding from variable reference to declaration.
- Differentiate between *static* and *dynamic scoping*.
- Discuss the relationship between the lexical layout of a program and the representation and structure of a referencing environment for that program.
- Define *lexical addressing* and consider how it obviates the need for identifiers in a program.
- Discuss program translation as a means of improving the efficiency of execution.
- Learn how to resolve references in functions to parts of the program not currently executing (i.e., *the FUNARG problem*).
- Understand the difference between *deep*, *shallow*, and *ad hoc binding* in passing first-class functions as arguments to procedures.

6.2.1 What Is a Closure?

- A *closure* is a function that remembers the lexical environment in which it was created.
- A closure can be thought of as a pair of pointers:
 - One to a block of code (defining the function)
 - One to an environment (in which function was created).
- The bindings in the environment are used to evaluate the expressions in the code.
- A closure encapsulates data and operations and, thus, bears a resemblance to an object as used in object-oriented programming.
- Closures are powerful constructs in functional programming, and an essential element in the study of binding and scope.

6.2.2 Static Vis-à-Vis Dynamic Bindings

Static bindings are *fixed* before run-time. Example: `int a;`
Dynamic bindings are *changeable* during run-time. Example: `a = 1;`

Table 6.1 Static Vis-à-Vis Dynamic Bindings

6.3 Introduction (1 of 2)

- Variables appear as either *references* or *declarations*.
- The value named by a variable is called its *denotation*.

```
1 > ((lambda (x)
2 >   (+ 7
3 >     ((lambda (a b)
4 >       (+ a b x)) 1 2))) 5)
5 15
```

- The denotations of `x`, `a`, and `b` are 5, 1, and 2, respectively.
- The `x` on line 1 and the `a` and `b` on line 3 are declarations, while the `a`, `b`, and `x` on line 4 are references.
- A reference to a variable (e.g., the `a` on line 4) is bound to a declaration of a variable (e.g., the `a` on line 3).

6.3 Introduction (2 of 2)

- Declarations have *limited scope*.
- The *scope* of a variable declaration in a program is the region of that program (i.e., a range of lines of code) within which references to that variable refer to the declaration (Friedman, Wand, and Haynes 2001).
- The scope of the declaration of `a` in the preceding example is line 4—the same as for `b`. The scope of the declaration of `x` is lines 2–4.
- The *scope rules* of a programming language indicate to which declaration a reference is bound.
- Languages where that binding can be determined by examining the text of the program *before run-time* use *static scoping*.
- Languages where the determination of that binding requires information available *at runtime* use *dynamic scoping*.

Table 6.2 Static Scoping Vis-à-Vis Dynamic Scoping

<i>Static scoping</i>	A reference is bound to a declaration <i>before</i> run-time, e.g., based on the spatial relationship of nested program blocks to each other, i.e., <i>lexical scoping</i> .
<i>Dynamic scoping</i>	A reference is bound to a declaration <i>during</i> run-time, e.g., based on the calling sequences of procedures on run-time call stack.

6.4.1 Lexical Scoping

```
1 > ((lambda (x)
2 >   (+ 7
3 >     ((lambda (a b)
4 >       (+ a
5 >         ((lambda (c a)
6 >           (+ a b x)) 3 4))) 1 2))) 5)
7 19
```

- This entire expression (lines 1–6) is a block, which contains a nested block (lines 2–6), which itself contains another block (lines 3–6), and so on.
- Lines 5–6 are the innermost block and lines 1–6 constitute the outermost block; lines 3–6 make up an intervening block.

Lexical Scoping Procedure

- Start with the innermost block of the expression containing the reference and search within it for its declaration.
- If it is not found there, search the next block enclosing the one just searched. If the declaration is not found there, continue searching in this innermost-to-outermost fashion until a declaration is found.
- After searching the outermost block, if a declaration is not found, the variable reference is free (as opposed to bound).
- Due to the scope rules of Scheme and the lexical layout of the program that it relies upon, reveals that the reference to `x` in line 6 of the example Scheme expression previously is bound to the declaration of `x` on line 1.
- Neither the scope rule nor the procedure yields the scope of a declaration.

Shadow, Scope Hole, and Visibility

- The scope of a declaration is the region of the program within which references refer to the declaration. In this example, the scope of the declaration of `x` is lines 2–6.
- The scope of the declaration of `a` on line 3, by contrast, is lines 4–5 rather than lines 4–6, because the inner declaration of `a` on line 5 *shadows* the outer declaration of `a` on line 3.
- The inner declaration of `a` on line 5 creates a *scope hole* on line 6, so that the scope of the declaration of `a` on line 3 is lines 4–5 and not lines 4–6.
- The *visibility* of a declaration in a program constitutes the regions of that program where references are bound to that declaration—this is the definition of scope given and used previously.
- Scope refers to the entire block of the program where the declaration is applicable.
- Thus, the scope of a declaration includes scope holes since the bindings still exist but are hidden.
- The visibility of a declaration is a subset of the scope of that declaration and, therefore, is bounded by the scope.

**Nesting of blocks progresses from left to right.
On line 2, the declaration of a on line 3 is not in scope:**

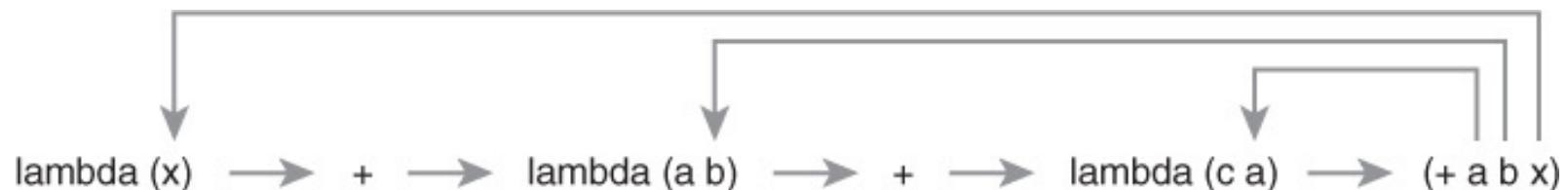
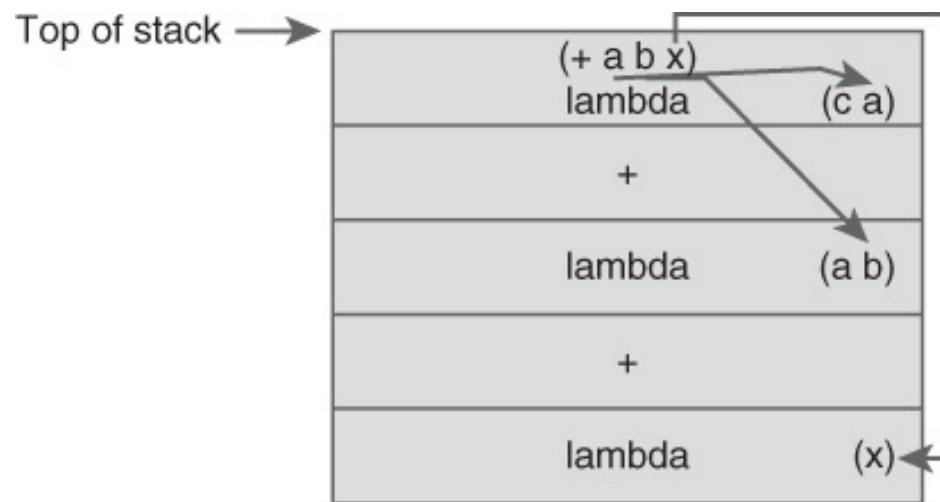


Figure 6.1 Run-time Call Stack at the Time the Expression $(+ a b x)$ Is Evaluated



The arrows indicate to which declarations the references to a , b , and x are bound.

6.4.1 Lexical Scoping (Local Vis-à-Vis Nonlocal References) (1 of 2)

- A reference can either be local or nonlocal.
- A *local reference* is bound to a declaration in the set of declarations (e.g., the formal parameter list) associated with the innermost block in which that reference is contained.
- Sometimes that block is called the *local block*.
- All of the nested blocks enclosing the innermost block containing the reference are sometimes referred to as *ancestor blocks* of that block.
- In a lexically scoped language, we search both the local and ancestor blocks to find the declaration to which a reference is bound.

6.4.1 Lexical Scoping (Local Vis-à-Vis Nonlocal References) (2 of 2)

- *We must determine the declaration to which a reference is bound so that we can determine the value bound to the identifier at that reference so that we can evaluate the expression containing that reference.*
- The concept of an *environment*, which is a core element of any interpreter:
 - A set or mapping of name–value pairs that associates variable names (or symbols) with their current bindings

scope(*<declaration>*) = <*a set of program points*>
referencing environment(*<a program point>*) = <*a set of variable bindings*>

6.5 Lexical Addressing (1 of 4)

- Identifiers are necessary for writing programs, but unnecessary for executing them.
- Assume we number the innermost-to-outermost blocks of an expression from 0 to n.
- *Lexical depth* is an integer representing a block with respect to all of the nested blocks it contains.
- Assume that we number each formal parameter in the declaration list associated with each block from 0 to m.
- The *declaration position* of a particular identifier is an integer representing the position in the list of identifiers of a lambda expression of that identifier.

6.5 Lexical Addressing (2 of 4)

```
; partially converted to lexical addresses,  
; where references are replaced with  
; (identifier, depth, position) triples  
> ((lambda (x)  
>   (+ 7  
>     ((lambda (a b)  
>       (+ (a : 1 0)  
>         ((lambda (c a)  
>           (+ (a : 0 1) (b : 1 1) (x : 2 0))) 3 4))) 1 2))) 5)  
19
```

- Given only a lexical address (i.e., lexical depth and declaration position), we can (efficiently) lookup the binding associated with the identifier in a reference.
- We can purge the identifiers from each lexical address.

6.5 Lexical Addressing (3 of 4)

```
;; fully converted to lexical addresses,  
;; where identifiers are completely purged,  
;; references are replaced with (depth, position) pairs.  
> ((lambda (x)  
>   (+ 7  
>     ((lambda (a b)  
>       (+ (1 0)  
>         ((lambda (c a)  
>           (+ (0 1) (1 1) (2 0))) 3 4))) 1 2))) 5)  
19
```

6.5 Lexical Addressing (4 of 4)

The formal parameter lists following each lambda are also unnecessary and, therefore, can be replaced with their length:

```
; fully converted to lexical addresses,  
; where identifiers are completely purged,  
; references are replaced with (depth, position) pairs, and  
; formal parameter lists are replaced by their length.  
> ((lambda 1  
>   (+ 7  
>     ((lambda 2  
>       (+ (1 0)  
>         ((lambda 2  
>           (+ (0 1) (1 1) (2 0))) 3 4))) 1 2))) 5)  
19
```

Table 6.3 Lexical Depth and Position in a Referencing Environment

depth:	0	1	0	1	2	
position:	0	1	0	1	0	
environment:	(((c 3) (a 4))	((a 1) (b 2))	((x 5)))	

How does this map to a list-based data structure?

6.6 Free or Bound Variables (1 of 2)

- A variable v occurs *free* in an expression e if and only if there is a reference to v within e that is not bound by any declaration of v within e .
- A variable v occurs *bound* in an expression e if and only if there is a reference to v within e that is bound by some declaration of v in e .
- In the expression `((lambda (x) x) y)`
 - The x in the body of the `lambda` expression occurs bound to the declaration of x in the formal parameter list.
 - The argument y occurs free because it is unbound by any declaration in this expression.

6.6 Free or Bound Variables (2 of 2)

- The semantics of an expression without any free variables is fixed.
- Consider the identity function `(lambda (x) x)`. It has no free variables and its meaning is always fixed as “return the value that is passed to it.”
- The semantics of the following expression, which also has no free variables, is always:

```
(lambda (x)
        (lambda (f)
            (f x))))
```

“a function that accepts a value `x` and returns ‘a function that accepts a function `f` and returns the result of applying the function `f` to the value `x`.’”

Outline

- 6.1 Chapter Objectives
- 6.2 Preliminaries
- 6.3 Introduction
- 6.4 Static Scoping
- 6.5 Lexical Addressing
- 6.6 Free or Bound Variables
- **6.7 Dynamic Scoping**
- **6.8 Comparison of Static and Dynamic Scoping**
- **6.9 Mixing Lexically and Dynamically Scoped Variables**
- 6.10 The FUNARG Problem
- 6.11 Deep, Shallow, and Ad Hoc Binding
- 6.12 Thematic Takeaways

6.7 Dynamic Scoping (1 of 2)

```
1 ((lambda (x y)
2   (let ((proc2 (lambda () (cons x (cons y (cons (+ x y) '()))))))
3     (let ((proc1 (lambda (x y) (cons x (proc2)))))
4       (cond
5         ((zero? (read)) (proc1 5 20))
6         (else (proc2))))))
7 10 11)
```

- We see nonlocal references to `x` and `y` in the definition of `proc2` on line 2, which does not provide declarations for `x` and `y`.
- To resolve those references so that we can evaluate the `cons` expression, we must determine to which declarations the references to `x` and `y` are bound.

6.7 Dynamic Scoping (2 of 2)

- While static scoping involves a search of the program text, dynamic scoping involves a search of the run-time call stack.
- Concept of *static call graph*
 - Indicates which procedures have access to each other (Figure 6.2)
- Concept of the *call chain* (or *dynamic call graph*) of an expression
 - Depicts the series of functions called by the program as they would appear on the run-time call stack

Figure 6.2 Static Call Graph of the Program Used to Illustrate Dynamic Scoping in Section 6.7

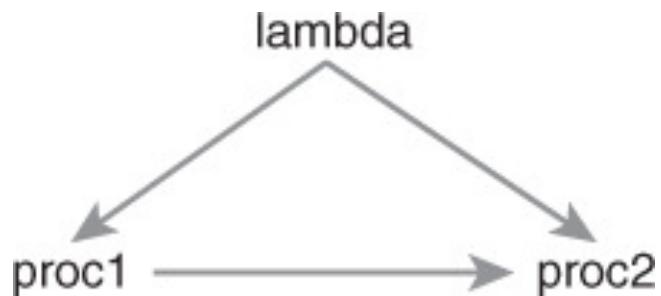


Figure 6.3 The Two Run-Time Call Stacks Possible from the Program Used to Illustrate Dynamic Scoping in Section 6.7

- The stack on the left corresponds to call chain $\text{lambda}^{(x\ y)} \rightarrow \text{proc1}^{(x\ y)} \rightarrow \text{proc2}$.
- The stack on the right corresponds to call chain $\text{lambda}^{(x\ y)} \rightarrow \text{proc2}$.

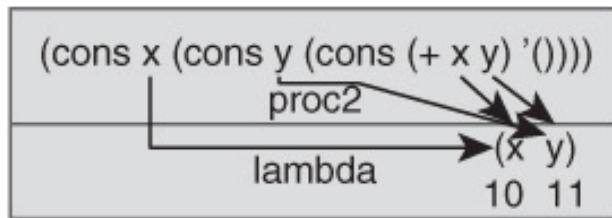
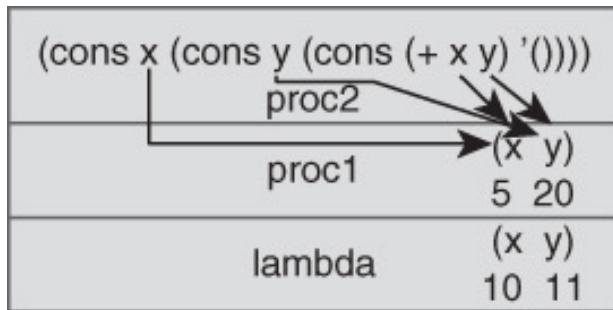


Table 6.5 Advantages and Disadvantages of Static and Dynamic Scoping

Scoping	Advantages	Disadvantages
Static	improved readability; easier program comprehension; predictability; type checking/validation	larger scopes than necessary; can lead to several globals; can lead to all functions at the same level; harder to implement in languages with nested and first-class procedures
Dynamic	flexibility	reduced readability; reduced reliability; type checking/validation; can be less efficient to implement; difficult to debug; no locality of access; no way to protect local variables; easier to implement in languages with nested and first-class procedures

List 6.2 A Perl Program Demonstrating Dynamic Scoping

- The output of this program is:

```
Before the call to proc1 --- l: 10, d: 11
Inside the call to proc1 --- l: 5, d: 20
Inside the call to proc2 --- l: 10, d: 20
After the call to proc2 --- l: 5, d: 20
After the call to proc1 --- l: 11, d: 12
```

- We need not run the program to determine to which declaration the reference to `d` on line 37 is bound.
 - We can determine the call chain of the procedures, before run-time, by examining the text of the program.
- In most programs we cannot determine the call chain of procedure before run-time—primarily due to run-time input.

**Figure 6.4 Depiction of Run-Time Stack at Call to print
on Line 37 of Listing 6.2**

procedure names	activation records	variables
proc2		
	20	d
proc1	5	l
	11	d
main	10	l

Listing 6.3 A Perl Program, Whose Run-Time Call Chain Depends on Its Input, Demonstrating Dynamic Scoping

- The call chain depends on program input.
- If the input is 5, then the call chain is

main → proc1 → proc2

and the output is the same as the output for Listing 6.2.

- Otherwise, the call chain is

main → proc2

and the output is:

```
Before the call to proc1 --- l: 10, d: 11
Inside the call to proc2 --- l: 10, d: 11
After the call to proc1 --- l: 11, d: 11
```

Outline

- 6.1 Chapter Objectives
- 6.2 Preliminaries
- 6.3 Introduction
- 6.4 Static Scoping
- 6.5 Lexical Addressing
- 6.6 Free or Bound Variables
- 6.7 Dynamic Scoping
- 6.8 Comparison of Static and Dynamic Scoping
- 6.9 Mixing Lexically and Dynamically Scoped Variables
- **6.10 The FUNARG Problem**
- **6.11 Deep, Shallow, and Ad Hoc Binding**
- **6.12 Thematic Takeaways**

6.10 The FUNARG Problem

- With first-class procedures in the discussion of scope, resolving nonlocal references suddenly becomes more complex.
- The question is: To which declaration does a reference in the body of a passed or returned function bind?
- The difficulty arises when a nested function makes a nonlocal reference to an identifier in the environment in which the function is defined, but not invoked.
- Must determine the environment in which to resolve that reference so that we can evaluate the body of the function
- *The problem is that the environment in which the function is created may not be on the stack.*
- There are two instances of the FUNARG problem:
 - The downward FUNARG problem
 - The upward FUNARG problem

6.10.1 The Downward FUNARG Problem

- Involves passing a function (called a *downward FUNARG*) to another function

```
1 ((lambda (x y)
2   ((lambda (proc2)
3     ((lambda (proc1) (proc1 5 20))
4      (lambda (x y) (cons x (proc2))))))
5     (lambda () (cons x (cons y (cons (+ x y) '()))))))
6 10 11)
```

- The functions passed on lines 4 and 5, and accessed through the parameters `proc1` and `proc2`, respectively, are downward FUNARGS.

6.10.2 The Upward FUNARG Problem (1 of 4)

- Involves returning a function (called an *upward* FUNARG) from a function, rather than passing functions to a function
- Classical example of an upward FUNARG in Scheme:

```
1  (define add_x
2      (lambda (x)
3          (lambda (y)
4              (+ x y))))))
5
6  (define main
7      (lambda ()
8          (let ((add5 (add_x 5))
9                (add6 (add_x 6)))
10             (cons (add5 2) (cons (add6 2) '())))))
11  (main))
```

6.10.2 The Upward FUNARG Problem (2 of 4)

- The function `add_x` returns a closure (lines 3–4), which adds its argument (i.e., `y`) to the argument to `add_x` (i.e., `x`) and returns the result.
- The `add_x` function provides the simplest non-trivial example of a closure. The `add_x` function creates (and returns) a closure around the inner function.
- The returned function contains references to data that no longer exists on the stack.
- This is the essence of the FUNARG problem—how to implement first-class functions in a stack-based language.

Figure 6.5 Illustration of the Upward FUNARG Problem Highlighting a Reference to a Declaration in a Nonexistent Activation Record

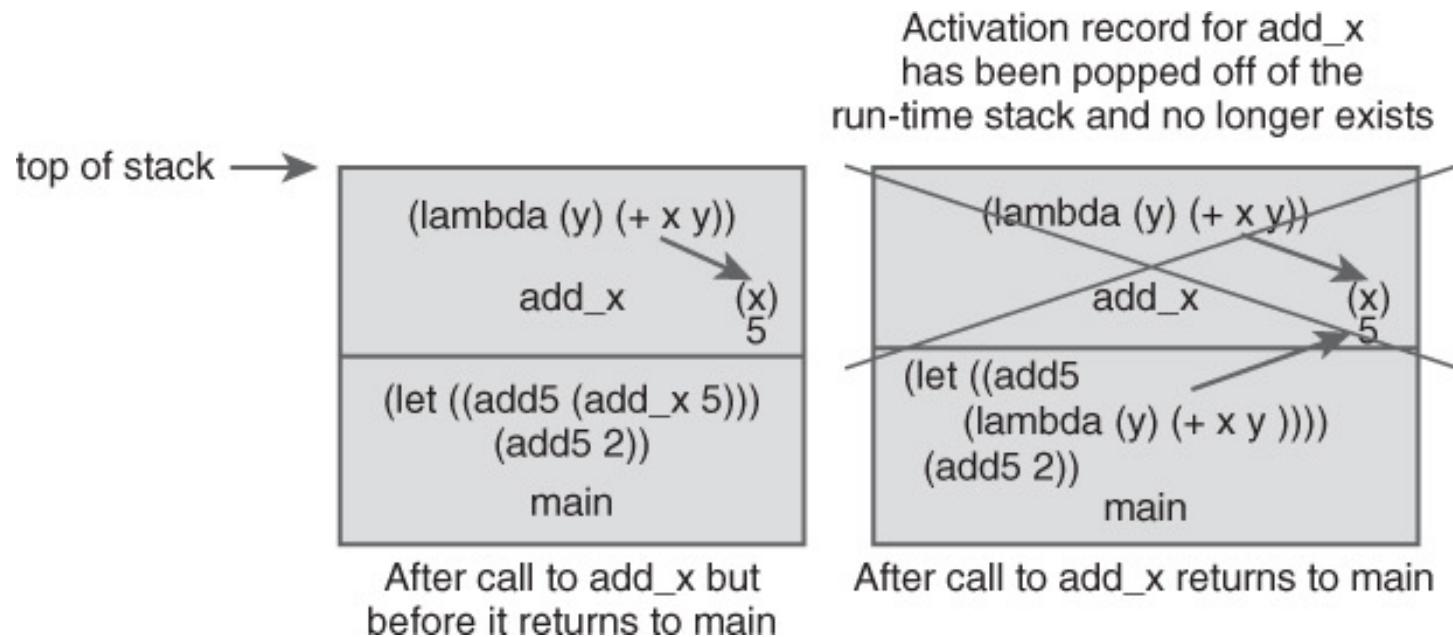


Table 6.6 Example Data Structure Representation of Closures

Name of Closure	Closure	
	expression	environment
add5	(lambda (y) (+ x y))	(x 5)
add6	(lambda (y) (+ x y))	(x 6)

6.10.2 The Upward FUNARG Problem (3 of 4)

- Python supports both first-class procedures and first-class closures.
- Python rendition of the make adder program
- new_counter Scheme function that clones or instantiates counter closures
- new_counter function resembles a constructor—it constructs new counters (i.e., objects).
- Closures and objects share similarities
 - Encapsulation of behavior and state
 - Information hiding
 - Arbitrary construction at the programmer's discretion (e.g., new_counter)
 - Existence of each in a separate memory space

6.10.2 The Upward FUNARG Problem (4 of 4)

- Python renditions of the `new_counter` function
- Also notice here that we use a named (i.e., `def`) rather than anonymous (i.e., `lambda`) function.

6.10.3 Relationship Between Closures and Scope

- A closure is a function with free or open variables that are bound to declarations determinable before run-time.
- The declarations to which the *open* variables are bound are *closed* before run-time (i.e., static scoping) rather than left *open* until run-time (i.e., dynamic scoping).
- *Closures*—functions with free variables—and *combinators*—functions without free variables—are opposites of each other.

6.10.4 Uses of Closures

First-class closures are a fundamental primitive in programming languages from which to construct and conceive:

- powerful abstractions (e.g., control structures) and
- concepts (e.g., parameter-passing mechanisms including as lazy evaluation).

6.10.5 The Upward and Downward FUNARG Problem in a Single Function

- Some functions accept one or more functions as arguments *and* return a function as a value.
- They involve both downward and upward FUNARG problems.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

(define list-of
  (lambda (pred)
    (lambda (lst)
      (cond
        ((null? lst) #t)
        ((pred (car lst)) ((list-of pred) (cdr lst)))
        (else #f))))
```

6.10.6 Addressing the FUNARG Problem

- Lambda lifting (λ -*lifting*) involves converting a closure
 - i.e., a λ -expression with free variablesinto a *pure* function
 - i.e., a λ -expression with no free variablesby passing values for those free variables as arguments to the λ -expression containing the free variables itself.
- λ -*lifting* is a simple solution, but it does not work in general.
- Another approach: build a closure and pass it to the FUNARG as a argument when the FUNARG is invoked

6.11 Deep, Shallow, and Ad Hoc Binding

- *Deep binding*, which uses the environment at the time the passed function was created
- *Shallow binding*, which uses the environment of the expression that *invokes* the passed function
- *Ad hoc binding*, which uses the environment of the invocation expression in which the procedure is *passed* as an argument

Working Example

```
(let ((y 3))
  (let ((x 10)
        ;; to which declaration of y is the reference to y bound?
        (f (lambda (x) (* y (+ x x)))))

(let ((y 4))
  (let ((y 5)
        (x 6)

        (g (lambda (x y) (* y (x y)))))

(let ((y 2))

  (g f x))))))
```

Deep Binding (1 of 4)

```
(let ((y 3))
  (let ((x 10)
        ; 6      ?      6 6
        (f (lambda (x) (* y (+ x x))))))
    (let ((y 4))
      (let ((y 5)
            (x 6)
            ; f 6      6   f 6
            (g (lambda (x y) (* y (x y)))))
        (let ((y 2))
          ; 6
          (g f x)))))))
```

Deep Binding (2 of 4)

```
(let ((y 3))
  (let ((x 10)
        ; 6      3      6 6
        (f (lambda (x) (* y (+ x x)))))

  (let ((y 4))
    (let ((y 5)
          (x 6)
          ; f 6      6  f 6
          (g (lambda (x y) (* y (x y)))))

      (let ((y 2))
        ; 6
        (g f x)))))))
```

Deep Binding (3 of 4)

```
(let ((y 3))
  (let ((x 10)
        ; 6      36
        (f (lambda (x) (* y (+ x x)))))

(let ((y 4))
  (let ((y 5)
        (x 6)
        ; f 6      6      36
        (g (lambda (x y) (* y (x y)))))

(let ((y 2))
  ; 6
  (g f x))))))
```

Deep Binding (4 of 4)

```
(let ((y 3))
  (let ((x 10)
        ; 6          36
        (f (lambda (x) (* y (+ x x)))))

(let ((y 4))
  (let ((y 5)
        (x 6)
        ; f 6          216
        (g (lambda (x y) (* y (x y)))))

(let ((y 2))
  ; 6
  (g f x))))))
```

Shallow Binding (1 of 3)

```
(let ((y 3))
  (let ((x 10)
        ; 6      4    12
        (f (lambda (x) (* y (+ x x)))))

  (let ((y 4))
    (let ((y 5)
          (x 6)
          ; f 6      6
          (g (lambda (x y) (* y (x y)))))

    (let ((y 2))
      ; 6
      (g f x)))))))
```

Shallow Binding (2 of 3)

```
(let ((y 3))
  (let ((x 10)
        ; 6          48
        (f (lambda (x) (* y (+ x x)))))

(let ((y 4))
  (let ((y 5)
        (x 6)
        ; f 6          6  48
        (g (lambda (x y) (* y (x y)))))

(let ((y 2))
  ; 6
  (g f x))))))
```

Shallow Binding (3 of 3)

The environment in which the function is called is:

```
(let ((y 3))
  (let ((x 10)
        ; 6      48
        (f (lambda (x) (* y (+ x x)))))

(let ((y 4))
  (let ((y 5)
        (x 6)
        ; f 6      288
        (g (lambda (x y) (* y (x y)))))

  (let ((y 2))
    ; 288
    (g f x))))))
```

Ad Hoc Binding (1 of 3)

```
(let ((y 3))
  (let ((x 10)
        ; 6      2      12
        (f (lambda (x) (* y (+ x x)))))

  (let ((y 4))
    (let ((y 5)
          (x 6)
          ; f 6      6
          (g (lambda (x y) (* y (x y)))))

    (let ((y 2))
      ; 6
      (g f x))))))
```

Ad Hoc Binding (2 of 3)

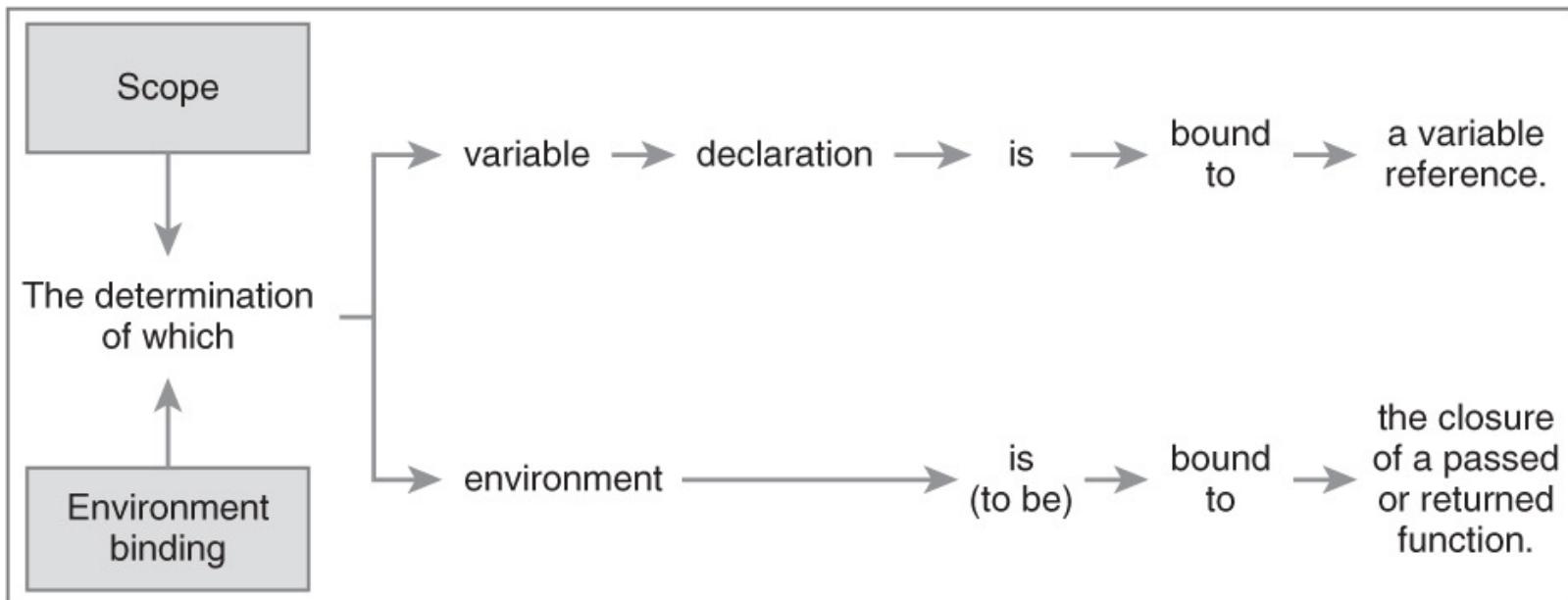
```
(let ((y 3))
  (let ((x 10)
        ; 6          24
        (f (lambda (x) (* y (+ x x))))))
  (let ((y 4))
    (let ((y 5)
          (x 6)
          ; f 6          6  24
          (g (lambda (x y) (* y (x y)))))
      (let ((y 2))
        ; 6
        (g f x))))))
```

Ad Hoc Binding (3 of 3)

The environment in which the function is called is:

```
(let ((y 3))                                (((y 2))
  (let ((x 10)                               ((y 5)
    ; 6          24                           (x 6)
    (f (lambda (x) (* y (+ x x))))))     (g (lambda (x y) (* y (x y))))) 
  (let ((y 4))                               ((y 4)))
    (let ((y 5)                               ((x 10)
      (x 6)                                 (f (lambda (x) (* (y (+ x x)))))))
      ; f 6          144                     ((y 3)))
      (g (lambda (x y) (* y (x y))))) )
  (let ((y 2))                               (g f x)))))))
```

Table 6.7 Scoping Vis-à-Vis Environment Binding

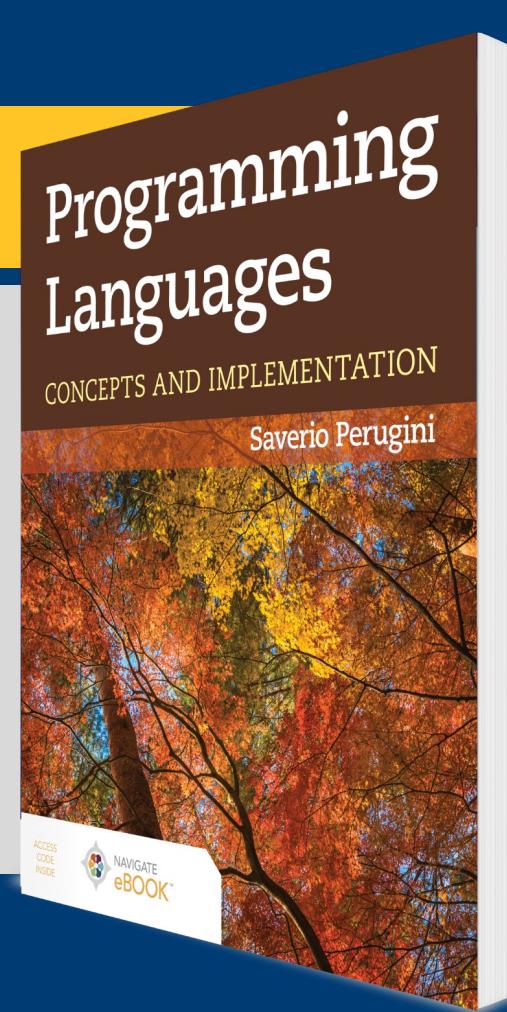


6.12 Thematic Takeaways

- Programming language concepts often have options, as with scoping (static or dynamic) and nonlocal reference binding (deep, shallow, or ad hoc).
- A *closure*—a function that *remembers* the lexical environment in which was created—is an essential element in the study of language concepts.
- The concept of *binding* is a universal and fundamental concept in programming languages. Languages have many different types of bindings; for example, scope refers to the binding of a reference to a declaration.
- Determining the scope in a programming language that uses manifest typing is challenging because manifest typing blurs the distinction between a variable declaration and a variable reference.
- Lexically scoped identifiers are useful for writing and understanding programs, but are superfluous and unnecessary for evaluating expressions and executing programs.
- The resolution of nonlocal references to the declarations to which they are bound is challenging in programming languages with support for first-class functions. These languages must address the FUNARG problem.

CHAPTER 7

Type Systems



Copyright © 2023 by Jones & Bartlett Learning, LLC an Ascend Learning Company. www.jblearning.com.

Chapter 7: Type Systems

Clumsy type systems drive people to dynamically typed languages.

—Robert Griesemer

We study programming language concepts related to types—particularly, *type systems* and *type inference*—in this chapter.

Outline

- **7.1 Chapter Objectives**
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.1 Chapter Objectives

- Compare the two varieties of *type systems* for *type checking* in programming languages: *statically typed* and *dynamically typed*.
- Describe *type conversions* (e.g., *type coercion* and *type casting*), *parametric polymorphism*, and *type inference*.
- Differentiate between *parametric polymorphism* and *function overloading*.
- Differentiate between *function overloading* and *function overriding*.

Outline

- 7.1 Chapter Objectives
- **7.2 Introduction**
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.2 Introduction (1 of 2)

- The *type system* in a programming language broadly refers to the language's approach to type checking.
- In a *static type system*, types are checked and almost all type errors are detected before run-time.
- In a *dynamic type system*, types are checked and most type errors are detected at run-time.
- Languages with static type systems are said to be *statically typed* or to use *static typing*.
- Languages with dynamic type systems are said to be *dynamically typed* or to use *dynamic typing*.
- Reliability, predictability, safety, and ease of debugging are advantages of a statically typed language.
- Flexibility and efficiency are benefits of using a dynamically typed language.

7.2 Introduction (2 of 2)

- There are a variety of methods for achieving a degree of flexibility within the confines of the type safety afforded by some statically typed languages:
 - *parametric polymorphism*,
 - *ad hoc polymorphism*, and
 - *type inference*.
- Why ML and Haskell?

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- **7.3 Type Checking**
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.3 Type Checking

- A *type* is a set of values (e.g., `int` in C = $\{-2^{15} \dots 2^{15} - 1\}$) and the permissible operations on those values (e.g., `+` and `*`).
- *Type checking* verifies that the values of types and (new) operations on them—and the values they return—abide by these constraints.
- Languages that permit programmers to deliberately violate the integrity constraints of types (e.g., by granting them access to low-level machine primitives and operations) have *unsound* or *unsafe type systems*.
 - e.g., Fortran, C, and C++ are *weakly typed languages*.
- In contrast, other languages do not permit programmers to circumvent type constraints
 - e.g., Java, ML, and Haskell all have a *sound* or *safe type system* and are, thus, sometimes referred to as *strongly typed* or *type safe languages* (Table 7.1).

Table 7.1 Features of Type Systems Used in Programming Languages

Concept	Definition	Example(s)
<i>Static type system</i>	Types are checked and almost all type errors are detected before run-time.	C/C++
<i>Dynamic type system</i>	Types are checked and most type errors are detected at run-time.	Python
<i>Safe type system</i>	Does not permit the integrity constraints of types to be deliberately violated.	C#, ML
<i>Unsafe type system</i>	Permits the integrity constraints of types to be deliberately violated.	C/C++
<i>Explicit typing</i>	Requires the type of each variable to be explicitly declared.	C/C++
<i>Implicit typing</i>	Does not require the type of each variable to be explicitly declared.	Python

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- **7.4 Type Conversion, Coercion, and Casting**
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.4 Type Conversion, Coercion, and Casting (1 of 2)

- 7.4.1 Type Coercion: Implicit Conversion
- 7.4.2 Type Casting: Explicit Conversion
- 7.4.3 Type Conversion Functions: Explicit Conversion

7.4 Type Conversion, Coercion, and Casting (2 of 2)

- There are a variety of methods for providing programmers with a degree of flexibility within the confines of the type safety afforded by some statically typed languages, thereby mitigating the rigidity enforced by a sound type system.
- These methods include *conversions* of various sorts, *parametric* and *ad hoc polymorphism*, and *type inference*.

7.4.1 Type Coercion: Implicit Conversion

- *Coercion* is an implicit conversion in which values can deviate from the type required by an operator or function without warning or error because the appropriate conversions are made automatically before or at run-time and are transparent to the programmer.
- See the C program `coercion.c`
- Notice also that coercion happens automatically without any intervention from the programmer.
- See the C program `storage.c`
- There are no guarantees with coercion.
- Java does not perform coercion (see `NoCoercion.java`), even between `floats` and `doubles` (see `NoCoercion2.java`)

7.4.2 Type Casting: Explicit Conversion

- There are two forms of explicit type conversions:
 - type casts and
 - conversion functions.
- A `type cast` is an explicit conversion that entails *interpreting* the bit pattern used to represent a value of a particular type as another type.
- See the C program `cast.c`.

7.4.3 Type Conversion Functions: Explicit Conversion

- Some languages also support built-in or library functions to convert values from one data type to another.
- See the C program `conversion.c`.
- Since the statically typed language ML does not have coercion, it needs provisions for converting values between types.
- ML supports conversions of values between types through functions.
- Conversion functions are necessary in Haskell, even though types can be mixed in some Haskell expressions.

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- **7.5 Parametric Polymorphism**
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.5 Parametric Polymorphism: Monomorphic Function Types

- Both ML and Haskell assign a unique type to every value, expression, operator, and function.
- Recall that the type of an operator or function describes the types of its domain and range.
- Certain operators require values of a particular type.
 - e.g., the `div` (i.e., division) operator in ML requires two operands of type `int` and has type `fn : int * int -> int` ,
 - whereas the `/` (i.e., division) operator in ML requires two operands of type `real` and has type `fn : real * real -> real` .
- These operators are *monomorphic*, meaning they have only one type.

7.5 Parametric Polymorphism: Polymorphic Function Types (1 of 3)

- Other operators or functions are *polymorphic*, meaning they can accept arguments of different types.
 - For example, the type of the `(+)` (i.e., prefix addition) operator in Haskell is

$$(+)\text{ :: Num } a \Rightarrow (a, a) \rightarrow a$$

indicating that if type `a` is in the type class `Num`, then the `+` operator has type

$$(a, a) \rightarrow a$$

- In other words, `(+)` is an operator that maps two values of the same type `a` to a value of the same type `a`.
- If the first operand to the `(+)` operator is of type `Int`, then `(+)` is an operator that maps two `Ints` to an `Int`.

7.5 Parametric Polymorphism: Polymorphic Function Types (2 of 3)

- With this type of polymorphism, referred to as *parametric polymorphism*, a function or data type can be defined generically so that it can handle arguments in an identical manner, no matter what their type.
- The types themselves in the type signature are parameterized.

7.5 Parametric Polymorphism: Polymorphic Function Types (3 of 3)

- A polymorphic function type in ML or Haskell specifies that the type of any function with that polymorphic type is one of multiple monomorphic types.
- Recall that a polymorphic function type is a type expression containing type variables.
- The polymorphic type `reverse :: [a] -> [a]` in Haskell is a shorthand for a collection of the following (non-exhaustive) list of types:

```
reverse :: [Int] -> [Int],    reverse :: [String] -> [String] ,
```

- and so on.
- Unlike in languages with unsafe type systems (e.g., C or C++), in ML, the programmer is not permitted—because a program doing so will not run—to deviate at all from the required types when invoking an operator or function.

7.5 Parametric Polymorphism

- In Haskell, as in ML, the programmer is not permitted to deviate at all from the required types when invoking an operator or function.
- However, unlike ML, Haskell has a hierarchy of type classes, where a class is a collection of types, which provides flexibility in function definition and application.
- Haskell's type class system comprises a hierarchy of interoperable type, where a value of a type (e.g., `Integral`) is also considered a value of one of the supertypes of that type in the hierarchy (e.g., `Num`).

Contrast a square Function in Haskell with That in ML

Haskell

```
Prelude> square(n) = n*n
```

```
Prelude> :type square
square :: Num a => a -> a
```

ML

```
- fun square(n) = n*n;
val square = fn : int -> int
```

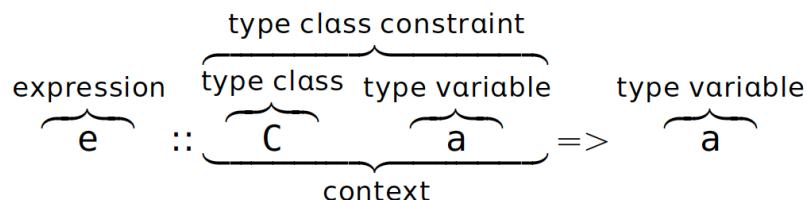
In Haskell, the type of `square` is called a *qualified type* and `Num` is a *type class*.

Qualified or Constrained Types in Haskell

- The `fromInteger` function is implicitly (i.e., automatically and transparently to the programmer) applied to every literal number without a decimal point:

```
Prelude> :type 1
1 :: Num p => p
```

- This response indicates that if type `p` is in the type class `Num`, then `1` has the type `p`.
- In other words, `1` is of some type in the `Num` class.
- Such a type is called a *qualified type* or *constrained type* (Table 7.2).



- A *type class* is a collection of types that are guaranteed to have definitions for a set of functions—like a Java interface.

Table 7.2 The General Form of a *Qualified Type* or *Constrained Type* and an Example

General:

$e :: C \ a \Rightarrow a$ means “If type a is in type class C , then e has type a .”

Example:

$3 :: \text{Num} \ a \Rightarrow a$ means “If type a is in type class Num , then 3 has type a .”

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- **7.6 Operator/Function Overloading**
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.6 Operator/Function Overloading (1 of 3)

- *Operator/function overloading* refers to using the same function name for multiple function definitions, where the type signature of each definition involves
 - a different return type,
 - different types of parameters,
 - and/or a different number of parameters.
- When an overloaded function is invoked, the applicable function definition to bind to the function call (obtained from a collection of definitions with the same name) is determined based on
 - the number and/or
 - the types of arguments used in the invocation.
- Function/operator overloading is also called *ad hoc polymorphism*.

7.6 Operator/Function Overloading (2 of 3)

- See `overloading.hs`
- In C, functions cannot be overloaded (see `nooverloading.c`)
- ML, Haskell, and C do not support function overloading.
- C++ and Java do support function overloading:
- See `overloading.cpp`, `overloading2.cpp`, and `Overloading.java`

7.6 Operator/Function Overloading (3 of 3)

- The Haskell type class system supports the definition of what seem to be overloaded functions like `square`.
- The flexibility fostered by a type or class hierarchy in the definition of functions is similar to ad hoc polymorphism (i.e., overloading), but is called *interface polymorphism*.
- ML and Haskell programs are thoroughly type-checked before run-time.
- Almost no ML or Haskell program that can run will ever have a type error.

Table 7.3 Parametric Polymorphism Vis-à-Vis Function Overloading

Type Concept	Function Definitions	Number of Parameters	Types of Parameters	Example Type Signature(s)
Parametric Polymorphism Function Overloading (Ad Hoc Polymorphism)	single multiple	same varies	parameterized instantiated	[a] -> [a] int -> int int * bool -> float int * float * char -> bool

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- **7.7 Function Overriding**
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.7 Function Overriding (1 of 3)

Function overriding (also called *function hiding*) occurs when multiple function definitions share the same function name, but only one of those function definitions is visible at any point in the program due to the presence of scope holes.

7.7 Function Overriding (2 of 3)

```
1 (define overriding
2   (lambda ()
3     (let ((f (lambda ()
4       (let ((g (lambda ()
5         (let ((f (lambda () (+ 1 2))))
6           ;; call to inner f
7           (f))))))
8       (g))))))
9     ;; call to outer f
10    (f))))
```

7.7 Function Overriding (3 of 3)

- Here, the call to function `f` on line 10 binds to the outermost definition of `f` (starting on line 3) because the innermost definition of `f` (line 5) is not visible on line 10—it is defined in a nested block.
- The call to function `f` on line 7 binds to the innermost definition of `f` (line 5) because on line 7 where `f` is called, the innermost definition of `f` (line 5) shadows the outermost definition of `f`.
- In other words, the outermost definition of `f` is not visible on line 7.

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- **7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing**
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing

- The concepts of *static/dynamic typing* and *explicit/implicit typing* are sometimes confused and used interchangeably.
- The modifiers “static” or “dynamic” on “typing” (or “checking”) indicate the time at which types and type errors are checked.
- However, the types of those variables can be declared explicitly (e.g., `int x = 1;` in Java) or implicitly (e.g., `x = 1` in Python).
- Languages that require the type of each variable to be explicitly declared use *explicit typing*.
- Languages that do not require the type of each variable to be explicitly declared use *implicit typing*, which is also referred to as *manifest typing* (Table 7.1).
- Statically typed languages can use either explicit (e.g., Java) or implicit (e.g., ML and Haskell) typing.
- Dynamically typed languages typically use implicit typing (e.g., Python, JavaScript, Ruby).

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- **7.9 Type Inference**
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.9 Type Inference (1 of 3)

- Explicit type declarations of values and variables help inform a static type system.
- See also `declaring.hs`
- In some languages with first-class functions, especially statically typed languages, functions have types.
- Instead of ascribing a type to each individual parameter and the return type of a function, we can declare the type of the entire function.
- Explicitly declaring types requires effort on the part of the programmer and can be perceived as requiring more effort than necessary to justify the benefits of a static type system.

7.9 Type Inference (2 of 3)

- Type inference is a concept of programming languages that represents a compromise and attempts to provide the best of both worlds.
- Type inference refers to the automatic deduction of the type of a value or variable without an explicit type declaration.
- ML and Haskell use type inference, so the programmer is not required to declare the type of any variable.
- ML and Haskell include a built-in type inference engine (i.e., *Hindley–Milner algorithm*) to deduce the type of a value based on context.

7.9 Type Inference (3 of 3)

- Strong typing provides safety, but requires a type to be associated with every name.
- The use of type inference in a statically typed language obviates the need to associate a type with each identifier:

Static, Safe Type System + Type Inference Obviates the Need to Declare Types

Static, Safe Type System + Type Inference \rightsquigarrow Reliability/Safety + Manifest Typing

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- **7.10 Variable-Length Argument Lists in Scheme**
- 7.11 Thematic Takeaways

7.10 Variable-Length Argument Lists in Scheme (1 of 2)

- Every function in Scheme is defined to accept only one list argument.
- Arguments to any Scheme function are always received collectively as one list, not as individual arguments.
- It is up to the programmer to decompose that argument list and group individual arguments in the formal parameter specification of the function definition using dot notation.
- Moreover, Scheme, like ML and Haskell, does pattern matching from this single list of arguments to the specification of the parameter list in the function definition.

eval / apply in Scheme

- eval and apply are the heart of any interpreter.
- eval is the Scheme primitive analog of evaluate_expr function in our coming Camille interpreter.
- eval accepts an expression and an environment as arguments.

```
> (define f (lambda (x) (cons x '())))
> (g 5)
> (5)
> (define f2 (list 'lambda '(x) (list cons 'x '())))
> f2
'(lambda (x) (cons x '()))
> (eval f)
#<procedure:f>
> ((eval f) 5)
'(5)
```

apply in Scheme

- `apply` is the Scheme primitive analog of the family of `apply_*` functions in our coming Camille interpreter (e.g., `apply_primitive`, `apply_closure`, and so on).
- `apply` accepts a function and its arguments as arguments:

```
>(apply + '(1 2 3))
```

```
> 6
```

7.10 Variable-Length Argument Lists in Scheme (2 of 2)

```
(define f (lambda (x) x)) ;;; continued from previous column
(f 1) ;;; only 1 argument passed
(f '(1 2 3))
;; x is just the list (1 2 3)
(define f (lambda x x))
(f 1 2 3)
(f 1)
;; uses pattern matching like ML and Haskell
;; g and h take a variable number of arguments
(define g (lambda (x . y) x))
(define h (lambda (x . xs) xs))
```

```
(g ' (1 2 3))
(h ' (1 2 3))
(write "a")
(newline)
;; now 2 arguments passed
(g 1 ' (2 3))
(h 1 ' (2 3))
;; now 3 arguments passed
(g 1 2 3)
(h 1 2 3)
```

Table 7.4 Scheme Vis-à-Vis ML and Haskell for Fixed- and Variable-Sized Argument Lists

	Natively		Through Simulation	
	fixed-size	variable-size	fixed-size	variable-size
Scheme	✓ (only one argument)	✗	✓ (use .)	✓ (use .)
ML	✓ (one or more arguments)	✗	N/A	✗
Haskell	✓ (one or more arguments)	✗	N/A	✗

Table 7.5 Scheme Vis-à-Vis ML and Haskell for Reception and Decomposition of Argument(s)

	Parameter(s) Reception	Single List Arg Decomposition	Example
Scheme	as a list	✗	N/A
ML	as a tuple	✓ (use <code>::</code>)	<code>x::xs</code>
Haskell	as a tuple	✓ (use <code>:</code>)	<code>x:xs</code>

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- **7.11 Thematic Takeaways**

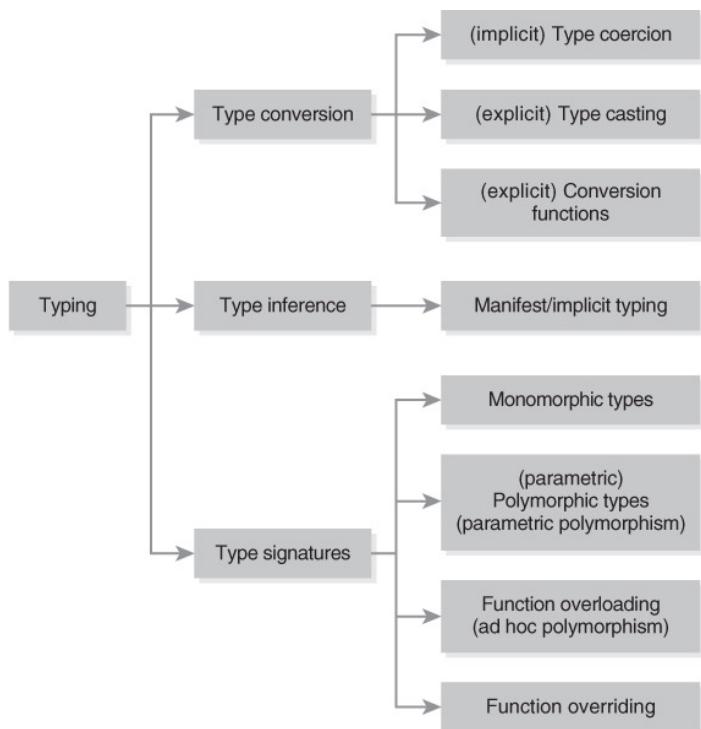
7.11 Thematic Takeaways

- Languages using *static type checking* detect nearly all type errors before run-time; languages using *dynamic type checking* delay the detection of most type errors until run-time.
- The use of automatic *type inference* allows a statically typed language to achieve reliability and safety without the burden of having to declare the type of every value or variable:

Static, Safe Type System + Type Inference \rightsquigarrow Reliability/Safety + Manifest Typing

- There are practical trade-offs between *statically* and *dynamically typed* languages—such as other issues in the design and use of programming languages.

Figure 7.1 Hierarchy of Concepts to Which the Study of Typing Leads



Chapter 8: Currying and Higher-Order Functions

[T]here are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—Tony Hoare, 1980 ACM A. M. Turing Award Lecture

The concept of *static typing* leads to *type inference* and *type signatures* for functions (all of which are covered in Chapter 7), which lead to the concepts of *currying* and *partial function application*, which we discuss in this chapter. All of these concepts are integrated in the context of higher-order functions, which also provide us with tools and techniques for constructing well-designed and -factored software systems, including interpreters (which we build in Chapters 10–12).

Outline

- **8.1 Chapter Objectives**
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

1.2 Chapter Objectives

- Explore the programming language concepts of *partial function application* and *currying*.
- Describe *higher-order functions* and their relationships to curried functions, which together support the development of well-designed, concise, elegant, and reusable software.

Outline

- 8.1 Chapter Objectives
- **8.2 Partial Function Application**
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

8.2 Partial Function Application (1 of 3)

With *partial function application*, for any function $f(p_1, p_2, \dots, p_n)$,

$$f(a_1) = g(p_2, p_3, \dots, p_n)$$

such that

$$g(a_2, a_3, \dots, a_n) = f(a_1, a_2, a_3, \dots, a_n)$$

Table 8.1 Type Signatures and λ -Calculus for a Variety of Higher-Order Functions

Concept	Function	Type Signature	λ -Calculus
com fun appl	apply	$((a \times b \times c) \rightarrow d) \times a \times b \times c \rightarrow d$	$= \lambda(f, x, y, z).f(x, y, z)$
part fun appl 1	papply1	$((a \times b \times c) \rightarrow d) \times a \rightarrow ((b \times c) \rightarrow d)$	$= \lambda(f, x).\lambda(y, z).f(x, y, z)$
part fun appl n	papply	$((a \times b \times c) \rightarrow d) \times a \rightarrow ((b \times c) \rightarrow d)$ $((a \times b \times c) \rightarrow d) \times a \times b \rightarrow (c \rightarrow d)$ $((a \times b \times c) \rightarrow d) \times a \times b \times c \rightarrow (\{\} \rightarrow d)$	$= \lambda(f, x).\lambda(y, z).f(x, y, z)$ $= \lambda(f, x, y).\lambda(z).f(x, y, z)$ $= \lambda(f, x, y, z).\lambda().f(x, y, z)$
currying	curry	$(a \times b \times c) \rightarrow d \rightarrow (a \rightarrow (b \rightarrow (c \rightarrow d)))$	$= \lambda(f).\lambda(x).\lambda(y).\lambda(z).f(x, y, z)$
uncurrying	uncurry	$a \rightarrow (b \rightarrow (c \rightarrow d)) \rightarrow ((a \times b \times c) \rightarrow d)$	$= \lambda(f).\lambda l.f(car l)(cad r l)(cad dr l)$

Each signature assumes a ternary function $f : (a \times b \times c) \rightarrow d$. All of these functions except apply return a function. In other words, all but apply are closed operators.

Table 8.2 Definitions of papply1 and papply in Scheme

(define papply1 (lambda (fun arg) (lambda x (apply fun (cons arg x))))))	(define papply (lambda (fun . args) (lambda x (apply fun (append args x))))))
---	--

8.2 Partial Function Application (2 of 3)

More generally, with *partial function application*, for any function $f(p_1, p_2, \dots, p_n)$,

$$f(a_1, a_2, \dots, a_m) = g(p_{m+1}, p_{m+2}, \dots, p_n)$$

where $m \leq n$, such that

$$g(a_{m+1}, a_{m+2}, \dots, a_n) = f(a_1, a_2, \dots, a_m, a_{m+1}, a_{m+2}, \dots, a_n)$$

8.2 Partial Function Application (3 of 3)

More formally, assuming an n -ary function f , where $n > 0$:

$$\underbrace{papply(\dots(papply(\underbrace{papply(f, 1)}, 2), \dots), n)}_{\text{argumentless function fixpoint}}$$

$(n-2)$ -ary-function
 $\overbrace{\hspace{10em}}$
 $(n-1)$ -ary function

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- **8.3 Currying**
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

8.3 Currying

- *Currying* refers to converting an n -ary function into one that accepts only one argument and returns a function, which also accepts only one argument and returns a function that accepts only one argument, and so on.
- For now, we can think of a curried function as one that permits transparent partial function application (i.e., without calling `papply1` or `papply`).
- In other words, a curried function (or a function written in curried form, as discussed next) can be partially applied without calling `papply1` or `papply`.
- Later, we see that a curried function is not being partially applied at all.

8.3.1 Curried Form in Haskell (1 of 4)

- Consider the following two definitions of a power function (i.e., a function that computes a base b raised to an exponent e , b^e) in Haskell:

```
1 Prelude> :{
2 Prelude| powucf(0, _) = 1
3 Prelude| powucf(1, b) = b
4 Prelude| powucf(_, 0) = 0
5 Prelude| powucf(e, b) = b * powucf(e-1, b)
6 Prelude|
7 Prelude| powcf 0 _ = 1
8 Prelude| powcf 1 b = b
9 Prelude| powcf _ 0 = 0
10 Prelude| powcf e b = b * powcf (e-1) b
11 Prelude| :}
```

- These two definitions are almost the same.
- But the types of these functions are different.
- The definition of the `powucf` function has a comma between each parameter in the tuple of parameters, and that tuple is enclosed in parentheses
- There are no commas and parentheses in the parameters tuple in the definition of the `powcf` function.

8.3.1 Curried Form in Haskell (2 of 4)

```
12 Prelude> :type powucf
13 powucf :: (Num a, Num b, Eq a, Eq b) => (a, b) -> b
14 Prelude>
15 Prelude> :type powcf
16 powcf :: (Num t1, Num t2, Eq t1, Eq t2) => t1 -> t2 -> t2
```

- The type of the `powucf` function states that it accepts a tuple of values of a type in the `Num` class and returns a value of a type in the `Num` class.
- In contrast, the type of the `powcf` function indicates that it accepts a value of a type in the `Num` class and returns a function mapping a value of a type in the `Num` class to a value of the same type in the `Num` class.

8.3.1 Curried Form in Haskell (3 of 4)

The definition of `powcf` is written in *curried form*, meaning that it accepts only one argument and returns a function, also with only one argument:

```
17 Prelude> square = powcf 2
18 Prelude>
19 Prelude> :type square
20 square :: (Num t2, Eq t2) => t2 -> t2
21 Prelude>
22 Prelude> cube = powcf 3
23 Prelude>
24 Prelude> :type cube
25 cube :: (Num t2, Eq t2) => t2 -> t2
26 Prelude>
27 Prelude> (powcf 2) 3
28 9
29 Prelude> square 3
30 9
31 Prelude> (powcf 3) 3
32 27
33 Prelude> cube 3
34 27
```

8.3.1 Curried Form in Haskell (4 of 4)

- By contrast, the definition of `powucf` is written in *uncurried form*, meaning that it must be invoked with arguments for all of its parameters with parentheses around the argument list and commas between individual arguments.
- `powucf` cannot be partially applied, without the use of `papply1` or `papply`
 - It must be completely applied:

```
35 Prelude> powucf(2,3)
36 9
37 Prelude>
38 Prelude> powucf(2)
39
40 <interactive>:36:1: error:
41     Non type-variable argument in the constraint: Num (a, b)
42     (Use FlexibleContexts to permit this)
43     When checking the inferred type
44     it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b
45 Prelude>
46 Prelude> powucf 2
47
48 <interactive>:38:1: error:
49     Non type-variable argument in the constraint: Num (a, b)
50     (Use FlexibleContexts to permit this)
51     When checking the inferred type
52     it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b
```

8.3.2 Currying and Uncurrying (1 of 2)

In general, currying transforms a function $f_{uncurried}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \rightarrow r$$

into a function $f_{curried}$ with the type signature

$$p_1 \rightarrow (p_2 \rightarrow (\cdots \rightarrow (p_n \rightarrow r) \cdots))$$

such that

$$f_{uncurried}(a_1, a_2, \dots, a_n) = (\cdots ((f_{curried}(a_1))(a_2)) \cdots)(a_n)$$

8.3.2 Currying and Uncurrying (2 of 2)

Currying $f_{uncurried}$ and running the resulting $f_{curried}$ function has the same effect as progressively partially applying $f_{uncurried}$.

Inversely, uncurrying transforms a function $f_{curried}$ with the type signature

$$p_1 \rightarrow (p_2 \rightarrow (\cdots \rightarrow (p_n \rightarrow r) \cdots))$$

into a function $f_{uncurried}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \rightarrow r$$

such that

$$f_{uncurried}(a_1, a_2, \dots, a_n) = (\cdots ((f_{curried}(a_1))(a_2)) \cdots)(a_n)$$

8.3.3 The `curry` and `uncurry` Functions in Haskell

The built-in Haskell functions `curry` and `uncurry` are used to convert a binary function between uncurried and curried forms:

```
73 Prelude> :type curry
74 curry :: ((a,b) -> c) -> a -> b -> c
75 Prelude>
76 Prelude> :type uncurry
77 uncurry :: (a -> b -> c) -> (a,b) -> c
```

Table 8.3 Definitions of `curry` and `uncurry` in Curried Form in Haskell for Binary Functions

curry :: ((a,b)->c) -> a->b->c	uncurry :: (a->b->c) -> ((a,b)->c)
curry f a b = f (a,b)	uncurry f (a,b) = f a b

Notice that the definitions of `curry` and `uncurry` in Haskell are written in curried form.

Table 8.4 Definitions of curry and uncurry in Scheme for Binary Functions

(define curry (lambda (fun_ucf) (lambda (x) (lambda (y) (fun_ucf x y))))	(define uncurry (lambda (fun_cf) (lambda args ; (x y) ((fun_cf (car args)) (cadr args)))) ;; x y
--	---

8.3.4 Flexibility in Curried Functions

- A curried function is more flexible than its uncurried analog because it can effectively be invoked in n ways, where n is the number of arguments its uncurried analog accepts:
 - The one and only way its uncurried analog is invoked (i.e., with all arguments as a complete application)
 - The one and only way it itself can be invoked (i.e., with only one argument)
 - $n - 2$ other ways corresponding to implicit partial applications of each returned function
- Any curried function can *effectively* be invoked with arguments for any prefix, akin to partial function application, including all of the parameters of its uncurried analog, without parentheses around the list of arguments or commas between individual arguments.

8.3.5 All Built-in Functions in Haskell Are Curried (1 of 2)

- This is why Haskell is referred to as a *fully* curried language.
- This is not the case in ML.
- Consider our final definition of `mergesort` in Haskell given in online Appendix C.
 - Neither the `mergesort` function nor the `compop` function is curried.
 - Thus, we cannot pass in the built-in `<` or `>` operators, because they are curried.

Converting an Infix Operator to a Prefix Operator

- When passing an operator as an argument to a function, the passed operator must be a prefix operator.
- Since the operators `<` and `>` are infix operators, we cannot pass them to this version of `mergesort` without first converting them to prefix operators.
- We can convert an infix operator to a prefix operator in Haskell either by wrapping it in a user-defined function or by enclosing it within parentheses:

```
44 Prelude> :type (+)
45 (+) :: Num a => a -> a -> a
46 Prelude>
47 Prelude> (+) 7 2
48 9
49 Prelude> add1 = (+) 1
50 Prelude>
51 Prelude> :type add1
52 add1 :: Num a => a -> a
53 Prelude>
54 Prelude> add1 9
55 10
```

8.3.5 All Built-in Functions in Haskell Are Curried (2 of 2)

The function `mergesort` is an ideal candidate for currying because by applying it in curried form with the `<` or `>` operators, we get back ascending-sort and descending-sort functions, respectively:

```
126 Prelude> (curry mergesort) (<) [9,8,7,6,5,4,3,2,1]
127 [1,2,3,4,5,6,7,8,9]
128 Prelude>
129 Prelude> ascending_sort = (curry mergesort) (<)
130 Prelude>
131 Prelude> :type ascending_sort
132 ascending_sort :: Ord a => [a] -> [a]
133 Prelude>
134 Prelude> ascending_sort [9,8,7,6,5,4,3,2,1]
135 [1,2,3,4,5,6,7,8,9]
136 Prelude>
137 Prelude> (curry mergesort) (>) [1,2,3,4,5,6,7,8,9]
138 [9,8,7,6,5,4,3,2,1]
139 Prelude>
140 Prelude> descending_sort = (curry mergesort) (>)
141 Prelude>
142 Prelude> :type descending_sort
143 descending_sort :: Ord a => [a] -> [a]
144 Prelude>
145 Prelude> descending_sort [1,2,3,4,5,6,7,8,9]
146 [9,8,7,6,5,4,3,2,1]
```

8.3.6 Supporting Curried Form Through First-Class Closures

- Any language with first-class closures can be used to define functions in curried form.
- For instance, because Python supports first-class closures, we can define the `pow` function in curried form in Python:

```
>>> def pfa_pow(e):
...     def pow_e(b):
...         if e == 0:
...             return 1
...         elif e == 1:
...             return b
...         elif b == 0:
...             return 0
...         else: return b * (pfa_pow(e-1) (b))
...     return pow_e
...
>>> pfa_pow(2) (3)
9
>>> square = pfa_pow(2)
>>>
>>> square(3)
9
>>> pfa_pow(3) (3)
27
>>> cube = pfa_pow(3)
>>>
>>> cube(3)
27
```

Here the curried form is too tightly woven into the function definition (like Scheme, but unlike ML/Haskell). Moreover, `pfa_pow` cannot be completely applied (again, like Scheme, but unlike ML/Haskell).

8.3.7 ML Analogs (Curried Form in ML)

Same as in Haskell

```
fun pow 0 _ = 1
| pow 1 b = b
| pow _ 0 = 0
| pow e b = b * pow (e-1) b;

(*
int * int -> int
=>
int -> int -> int
*)
val square = pow 2;
val cube = pow 3;
```

- Not all built-in ML functions are curried as in Haskell.
 - e.g., map is curried, while Int.+ is uncurried.
- Also, there are no built-in curry and uncurry functions in ML.

8.4 Putting It All Together: Higher-Order Functions

- Curried functions open up new possibilities in programming, especially with respect to higher-order functions (HOFs).
 - A HOF, such as `map` in Scheme, is a function that either accepts functions as arguments or returns a function as a value, or both.
 - Such functions capture common, typically recursive, programming patterns as functions.
- They provide the glue that enables us to combine simple functions to make more complex functions.
- Most HOFs are curried, which makes them powerful and flexible.
- Writing a program to solve a problem with HOFs requires:
 - Creative insight to discern the applicability of a HOF approach to solving a problem
 - The ability to decompose the problem and develop atomic functions at an appropriate level of granularity to foster:
 - A solution to the problem at hand by composing atomic functions with HOFs
 - The possibility of recomposing the constituent functions with HOFs to solve alternative problems in a similar manner

8.4.1 Functional Mapping

- The `map` function in ML and Haskell accepts only a unary function and returns a function that accepts a list and applies the unary function to each element of the list, and returns a list of the results.
- The HOF `map` is also built into both ML and Haskell and is curried in both.

Mapping in Haskell (left) Vis-à-Vis ML (right)

Haskell

```
ourmap f [] = []
ourmap f (x:xs) = (f x):(ourmap f xs)
```

```
square n = n*n
```

```
ans = ourmap square [1,2,3,4,5,6]
```

```
squarelist lon = map square lon
```

```
ans2 = squarelist [1,2,3,4,5,6]
```

vis-à-vis

```
squarelist = map square
```

ML

```
fun ourmap f nil = nil
| ourmap f (x::xs) = (f x)::ourmap f xs;
```

```
fun square x = x*x;
```

```
ourmap square [1,2,3,4,5,6];
```

```
fun squarelist lon = map square lon;
```

```
squarelist [1,2,3,4,5,6];
```

vis-à-vis

```
val squaralist = map square;
```

8.4.2 Functional Composition

- The function composition operator that accepts only two unary functions and returns a function that invokes the two in succession.
- In mathematics, $g \circ f = g(f(x))$, which means
 - “first apply f and then apply g ” or
 - “ g followed by f ” or “ g of f of x .”
- The functional composition operator is `.` in Haskell and `o` in ML:

```
- (op o);
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- fun add3 x = x+3;
val add3 = fn : int -> int
- fun mult2 x = x*2;
val mult2 = fn : int -> int
- val add3_then_mult2 = mult2 o add3;
val add3_then_mult2 = fn : int -> int
- val mult2_then_add3 = add3 o mult2;
val mult2_then_add3 = fn : int -> int
- add3_then_mult2 4;
val it = 14 : int
- mult2_then_add3 4;
val it = 11 : int
```

8.4.3 Sections in Haskell (1 of 4)

In Haskell, any binary function or binary prefix operator (e.g., `div` and `mod`) can be converted into an equivalent infix operator by enclosing the name of the function in grave quotes (e.g., ``div``):

```
Prelude> add x y = x+y
Prelude>
Prelude> :type add
add :: Num a => a -> a -> a
Prelude>
Prelude> 3 `add` 4
7
Prelude> 7 `div` 2
3
Prelude> div 7 2
3
Prelude> 7 `div` 2
3
Prelude> mod 7 2
1
Prelude> 7 `mod` 2
1
```

8.4.3 Sections in Haskell (2 of 4)

Parenthesizing an infix operator in Haskell converts it to the equivalent curried prefix operator:

```
Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> (+) (1,2)
<interactive>:12:1: error:
  Non type-variable argument in the constraint: Num (a, b)
  (Use FlexibleContexts to permit this)
  When checking the inferred type
    it :: forall a b. (Num a, Num b, Num (a, b)) =>
                  (a, b) -> (a, b)
Prelude> (+) 1 2
3
```

8.4.3 Sections in Haskell (3 of 4)

- An operator in Haskell can be partially applied only if it is both curried and invocable in prefix form:

```
Prelude> :type (+) 1
(1 +) :: Num a => a -> a
```

- This convention also permits one of the arguments to be included in the parentheses
 - Which both converts the infix binary operator to a prefix binary operator and partially applies it in one stroke:

```
Prelude> :type (1+)
(1 +) :: Num a => a -> a
Prelude> (1+) 3
4
Prelude> :type (+3)
flip (+) 3 :: Num a => a -> a
Prelude> (+3) 1
4
```

Uses of Sections

1. Constructing simple and succinct functions. Example: `(+3)`
2. Declaring the type of an operator (because an operator itself is not a valid expression in Haskell). Example: `(+) :: Num a => a -> a -> a`
3. Passing a function to a higher-order function. Example: `map (+1) [1, 2, 3, 4]`

8.4.3 Sections in Haskell (4 of 4)

- The same is not possible in ML because built-in operators (e.g., + and *) are not curried in ML.
- To convert an infix operator (e.g., + and *) to the equivalent prefix operator in ML, we must enclose the operator in parentheses (as in Haskell) and also include the lexeme `op` after the opening parenthesis:

```
- val add3_then_mult2 = (op o) (mult2, add3);  
val add3_then_mult2 = fn : int -> int
```

- The concepts of mapping, functional composition, and sections are interrelated:

```
Prelude> inclist = map ((+) 1)  
Prelude>  
Prelude> :type inclist  
inclist :: Num b => [b] -> [b]  
Prelude>  
Prelude> inclist [1,2,3,4,5,6]  
[2,3,4,5,6,7]
```

8.4.4 Folding Lists

- The built-in ML and Haskell functions `foldl` (“fold left”) and `foldr` (“fold right”), like `map`, capture a common pattern of recursion.
- These list folding functions are helpful for defining a variety of functions.
- The functions `foldl` and `foldr` both accept only
 - a prefix binary function (sometimes called the folding function or the combining function),
 - a base value (i.e., the base of the recursion), and
 - a list, in that order:
`Prelude> :type foldl`
`foldl :: (a -> b -> a) -> a -> [b] -> a`
`Prelude> :type foldr`
`foldr :: (a -> b -> b) -> b -> [a] -> b`

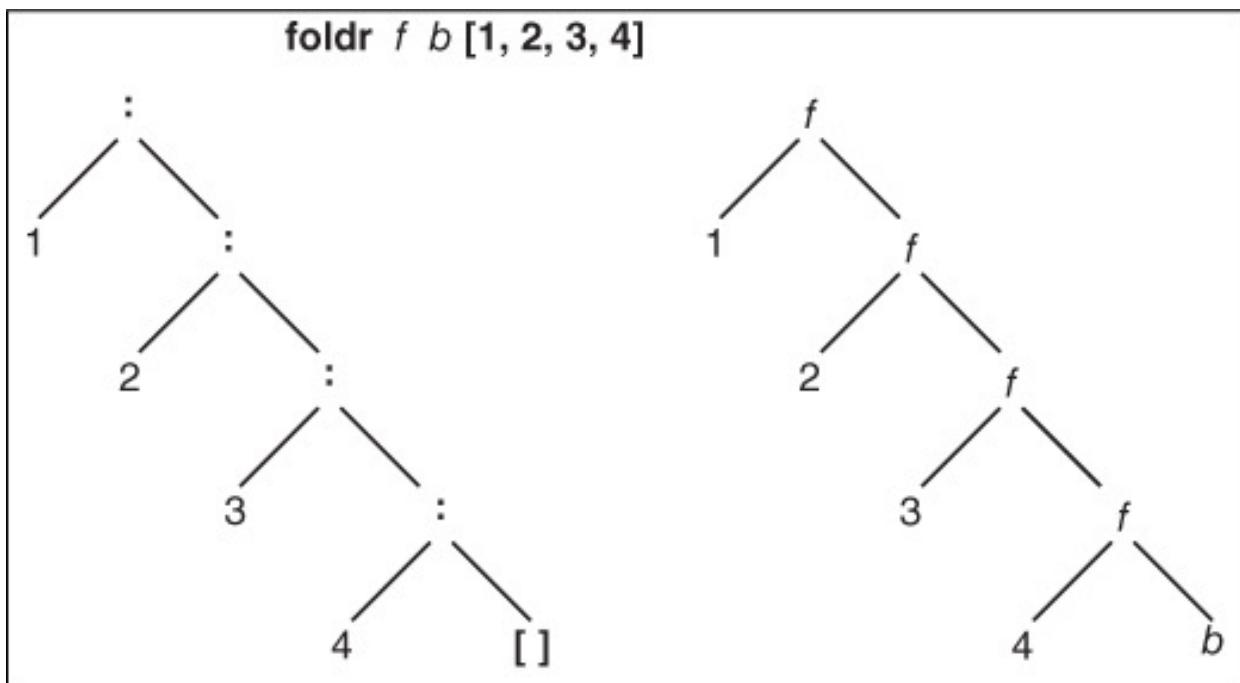
foldr in Haskell

The function `foldr` folds a function, given an initial value, across a list from right to left:

$$\text{foldr} \oplus v [e_0, e_1, \dots, e_n] = e_0 \oplus (e_1 \oplus (\dots (e_{n-1} \oplus (e_n \oplus v)) \dots))$$

where \oplus is a symbol representing an operator.

Figure 8.1 foldr Using the Right-associative : cons Operator



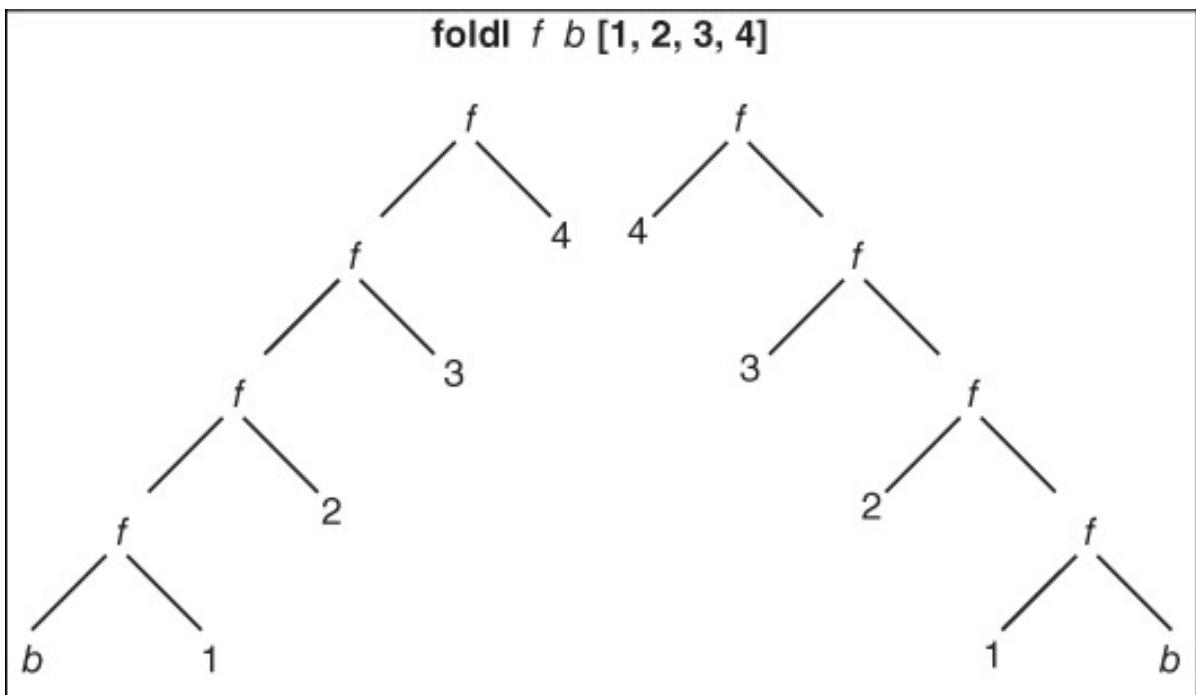
foldl in Haskell

The function `foldl` folds a function, given an initial value, across a list from left to right:

$$\text{foldl } \oplus v [e_0, e_1, \dots, e_n] = ((\dots ((v \oplus e_0) \oplus e_1) \dots) \oplus e_{n-1}) \oplus e_n$$

where \oplus is a symbol representing an operator. Notice that the initial value v appears on the left-hand side of the operator with `foldl` and on the right-hand side with `foldr`.

Figure 8.2 foldl in Haskell (left) Vis-à-Vis foldl in ML (right)



foldr in ML

- The types of foldr in ML and Haskell are the same.

```
- foldr;  
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

```
Prelude> :type foldr  
foldr :: (a -> b -> b) -> b -> [a] -> b
```

- foldr has the same semantics in ML and Haskell.

```
- foldr (op ~) 0 [1,2,3,4]; (* 1-(2-(3-(4-0))) *)  
val it = ~2 : int
```

```
Prelude> foldr (-) 0 [1,2,3,4] -- 1-(2-(3-(4-0)))  
-2
```

foldl in ML

- The types of `foldl` in ML and Haskell differ:

```
- foldl;  
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

```
Prelude> :type foldl  
foldl :: (a -> b -> a) -> a -> [b] -> a
```

- `foldl` has different semantics in ML and Haskell.
- In ML, the function `foldl` is computed as follows:

$$\text{foldl } \oplus v [x_0, x_1, \dots, x_n] = x_n \oplus (x_{n-1} \oplus (\dots \oplus (x_1 \oplus (x_0 \oplus v)) \dots))$$

- Unlike in Haskell, `foldl` in ML is the same as `foldr` in ML (or Haskell) with a reversed list:

```
- foldl (op -) 0 [1,2,3,4]; (* 4-(3-(2-(1-0))) *)  
val it = 2 : int  
- foldr (op -) 0 [4,3,2,1]; (* 4-(3-(2-(1-0))) *)  
val it = 2 : int
```

```
Prelude> foldr (-) 0 [4,3,2,1] -- 4-(3-(2-(1-0)))  
2
```

8.4.5 Crafting Cleverly Conceived Functions with Curried HOFs

- Curried HOFs are powerful programming abstractions that support the definition of functions succinctly.
- We demonstrate the construction of the following three functions using curried HOFs:
 - `implode`: a list-to-string conversion function (online Appendix B)
 - `string2int`: a function that converts a string representing a non-negative integer to the corresponding integer
 - `powerset`: a function that computes the powerset of a set represented as a list

implode (1 of 3)

Consider the following explode and implode functions from online Appendix B:

```
- explode;  
val it = fn : string -> char list  
- explode "apple";  
val it = [#"a",#"p",#"p",#"l",#"e"] : char list  
- implode;  
val it = fn : char list -> string  
- implode [#"a", #"p", #"p", #"l", #"e"];  
val it = "apple" : string  
- implode (explode "apple");  
val it = "apple" : string
```

implode (2 of 3)

- We can define `implode` using HOFs:

```
- val implode = foldr (op ^) #"";  
stdIn:1.29-1.31 Error: character constant not length 1
```

- However, the string concatenation operation `^` only concatenates strings, and not characters:

```
- "hello " ^ "world";  
val it = "hello world" : string  
- #"h" ^ #"e";  
stdIn:6.1-6.12 Error: operator and operand don't agree  
[tycon mismatch]  
    operator domain: string * string  
    operand:          char * char  
in expression:  
  #"h" ^ #"e"
```

- We need a helper function that converts a value of type `char` to value of type `string`:

```
- str;  
val it = fn : char -> string
```

implode (3 of 3)

Now we can use the HOFs foldr, map, and \circ (i.e., functional composition) to compose the atomic elements:

```
- (* parentheses unnecessary, but present for clarity *)
- val implode = (foldr op ^ "") o (map str);
val implode = fn : char list -> string
- val implode = foldr op ^ "" o map str;
val implode = fn : char list -> string
- implode [#"a", #"p", #"p", #"l", #"e"];
val it = "apple" : string
- foldr op ^ "" (map str [#"a", #"p", #"p", #"l", #"e"]);
val it = "apple" : string
- foldr op ^ "" ["a", "p", "p", "l", "e"];
val it = "apple" : string
- "a" ^ ("p" ^ ("p" ^ ("l" ^ ("e" ^ ""))));
```

val it = "apple" : string

Notice: The functions `map` and `foldr` (and `foldl`) are defined in curried form in ML.

string2int (1 of 4)

- Let's implement a function that converts a string representing a nonnegative integer into the equivalent integer.

```
- (* has type string -> int *)  
  
- string2int "0"  
0  
- string2int "123"  
123  
- string2int "321"  
321  
- string2int "5643452"  
5643452
```

- We can use `explode` to decompose a string into a list of chars.
- We must recognize that, for example, $123 = (3 + 0) + (2 * 10) + (1 * 100)$.
- We start by defining a function that converts a `char` to an `int`:

```
- fun char2int c = ord c - ord #"0";  
val char2int = fn : char -> int
```

string2int (2 of 4)

- Now we can define another helper function that invokes `char2int` and acts as an accumulator for the integer being computed:

```
- fun helper(c, sum) = char2int c + 10*sum;  
val helper = fn : char * int -> int
```

- We are now ready to glue the elements together with `foldl`:

```
(* helper (#"3", helper (#"2", helper (#"1", 0))) *)  
- foldl helper 0 (explode "123");  
val it = 123 : int
```

string2int (3 of 4)

- Since we use `foldl` in ML, we can think of the characters of the reversed string as being processed from right to left.
- The function `helper` converts the current character to an `int` and then adds that value to the product of 10 times the running sum of the integer representation of the characters to the right of the current character:

```
- foldl helper 0 (explode "123");
val it = 123 : int
- foldl helper 0 [#"1",#"2",#"3"];
val it = 123 : int
- helper(#"3", helper(#"2", helper(#"1", 0)));
val it = 123 : int
- foldl (fn (c, sum) => char2int c + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int
- foldl (fn (c, sum) =>
          ord c - ord #"0" + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int
```

string2int (4 of 4)

- Thus, we have:

```
- fun string2int s = foldl helper 0 (explode s);
val string2int = fn : string -> int
```

- After inlining an anonymous function for helper, the final version of the function is:

```
- fun string2int s =
  foldl (fn (c, sum) => ord c - ord #"0" + 10*sum) 0 (explode s);
val string2int = fn : string -> int
- string2int "0";
val it = 0 : int
- string2int "1";
val it = 1 : int
- string2int "123";
val it = 123 : int
- string2int "321";
val it = 321 : int
- string2int "5643452";
val it = 5643452 : int
```

powerset (1 of 2)

The following code from online Appendix B is the definition of a powerset function:

```
$ cat powerset.sml
fun powerset(nil) = [nil]
| powerset(x::xs) =
  let
    fun insertineach(_, nil) = nil
    | insertineach(item, x::xs) =
        (item::x)::insertineach(item, xs);
    val y = powerset(xs)
  in
    insertineach(x, y)@y
  end;
$ 
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
```

powerset (2 of 2)

Using the HOF map, we can make this definition more succinct:

```
$ cat powerset.sml
fun powerset nil = [nil]
|   powerset (x::xs) =
  let
    val temp = powerset xs
  in
    (map (fn excess => x::excess) temp) @ temp
  end;
$ 
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
- powerset [1];
val it = [[1], []] : int list list
- powerset [1,2];
val it = [[1,2],[1], [2], []] : int list list
- powerset [1,2,3];
val it = [[1,2,3],[1,2], [1,3], [1], [2,3], [2], [3], []] : int list list
```

Use of the built-in HOF map in this revised definition obviates the need for the nested helper function `insertineach`.

Summary

- Higher-order functions support the capture and reuse of a pattern of recursion or, more generally, a pattern of control.
- Curried HOFs provide the glue that enables programmers to compose reusable atomic functions together in creative ways.
- The resulting functions can be used in concert to craft a malleable/reconfigurable program.
- What results is a general set of (reusable) tools resembling an API rather than a monolithic program.
- This style of modular programming makes programs easier to debug, maintain, and reuse (Hughes 1989).

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- **8.5 Analysis**
- 8.6 Thematic Takeaways

8.5 Analysis (1 of 3)

- Higher-order functions capture common, typically recursive, programming patterns as functions.
- When HOFs are curried, they can be used to automatically define atomic functions—rendering the HOFs more powerful.
- Curried HOFs help programmers define functions in a modular, succinct, and easily modifiable/reconfigurable fashion.
- In this style of programming, programs are composed of a series of concise function definitions that are defined through the application of (curried) HOFs
 - map;
 - functional composition: \circ in ML and $.$ in Haskell; and
 - foldl/foldr.

8.5 Analysis (2 of 3)

- Programming becomes essentially the process of creating composable building blocks and combining them like LEGO(R) bricks in creative ways to solve a problem.
- The resulting programs are more concise, modular, and easily reconfigurable than programs where each individual function is defined literally (i.e., hardcoded).
- The challenge and creativity require determining the appropriate level of granularity of the atomic functions, figuring out how to automatically define them using (built-in) HOFs, and then combining them using other HOFs into a program so that they work in concert
- Resembles building a library or API more than an application program.
- Focus is more on identifying, developing, and using the appropriate higher-order abstractions than on solving the target problem.

8.5 Analysis (3 of 3)

- Once the abstractions and essential elements have crystallized, solving the problem at hand is an afterthought.
- The pay-off, of course, is that the resulting abstractions can be reused in different arrangements in new programs to solve future problems.

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- **8.6 Thematic Takeaways**

8.6 Thematic Takeaways

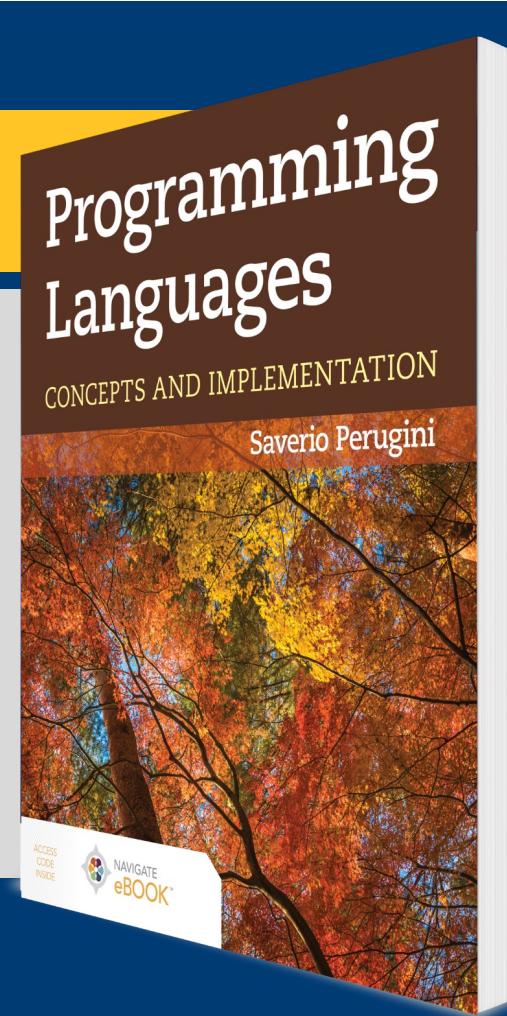
- First-class, lexical closures are an important primitive construct for creating programming abstractions (e.g., partial function application and currying).
- Higher-order functions capture common, typically recursive, programming patterns as functions.
- Currying a higher-order function enhances its power because such a function can be used to automatically define new functions.
- Curried, higher-order functions also provide the glue that enables you to combine these atomic functions to construct more complex functions.

HOFs + Currying Concise Functions + Reconfigurable Programs

HOFs + Currying (Curried HOFs) ↗ Modular Programming

CHAPTER 9

Data Abstraction



Copyright © 2023 by Jones & Bartlett Learning, LLC an Ascend Learning Company. www.jblearning.com.

Chapter 9: Type Systems and Data Abstraction

Reimplementing [familiar] algorithms and data structures in a significantly different language often is an aid to understanding of basic data structure and algorithm concepts.

—Jeffrey D. Ullman, Elements of ML Programming (1997)

Type systems support data abstraction and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types.

Outline

- **9.1 Chapter Objectives**
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.1 Chapter Objectives

- Introduce *aggregate data types* (e.g., arrays, records, unions) and type systems supporting their construction in a variety of programming languages.
- Introduce *inductive data types*—an aggregate data type that refers to itself—and *variant records*—a data type useful as a node in a tree representing a computer program.
- Introduce *abstract syntax* and its role in representing a computer program.
- Describe the design, implementation, and manipulation of efficacious and efficient data structures representing computer programs.
- Explore the conception and use of a data structure as an *interface*, *implementation*, and *application*, which render it an *abstract data type*.
- Recognize and use a *closure representation* of a data structure.
- Describe the design and implementation of data structures for language environments using a variety of representations.

Outline

- 9.1 Chapter Objectives
- **9.2 Aggregate Data Types**
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.2 Aggregate Data Types

- 9.2.1 Arrays
- 9.2.2 Records
- 9.2.3 Undiscriminated Unions
- 9.2.4 Discriminated Unions

9.2 Aggregate Data Types: Arrays and Records

- An *array* is an aggregate data type indexed by integers:

```
/* declaration of integer array scores */
int scores[10];
/* use of integer array scores */
scores[0] = 97;
scores[1] = 98;
```

- A *record* (also referred to as a *struct*) is an aggregate data type indexed by strings called *field names*:

```
/* declaration of struct employee */
struct {
    int id;
    double rate;
} employee;
/* use of struct employee */
employee.id = 555;
employee.rate = 7.25;
```

9.2.3 Undiscriminated Unions

An *undiscriminated* union is an aggregate data type that can only store a value of one of multiple types (i.e., a union of multiple types):

```
/* declaration of an undiscriminated union int_or_double */
union {
    /* C compiler only allocates memory for the largest */
    int id;
    double rate;
} int_or_double;
int main() {
    /* use of union int_or_double */
    int_or_double.id = 555;
    int_or_double.rate = 7.25;
}
```

9.2.4 Discriminated Unions

- A *discriminated union* is a record containing a union as one field and a flag as the other field.
- The flag indicates the type of the value currently stored in the union.

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- **9.3 Inductive Data Types**
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.3 Inductive Data Types

- An *inductive data type* is an aggregate data type that refers to itself.
- The type being defined is one of the constituent types of the type being defined.
- A node in a singly linked list is a classical example of an inductive data type.
- The node contains some value and a pointer to the next node, which is also of the same node type:

```
struct node_tag {  
    int id;  
    struct node_tag* next;  
};  
struct node_tag head;
```

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- **9.4 Variant Records**
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.4 Variant Records

- A *variant record* is an aggregate data type that is a union of records (i.e., a union of structs).
- It can hold any one of a variety of records.
- Each constituent record type is called a *variant* of the union type.
- Variant records can also be inductive.

Variant Record Example

Consider the following EBNF definition of a list:

```
<L>    ::=  <A>
<L>    ::=  <A><L>
<A>    ::=  -231 | ... | 231 - 1 (i.e., ints)
```

```
1  /* a variant record: a union of structs */
2  #include<stdio.h>
3
4  int main() {
5
6      typedef union llist_tag {
7
8          /* <L> ::= <A> variant */
9          struct aatom_tag {
10              int number;
11          } aatom;
12
13 }
```

9.4.1 Variant Records in Haskell

- *Type systems* support *data abstraction* and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types.
- A *type system* of a programming language includes the mechanism for creating new data types from existing types.
- ML and Haskell each have a powerful type system.

Table 9.1 Support for C/C++ Style structs and unions in ML, Haskell, Python, and Java

Data Type	C/C++	ML	Haskell	Python	Java
records	struct	type	type	class	class
unions/variant records	union	datatype	data	class	class

9.4.2 Variant Records in Scheme: `(define-datatype ...)` and `(cases ...)`

- Unlike ML and Haskell, Scheme does not have built-in support for defining and manipulating variant records, so we need a tool for these tasks in Scheme.
- The `(define-datatype ...)` and `(cases ...)` extensions to Racket Scheme created by Friedman, Wand, and Haynes (2001) provide support for constructing and deconstructing respectively, variant records in Scheme.

(define-datatype . . .)

- The (define-datatype . . .) form defines variant records.

```
#lang eopl
(define-datatype llist llist?
  (atom
    (aatom_tag number?))
  (aatom_llist
    (aatom_tag number?)
    (next llist?)))
```

Syntax:

```
(define-datatype <type-name> <type-predicate-name>
  {(<variant-name> {(<fieldname> <predicate>) }*)}+)
```

- A new function called a *constructor* is automatically created for each variant to construct data values belonging to that variant.
- Data types can be mutually-recursive (e.g., recall grammar for S-expressions)

A List of Integers (1 of 2)

This definition automatically creates a linked list variant record and an *implementation* of the following *interface*:

- A unary function `aatom`, which creates an atom node
- A binary function `aatom_llist`, which creates an atom list node
- A binary predicate `llist?`

(cases ...)

- The (cases ...) form, in the EOPL extension to Racket Scheme, provides support for decomposing and manipulating the constituent parts of a variant record created with the constructors automatically generated with the (define-datatype ...) form.

Syntax:

```
(cases <type-name> <expression>
  {(<variant-name> ({<field-name>}*) <consequent>) }*
  (else <default>))
```

- (cases ...) provides a convenient way to manipulate data types created with (define-datatype ...)
- Can think of (cases ...) as pattern matching (values bound to symbols)

A List of Integers (2 of 2)

The following function accepts a value of type `llist` as an argument and manipulates its fields with the `cases` form to sum its nodes:

```
(define llist_sum
  (lambda (ll)
    (cases llist ll
      (aatom (aatom_tag) aatom_tag)
      (aatom_llist (aatom_tag next)
        (+ aatom_tag (llist_sum next))))))
  > (llist_sum ouraatom)

3
> (llist_sum ouraatom_llist)
5
> (llist_sum ourllist)
6
```

Table 9.2 Support for Composition (Definition) and Decomposition (Manipulation) of Variant Records in a Variety of Programming Languages

Language	Composition	Decomposition
C	union of structs with flag	switch (...) { case ... }
C++	union of structs with flag	switch (...) { case ... }
Java	class with flag	switch (...) { case ... }
Python	class with flag	if/elif/else
ML	datatype	pattern-directed invocation
Haskell	data	pattern-directed invocation
Racket Scheme	define-datatype form	cases form

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- **9.5 Abstract Syntax**
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.5 Abstract Syntax (1 of 5)

- Consider the string `((lambda (x) (f x)) (g y))` representing an expression in λ -calculus.
- This program string is an external representation (i.e., it is external to the system processing it) and uses *concrete syntax*.
- Programs in concrete syntax are not readily processable.

9.5 Abstract Syntax (2 of 5)

Notably, the preceding program is more manipulable and, thus, processable when represented using the following definition of an expression data type:

```
(define-datatype expression expression?
  (variable-expression
    (identifier symbol?))
  (lambda-expression
    (identifier symbol?)
    (body expression?))
  (application-expression
    (operator expression?)
    (operand expression?)))
```

9.5 Abstract Syntax (3 of 5)

- An *abstract-syntax tree* (AST) is similar to a parse tree, except that it uses abstract syntax or an *internal representation* (i.e., it is internal to the system processing it) rather than *concrete syntax*.
- While the structure of a parse tree depicts how a sentence (in concrete syntax) conforms to a grammar:
 - the structure of an abstract-syntax tree illustrates how the sentence is represented internally,
 - typically with an inductive, variant record data type.
- Figure 9.1 illustrates an AST for the λ -calculus expression.

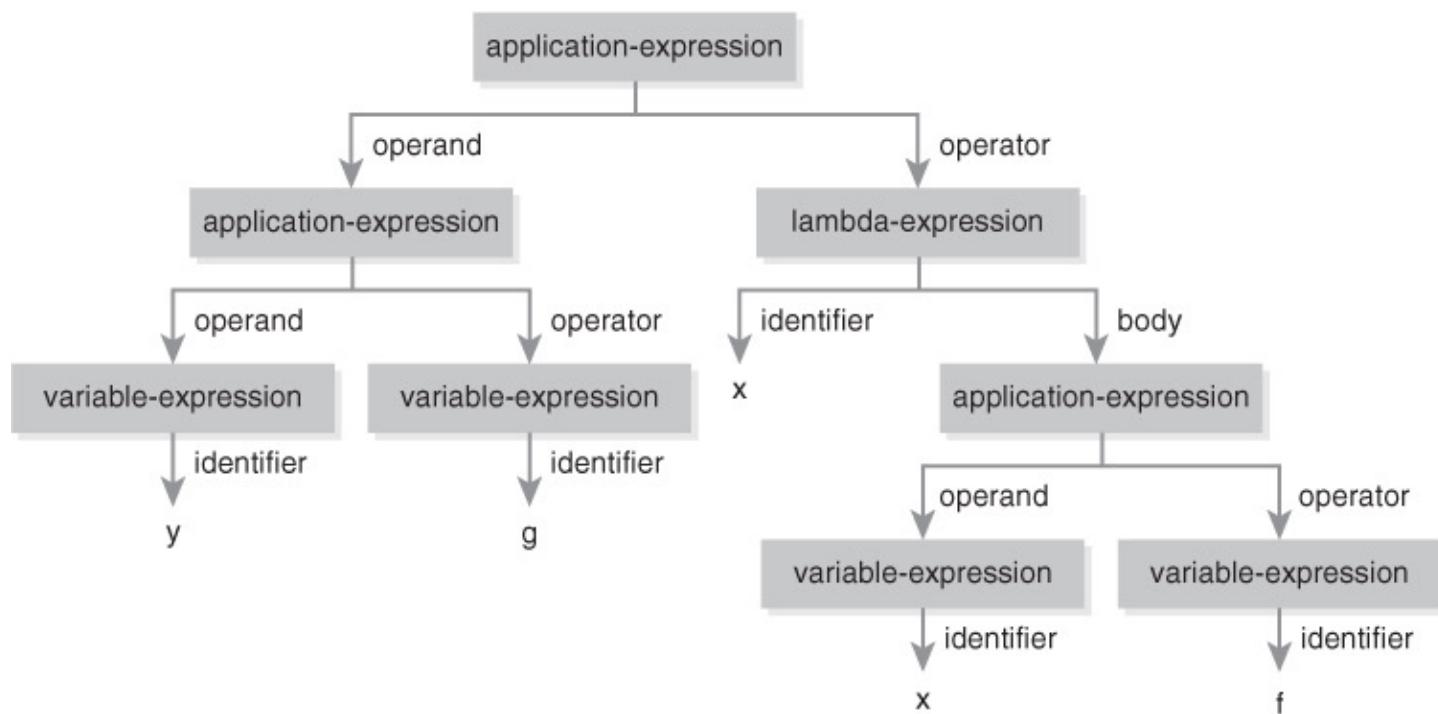
((lambda (x) (f x)) (g y))

9.5 Abstract Syntax (4 of 5)

- Abstract syntax is a representation of a program as a data structure—in this case, an inductive variant record.
- Consider the following grammar for λ -calculus, which is annotated with variants of this expression inductive variant record data type above the right-hand side of each production rule:

```
<expression> ::= <identifier>
                           variable-expression (identifier)
                           lambda-expression (identifier body)
<expression> ::= (lambda (<identifier>) <expression>)
                           application-expression (operator operand)
<expression> ::= (<expression><expression>)
```

**Figure 9.1 Abstract-Syntax Tree for
((lambda (x) (f x)) (g y))**



9.5 Abstract Syntax (5 of 5)

- Simple grammar for λ -calculus expressions revisited
- *Concrete* vs. *abstract* syntax (external vs. internal representation)
- expression datatype
- One-to-one mapping between production rules and constructors

Inductive Data Types and Abstract Syntax Summary

- Discriminated unions
- *Inductive data types* (e.g., *variant record*: a union of structs)
- `define-datatype` constructs inductive data types (specifically, *variant records*)
- `cases` decomposes inductive data types
- *Concrete syntax vis-à-vis abstract syntax*

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- **9.6 Abstract-Syntax Tree for Camille**
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.6 Abstract-Syntax Tree for Camille

A goal of Part II of this text is to establish an understanding of data abstraction techniques so we can harness them in our construction of environment-passing interpreters, for purposes of simplicity and efficiency, in Part III.

9.6.1 Camille Abstract-Syntax Tree Data Type: TreeNode

The following abstract-syntax tree data type `TreeNode` is used in the abstract-syntax trees of Camille programs for our Camille interpreters developed in Part III:

```
41  class Tree_Node:  
42      def __init__(self, type, children, leaf, linenumber):  
43          self.type = type  
44          # save the line number of the node so run-time  
45          # errors can be indicated  
46          self.linenumber = linenumber  
47          if children:  
48              self.children = children  
49          else:  
50              self.children = [ ]  
51              self.leaf = leaf  
52  # end expression data type #
```

Figure 9.2 (left) Visual Representation of `TreeNode` Python class. (right) A Value of Type `TreeNode` for an Identifier

TreeNode	type: node type (e.g., <code>ntNumber</code>)
	leaf: primary data associated with node
	children: list of child nodes
	linenumber: line number in which the node occurs
TreeNode	type: <code>ntIdentifier</code>
	leaf: <code>x</code>
	children: []
	linenumber: <code>l</code>

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- **9.7 Data Abstraction**
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.7 Data Abstraction

Data abstraction involves the conception and use of a data structure as:

- an *interface*, which is implementation-neutral and contains function declarations;
 - an *implementation*, which contains function definitions; and
 - an *application*, which is also *implementation-neutral* and contains invocations to functions in the implementation; the application is sometimes called the *main program* or *client code*.
-
- The underlying implementation can change without disrupting the client code as long as the contractual signature of each function declaration in the interface remains unchanged.
 - In this way, the implementation is *hidden* from the application.
 - A data type developed this way is called an *abstract data type* (ADT).

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- **9.8 Case Study: Environments**
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.8 Case Study: Environments (1 of 4)

- 9.8.1 Choices of Representation
- 9.8.2 Closure Representation in Scheme
- 9.8.3 Closure Representation in Python
- 9.8.4 Abstract-Syntax Representation in Python

9.8 Case Study: Environments (2 of 4)

- A *referencing environment* is a mapping that associates variable names (or symbols) with their current bindings at any point in a program in an implementation of a programming language
- A *symbol table* is an example of an environment.
- A symbol table is used in a compiler to associate variable names with lexical address information.
- An environment is a *mapping* (a set of pairs)
 - *domain*: the finite set of Scheme symbols
 - *range*: the set of all Scheme values
(e.g., $\{(a, 4), (b, 2), (c, 3), (x, 5)\}$).

9.8 Case Study: Environments (3 of 4)

Consider an interface specification of an environment, where formally an environment expressed in the mathematical form

$$\{(s_1, v_1), (s_2, v_2), \dots, (s_n, v_n)\}$$

is a mapping (or a set of pairs) from the *domain*—the finite set of Scheme symbols—to the *range*—the set of all Scheme values:

$$\begin{aligned} (\text{empty-environment}) &= [\emptyset] \\ (\text{apply-environment } [f|s]) &= f(s) \\ (\text{extend-environment } ' (s_1, s_2, \dots, s_n) ' (v_1, v_2, \dots, v_n) [f]) &= [g], \end{aligned}$$

where $g(s') = v_i$ if $s' = s_i$ for some i , $1 \leq i \leq n$, and $f(s')$ otherwise;
[v] means “the representation of data v .”

9.8 Case Study: Environments (4 of 4)

- The environment $\{(a, 4), (b, 2), (c, 3), (x, 5)\}$ may be constructed and accessed with the following client code:

```
> (define simple-environment
  (extend-environment '(a b) '(1 2)
    (extend-environment '(c d e) '(3 5 5)
      (empty-environment))))
> (apply-environment simple-environment 'e)
5
```

- Constructors* create: empty-environment and extend-environment
- Observers* extract: apply-environment

9.8.1 Choices of Representation

- We consider the following representations for an environment:
 - Data structure representation (e.g., lists)
 - Abstract-syntax representation (ASR)
 - Closure representation (CLS)

9.8.2 Closure Representation in Scheme (1 of 2)

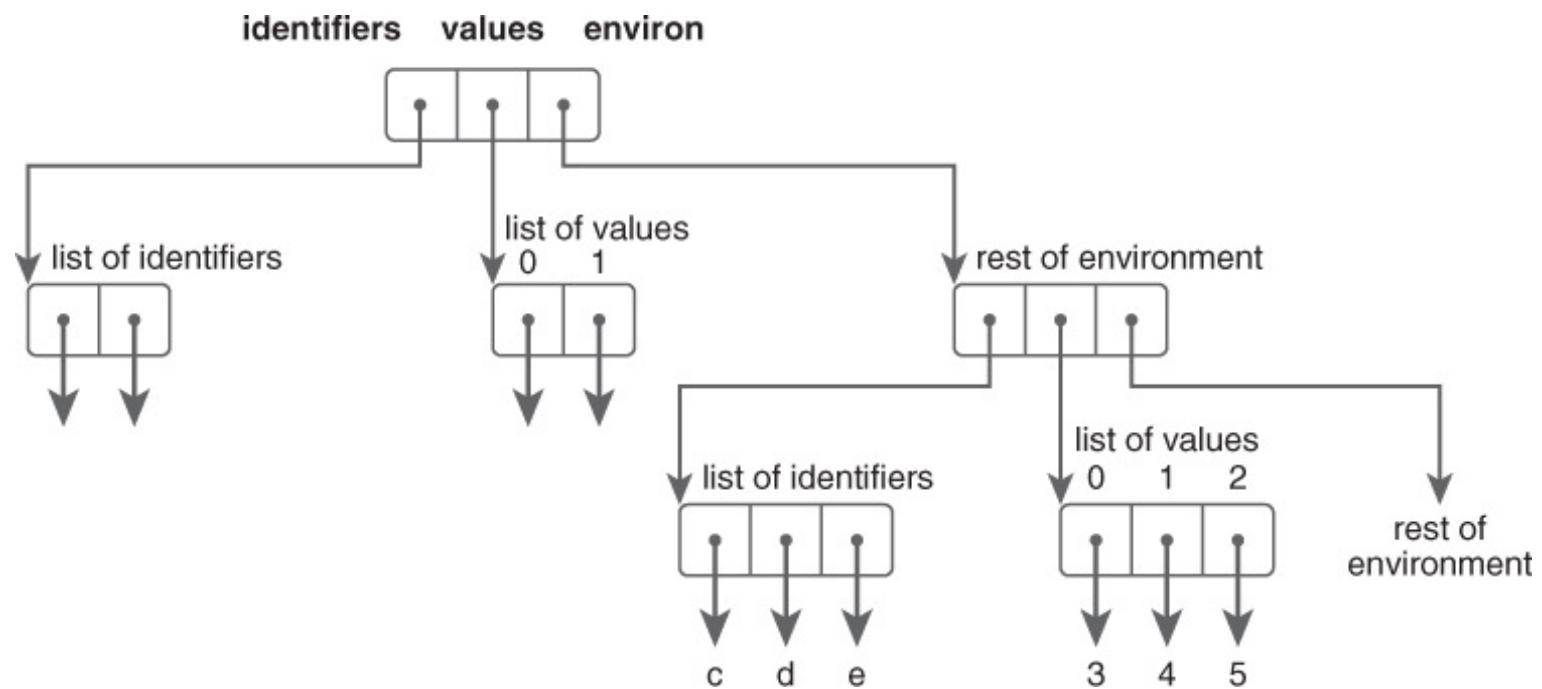
- Often the set of *values* of a data type can be advantageously *represented* as a set of *functions*, particularly when the abstract data type has multiple constructors but only a single observer.
- Moreover, languages with first-class functions facilitate use of a closure representation.
- Representing a data structure as a function—here, a closure—is a non-intuitive use of functions, because we do not typically think of data as code.
- How can we represent an environment (which we think of as a data structure) as a function?
- The most natural closure representation for the environment is a Scheme closure that accepts a symbol and returns its associated value.

9.8.2 Closure Representation in Scheme (2 of 2)

- Getting acclimated to the reality that the data structure is a function can be a cognitive challenge.
- Step through the evaluation of the following application code:

```
1 > (define simple-environment
2     (extend-environment '(a b) '(1 2)
3         (extend-environment '(c d e) '(3 4 5)
4             (empty-environment))))
5
6 > (apply-environment simple-environment 'e)
7 5
```

Figure 9.3 An Abstract-Syntax Representation of a Named Environment in Python



Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- **9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis**
- 9.10 Thematic Takeaways

9.9.3 Comparison of ML and Haskell

Haskell	= ML + Lazy Evaluation	- Side Effects
ML	= Lisp - Homoiconicity	+ Safe Type System
Haskell	= Lisp - Homoiconicity	+ Safe Type System
	- Side Effects	+ Lazy Evaluation

Table 9.7 Comparison of the Main Concepts and Features of ML and Haskell

Concept	ML	Haskell
lists	homogeneous	homogeneous
cons	::	:
append	@	++
integer equality	=	==
integer inequality	<>	/=
strings	not a list of characters use <code>explode</code>	a list of Characters
renaming parameters	<code>lst as (x :: xs)</code>	<code>lst@ (x : xs)</code>
functional redefinition	permitted	not permitted
pattern-directed invocation	yes, with	yes
parameter passing	call-by-value, strict, applicative-order evaluation	call-by-need, non-strict, normal-order evaluation
functional composition	o	.
infix to prefix	(op operator)	(operator)
sections	not supported	supported, use (operator)
prefix to infix		'operator'
user-defined functions	introduced with <code>fun</code> can be defined at the prompt or in a script	must be defined in a script
anonymous functions	<code>(fn tuple => body)</code>	<code>(\ tuple -> body)</code>
curried form	omit parentheses, commas	omit parentheses, commas
curried	partially	fully
type declaration	:	::
type definition	<code>type</code>	<code>type</code>
data type definition	<code>datatype</code>	<code>data</code>
type variables	prefaced with ' written before data type name	written after data type name
function type	optional, but if used, embedded within function definition	optional, but if used, precedes function definition
type inference/checking	Hindley-Milner	Hindley-Milner
function overloading	not supported	supported through qualified types and type classes
ADTs	module system (structures, signatures, and functors)	class system

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- **9.10 Thematic Takeaways**

9.10 Thematic Takeaways (1 of 2)

- A goal of a *type system* is to support *data abstraction* and, in particular, the definition of *abstract data types* that have the properties and behavior of primitive types.
- An inductive *variant record* data type—a union of records—is particularly useful for representing an *abstract-syntax tree* of a computer program.
- Data types and the functions that manipulate them are natural reflections of each other.
- The conception and use of an *abstract data type* data structure are distributed among an implementation-neutral *interface*, an *implementation* containing function *definitions*, and an *application* containing invocations to functions in the implementation.

9.10 Thematic Takeaways (2 of 2)

- The underlying representation/implementation of an abstract data type can change without breaking the application code as long as the contractual signature of each function declaration in the interface remains unchanged.
- In this way, the implementation *hidden* from the application.
- A variety of representation strategies for data structures are possible, including list, abstract syntax, and closure representations.
- Well-defined data structures as abstract data types are an essential ingredient in the implementation of a programming language (e.g., interpreters and compilers).
- A programming language with an expressive type system is indispensable for the construction of efficacious and efficient data structures.

More on Functional Programming

It's still programming

- Lots in common with imperative programming, of course
 - Issues of naming, scoping, types, expressions, control flow still arise
 - All languages must be scanned, parsed, etc
 - Functional languages make very heavy use of subroutines
 - Concurrency & nondeterminacy apply to these languages as much as imperative languages (concurrency is often much easier, for reasons we'll see later)
- Boundaries tend to blur a bit
 - Functional features being added to imperative languages
 - Python can be written in a largely functional style
 - C# supports monads, a key feature of functional programming.
 - Functional features being added to VB, Java 8, even C++
 - Third-party library adds some functional features to FORTRAN
 - The most common logic language (Prolog) offers some imperative features
 - So do most LISP descendants
 - It's straightforward to build a logic programming system in most functional languages

Pure functions

- In most functional languages, functions are *pure*—they have no side effects, always return the same value for a given set of parameters, and do not depend on anything that isn't passed as a parameter. This leads to some useful features:
 - Lazy evaluation
 - If it's more convenient to leave an expression unevaluated, I can do that; it'll have the same value whether I evaluate it now, or next Tuesday. Thus I can wait until I need it (if ever).
 - Memoization
 - Likewise, once I've evaluated it for a given set of parameters, I can store the value and look it up next time; it's not going to change if the server slows down, the vendor changes, etc.
 - Concurrency is trivial
 - If I have 100 function evaluations pending, and 100 processors, I can parallelize them completely; none depend on the others or can affect the others.
 - This allows divide-and-conquer, partial accumulation, associativity

Higher-order functions: Example

- Map function
 - `(map f L)` applies function `f` to each member of list `L`
 - `(map sqr '(1 2 3))` returns the list `(1 4 9)`
- Function as parameter to handle errors:
 - `(define (safe-div numer denom if-error)
#error handling function as parameter
 (if (= denom 0)
 (if-error) # divide by 0, call error function
 (quotient numer denom)))
otherwise it's safe, return int quotient`

Customizing functions

- We can have complex library functions and customize them to what we need:
- ```
(define (generic-safe-div numer denom if-err if-success)
 (if (= denom 0)
 (if-err)
 (if-success (div numer denom))))
```
- ```
(define (my-safe-div numer denom)
  (generic-safe-div numer denom my-if-err my-if-succ))
```
- This is an example of *partial application*. We apply some of the parameters of a more complex function to produce a simpler function.
- Obviously, we can define multiple versions of my-if-err and my-if-succ, producing the exact behavior we want from my-safe-div. Or several different versions, depending on our needs.

Currying

- Named after the mathematician Haskell Curry
- Allows a multi-parameter function call to be treated as a series of 1-parameter calls.

```
(define (F x y z)
  # function body
)
# Suppose x = 5, y = 3.2, z = "ABC"
(define (f1 y z)
  (F 5 y z))  # f1 is a function of y & z
(define (f2 z)
  (f1 3.2 z))  # f2 is a function of z
(f2 "ABC")
```

- Obviously, we could have started with currying z, then y, leaving us with a function of x.
- This is useful in building a ‘custom’ version of a function
- Also, the map function requires a 1-parameter function passed to it. This lets us build one.

Other higher-order functions

- Among the most common:
- (`(fold f s L)`): `f` is a function that takes 2 parameters. The value `s` and the first item in `L` are passed to `f`; the result is passed with the second element of `L`, and so on. Typical uses: sum, min, max, etc.
 - `(define (sum L) (fold + 0 L))`
`(define (product L) (fold * 1 L))`
- (`(filter f L)`): Return a new list, containing all elements of `L` for which `f` returns true.

Evaluation of function parameters

- Some languages use *normal order* - this is also known as *lazy evaluation*.
 - Function parameters are not evaluated until they are needed; once evaluated, they are memoized.
 - If a parameter is undefined, but also not needed—no harm, no foul.
 - This also allows defining a potentially infinite data structure—a list of all integers, for example. It only grows as big as the number of times the ‘next’ item is requested.
- Other (most) languages use *applicative order*—all parameters are evaluated before the function begins. Undefined parameters are a runtime error.
- Some languages allow a choice—generally evaluation can be forced if a language uses lazy evaluation, and a normal-order (lazy evaluation) version of Racket is available, though it usually uses applicative order.

Strictness & lazy evaluation

- Evaluation order can affect execution speed, but also program correctness
 - A program encountering a dynamic semantic error or infinite regression in an ‘unneeded’ subexpression under applicative-order evaluation may run successfully under normal-order evaluation
- A side-effect-free function is said to be strict if it is undefined (doesn’t terminate or encounters an error) if any of its parameters are undefined. Such a function can evaluate all of its arguments, so can safely use applicative order (results won’t depend on evaluation order)
- A function is nonstrict if it doesn’t impose this requirement –if it is sometimes defined even if some arguments aren’t.
- A language is strict if it’s defined such that functions are always strict; a language is nonstrict if it allows nonstrict functions
- If a language always evaluates in applicative order, then it is strict, since a call with an undefined argument will result in an error.
- Contrapositively, a nonstrict language cannot use applicative order; it must use normal order to avoid evaluating unneeded arguments
 - Standard ML, OCaml, and Scheme are strict; Miranda and Haskell are nonstrict

Implementation issues

- Trivial Update Problem
 - Suppose we have a data structure with 100K+ items in it. We need to change one of them.
 - In a procedural language, this is no problem; we identify the item & update it; or delete the old value & insert the new one. These change the data structure.
 - But in a functional language, data (including aggregated data) is immutable!
 - To handle this functionally, we return a new struct with the item added. How can this be done in a memory-efficient manner?
 - In practice, some functional languages (such as Racket) do allow procedural-style interactions with the (set! Item collection) function. But this loses the benefit of functional-style programming.
 - This can be mitigated significantly by careful choice of data structures, but this must be considered at language implementation time

Dealing with the real world

- In functional languages, functions have no side effects and always return the same result from the same parameters.
- But the real world is messy.
 - Each call to `read_data()` is expected to return a new item, and it doesn't have to be the same as the one before it; and sooner or later, it'll probably fail because of end-of-file.
 - Side effects include things such as:
 - Mark the invoice as paid
 - Update the database
 - Display the new score
 - Play the sound effect
 - Send the email
 - Print the document
 - Our clients just call these *effects*, and consider them to be the point of the program.
- Thus, we have to make some compromises to deal with the real world. But we can contain the state within specific types of containers.

Real world operations

- We may have:
 - An operation that might fail (Maybe)
 - An operation that might return different types of data on success or failure (e.g., either a record, or an error message). (Either)
 - An operation that we might need to wait to complete (Async)
 - Retrieving data that the user hasn't typed in yet, or that we have to retrieve from a drive (Reader)
 - Data that needs to be sent to a display, which is a side effect (Writer)
 - A global context that a function might affect (State)
 - Or many, many other things.
- We're going to look at the simplest 2: Maybe and Either.
 - Some languages use the name Option instead of Maybe.

Why do we do this?

- This allows our functions to remain pure.
- Instead of “this function returns a number, unless an error happens, in which case it raises an exception,” we have “this function returns a Maybe.”
- Instead of “this function returns the requested customer record, or an error message, or an exception from the database,” we have “this function returns an Either: On success, the requested customer record; on failure, a list of one or more error messages (strings).”
- Thus, we have a consistent interface, and can sequence our operations reliably.
- State and side effects are encapsulated in our container

Elevated worlds

- We are *not* going to go into the mathematical theory behind these.
- They're called *monads*. For programming-language purposes*, a monad requires 3 things:
 - A container to hold the data
 - A function (often called *lift*, or *pure*, or *return*) that moves data into those containers
 - A function (usually called *bind* or *chain*) for combining functions that take 'normal' data and return 'elevated' data (that is, data in a container)
- Our running example will be integer division.
- **Mathematically, a monad is a monoid in the category of endofunctors. What's the problem?*

Maybe

- Think of *Maybe* as a box. It might contain a value, it might contain nothing.
- We need a way to put data into the box. If x is a number, it's just the value of x ; otherwise it's nothing:

```
(define (lift-maybe x)
  (if (number? x)
      (just x)
      nothing))
```

- We can rewrite our division function to return a Maybe:
- ```
(define (safe-div x y)
 (if (= y 0)
 nothing
 (just (quotient x y))))
```
- And yes, we could add logic to check they're both numbers (or both integers) first, and return nothing if either test fails
- What's the advantage? It's a pure function again. It no longer returns perhaps a number, perhaps nothing. It always returns a Maybe.

# ‘world-crossing’ functions

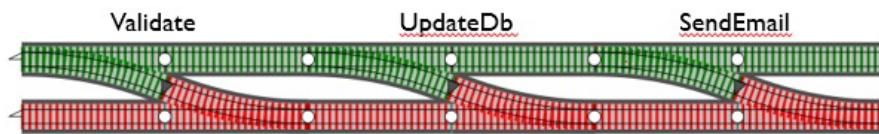
- But now we have a problem.
- Safe-div takes in a number and returns a Maybe. Suppose we’re going to pass the result to some other function. That function expects a number, not a Maybe. Do we have to modify that function to take a Maybe as well? If we’ll have to rewrite all our functions, it’s not worth it.
- No, we need a function to *bind* or *chain* function calls together:
- ```
(define (chain f m) ; m = a maybe value
  (if (nothing? m)
      nothing
      (f (from-just 0 m))))
```
- Let’s unpack this....

What's happening here?

- If an earlier operation failed (`m` is `nothing`), then we can't continue; return `nothing` at once.
- If we have a value, we need to remove it from `just` (which is needed to indicate this is a `Maybe` value).
- But, when you're writing general code, it's possible you're calling `from-just` on `nothing`, or something that isn't just a value.
- So we have to specify what to return in that case.
 - Here we return `0`. Returning `#f` is another popular choice.
- If `m` is just a value, pass it to `f`; otherwise pass `0` to `f`.
 - Yes, in this case, we know `m` isn't `nothing`, but the function is written for the general case.

Parallel tracks

- In effect, we have 2 parallel tracks: A success track and a failure track.
- If we ever get back nothing, for whatever reason, the remaining chained functions are bypassed and never called.



- Most languages have syntactic sugar to avoid nesting multiple function calls. For example, suppose x and y in safe-div could be either numbers or expressions, in some convenient notation. We have a function called eval that returns a numeric value (Maybe).

A more powerful division routine

- ```
(define (safe-div-2 x y)
 (do
 [m <- (eval x)]
 [n <- (eval y)])
 (safe-div m n)))
```
- This syntax masks the nested calls to chain. If either call to eval returns nothing, safe-div-2 returns nothing at once; the remaining actions are skipped.

# The Either type

- An Either is similar to a Maybe, except instead of holding just a value or nothing, it definitely holds something.
- That something is labeled as a success or failure. The data type it holds can differ based on that label:
- ```
(define (safe-div x y)
  (if (and (number? x) (number? y))
      (if (= y 0)
          (failure "division by 0 error")
          (success (quotient x y)))
      (failure "safe-div: x or y not a number"))))
```
- Likewise, we can define a failure to hold a list of strings (error messages) if we prefer, or anything else we feel like.

Bind with Either

- (define (chain f e) ; e = an either value
 (if (failure? e)
 e ; return the failure we got, bypass f
 (f (from-success 0 e))))
- In this case, if we're calling f, we know e isn't a failure, so it's a success, so the 0 parameter won't be needed, but the function has the parameter and we have to specify a value, even if it's never used.
- Using these tools, we can move our error handling inside the functions, and so don't need try/except, or elaborate if/else logic to deal with possible errors.

Lists as monads

- Likewise, lists are
 - A container
 - With functions that move data into a list
 - And functions that allow applying single-parameter functions to them (map, filter, etc)
- Therefore lists are monads
- These can be used as building blocks to build more abstract data structures
 - OCaml's popularity in the financial services industry is largely due to its type system, allowing complex behavior & context to be managed reliably

Application: Parser Combinators

- A common application is to build a parser.
- We can use higher-order functions to build up combinations
- So, for example, begin with a function that can parse a single character.
 - Pass in the character to be matched, and the string to be parsed.
 - If the first character matches, return success, the matched character as a singleton list, and the rest of the input string
 - Otherwise return a failure, error message in a singleton list, and the unchanged input.
 - Note the pattern: Success or failure, and a list consisting of a singleton sublist, and the remaining input as a string

Parse one character

```
(define (parse-specific-char ch input-str)
  (let ([chars (string->list input-str)])
    (if (eq? ch (first chars))
        (success (list (list (first chars)) (list->string (rest chars))))
        (failure (list (list "not a match with" ch) input-str)))))
```

- But of course, we want to be able to match more than one character. We need to be able to match any of a group:

```
(define (any opt-list input-str)
  ; returns which character in opt-list, if any, first char of input-str matches
  (let ([chars (string->list input-str)]) ; turn input-str into list of chars
    (define (iter opt-list ch)
      (if (empty? opt-list)
          (failure (list '("no match") input-str))
          (if (eq? (first opt-list) ch)
              (success (list (list ch) (list->string (rest chars))))
              (iter (rest opt-list) ch)))))
    (iter opt-list (first chars))))
```

Parsing character types

```
(define (parse-decimal-digit input-str)
  (any (string->list "1234567890")))
```

- We can also combine types; another higher-level function will make that easier:

```
(define (either-or f1 f2 input)
  (let ([result (f1 input)])
    (if (success? result)
        result
        (f2 input))))
```

- Apply f1; return result if successful; if not successful, apply f2.
- So once we have (parse-alphabetic) and (parse-numeric) functions,

```
(define (parse-alphanumeric input)
  (either-or parse-alphabetic parse-numeric input))
```

Sequencing operations

- We may have some constructs that we expect to be something followed by a slightly different something
 - “An identifier is an alphabetic character or underscore, followed by any number of alphanumeric or underscore characters.”
- For this we need a higher-order function (and-then f1 f2 input)
 - Apply f1. If f1 fails, return failure immediately.
 - If f1 succeeds, apply f2
 - If f2 succeeds, return success
 - If f1 succeeds and f2 fails, is that good enough? Sometimes yes (an identifier can be a single alphabetic character), sometimes no (in Pascal, a pointer dereference operator ^ must be followed by an identifier).
 - Add parameter to the function whether success on f2 is required.

Combining operations = clear code

- Once we have the tools to connect things, the code becomes simple:

```
(define (parse-alphanumeric-char input)
  (either-or parse-alphabetic-char parse-numeric-char input))
(define (alphanumeric-string input-str)
  (many parse-alphanumeric-char input-str))
(define (alpha-string input-str)
  (many parse-alphabetic-char input-str))
(define (add-op input-str)
  (any (string->list "+-") input-str))
(define (nonzero-digit input-str)
  (any (string->list "123456789") input-str))
(define (decimal-digit input-str)
  (any (string->list "0123456789") input-str))
(define (hex-digit input-str)
  (any (string->list "0123456789abcdefABCDEF") input-str))
(define (digit-string input-str)
  (many decimal-digit input-str))
```

Concurrency in Functional Languages

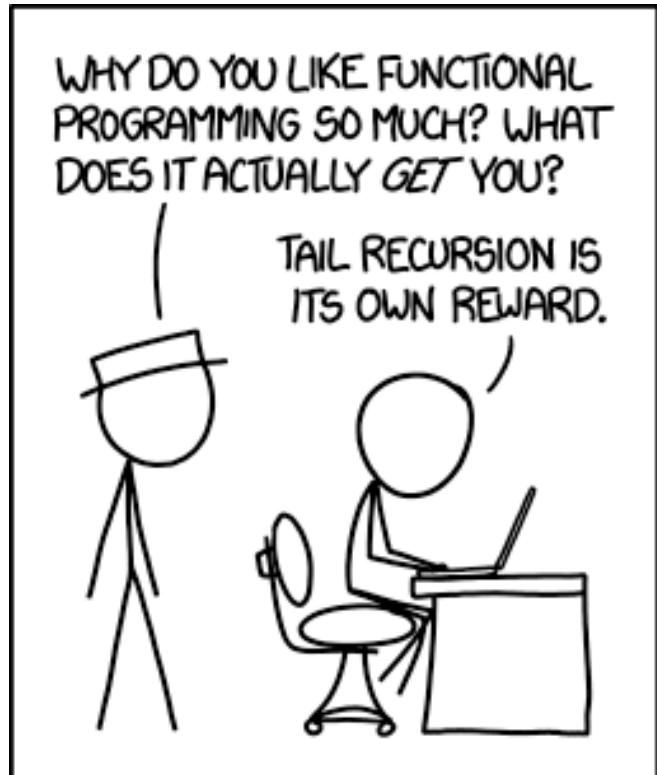
- Because data is immutable, we can't rely on mutexes, semaphores, etc.
- Most functional languages use an explicit *message-passing* protocol (set of functions) to allow inter-thread communication
- But because data is immutable, the most common synchronization problem—1 thread modifying data just as another thread is reading it, or 2 threads trying to write data at the same time—simply doesn't occur.

Why use functional programming?

- Functions can be composed and treated as modules; no side effects, results are always the same for a particular input.
 - As long as rules on type compatibility are met, this is always safe.
- Easier to reason about program behavior & prove correctness
- Functional programs tend to be more compact than procedural code
- Since there's no shared mutable state, there's no interaction between parts of a program except what's defined via function calls.
 - So undocumented side effects, misordered updates, dangling or uninitialized references simply don't occur.

So why aren't we using functional programming all the time?

- Most programmers start out learning procedural languages, so a functional style *looks* hard.
- A lot of online tutorials use Haskell, which most programmers aren't familiar with and has an unusual syntax.
- Functional languages, while easier to write & debug, can be harder to read.
- Many functional languages aren't fully portable, sometimes lacking in library packages, or (especially) debugging & profiling tools.
 - Though this is getting better—Rust is a particular example.
- Some problems (e.g. user interaction) map more directly to procedural style
- If raw execution speed is a criterion, functional languages will lag
 - Though again, this is getting better, and the ease of maintenance of functional languages is a plus. F#, Rust, Racket offer good performance.



- “*Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.*”
- More seriously, though, it lets you reason equationally about your code. Build correct functions, compose them correctly, and you have a provably correct program.