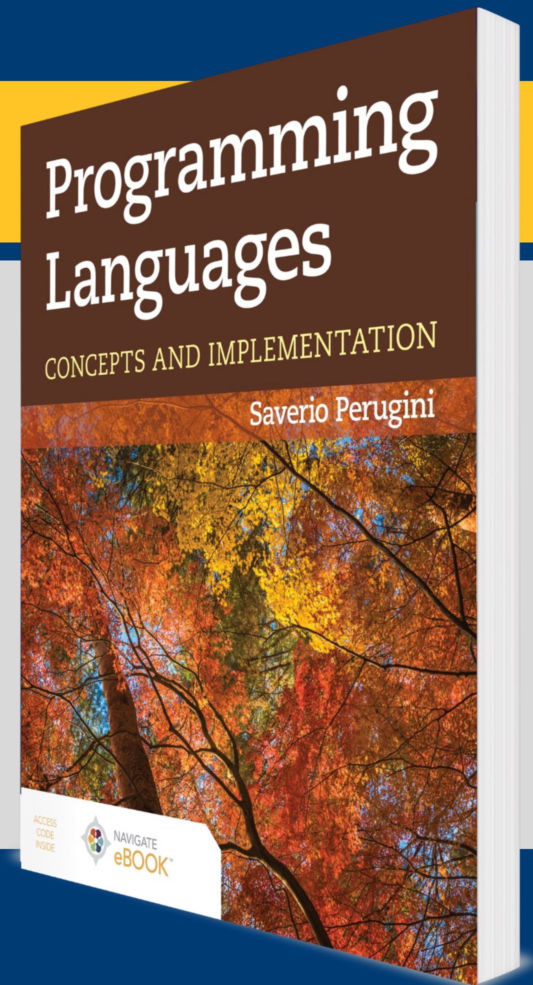
CHAPTER 6

Binding and Scope



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

Static scoping	A reference is bound to a declaration *before* run-time, e.g., based on the spatial relationship of nested program blocks to each other, i.e., ***lexical scoping***.
Dynamic scoping	A reference is bound to a declaration *during* 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:

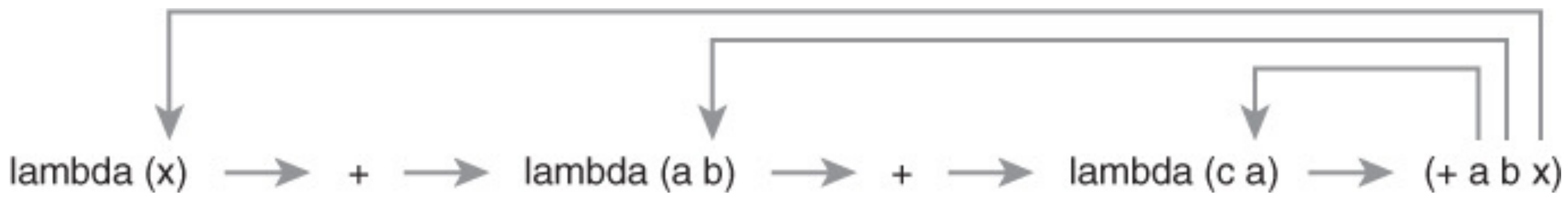
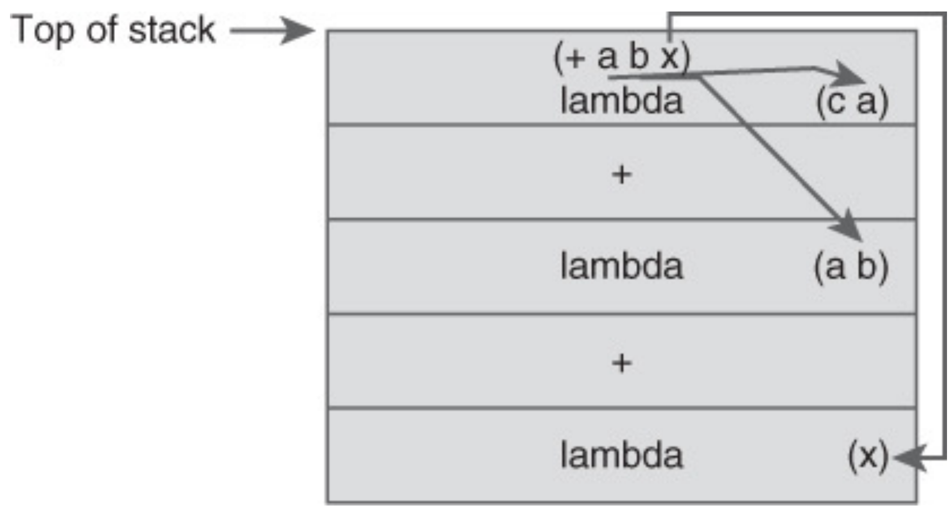


Figure 6.1 Run-time Call Stack at the Time the Expression `(+ a b x)` Is Evaluated



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

$$\begin{array}{rcl} \text{scope}(<\mathit{declaration}>) & = & <\textit{a set of program points}> \\ \text{referencing environment}(<\textit{a program point}>) & = & <\textit{a set of variable bindings}> \end{array}$$

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		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

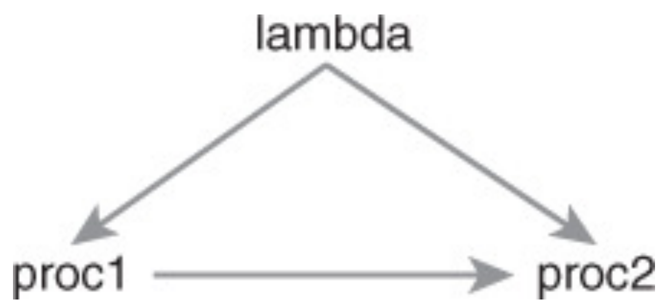
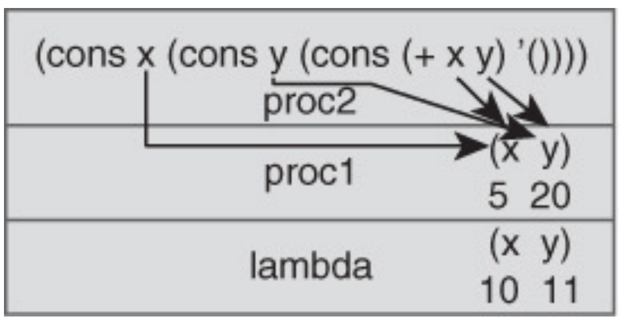


Figure 6.3 The Two Run-Time Call Stacks Possible from the Program Used to Illustrate Dynamic Scoping in Section 6.7

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



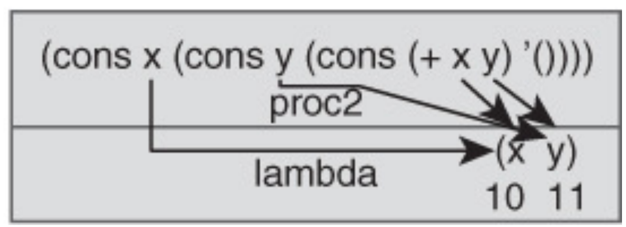


Table 6.5 Advantages and Disadvantages of Static and Dynamic Scoping

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

Listing 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:

```scheme
(define add_x
  (lambda (x)
    (lambda (y)
      (+ x y))))

(define main
   (lambda ()
      (let ((add5 (add_x 5))
            (add6 (add_x 6)))
         (cons (add5 2) (cons (add6 2) '())))))
(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

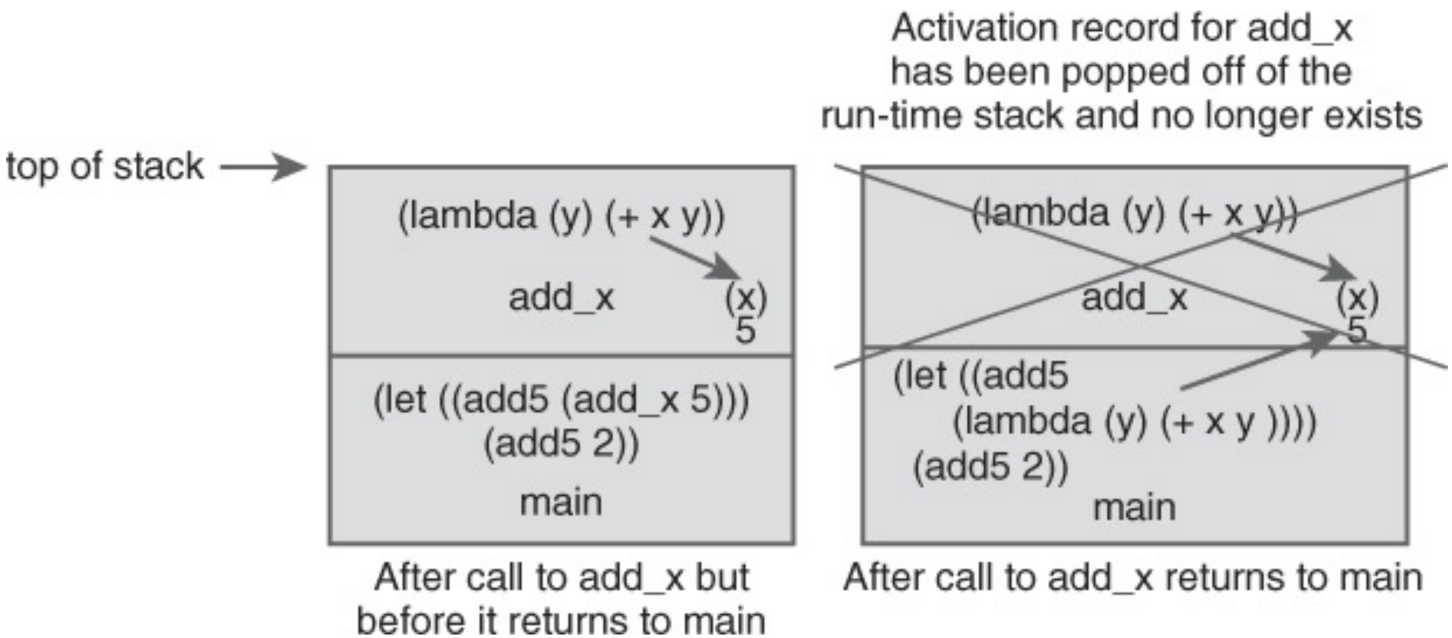


Table 6.6 Example Data Structure Representation of Closures

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 variables

into a *pure* function
  - i.e., a λ-expression with no free variables

by 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)

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

The environment in which the function is called is:

```
(((y 4))
 ((x 10)
  (f (lambda (x) (* (y (+ x x))))))
 ((y 3)))
```

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)

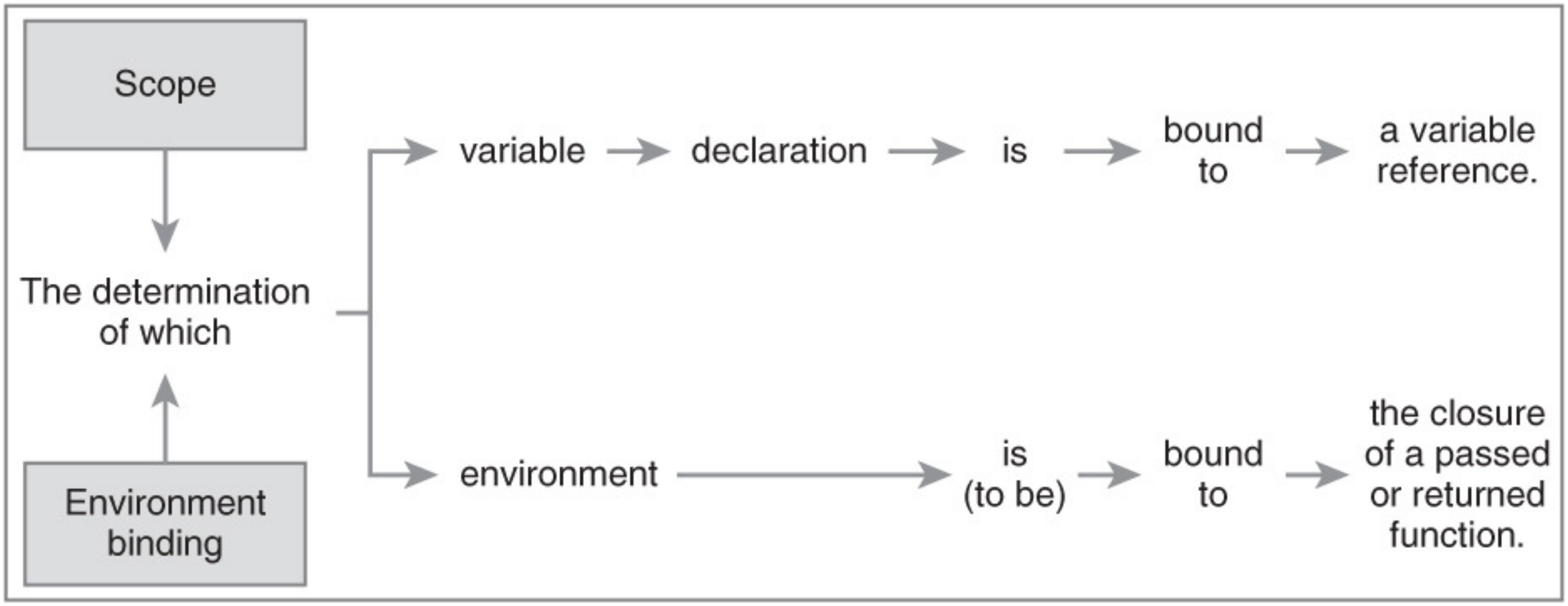
The environment in which the function is called is:

```
(let  ((y 3))
  (let ((x 10)
                  ; 6      24
        (f (lambda (x) (* y (+ x x)))))

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

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

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.