```
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and (not (eqv? (caadr exp) var))
            (occurs-free? var (caddr exp))))
      (else (or (occurs-free? var  (car exp))
                (occurs-free? var (cadr exp)))))))

(define occurs-bound?
  (lambda (var exp)
    (cond
      ((symbol? exp) #f)
      ((eqv? (car exp) 'lambda)
       (or (occurs-bound? var (caddr exp))
           (and (eqv? (caadr exp) var)
                (occurs-free? var (caddr exp)))))
      (else (or (occurs-bound? var  (car exp))
                (occurs-bound? var (cadr exp)))))))
```

Figure 1.1 `occurs-free?` and `occurs-bound?`

Exercise 1.22 [⋆] Scheme `lambda` expressions may have any number of formal parameters, and Scheme procedure calls may have any number of operands. Modify the formal definitions of occurs free and occurs bound to allow `lambda` expressions with any number of formal parameters and procedure calls with any number of operands. Then modify the procedures `occurs-free?` and `occurs-bound?` to follow these new definitions.

Exercise 1.23 [⋆] Extend the formal definitions of occurs free and occurs bound to include `if` expressions.

Exercise 1.24 [⋆⋆] Extend the formal definitions of occurs free and occurs bound to include Scheme `let` and `let*` expressions.

Exercise 1.25 [⋆] Extend the formal definitions of occurs free and occurs bound to include Scheme quotations (expressions of the form `(quote ⟨datum⟩)`).

Exercise 1.26 [⋆⋆] Extend the formal definitions of occurs free and occurs bound to include Scheme assignment (`set!`) expressions.

1.3.2 Scope and Lexical Address

The next problem is to associate with each variable reference the declaration to which it refers. It turns out to be easier to think about the reverse problem: given a declaration, which variable references refer to it?

Typically, the binding rules of a language associate with each declaration of a variable a region of the program within which the declaration is effective. For example, in the Scheme expression

```
(lambda (x) ...)
```

the region for `x` is the body of the `lambda` expression, and in a top-level definition

```
(define x ...)
```

the region is the whole program.

This is not the entire story, however, because many modern languages, including Scheme, allow regions to be *nested* within each other, as when one lambda expression appears in the body of another. Such languages are said to be *block-structured*, and the regions are sometimes called *blocks*.

For example, in Scheme the body of the `lambda` expression above might contain another declaration of `x`. In this case the inner declaration takes precedence over the outer one. Consider

```
> (define x                                ; call this x1
    (lambda (x)                            ; call this x2
      (map
        (lambda (x)                        ; call this x3
          (+ x 1))                         ; refers to x3
        x)))                               ; refers to x2
> (x '(1 2 3))                             ; refers to x1
(2 3 4)
```

Here the expression `(+ x 1)` is within the region of all three declarations of `x`. It therefore takes its binding from the innermost declaration of `x`, the one on the fourth line. Block-structured languages whose scope rules work in this way are said to use *lexical binding*.

We define the *scope* of a variable declaration to be the text within which references to the variable refer to the declaration. Thus the scope of a declaration is the region of text associated with the declaration, excluding any inner regions associated with declarations that use the same variable name. We say that the inner declaration of `x` *shadows* the outer declarations of `x`, or that the inner declaration creates a *hole* in the scope of the outer one. Alternatively, we may speak of the declarations that are *visible* at the point of a variable reference, meaning those that contain the variable reference within their scope.

The declaration of a variable v has a scope that includes all references to v that *occur free* in the region associated with the declaration. Those references to v that *occur bound* in the region associated with its declaration are shadowed by inner declarations.

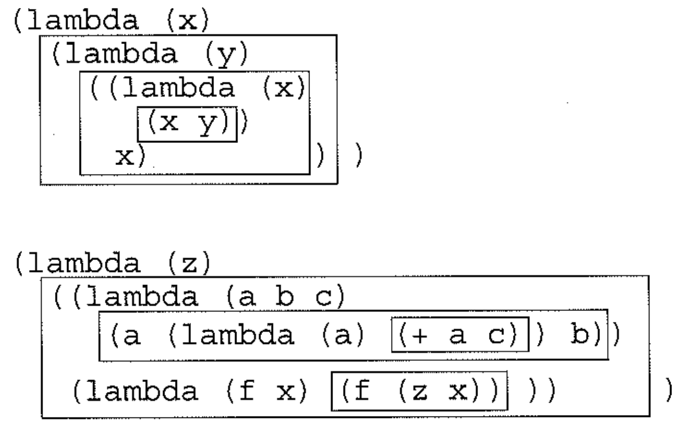
Applying this to the preceding example, the region of the x declared on the first line is the read-eval-print loop's top level, which includes the body of the definition: however, its scope does not include the body of the defined procedure, since x does not occur free in the procedure `(lambda (x) ...)`. The scope of the formal parameter x in the fourth line is the lambda expression's body, `(+ x 1)`. This formal parameter creates a hole in the scope of the formal parameter x in the second line. The scope of the x in the second line includes the reference to x as the second argument to `map`, but not the reference to x as the first argument to +. The inner declarations of x shadow the outer declarations of x.

In a language with lexical binding, there is a simple algorithm for determining the declaration to which a variable reference refers. Search the regions enclosing the reference, starting with the innermost. As each successively larger region is encountered, check whether a declaration of the given variable is associated with the block. If one is found, it is the declaration of the variable. If not, proceed to the next enclosing region. If the outermost (top-level or global) region is reached and no declaration is found, the variable reference is free.

Exercise 1.27 [★] In the following expressions, draw an arrow from each variable reference to its associated formal parameter declaration.

```
(lambda (x)
  (lambda (y)
    ((lambda (x)
       (x y))
     x)))

(lambda (z)
  ((lambda (a b c)
     (a (lambda (a) (+ a c)) b))
   (lambda (f x)
     (f (z x)))))
```



```
(lambda (x)
 (lambda (y)
  ((lambda (x)
     (x y))
   x)      ) )

(lambda (z)
 ((lambda (a b c)
    (a (lambda (a) (+ a c)) b))
  (lambda (f x) (f (z x)) ))     )
```

Figure 1.2 Contour diagrams

Exercise 1.28 [★] Repeat the above exercise with programs written in a block-structured language, other than Scheme.

It is sometimes more helpful to picture the borders of regions, rather than the interiors of regions. These borders are called *contours*. For example, the contours in the preceding exercise can be drawn as in figure 1.2.

Execution of the scoping algorithm may then be viewed as a journey outward from a variable reference. In this journey a number of contours may be crossed before arriving at the associated declaration. The number of contours crossed is called the *lexical* (or *static*) *depth* of the variable reference. It is customary to use "zero-based indexing," thereby not counting the last contour crossed. For example, in

```
(lambda (x y)
  ((lambda (a)
     (x (a y)))
   x))
```

the reference to x on the last line and the reference to a have lexical depth zero, while the references to x and y in the third line have lexical depth one.

The declarations associated with a region may be numbered in the order of their appearance in the text. Each variable reference may then be associated with two numbers: its lexical depth and its position, again using zero-based indexing, of its declaration in the declaring contour (its *declaration position*). Taken together, these two numbers are the variable reference's *lexical address*.

To illustrate lexical addresses, we may replace every variable reference v in an expression by

$$(v : d\ p)$$

where d is its lexical depth and p is its declaration position. The above example then becomes

```
(lambda (x y)
  ((lambda (a)
     ((x : 1 0) ((a : 0 0) (y : 1 1))))
   (x : 0 0)))
```

Since the lexical address completely specifies each variable reference, variable names are then superfluous! Thus variable references could be replaced by expressions of the form (: d p), and formal parameter lists could be replaced by their length, as in

```
(lambda 2
  ((lambda 1
     ((: 1 0) ((: 0 0) (: 1 1))))
   (: 0 0)))
```

Names for lexically-bound variables are certainly a great help in writing and understanding programs, but they are not necessary in executing programs.

Exercise 1.29 [$\star$] What is wrong with the following lexical-address expression?

```
(lambda (a)
  (lambda (a)
    (a : 1 0)))
```

Exercise 1.30 [$\star$] Write a Scheme expression that is equivalent to the following lexical-address expression from which variable names have been removed.

```
(lambda 1
  (lambda 1
    (: 1 0)))
```

Compilers routinely calculate the lexical address of each variable reference. Once this has been done, the variable names may be discarded unless they are required to provide debugging information.

Exercise 1.31 [$\star\star$] Consider the subset of Scheme specified by the BNF rules

```
⟨expression⟩ ::= ⟨identifier⟩
             ::= (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
             ::= (lambda ({⟨identifier⟩}*) ⟨expression⟩)
             ::= ({⟨expression⟩}+)
```

Write a procedure `lexical-address` that takes any expression and returns the expression with every variable reference v replaced by a list (v : d p), as above. If the variable reference v is free, produce the list (v free) instead.

```
> (lexical-address '(lambda (a b c)
                      (if (eqv? b c)
                        ((lambda (c)
                           (cons a c))
                         a)
                        b)))
(lambda (a b c)
  (if ((eqv? free) (b : 0 1) (c : 0 2))
    ((lambda (c)
       ((cons free) (a : 1 0) (c : 0 0)))
     (a : 0 0))
    (b : 0 1)))
```

Exercise 1.32 [$\star\star$] Write the procedure `un-lexical-address`, which takes lexical-address expressions with formal parameter lists and with variable references of the form (: d p), or (v free) and returns an equivalent expression formed by substituting standard variable references for the lexical-address information, or `#f` if no such expression exists.

```
> (un-lexical-address '(lambda (a)
                         (lambda (b c)
                           ((: 1 0) (: 0 0) (: 0 1)))))
(lambda (a) (lambda (b c) (a b c)))
> (un-lexical-address '(lambda (a) (lambda (a) (: 0 1))))
#f
```

Exercise 1.33 [$\star\star$] Some languages do not allow an inner declaration to declare a variable already declared in an outer declaration. Write a procedure that takes a lambda calculus expression and checks to see if it contains such a redeclaration.

Further Reading

Scheme was introduced in (Sussman & Steele, 1975; 1998). Its development is recorded in (Steele & Sussman, 1978; Clinger *et al.*, 1985; Rees *et al.*, 1986; Clinger *et al.*, 1991; Kelsey *et al.*, 1998). The standard definitions of Scheme are provided by the IEEE standard (1991) and the *Revised*^{5} *Report on the Algorithmic Language Scheme* (Kelsey *et al.*, 1998). (Dybvig, 1987; 1996) provides a short introduction to Scheme that includes a number of insightful examples.

Those new to recursive programming and symbolic computation might look at *The Little Schemer* (Friedman & Felleisen, 1996), or *The Little MLer* (Felleisen & Friedman, 1996), or for the more historically-minded, *The Little LISPer* (Friedman, 1974).

The lambda calculus was introduced in (Church, 1941) to study mathematical logic. Introductory treatments of the lambda calculus may be found in (Hankin, 1994), (Peyton Jones, 1987), or (Stoy, 1977). (Barendregt, 1981; 1991) provides an encyclopedic reference.

2 Data Abstraction

2.1 Specifying Data via Interfaces

Every time we decide to represent a certain set of quantities in a particular way, we are defining a new data type: the data type whose values are those representations and whose operations are the procedures that manipulate those entities.

The representation of these entities is often complex, so we do not want to be concerned with their details when we can avoid them. We may also decide to change the representation of the data. The most efficient representation is often a lot more difficult to implement, so we may wish to develop a simple implementation first and only change to a more efficient representation if it proves critical to the overall performance of a system. If we decide to change the representation of some data for any reason, we must be able to locate all parts of a program that are dependent on the representation. This is accomplished using the technique of *data abstraction.*

Data abstraction divides a data type into two pieces: an *interface* and an *implementation*. The interface tells us what the data of the type represents, what the operations on the data are, and what properties these operations may be relied on to have. The *implementation* provides a specific representation of the data and code for the operations that makes use of the specific data representation.

A data type that is abstract in this way is said to be an *abstract data type*. The rest of the program, the *client* of the data type, manipulates the new data only through the operations specified in the interface. Thus if we wish to change the representation of the data, all we must do is change the implementation of the operations in the interface.