Procedures without names, which are not the binding of a variable, are said to be *anonymous*. In most other languages, procedures are never anonymous: they may be created only via declarations that name them. (Of course anonymity is relative to context: if an anonymous procedure is bound to a parameter via procedure call, it is not anonymous in the context of the called procedure.)

Anonymous procedures are often used as arguments. We illustrate this using the procedures `map` and `andmap`, which generally take two arguments: a procedure and a list. The list may be of any length, and the procedure must take one argument. The procedure `map` builds a new list whose elements are obtained by calling the procedure with the elements of the original list. The procedure `andmap` applies the procedure to each element of the list and returns true if all are true. Otherwise it returns false.

```
> (map (lambda (n) (+ n 2)) '(1 2 3 4 5))
(3 4 5 6 7)
> (define add2
    (lambda (n)
      (+ n 2)))
> (map add2 '(1 2 3 4 5))
(3 4 5 6 7)
> (andmap number? '(1 2 3 4 5))
#t
> (map null? '((a) () () (3)))
(#f #t #t #f)
> (andmap null? '((a) () () (3)))
#f
> (map car '((a b) (c d) (e f)))
(a c e)
> (map list '(a b c d))
((a) (b) (c) (d))
> (map (lambda (f) (f '(a b c d)))
       (list car cdr cadr cddr caddr))
(a (b c d) b (c d) c)
```

1.3.2 First-Class Procedures

A value is said to be *first class* if it may be passed to and returned from procedures and stored in data structures. In Scheme, *all* values are first class, including procedures. In other languages, simple values such as numbers are first class, compound values such as records and arrays are sometimes

first class, and procedures are almost never first class. Though it is usually possible to pass procedures as arguments, it is often impossible to return them as values or store them in data structures. (See chapter 10 for a discussion of the implementation of such languages.) First class procedures contribute greatly to the expressive power of a language.

For an example of a procedure that takes procedural arguments and returns a procedural result, consider the problem of defining a procedure that performs functional composition. Assume that f and g are two functions of one argument such that Range(g) $\subseteq$ Domain(f). Then the *composition* of f and g, $f \circ g$, is defined by this equation:

$$(f \circ g)(x) = f(g(x)).$$

The assumption about the range of g and the domain of f ensures that every possible result from g is a possible argument to f. It is straightforward to define composition in Scheme.

```
> (define compose
    (lambda (f g)
      (lambda (x)
        (f (g x)))))
> (define add4 (compose add2 add2))
> (add4 5)
9
> ((compose car cdr) '(a b c d))
b
> ((compose list (compose cdr cdr))
   '(a b c d))
((c d))
```

• *Exercise 1.3.1*

What is unusual about the following expression?

```
((lambda (x)
   (list x (list (quote quote) x)))
 (quote (lambda (x)
          (list x (list (quote quote) x)))))
```

Try to figure out what it does without typing it into a Scheme system. Can similar behavior be achieved without using `list`? □

Here is an implementation of cells.

```
(define cell-tag "cell")

(define make-cell
  (lambda (x)
    (vector cell-tag x)))

(define cell?
  (lambda (x)
    (if (vector? x)
        (if (= (vector-length x) 2)
            (eq? (vector-ref x 0) cell-tag)
            #f)
        #f)))

(define cell-ref
  (lambda (x)
    (if (cell? x)
        (vector-ref x 1)
        (error "Invalid argument to cell-ref:" x))))
```

Fill in the values of the following transcript.

```
> (define c (make-cell 5))
> c

> (cell? c)

> (cell-ref c)
```

□

∘ *Exercise 1.3.3*
Consider two or three other languages you know or for which you can find documentation. What restrictions, if any, are imposed on procedures that keep them from being first class? Is it possible to create anonymous procedures? □

Here is an example of a procedure that takes a numeric value and returns a procedure.

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

When f is passed a number x it returns a procedure that takes a number and adds x to it.

```
> (define new-add2 (f 2))
> (new-add2 4)
6
```

Here *new-add2* has the same behavior as the *add2* procedure defined earlier. So what is the point of defining f? If a computation requires generating many of these add-n procedures for different values of n, or if the values of n are unknown at the time the program is written, then a procedure like f is called for.

```
> (define add3 (f (+ 1 2)))
> (add3 5)
8
> ((f 5) 6)
11
> (define g
    (lambda (a)
      (lambda (d)
        (cons a d))))
> ((g 'a) '(b c))
(a b c)
> (map (g 'a) '((b c) (1 2)))
((a b c) (a 1 2))
```

Having functions of more than one argument is certainly convenient, but it is not absolutely necessary. Using the technique just illustrated, any procedure p of $n \geq 2$ arguments can always be transformed into a procedure p' of one argument that returns a procedure of $n-1$ arguments such that

$$((p' x_1)\, x_2 \ldots x_n) = (p\, x_1 \ldots x_n)$$

By repeating this transformation $n-1$ times, we obtain a procedure p'' such that

$$(\ldots((p'' x_1)\, x_2) \ldots x_n) = (p\, x_1\, x_2 \ldots x_n)$$

This transformation is known as *currying*. The procedures `f` and `g` above are curried versions of the addition and `cons` procedures, respectively. Of course if an existing procedure is to be replaced by a curried version, all calls to the procedure must be changed.

A curried version of a procedure normally takes the first argument first, but this is not always what is desired. The following example illustrates the use of a "reverse-curried" version of `cons`.

```
> (define h
    (lambda (d)
      (lambda (a)
        (cons a d))))
> ((h '(b c)) 'a)
(a b c)
> (map (h '(b c)) '(a 1 2))
((a b c) (1 b c) (2 b c))
```

• *Exercise 1.3.4*

Write a procedure `curry2` that takes a procedure of two arguments and returns a curried version of the procedure that takes the first argument and returns a procedure that takes the second argument. For example,

```
> (((curry2 +) 1) 2)
3
> (define consa ((curry2 cons) 'a))
> (consa '(b))
(a b)
```

□

∘ *Exercise 1.3.5*

Write a curried version of `compose`. Can you think of a use for it? □

∘ *Exercise 1.3.6*

A language could be designed so that if a procedure is passed fewer arguments than it expects, it simply returns a procedure that takes the rest of the arguments. Thus procedures are "automatically" curried. What are the advantages and disadvantages of this feature? □