Exercise 11.10

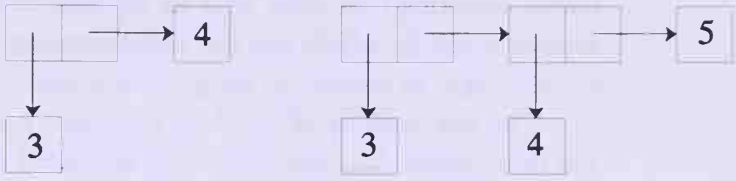
In the imperative-style program for **swapper**, we used while loops to repeat certain steps when a given condition is true. In some languages, while loops are not implemented so another device must be used. Such languages often use a "goto" statement as a means of returning control to a previous step in the program. As in the first imperative version of **member?** in Program 11.7, we can simulate a goto statement by invoking a procedure of no arguments (a thunk). Program 11.11 is a mystery program that is written in imperative style and invokes various thunks to move the control to the body of the thunks. Assume that a global stack **stk** is initially empty. What is returned when we invoke:

```
(mystery 'a 'z '(c r a z y))
```

11.3 Box-and-Pointer Representation of Cons Cells

The box-and-pointer representation gives us a convenient graphical way of visualizing the objects constructed using **cons**. An object that is not a pair, such as a number, a symbol, or a boolean, is denoted by enclosing the object in a box (i.e., we put a square or rectangle around the object). For example, we represent the number 5 by enclosing the numeral 5 in a box. The constructor **cons** produces a pair represented by a *cons cell*, which is a double box (a horizontal rectangle divided into two boxes by a vertical line) with a pointer (arrow) emerging from the center of each of the two boxes. The pointer emerging from the center of the box on the left points to the box containing the car of the pair represented by the cons cell. The pointer from the center of the box on the right points to the box containing the **cdr** of the pair represented by the cons cell. Figure 11.12(a) shows the box-and-pointer representation of the improper list (or dotted pair) **(cons 3 4)**. The pointer from the left side points to the **car**, which is **3**, and the pointer from the right side points to the **cdr**, which is **4**. When **(cons 3 4)** is entered into Scheme, the improper list **(3 . 4)** is returned. We call the pointer from the left side of a cons cell the *car pointer* and the pointer from the right side of a cons cell the *cdr pointer*.

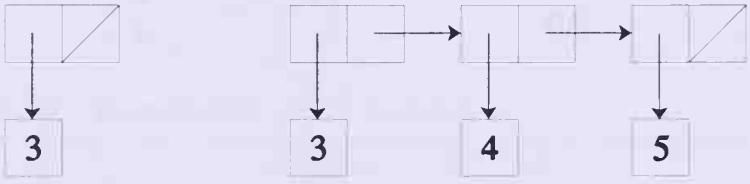
The value of **(cons 3 (cons 4 5))** is represented by two cons cells, one for each **cons**. Figure 11.12(b) shows the box-and-pointer configuration for this value. The car pointer of the first cons cell points to the number **3**, and the cdr pointer of the first cons cell points to the second cons cell. The car pointer of the second cons cell points to the number **4**, and the cdr pointer of the second cons cell points to the number **5**. In this way, we can build up



(cons 3 4) (cons 3 (cons 4 5))

(a) (b)

Figure 11.12 Box-and-pointer diagrams



(cons 3 '()) (cons 3 (cons 4 (cons 5 '())))

(a) (b)

Figure 11.13 Box-and-pointer diagrams for proper lists

the box-and-pointer representations of the values of more complicated cons expressions.

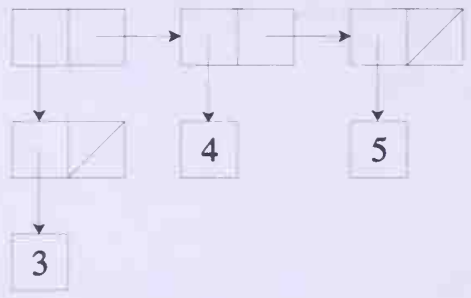
We now look at the representation of a proper list. We begin with the list `(cons 3 '())`, for which Scheme displays `(3)`. Once again a cons cell is created by `cons`, and this time the car pointer points to the number `3`. But how shall we represent the fact that the cdr pointer points to `()`? We indicate that the `cdr` is the empty list by drawing a diagonal line in the right half of the cons cell. This is illustrated in Figure 11.13(a).

The list `(cons 3 (cons 4 (cons 5 '())))`, which appears on the screen as `(3 4 5)`, is represented as the linked cells in Figure 11.13(b). Another interesting list to consider is

```
(cons (cons 3 '()) (cons 4 (cons 5 '())))
```

which appears on the screen as `((3) 4 5)`. The box-and-pointer represen-

tation contains four cons cells as illustrated in Figure 11.14. The first **cons** creates a cell in which the car pointer points to the cons cell created by the second **cons**, in which the car pointer points to 3 and the cdr pointer indicates `()`. The cdr pointer of the first cons cell points to the cons cell created by the third **cons**. The car pointer in the third cons cell points to **4**, and its cdr pointer points to the cons cell created by the fourth **cons**. In this fourth cons cell, the car pointer points to **5**, and the cdr pointer indicates `()`. Thus each **cons** in an expression creates a new cons cell in which the car pointer points to the car part and the cdr pointer points to the cdr part of the cell.



(cons (cons 3 '()) (cons 4 (cons 5 '())))

Figure 11.14 Box-and-pointer diagram

If we define **a** to be `(cons 3 '())` by writing

```
(define a (cons 3 '()))
```

we can indicate this binding by a pointer from the name **a** to the cons cell created by **cons**, as illustrated in Figure 11.15(a). If we now use **set!** to change this binding, say

```
(set! a (cons 4 (cons 5 '())))
```

we can think of this as disconnecting the pointer from **a** to the linked cells representing `(cons 3 '())` and connecting it to the linked cells representing `(cons 4 (cons 5 '()))`, as illustrated in Figure 11.15(b).

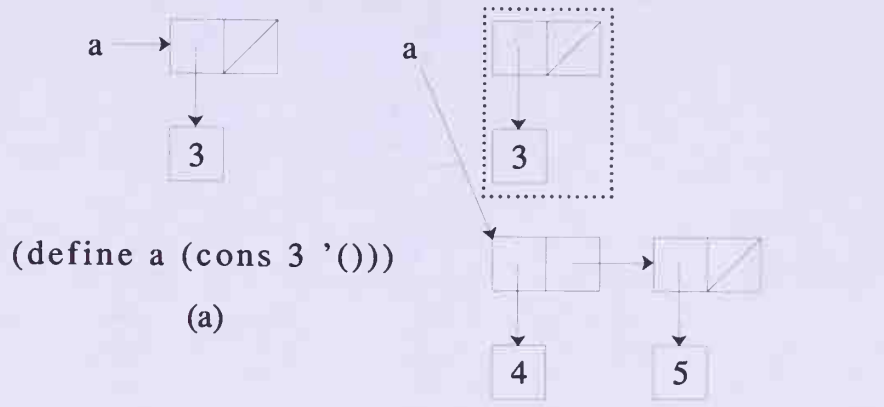


Figure 11.15 Representing define and set!

Now suppose that `a` and `b` are defined as:

```
(define a (cons 1 (cons 2 '())))
```

```
(define b (cons (cons 3 '()) (cons 4 (cons 5 '()))))
```

as illustrated in Figure 11.16(a,b). We next define `c` to be:

```
(define c (cons a (cdr b)))
```

The `cons` in the definition of `c` creates a cons cell (to which `c` points) in which the car pointer points to the same cell as does `a` and the cdr pointer points to the same cell as does the cdr pointer of the cons cell to which `b` points. This is illustrated in Figure 11.16(c). It is clear from this representation that `a` and `b` have not been changed when we defined `c`, and `a` $\Longrightarrow$ `(1 2)`, `b` $\Longrightarrow$ `((3) 4 5)`, and `c` $\Longrightarrow$ `((1 2) 4 5)`.

The procedures `set-car!`, `set-cdr!`, and `append!`, which we discuss next, actually do change the objects to which they are applied. For example, when we use the same definitions of `a` and `b` given above and illustrated in Figure 11.16(a,b), and invoke

```
(set-car! b a)
```

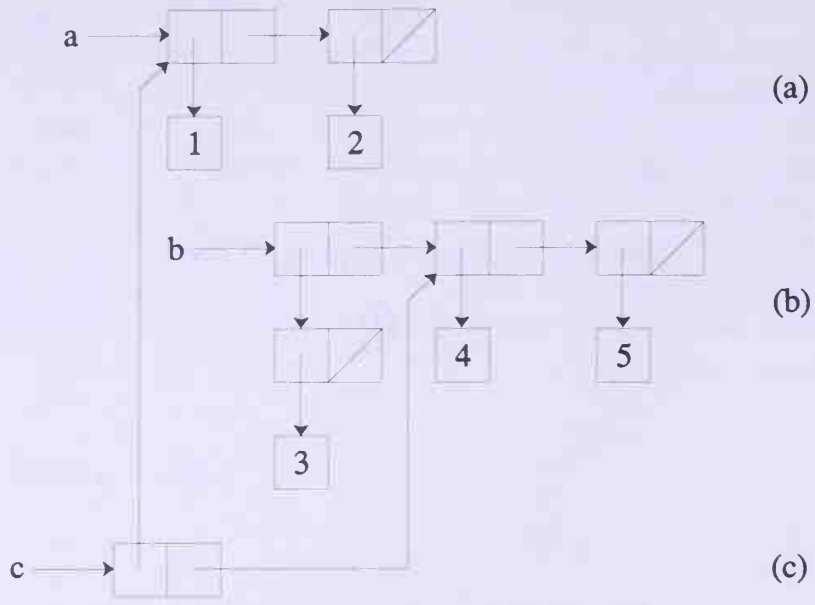


Figure 11.16 `(define c (cons a (cdr b)))`

then the car pointer of the cons cell to which `b` points is disconnected and is made to point to the same linked cell as does `a`. This is illustrated in Figure 11.17. Now:

```
b ⟹ ((1 2) 4 5)
a ⟹ (1 2)
```

Thus `b` has been changed by `set-car!` so that we can say that `set-car!` caused a mutation in the list structure of `b`. This enables us to use `set-car!` in begin expressions since this mutation is a side effect. Let us compare `b` and `c`. They are `equal?`, but not `eq?`. Also the application of `set-car!` had the effect of disconnecting the previous car of `b` (see the dotted box in Figure 11.17) which is now "garbage" to be recycled in the next "garbage collection." In general, cells are garbage if they are *not* pointed to by nongarbage.

The invocation (`set-cdr!` *pair value*) does the same kind of reconnecting of the cdr pointer of *pair* so that it points to *value*. The box-and-pointer diagrams in Figure 11.18 illustrate `c` and `d` defined by

```
(define c (cons 1 (cons 2 (cons 3 '()))))

(define d (cons 4 (cons 5 (cons 6 '()))))
```

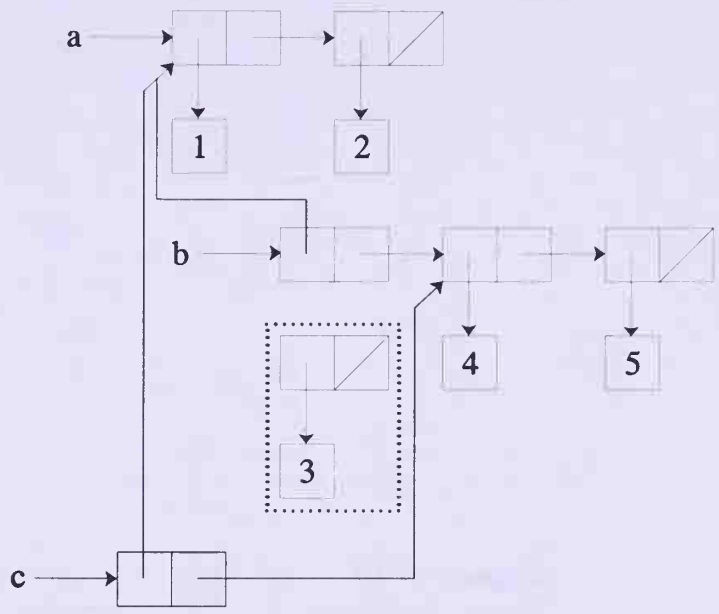


Figure 11.17 `(set-car! b a)` in Figure 11.16

Let us first define `w` to be

```
(define w (cons c d))
```

so that `w` ⟹ `((1 2 3) 4 5 6)`. (See Figure 11.18.)

We pause to make an important observation about the predicate `eq?`. Two items are the same in the sense of `eq?` if they point to the same object. From Figure 11.18, we see that

```
(eq? (car w) c) ⟹ #t
```

since they both point to the same chain of linked cells.

If we next call

```
(set-cdr! c d)
```

the cdr pointer of `c` is changed to refer to `d` (Figure 11.19) and now `c` ⟹ `(1 4 5 6)`. But the side effects of `(set-cdr! c d)` extend to all objects that have pointers to `c`; now `w` ⟹ `((1 4 5 6) 4 5 6)`. Thus care must be taken when using procedures like `set-car!` and `set-cdr!` that cause mutations in the list structure that unexpected or unwanted changes in other data objects do not

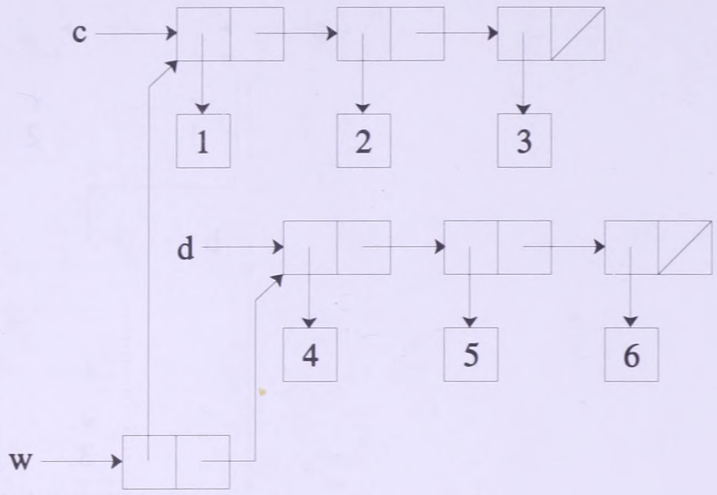


Figure 11.18 `(eq? (car w) c)` ⟹ #t

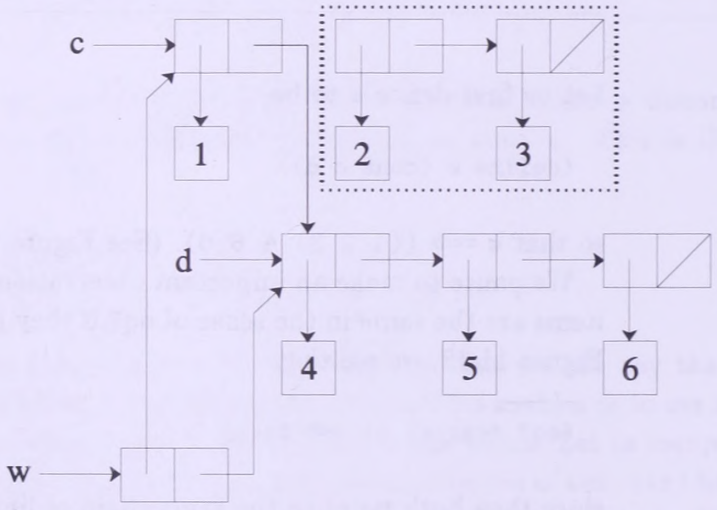


Figure 11.19 `(set-cdr! c d)` in Figure 11.18

occur. The two procedures **`set-car!`** and **`set-cdr!`** cause mutations in the list structure of data objects, but the values that they return are unspecified and may differ in different implementations of Scheme. Thus to write programs that are *portable* (run in various implementations of Scheme), it is necessary to avoid using the values returned by these procedures.

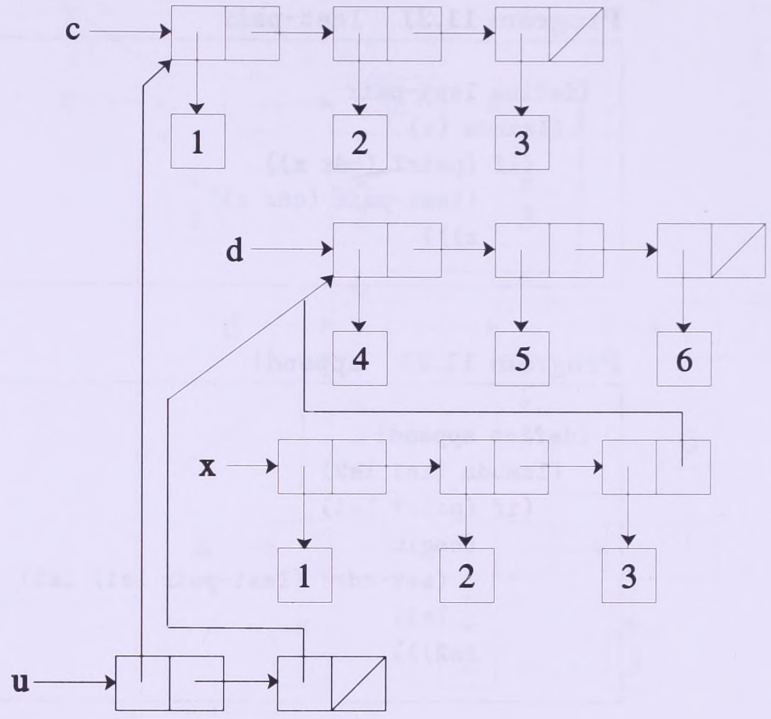


Figure 11.20 `(define x (append c d))`

The procedure **append** was defined in Program 4.1 as:

```
(define append
  (lambda (ls1 ls2)
    (if (null? ls1)
        ls2
        (cons (car ls1) (append (cdr ls1) ls2)))))
```

If `c` and `d` are again defined as in Figure 11.18, and we define

```
(define u (cons c (cons d '())))
```

and

```
(define x (append c d))
```

then **append** makes a copy of `c` and changes the cdr pointer of the last cons cell in this copy to point to `d`. (See Figure 11.20.) Then:

Program 11.21 `last-pair`

```
(define last-pair
  (lambda (x)
    (if (pair? (cdr x))
        (last-pair (cdr x))
        x)))
```

Program 11.22 `append!`

```
(define append!
  (lambda (ls1 ls2)
    (if (pair? ls1)
        (begin
          (set-cdr! (last-pair ls1) ls2)
          ls1)
        ls2)))
```

```
x ⟹ (1 2 3 4 5 6)
c ⟹ (1 2 3)
d ⟹ (4 5 6)
u ⟹ ((1 2 3) (4 5 6))
```

In making the copy of `c`, `x` had to create three cons cells.

The procedure `append!` offers a more efficient way of appending one list to another. However, it has side effects that must be considered, so it should not be used indiscriminately. Let us begin by defining the procedure `last-pair` (see Program 11.21), which takes as its argument a nonempty list and returns the list consisting of the last value in the list. For example:

```
(last-pair '(1 2 3)) ⟹ (3)
```

We then define `append!` in Program 11.22. Here `last-pair` cdr's down the list `ls1` until it reaches the last pair in `ls1`. Then `set-cdr!` redirects the cdr pointer to `ls2` instead of the empty list. The last line in the begin expression returns this mutated list `ls1`. For example, if we apply `append!` to the two lists `c` and `d` defined above by writing

```
(define y (append! c d))
```

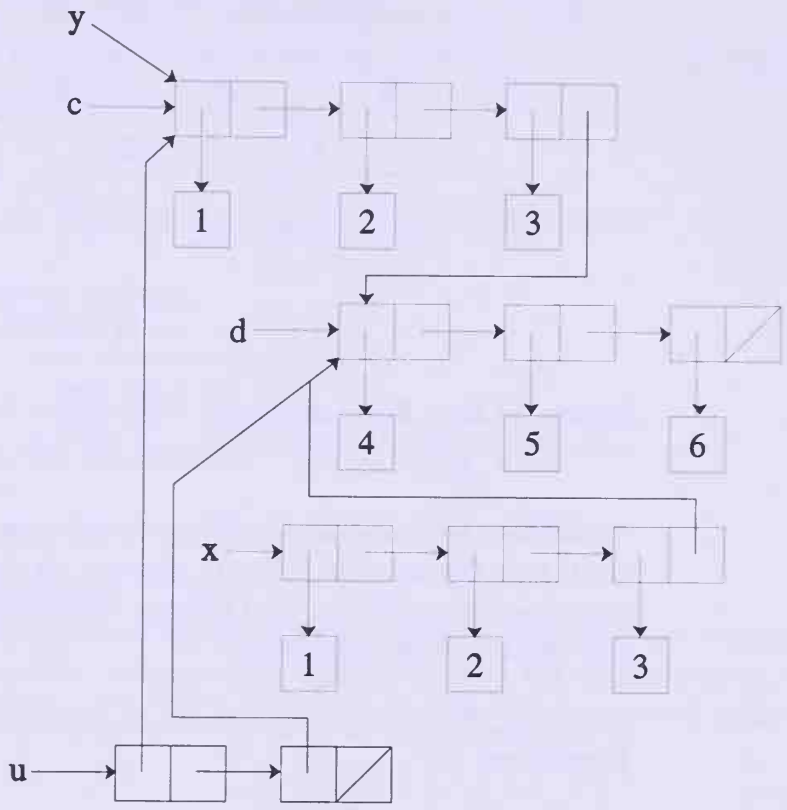


Figure 11.23 `(define y (append! c d))` in Figure 11.20

then `y` is obtained by connecting the cdr pointer of the last cons cell in `c` to `d`. (See Figure 11.23.) We now have

```
y ⟹ (1 2 3 4 5 6)
c ⟹ (1 2 3 4 5 6)
d ⟹ (4 5 6)
x ⟹ (1 2 3 4 5 6)
u ⟹ ((1 2 3 4 5 6) (4 5 6))
```

The last result is a side effect of using **`append!`** on `c`, for the `c`, which also appears in the definition of `u`, has been mutated. The value of `x` is not changed because it originally makes a copy of `c` and has no pointer to `c`. Thus each time we have a choice of using one of the procedures **`set-car!`**, **`set-cdr!`**, or **`append!`**, we must decide whether we want a copy of the original lists made by using suitable procedures of **`cons`**, **`car`**, **`cdr`**, and **`append`** or mutations of the original lists taking into account the possible side effects. We must be

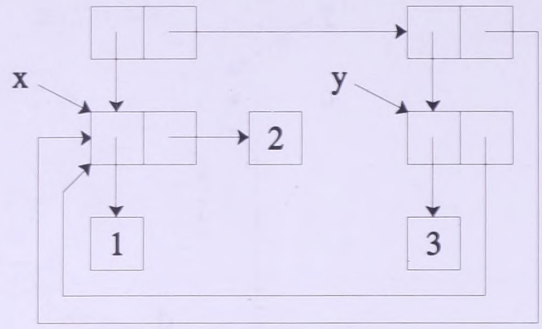


Figure 11.24 Box-and-pointer diagram for Exercise 11.12

careful that undesirable side effects do not occur when using **set-car!**, **set-cdr!**, and **append!**. In the next chapter, we shall see examples where these procedures can safely be used because the variables affected are local and the side effects can be controlled.

Exercises

Exercise 11.11

Draw a box-and-pointer diagram for the following:

```
(let ((x (list 1 2 3)))
  (let ((y (list 4 5 6)))
    (let ((z (cons x y)))
      (set-cdr! x y)
      z)))
```

Exercise 11.12

Write a let expression in the style of Exercise 11.11 that generates the entire structure shown in the box-and-pointer diagram in Figure 11.24.

Exercise 11.13

a. Draw a box-and-pointer diagram for the following:

```
(let ((x (cons 1 1)))
  (set-car! x x)
  (set-cdr! x x)
  x)
```

b. Define a procedure that recognizes such cons cells.

c. Define a procedure that takes a list as its argument and removes all occurrences of such cons cells.

Exercise 11.14

Conduct the following experiment, explaining the results:

```
[1] (define mystery
      (lambda(x)
        (let ((box (last-pair x)))
          (set-cdr! box x)
          x)))
[2] (define ans (mystery (list 'a 'b 'c 'd)))
[3] ans
?__________
```

Exercise 11.15

Let us consider only flat lists in this exercise. We know that we can write a procedure that determines the length of a list. We allow, however, that the cdr of the last cell of the list might point back to some portion of the list. For example,

```
(let ((x (list 'a 'b 'c 'd 'e)))
  (set-cdr! (last-pair x) (cdr (cdr x)))
  x)
```

We can print such lists by invoking `(writeln x)`; however, the printing of the list will not terminate. Redefine `writeln` so that if it discovers one of these lists, it prints something appropriate. For example,

`(writeln x)` $\Longrightarrow$ `(a b c d e c d e ...)`

Hint: Define a predicate `cycle?` which determines if a list is a flat cycle. Also, reconstruct a list like `(a b c d e c d e ...)` that has the string `"..."` as the last of its nine elements.

Exercise 11.16: `efface`

Create the box-and-pointer diagrams for `x`, `y`, `z`, `a`, `a*`, `b`, `b*`, `c`, and `c*` before and after the invocation of `efface` in `test-efface`.

```
(define efface
  (lambda (x ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) x) (cdr ls))
      (else (let ((z (efface x (cdr ls))))
              (set-cdr! ls z)
              ls)))))

(define test-efface
  (lambda ()
    (let ((x (cons 1 '())))
      (let ((y (cons 2 x)))
        (let ((z (cons 3 y)))
          (let ((a (cons 4 z)) (a* (cons 40 z)))
            (let ((b (cons 5 a)) (b* (cons 50 a)))
              (let ((c (cons 6 b)) (c* (cons 60 b)))
                (writeln x y z a a* b b* c c*)
                (efface 3 c)
                (writeln x y z a a* b b* c c*)))))))))
```

Exercise 11.17

Using the definition of **efface** from Exercise 11.16, describe the behavior of **(test-efface2)** and **(test-efface3)**. Explain the difference.

```
(define test-efface2
  (lambda ()
    (let ((ls (list 5 4 3 2 1)))
      (writeln (efface 3 ls))
      ls)))

(define test-efface3
  (lambda ()
    (let ((ls (list 5 4 3 2 1)))
      (writeln (efface 5 ls))
      ls)))
```

Exercise 11.18: **smudge**

Create the box-and-pointer diagrams for **x**, **y**, **z**, **a**, **a***, **b**, **b***, **c**, and **c*** before and after the invocation of **smudge** in **test-smudge**.

```
(define smudge
  (lambda (x ls)
    (letrec
      ((smudge/x
         (lambda (ls*)
           (cond
             ((null? (cdr ls*)) ls*)
             ((equal? (car ls*) x) (shift-down ls* (cdr ls*)))
             (else (smudge/x (cdr ls*)))))))
      (if (null? ls)
          ls
          (begin
            (smudge/x ls)
            ls))))))

(define shift-down
  (lambda (box1 box2)
    (set-car! box1 (car box2))
    (set-cdr! box1 (cdr box2))))

(define test-smudge
  (lambda ()
    (let ((x (cons 1 '())))
      (let ((y (cons 2 x)))
        (let ((z (cons 3 y)))
          (let ((a (cons 4 z)) (a* (cons 40 z)))
            (let ((b (cons 5 a)) (b* (cons 50 a)))
              (let ((c (cons 6 b)) (c* (cons 60 b)))
                (writeln x y z a a* b b* c c*)
                (smudge 3 c)
                (writeln x y z a a* b b* c c*)))))))))
```

Exercise 11.19: **count-pairs**

The procedure **count-pairs** counts the number of cons cells in a data structure. It is defined using the global variable ***seen-pairs*** and the helping predicate **dont-count?**.

```
(define *seen-pairs* '())
```

```
(define count-pairs
  (lambda (pr)
    (if (dont-count? pr)
        0
        (begin
          (set! *seen-pairs* (cons pr *seen-pairs*))
          (add1 (+ (count-pairs (car pr))
                   (count-pairs (cdr pr))))))))

(define dont-count?
  (lambda (s)
    (or (not (pair? s)) (member? s *seen-pairs*))))
```

a. Create a box-and-pointer diagram for **y** just prior to the invocation of **count-pairs** in the procedure **test-count-pairs**.

```
(define test-count-pairs
  (lambda ()
    (let ((x (cons 'a (cons 'b (cons 'c '())))))
      (let ((y (cons x (cons x (cons x x)))))
        (set-cdr! (last-pair x) x)
        (writeln (count-pairs y))
        (count-pairs y))))))
```

b. Explain the behavior of **test-count-pairs**. Why are the answers different? Rewrite **count-pairs** functionally so that the two answers are the same. *Hint*: Look back at the definition of **vector-insertsort!** in Program 10.5.

c. Rewrite **count-pairs** using local state, which gets changed with **set!**. Here is a skeleton:

```
(define count-pairs
  (lambda (pr)
    (count-pairs/seen pr '())))

(define count-pairs/seen
  (lambda (pr seen-pairs)
    (letrec
      ((count (lambda (pr) ...)))
      (count pr))))
```

d. Rewrite `count-pairs` using private local state. *Hint*: Look at the skeleton below. We must set `seen-pairs` back to the empty list just prior to invoking `count`. Here is a skeleton:

```
(define count-pairs
  (let ((seen-pairs "any list of pairs"))
    (letrec
      ((count (lambda (pr) ...)))
      (lambda (pr)
        (set! seen-pairs '())
        (count pr)))))
```

e. Rewrite `count-pairs` using private local state, setting `seen-pairs` to the empty list after invoking `count` instead of before invoking `count`. This way we know that `seen-pairs` is always the empty list before and after invoking `count-pairs`.

The next seven problems are related. Work them in order and you will learn about what computers cannot do.

Exercise 11.20: Turing Tapes

Consider a list that grows in both directions: `(... c b a x y ...)`. We call such a list an *unbounded tape*, or just `tape`. We use a positive number of 0's as left and right borders to indicate where the *interesting information* on the tape resides: `(... 0 c b a x y 0 ...)`. Any symbol would work as the border symbol; no border value can appear within the interesting information on the tape, for if it did then it would indicate a border. Included as part of the data abstraction of a tape is a location on the tape. Tapes can be read only a character at a time. For the purposes of this discussion, `x` is the character being read on the above tape.

There are four procedures defined over tapes. The first one is `at`, which takes a tape and returns the character being read. The second one is `overwrite`, which takes a character, `c`, and a tape, and returns an equivalent tape except that the character being read is replaced by `c`. We can characterize the relation between `overwrite` and `at` with the following equation. Let `t` be a tape and let `c` be a character; then:

$$\texttt{(at (overwrite c t))} \equiv \texttt{c}$$

The third and fourth procedures are `left` and `right`. These take a tape and return an equivalent tape except that the character being read is the one

just to the left (or right) of the one that was previously being read. In our example, this would mean that the character being read is now `a` (or `y`). Since the tape is unbounded in both directions, there is no concern about falling off the tape. We have the identities:

```
(left (right tape)) ≡ tape ≡ (right (left tape))
```

In our use of tapes, we always do **overwrite** followed by either **left** or **right** but not both. We refer to this operation as *reconfiguring* the tape. In order to reconfigure a tape, we need a character to write and a direction in which to move:

```
(define reconfigure
  (lambda (tape character direction)
    (if (eq? direction 'left)
        (left (overwrite character tape))
        (right (overwrite character tape)))))
```

We now consider a possible representation of tapes. In this representation a tape is composed of two non-null, finite lists. We call these two lists *left part* and *right part*. The left part contains everything to the left of where we are reading until the left end of the tape, but it is *reversed*. In our example, that is `(a b c 0)`. The right part contains everything from where we are reading until the right end of the tape. In our example above, that is `(x y 0)`. Thus, the tape is represented by the list `((a b c 0) (x y 0))`. We can now define **at** and **overwrite**:

```
(define at
  (lambda (tape)
    (let ((right-part (2nd tape)))
      (car right-part))))

(define overwrite
  (lambda (char tape)
    (let ((left-part (1st tape)) (right-part (2nd tape)))
      (let ((new-right-part (cons char (cdr right-part))))
        (list left-part new-right-part)))))
```

We have only to define the procedures **left** and **right**. Let us consider what is involved in moving to the right. In our example this would mean that we are looking at `y`. If that is so, then the right part would become `(y 0)`. What would happen to the `x`? Since it is now to the left of where we are

reading, it would be moved into the left part and would become the first item in the left part. Here is a first try at **right**:

```
(define right
  (lambda (tape)
    (let ((left-part (1st tape)) (right-part (2nd tape)))
      (let ((new-left-part (cons (car right-part) left-part))
            (new-right-part (cdr right-part)))
        (list new-left-part new-right-part)))))
```

This is very close to correct, but it might violate the restriction that the left part and the right part must be non-null. Consider invoking **right** on the tape **((y x a b c 0) (0))**. Using this incorrect version of **right**, the new tape would become **((0 y x a b c 0) ())**, and that violates the non-null condition that each part must satisfy. If **new-right-part** is the empty list, we must replace it by **(0)**, which represents the right end of the tape. Here is the improved version of **right**:

```
(define right
  (lambda (tape)
    (let ((left-part (1st tape)) (right-part (2nd tape)))
      (let ((new-left-part (cons (car right-part) left-part))
            (new-right-part (cdr right-part)))
        (list new-left-part (check-null new-right-part))))))

(define check-null
  (lambda (part)
    (if (null? part)
        (list 0)
        part)))
```

Write the procedure **left**, and test **reconfigure** with the procedure below:

```
(define test-reconfigure
  (lambda ()
    (let ((tape1 (list (list 'a 'b 'c 0) (list 'x 'y 0))))
      (let ((tape2 (reconfigure tape1 'u 'right))
            (tape3 (reconfigure tape1 'd 'left)))
        (let ((tape4 (reconfigure tape2 'v 'right))
              (tape5 (reconfigure tape3 'e 'left)))
          (let ((tape6 (reconfigure tape4 'w 'right))
                (tape7 (reconfigure tape5 'f 'left)))
            (let ((tape8 (reconfigure tape6 'x 'right))
                  (tape9 (reconfigure tape7 'g 'left)))
              (list tape8 tape9)))))))))
```

Exercise 11.21: `list->tape`, `tape->list`

Define a pair of procedures that builds an interface for handling tapes. The first procedure, `list->tape`, takes a list, `ls`, of characters that contains no 0's and produces a tape, `t`, with the condition that `ls` is the same as `(right t)` minus trailing zeros. For example, if `ls` is `(x y)`, then `(list->tape ls)` returns `((0) (x y 0))`. The procedure `tape->list` takes a tape and returns a list. The resultant list does not keep track of where on the tape it is reading. Hence, it is *not* always the case that `(list->tape (tape->list t)) = t`; however, it is always the case that `(tape->list (list->tape ls)) = ls`. For example, if the tape, `t`, is `((a b c 0) (x y 0))`, then `(tape->list t)` is `(c b a x y)`, but `(list->tape '(c b a x y))` is `((0) (c b a x y 0))`, not `((a b c 0) (x y 0))`. Not only must the left part be reversed, but no 0's should appear in the resultant list. Rewrite `test-reconfigure` so that it uses `list->tape` and `tape->list`.

Exercise 11.22

In the procedure `test-reconfigure` from the previous exercise, we used `tape1` twice. Generally that does not happen. More frequently, a tape is used as an argument in an iterative program. Consider the following experiment:

```
[1] (define shifter
      (letrec
        ((shift-to-0
           (lambda (tape)
             (let ((c (at tape)))
               (cond
                 ((equal? c 0) tape)
                 (else (shift-to-0 (reconfigure tape c 'right))))))))
        shift-to-0))
[2] (shifter (list (list 0) (list 'a 'b 'c 0)))
```

When the tape is used in this fashion, we no longer need to make a new copy each time we reconfigure the tape. For example, we can redefine `overwrite` to change the value that `at` returns just by using `set-car!`:

```
(define overwrite
  (lambda (char tape)
    (let ((right-part (2nd tape)))
      (set-car! right-part char)
      tape)))
```

In changing the definition of `overwrite`, we exchanged one invocation of set-car! for three uses of cons and one use of cdr, but `test-reconfigure` no longer produces the same result. Why? Redefine `right` and `left` to use as few invocations of cons as possible. Test `shifter` as in [2].

Exercise 11.23

We can write interesting procedures that begin with an empty tape (i.e, `(list->tape '())`). Test the procedures below using an empty tape and determine which ones do not halt:

```
(define busy-beaver
  (letrec
    ((loopright
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (loopright (reconfigure tape 'a 'right)))
             (else (maybe-done (reconfigure tape 'a 'right)))))))
     (maybe-done
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a) (reconfigure tape 'a 'right))
             (else (continue (reconfigure tape 'a 'left)))))))
     (continue
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (maybe-done (reconfigure tape 'a 'left)))
             (else (loopright (reconfigure tape 'a 'right))))))))
    loopright))

(define endless-growth
  (letrec
    ((loop
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 0)
              (loop (reconfigure tape 'a 'right))))))))
    loop))
```

```
(define perpetual-motion
  (letrec
    ((this-way
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (that-way (reconfigure tape 0 'right)))
             (else (that-way (reconfigure tape 'a 'right)))))))
     (that-way
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (this-way (reconfigure tape 0 'left)))
             (else (this-way (reconfigure tape 'a 'left)))))))
    this-way))

(define pendulum
  (letrec
    ((loopright
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (loopright (reconfigure tape 'a 'right)))
             (else (loopleft (reconfigure tape 'a 'left)))))))
     (loopleft
       (lambda (tape)
         (let ((c (at tape)))
           (cond
             ((equal? c 'a)
              (loopleft (reconfigure tape 'a 'left)))
             (else (loopright (reconfigure tape 'a 'right)))))))
    loopright))
```

Exercise 11.24

Each of the procedures in the previous exercise looks about the same. Each takes a tape as an argument. Then it reconfigures the tape according to the current character and either returns the reconfigured tape or passes it along to another procedure. We can think of each cond line as a list of five elements. For example, the first cond line of **continue** in **busy-beaver** could be represented by: **(continue a a left maybe-done)**. We can interpret this line as follows: *From the state of* **continue**, *if there is an* **a**, *overwrite it*

with an a, *move* left, *and consider only the lines that start with* maybe-done. The entire **busy-beaver** procedure could be represented by a list of all the transcribed cond lines:

```
(define busy-beaver-lines
  '((loopright a a right loopright)
    (loopright 0 a right maybe-done)
    (maybe-done a a right halt)
    (maybe-done 0 a left continue)
    (continue a a left maybe-done)
    (continue 0 a right loopright)))
```

We use the convention that we start the computation at `(car (car busy-beaver-lines))`. We also assume that **halt** is self-explanatory.

Using the representation of tapes that we have developed thus far, define a procedure **run-lines** that takes a set of lines, like the **busy-beaver-lines**, and a tape and returns the same result as `(busy-beaver tape)`. The procedure below will get you started. You need only define the procedures called by **run-lines**.

```
(define run-lines
  (lambda (lines tape)
    (letrec
      ((driver
         (lambda (state tape)
           (if (eq? state 'halt)
               tape
               (let ((matching-line
                       (find-line state (at tape) lines)))
                 (driver
                   (next-state matching-line)
                   (reconfigure
                     tape
                     (next-char matching-line)
                     (next-direction matching-line))))))))
      (driver (current-state (car lines)) tape))))
```

Such a set of lines is called a *Turing machine*, named for Alan M. Turing. Turing claimed that with a small set of characters, including the 0, he could use his machines to compute whatever a computer could. Then he showed that no one can write a procedure **test-lines**, like **run-lines**, that takes

an arbitrary machine and an arbitrary tape and determines whether (run-lines machine tape) halts. This result is so important that it has been given a name, the *halting problem*. All this was done in 1936!

Exercise 11.25
Often it is possible to remove a test by careful design. In the definition of reconfigure, there is a superfluous test. Rewrite busy-beaver replacing 'left by left and 'right by right, so that a revised definition of reconfigure works.

Exercise 11.26: New Representation
Consider a representation of tapes that keeps what it is reading separately. For example, we might choose a list of three elements

```
(at left-part right-part-less-at)
```

Then we could redefine overwrite and right as follows:

```
(define overwrite
  (lambda (char tape)
    (let ((left (2nd tape)) (right (3rd tape)))
      (list char left right))))

(define right
  (lambda (tape)
    (let ((char (1st tape))
          (left (2nd tape))
          (right (3rd tape)))
      (list (car right)
            (cons char left)
            (check-null (cdr right))))))
```

Use this representation of tapes and test busy-beaver. Then redefine all necessary procedures so that the use of cons is minimal.
