Chapter 17

Iteration

Like recursion, iteration is a programming technique that enables a programmer to make things happen repetitively. However, unlike recursion, iteration typically involves destructive programming. This chapter presents several special forms that facilitate incorporating iteration in Scheme programs. The `set!` special form causes the value of a variable to be destructively modified. The `while` special form iteratively evaluates the expressions in its body as long as some condition holds. (Destructive programming is required if a condition that initially evaluates to *true* is eventually going to evaluate to *false*.) The `dotimes` and `dolist` special forms, respectively, automate kinds of iteration that are analogous to numerical and list-based recursion.

17.1 The `set!` Special Form

The purpose of the `set!` special form is to destructively modify (i.e., change) the value of a variable (i.e., the value associated with a symbol in some environment). After using `set!` to change the value of a variable, its previous value will be lost forever—unless it was saved elsewhere prior to setting the new value.

The syntax of the `set!` special form. The `set!` special form has the following syntax:

```
(set! var newVal)
```

where *`var`* is any symbol expression, and *`newVal`* can be any Scheme expression. Each of the following are legal instances of the `set!` special form:

```
(set! x 3)
(set! myVar (* 8 10))
(set! yourVar (member 3 '(1 2 3 4 5)))
```

The semantics of the `set!` special form. Like most Scheme expressions, the evaluation of a `set!` special form depends on the environment in which it is being evaluated. The simplest case is that of a `set!` special form being evaluated with respect to the Global Environment. In that case, the evaluation proceeds as follows.

(1) *`newVal`* is evaluated in the Global Environment, generating some Scheme datum D.

(2) D is inserted as the new value for *`var`* in the Global Environment.

(3) The `set!` special form itself evaluates to *void* (i.e., no value).

Example 17.1.1: The `set!` special form

```
> (define x 100)        ⟵ Create a global variable x with value 100
```

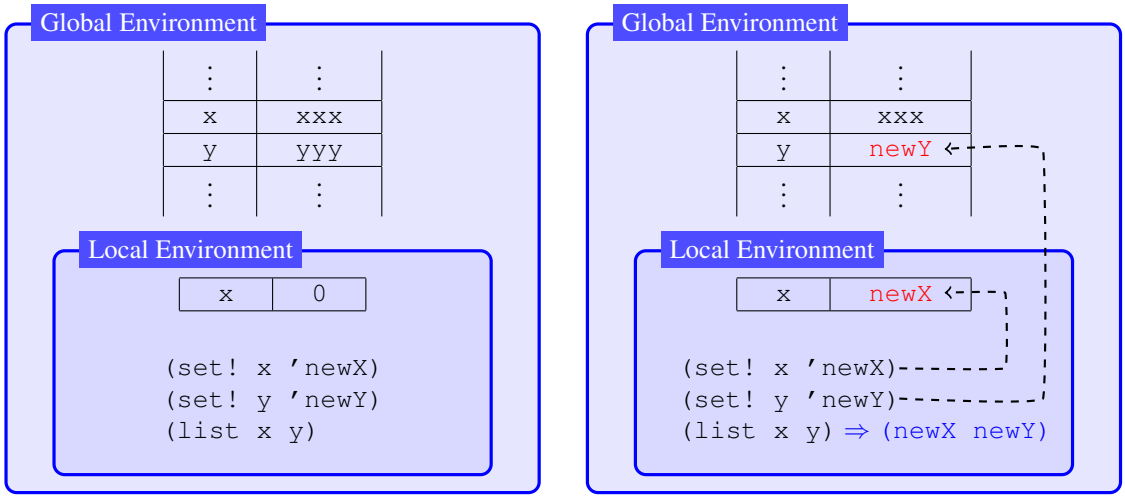

Figure 17.1: The environments for Example 17.1.2 before (left) and after (right) evaluating the `set!` expressions

```
> x
100
> (set! x (* 5 5))      ⟵ Change the value associated with x
> x
25                      ⟵ Observe that x now has the value 25
> (set! y 50)           ⟵ No can do! There is no entry for a variable named y.
Error!
```

As the last example demonstrates, DrScheme will report an error if you try to use `set!` *for a symbol that has no entry in the relevant environment.*

Next, suppose that `(set!` *`var newVal`*`)` is being evaluated with respect to a local environment $\mathcal{E}_1$ that is nested directly inside the Global Environment (i.e., $\mathcal{E}_1 \subset \mathcal{E}_0$). In this case, *`newVal`* will be evaluated with respect to the environment $\mathcal{E}_1$, generating some Scheme datum D. Next, the appropriate variable named *`var`* must be located. If there is an entry for the symbol *`var`* in the local environment $\mathcal{E}_1$, then D will be inserted as the value for *`var`* in that entry. Otherwise, D will be inserted as the value for *`var`* in the Global Environment. (If neither environment contains an entry for *`var`*, then attempting to use `set!` to change its value would cause an error.)

Example 17.1.2: Using `set!` within a local environment

The following interactions begin by creating global variables named `x` *and* `y`*, and then using a* `let` *to create a local variable named* `x`*.*

```
> (define x 'xxx)
> (define y 'yyy)
> (let ((x 0))
    (set! x 'newX)
    (set! y 'newY)
    (list x y))
(newX newY)
> x
xxx
> y
```

```
newY
```

In the body of the `let`*, the two* `set!` *expressions are evaluated with respect to the local environment, as illustrated in Fig. 17.1. Because that local environment has an entry for* `x`*, the expression* `(set! x 'newX)` *changes the value of* `x` *in the local environment. In contrast, there is no entry for* `y` *in the local environment; therefore, the expression* `(set! y 'newY)` *changes the value of* `y` *in the Global Environment. Note that the list generated as the output value for the* `let` *expression contains the new values for the local variable* `x` *and the global variable* `y`*. Finally, evaluating* `x` *in the Global Environment shows that the global variable* `x` *has not been affected, whereas the global variable* `y` *has a new value.*

More generally, suppose that `(set! var newVal)` is being evaluated with respect to an environment $\mathcal{E}_n$ that is nested inside other environments, as follows: $\mathcal{E}_n \subset \mathcal{E}_{n-1} \subset \ldots \subset \mathcal{E}_2 \subset \mathcal{E}_1 \subset \mathcal{E}_0$. The following steps are carried out:

(1) The expression *newVal* is evaluated with respect to the environment $\mathcal{E}_n$, generating some datum D.

(2) The environments, $\mathcal{E}_n, \mathcal{E}_{n-1}, \ldots, \mathcal{E}_0$, are scanned, in order, to find the first one that contains an entry for `var`. Call that environment $\mathcal{E}_j$.

(3) D is inserted as the new value for *var* in the environment $\mathcal{E}_j$.

(4) The `set!` special form itself evaluates to *void.*

Example 17.1.3: Using `set!` in deeply nested environments

Consider the following multiply nested `let` *expressions. As illustrated in Fig. 17.2, each of the first three* `let` *expressions creates a local environment with a variable named* `x`*. The fourth* `let` *creates a local environment with a variable named* `w`*. The* `set!` *special form is evaluated with respect to the innermost local environment, $\mathcal{E}_4$, but ends up changing the value of* `x` *in the environment, $\mathcal{E}_3$, because that is the ancestor environment that is closest to $\mathcal{E}_4$ that contains a variable named* `x`*.*

```
> (let ((x 10))    ;; Environment E1
    (let ((x 20))    ;; Environment E2
      (let ((x 30))    ;; Environment E3
        (let ((w 40))    ;; Environment E4
          (printf "Evaluating x in env. E4 before: ~A~%" x)
          (set! x 99)
          (printf "Evaluating x in env. E4 after:  ~A~%" x))
        (printf "Evaluating x in env. E3 after:  ~A~%" x))
      (printf "Evaluating x in env. E2 after:  ~A~%" x))
    (printf "Evaluating x in env. E1 after:  ~A~%" x))
Evaluating x in env. E4 before: 30
Evaluating x in env. E4 after:  99
Evaluating x in env. E3 after:  99
Evaluating x in env. E2 after:  20
Evaluating x in env. E1 after:  10
```

Notice that there is no way to use `set!` *in environments $\mathcal{E}_4$ or $\mathcal{E}_3$ to change the value of the variable named* `x` *in either $\mathcal{E}_2$ or $\mathcal{E}_1$, because those variables are effectively blocked by the presence of the variable named* `x` *in $\mathcal{E}_3$.*

⋆ Although it is important to understand which variable is affected when a `set!` special form is evaluated

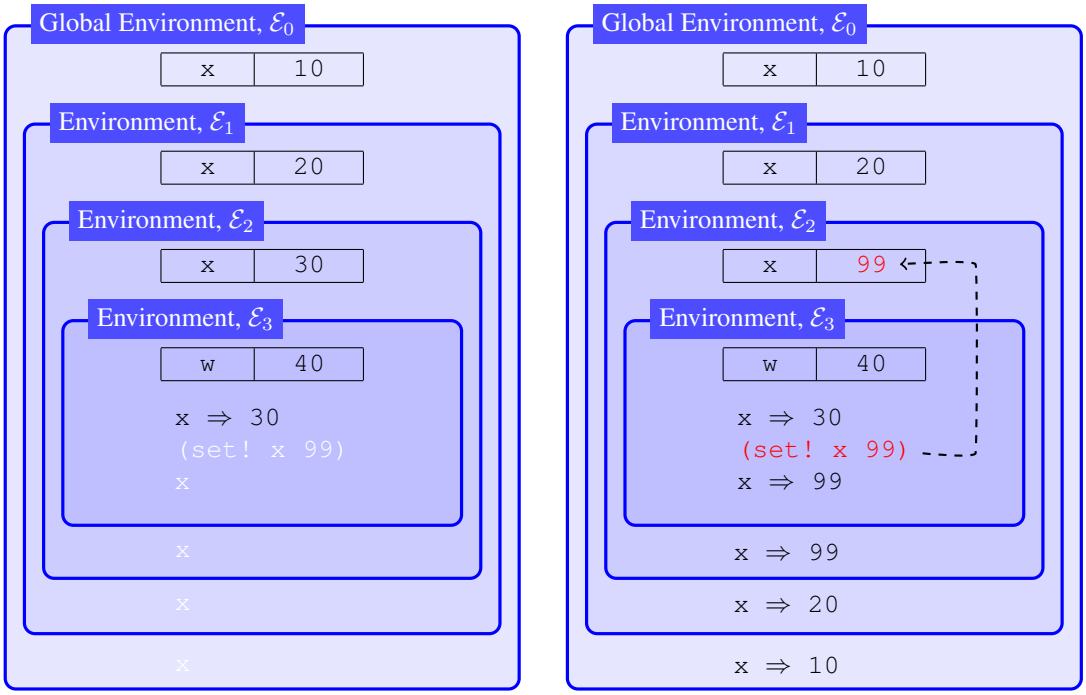

Figure 17.2: The environments for Example 17.1.3 before (left) and after (right) evaluating the `set!` expression

in a particular environment, a programmer typically avoids difficult cases by not having multiple variables with the same name in different environments.

What now? Okay, so we can use `set!` to change the value of a variable. How can we use that capability to our advantage? The next section introduces the `while` special form which, together with `set!`, enables a kind of computation called *iteration* that can be a useful alternative to recursion.

17.2 The `while` Special Form

The purpose of the `while` special form is to enable a kind of *looping* behavior called *iteration.* A typical example might be glossed as: "As long as (while) some condition C holds, do some action A". The semantics for this example could be summarized as follows:

(1) Evaluate condition C.

(2) If C evaluates to *true,* then:

(3) Do action A and go back to Step 1;

(4) Otherwise, stop.

For example, C might be the condition "the value of x is positive", and A might be the compound action "print out the value of x and then decrease x's value by one." As this example suggests, the `while` special form only makes sense in the context of destructive programming because, without destructive programming, the specified condition would always evaluate to the same thing. In particular, in the context of non-destructive programming, if the condition C evaluated to *true* in Step 1, then it would forever evaluate to *true,* leading to a situation where the action A would be repeated forever. However, as in the example, if the value of x decreases by one on each *iteration,* then the condition might eventually evaluate to *false.*

⋆ A is the action that is done *iteratively.* Each doing of Step 3 is called an *iteration.*

The syntax of the `while` special form. The `while` special form has the following syntax:

```
(while condExpr
  expr1
  expr2
  ...
  exprk)
```

where *condExpr* is the *condition,* and the expressions, $expr_1, \ldots, expr_k$, together constitute the *body.* Note that the syntax of the `while` special form is identical to that of the `when` special form—except for its name, of course. The essential difference is in the semantics.

The semantics of the `while` special form. A `while` special form is evaluated as follows:

(1) Evaluate *condExpr*.

(2) If *condExpr* evaluates to (something that counts as) *true,*

(3) Then evaluate each of the expressions, $expr_1, \ldots, expr_k$, in order, then go back to Step 1.

(4) Otherwise, return *void* as the output of the `while` expression.

Note that the only way out of a `while` loop is via Step 4. Hence, a `while` special form always evaluates to *void*—unless the condition stays *true* forever, leading to an infinite loop.

Example 17.2.1

The following interactions demonstrate that: (1) the expressions in the body of a `while` *may be evaluated zero times, and (2) a* `while` *expression evaluates to* void.

```
> (while #f (printf "hi!"))
> (void? (while #f (printf "hi!")))
#t
```

Example 17.2.2: A typical `while` loop

This example illustrates how `let`, `while` *and* `set!` *can be combined to create a useful* while loop.

```
> (let ((counter 0))
    (while (< counter 4)
      (printf "counter = ~A~%" counter)
      ;; Increment the value of COUNTER
      (set! counter (+ counter 1)))
    ;; after the WHILE:
    counter)
counter = 0
counter = 1
counter = 2
counter = 3
4
```

In this example, the `let` *special form creates a local environment in which the variable* `counter` *has an initial value of* `0`. *As long as the value of* `counter` *is less than* `4`, *the expressions in the body of the* `while` *are evaluated, leading to side-effect printing. In addition, on each iteration, the value of* `counter` *is incremented by one. As a result, the condition* `(< counter 4)` *will eventually evaluate*

to `#f`*, stopping the loop. Notice that after the* `while` *loop,* `counter` *has the value* `4`*, which caused the condition to become false.*

⋆ If you forget to increment the counter variable in the while loop, then the value of `counter` will never change, and the condition will forever evaluate to `#t`, resulting in an infinite loop. Whoops!

Example 17.2.3: Summing numbers iteratively

The following combination of `let`*,* `while` *and* `set!` *computes the sum of the numbers from* 1 *to* 100*. Notice the use of an accumulator variable whose value is destructively modified on each iteration.*

```
> (let ((counter 0)
        (acc 0))
    (while (<= counter 100)
           (set! acc (+ acc counter))
           (set! counter (+ counter 1)))
    ;; After the WHILE loop:  the accumulator has the answer
    acc)
5050
```

In-Class Problem 17.2.1

Define a function, called `sum-iter`*, that takes a positive integer* `n` *as its only input and returns as its output the sum of the numbers from* `1` *to* `n`*. Define another function, called* `facty-iter`*, that takes a positive integer* `n` *as its only input and returns as its output the factorial of* `n` *(i.e., the product of the numbers from* `1` *to* `n`*). For each function, use* `let`*,* `while` *and* `set!`*, as demonstrated in the previous example, to implement the desired iteration. Here are some examples of the desired behavior.*

```
> (sum-iter 4)
10
> (facty-iter 4)
24
```

Example 17.2.4: Using `random` within a `while` loop

This example illustrates that the value of a variable can be set to a random value on each iteration, leading to a while loop having an unpredictable number of iterations.

```
 > (let ((val (random 4)))
     (while (< val 3)
            (printf "val: ~A~%" val)
            (set! val (random 4)))
     ;; after the WHILE:
     val)
val: 2
val: 0
val: 2
val: 1
3
> (let ((val (random 4)))
```

```
      (while (< val 3)
             (printf "val: ~A~%" val)
             (set! val (random 4)))
      ;; after the WHILE:
      val)
val: 1
val: 0
val: 1
val: 2
val: 1
val: 0
3
```

Example 17.2.5: Implementing our own version of `while`

The following `my-while` *function provides essentially the same behavior as the* `while` *special form. However, to ensure proper behavior, the condition and the body are encapsulated within* `lambda` *functions, each of which takes zero inputs. The* `my-while` *function causes the desired condition to be evaluated by applying* `cond-func` *to zero inputs; and it causes the desired body expressions to be evaluated by applying* `body-func` *to zero inputs. Note that the recursive call to* `my-while` *is applied to the same inputs! As a result, the* `my-while` *function is implicitly relying on* `cond-func` *or* `body-func` *to be destructive to avoid going into an endless loop.*

```
;;  MY-WHILE
;; ------------------------------------------------------------
;;  INPUTS:  COND-FUNC, a function that takes zero inputs
;;           BODY-FUNC, a function that takes zero inputs
;;  OUTPUT:  VOID
;;  SIDE EFFECT:  As long as (COND-FUNC) evaluates to
;;    (something that counts as) true, MY-WHILE evaluates
;;    (BODY-FUNC).

(define my-while
  (lambda (cond-func body-func)
    (when (cond-func)              ⟵ Apply cond-func to zero inputs
      (body-func)                  ⟵ Apply body-func to zero inputs
      ;; Recursively call MY-WHILE with the same inputs!
      (my-while cond-func body-func))))
```

The following interaction demonstrates the use of the `my-while` *function.*

```
> (let ((ctr 3))
    (printf "MY-WHILE example:~%")
    (my-while (lambda ()                    ⟵ cond-func
                (> ctr 0))
              (lambda ()                    ⟵ body-func
                (printf "  ctr: ~A~%" ctr)
                (set! ctr (1- ctr)))))
MY-WHILE example:
  ctr: 3
  ctr: 2
  ctr: 1
```

Note that because `my-while` *is a function, the expression,* `(my-while ...)`*, is evaluated by the Default Rule. As a result, both* `lambda` *expressions are evaluated* before *calling the* `my-while` *function. Therefore, the corresponding* `lambda` *functions are* created *in the context of the local environment that includes an entry for the symbol* `ctr`*. Thus, whenever these* `lambda` *functions are eventually called, their bodies are evaluated in an environment that is nested within the environment that has an entry for* `ctr`*. Thus, any occurrences of the symbol* `ctr` *in the bodies of those functions will refer to the local variable* `ctr`*, as desired.*

Here is the equivalent example done using the `while` *special form.*

```
> (let ((ctr 3))
    (printf "WHILE example:~%")
    (while (> ctr 0)
           (printf "  ctr: ~A~%" ctr)
           (set! ctr (1- ctr))))
WHILE example:
  ctr: 3
  ctr: 2
  ctr: 1
```

17.3 Converting Tail Recursion to Iteration

Recall from Section 14.2 that for any tail-recursive function, DrScheme can employ a memory-saving trick whereby a single function-call box is repeatedly recycled, instead of creating a new function-call box for each recursive function call. In effect, what DrScheme does is to convert a tail-recursive function call into iteration. This section describes the process.

We begin by recalling the tail-recursive `print-n-dashes` function, seen previously in Example 14.2.1, and then showing how it can be implemented iteratively using `while` and `set!`.

Example 17.3.1: Implementing `print-n-dashes` iteratively

Here is the `print-n-dashes` *function. Note that it is tail recursive.*

```
;;  PRINT-N-DASHES
;; ---------------------------------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints N dashes in the Interactions Window

(define print-n-dashes
  (lambda (n)
    (cond
      ;; Base Case: n <= 0
      ((<= n 0)
       (newline))
      ;; Recursive Case: n > 0
      (#t
       ;; Print one dash
       (printf "-")
       ;; Let the recursive func call print the rest of the dashes
       (print-n-dashes (- n 1))))))
```

Here is an equivalent function, called `print-n-dashes-iter`*, that is implemented using iteration.*

```
(define print-n-dashes-iter
  (lambda (n)
    ;; Iterative Case:  N > 0
    (while (> n 0)
           (printf "-")
           (set! n (- n 1)))
    ;; Base Case:  N <= 0
    (newline)))
```

The following interactions demonstrate that the two functions provide the same functionality.

```
> (print-n-dashes 5)
-----
> (print-n-dashes 8)
--------
> (print-n-dashes-iter 5)
-----
> (print-n-dashes-iter 8)
--------
```

For the recursive function, the recursive case involves the printing of a single dash followed by a tail-recursive function call with the input `(- n 1)`*. For the iterative function, each iteration involves the printing of a single dash followed by* destructively *decrementing the value of* `n` *by one. For both functions, the base case is reached when the value of* `n` *is zero, resulting in a newline being generated. For the recursive function, the base case is one of the cases that is explicitly handled by the* `cond` *expression; for the iterative function, the base case is what happens after the* `while` *loop is completed.*

We can make the comparison between the recursive and iterative versions even clearer if we implement the recursive version as follows:

```
(define print-n-dashes-alt
  (lambda (n)
    ;; Recursive Case:  N > 0
    (when (> n 0)
      (printf "-")
      (print-n-dashes (- n 1)))
    ;; Base Case:  N <= 0
    (newline)))
```

When DrScheme evaluates the expression `(print-n-dashes-alt 5)`*, it uses the trick of recycling a single function-call box for all of the tail-recursive function calls. In that single function-call box, there is a local environment in which the value of the local variable* `n` *starts out at* `5`*, but then decrements by one for each recursive function call. Similarly, when DrScheme evaluates the expression* `(print-n-dashes-iter 5)`*, there is a single function-call box in which the value of the local variable* `n` *starts out at* `5`*, but then decrements by one on each iteration.*

Unlike in the above example, most of the times we want to create a tail-recursive function, we start by defining a tail-recursive helper function that takes extra inputs (e.g., accumulators). Afterward, we define a wrapper function that calls the tail-recursive helper function with suitable inputs. These more typical cases of tail-recursion can also be easily converted into iteration, as demonstrated by the following example.

Example 17.3.2

Recall the `sum-to-n-acc` *function, seen previously in Example 14.3.2.*

```
;;  SUM-TO-N-ACC
;; -------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;           ACC, a number (an accumulator)
;;  OUTPUT:  When called with ACC=0, the output is the value
;;             0 + 1 + 2 + ... + N.
;;           More generally, the output is the value of
;;             ACC + 0 + 1 + 2 + ... + N.

(define sum-to-n-acc
  (lambda (n acc)
    (cond
      ;; Base Case:  n = 0
      ((= n 0)
       (printf "Base Case (n=0, acc=~A)~%" acc)
       ;; Return the accumulator!
       acc)
      ;;Recursive Case:  n > 0
      (#t
       (printf "Recursive Case (n=~A, acc=~A)~%" n acc)
       ;; Make recursive function call with updated inputs
       (sum-to-n-acc (- n 1) (+ acc n))))))
```

Here is the corresponding wrapper function:

```
;;  SUM-TO-N-WR
;; ------------------------------------------------
;;  INPUT:  N, a non-negative integer
;;  OUTPUT: The sum, 0 + 1 + 2 + ... + N

(define sum-to-n-wr
  (lambda (n)
    ;; Call the accumulator-based helper with ACC=0:
    (sum-to-n-acc n 0)))
```

And here is an iterative function, `sum-to-n-iter`*, that performs the same computation:*

```
;;  SUM-TO-N-ITER
;; ------------------------------------------------
;;  INPUT:  N, a non-negative integer
;;  OUTPUT: The sum, 0 + 1 + 2 + ... + N

(define sum-to-n-iter
  (lambda (n)
    ;; Create local variable ACC whose initial value is 0
    (let ((acc 0))
      ;; Iterative Case:  N > 0
      (while (> n 0)
        ;; Accumulate the current value of N
        (set! acc (+ acc n))
```

```
          ;; Decrement N by one
          (set! n (- n 1)))
        ;; After the WHILE, ACC has the answer
        acc)))
```

When there are multiple base cases and multiple recursive cases, the conversion from recursion to iteration can still be done quite easily.

Example 17.3.3

The following tail-recursive function walks through a list of numbers, searching for an occurrence of num*. Along the way, it prints out information about the numbers it passes by: + for numbers bigger than* num*, – for numbers smaller than* num*. As its output, it returns* #f *if* num *wasn't found in the list; otherwise, it returns the index of the position where* num *was found. Although this function can easily be implemented without multiple base cases or multiple recursive cases, implementing it in this way enables us to demonstrate the general process of converting a tail-recursive function to an iterative function.*

```
;;  INDEX-OF-NUM-IN-LIST-ACC
;; -------------------------------------------------------
;;  INPUTS:  NUM, a number
;;           LISTY, a list of numbers
;;           INDY, current index
;;  OUTPUT:  When called with INDY = 0, the output is
;;    the index of the first occurrence of NUM in LISTY;
;;    or #f if NUM does not occur in LISTY.

(define index-of-num-in-list-acc
  (lambda (num listy indy)
    (cond
      ;; Base Case 1:  LISTY empty
      ((null? listy)
       ;; NUM not found
       #f)
      ;; Base Case 2:  NUM found!
      ((= num (first listy))
       ;; return the current index
       indy)
      ;; Recursive Case 1:  (FIRST LISTY) > NUM
      ((> (first listy) num)
       (printf "+")
       (index-of-num-in-list-acc num (rest listy) (+ indy 1)))
      ;; Recursive Case 2:  (FIRST LISTY) < NUM
      (else
       (printf "-")
       (index-of-num-in-list-acc num (rest listy) (+ indy 1))))))

;;  INDEX-OF-NUM-IN-LIST  --  wrapper function
;; -------------------------------------------------------
;;  INPUTS:  NUM, a number
;;           LISTY, a list of numbers
;;           INDY, current index
```

```
;;  OUTPUT:  The index of the first occurrence of NUM
;;    in LISTY; or #f if NUM does not occur in LISTY.

(define index-of-num-in-list
  (lambda (num listy)
    ;; Call tail-recursive helper with INDY = 0:
    (index-of-num-in-list num listy 0)))
```

Below, the `index-of-num-in-list-iter` *function is an iterative implementation that exhibits the same behavior. Notice the use of two extra variables,* `continue?` *and* `answer`*. The* `while` *loop keeps going as long as the value of* `continue?` *is* `#t`*. When the base case is reached,* `continue?` *is set to* `#f` *to stop the* `while` *loop; and, because the* `while` *expression evaluates to* void*, the* `answer` *variable is used to store the desired answer so that it can be recalled after the* `while` *loop is finished.*

```
(define index-of-num-in-list-iter
  (lambda (num listy)
    (let ((indy 0)
          (answer #f)
          (continue? #t))
      (while continue?
             (cond
               ;; Base Case 1:  LISTY empty
               ((null? listy)
                ;; Stop the WHILE loop
                (set! continue? #f)
                ;; Num not found
                (set! answer #f))
               ;; Base Case 2:  NUM found!
               ((= num (first listy))
                ;; Stop the WHILE loop
                (set! continue? #f)
                ;; INDY is where we found NUM
                (set! answer indy))
               ;; Recursive Case 1:  (FIRST LISTY) > NUM
               ((> (first listy) num)
                (printf "+")
                ;; Set vars for next iteration
                (set! listy (rest listy))
                (set! indy (+ indy 1)))
               ;; Recursive Case 2:  (FIRST LISTY) < NUM
               (else
                (printf "-")
                ;; Set vars for next iteration
                (set! listy (rest listy))
                (set! indy (+ indy 1)))))
      ;; After the WHILE loop:
      answer)))
```

The following interactions demonstrate that the recursive and iterative functions have the same behavior.

```
> (index-of-num-in-list 3 '(5 4 9 0 0 2 3 8 7 6 9))
+++---6
> (index-of-num-in-list-iter 3 '(5 4 9 0 0 2 3 8 7 6 9))
```

```
+++---6
> (index-of-num-in-list 3 '(2 5 1 6 3 8 8))
-+-+4
> (index-of-num-in-list-iter 3 '(2 5 1 6 3 8 8))
-+-+4
```

17.4 The `dotimes` Special Form

As its name suggests, the `dotimes` special form enables something to be done a certain number of times. In particular, for a given value n, the `dotimes` special form will evaluate the expressions in its body n times, once for each value in the set $\{0, 1, 2, \ldots, n-1\}$. The `dotimes` special form will be particularly useful for iteratively walking through *vectors,* to be discussed in the next chapter.

The syntax of the `dotimes` special form. The `dotimes` special form has the following syntax:

(`dotimes` (*var* *numExpr*)
  $expr_1$
  $expr_2$
  ...
  $expr_k$)

where:

- *var* is a symbol that will be the name of a counter variable in a local environment created by the `dotimes`;
- *numExpr* is any expression that evaluates to a non-negative integer, say, n, that will specify the number of *iterations* to be performed by the `dotimes`; and
- The expressions, $expr_1$, $expr_2$, ..., $expr_k$, are any k expressions that together constitute the *body* of the `dotimes`.

The semantics of the `dotimes` special form. A `dotimes` special form is evaluated as follows. First, the expression *numExpr* is evaluated, resulting in a non-negative integer n. Second, a local environment $\mathcal{E}$ is created containing a variable *var* whose value is initially set to zero. Next, the following steps are performed n times:

- The expressions, $expr_1$, $expr_2$, ..., $expr_k$, are evaluated with respect to that new local environment.
- The value of *var* in the local environment is incremented by one.

Thus, the expressions in the body of the `dotimes` are evaluated n times, once for each value of *var* in the range, $\{0, 1, 2, \ldots, n-1\}$. Note that the expressions in the body may refer to the variable *var*. When they are evaluated, the current value of *var* will be taken from the local environment.

Finally, the `dotimes` special form evaluates to *void.* Therefore, the usefulness of `dotimes` comes not from any output value, but from the side effects that occur by evaluating the expressions in its body with respect to the new local environment.

Example 17.4.1: Illustrating the `dotimes` special form

The following examples illustrate how the `dotimes` *special form can be used.*

```
> (dotimes (i 5)
           ;; The body:
           (printf "i: ~A~%" i))
i: 0
i: 1
```

```
i: 2
i: 3
i: 4
> (dotimes (i (+ 1 2))
           ;; The body:
           (printf "~A + ~A = ~A~%" i i (+ i i)))
0 + 0 = 0
1 + 1 = 2
2 + 2 = 4
```

In these examples, the body consists of a single expression; however, that need not be the case in general. Notice that in the first example, the numerical expression `5` *ensures that the expression in the body will be evaluated* five *times, once for each value of* `i` *in the range* $\{0, 1, 2, 3, 4\}$. *In the second example, the body of the* `dotimes` *is evaluated three times, since* `(+ 1 2)` *evaluates to* `3`.

Example 17.4.2: Implementing our own version of `dotimes`

The `my-dotimes` *function, defined below, implements essentially the same behavior as the* `dotimes` *special form. However, like* `my-while`*, defined earlier, it uses a lambda function, called* `body-func`*, to encapsulate the expressions in the body of the* `dotimes`.

```
;;  MY-DOTIMES
;; ------------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;           BODY-FUNC, a function that takes zero inputs
;;  OUTPUT:  void
;;  SIDE EFFECT:  Evaluates the expression (BODY-FUNC I)
;;    N times, once for each value of I in {0, 1, 2, ..., N-1}

(define my-dotimes
  (lambda (n body-func)
    (let ((i 0))
      (while (< i n)
             (body-func i)
             (set! i (1+ i))))))
```

Here is a simple demonstration of its behavior:

```
> (my-dotimes 4 (lambda (i)
                  (printf "--- i = ~A~%" i)))
--- i = 0
--- i = 1
--- i = 2
--- i = 3
```

17.5 The `dolist` Special Form

The `dolist` special form automates a kind of iteration that involves walking through a list. If you want to do something in response to each element of a list, then `dolist` may come in handy.

The syntax of the `dolist` special form. The syntax of the `dolist` special form is as follows:

```
(dolist (elt listExpr)
  expr_1
  expr_2
  ...
  expr_k)
```

where *elt* is any symbol expression, and `listExpr` is any Scheme expression that evaluates to a list. The expressions, $expr_1, \ldots, expr_k$, constitute the body of the `dolist` expression.

The semantics of the `dolist` special form. A `dolist` special form is evaluated as follows. First, *listExpr* is evaluated. It must evaluate to some list L; otherwise, there will be an error. Next, a local environment is created that contains a variable called *elt*. For each element of the list L, the expressions in the body of the `dotimes` are evaluated, in order, with respect to that local environment. (Typically, the expressions in the body contain references to the symbol *elt*.) Thus, if there are n elements in the list L, then the expressions in the body of the `dotimes` will be evaluated n times, once for each element of L.

Example 17.5.1

The following interactions demonstrate the semantics of the `dolist` *special form.*

```
> (dolist (elt '(a b c))
    (printf "elt: ~A~%" elt))
elt: a
elt: b
elt: c
> (dolist (elt (cons 1 (cons 2 (cons 3 ()))))
    (printf "elt: ~A" elt)
    (printf " <---~%"))
elt: 1 <---
elt: 2 <---
elt: 3 <---
```

⋆ Note that unlike many recursive functions on lists, there is no way to break out of a `dolist` before processing all of the elements in the given list.

Example 17.5.2: Implementing our own version of `dolist`

The following function provides the same functionality as the `dolist` *special form, except that the expressions in the body are encapsulated within a* `lambda` *function that expects a single input,* `elt`.

```
;;  MY-DOLIST
;; ------------------------------------------------------------
;;  INPUTS:  LISTY, a list
;;           BODY-FUNC, a function that takes a single input,
;;             for example, any element of LISTY
;;  OUTPUT:  void
;;  SIDE EFFECT:  For each element ELT of LISTY, MY-DOLIST
;;                applies BODY-FUNC to ELT.

(define my-dolist
  (lambda (listy body-func)
```

```
        (while (not (null? listy))
               (body-func (first listy))
               (set! listy (rest listy)))))
```

Here are some examples of its use.

```
> (my-dolist '(a b c)
             (lambda (elt)
               (printf "elt: ~A~%" elt)))
elt: a
elt: b
elt: c
> (my-dolist (cons 1 (cons 2 (cons 3 ())))
               (lambda (elt)
                 (printf "elt: ~A" elt)
                 (printf " <---~%")))
elt: 1 <---
elt: 2 <---
elt: 3 <---
```

17.6 Summary

Write the summary!

Special Forms Introduced in this Chapter

`set!`	Destructively modify the value of a variable
`while`	As long as some condition holds, evaluate expressions in body
`dotimes`	Evaluate expressions in the body n times, once for each value in $\{0, 1, \ldots, n-1\}$
`dolist`	Evaluate expressions in the body, once for each element of a given list

Chapter 18

Vectors

A *vector* is a data structure that, like a list, contains an ordered sequence of data. However, there are many significant differences between vectors and lists.

Recall that lists can be incrementally extended, one element at a time, using the `cons` function. As a result, as illustrated in Fig. 18.1, the individual *cons cells* in which the various elements of a list are stored frequently end up being scattered haphazardly about the computer's memory. For this reason, accessing elements of a list can be relatively slow. For example, to access the one-thousandth element of a list typically requires recursively walking through the first thousand cons cells in the chain. However, as has been demonstrated repeatedly in previous chapters, much recursive processing of lists can be done while only accessing the *first* and *rest* of a list. For these sorts of applications, lists are quite handy. Furthermore, many list-based functions can be written non-destructively, which facilitates testing and debugging, since the performance of a non-destructive function depends only on the code within its body.

In sharp contrast to lists, vectors are stored within a single, contiguous block of memory—which has both advantages and disadvantages. The principal advantage is that every item of a vector can be accessed very quickly, whether the first element or the thousandth. To demonstrate why, Fig. 18.2 illustrates the layout of the elements in a typical vector. Notice that each slot has a numerical *index,* starting at zero. A crucial feature of the layout is that each slot takes up the same amount of space (e.g., four bytes per slot in the figure). Since each slot has the same size, the memory address of any given slot is very easy to compute. For example, if the start of the vector is located at memory address 1000, and each slot is four bytes wide, then the address of the i^{th} slot is $1000 + 4 * i$. (Thankfully, DrScheme takes care of such low-level details. A Scheme programmer need never deal directly with memory addresses.) An important feature of the "Random Access Memory" (RAM) found in modern computers is that the contents of any memory address can be fetched in the same (very small) amount of time. (The term "random" here is used to indicate that the contents of any randomly selected memory address can be fetched in the same, very small amount of time.) Thus, the time required to fetch the first element of a vector (i.e., the element with index 0) is the same as the time required to fetch the thousandth element.

One disadvantage of vectors is that the block of memory that will hold the vector's elements must be allocated in one chunk. As a result, vectors cannot be easily extended to accommodate new elements. In particular, if a vector is created to hold up to N elements, then it will never be able to hold more than N elements. There is no analog to the `cons` function for vectors. Each vector has a fixed *length.*

Since not all of a vector's elements may be known at the time the block of memory is allocated, *destructive programming* is typically used with vectors. For example, a Scheme function might decide to set the 24th element of a vector to be `#t`, then later set the 36th element to be `34.2`, and still later set the 24th element to be `#f` (erasing the prior contents of the 24th slot). Just as accessing any desired slot of a vector is very efficient, so too is the operation of destructively modifying the contents of any desired slot. (Again, this is a feature of Random Access Memory.) Thus, the speed of accessing and modifying the contents of a vector is balanced by the challenge of using destructive programming, which can make testing and debugging your functions a little more difficult. Nonetheless, the tradeoff can be quite worthwhile; and if care is taken, the risks associated with using destructive programming can be mitigated.

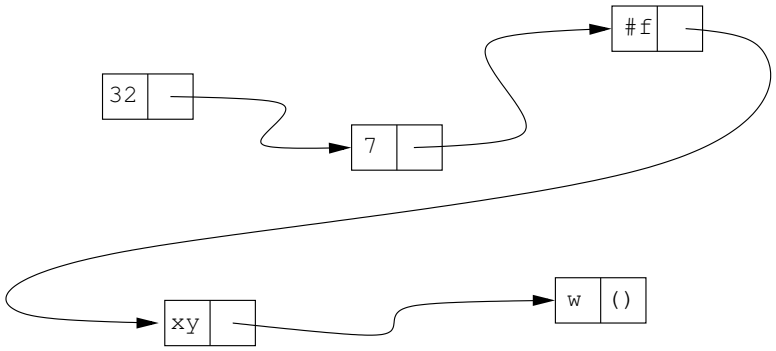

Figure 18.1: A scattered chain of cons cells representing the list `(32 7 #f xy w)`

Index	Element	Memory Address
0	32	1000
1	#t	1004
2	()	1008
3	3.14	1012
⋮	⋮	⋮
i	26	1000 + i*4
⋮	⋮	⋮

Note: This example presumes that each element of the vector occupies four bytes of memory.

Figure 18.2: Accessing the elements of a vector by their numerical indices

18.1 Vector Expressions

Chapter 2 introduced several types of primitive data expressions, including numbers, booleans, the empty list, and symbols. For each type of expression, the syntactic rules for character sequences denoting that type of data were described. Then, Chapter 3 described how instances of those data types are evaluated. Similarly, Chapter 6 presented the syntactic rules for character sequences that denote non-empty lists, and described how the Default Rule evaluates non-empty lists. Following this trend, this section presents the `#(...)` syntax—called the *pound* syntax—for character sequences that denote vectors, as well as the semantics of evaluation for vectors. (Vectors evaluate to themselves.) It should be noted, however, that the *pound* syntax has limited use because the vectors it denotes are *immutable* (i.e., their contents cannot be changed). The *pound* syntax is typically only useful for testing functions that look at the contents of vectors, and perhaps do some computations based on those contents, but do not try to change the contents in any way. After presenting the *pound* syntax, the next section presents built-in functions for creating new vectors, which turn out to be more useful in practice.

A vector is a datum. Scheme provides a special syntax, the *pound* syntax, for denoting vectors. Just as the expression `(a b c)` can be used to represent (or denote) a list containing the three elements `a`, `b` and `c`, the expression `#(a b c)` can be used to represent (or denote) a *vector* containing the three elements `a`, `b` and `c`. In general, any expression of the form `#(`$e_1, e_2, \ldots, e_n$`)` can be used to represent a vector containing the n elements denoted by the expressions $e_1, e_2, \ldots, e_n$. Thus, for example, the expression `#(x #t () 32)` denotes a vector containing four elements: a symbol, a boolean, the empty list, and a number.

One important fact about the `#(...)` syntax is that the vectors it represents are *immutable* (i.e., their contents cannot be changed). This limits the usefulness of the `#(...)` syntax. However, it can be useful when testing functions that don't need to modify their vector inputs (e.g., functions that print out the contents of a vector).

⋆ Unlike lists, *vectors evaluate to themselves!*

Example 18.1.1: Demonstrating that vectors evaluate to themselves

The following interactions use the `#(...)` *syntax to demonstrate that vectors evaluate to themselves.*

```
> #(1 2 a b #t (+ 2 3))      ⟵ The pound syntax denotes a vector
#(1 2 a b #t (+ 2 3))        ⟵ That vector evaluates to itself!
> #(a b #(c d e) (x y z))
#(a b #(c d e) (x y z))
```

With the `#(...)` *syntax, there is no need to quote the subsidiary expressions. For example,* `#(a b c)` *simply denotes the vector containing the three symbols* `a`*,* `b` *and* `c`*. When the vector gets evaluated, its elements are* not *evaluated! The vector simply evaluates to itself—without evaluating any of its elements. The Default Rule for evaluating non-empty lists does not apply to vectors!*

Incidentally, DrScheme uses the `#(...)` syntax to report results that are vectors, whether they are immutable or mutable.

⋆ Remember: Vectors created with the `#` syntax are *immutable!* Once created, their contents can't be changed.

18.2 Constructing Vectors

Scheme provides two built-in functions, called `vector` and `make-vector`, that can be used to create new vectors. The `vector` function is similar to the built-in `list` function, mentioned briefly in Section 16.1. The `make-vector` function is typically the most practical. It enables the programmer to create a vector of any specified length.

18.2.1 The Built-in `vector` Function

The `vector` function is a built-in function that works very much like the built-in `list` function, mentioned briefly in Section 16.1. Whereas `list` constructs a list containing the specified elements, `vector` constructs a vector containing the specified elements. Unlike the vectors denoted by the `#(...)` syntax, vectors created by the `vector` function are *mutable* (i.e., their contents *can* be changed—we'll see how momentarily).

Example 18.2.1: Using the `vector` function

Here's the contract for the `vector` *function, followed by some examples of its use.*

```
;;  VECTOR  --  built-in function
;; ------------------------------------------------------------
;;  INPUTS:  ELT1, ELT2, ELT3, ... :  any number of inputs
;;  OUTPUT:  A *mutable* vector containing the specified elements

> (vector 1 (+ 2 3) 'a)
#(1 5 a)
> (list 1 (+ 2 3) 'a)
(1 5 a)
> (define vecky (vector 1 (+ 2 3) 'a))
> vecky
#(1 5 a)
> (define vecky-two (vector 100 vecky 200))
> vecky-two
#(100 #(1 5 a) 200)
```

Note that since `vector` *is a built-in function, expressions such as* `(vector 1 (+ 2 3) 'a)` *are evaluated by the Default Rule. Thus, all input expressions are evaluated* before *being passed as input to the* `vector` *function—which is why the* `a` *in the first two examples had to be quoted.*

18.2.2 The `make-vector` Function

The `make-vector` function is a built-in function that can be used to create a vector of any specified length. It is the most common way of creating a vector because it can be used, for example, to easily create a vector with, say, 1000 slots. Like with the `vector` function, vectors created by `make-vector` are *mutable* (i.e., their contents can be changed). By default, `make-vector` inserts the value `0` into each slot of the vector it creates.

Example 18.2.2: Using the `make-vector` function

Here's the contract for the `make-vector` *function, followed by some examples of its use.*

```
;;  MAKE-VECTOR -- built-in function
;; ------------------------------------------------------------
;;  INPUT:   NUM-SLOTS, a non-negative integer
;;  OUTPUT:  A vector with NUM-SLOTS slots, each initially 0

> (make-vector 5)
#(0 0 0 0 0)
> (make-vector 15)
#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

18.3 Accessing Information Stored in a Vector

To be of any use, the information stored in a vector must be accessible to the programmer. Scheme provides the `vector-ref` function to access the contents of any specified slot in a vector, and the `vector-length` function to access the (fixed) number of elements in a vector.

18.3.1 The `vector-ref` Function

Each element of a vector is identified by its numerical index. The built-in `vector-ref` function provides an easy way to access any element of a vector by its index.

Example 18.3.1: Using `vector-ref`

Here is the contract for the `vector-ref` *function, followed by some examples of its use.*

```
;;  VECTOR-REF  --  built-in function
;; ------------------------------------------------
;;  INPUTS:  VECKY, a vector
;;           INDY, an index
;;  OUTPUT:  The item of VECKY stored at slot INDY

> (define vecky #(a b c d e))
> (vector-ref vecky 0)
a
> (vector-ref vecky 2)
c
```

⋆ Although the `vector-ref` and `list-ref` functions may appear similar syntactically, they operate quite differently. The `vector-ref` function accesses the specified element of a vector nearly instantaneously, while the `list-ref` function walks through each element of the specified list until it finds the one with the desired index. Thus, `vector-ref` is much more efficient than `list-ref`.

18.3.2 The `vector-length` Function

As already mentioned, each vector has a fixed length. For this reason, the length of the vector can be stored with the vector itself, when it is created. Thus, the built-in `vector-length` function does *not* need to walk through the entire vector to figure out how long it is; instead it can simply look up the *length* information that is stored with the vector. (The `length` of a vector is an example of a *field* in a data structure, which will be discussed in the next chapter.) Thus, using `vector-length` is *very fast* with any vector, no matter how long. This is quite different from the built-in `length` function for lists which must walk all the way through a list in order to figure out how many elements it has.

Example 18.3.2: Using `vector-length`

Here is the contract for `vector-length`*, followed by some examples of its use.*

```
;;  VECTOR-LENGTH  -- built-in function
;; ------------------------------------------------------
;;  INPUT:   VECKY, a vector
;;  OUTPUT:  The number of elements/slots in that vector

> (define vecky #(a b c d e))
> (vector-length vecky)
5
> (vector-length (make-vector 25))
25
```

In-Class Problem 18.3.1: Fetching a random element from a vector

Define a function, called `fetch-random-element`*, that satisfies the following contract:*

```
;;  FETCH-RANDOM-ELEMENT
;; ------------------------------------------------------------
;;  INPUT:   VECKY, a vector
;;  OUTPUT:  One of the elements of VECKY, chosen at random
```

Here are some examples of the desired behavior:

```
> (fetch-random-element #(a b c d e f))
c
> (fetch-random-element #(a b c d e f))
a
> (fetch-random-element #(a b c d e f))
d
```

Hint: Use the built-in `vector-length`*,* `random` *and* `vector-ref` *functions.*

18.4 Recursively Processing Vectors

Since a vector is an ordered sequence of elements, we can define recursive functions to walk through vectors in much the same way that we defined recursive functions to walk through lists. One example of this will be given below. However, writing recursive functions to walk through vectors would quickly grow tiresome because it would involve repeating the same kind of pattern over and over again. To avoid this kind of repetition, we will instead use the `dotimes` special form to enable us to iteratively walk through any vector. However, before introducing this use of `dotimes`, we first demonstrate how to manually write a recursive function to walk through some or all of a vector.

Example 18.4.1: Manually walking through a vector

The following function prints out the contents of a vector from a given starting index. On each recursive function call, the value of the index is incremented, until it goes past the end of the vector. Recall that if a vector has length n, then the legal indices range from 0 to $n-1$.

```
;;  PRINT-VECTOR-FROM
;; ------------------------------------------------
;;  INPUTS:  VECKY, a vector
;;           I, a non-negative integer, no greater than
;;              the length of VECKY
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints out the elements of VECKY
;;                from index I onward.

(define print-vector-from
 (lambda (vecky i)
   (cond
    ;; Base Case:  I is too big
    ;;  Note:  The last legal index of VECKY is LENGTH-1
    ((>= i (vector-length vecky))
     (void))
    ;; Recursive Case:  I is a legal index
    (else
     ;; Print one element
     (printf "Element ~A of VECKY is: ~A~%" i (vector-ref vecky i))
     ;; Let recursion print the rest of the elements
     (print-vector-from vecky (+ i 1))))))

> (print-vector-from #(a b c d) 0)
Element 0 of VECKY is: a
Element 1 of VECKY is: b
Element 2 of VECKY is: c
Element 3 of VECKY is: d
> (print-vector-from #(a b c d) 2)
Element 2 of VECKY is: c
Element 3 of VECKY is: d
```

The wrapper function, `print-vector-wr`*, can then be defined as follows:*

```
;;  PRINT-VECTOR-WR
;; ------------------------------------------------
;;  INPUT:   VECKY, a vector
;;  OUTPUT:  None
```

```
;;  SIDE EFFECT:  Prints out the elements of VECKY

(define print-vector-wr
  (lambda (vecky)
    (print-vector-from vecky 0)))

> (print-vector-wr #(a b c))
a
b
c
```

⋆ In general, any function that needs to recursively process the elements of a vector can do so by defining a helper function that includes an extra input `i` whose value is initially zero and increments by one on each recursive function call until it exceeds the last legal index for the given vector.

However, the `dotimes` special form simplifies the task of walking through a vector.

Example 18.4.2: Printing the contents of a vector using `dotimes`

The `print-vector` function, below, illustrates the use of the `dotimes` special form to walk through a vector, printing out its contents. Note the use of the `vector-length` function to specify the number of iterations to perform.

```
;;  PRINT-VECTOR
;; ------------------------------------------------------------
;;  INPUT:   VECKY, a vector
;;  OUTPUT:  None
;;  SIDE EFFECT:  Displays the contents of VECKY in the
;;                Interactions Window, one element per row

(define print-vector
  (lambda (vecky)
    ;; I takes on the values: 0, 1, 2, ..., LENGTH-1
    (dotimes (i (vector-length vecky))
             ;; Print out the Ith element of VECKY
             (printf "~A: ~A~%" i (vector-ref vecky i)))))

> (define vecky #(a b c d e))
> (print-vector vecky)
a
b
c
d
e
```

Example 18.4.3: Printing the contents of a vector in reverse order

The following function prints out the contents of a given vector in reverse order. Notice the use of the local variable `rev-i`*, whose value is the index of the next element to be printed. For example, you should convince yourself that for a vector of length four, the counter variable* `i` *will range from* `0` *to* `3`*, while the local variable* `rev-i` *will range from* `3` *to* `0`*.*

```
;;  PRINT-IN-REVERSE
;; ------------------------------------------------------------
;;  INPUT:   VECKY, a vector
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints out the contents of VECKY
;;                in reverse order

(define print-in-reverse
  (lambda (vecky)
    ;; LEN = number of elements in VECKY
    (let ((len (vector-length vecky)))
      ;;   I  :  takes on the values from 0 to LEN-1
      ;; REV-I:  takes on the values from LEN-1 to 0
      (dotimes (i len)
               ;; The BODY:
               (let ((rev-i (- len i 1)))
                 (printf "vecky[~A] = ~A~%" rev-i
                         (vector-ref vecky rev-i))))
      (void))))

> (print-in-reverse #(a b c d))
vecky[3] = d
vecky[2] = c
vecky[1] = b
vecky[0] = a
```

In-Class Problem 18.4.1: Using `dotimes` to print out every other element of a vector

Define a function, called `print-every-other-one-veck`*, that takes a vector as its only input. It should not return any output value. Instead, it should print out every other element of the given vector. Here is the contract, followed by some examples:*

```
;;  PRINT-EVERY-OTHER-ONE-VECK
;; ------------------------------------------------------
;;  INPUT:   VECK, a vector
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints out every other element of VECK

> (print-every-other-one-veck #(a b c d e))
a
c
e
> (print-every-other-one-veck #(a a b b c c d d))
a
b
```

```
c
d
```

Hint: Use the `even?` *function. If the current index is even, then print out the corresponding element.*

In-Class Problem 18.4.2: Testing whether two vectors are "equal"

Define a function, called `vector-equal?`*, that satisfies the following contract:*

```
;;  VECTOR-EQUAL?
;; -----------------------------------------------------
;;  INPUTS:  VECK-ONE and VECK-TWO, any vectors
;;  OUTPUT:  #t if VECK-ONE and VECK-TWO have the same
;;    elements, in the same order; #f otherwise.
```

Here are some examples:

```
> (vector-equal? #(a b c) #(a b c))
#t
> (vector-equal? (make-vector 3) (vector 0 0 0))
#t
> (vector-equal? #(a b c) #(a b c d))
#f
```

Notice that the two input vectors cannot be equal if they have different lengths. Therefore, `vector-equal?` *can immediately return* `#f` *if the two vectors have different lengths. On the other hand, if they do have the same length, then it can call a helper function,* `vector-equal?-helper`*, to manually walk through the vectors in parallel, comparing their corresponding elements. Note that using* `dotimes` *is not an option for this problem because the helper function should be able to stop early if it ever discovers corresponding elements that are not the same. Here's the contract for the helper function:*

```
;;  VECTOR-EQUAL?-HELPER
;; ------------------------------------------------------------
;;  INPUTS:  VECK-ONE, VECK-TWO, two vectors of the same length
;;           I, an index
;;  OUTPUT:  #t if the corresponding elements of VECK-ONE and
;;             VECK-TWO are the same from index I onward.
;;           #f otherwise.
```

Notice that the helper function is only ever called on vectors having the same length.

After you've implemented this function, you may wish to know that the built-in `equal?` *function can be used to test the equality of vectors, as illustrated below.*

```
> (equal? #(1 2 3) #(1 2 3))
#t
> (equal? #(a b c) (vector 'a 'b 'c))
#t
> (equal? #(0 0 0 0) (make-vector 4))
#t
```

18.5 Destructively Modifying a Vector

The `vector-set!` function is provided to enable a programmer to destructively modify the contents of a specified slot in a vector.

⋆ The name of the function ends with an exclamation point to remind us of its destructive side effect.

Example 18.5.1: The `vector-set!` function

Here is the contract for the `vector-set!` *function, followed by an example of its use.*

```
;;  VECTOR-SET!  --  Built-in Function
;; ----------------------------------------
;;  INPUTS:  VECKY, a vector
;;           INDY, a numerical index
;;           NEW-VAL, anything
;;  OUTPUT:  *void*
;;  SIDE EFFECT:  Destructively changes the contents of VECKY
;;    at slot INDY to become NEW-VAL

> (define vecko (vector 0 10 20 30))
> vecko
#(0 10 20 30)
> (vector-set! vecko 2 'x)
> vecko
#(0 10 x 30)
```

In-Class Problem 18.5.1: Initializing a vector

Define a function, called `init-veck`*, that satisfies the following contract:*

```
;;  INIT-VECK
;; ----------------------------------------------------
;;  INPUT:   VECK, a vector
;;  OUTPUT:  Don't care
;;  SIDE EFFECT:  Sets the value of slot 0 to 0, the value
;;    of slot 1 to 1, and so on.
```

Here is an example of its use:

```
> (define vecky (make-vector 5))
> vecky
#(0 0 0 0 0)
> (init-veck vecky)
> vecky
#(0 1 2 3 4)
```

Hint: Use `dotimes` *and* `vector-set!`.

In-Class Problem 18.5.2: Swapping elements of a vector

Define a destructive function, called `vector-swap!`*, that destructively modifies a vector by swapping two of its elements as specified by the following contract:*

```
;;  VECTOR-SWAP!
;; ---------------------------------------------
;;  INPUTS:  VECKY, a vector
;;           I, J, two numerical indices
;;  OUTPUT:  don't care
;;  SIDE EFFECT:  Destructively swaps the contents of VECKY
;;                  at slots I and J.
```

Here are some examples of its use:

```
> (define vecky (vector 'a 'b 'c 'd 'e 'f))
> vecky
#(a b c d e f)
> (vector-swap! vecky 1 4)
> vecky
#(a e c d b f)
> (vector-swap! vecky 0 4)
> vecky
#(b e c d a f)
```

18.6 Summary

Write the summary!

Built-In Functions Introduced in this Chapter

`make-vector`	Construct a vector of a specified length
`vector`	Construct a vector containing the specified items
`vector-length`	Fetches the length of a given vector
`vector-ref`	Fetches a specified element of a given vector
`vector->list`	Convert vector into a list (cf. Problem 18.10)
`list->vector`	Convert list into a vector (cf. Problem 18.11)

Chapter 19

Data Structures

The early chapters of this book introduced several kinds of primitive data in Scheme, including numbers, booleans, the empty list, void, and symbols. Each instance of primitive data is indivisible in the sense that it does not have any parts that the programmer can access or modify. In contrast, a *data structure* is an organized collection of data whose parts the programmer can access or modify. Data structures come in many varieties. For example:

- A *vector* is an example of a data structure whose slots are accessed by their numerical indices. Such data structures may be called *index-based* data structures.
- A *cons cell* is an example of a data structure whose slots are accessed by *name.* The named slots in such a data structure are called *fields;* and the data structures are called *field-based* data structures. For example, the fields in a cons cell are called *first* and *rest.*[1]
- A *list* is an example of a *composite* data structure that is *recursively defined.* In particular, a list is defined by the following two rules:

  (Base Case) The empty list is a list.

  (Recursive Case) A cons cell whose *rest* field contains a list is a list.

  The recursive nature of lists is what enables us to define recursive functions that process any list, no matter how complicated.

This chapter focuses on *field-based* data structures.

Example 19.0.1: Motivating field-based data structures, I

Suppose you wanted to write a program that needed to represent dates, such as October 22, 1958 *or* December 7, 1941. *Since each date includes a month, a day, and a year, you could easily store each date in a vector with three slots. However, to help distinguish your dates-as-vectors from other small vectors, you might want to include an extra slot whose value would be some easily recognizable symbol, such as* `i-am-a-date!`. *Using this approach, the above-mentioned dates might be represented by the following vectors:*

0	i-am-a-date!
1	1958
2	10
3	22

0	i-am-a-date!
1	1941
2	12
3	7

The following interactions demonstrate how you might use such dates-as-vectors:

```
> (define my-birthday (vector 'i-am-a-date! 1958 10 22))
```

[1] Although a vector is primarily an index-based data structure, it also contains a field, called *length,* whose value is accessed by the `vector-length` function.

```
> my-birthday
#(i-am-a-date! 1958 10 22)
> (vector-ref my-birthday 2)
10
> (vector-ref my-birthday 1)
1958
```

Notice that to access a particular item requires knowing which index to use (e.g., 2 *for month, and* 1 *for year).*

The following example takes a more structured approach to storing date information in vectors.

Example 19.0.2: Motivating field-based data structures, II

First, we define some useful constants.

```
;; The unique identifier for dates

(define *date-id* 'i-am-a-date!)

;; Names for the various indices

(define *date-year-index* 1)
(define *date-month-index* 2)
(define *date-day-index* 3)

;;  *MONTHS*  --  A vector of names for the months
;; ------------------------------------------------
;;  To enable using the standard numbers for months,
;;  we use a vector of length 13, ignoring slot 0.
;;  And, since the names of the months won't change,
;;  we can use an *immutable* vector.

(define *months*
  #(0 Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))
```

Next, we define a constructor *function (i.e., a function that creates an instance of a* date-as-vector.

```
;;  MAKE-DATE  --  a constructor function
;; ------------------------------------------------------------
;;  INPUTS:  YEAR, an integer
;;           MONTH, an integer between 1 and 12
;;           DAY, a positive integer between 1 and 31
;;  OUTPUT:  A "date-as-vector" containing the given information

(define make-date
  (lambda (year month day)
    (vector *date-id* year month day)))
```

Although it may not be used all the much, here's a type-checker predicate for a date-as-vector.

```
;;  DATE?  --  type-checker predicate for dates
```

```
;; ---------------------------------------------------
;;  INPUT:  DATUM, anything
;;  OUTPUT:  #t, if DATUM is a "date-as-vector"

(define date?
  (lambda (datum)
    ;; Output #t if DATUM is a vector with four slots,
    ;; and whose zeroeth slot contains the *DATE-ID*:
    (and (vector? datum)
         (= 4 (vector-length datum))
         (eq? *date-id* (vector-ref datum 0)))))
```

Next, we define some accessor functions *(i.e., functions that enable us to access the information stored in a* date-as-vector).

```
;;  DATE-YEAR  --  Accessor Function
;; ------------------------------------------------
;;  INPUT:   DATEY, a "date-as-vector"
;;  OUTPUT:  The year stored in DATEY

(define date-year
  (lambda (datey)
    (vector-ref datey *date-year-index*)))

;;  DATE-MONTH  --  Accessor Function
;; ------------------------------------------------
;;  INPUT:   DATEY, a "date-as-vector"
;;  OUTPUT:  The month stored in DATEY

(define date-month
  (lambda (datey)
    (vector-ref datey *date-month-index*)))

;;  DATE-DAY  --  Accessor Function
;; ------------------------------------------------
;;  INPUT:   DATEY, a "date-as-vector"
;;  OUTPUT:  The day stored in DATEY

(define date-day
  (lambda (datey)
    (vector-ref datey *date-day-index*)))
```

And, finally, some mutator functions *(i.e., functions that enable us to destructively modify the contents of a* date-as-vector).

```
;;  SET-DATE-YEAR!  --  Mutator Function
;; ------------------------------------------------------
;;  INPUTS:  DATEY, a "date-as-vector"
;;           NEW-VAL
;;  OUTPUT:  Don't care
;;  SIDE EFFECT:  Destructively modifies the contents of
;;                the YEAR slot to be NEW-VAL.
```

```
(define set-date-year!
  (lambda (datey new-val)
    (vector-set! datey *date-year-index* new-val)))

;;  SET-DATE-MONTH!  --  Mutator Function
;; ------------------------------------------------------
;;  INPUTS:  DATEY, a "date-as-vector"
;;           NEW-VAL
;;  OUTPUT:  Don't care
;;  SIDE EFFECT:  Destructively modifies the contents of
;;                the MONTH slot to be NEW-VAL.

(define set-date-month!
  (lambda (datey new-val)
    (vector-set! datey *date-month-index* new-val)))

;;  SET-DATE-DAY!  --  Mutator Function
;; ------------------------------------------------------
;;  INPUTS:  DATEY, a "date-as-vector"
;;           NEW-VAL
;;  OUTPUT:  Don't care
;;  SIDE EFFECT:  Destructively modifies the contents of
;;                the DAY slot to be NEW-VAL.

(define set-date-day!
  (lambda (datey new-val)
    (vector-set! datey *date-day-index* new-val)))
```

The following interactions demonstrate the use of dates-as-vectors *to store date information:*

```
> (define my-birthday
    (make-date 1958 10 22))          ←— Creating a date
> (date? my-birthday)                ←— Using the type checker
#t
> (date? (make-date 1941 12 7))
#t
> (date? (make-vector 4))
#f
> (date-year my-birthday))           ←— Accessing the contents
1958
> (date-month my-birthday)
10
> (date-day my-birthday)
22
> (set-date-month! my-birthday 11)   ←— Changing the contents
> my-birthday
#(i-am-a-date! 1958 11 22)
> (set-date-year! my-birthday 1968)
> my-birthday
#(i-am-a-date! 1968 11 22)
```

19.1 The `define-struct` Special Form

The preceding examples demonstrate that vectors can be used as data structures to store and manage any kinds of data, and that a variety of functions can be defined to facilitate common tasks associated with using these vectors-as-data-structures: constructing new instances of the data structures, and accessing or mutating their contents. Although these functions are not complicated, providing their definitions is both time-consuming and unilluminating. For this reason, Scheme provides a special form, called `define-struct`, that makes defining new data structures quite easy. In particular, evaluating a `define-struct` special form results in the automatic definition of all the needed constructor, accessor and mutator functions.

A `define-struct` special form has the following syntax:

`(define-struct` *structName* `(`*fname*$_1$ *fname*$_2$ ... *fname*$_k$`))`

where:

- *structName* is a symbol that will be the name of the data structure; and
- *fname*$_1$, *fname*$_2$, ..., *fname*$_k$ are k symbols that will be the names of the *fields* of the new data structure.

The semantics of the `define-struct` special form stipulates that its evaluation generates no output, but has the side effect of defining all of the following functions:

- a constructor function, named `make-`*structName*;
- a type-checker predicated, named *structName*`?`;
- k accessor functions, named *structName*–*fname*$_1$, ..., *structName*–*fname*$_k$; and
- k mutator functions, named `set-`*structName*–*fname*$_1$`!`, ..., `set-`*structName*–*fname*$_k$`!`.

Example 19.1.1: Using `define-struct` to define a `date` data structure

The following expression is all that is needed to define a `date` *data structure in Scheme:*

```
(define-struct date (year month day))
```

Its evaluation generates all of the following functions:

- *Constructor:* `make-date`
- *Type checker:* `date?`
- *Accessors:* `date-year`, `date-month`, `date-day`
- *Mutators:* `set-date-year!`, `set-date-month!`, `set-date-day!`

The following interactions demonstrate their use:

```
> (define-struct date (year month day))
> (define my-bday (make-date 1958 10 22))
> (date-year my-bday)
1958
> (date-month my-bday)
10
> (set-date-year! my-bday 1978)
> (date-year my-bday)
1978
>my-bday
#<date>
```

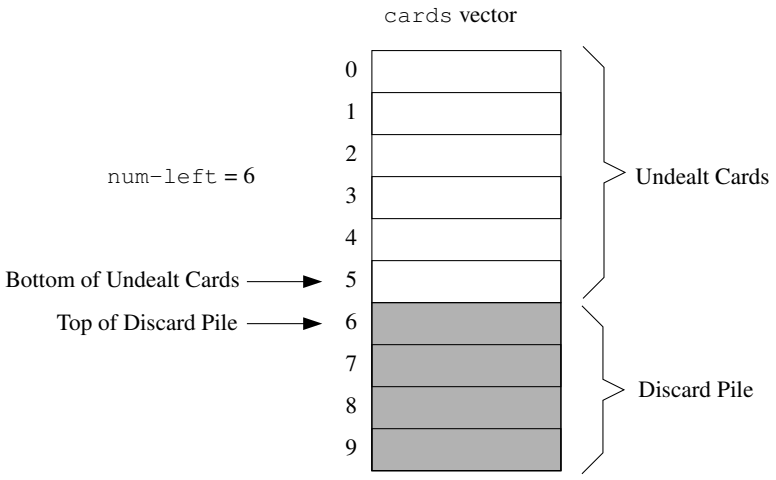

Figure 19.1: Partitioning a vector of cards

The last expression demonstrates that DrScheme is not terribly helpful when displaying instances of our new data structure: it just reports that it is such an instance. However, we can define our own display function, as follows.

```
;;  PRINT-DATE
;; -------------------------------------------------------
;;  INPUT:   DATEY, an instance of a DATE data structure
;;  OUTPUT:  None
;;  SIDE EFFECT:  Displays the date stored in DATEY in
;;   the following format:  Month Day, Year.

(define print-date
  (lambda (datey)
    (printf "~A ~A, ~A~%"
            ;; Fetch name of month from *MONTHS* vector
            (vector-ref *months* (date-month datey))
            (date-day datey)
            (date-year datey))))
```

Here are some examples of its use:

```
> (print-date my-bday)
Oct 22, 1958
> (print-date (make-date 1941 12 7))
Dec 7, 1941
```

Example 19.1.2: Implmenting a deck of cards

Below, we define a `deck` *data structure that has two fields:* `cards` *and* `num-left`*. The* `cards` *field will be a vector that is partitioned into two sections, as illustrated in Fig. 19.1. The first section will contain all of the cards that have not yet been dealt; the second section will contain those that have already been*

dealt. Thus, the second section of the vector is effectively a "discard pile". The `num-left` *field serves two purposes. First, it keeps track of the number of cards that have not yet been dealt (i.e., that remain available for dealing). Second, it corresponds to the index at the top of the discard pile. For example, as shown in the figure, suppose that there are ten cards in the deck and* `num-left` *is* `6`*. This means that there are six cards that have not yet been dealt. Those cards are located in the first part of the* `cards` *vector, at positions* `0`, `1`, ..., `5`*. The "discard pile" begins at index* `6`*. The four cards in the discard pile are at positions* `6`, `7`, `8` *and* `9`*. In general, the cards available for dealing have indices between* 0 *and* (`num-left` − 1)*, while the discard pile starts at index* `num-left`*.*

The `deck` *data structure is defined below. Notice that it is preceded by a block of comments that together specify the name of the data structure and, for each field, the name of that field along with a brief explanation of what that field is for.*

```
;;  A DECK data structure
;; ------------------------------------------------------
;;    CARDS:     A vector of cards
;;    NUM-LEFT:  Number of cards still left in the deck
;;                 (i.e., still available for dealing)

(define-struct deck (cards num-left))
```

For testing purposes, we define a `print-deck` *function that prints out the entire contents of a* `deck` *data structure. Later on (e.g., when using the* `deck` *data structure as part of a program that implements a card game) we might define a different function that does not show the players the order of the cards in the deck!*

```
;;  PRINT-DECK
;; ---------------------------------------------
;;  INPUT:   DECKY, a DECK data structure
;;  OUTPUT:  None
;;  SIDE EFFECT:  Displays contents of DECKY

(define print-deck
  (lambda (decky)
    (printf "Deck of cards: ~A, num-left: ~A~%"
            (deck-cards decky) (deck-num-left decky))))
```

Next, we define a destructive function, called `deal!`*, that deals one randomly chosen card from the deck. To preserve the partition between undealt cards and the discard pile, the* `deck` *data structure is modified as follows. First, the randomly selected card (i.e., the one to be dealt) is swapped with the card at the bottom of the* not-yet-dealt *partition (i.e., the card whose index is* (`num-left` − 1)*). Next, the value of the* `num-left` *field is decremented. In this way, the newly dealt card is effectively moved to the "top" of the "discard pile".*

```
;;  DEAL!
;; -----------------------------------------------------
;;  INPUT:  DECKY, a DECK data structure
;;  OUTPUT:  One of the cards from DECKY, selected
;;           at random
;;  SIDE EFFECT:  Modifies DECKY so that the newly dealt
;;    card joins the "discard" pile (i.e., becomes dealt)
;;    at the end of the CARDS vector.
```

```
(define deal!
  (lambda (decky)
    (let* (;; The number of cards left in the deck
           (num-lefty (deck-num-left decky))
           ;; A random index into the cards still left in the deck
           (rnd-indy (random num-lefty))
           ;; The CARDS vector from DECKY
           (veck-o-cards (deck-cards decky))
           ;; The card we shall output
           (dealt-card (vector-ref veck-o-cards rnd-indy))
           ;; The index of card to be swapped with DEALT-CARD
           (indy-to-move (- num-lefty 1))
           ;; The card to be moved
           (card-to-move (vector-ref veck-o-cards indy-to-move)))
      ;; Swap DEALT-CARD and CARD-TO-MOVE
      ;; (i.e., move newly dealt card into discard pile)
      (vector-set! veck-o-cards rnd-indy card-to-move)
      (vector-set! veck-o-cards indy-to-move dealt-card)
      ;; Decrement the NUM-LEFT field
      (set-deck-num-left! decky indy-to-move)
      ;; Output:
      dealt-card)))
```

Because the `deal!` *function randomly selects cards from those that have not yet been dealt, there is no need to simulate any "shuffling"; instead, shuffling the deck is simply a matter of resetting the* `num-left` *field so that all cards become available for dealing.*

```
;;  SHUFFLE!
;; -------------------------------------------------------
;;  INPUT:   DECKY, a DECK data structure
;;  OUTPUT:  None
;;  SIDE EFFECT:  Makes all cards available for dealing

(define shuffle!
  (lambda (decky)
    ;; Reset the NUM-LEFT field to its initial value
    (set-deck-num-left! decky (vector-length (deck-cards decky)))
    ;; Output the modified DECK
    decky))
```

The following interactions demonstrate the use of the `deck` *data structure, using a test deck that contains only ten cards.*

```
> (define decky
    (make-deck (vector 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j) 10))
> (print-deck decky)
Deck of cards: #(a b c d e f g h i j), num-left: 10
> (deal! decky)
d
> (print-deck decky)
Deck of cards: #(a b c j e f g h i d), num-left: 9
> (deal! decky)
e
```

```
> (print-deck decky)
Deck of cards: #(a b c j i f g h e d), num-left: 8
> (deal! decky)
c
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 7
> (deal! decky)
g
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 6
> (shuffle! decky)
#<deck>
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 10
> (deal! decky)
a
> (print-deck decky)
Deck of cards: #(d b h j i f g c e a), num-left: 9
```

We conclude this chapter by introducing two built-in functions. The first function, called `format`, works just like `printf`, except that instead of printing stuff out as a side effect, it generates a Scheme string as its output.

Example 19.1.3: The built-in `format` function

The following interactions demonstrate that `format` *and* `printf` *generate the same string of text. The only difference is that* `format` *returns that string as its output value, whereas* `printf` *prints it out to the Interactions Window as a side effect.*

```
> (format "~A + ~A = ~A~%" 2 3 (+ 2 3))
"2 + 3 = 5\n"
> (printf (format "~A + ~A = ~A~%" 2 3 (+ 2 3)))
2 + 3 = 5
> (printf "~A + ~A = ~A~%" 2 3 (+ 2 3))
2 + 3 = 5
> (format "~A~%  ~A~%    ~A~%" 1 2 3)
"1\n  2\n    3\n"
> (printf (format "~A~%  ~A~%    ~A~%" 1 2 3))
1
  2
    3
> (printf "~A~%  ~A~%    ~A~%" 1 2 3)
1
  2
    3
```

The next function is called `string-append`. It is analogous to the built-in `append` function, except that it appends strings instead of lists.

Example 19.1.4: The built-in `string-append` function

The contract for the `string-append` *function is given below, followed by some examples of its use.*

```
;;  STRING-APPEND  --  built-in function
;; ------------------------------------------------------
;;  INPUT:   Any number of strings
;;  OUTPUT:  A single string formed by concatenating all
;;           of the input strings.

> (string-append "String one!" "String two!")
"String one!String two!"
> (printf (string-append "String one!\n" "String two!\n"))
String one!
String two!
```

In later chapters, when we implement games, it will be useful to separate the tasks of (1) generating the output string as a Scheme datum, and (2) printing it out as a side effect. Although for a text-based interface it may suffice to print out the string in the Interactions Window, for a graphical interface, we may want to display the string in a graphical window, which `printf` cannot do.

19.2 Summary

Write the summary!

Special Forms Introduced in this Chapter

`define-struct`	For specifying new data structures

Built-in Functions Introduced in this Chapter

`format`	Like `printf`, but generates a string as output
`string-append`	For concatenating strings together