Part II

Destructive Programming in Scheme
Chapter 17

Iteration

Like recursion, iteration is a technique that enables a programmer to make the computer do things repetitively. However, unlike recursion, iteration typically involves destructive programming. This chapter presents several special forms that facilitate incorporating iteration in Scheme programs. The \texttt{set!} special form causes the value of a variable to be destructively modified. The \texttt{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 \texttt{true} is eventually going to evaluate to \texttt{false}.) The \texttt{dotimes} and \texttt{dolist} special forms, respectively, automate kinds of iteration that are analogous to numerical and list-based recursion.

17.1 The \texttt{set!} Special Form

The purpose of the \texttt{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 \texttt{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.

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{Example 17.1.1: The \texttt{set!} special form}
\hline
> (define x 100) \hspace{1cm} \textleftrightarrow \hspace{1cm} \text{Create a global variable \texttt{x} with value 100} \\
> x \hspace{1cm} \text{100} \\
> (set! x (* 5 5)) \hspace{1cm} \textleftrightarrow \hspace{1cm} \text{Change the value associated with \texttt{x}} \\
> x \hspace{1cm} \text{25} \\
> (set! y 50) \hspace{1cm} \textleftrightarrow \hspace{1cm} \text{No can do! There is no entry for a variable named \texttt{y}.} \\
Error!
\hline
\end{tabular}
\end{center}

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

The syntax of the \texttt{set!} special form. The \texttt{set!} special form has the following syntax:

\begin{verbatim}
(set! var newVal)
\end{verbatim}

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

\begin{verbatim}
(set! x 3)  
(set! myVar (* 8 10))  
(set! yourVar (member 3 '(1 2 3 4 5)))
\end{verbatim}
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).

Using `set!` within a local environment. Recall that when a symbol is evaluated within a local environment, the local environment takes precedence over any parent environments. For example, if a symbol `x` is being evaluated within a local environment, then drScheme looks first in that local environment for an entry for `x`. If it finds one, then it uses the associated value; otherwise, it checks the parent environment(s).

The same sort of precedence of local environments applies when using the `set!` special form within a local environment to change the value for a symbol. For example, suppose that `(set! var newVal)` is being evaluated with respect to a local environment `E_1` that is nested directly inside the Global Environment (i.e., `E_1 ⊂ E_0`). In this case, `newVal` will be evaluated with respect to the local environment `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 `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)     ;; Global variable, x
> (define y 'yyy)     ;; Global variable, y
> (let ((x 0))
    (set! x 'newX)     ;; Change value of LOCAL variable, x
    (set! y 'newY)     ;; Change value of GLOBAL variable, y
    (list x y))
(newX 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.

(Optional) Using `set!` within deeply nested environments. Suppose that `(set! var newVal)` is being evaluated with respect to an environment $E_n$ that is nested inside other environments, as follows: $E_n \subset E_{n-1} \subset \ldots \subset E_2 \subset E_1 \subset E_0$. The following steps are carried out:

1. The expression `newVal` is evaluated with respect to the environment $E_n$, generating some datum $D$.
2. The environments, $E_n, E_{n-1}, \ldots, E_0$, are scanned, in order, to find the first one that contains an entry for `var`. Call that environment $E_j$.
3. $D$ is inserted as the new value for `var` in the environment $E_j$.
4. The `set!` special form itself evaluates to `void`.

Example 17.1.3: (Optional) Using `set!` within 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, $E_3$, but ends up changing the value of `x` in the environment, $E_2$, because that is the ancestor environment that is closest to $E_4$ and contains a variable named `x`.

```
> (let ((x 10)) ;; Environment $E_1$
  (let ((x 20)) ;; Environment $E_2$
    (let ((x 30)) ;; Environment $E_3$
      (let ((w 40)) ;; Environment $E_4$
        (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 $E_4$ or $E_3$ to change the value of the variable named `x` in either $E_2$ or $E_1$, because those variables are effectively blocked by the presence of the variable named `x` in $E_3$. 
Although it is important to understand which variable is affected when a `set!` special form is evaluated 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 convenient 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 Condy holds, do some action Acty.” The semantics for this example could be summarized as follows:

1. Evaluate the condition `Condy`.
2. If `Condy` evaluates to `true` (or something that counts as true) then:
   1. Do the action `Acty` and go back to Step 1;
   2. Otherwise, stop.

For example, `Condy` might be the condition “the value of \( x \) is positive”, and `Acty` might be the compound action “first print out the value of \( x \); 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 condition `Condy` would always evaluate to the same thing. In particular, in the context of non-destructive programming, if `Condy` evaluated to `true` in Step 1, then it would forever evaluate to `true`, leading to a situation where the action `Acty` 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`.

* `Acty` 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 (a.k.a., Condy); and the expressions, expr1, ..., exprk, together constitute the body (a.k.a., the compound action Acty). Notice 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 true (or something that counts as true),
3. Then evaluate each of the expressions, expr1, ..., exprk, 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!")) ← Body evaluated zero times
> (void? (while #f (printf "hi!"))) ← The while expr. evaluates to void
#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)
    ;; BODY of the WHILE:
    (printf "counter = " 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 completes, counter has the value 4 (i.e., the value that caused the condition (< counter 4) to become false).

* If you forget to modify the value of a counter variable in a while loop, then the condition that controls the while loop may 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.

```scheme
> (let ((counter 0) (acc 0))
  (while (<= counter 100)
    ;; accumulate the current value of counter
    (set! acc (+ acc counter))
    ;; increment the value of 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.

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

In-Class Problem 17.2.2: An iterative version of transfer-all

Recall the accumulator-based, tail-recursive transfer-all function from Example 16.4.3. Define an iterative version of this function, called transfer-all-iter. Your function should not be recursive; instead, it should use let, while and set!, to iteratively transfer all of the elements of its first input onto its second input.
### In-Class Problem 17.2.3: Iterative version of the `insert` function

Recall the accumulator-based, tail-recursive `insert-acc` function from In-Class Problem 16.5.3. It began by recursively accumulating all of the numbers from the sorted list that were smaller than `item`, at which point it had effectively located the appropriate insertion point. Define an iterative function, called `insert-iter`, that uses `let` to create a local variable called `acc` that starts off empty. Next, it should use a `while` loop to iteratively accumulate all of the numbers from the sorted list that are smaller than `item`, at which point it will have located the appropriate insertion point for `item`. After the `while` loop, it can then use `transfer-all` (cf. Example 16.4.3) or `transfer-all-iter` (cf. In-Class Problem 17.2.2) to transfer all of the accumulated numbers onto the remains of the sorted list. Note that your `insert-iter` function should satisfy the same contract as the `insert-wr` function, not the `insert-acc` function. Instead of taking an extra `acc` input, it creates a local variable called `acc`.

### In-Class Problem 17.2.4: Iterative version of `insertion-sort`

Using the same approach as in In-Class Problem 17.2.3, define an iterative version of the insertion-sort algorithm. Call your function `isort-iter`. You may wish to review the `isort-acc` and `isort-wr` functions from In-Class Problem 16.5.4.

### 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.

```scheme
> (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: (Optional) Implementing our own version of \texttt{while}

The following \texttt{my-while} function provides essentially the same behavior as the \texttt{while} special form. However, it is a little clunkier to use because, unlike special forms, function-call expressions are evaluated by the Default Rule. Therefore, to ensure proper behavior, the condition and the body must be encapsulated within lambda functions, called \texttt{cond-func} and \texttt{body-func}, each of which takes zero inputs. The \texttt{my-while} function causes the desired condition to be evaluated by applying \texttt{cond-func} to zero inputs; and it causes the desired body expressions to be evaluated by applying \texttt{body-func} to zero inputs. Notice that, unlike almost all of the recursive functions seen in Part I of this book, the recursive call to \texttt{my-while} is applied to the same inputs! In so doing, the \texttt{my-while} function is implicitly relying on \texttt{cond-func} or \texttt{body-func} to be destructive, to avoid going into an endless loop.

\begin{verbatim}
;; 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))))
\end{verbatim}

The following interaction demonstrates the use of the \texttt{my-while} function.

\begin{verbatim}
> (let ((ctr 3))
  (printf "MY-WHILE example:\n")
  (my-while (lambda ()
               (> ctr 0))
            (lambda ()
               (printf " ctr: \"~A\" \" %ctr")
               (set! ctr (1- ctr)))))
MY-WHILE example:
ctr: 3
ctr: 2
ctr: 1
\end{verbatim}

Note that because \texttt{my-while} is a function, the expression, \texttt{(my-while...)}, is evaluated by the Default Rule. As a result, both lambda expressions are evaluated before calling the \texttt{my-while} function. Therefore, the corresponding lambda functions are created in the context of the local environment that includes an entry for the symbol \texttt{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 \texttt{ctr}. Thus, any occurrences of the symbol \texttt{ctr} in the bodies of those functions will refer to the local variable \texttt{ctr}, as desired.

Here is the equivalent example done using the \texttt{while} special form.

\begin{verbatim}
> (let ((ctr 3))
  (printf "WHILE example:\n")
\end{verbatim}
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!`.

<table>
<thead>
<tr>
<th>Example 17.3.1: Implementing <code>print-n-dashes</code> iteratively</th>
</tr>
</thead>
</table>

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

```scheme
;; 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.

```scheme
(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.

```scheme
> (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.

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.

```scheme
;; 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: (Optional) A more complex example of converting from recursion to iteration

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) #f)
      ;; NUM not found
      ;; Base Case 2: NUM found!
      ((= num (first listy)) 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-acc 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) #f)
          ;; 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
 (if (> (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:

\[
\text{dotimes (var numExpr)}
\text{expr}_1
\text{expr}_2
\ldots
\text{expr}_k
\]

where:

- \textit{var} is a symbol that will be the name of a counter variable in a local environment created by the dotimes;
- \textit{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 \texttt{dotimes}.

The semantics of the \texttt{dotimes} special form. A \texttt{dotimes} special form is evaluated as follows. First, the expression $\texttt{numExpr}$ is evaluated, resulting in a non-negative integer $n$. Second, a local environment $\mathcal{E}$ is created containing a variable $\texttt{var}$ 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 $\texttt{var}$ in the local environment is incremented by one.

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

Finally, the \texttt{dotimes} special form evaluates to \texttt{void}. Therefore, the usefulness of \texttt{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.

\begin{example}[Example 17.4.1: Illustrating the \texttt{dotimes} special form]

The following examples illustrate how the \texttt{dotimes} special form can be used.

\begin{verbatim}
> (dotimes (i 5)
   ;; The body:
   (printf "i: " ~A"\n" i))
i: 0
i: 1
i: 2
i: 3
i: 4

> (dotimes (i (+ 1 2))
   ;; The body:
   (printf "\nA + \nA = " ~A"\n" i i (+ i i)))
0 + 0 = 0
1 + 1 = 2
2 + 2 = 4
\end{verbatim}

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 \texttt{dotimes} is evaluated three times, since $(+ 1 2)$ evaluates to 3.
\end{example}

\begin{example}[Example 17.4.2: (Optional) Implementing our own version of \texttt{dotimes}]

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

\begin{verbatim}
;; --- MY-DOTIMES
;; -------------------------------------------------------------
;; INPUTS: N, a non-negative integer
;; BODY-FUNC, a function that takes zero inputs
\end{verbatim}

\end{example}
\[;\; \text{OUTPUT: void} ;\; \text{SIDE EFFECT: Evaluates the expression (BODY-FUNC I)} ;\; \text{N times, once for each value of I in } 0, 1, 2, \ldots, N-1\]

\[
\text{(define my-dotimes}
\quad \text{(lambda (n body-func)}
\quad \quad \text{(let (i 0)}
\quad \quad \quad \quad \text{(while (< i n)}
\quad \quad \quad \quad \quad \quad \text{(body-func i)}
\quad \quad \quad \quad \quad \quad \quad \text{(set! i (1+ i)))}}
\quad \text{))))
\]

Here is a simple demonstration of its behavior:

\[
\text{> (my-dotimes 4 (lambda (i)}
\text{\quad (printf "--- i = \"A\" i))})}
\]
\[
\text{--- i = 0}
\text{--- i = 1}
\text{--- i = 2}
\text{--- i = 3}
\]

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:

\[
\text{(dolist (elt listExpr) expr1 expr2 \ldots exprk)}
\]

where **elt** is any symbol expression, and **listExpr** is any Scheme expression that evaluates to a list. The expressions, \(\text{expr}_1, \ldots, \text{expr}_k\), 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 \(E'\) is created that contains a variable called **elt**. Then, for each element \(e\) of the list \(L\), the following steps are taken: (1) the value of **elt** is set to \(e\); and (2) the expressions, \(\text{expr}_1, \ldots, \text{expr}_k\), are evaluated—in order—with respect to the local environment \(E'\). (Typically, the expressions in the body contain references to the symbol **elt**, which allows them to do computations involving the current element \(e\).) Thus, if there are \(n\) elements in the list \(L\), then the expressions in the body of the **dolist** 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.

\[
\text{> (dolist (elt '(a b c))}
\quad \text{(printf "elt: \"A\" elt))}
\]
\[
\text{elt: a}
\text{elt: b}
\]

\[
\text{Example 17.5.1}
\]

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

\[
\text{> (dolist (elt '(a b c))}
\quad \text{(printf "elt: \"A\" elt))}
\]
\[
\text{elt: a}
\text{elt: b}
\]

\[
\text{Example 17.5.1}
\]

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

\[
\text{> (dolist (elt '(a b c))}
\quad \text{(printf "elt: \"A\" elt))}
\]
\[
\text{elt: a}
\text{elt: b}
\]
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: (Optional) 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)
    ;; As long as LISTY is non-empty...
    (while (not (null? listy))
      ;; Evaluate the "body" with ELT = (FIRST LISTY)
      (body-func (first listy))
      ;; The next iteration will deal with (REST 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

This chapter introduced a kind of repetitive processing called iteration, that is an alternative to recursion. Iterative programming typically requires destructive programming. In particular, it frequently requires the programmer to create and modify various variables to control the iterative process. This can make tracking down errors in iterative programming more difficult. However, for many tasks, iterative solutions can be elegant and efficient.

The set! special form allows a programmer to change the value of an existing variable to any desired value. Frequently, set! is used to increment or decrement the value of a counter variable in looping constructs.

The while special form provides a basic form of iteration where the programmer must explicitly specify the condition that determines how many iterations are performed, and explicitly modify the values of variables upon which the condition typically depends. The let, while and set! special forms are frequently used together. For example, a let special form can be used to create some local variables (e.g., a counter variable and one or more accumulators), a while special form can then be placed within the body of the let to provide the desired repetitive behavior, and the set! special form can be used within the body of that while to control its iterative behavior. With such low-level control, the programmer can specify a wide variety of iterative computational behaviors that can be extremely useful; however, care must be taken to ensure that, for example, counter variables are correctly updated to avoid entering into an infinite loop.

Next, this chapter demonstrated how tail-recursive functions can be transformed into equivalent iterative functions. The equivalence of tail-recursion and iteration is what DrScheme relies on when it recycles a single function call box for all of the recursive function calls of a tail-recursive function.

Because certain iterative computations are so frequently needed, the dotimes and dolist special forms are provided. They shield the programmer from the low-level details of, for example, setting up a local environment for a counter variable, and managing the changing of its value throughout the iterative process. The dotimes special form enables a programmer to make some set of computations happen a specified number of times, once for each value of a counter variable, for example, \(i \in \{0, 1, 2, \ldots, n - 1\}\). In each iteration, the expressions in the body of the dotimes may refer to the current value of \(i\). Similarly, the dolist special form enables a programmer to make some set of computations happen, once for each element of a given list.

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