Chapter 9

State

Faced with the choice between changing one's mind and proving that there is no need to do so, almost everyone gets busy on the proof.

John Kenneth Galbraith

The subset of Scheme we have used until this chapter provides no means to change the value associated with a name. This enabled very simple evaluation rules for names, as well as allowing the substitution model of evaluation. Since the value associated with a name was always the value it was defined as, no complex evaluation rules are needed to determine what the value associated with a name is. In this chapter, we will introduce special forms known as *mutators* that allow programs to change the value associated with a given name. Introducing mutation does not change the computations we can express—every computation that can be expressed using mutation could also be expressed functionally. It does, however, make it possible to express certain computations more efficiently, elegantly, and clearly than could be done without it.

9.1 Mutation

The first mutator we describe is the `set!` special form expression. The exclamation point (`!`) at the end of the name (pronounced "bang!") is a naming convention used to indicate that a special form or procedure may mutate state. The `set!` expression replaces the value associated with a name. A set expression is also known as an *assignment*. It assigns a new value to a variable.

The grammar rule for the set expression is:

Expression	::⇒	*SetExpression*
SetExpression	::⇒	`(set!` *Name Expression*`)`

The evaluation rule is:

Evaluation Rule 7: Set. To evaluate a set expression, evaluate the expression, and replace the value associated with the name with the value of the expression. A set expression has no value.

For example:

```
> (define num 200)
> num
200
> (set! num 150)
> num
150
```

The set expression changes the value associated with the name `num` from the originally defined value to the value of the set expression. Note that the set expression does not produce an output value. Set expressions are used for their *side-effects*. They change the value of some state (namely, the value associated with the name in the set expression), but do not produce an output.

9.1.1 Begin Expression

Since set expressions do not have a value, they are often used inside a begin expression. The grammar rule for begin is:

Expression	::⇒	*BeginExpression*
BeginExpression	::⇒	`(begin` *MoreExpressions Expression*`)`

The evaluation rule for begin is:

> **Evaluation Rule 8: Begin.** To evaluate a begin expression, `(begin` *`Expression`*$_1$ *`Expression`*$_2$ `...` *`Expression`*$_k$`)`, evaluate each subexpression in order from left to right. The value of the begin expression is the value of the last subexpression, *`Expression`*$_k$.

The begin expression is useful when we are evaluating expressions that have side-effects. This means the expression is important not for the value it produces (since the begin expression ignores the values of all expressions except the last one), but for some change to the state of the machine it causes.

The special define syntax for procedures includes a hidden begin expression. The syntax,

```
(define (Name Parameters)
   MoreExpressions Expression)
```

is an abbreviation for:

```
(define Name
   (lambda (Parameters)
       (begin MoreExpressions Expression)))
```

The let expression also includes a hidden begin expression. The let expression

```
(let ((Name1 Expression1)
      (Name2 Expression2)
      ...
      (Namek Expressionk))
   MoreExpressions Expression)
```

is equivalent to the application expression:

```
((lambda (Name₁ Name₂ ... Nameₖ)
    (begin MoreExpressions Expression))
 Expression₁
 Expression₂
 ...
 Expressionₖ)
```

9.1.2 Impact of Set Expressions

Introducing the set expression presents many complications for our programming model. In particular, it invalidates the substitution model of evaluation we introduced in Section 3.6.2. Previously, all the procedures we could define were *functional*—every time they are applied to the same inputs, they produce the same output. Set expressions allow us to define non-functional procedures that produce different results for different applications even with the same inputs.

For example, consider the `update-counter!` procedure:

```
(define counter 0)

(define (update-counter!)
  (set! counter (+ counter 1))
  counter)
```

Every time we evaluate `(update-counter!)` it increments the value associated with the name `counter`, and produces as output the resulting value of `counter`. Hence, the value of `(update-counter!)` is 1 the first time it is evaluated, `2` the next time, and so forth.

Consider evaluating the expression

```
(+ counter (update-counter!))
```

Recall that the evaluation rule for the application expression does not specify in which order the subexpressions are evaluated. But, the value of the name expression `counter` depends on whether it is evaluated before or after the application of `update-counter!` is evaluated! This means, there is no clear value for the expression: if the second subexpression, `counter`, is evaluated before the third subexpression, `(update-counter!)`, the value of the expression is 1 the first time it is evaluated, and 3 the second time it is evaluated. Alternately, but consistently with the evaluation rules, the third subexpression could be evaluated before the second subexpression. With this ordering, the value of the expression is 2 the first time it is evaluated, and 4 the second time it is evaluated.

9.1.3 Names, Places, Frames, and Environments

Because of the power set expressions provide to change the value associated with a name, the order in which expressions are evaluated now matters. As a result of adding mutation, we will need to revisit several of our other evaluation rules and change the way we think about processes.

Previously, a *Name* was an identifier that has an associated value. Since the value associated with a name can now change, we instead think of a name as a way to identify a *place*. A place holds a value. A *frame* is a collection of places.

An *environment* is a pair consisting of a frame and a pointer to a parent environment. A special environment known as the *global environment* has no parent environment. The global environment exists when the interpreter starts, and is maintained for the lifetime of the interpreter. Other environments may be created and destroyed as a program is evaluated. All environments besides the global environment have a parent which is another environment. The ultimate parent environment of any environment (other than the global environment which has no parent) is the global environment. This means that if we start with any environment, and continue to follow their parent pointers until there is no parent, we will always end in the global environment.

The key change to our evaluation model is that whereas before we could think about evaluating expressions without any notion of *where* they are evaluated, once we introduce mutation, we need to consider the environment in which an expression is evaluated. An environment captures the current state of the interpreter. The value of an expression depends on both the expression itself, and on the environment in which it is evaluated.

Figure 9.1 illustrates some environments. The global environment contains a

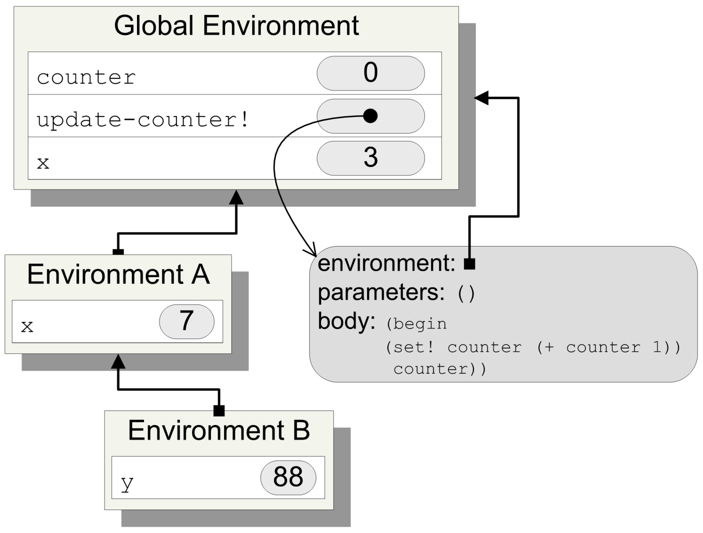

Figure 9.1: Sample environments.

frame that has three names. Each name has an associated place that contains the value associated with that name. The value associated with `counter` is the value in the place next to `counter`, currently 0. The value associated with `set-counter!` is the procedure we defined in the previous subsection. A procedure is characterized by the parameters (in this case, the parameter list is empty), and the body of the procedure (the begin expression shown in the figure). In addition, a procedure has an environment pointer. This points to the environment in which the procedure body is to be evaluated. (We'll cover the new evaluation rule for procedures later.)

9.1.4 Evaluation Rules with State

Introducing mutation requires us to revise the evaluation rule for names, the definition rule, and the application rule for constructed procedures. All of these rules are adapted to be more precise about how values are associated with names by using environments.

The evaluation rule for a name expression becomes:

> **Evaluation Rule 2 (with state): Names.** A name expression evaluates to the value associated with the name. To find the value associated with a name, look for the name in the frame associated with the evaluation environment. If it contains a place with the name, the value of the name expression is the value in that place. If it does not contain a place with the name, the value of the name expression is the value of the name expression evaluated in the parent environment if the current evaluation environment has a parent. Otherwise, the name expression evaluates to an error (the name is not defined).

For example, to evaluate the value of the name expression `x` in Environment B in Figure 9.1, we would first look in the frame in Environment B for a place named `x`. Since there is no place named `x` in that frame, we follow the parent environment pointer for Environment B to Environment A, and evaluate the value of the name expression `x` in Environment A. The frame in Environment A contains a place named `x`, that contains the value 7. Hence, the value of evaluating `x` in Environment B is 7. If we instead evaluated the same expression in the Global Environment, the value would be 3, since that is the value in the place named `x` in the frame of the Global Environment.

To evaluate the value of the name expression `y` in Environment A, we would first look in the frame in Environment A for a place named `y`. Since none is found, we continue by evaluating the expression in the parent environment, which is the global environment. The global environments frame does not contain a place named `y`, and the global environment has no parent, so the expression has no value. It is an error, since there is no place named `y` visible from Environment A.

We also need a new evaluation rule for definitions:

> **Definition Rule (with state).** A definition creates a new place named after the definition name in the frame associated with the evaluation environment. The value in the place is value of the expression. If there is already a place with the name in the current frame, the definition replaces the old place with the new place and value.

The part of the rule that deals with redefinitions means we could use `define` in some situations to mean something similar to `set!`. It does not mean the same thing, however, since a set expression finds the place associated with the name and puts a new value in that place. A set expression evaluation follows the Evaluation Rule 2 (with state) above to find the place associated with a name. Hence, `(define` *`Name Expr`*`)` has different meaning from `(set!` *`Name Expr`*`)` when there is no place named *Name* in the current execution environment. To avoid this confusion, we encourage you to only use `define` for the first definition of a name in the global environment, and to always use `set!` when the intent is to change the value associated with a name.

The final evaluation rule that must change as a result of mutation is the evaluation rule for applications of constructed procedures. The substitution model fails us now, since the value of an expression depends on the environment in which it is evaluated. The new application rule replaces substitution with creating a new environment containing places named for the parameters containing the values of the corresponding operand expressions:

> **Application Rule 2 (with state): Constructed Procedures.** To apply a constructed procedure:
>
> 1. Construct a new environment, whose parent is the environment to which the environment pointer of the applied procedure points.
> 2. Create a place in the frame of the new environment for each parameter containing the value of the corresponding operand expression.
> 3. Evaluate the body of the procedure in the newly created environment. The resulting value is the value of the application.

For example, consider evaluating the application expression `(max 3 4)` where `max` is the procedure defined in Example 3.7:

```
(define (max a b) (if (> a b) a b)))
```

To evaluate an application of `max`, we follow Application Rule 2 above. First, we create a new environment. Since `max` was defined in the global environment, its environment pointer points to the global environment. Hence, the parent environment for the new environment is the global environment. Then, we create places in the new environment's frame named `a` and `b`. The value in the place associated with `a` is 3, the value of the first operand expression. The value in the place associated with `b` is 4. Then, we follow step 3, evaluating the body expression, `(if (> a b) a b)`, in the newly created environment. Figure 9.2 shows the environment where the body expression is evaluated. When we evaluate the name expression `a` in this environment, the place associated with `a` is found in the execution environment and it contains the value 3.

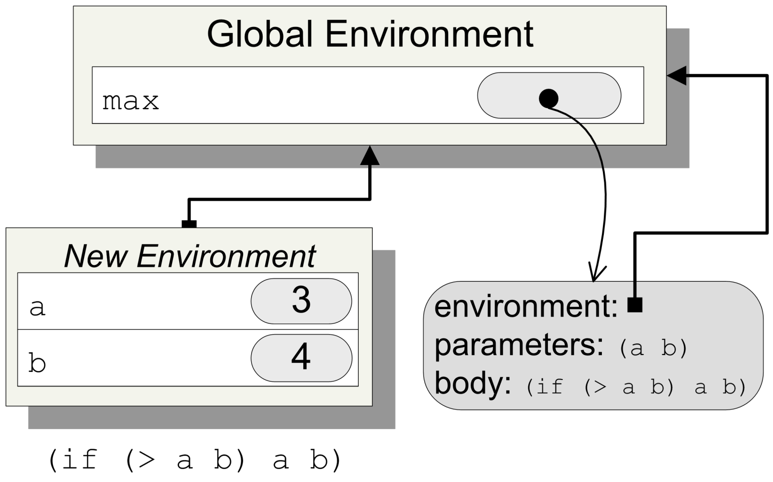

Figure 9.2: Environment created to evaluate `(max 3 4)`.

The impact of the new application rule is more pronounced when we consider a procedure that creates a new procedure. For example, the `make-incrementer` procedure takes the increment number as input and produces as output a procedure. The output procedure takes a number as input, and ouputs that number plus the increment number.

```
(define (make-incrementer inc)
  (lambda (n) (+ n inc)))
```

Suppose we evaluate the definition `(define inc (make-incrementer 1))`. The resulting environment is showing in Figure 9.3. The name `inc` has a value that is the procedure resulting from the application of `(make-incrementer 1)`. To evaluate the application, we follow the application rule above and create a new environment containing a frame with the parameter names (in this case, just `inc`) and their associated operand values (in this case, 1). The result of the application is the value of evaluating its body in this new environment. Since the body is a lambda expression, it evaluates to a procedure. That procedure was created in the execution environment that was created to evaluate the application of `make-incrementer`, hence, its environment pointer points to the created new environment (which contains a place named `inc` holding the parameter value, 1) instead of pointing to the global environment.

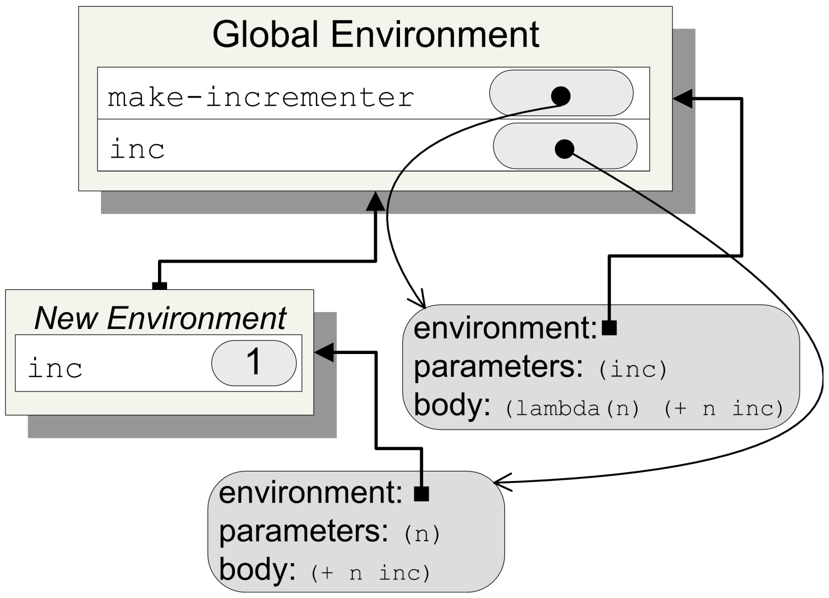

Figure 9.3: Environment after `(define inc (make-incrementer 1))`.

To evaluate an application of the `inc` procedure, now defined in the global environment, we follow the evaluation rules. Evaluating `(inc 149)` involves using the application rule. It creates a new environment with a frame containing the place `n` and its associated value 149. Inside that environment, we evaluate the body of the procedure, `(+ n inc)`. The value for `n` is found in the execution environment. The value for `inc` is not found their, so evaluation continues by looking in the parent environment. This contains a place `inc` containing the value 1. The `inc` name in the global environment is not reached. Figure 9.1.4 illustrates the environment for evaluating the body of the `inc` procedure.

Exercise 9.1.($\star$) Devise a Scheme expression that could have four possible values, depending on the order in which application subexpressions are evaluated. $\diamondsuit$

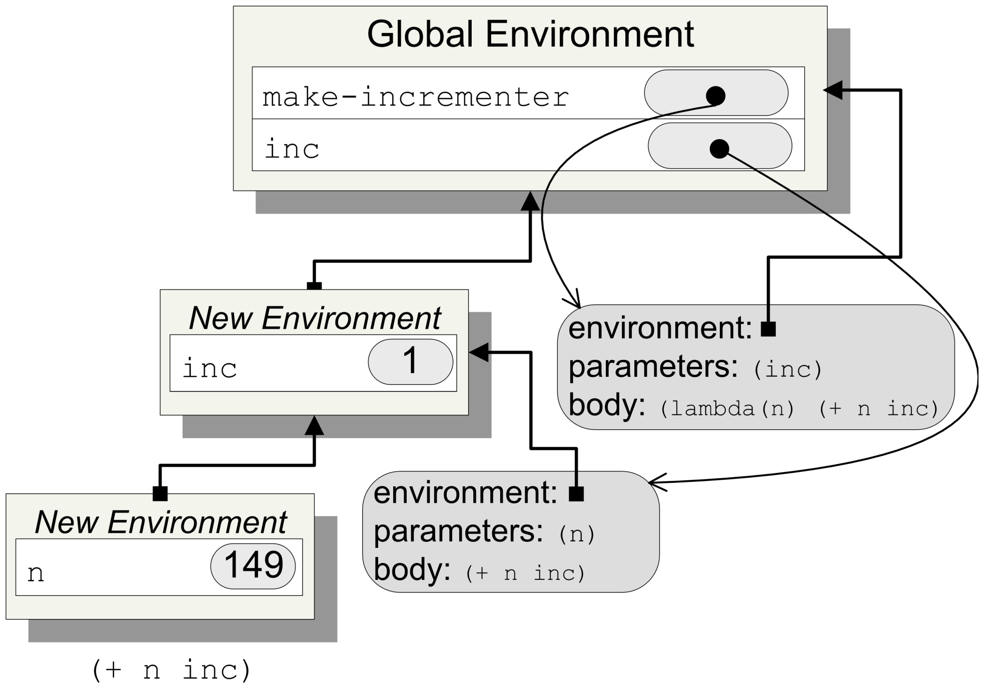

Figure 9.4: Environment for evaluating the body of `(inc 149)`.

Exercise 9.2. Draw the environment after evaluating:

```
> (define alpha 0)
> (define beta 1)
> (define update-beta!
      (lambda () (set! beta (+ alpha 1)))
> (set! alpha 3)
> (update-beta!)
> (set! alpha 4)
```

♢

Exercise 9.3. Draw the environment after evaluating the following expressions, and explain what the value of the final expression is. (You may want to rewrite the `let` as an application.)

```
> (define (make-applier proc) (lambda (x) (proc x))
> (define p (make-applier (lambda (x) (let ((x 2)) x))))
> (p 4)
```

♢

9.1.5 Pair Mutators

The `set-car!` and `set-cdr!` procedures change the values in the components of a pair:

- `(set-car!` *Expression*$_{pair}$ *Expression*$_{value}$ — replaces the value in the first component of the pair to which the *Expression*$_{pair}$ evaluates with the value of *Expression*$_{value}$. If the *Expression*$_{pair}$ does not evaluate to a pair, the expression is an error.
- `(set-cdr!` *Expression*$_{pair}$ *Expression*$_{value}$ — replaces the value in the second component of the pair to which the *Expression*$_{pair}$ evaluates

with the value of *Expression*$_{value}$. If the *Expression*$_{pair}$ does not evaluate to a pair, the expression is an error.

For example:

```
> (define pair (cons 1 2))
> (set-car!  pair 3)
> pair
(3 . 2)
> (set-cdr!  pair 4)
> pair
(3 . 4)
```

The `set-cdr!` procedure allows us to create a pair where the second part of the pair is itself!

```
> (set-cdr!  pair pair)
```

This creates the rather frightening object shown in Figure 9.1.5. Every time we apply `cdr` to `pair`, we get the same pair as the output. Hence, the value of

```
(car (cdr (cdr (cdr (cdr (cdr pair))))))
```

is 3.

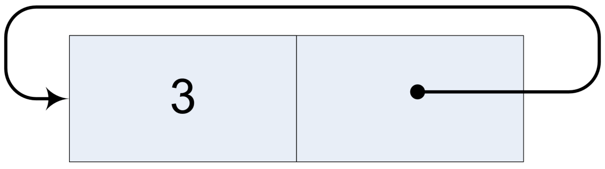

Figure 9.5: Pair created by evaluating `set-cdr!  pair pair`.

Exercise 9.4. Using the `length` procedure defined in Section 5.3.1, what does `(length pair)` evaluate to? (Where `pair` it the pair shown in Figure 9.1.5.) ♢

Exercise 9.5.(⋆⋆) Define a `list?` procedure that behaves correct on inputs like `pair`. Circular structures like this are not lists. ♢

9.2 Imperative Programming

Mutation enables a style of programming known as *imperative programming*. Whereas *functional programming*, on which we have focused until now, is concerned with defining procedures that can be composed to solve a problem, imperative programming is primarily concerned with modifying state in ways that lead to the desired state which provides a solution to a problem. The main operation in function programming is application. A functional program applies a series of procedures, passing the outputs of one application as the inputs of the next procedure application. With imperative programming, the primary operation is assignment (embodied by `set!` in Scheme, but typically by an assignment operator, := or =, in languages designed for imperative programming such as Pascal, Algol60, Java, and C++).

The next subsection presents imperative-style versions of some of the procedures we have seen in previous chapters for manipulating lists. The following subsection describes some imperative control structures.

9.2.1 List Mutators

All the procedures for changing the value of a list in Section 5.3.3 involved constructing new lists. If our goal is only to change some elements in an existing list, this is inefficient. It wastes memory to construct a new list, and may require more running time than a procedure that modifies the input list instead. Here, we revisit some of the procedures from Section 5.3.3, but this time instead of producing new lists with the desired property, we modify the input list.

Example 9.1: Mapping. The `map` procedure produces a new list that is the result of applying the same procedure to every element in the input list. Using mutation, we can define a `map!` procedure that modifies the elements in the input list instead.

```
(define (map! f p)
  (if (null? p) (void)
      (begin
        (set-car! p (f (car p)))
        (map! f (cdr p)))))
```

The base case uses `(void)` to evaluate to no value. The `void` procedure takes no inputs and outputs no value. Unlike `map` which evaluates to a list, there is no output value for a `map!` application. The purpose of the application is to mutate the input list.

The running time of the `map!` procedure is in $\Theta(n)$ where n is the number of elements in the input list, assuming the procedure passed to `map` has constant running time. There will be n recursive applications of `map!` since each one passes in a list one element shorter than the input list, and each application requires constant time. This is asymptotically identical to the `map` procedure, but we would expect the actual running time to be faster since there is no need to construct a new list.[1]

Compare the definition of `map!` to the earlier definition of `map`:

```
(define (map f p)
  (if (null? p) null
      (cons (f (car p)) (map f (cdr p)))))
```

Whereas the functional `map` procedure needs to use `cons` to build up the output list, the imperative `map!` procedure uses `set-car!` to mutate the elements in the list.

The example interaction below illustrates the different between `map` and `map!`: `map` is functional—it produces a value but does not modify its input list; `map!` is imperative—it modifies the input list but produces no value.

```
> (define a (list 1 2 3))
> (map square a)
(1 4 9)
> a
(1 2 3)
> (map!  square a)
> a
(1 4 9)
```

[1]The memory consumed is asymptotically different. The `map` procedure must maintain n stack frames because of the `cons`, and allocates n new `cons` cells, so it requires memory in $\Theta(n)$. The `map!` procedure is tail recursive (so no stack needs to be maintained), and does not allocate any new `cons` cells, so it requires constant memory.

Example 9.2: Filtering. The `filter` procedure takes as inputs a test procedure as a parameter, and produces as output a list containing the elements of the input list to which applying the test procedure evaluates to a true value. In Example 5.3.3, we defined `filter` as:

```
(define (filter test p)
   (if (null? p) null
       (if (test (car p))
           (cons (car p) (filter test (cdr p)))
           (filter test (cdr p)))))
```

An imperative version of `filter` would remove the unsatisfying elements from the input list, instead of creating a new list. We define `filter!` by using `set-cdr!` to skip over elements that should not be included in the filtered list.

```
(define (filter! test p)
  (if (null? p) null
      (begin
        (set-cdr! p (filter! test (cdr p)))
        (if (test (car p))
            p
            (cdr p)))))
```

Assuming the test procedure has constant running time, the running time of the `filter!` procedure is in $\Theta(n)$ where n is the length of the input list. This is comparable to the `filter` procedure.

Unlike `map!` which produces no output value, `filter!` does produce an output value. This is necessary to produce the correct behavior when the first element is not in the list. Consider this example:

```
> (define a (list 1 2 3 1 4))
> (filter! (lambda (x) (> x 1)) a)
(2 3 4)
> a
(1 2 3 4)
```

The value of a after the filter! application still includes the initial 1, although the fourth element 1 is removed. There is no way for a procedure to remove the first element of the list: the set-car! and set-cdr! procedures only enable us to change what the pair's components contain. Hence, the way filter! should be used is with set!:

```
(set! a (filter! (lambda (x) (> x 1)) a))
```

Example 9.3: Append. The append procedure takes as input two lists and produces as output a list consisting of the elements of the first list followed by the elements of the second list. An imperative version of this procedure instead mutates the first list so after the application it now contains the elements of both lists.

```
(define (append! p q)
  (if (null? p)
      (error "Cannot append to an empty list")
      (if (null? (cdr p))
          (set-cdr! p q)
          (append! (cdr p) q))))
```

Our definition disallows appending to null—this is necessary since if the input is null there is no pair to modify. The append! library procedure in MzScheme takes a different approach. It is defined when the first list is null, and produces the value of the second list as output in this case. This has unexpected behavior when an expression like (append! a b) is evaluated where the value of a is null since a is not modified to be the appended list.

As with append, the running time of the append! procedure is in $\Theta(n)$ where n is the number of elements in the first input list.

Exercise 9.6.($\star\star$) Define an imperative-style procedure, reverse!, that reverses the elements of a list. Is it possible to implement a reverse! procedure that is asymptotically faster than the reverse procedure from Section 5.3.3? $\diamondsuit$

9.2.2 Imperative Control Structures

The imperative style of programming makes progress by using assignments to manipulate state. In many cases, solving a problem requires repeated operations. With functional programming, this is done using recursive definitions. We make progress towards a base case by passing in different values for the operands with each recursive application. With imperative programming, we can make progress by changing state repeatedly without needing to passing in different operands.

A common control structure in imperative programming is a *while loop*. A while loop has a test condition and a body. The test condition is a predicate. If it evaluates to true, the while loop body is executed. Then, the test condition is evaluated again. The while loop continues to execute until the test condition evaluates to false.

We can define `while` as a procedure that takes as input two procedures, a test procedure and a body procedure, each of which take no parameters. Even though the test and body procedures take no parameters, they need to be procedures instead of expressions, since every iteration of the loop should re-evaluate the test and body expressions of the passed procedures.

```
(define (while test body)
  (if (test)
      (begin
        (body)
        (while test body))
      (void))) ; no value
```

We can use the `while` procedure to implement Fibonacci similarly to the `fast-fibo` procedure:

```
(define (while-fibo n)
  (let ((a 1)
        (b 1)
        (left (- n 3)))
    (while (lambda () (>= left 0))
           (lambda ()
             (set! b (+ a b))
```

```
                 (set! a (- b a))
                 (set! left (- left 1))))
        b))
```

The value of `b` is the final result of the `while-fibo` procedure. Each iteration through the while loop the body procedure is applied, updating the values of `a` and `b` to the next Fibonacci numbers. Note the assignment to `a`: the assigned value is computed as `(- b a)` instead of `b`. The reason for this is the previous `set!` expression has already changed the value of `b`, by adding `a` to it. Since the next value of `a` should be the old value of `b`, we can find the necessary value by subtracting `a`. An alternative approach, which would save the need to do subtraction, is to store the old value in a temporary value. We could use this as the body procedure instead:

```
(lambda ()
    (let ((oldb b))
       (set! b (+ a b))
       (set! a oldb)
       (set! left (- left 1))))
```

Programming languages developed for imperative programming provide control constructs similar to the `while` procedure defined above. For example, here is what the `while-fibo` procedure would look like in the Java programming language:[2]

```
static public int fibonacci (int n) {
   int a = 1;
   int b = 1;
   int left = n - 3;

   while (left >= 0) {
      int oldb = b;
      b = a + b;
      a = oldb;
```

[2]Don't worry if not everything in this code makes sense. The point of the example is to give you a sense what other programming languages look like, and how similar control structures are across different languages.

```
          left = left - 1;
       }

       return b;
    }
```

Here is what it would look like in Python:[3]

```
    def fibonacci (n):
       a = 1
       b = 1
       left = n - 3
       while left >= 0:
           oldb = b
           b = a + b
           a = oldb
           left = left - 1
       return b
```

Exercise 9.7. Define the `map!` example from the previous section using `while`. ◊

Exercise 9.8. Another popular imperative programming structure is a *repeat-until* loop. Define a `repeat-until` procedure that takes two inputs, a body procedure and a test procedure. The procedure should evaluate the body procedure repeatedly, until the test procedure evaluates to a true value. For example, using `repeat-until` we could define `factorial` as:

```
    (define (factorial n)
      (let ((fact 1))
        (repeat-until
         (lambda ()
           (set! fact (* fact n))
           (set! n (- n 1)))
```

[3]As with the Java example, don't worry if not everything in this code makes sense. We will be introducing Python soon.

```
          (lambda () (< n 1)))
         fact))
```

♢

9.3 Summary

Adding the ability to change the value associated with a name complicates our evaluation rules, but also enables simpler and more efficient solutions to many problems. Once we add assignment to our language, the order in which things happen affects the value of some expressions. Instead of evaluating expressions, we now need to always evaluate an expression in a particular execution environment. Mutation allows us to manipulate larger data structures efficiently since it is not necessary to copy the data structure to make any changes to it. We have already seen with the list mutation procedures that some changes can be done more efficiently using mutation. We can also use mutation to maintain complex data structures for representing tables of data such as a relational database.