9 Mutation

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 so far provides no way to change the value associated with a name. This enables 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 the value associated with a name.

This chapter introduces special forms known as *mutators* that allow programs to change the value in a given place. Introducing mutation does not change the computations we can express—every computation that can be expressed using mutation could also be expressed using the only purely functional subset of Scheme from Chapter 3. It does, however, make it possible to express certain computations more efficiently and clearly than could be done without it. Adding mutation is not free, however; reasoning about the value of expressions becomes much more complex.

mutators

9.1 Assignment

The **set!** (pronounced "set-bang!") special form associates a new value with an already defined name. The exclamation point at the end of **set!** follows a naming convention to indicate that an operation may mutate state. A set expression is also known as an *assignment*. It *assigns* a value to a variable.

assignment

The grammar rule for assignment is:

Expression ::⇒ *Assignment*
Assignment ::⇒ (**set!** *Name Expression*)

The evaluation rule for an assignment is:

> **Evaluation Rule 7: Assignment.** To evaluate an assignment, evaluate the expression, and replace the value associated with the name with the value of the expression. An assignment has no value.

Assignments do not produce output values, but 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.

Here is an example use of **set!**:

```
> (define num 200)
> num
200
> (set! num 150)
> (set! num 1120)
> num
1120
```

Begin expression. Since assignments do not evaluate to a value, they are often used inside a begin expression. A begin expression is a special form that evaluates a sequence of expressions in order and evaluates to the value of the last expression.

The grammar rule for the begin expression is:

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

The evaluation rule 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 values of all the subexpressions except the last one are ignored; these subexpressions are only evaluated for their side effects.

The begin expression must be a special form. It is not possible to define a procedure that behaves identically to a begin expression since the application rule does not specify the order in which the operand subexpressions are evaluated.

The definition syntax for procedures includes a hidden begin expression.

(**define** (*Name Parameters*) *MoreExpressions Expression*)

is an abbreviation for:

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

The let expression introduced in Section 8.1.1 also includes a hidden begin expression.

(**let** ((*Name*$_1$ *Expression*$_1$) (*Name*$_2$ *Expression*$_2$)
$\cdots$ (*Name*$_k$ *Expression*$_k$))
MoreExpressions Expression)

is equivalent to the application expression:

((**lambda** (*Name*$_1$ *Name*$_2$ ... *Name*$_k$)
(**begin** *MoreExpressions Expression*))
Expression$_1$ *Expression*$_2$ ... *Expression*$_k$)

9.2 Impact of Mutation

Introducing assignment presents many complications for our programming model. It invalidates the substitution model of evaluation introduced in Section 3.6.2 and found satisfactory until this point. All the procedures we can define without using mutation behave almost like mathematical functions—every time they are applied to the same inputs they produce the same output.[1] Assignments allow us to define non-functional procedures that produce different results for different applications even with the same inputs.

Example 9.1: Counter. Consider the *update-counter!* procedure:

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

To use *update-counter!*, we must first define the *counter* variable it uses:

(**define** *counter* 0)

Every time (*update-counter!*) is evaluated the value associated with the name *counter* is increased by one and the result is the new value of counter. Because of the hidden begin expression in the definition, the (**set!** *counter* (+ *counter* 1)) is always evaluated first, followed by *counter* which is the last expression in the begin expression so its value is the value of the procedure. Thus, the value of (*update-counter!*) is 1 the first time it is evaluated, 2 the second time, and so on.

The substitution model of evaluation doesn't make any sense for this evaluation: the value of *counter* changes during the course of the evaluation. Even though (*update-counter!*) is the same expression, every time it is evaluated it evaluates to a different value.

[1] Observant readers should notice that we have already used a few procedures that are not functions including the printing procedures from Section 4.5.1, and *random* and *read-char* from the previous chapter.

Mutation also means some expressions have undetermined values. Consider evaluating the expression (+ *counter* (*update-counter!*)). The evaluation rule for the application expression does not specify the order in which the operand 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!

The meaning of the expression is ambiguous: 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 still following 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.

9.2.1 Names, Places, Frames, and Environments

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

place

Since the value associated with a name can now change, instead of associating a value directly with a name we use a name as a way to identify a *place*. A place has a name and holds the value associated with that name. With mutation, we can change the value in a place; this changes the value associated with the place's name. A *frame* is a collection of places.

frame

environment

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. Initially, the global environment contains the built-in procedures. Names defined in the interactions buffer are placed in the global environment. Other environments are created and destroyed as a program is evaluated. Figure 9.1 shows some example environments, frames, and places.

Every environment has a parent environment except for the global environment. All other environments descend from the global environment. Hence, if we start with any environment, and continue to follow its parent pointers we always eventually reach the global environment.

The key change to our evaluation model is that whereas before we could evaluate 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.

9.2.2 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 must be adapted to be more precise about how values are associated with names by using places and environments.

Names. The new evaluation rule for a name expression is:

> **Stateful Evaluation Rule 2: Names.** To evaluate a name expression, search the evaluation environment's frame for a place with a name that matches the name in the expression. If such a place exists, the value of the name expression is the value in that place. Otherwise, the value of the name expression is the result of evaluating the name expression in the parent environment. If the evaluation environment has no parent, the name is not defined and the name expression evaluates to an error.

For example, to evaluate the value of the name expression x in Environment B in Figure 9.1, we first look in the frame of Environment B for a place named x. Since there is no place named x in that frame, we follow the parent pointer to Environment A, and evaluate the value of the name expression in Environment A. Environment A's frame contains a place named x that contains the value 7, so the value of evaluating x in Environment B is 7.

The value of the same expression in the Global Environment is 3 since that is the value in the place named x in the Global Environment's frame.

To evaluate the value of y in Environment A, we first look in the frame in Environment A for a place named y. Since no y place exists, evaluation con-

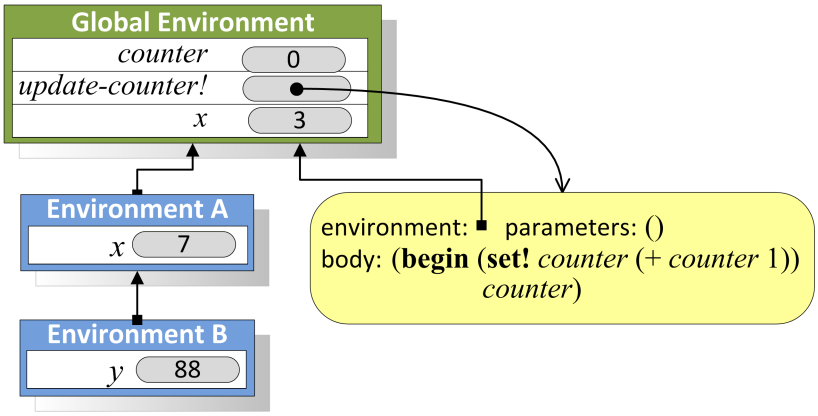

Figure 9.1. Sample environments.

The global environment contains a frame with three names. Each name has an associated place that contains the value associated with that name. The value associated with *counter* is the currently 0. The value associated with *set-counter!* is the procedure we defined in Example 9.1. A procedure is characterized by its parameters, body code, and a pointer to the environment in which it will be evaluated.

tinues 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 name is undefined and the evaluation results in an error.

Definition. The revised evaluation rule for a definition is:

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

The rule for redefinitions means we could use **define** in some situations to mean something similar to **set!**. The meaning is different, though, since an assignment finds the place associated with the name and puts a new value in that place. Evaluating an assignment follows the Stateful Evaluation Rule 2 to find the place associated with a name. Hence, (**define** *Name Expression*) has a different meaning from (**set!** *Name Expression*) when there is no place named *Name* in the current execution environment. To avoid this confusion, only use **define** for the first definition of a name and always use **set!** when the intent is to change the value associated with a name.

Application. The final rule that must change because of mutation is the application rule for constructed procedures. Instead of using substitution, the new application rule creates a new environment with a frame containing places named for the parameters.

> **Stateful Application Rule 2: Constructed Procedures.** To apply a constructed procedure:
>
> 1. Construct a new environment, whose parent is the environment of the applied procedure.
> 2. For each procedure parameter, create a place in the frame of the new environment with the name of the parameter. Evaluate each operand expression in the environment or the application and initialize the value in each place to 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.

Consider evaluating the application expression (*bigger* `3 4`) where *bigger* is the procedure from Example 3.3: (**define** (*bigger a b*) (**if** (> *a b*) *a b*))).

To evaluate an application of *bigger* follow Stateful Application Rule 2. First, create a new environment. Since *bigger* 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.

Next, create places in the new environment's frame named for the procedure parameters, *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. Figure 9.2 shows the resulting environment. The final step is to evaluate the body expression, (**if** (> *a b*) *a b*), in the newly created environment. The values of *a* and *b* are found in the application environment.

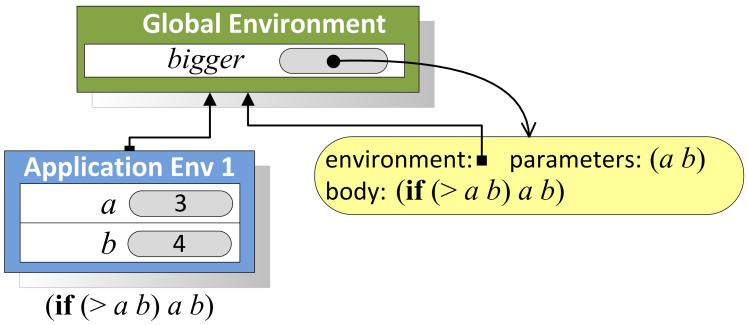

Figure 9.2. Environment created to evaluate (*bigger* 3 4).

The new application rule becomes more interesting when we consider procedures that create new procedures. For example, *make-adder* takes a number as input and produces as output a procedure:

(**define** (*make-adder v*) (**lambda** (*n*) (+ *n v*)))

The environment that results from evaluating (**define** *inc* (*make-adder* 1)) is shown in Figure 9.3. The name *inc* has a value that is the procedure resulting from the application of (*make-adder* 1). To evaluate the application, we follow the application rule above and create a new environment containing a frame with the parameter name, *inc*, and its associated operand value, 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

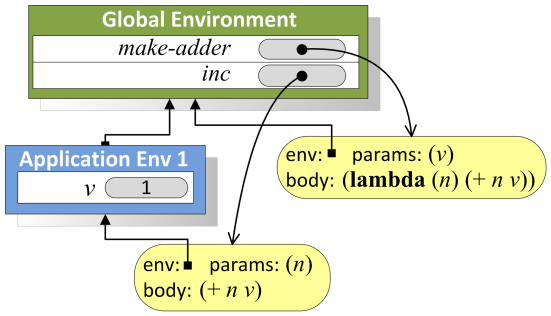

Figure 9.3. Environment after evaluating (define *inc* (*make-adder* 1)).

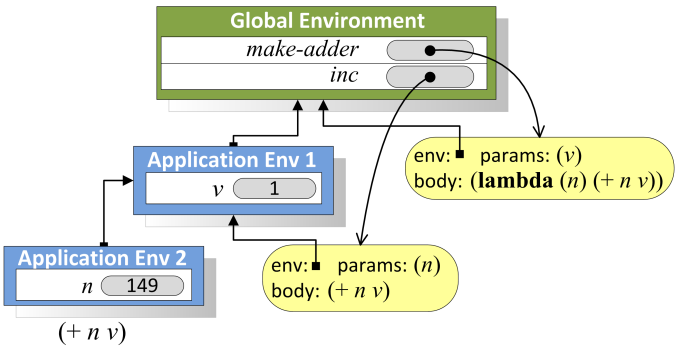

Figure 9.4. Environment for evaluating the body of (*inc* 149).

created to evaluate the application of *make-adder*, hence, its environment pointer points to the application environment which contains a place named *inc* holding the value 1.

Next, consider evaluating (*inc* 149). Figure 9.4 illustrates the environment for evaluating the body of the *inc* procedure. The evaluation creates a new environment with a frame containing the place *n* and its associated value 149. We evaluate the body of the procedure, (+ *n v*), in that environment. The value of *n* is found in the execution environment. The value of *v* is not found there, so evaluation continues by looking in the parent environment. It contains a place *v* containing the value 1.

Exercise 9.1. Devise a Scheme expression that has four possible values depending on the order in which application subexpressions are evaluated.

Exercise 9.2. Draw the environment that results 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 that results after evaluating the following expressions, and explain what the value of the final expression is. (Hint: first, rewrite the let expression as an application.)

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

9.3 Mutable Pairs and Lists

The Pair datatype introduced in Chapter 5 is *immutable*. This means that once a Pair is created, the values in its cells cannot be changed.[2] *immutable*

The MutablePair datatype is a mutable pair. A MutablePair is constructed using *mcons*, which is similar to *cons* but produces a MutablePair. The parts of a MutablePair can be extracted using the *mcar* and *mcdr* procedures, which behave analogously to the *car* and *cdr* procedures. A MutablePair is a distinct datatype from a Pair; it is an error to apply *car* to a MutablePair, or to apply *mcar* to an immutable Pair.

The MutablePair datatype also provides two procedures that change the values in the cells of a MutablePair:

set-mcar!: MutablePair $\times$ Value $\rightarrow$ Void
: Replaces the value in the first cell of the MutablePair with the value of the second input.

set-mcdr!: MutablePair $\times$ Value $\rightarrow$ Void
: Replaces the value in the second cell of the MutablePair with the value of the second input.

The Void result type indicates that *set-mcar!* and *set-mcdr!* do not output any value.

Here are some interactions using a MutablePair:

```
> (define pair (mcons 1 2))
> (set-mcar! pair 3)
> pair
(3 . 2)
> (set-mcdr! pair 4)
> pair
(3 . 4)
```

The *set-mcdr!* procedure allows us to create a pair where the second cell of the pair is itself: (*set-mcdr! pair pair*). This produces the rather frightening object shown in Figure 9.5.

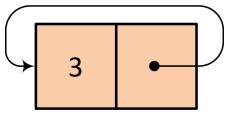

Figure 9.5. Mutable pair created by evaluating (*set-mcdr! pair pair*).

[2]The mutability of standard Pairs is quite a controversial issue. In most Scheme implementations and the standard definition of Scheme, a standard *cons* pair is mutable. But, as we will see later in the section, mutable pairs cause lots of problems. So, the designers of DrScheme decided for Version 4.0 to make the standard Pair datatype immutable and to provide a MutablePair datatype for use when mutation is needed.

Every time we apply *mcdr* to *pair*, we get the same pair as the output. Hence, the value of (*mcar* (*mcdr* (*mcdr* (*mcdr pair*)))) is 3.

We can also create objects that combine mutable and immutable Pairs. For example, (**define** *mstruct* (*cons* (*mcons* 1 2) 3)) defines *mstruct* as an immutable Pair containing a MutablePair in its first cell. Since the outer Pair is immutable, we cannot change the objects in its cells. Thus, the second cell of *mstruct* always contains the value 3. We can, however, change the values in the cells of the mutable pair in its first cell. For example, (*set-mcar!* (*car mstruct*) 7) replaces the value in the first cell of the MutablePair in the first cell of *mstruct*.

Mutable Lists. As we used immutable Pairs to build immutable Lists, we can use MutablePairs to construct MutableLists. A MutableList is either null or a MutablePair whose second cell contains a MutableList.

The MutableList type is defined by a library. To use it, evaluate the following expression: (*require scheme/mpair*). All of the examples in this chapter assume this expression has been evaluated. This library defines the *mlist* procedure that is similar to the *list* procedure, but produces a MutableList instead of an immutable List. For example, (*mlist* 1 2 3) produces the structure shown in Figure 9.6.

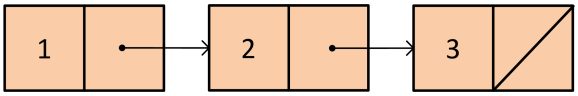

Figure 9.6. MutableList created by evaluating (*mlist* 1 2 3).

Each node in the list is a MutablePair, so we can use the *set-mcar!* and *set-mcdr!* procedures to change the values in the cells.

```
> (define m1 (mlist 1 2 3))
> (set-mcar! (mcdr m1) 5)
> (set-mcar! (mcdr (mcdr m1)) 0)
> m1
{1 5 0} ; DrScheme denotes MutableLists using curly brackets.
```

Many of the list procedures from Chapter 5 can be directly translated to work on mutable lists. For example, we can define *mlist-length* as:

```
(define (mlist-length m)
  (if (null? m) 0 (+ 1 (mlist-length (mcdr m)))))
```

As shown in Exercise 9.4, though, we need to be careful when using *mcdr* to recurse through a MutableList since structures created with MutablePairs can include circular pointers.

Exercise 9.4. What does (*mlist-length pair*) evaluate to for the pair shown in Figure 9.5?

Exercise 9.5. [⋆] Define a *mpair-circular?* procedure that takes a MutablePair as its input and outputs true when the input contains a cycle and false otherwise.

9.4 Imperative Programming

Mutation enables a style of programming known as *imperative programming*. Whereas *functional programming* 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 a state that provides a solution to a problem.

imperative programming

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 to the next procedure application. With imperative programming, the primary operation is assignment (performed by **set!**, *set-mcar!*, and *set-mcdr!* in Scheme; but typically by an assignment operator, often := or =, in languages designed for imperative programming such as Pascal, Algol60, Java, and Python).

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

9.4.1 List Mutators

All the procedures for changing the value of a list in Section 5.4.3 actually do not change any values; instead they construct new lists. When our goal is only to change some elements in an existing list, this wastes memory constructing 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 5.4.3, but instead of producing new lists with the desired property these procedures modify the input list.

Example 9.2: Mapping. The *list-map* procedure (from Example 5.4) produces a new list that is the result of applying the same procedure to every element in the input list.

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

Whereas the functional *list-map* procedure uses *cons* to build up the output list, the imperative *mlist-map!* procedure uses *set-car!* to mutate the input list's elements:

```
(define (mlist-map! f p)
  (if (null? p) (void)
    (begin (set-mcar! p (f (mcar p)))
           (mlist-map! f (mcdr p)))))
```

The base case uses (*void*) to evaluate to no value. Unlike *list-map*, *mlist-map!* produces no output but is used for its side effects.

Assuming the procedure passed as f has constant running time, the running time of the *mlist-map!* procedure is in $\Theta(n)$ where n is the number of elements in the input list. There will be n recursive applications of *mlist-map!* since each one passes in a list one element shorter than the input list, and each application requires constant time. This is asymptotically the same as the *list-map* procedure, but we would expect the actual running time to be faster since there is no need to construct a new list.

The memory consumed is asymptotically different. The *list-map* procedure allocates n new *cons* cells, so it requires memory in $\Theta(n)$ where n is the number of elements in the input list. The *mlist-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 9.3: Filtering. The *list-filter* procedure takes as inputs a test procedure and a list and outputs a list containing the elements of the input list for which applying the test procedure evaluates to a true value. In Example 5.5, we defined *list-filter* as:

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

An imperative version of *list-filter* removes the unsatisfying elements from a mutable list. We define *mlist-filter!* using *set-mcdr!* to skip over elements that should not be included in the filtered list:

```
(define (mlist-filter! test p)
  (if (null? p) null
    (begin (set-mcdr! p (mlist-filter! test (mcdr p)))
           (if (test (mcar p)) p (mcdr p))))))
```

Assuming the test procedure has constant running time, the running time of the *mlist-filter!* procedure is linear in the length of the input list. As with *mlist-map!*, the space used by *mlist-filter!* is constant, which is better than the $\Theta(n)$ space used by *list-filter*.

Unlike *mlist-map!*, *mlist-filter!* outputs a value. This is needed when the first element is not in the list. Consider this example:

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

```
{2 3 4}
> a
{1 2 3 4}
```

The value of *a* still includes the initial 1. There is no way for *mlist-filter!* to remove the first element of the list: the *set-mcar!* and *set-mcdr!* procedures only enable us to change what the mutable pair's components contain.

To avoid this, *mlist-filter!* should be used with **set!** to assign the variable to the resulting mutable list:

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

Example 9.4: Append. The *list-append* procedure takes as input two lists and produces 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 to contain the elements of both lists.

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

The *mlist-append!* procedure produces an error when the first input is *null* — this is necessary since if the input is *null* there is no pair to modify.[3]

Like *list-append*, the running time of the *mlist-append!* procedure is in $\Theta(n)$ where n is the number of elements in the first input list. The *list-append* procedure copies the first input list, so its memory use is in $\Theta(n)$ where n is the number of elements in the first input list. The memory use of *mlist-append!* is constant: it does not create any new cons cells to append the lists.

Aliasing. Adding mutation makes it possible to define many procedures more efficiently and compactly, but introduces many new potential pitfalls in producing reliable programs. Since our evaluation model now depends on the environment in which an expression is evaluated, it becomes much harder to reason about code by itself.

One challenge introduced by mutation is *aliasing*. There may be different ways to refer to the same object. This was true before mutation also, but didn't matter since the value of an object never changed. Once object values can change, however, aliasing can lead to surprising behaviors. For example, *aliasing*

```
> (define m1 (mlist 1 2 3))
> (define m2 (mlist 4 5 6))
> (mlist-append! m1 m2)
```

[3] The *mappend!* library procedure in DrScheme takes a different approach: when the first input is null it 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 the value of *a* is not modified.

> (**set!** *m1* (*mlist-filter!* (**lambda** (*el*) (= (*modulo el* 2) 0)) *m1*))

The value of *m2* was defined as {4 5 6}, and no expressions since then explicitly modified *m2*. But, the value of *m2* has still changed! It changed because after evaluating (*mlist-append! m1 m2*) the *m1* object shares cells with *m2*. Thus, when the *mlist-filter!* application changes the value of *m1*, it also changes the value of *m2* to {4 6}.

The built-in procedure *eq?* takes as input any two objects and outputs a Boolean. The result is true if and only if the inputs are the same object. For example, (*eq?* 3 3) evaluates to true but (*eq?* (*mcons* 1 2) (*mcons* 1 2)) evaluates to false. Even though the input pairs have the same value, they are different objects—mutating one of the pairs does not effect the value of the other pair.

For the earlier *mlist-append!* example, (*eq? m1 m2*) evaluates to false since *m1* and *m2* do not refer to the same object. But, (*eq?* (*mcdr m1*) *m2*) evaluates to true since the second cell of *m1* points to the same object as *m2*. Evaluating (*set-mcar! m2* 3) changes the value of both *m1* and *m2* since the modified cell is common to both structures.

Exercise 9.6. Define an imperative-style procedure, *mlist-inc!* that takes as input a MutableList of Numbers and modifies the list by adding one to the value of each element in the list.

Exercise 9.7. [⋆] Define a procedure *mlist-truncate!* that takes as input a MutableList and modifies the list by removing the last element in the list. Specify carefully the requirements for the input list to your procedure.

Exercise 9.8. [⋆] Define a procedure *mlist-make-circular!* that takes as input a MutableList and modifies the list to be a circular list containing all the elements in the original list, repeated indefinitely. For example, (*mlist-make-circular!* (*mlist* 3)) should produce the same structure as the circular pair shown in Figure 9.5.

Exercise 9.9. [⋆] Define an imperative-style procedure, *mlist-reverse!*, that reverses the elements of a list. Is it possible to implement a *mlist-reverse!* procedure that is asymptotically faster than the *list-reverse* procedure from Example 5.4?

Exercise 9.10. [⋆⋆] Define a procedure *mlist-aliases?* that takes as input two mutable lists and outputs true if and only if there are any mcons cells shared between the two lists.

9.4.2 Imperative Control Structures

The imperative style of programming makes progress by using assignments to manipulate state. In many cases, solving a problem requires repeated opera-

tions. 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 pass 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. *while loop*

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 result value
```

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

```
(define (fibo-while n)
  (let ((a 1) (b 1))
    (while (lambda () (> n 2))
           (lambda () (set! b (+ a b))
                      (set! a (- b a))
                      (set! n (- n 1))))
    b))
```

The final value of b is the result of the *fibo-while* procedure. In each iteration, the body procedure is applied, updating the values of a and b to the next Fibonacci numbers.

The value assigned to a is computed as $(- \; b \; a)$ instead of b. The reason for this is the previous assignment 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. The fact that the value of a variable can change depending on when it is used often makes imperative programming trickier than functional programming.

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! n (- n 1))))
```

Many programming languages provide control constructs similar to the *while* procedure defined above. For example, here is a version of the procedure in the Python programming language:

```
def fibonacci(n):
  a = 1
  b = 1
  while n > 2:
    a, b = b, a + b
    n = n - 1
  return b
```

We use Python starting in Chapter 11, although you can probably guess what most of this procedure means without knowing Python. The most interesting statement is the double assignment: *a, b = b, a + b*. This assigns the new value of *a* to the old value of *b*, and the new value of *b* to the sum of the old values of *a* and *b*. Without the double assignment operator, it would be necessary to store the old value of *b* in a new variable so it can be assigned to *a* after updating *b* to the new value.

Exercise 9.11. Define *mlist-map!* from the previous section using *while*.

repeat-until

Exercise 9.12. Another common 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)))
     (lambda () (< n 1)))
    fact))
```

Exercise 9.13. [⋆⋆] Improve the efficiency of the indexing procedures from Section 8.2.3 by using mutation. Start by defining a mutable binary tree abstraction, and use this and the MutableList data type to implement an imperative-style *insert-into-index!* procedure that mutates the input index by adding a new word-position pair to it. Then, define an efficient *merge-index!* procedure that takes two mutable indexes as its inputs and modifies the first index to incorporate all word occurrences in the second index. Analyze the impact of your changes on the asymptotic running time.

9.5 Summary

Adding the ability to change the value associated with a name complicates our evaluation rules, but enables simpler and more efficient solutions to many problems. Mutation allows us to efficiently manipulate larger data structures since it is not necessary to copy the data structure to make changes to it.

Once we add assignment to our language, the order in which things happen affects the value of some expressions. Instead of evaluating expressions using substitution, we now need to always evaluate an expression in a particular execution environment.

The problem with mutation is that it makes it much tougher to reason about the meaning of an expression. In the next chapter, we introduce a new kind of abstraction that packages procedures with the state they manipulate. This helps manage some of the complexity resulting from mutation by limiting the places where data may be accessed and modified.