14 Declaring Special Forms

14.1 Overview

In Scheme, operators are applied to their operands by enclosing the operator followed by its operands in parentheses. The call structure for applying an operator to its operands is:

(*operator operand* ...)

When such an application is made, the operator and the operands are evaluated in an unspecified order,[1] and then the procedure (which is the value of the operator) is applied to the arguments (which are the values of the operands).

We have also encountered several special forms in which the subexpressions following the keyword are treated differently from the operands of a procedure. Examples of these are `and`, `begin`, `cond`, `case`, `define`, `if`, `lambda`, `let`, `let*`, `letrec`, `or` and `set!`, each with a syntax of its own. Some of these, like `let`, have been introduced to make it easier to read programs, for any program using `let` could be rewritten using an application of a `lambda` expression in place of each `let` expression. Such keywords are referred to as *derived keywords*. One of the convenient features of Scheme is that it is an extensible language that allows the user to add new special forms to make the language more convenient to use and to provide a mechanism to do tasks that procedures cannot perform. We shall study two mechanisms for making such additions in this chapter.

---

[1] Programs that rely on an order of evaluation are said to be ill formed. Since the order of evaluation is implementation dependent, such programs are not portable, and they can not, in general, be transferred from one implementation to another.

The action of taking an expression and rewriting it in terms of something we understand happens when we work with natural language. As we read a passage, we often look in a syntax table, a dictionary, and substitute the meaning of the word for the word itself. In Scheme, however, we restrict those items for which substitutions can be made (we also say "which can be transformed") to be lists that begin with a keyword (these are the special forms). Before an expression can be evaluated, all special forms in the expression must be transformed into expressions that are "understood." To carry the metaphor a bit further, we cannot understand the complete thought conveyed by the author of a passage until we have transformed all terms into words we understand. In a sense, we cannot evaluate the author's passage without the appropriate substitutions taking place. Similarly, we cannot evaluate a Scheme expression until all the transformations have occurred. Each transformation brings the expression closer to one in which all terms are familiar. Thus, we do not evaluate an expression with a list that begins with a derived keyword. When all such lists have been transformed, it is time to evaluate the expression. Prior to evaluation there is a recursive program that removes all such lists.[2]

14.2 Declaring a Simple Special Form

In this book we have used several special forms without defining them as procedures. In fact, it is the nature of these forms that they cannot (or should not) be defined as procedures either because some of their *operands* are not to be evaluated or because the order of evaluation of their operands is not the same as in a procedure application. We use the terminology that we *define* procedures, but we *declare* special forms. The mechanism for declaring special forms will be explained in the course of making a specific extension to the syntax.

If we write

```
(define sm (+ 3 4))
```

[2] We shall not write that procedure here, since the way it is written is determined by what the system assumes it *knows*. For purposes of discussion, we assume the system knows `define`, `if`, `lambda`, `quote`, and `set!`. Other systems might know about a different set of special forms. For example, `if` might be described in terms of `cond`, thereby causing us to assume that the system knows `cond`. This freedom of choice gives implementors the flexibility they need for efficient implementation.

the expression (+ 3 4) is evaluated and its value is bound to the variable sm. Suppose that we want to assign this expression to the variable sm but postpone the evaluation of the expression (+ 3 4) until we actually need the value of sm. One way of doing this is to encapsulate the expression (+ 3 4) within the body of a lambda expression having no arguments. We could then write

```
(define sm (lambda () (+ 3 4)))
```

The body of a lambda expression is not evaluated until that lambda expression is applied to its arguments, and since the thunk (lambda () (+ 3 4)) has no arguments, it is invoked by merely enclosing the lambda expression in parentheses. Since the thunk in this case is bound to the variable sm, we can invoke it by enclosing sm in parentheses, that is, by writing (sm). We are thus able to postpone the evaluation of an expression until we need it by making it into a thunk and binding a variable to that thunk. It would be nice to have a procedure **freeze** that, when applied to an operand, has the effect of forming a thunk that has that operand as its body. Suppose we write:

```
(define freeze
  (lambda (expr)
    (lambda () expr)))
```

Then we would write:

```
(define sm (freeze (+ 3 4)))
```

But when the define expression is evaluated, before being bound to sm, the expression (**freeze** (+ 3 4)) is evaluated. Since **freeze** is a procedure, its operand (+ 3 4) is evaluated. Thus we defeated the purpose for which we wrote the procedure **freeze**, which was to postpone the evaluation of its operand until sm is called. What happened is that (+ 3 4) is evaluated during the definition of sm instead of when sm is called. Thus **freeze** cannot be a procedure; it has to be the keyword of a special form if it is to accomplish what we want.

To declare this special form with keyword **freeze**, we make use of a special form with keyword macro.[3] We would like **freeze** to have the syntax (**freeze** *expr*) and to transform into the thunk (lambda () *expr*) without evaluating

[3] At the time this book is being written, the Scheme community has not yet agreed upon a standard way of declaring special forms. In this book, we use two methods that have been

the expression *expr*. We call the expression (`freeze` *expr*) the *macrocode*, and we want to transform the macrocode into the *macroexpansion*

```
(lambda () expr)
```

In general, a *macro* is a procedure that transforms macrocode into the corresponding macroexpansion.

When an expression is entered into the system, the first subexpression is checked to see if it is a keyword of some special form. If it is, then the macrocode (in our case, (`freeze` *expr*)) is replaced by the corresponding macroexpansion. Then at run time, the computer sees only the macroexpansion (`lambda ()` *expr*) in the program as if we had written the macroexpansion into the program instead of the macrocode. Thus the subexpression *expr* of the special form (`freeze` *expr*) was not evaluated when the procedure (or thunk) was created by evaluating (`lambda ()` *expr*).

How is the macroexpansion accomplished? We have to write a procedure that literally transforms the macrocode into the macroexpansion of that code. Let us call that procedure `freeze-transformer`; it takes the macrocode `code` as its argument and returns the code for the macroexpansion. In our case, the macroexpansion is a list containing the three items that make up a lambda expression: the symbol `lambda`, the empty list of arguments, and the body. Thus we can define `freeze-transformer` to be:

```
(define freeze-transformer
  (lambda (code)
    (make-lambda-expression '() (list (2nd code)))))
```

where `make-lambda-expression` is applied to the formal parameter(s) (in this case, it is the empty list) and a list of expressions (in this case, it is a list containing only one element). The second expression in the macrocode is *expr*. In our specific example, that is the list `(+ 3 4)`. We define `make-lambda-expression` to be:

```
(define make-lambda-expression
  (lambda (parameters body-expressions)
    (cons 'lambda (cons parameters body-expressions))))
```

---

included in some implementations. These methods use special forms with keywords `macro` and `extend-syntax`. If these are not implemented in the version you are using, read the manual for your implementation to see how it declares special forms, and use that method instead. In general, until a standard is agreed upon, code including user-made special forms is not portable.

Now that we have defined the freeze-transformer, we can declare the special form with keyword freeze using the special form with keyword macro as follows:

```
(macro freeze freeze-transformer)
```

We can conceive of this process of declaring a special form as if macro places the keyword freeze in a global table we call the *syntax table*, along with its *transformer*, which is the procedure freeze-transformer. Thus each entry in the syntax table consists of a keyword and its associated transformer. When a program is entered and the symbol freeze is found in the first position of an expression, it looks it up in the syntax table, and if it finds it there, it passes the macrocode ((freeze *expr*) in this case) to the transformer. The transformer then returns the macroexpansion (in our example, (lambda () *expr*)). This macroexpansion is inserted into the program in place of the macrocode. It is customary to refer to the keyword freeze as a macro, though the macro actually is the whole macrocode. Following custom, we shall say "the macro freeze."

We can also unwrap the various helping procedures used in defining the procedure freeze-transformer to get a self-contained representation for the macro declaration. For example, we can replace

```
(make-lambda-expression '() (list (2nd code)))
```

by the body of its lambda expression with its parameters replaced by the arguments to which they are bound to get:

```
(define freeze-transformer
  (lambda (code)
    (cons 'lambda (cons '() (list (2nd code))))))
```

Finally, replacing freeze-transformer by its lambda expression gives us

Program 14.1 freeze

```
(macro freeze
  (lambda (code)
    (cons 'lambda (cons '() (list (2nd code))))))
```

as a self-contained form of the declaration of the macro `freeze`. Either the version using the helping procedures or this final self-contained version declares the macro `freeze`. You may use the version you find more convenient.

14.3 Macros

In general, the special form with keyword `macro` has the syntax

(macro *name transformer*)

where *name* is the keyword of the new special form being declared and *transformer* is a procedure of one argument that takes the macrocode and returns the macroexpansion. In our example above, `freeze` is the keyword, and

```
(lambda (code)
  (cons 'lambda (cons '() (list (2nd code)))))
```

is the transformer. Thus we summarize by recalling that when a program containing an expression starting with a keyword for a special form is entered, the system replaces the macrocode by the code returned when the macrocode is passed to the keyword's transformer. It is this expansion that is seen when the program is run.

The macro `freeze` can also be implemented to take several subexpressions; this would let us write, for example,

```
(freeze (writeln "Hello") "How are you?")
```

and would macro expand into

```
(lambda () (writeln "Hello") "How are you?")
```

In general, we would like `freeze` to have the syntax

(freeze $expr_1$ $expr_2$ ...)

where the ellipsis (three dots) means that there is a finite number of expressions following the word `freeze` and that there is at least one such expression.[4]

---

[4] In general, the notation *thing* ... means zero or more occurrences of *thing*, whereas $thing_1$ $thing_2$ ... means one or more occurrences of *thing*.

This is a pattern for our macrocode but it cannot be used as the macrocode itself since it contains the ellipsis and the special form macro will not know what to do with it. Using a similar notation, we can say that a pattern for the macroexpansion is:

(**lambda** () $expr_1$ $expr_2$ ...)

A convenient notation to indicate that the first pattern is to be expanded into the second pattern is:

(**freeze** $expr_1$ $expr_2$ ...) $\equiv$ (**lambda** () $expr_1$ $expr_2$ ...)

The symbol $\equiv$ can be read "macro expands to." We call a statement that has the macro pattern on the left and the expansion pattern on the right a *syntax table entry*.

In any actual case, the macrocode is a list that starts with the keyword freeze and always has at least one expression following it. If we represent this macrocode by the variable code again, then (cdr code) is just a list of the expressions that make up the body of the lambda expression into which the macrocode is expanded. The freeze-transformer procedure defined above can be modified so that it produces the right macroexpansion for this version of freeze:

```
(define freeze-transformer
  (lambda (code)
    (make-lambda-expression '() (cdr code))))
```

It would be convenient if Scheme were to have a way of taking the two sides of the syntax table entry and declare the special form for us. In essence, the system would be writing the transform procedure for us and using it to declare the macro. Such a special form, called extend-syntax,[5] was developed (see Kohlbecker, 1986). It has the following syntax:

---

[5] Here is a way to get macro if you have extend-syntax in your implementation of Scheme:

```
(extend-syntax (macro)
  ((macro name transformer)
   (let ((t transformer))
     (extend-syntax (name)
       (x ((with ((w 'with)) w) ((v (t 'x))) v))))))
```

See Dybvig, 1987, for a discussion of extend-syntax's with clauses.

```
(extend-syntax (name ...) (macro-pattern expansion-pattern) ...)
```

where *macro-pattern* and *expansion-pattern* are the left and right sides, respectively, of the syntax table entry for the macro called *name*. Using **extend-syntax**, the declaration of the macro **freeze** becomes:

Program 14.2 **freeze**

```
(extend-syntax (freeze)
  ((freeze expr1 expr2 ...) (lambda () expr1 expr2 ...)))
```

Since no standard way of making special forms has been agreed upon, we shall demonstrate both ways of doing it—that is, using **macro** and **extend-syntax** in the rest of this chapter.

Along with the macro **freeze**, there is the procedure **thaw**, which invokes a frozen entity (a thunk) and returns its value. The procedure **thaw** is defined as follows:

Program 14.3 **thaw**

```
(define thaw
  (lambda (thunk)
    (thunk)))
```

To show how it is used, we define:

```
(define th (freeze (display "A random number is: ") (random 10)))

(thaw th) ⟹ A random number is: 7
(thaw th) ⟹ A random number is: 3
```

Each time the thunk is thawed, the expressions are reevaluated. Thus each time we thawed the thunk **th** in the example, another random number is computed and returned.

There are occasions when we want to postpone the evaluation of an expression but have it be evaluated only the first time it is called and thereafter not have to reevaluate the expression each time it is called again but rather return on each subsequent call the value already evaluated. This would be advantageous if the same long calculation is involved each time the procedure

Program 14.4 make-promise, force

```
(define make-promise "procedure")
(define force "procedure")

(let ((delayed-tag "delay") (value-tag "-->"))
  (set! make-promise (lambda (thunk) (cons delayed-tag thunk)))
  (set! force
    (lambda (arg)
      (if (and (pair? arg) (eq? (car arg) delayed-tag))
          (begin
            (set-car! arg value-tag)
            (set-cdr! arg (thaw (cdr arg)))))
      (cdr arg))))
```

is called and the result obtained is the same, in the absence of side effects. We propose to evaluate the postponed expression only the first time it is called and on subsequent calls to return the already computed value. We declare the special form `delay` to postpone the evaluation by creating a *promise*, and a corresponding procedure `force` to evaluate (or "force") the promise. When the promise is forced for the first time, the value of the postponed expression is computed and returned. Each succeeding time the promise is forced, the same value that was computed the first time is returned. Consider the following:

```
(define pr (delay (display "A random number is: ") (random 10)))

(force pr) ⟹ A random number is: 6
(force pr) ⟹ 6
(force pr) ⟹ 6
```

and it continues returning 6 each time it is forced from now on.

The syntax table entry for `delay` is

(delay *expr*$_1$ *expr*$_2$ ...) ≡ (make-promise (freeze *expr*$_1$ *expr*$_2$ ...))

where `make-promise` is a procedure that takes a thunk as its argument and returns a promise, which is a thunk tagged with `"delay"`. (See Program 14.4.) If `force`'s argument is a promise, force converts the promise into a *fulfillment*. A promise is converted into a fulfillment by tagging with `"-->"` the value obtained by thawing the promise's thunk. In any event, the value stored in the fulfillment is returned. Program 14.4 is written so as to protect the tags from accidental reassignment.

We can now proceed to declare the macro delay. It has the macrocode

(delay $expr_1$ $expr_2$ ...)

which macroexpands into

(make-promise (freeze $expr_1$ $expr_2$ ...))

As before, we cannot define delay to be a procedure because its arguments $expr_1$ $expr_2$ ... would be evaluated too early. Using extend-syntax, we can declare delay by simply writing:

Program 14.5 delay

```
(extend-syntax (delay)
  ((delay expr1 expr2 ...) (make-promise (freeze expr1 expr2 ...))))
```

Or, by using macro, we get

Program 14.6 delay

```
(define delay-transformer
  (lambda (code)
    (list 'make-promise (cons 'freeze (cdr code)))))

(macro delay delay-transformer)
```

As we have seen, in a procedure call, Scheme first evaluates the operands (producing arguments) and the operator (producing a procedure) and then applies the procedure to the arguments. We say that the arguments are passed to the procedure "by value." In some languages, arguments are passed to procedures as if they were thunks, and they are not thawed until they are actually used in the procedure. Such arguments are said to be passed to the procedure "by name."[6] We can write programs in Scheme so that procedures

[6] In the presence of side effects, this is an oversimplification.

accept arguments that are thunks. These arguments are thawed when they are used in the body of the procedure, so that passing of arguments by name can be accomplished in Scheme. Similarly, it is possible to pass arguments to procedures as promises, which are not forced until they are needed in the body of the procedures. In such cases, the arguments are said to be passed "by need." In Chapter 15, we shall study streams, which use arguments passed by need.

We have been using the special form with keyword **let**, which has the syntax[7]

$$\texttt{(let ((}var\ val\texttt{) ...)}\ expr_1\ expr_2\ \texttt{...)}$$

The syntax table entry for **let** is

$$\texttt{(let ((}var\ val\texttt{) ...)}\ expr_1\ expr_2\ \texttt{...)}$$
$$\equiv$$
$$\texttt{((lambda (}var\ \texttt{...)}\ expr_1\ expr_2\ \texttt{...)}\ val\ \texttt{...)}$$

The declaration of `let` is now a simple matter when we use **extend-syntax** as in Program 14.7.

Program 14.7 `let`

```
(extend-syntax (let)
  ((let ((var val) ...) expr1 expr2 ...)
   ((lambda (var ...) expr1 expr2 ...) val ...)))
```

To declare **let** with **macro**, we have to build an application that consists of a list containing a lambda expression followed by its operands. For the lambda expression, we need its parameter list and its body expressions. If **code** represents the macrocode, then the list of parameters is built up by first taking the **(2nd code)** to get a list of pairs of *var*'s and *val*'s. We extract the list of *var*'s by taking the **1st** of each pair in the list using **map** as follows:

[7] When using user-declared macros that have the same keywords as special forms in Scheme, you might want to avoid collisions with the built-in forms. We suggest that you surround the keywords of those you declare with equal signs; e.g., `=let=` in place of `let`.

```
(define make-list-of-parameters
  (lambda (code)
    (map 1st (2nd code))))
```

Similarly, we can build the list of operands from the macrocode by taking the **2nd** of each pair. This leads to:

```
(define make-list-of-operands
  (lambda (code)
    (map 2nd (2nd code))))
```

A list of the items in the body of the lambda expression we are building is obtained by taking the cddr of the macrocode. Thus:

```
(define make-list-of-body-items
  (lambda (code)
    (cddr code)))
```

With these helping procedures, we can write the transform procedure and declare it as the macro for `let`.

Program 14.8 `let`

```
(define let-transformer
  (lambda (code)
    (cons (make-lambda-expression
            (make-list-of-parameters code)
            (make-list-of-body-items code))
          (make-list-of-operands code))))

(macro let let-transformer)
```

This is really only half of the declaration of the macro `let` since there is also the so-called *named* `let`, which has a different syntax. We shall return to the *named* `let` in the exercises, where we rely on the following discussion of `letrec`. The above version of the macro declaration of `let` using the special form with keyword `macro` clearly illustrates the advantage of using `extend-syntax` to declare a macro. Exercise 14.6 at the end of this section suggests some interesting modifications to `let` so that it displays appropriate messages when an expression with keyword `let` is entered with an incorrect syntax. For example, if we write `(let ((a 3)))`, incorrect syntax should be signaled since a let expression must contain at least one subexpression following the binding pairs. If we use **macro** to declare our special forms, we must explicitly include tests in the definition of the transformer to determine if the syntax is correct. On the other hand, one of the great advantages of using **extend-syntax** is that it has built-in syntax checking, so we do not have to include our own tests for correct syntax. You may find it instructive to enter some let expressions with incorrect syntax in your implementation of Scheme and see the messages that are displayed.

We observed that in a let expression of the form

(let ((*var val***) ...)** $expr_1$ $expr_2$ **...)**

the expression *val* ... whose value will be bound to *var* ... cannot contain *var* ... recursively, for looking at the pattern for the macroexpansion,

((lambda (*var* **...)** $expr_1$ $expr_2$ **...)** *val* **...)**

we see that *val* ... is not in the scope of *var* ..., so any instance of *var* ... in *val* ... refers to an outer scope. The special form **letrec** does allow for a recursive scope.

The macro **letrec** has the syntax table entry:

(letrec ((*var val***) ...)** $expr_1$ $expr_2$ **...)**

$\equiv$

(let ((*var* **"any") ...) (begin (set!** *var val***) ...)** $expr_1$ $expr_2$ **...)**

In this expansion, if any one of the *val*'s contains instances of any of the *var*'s, that *val* is in the lexical scope of those *var*'s in the let expression of the macroexpansion. This allows the use of recursion in *var*. Let us now write the macro for **letrec**. Again, it is a simple matter to do so using **extend-syntax**.

Program 14.9 **letrec**

```
(extend-syntax (letrec)
  ((letrec ((var val) ...) expr1 expr2 ...)
   (let ((var "any") ...)
     (set! var val) ...
     expr1 expr2 ...)))
```

Consider the definition of the procedure **odd?**, which is defined using a letrec expression:

```
(define odd?
  (letrec
    ((even? (lambda (n) (if (zero? n) #t (odd? (sub1 n)))))
     (odd? (lambda (n) (if (zero? n) #f (even? (sub1 n))))))
    odd?))
```

It macroexpands into the following let expression:

```
(define odd?
  (let ((even? "any")
        (odd? "any"))
    (begin
      (set! even? (lambda (n) (if (zero? n) #t (odd? (sub1 n)))))
      (set! odd? (lambda (n) (if (zero? n) #f (even? (sub1 n))))))
    odd?))
```

Let us next look at how to declare `letrec` using `macro`. We first consider how we construct the pairs of the form `(var "any")`, which are in the let expressions of the macroexpansion. After we get the `var`'s from the `2nd` of the macrocode, we use `map` to give us the desired pairs of the form `(var "any")`. Similarly, we build the set! expressions, and finally, we build a list of expressions that complete the body of the let expression. This leads to the declaration of `letrec` using `macro` that is given in Program 14.10.

Program 14.10 `letrec`

```
(macro letrec
  (lambda (code)
    (cons 'let
      (cons (map (lambda (x) (list (1st x) "any")) (2nd code))
        (append
          (map (lambda (x) (cons 'set! x)) (2nd code))
          (cddr code))))))
```

Something you usually want to avoid is the creation of infinite loops. However, as an interesting demonstration of the use of `letrec`, we shall write a special form `cycle` that takes an arbitrary number of subexpressions and runs each subexpression in succession and then starts over again, repeating this loop indefinitely. The syntax table entry for `cycle` is

$$\texttt{(cycle } expr_1\ expr_2\ \texttt{...)} \equiv \texttt{(cycle-proc (freeze } expr1\ expr_2\ \texttt{...))}$$

Program 14.11 `cycle-proc`

```
(define cycle-proc
  (lambda (th)
    (letrec ((loop (lambda ()
                     (thaw th)
                     (loop))))
      (loop))))
```

where `cycle-proc` is defined in Program 14.11. In Chapter 17, we shall encounter several uses of `cycle-proc`.

The last special form that we discuss has keyword `or`. First why must `or` be a macro instead of a procedure? When we write `(or` e_1 e_2`)`, the first subexpression e_1 is evaluated, and if it is true, then its value is returned. If e_1 is false, only then is e_2 evaluated. If `or` were a procedure, both subexpressions would be evaluated before they are passed to `or`. The fact that the second subexpression is not evaluated unless the first is false allows us to include the following expression in a program:

```
(or (zero? x) (> (/ 10 x) 2))
```

and be sure that division by zero does not occur because the second subexpression is not evaluated if `x` is zero. Thus we want `or` to be a macro that can take any number of subexpressions, including no subexpressions. If `or` is called with no subexpressions, it returns false. Having taken care of the case of no subexpressions, we consider the following syntax table entry for `or` with several subexpressions:

`(or` e_1 e_2 `...)` $\equiv$ `(if` e_1 e_1 `(or` e_2 `...))`

This works because `if` first evaluates e_1 and if it is true, it returns the value of e_1 in the consequent. If e_1 is false, it skips to the alternative and returns the "recursive" value obtained for the alternative. This looks like recursion, but we must remember that these `or` expressions are not being evaluated. Rather they are macrocode, which is being transformed into `if` expressions that are the macroexpansions. We have treated the case of `(or` e`)`, which should have the same value as e, because using the syntax table entry, `(or` e`)` expands to `(if` e e `(or))` and `(or)` expands to `#f`.

We could use the above macroexpansion for `or`, but it does not work efficiently since if e_1 is true, it must be evaluated a second time in the consequent.

If e_1 includes some side effects, these would be done twice instead of once, and that is generally incorrect. We can avoid this double evaluation by including a let expression in the macroexpansion:

$$(\texttt{or}\ e_1\ e_2\ \ldots) \equiv (\texttt{let}\ ((\texttt{val}\ e_1))\ (\texttt{if val val}\ (\texttt{or}\ e_2\ \ldots)))$$

Once again, if we declare the macro according to this expansion pattern, it will work the way we want *almost* all of the time. But an unwanted behavior, known as *capturing*, can occur, as the following example illustrates. Suppose the macro or has been declared according to the above pattern. We then use it in the following program:

```
(let ((val #t))
  (or #f val))
```

We expect this to return #t. However, when the program is entered, the or expression is expanded into

```
(let ((val #f))
  (if val val val))
```

and the value returned is #f because the last val has been *captured* within the scope of the nearest binding, and unfortunately the variable val was also used in the let expression in the declaration of the macro or. There are several ways of avoiding this capturing. We shall make use of the fact that when a frozen entity is thawed, it is evaluated in the environment that was in effect when the entity was frozen. We first define a procedure, called or-proc, which takes a list of thunks as its operand. Then to declare the macro or, we freeze the operands and pass them to the procedure or-proc. Here is the definition of or-proc:

Program 14.12 or-proc

```
(define or-proc
  (lambda (th-list)
    (cond
      ((null? th-list) #f)
      (else (let ((v (thaw (car th-list))))
              (if v v (or-proc (cdr th-list))))))))
```

In this version, the thunks are not evaluated until they are thawed, so only one of the thunks is evaluated at a time until a true value is obtained. The rest remain unevaluated.

With this definition of or-proc, the syntax table entry for the macro or becomes:

(or *e* ...) ≡ (or-proc (list (freeze *e*) ...))

How are the cases of zero expressions and one expression handled by this entry? Now or-transformer can be defined and or can be declared:

Program 14.13 or

```
(define or-transformer
  (lambda (code)
    (list 'or-proc
          (cons 'list
                (map (lambda (e) (list 'freeze e))
                     (cdr code))))))

(macro or or-transformer)
```

We can also use extend-syntax to declare the macro or based on the above syntax table entry. We have:

Program 14.14 or

```
(extend-syntax (or)
  ((or e ...) (or-proc (list (freeze e) ...))))
```

Several more special forms are developed in the exercises. The ability to write your own special forms in Scheme is a powerful tool that can be used to make programs more readable. Most important, it allows you to build your own textual abstractions. In the next chapter, we shall make use of the special form delay to develop the idea of streams or "infinite lists."

Exercises

Exercise 14.1
What is the output of

```
(freeze-transformer '(freeze (cons 'a '(b c))))
```

What is the output of

```
(let-transformer '(let ((a 5) (b 2)) (* a b)))
```

What general statement can you conclude from these examples concerning the output when a transform procedure is applied to the quoted macrocode? Some implementations of Scheme have a procedure called **expand**, which converts the quoted macrocode into its macroexpansion.

Exercise 14.2
Declare the letrec macro using `extend-syntax` without using `let` in its macroexpansion.

Exercise 14.3
Consider the declaration of the macro `or`, below. Does this declaration suffer the variable capturing that we were able to avoid using `or-proc` and a list of thunks?

```
(extend-syntax (or)
  ((or) #f)
  ((or e) e)
  ((or e1 e2 ...) (let ((val e1) (th (freeze (or e2 ...))))
                    (if val val (thaw th)))))
```

Exercise 14.4: **and**
Declare a macro with keyword **and**, which, like **or**, may take any number of subexpressions. If called with no subexpressions, it is true. If all of its subexpressions are true, it evaluates to the last one; otherwise it is false. Test your macro on:

```
(and)
(and #t)
(and #f)
(and #t #t #t)
(and #t #t #f)
```

Note that the capturing problem need not arise in declaring **and**.

Exercise 14.5
The let expression

```
(let ((x 3))
  (let ((x 10) (y x))
    y))
```

evaluates to 3 because the x in the binding pair (y x) must look up its value in an environment other than the local environment of the expression

```
(let ((x 10) (y x))
  y)
```

The value 3 is found since that let expression is nested within the let expression with binding pair (x 3). If we had wanted the x in (y x) to refer to the x in (x 10), we would have had to put the (y x) in another nested let expression, as follows:

```
(let ((x 3))
  (let ((x 10))
    (let ((y x))
      y)))        ⟹ 10
```

In general, in the let expression

(let ((var_1 val_1) (var_2 val_2) (var_3 val_3)) $expr_1$ $expr_2$ ...)

instances of var_1 in val_2 and instances of var_1 or var_2 in val_3 cannot refer to var_1 or var_2 in this let expression but must find their values in a nonlocal environment. However, if we were to write nested let expressions, such as

(let ((var_1 val_1))
  (let ((var_2 val_2))
    (let ((var_3 val_3))
      $expr_1$ $expr_2$ ...)))

then instances of var_1 in val_2 can refer to the var_1 in the first binding pair, and instances of var_1 or var_2 in val_3 can refer to the var_1 or var_2 of the preceding two binding pairs. We used the Scheme special form **let*** in Section 10.2.5. It has a syntax similar to that of **let** but behaves as though the successive binding pairs are in nested let expressions. In fact, if there is only one such binding pair, then **let*** is the same as **let**, so that

```
(let* ((var val)) expr1 expr2 ...) ≡ (let ((var val)) expr1 expr2 ...)
```

and if there is more than one such binding pair,

```
        (let* ((var1 val1) (var2 val2) ...) expr1 expr2 ...)
                                 ≡
(let ((var1 val1)) (let* ((var2 val2) ...) expr1 expr2 ...))
```

Write `let*-transformer` or use `extend-syntax` to declare `let*`. Test it on the following:

```
(let* ((a 1) (b (+ a 2)) (c (* a b))) (+ a (- c b)))
```

Exercise 14.6

The procedure `let-transformer` is correct only if the user obeys `let`'s syntax. The special form `let` expects a list of $n+2$ elements. The first must be the symbol `let`; the second must be a list of pairs where each pair is a list of two elements, in which the first element must be a symbol. The remaining $n > 0$ elements can be arbitrary expressions. Here are some incorrect examples:

```
(let ((x 3) (y 4)))
(let ((3 3) (y 4)) (* x y))
(let ((x 3) (y 4 5)) (* x y))
(let x 3 (* x y))
(let (("x" 3) (y 4)) (* "x" y))
```

Rewrite `let-transformer` so that reasonable error indications, such as those shown below, are given to the user of `let`. Test these examples by invoking `let-transformer` on the individual lists in question:

```
(let-transformer '(let ((x 3) (y 4)))) ⟹
    Error: illegal let expression: (let ((x 3) (y 4)))

(let-transformer '(let ((3 3) (y 4)) (* x y))) ⟹
    Error: illegal let expression: (let ((3 3) (y 4)) (* x y))

(let-transformer '(let ((x 3) (y 4 5)) (* x y))) ⟹
    Error: illegal let expression: (let ((x 3) (y 4 5)) (* x y))

(let-transformer '(let x 3 (* x y))) ⟹
    Error: illegal let expression: (let x 3 (* x y))

(let-transformer '(let (("x" 3) (y 4)) (* "x" y))) ⟹
    Error: illegal let expression: (let (("x" 3) (y 4)) (* "x" y))
```

Exercise 14.7

The error information from the previous exercise does not pinpoint exactly where the error occurred. Redesign the information displayed so that you can better determine where the error occurred.

Exercise 14.8: named `let`

The macro `let` declared above did not include the case of the named `let`. The *named*-`let` has the syntax table entry:

```
(let name ((var val) ...)
  expr1 expr2 ...)
            ≡
((letrec
   ((name (lambda (var ...)
            expr1 expr2 ...)))
   name)
 val ...)
```

Define `let-transformer` or declare `let` using `extend-syntax` to include both cases, the ordinary `let` and the *named*-`let`. Do Exercise 5.7 using *named*-`let`.

Exercise 14.9: `cycle`

Define `cycle-transformer` or declare `cycle` using `extend-syntax`.

Exercise 14.10: `while`

The special form `while` is a control structure common to many programming languages. In `while`, an expression is evaluated repeatedly as long as a given condition is true. We can effect the behavior of a while expression as illustrated by the following program, which sums the numbers from 1 to 100:

```
(let ((n 100) (sum 0))
  (letrec ((loop (lambda ()
                   (if (positive? n)
                       (begin
                         (set! sum (+ sum n))
                         (set! n (sub1 n))
                         (loop))))))
    (loop)
    sum))
```

We would like to introduce the special form `while`, which allows us to write the above program as:

```
(let ((n 100) (sum 0))
  (while (positive? n)
         (set! sum (+ sum n))
         (set! n (sub1 n)))
  sum)
```

Thus **while** has the syntax table entry:

```
(while test expr1 expr2 ...)
          ≡
(letrec
  ((loop (lambda ()
           (if test (begin expr1 expr2 ... (loop))))))
  (loop))
```

Define **while-transformer** or declare **while** using **extend-syntax**. You must take into account the variable capturing that is caused when the variable **loop** occurs free in *test* or *expr* ... in the macroexpansion. The syntax table entry for **while** must then be modified to be of the form

```
(while test expr1 expr2 ...)
          ≡
(while-proc (freeze test) (freeze expr1 expr2 ...))
```

where **while-proc** is defined in Program 11.8. Test **while** on the above program.

Exercise 14.11: **repeat**

The special form **repeat** takes two expressions. It executes the first expression. Then it executes the second expression. If that returns true, the expression terminates with an unspecified value. If not, it repeats in much the same way as **while** from the previous exercise. Define **repeat-transformer** or declare **repeat** using **extend-syntax** by including **while** in its macroexpansion. Then redo the exercise without using **while**. Finally, write an expression using **repeat** that models the test program of the previous exercise.

Exercise 14.12: **for**

Write a special form that models the behavior of for expressions. Such expressions have the following syntax:

```
(for var initial step test expr1 expr2 ...)
```

The for expression is used for modeling iteration. The variable *var* is initialized to *initial*. Then the *test* is evaluated to determine whether it should terminate. If *test* is true, it does terminate. If *test* is false, then *expr* ... is evaluated. Finally, *var* is reset to the evaluation of *step*, and the process repeats.

Define `for-transformer` or declare `for` using `extend-syntax` given the syntax table entry below.

```
(for var initial step test expr1 expr2 ...)
                    ≡
(let ((var initial))
  (let ((step-thunk (freeze step))
        (test-thunk (freeze test))
        (body-thunk (freeze expr1 expr2 ...)))
    (while (not (thaw test-thunk))
           (thaw body-thunk)
           (set! var (thaw step-thunk)))))
```

This solution is subtle because each of *step*, *test*, and $expr_1$ $expr_2$ ... will be using *var*. For example, a typical use of for expressions is to add the elements of a vector:

```
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v))
          (sum 0))
      (for i 0 (add1 i) (= i n) (set! sum (+ sum (vector-ref v i))))
      sum)))
```

Exercise 14.13: `do`

The special form do has the syntax table entry:

```
(do ((var initial step) ...)
    (test exit1 exit2 ...)
  expr1 expr2 ...)
              ≡
((letrec
   ((loop (lambda (var ...)
            (cond
              (test exit1 exit2 ...)
              (else (begin expr1 expr2 ...)
                      (loop step ...))))))
   loop)
 initial ...)
```

The variable `loop` must not be among *var* ... and it must not be free in *test*, *exit*$_1$ *exit*$_2$ ..., *expr*$_1$ *expr*$_2$ ..., and *step* ... Redesign `for`'s syntax table entry using `do`. (See the previous exercise.)

Exercise 14.14: `begin0`
Consider the following syntax table entry for `begin0`:

```
(begin0 e)              ≡  e
(begin0 e1 e2 e3 ...)   ≡  (begin0-proc e1 (freeze e2 e3 ...))
```

`begin0` evaluates its subexpressions in order and returns the result of evaluating the first one. Define the procedure `begin0-proc`, which always takes exactly two arguments. Why is the syntax table entry

```
(begin0 expr1 expr2 ...) ≡ ((lambda args (car args)) expr1 expr2 ...)
```

incorrect? (*Hint*: Read the specification carefully. What can we say about the order of evaluation of operands?) Test `begin0-proc` by defining `begin0-transformer` or declaring `begin0` using `extend-syntax`.

Exercise 14.15: `begin`
Define `begin-transformer` or declare `begin` using `extend-syntax` without using `freeze` or the implied `begin` associated with lambda expressions.

Exercise 14.16: `cond`
Consider cond expressions that are restricted to including at least one expression following each test in every clause and where the last clause must be an else clause. They can be transformed into nested if expressions using the following two-patterned syntax table entry:

```
(cond (else e1 e2 ...)) ≡ (begin e1 e2 ...)

(cond (test e1 e2 ...) clauses ...)
≡
(if test (begin e1 e2 ...) (cond clauses ...))
```

Redefine `member-trace` and `factorial` below, using just the syntax table entry for cond expressions.

```
(define member-trace
  (lambda (item ls)
    (cond
      ((null? ls) (writeln "no") #f)
      ((equal? (car ls) item) (writeln "yes") #t)
      (else (writeln "maybe") (member-trace item (cdr ls))))))

(define factorial
  (lambda (n)
    (cond
      ((zero? n) 1)
      (else (* n (factorial (sub1 n)))))))
```

Exercise 14.17: **cond**

In order to declare the simplified **cond** with **extend-syntax**, the symbol **else** must be included in the first operand to **extend-syntax**. That is because **extend-syntax** has to be told what symbols it is supposed to be treating literally. In most cases, it is just the special form name, but for **cond** and **case**, it includes the symbol **else**. Fill in the rest of the declaration of **cond** below. (See the previous exercise.)

```
(extend-syntax (cond else)
  ((cond (else e1 e2 ...)) ?___________ )
  ((cond (test e1 e2 ...) clauses ...) ?___________ ))
```

Exercise 14.18: **variable-case**

Consider a variant of the case expression called **variable-case**. This expression is similar to **case**, except that instead of allowing its first operand to be any expression, it is limited to being a variable. Thus, using case we can write:

```
(case (remainder 35 10)
  ((2 4 6 8) (writeln "even") (remainder 35 10))
  ((1 3 5 7 9) (writeln "odd") (remainder 35 10))
  (else (writeln "zero") (remainder 35 10)))
```

but with **variable-case** we must write:

```
(let ((x (remainder 35 10)))
  (variable-case x
    ((2 4 6 8) (writeln "even") x)
    ((1 3 5 7 9) (writeln "odd") x)
    (else (writeln "zero") x)))
```

Complete the declaration of `variable-case` presented below, and then define `variable-case-transformer`. Explain why `keys` has been transformed into `(quote keys)`. *Hint*: Remember that `keys` will be a list.

```
(extend-syntax (variable-case else)
  ((variable-case var (else e1 e2 ...)) ?__________ )
  ((variable-case var (keys e1 e2 ...) clauses ...)
   (if (memv var (quote keys))
       (begin e1 e2 ...)
       (variable-case var clauses ...))))
```

Exercise 14.19

If we did not have `variable-case` from the previous exercise, then the case example above would require an additional evaluation of `(remainder 35 10)`. Instead, we can choose a variable, say `target`, that will always hold the value of the first operand of the most deeply nested case expression. Given this constraint, declare this variant of `case` using `extend-syntax`. *Hint*: If you use `variable-case`, you need only one rule for its syntax table entry, but remember to include `else` in the list of symbols to be taken literally. Test your program with the following case expression:

```
(case (remainder 35 10)
  ((2 4 6 8) (writeln "even") target)
  ((1 3 5 7 9) (writeln "odd") target)
  (else (writeln "zero") target))
```

Exercise 14.20: `object-maker`

In Chapter 12 we presented a set of object-oriented programs that had a particular pattern of use. For example, each object maker includes

```
(lambda msg (case (1st msg) ...))
```

Design a special form `object-maker` that abstracts this pattern of use. Are there other patterns of use with object makers that can be abstracted?
