desired behavior, you need some way of storing *the value of a single toss, so that you may then* refer *to it as often as you like. In short, you need a* `let` *special form, as illustrated below:*

```
> (let ((toss (toss-die)))
    (printf "My toss: ~s~%" toss)
    (printf "The square of my toss: ~s~%" (* toss toss))
    (* toss toss toss))
My toss: 4
The square of my toss: 16
64
> toss
ERROR:  reference to undefined identifier: toss
```

In this example, the `let` *special form creates a local variable named* `toss` *whose value is the result of randomly tossing a six-sided die. The expressions in the body of the* `let` *can then refer to* `toss`*—and thereby gain access to that stored value—as many times as needed. However, the local environment only exists while the* `let` *special form is being evaluated. Once the evaluation of the* `let` *is completed, its local environment evaporates. It is for this reason that any later attempt to evaluate* `toss` *will cause DrScheme to report an error, as shown above. (This example assumes that there is no entry for* `toss` *in the Global Environment.)*

15.3 Nested `let` Expressions and Nested Environments

When a `let` special form is evaluated with respect to the Global Environment, it creates a local environment, $\mathcal{E}_1$, that is nested inside the Global Environment. For convenience, we can represent this by writing $\mathcal{E}_1 \subset \mathcal{E}_0$, where $\mathcal{E}_0$ represents the Global Environment. Any expression in the body of that `let` may refer to any variable in that local environment $\mathcal{E}_1$, as well as any variable in the Global Environment $\mathcal{E}_0$, with the proviso that the local environment has priority. For this reason, the following `let` expression evaluates to `25` (i.e., 5 times 5) because the value of the symbol `x` is fetched from the local environment, not the Global Environment.

```
> (define x 100)
> (let ((x 5))
    (* x x))
25
```

Note that the value of the *asterisk* symbol is fetched from the Global Environment, because there is no entry in the local environment for that symbol. Note, too, that there is no way that an expression in the body of this `let` could refer to the global variable `x`, because the existence of the local variable `x` effectively blocks access to the globally defined `x`.

Continuing in this way, a `let` nested inside another `let` (i.e., a `let` expression that appears in the body of another `let`) creates a new local environment, $\mathcal{E}_2$, where $\mathcal{E}_2 \subset \mathcal{E}_1 \subset \mathcal{E}_0$. Thus, any expression in the body of that `let` is evaluated with respect to the environment $\mathcal{E}_2$, which implies that $\mathcal{E}_2$ has the highest priority, $\mathcal{E}_1$ has the next highest priority and, as always, the Global Environment $\mathcal{E}_0$ has the lowest priority. The following example demonstrates that this is the case.

Example 15.3.1

```
> (define x 100)
> (let ((x 5))
    (let ((x (* x x)))
      (printf "Innermost x: ~s~%" x))
    (printf "Middle x: ~s~%" x))
```

```
Innermost x: 25
Middle x: 5
> x
100
```

The `let` *special form creates a local variable* `x`*, in the environment* $\mathcal{E}_1$*, whose value is* `5`*. In the body of that* `let`*, the next* `let` *creates a* different *local variable, in the environment* $\mathcal{E}_2$*, that also happens to be called* `x`*. In the environment* $\mathcal{E}_2$*, the value of* `x` *is* `25` *(i.e.,* 5 *times* 5*). Note that its value is computed* before *the creation of the environment* $\mathcal{E}_2$*; its value is computed with respect to the environment* $\mathcal{E}_1$*. The first* `printf` *expression is evaluated with respect to the innermost environment* $\mathcal{E}_2$*, where* `x` *has the value* `25`*. Since there is no entry for the* `printf` *symbol in* $\mathcal{E}_2$ *or* $\mathcal{E}_1$*, its value is obtained from the Global Environment* $\mathcal{E}_0$*. The second* `printf` *expression is evaluated with respect to the environment* $\mathcal{E}_1$*, where* `x` *has the value* `5`*.*

It is important to point out that since the environment $\mathcal{E}_2$ *defines a local variable named* `x`*, then any expression evaluated with respect to that environment cannot access any other variable named* `x` *that might exist in any of the parent environments (e.g., the variable named* `x` *in the environment* $\mathcal{E}_1$*, or the variable named* `x` *in the Global Environment). In the same way, if some local environment has a variable named* `remainder`*, then any expression being evaluated with respect to that local environment cannot access the built-in* `remainder` *function, because the local variable named* `remainder` *would have priority over the globally defined* `remainder` *function.*

In general, a `let` expression that is evaluated with respect to some parent environment $\mathcal{E}$ creates a new local environment $\mathcal{E}'$ that is nested inside $\mathcal{E}$ (i.e., $\mathcal{E}' \subset \mathcal{E}$). To evaluate a symbol s with respect to the new environment $\mathcal{E}'$, involves the following *recursive* process:

(Base Case) If there is an entry in the environment $\mathcal{E}'$ that pairs s with a value v, then s evaluates to v in $\mathcal{E}'$.

(Recursive Case) Otherwise, the value for s is obtained by evaluating s in the parent environment $\mathcal{E}$.

Note that this process is recursive because if the parent environment does not have an entry for s, then s will have be be evaluated with respect to its parent environment, and so on, until, eventually, an ancestor environment is reached that has an entry for s. Note that if this process goes all the way to the Global Environment without finding any entry for s in any environment along the way (including the Global Environment), then evaluating s in the environment $\mathcal{E}'$ is undefined.

We can describe this process as follows. Since each environment other than the Global Environment is nested inside its parent environment, each environment $\mathcal{E}_n$ determines a chain of ancestor environments of the form, $\mathcal{E}_n \subset \mathcal{E}_{n-1} \subset \ldots \subset \mathcal{E}_2 \subset \mathcal{E}_1 \subset \mathcal{E}_0$, where $\mathcal{E}_0$ is the Global Environment. When a symbol is being evaluated with respect to the environment $\mathcal{E}_n$, the environment $\mathcal{E}_n$ has the highest priority and the Global Environment has the lowest priority. When evaluating a symbol s in the environment $\mathcal{E}_n$, the environments are checked, in order, from $\mathcal{E}_n$ to $\mathcal{E}_0$, until one is found that has an entry for s. The value for s in that entry will be the result of evaluating s in $\mathcal{E}_n$.

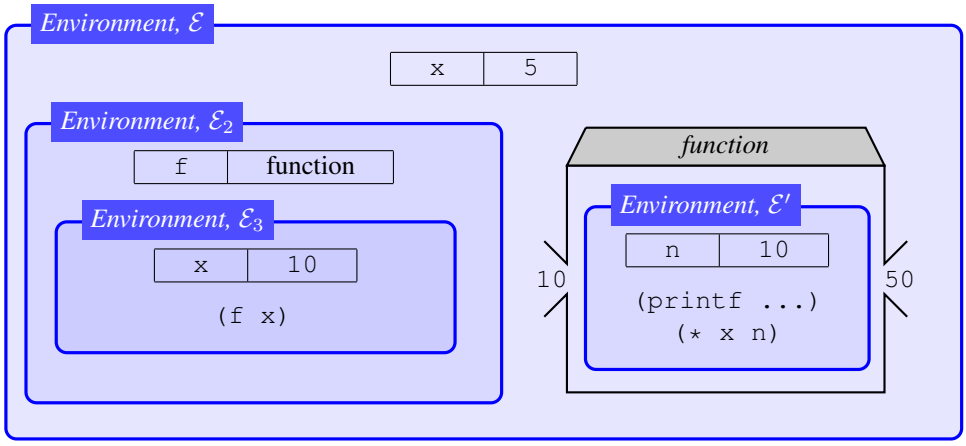
Similar considerations apply to the local environment that is automatically created when a `lambda` function is applied to inputs. The main difference is:

- ⋆ When a function f is *applied* to inputs, the local environment within which the *body* of that function is *evaluated* is nested inside the environment within which that function was *created*—regardless of the environment within which the function call expression is being evaluated!

In other words, if the function f was created by evaluating a `lambda` special form within an environment $\mathcal{E}$, then, when that function is applied to inputs, the local environment $\mathcal{E}'$ within which the body of f is evaluated is necessarily nested within the parent environment $\mathcal{E}$: $\mathcal{E}' \subset \mathcal{E}$.

Example 15.3.2

```
> (let ((x 5))
    (let ((f (lambda (n)
               (printf "Inside function body: x = ~s~%" x)
               (* x n))))
      (let ((x 10))
        (f x))))
Inside function body: x = 5
50
```



As illustrated above, evaluating the first `let` *creates a local environment* $\mathcal{E}$ *in which* `x` *has the value* `5`*. Next, the second* `let` *expression is evaluated with respect to the environment* $\mathcal{E}$*. According to the semantics for* `let` *expressions, the* value *for* `f` *is obtained by evaluating the appropriate expression with respect to the parent environment* $\mathcal{E}$*. In this case, that means evaluating the given* `lambda` *expression with respect to* $\mathcal{E}$*. This function, which was created within the environment* $\mathcal{E}$*, becomes the value for the local variable* `f` *in the new local environment* $\mathcal{E}_2$*, nested inside* $\mathcal{E}$*. Next, the third* `let` *expression creates a local environment* $\mathcal{E}_3$ *in which a new local variable* `x` *has the value* `10`*. Finally, the expression* `(f x)` *is evaluated with respect to the environment* $\mathcal{E}_3$ *using the Default Rule. In the environment* $\mathcal{E}_3$*, the symbol* `f` *evaluates to the previously created* `lambda` *function, and* `x` *evaluates to* `10`*; then that* `lambda` *function is applied to the input* `10`*. The key point is the following:* because the `lambda` function was created in the environment $\mathcal{E}$, the local environment $\mathcal{E}'$ within which the body of that function will be evaluated must be nested within $\mathcal{E}$ (i.e., $\mathcal{E}' \subset \mathcal{E}$)—regardless of the fact that the function call expression was evaluated within the environment $\mathcal{E}_3$*. Therefore, the symbol* `x` *that appears in the body of that function evaluates to* `5`*, courtesy of the parent environment* $\mathcal{E}$*,* `n` *evaluates to* `10`*, and* `(* x n)` *evaluates to* `50`*.*

15.4 Deriving the `let` Special Form from the `lambda` Special Form

If you're thinking that the evaluation of a `let` special form seems awfully close to the evaluation of a function call, you're right. In fact, each `let` special form expression is simply a convenient abbreviation for an expression in which a *lambda function* is applied to some input values. Before going into all the details, we give some examples illustrating the equivalence of expressions involving `let` and `lambda`.

Example 15.4.1

The following Interactions Window session shows the evaluation of a `let` *expression, followed by the evaluation of an equivalent expression involving the application of a* lambda function *to some inputs.*

```
> (let ((x (+ 2 3))
        (y (* 3 4)))
    (printf "x: ~s, y: ~s~%" x y)
    (+ x y))
x: 5, y: 12
17
> ((lambda (x y)
     (printf "x: ~s, y: ~s~%" x y)
     (+ x y))
   (+ 2 3)
   (* 3 4))
x: 5, y: 12
17
```

$\Longrightarrow$ *The semantics for the evaluation of the first expression* is identical to *the semantics for the evaluation of the second expression!*

In particular, for the `let` *expression, a local environment is set up in which the symbol* `x` *is associated with the value* `5` *and the symbol* `y` *is associated with the value* `12`*. After that, the two expressions in the body of the* `let` *are evaluated with respect to that local environment yielding some side-effect printing and an output value of* `17`*.*

The evaluation of the second expression is governed by the Default Rule for evaluating non-empty lists. The first entry in the list is a `lambda` *expression. It evaluates to a function. The other entries,* `(+ 2 3)` *and* `(* 3 4)`*, evaluate to the numbers* `5` *and* `12`*, respectively. When that function is applied to those inputs, a local environment is set up in which* `x` *and* `y` *are associated with the values* `5` *and* `12`*, respectively. Then the body of the* `lambda` *is evaluated, yielding side-effect printing and the output value* `17`*.*

Example 15.4.2

The following Interactions Window session first creates a global variable, `z`*. It then evaluates a* `let` *expression and an equivalent expression involving the application of a* `lambda` *function.*

```
> (define z 1000)
> (let ((x 3)
        (y 4))
    (* x y z))
12000
> ((lambda (x y)
     (* x y z))
   3
   4)
12000
```

Once again, the evaluation of the two expressions is the same. In particular, each involves a local environment containing entries for `x` *and* `y`*, with the respective values* `3` *and* `4`*. In addition, each involves the evaluation of the expression* `(* x y z)` *with respect to that local environment. Notice that in each case, the values for* `x` *and* `y` *are drawn from the local environment, whereas the value for* `z` *is drawn from the Global Environment. In each case, the value of the entire expression is* `12000`*.*

In general, a `let` expression of the form,

```
(let ((var1 val1)
      (var2 val2)
       ...
      (varn valn))
  expr1
  expr2
   ...
  exprk)
```

is equivalent to the following expression involving the application of a `lambda` function:

```
((lambda (var1 ... varn)
   expr1
   expr2
    ...
   exprk)
  val1 ... valn)
```

You should convince yourself that the local environments that are created in response to evaluating these two expressions are equivalent.

- ⋆ The reason we have `let` expressions is that they have a friendlier syntax for the cases where you want to create a local environment and then evaluate some expressions with respect to that local environment.

15.5 The `let*` Special Form

The syntax of the `let*` special form is nearly identical to that of the `let` special form. (The only difference is the presence of the ⋆ in `let*`.) However, the semantics is substantially different. In particular, the local environment is populated *incrementally,* as each *var*/*val* pair is processed. This difference allows a certain kind of incremental computation that turns out to be quite useful. When a `let` special form is evaluated, each val_i is evaluated with respect to the parent environment and, thus, none of the val_i expressions can depend on any of the variables in the nascent local environment. In contrast, when a `let*` special form is evaluated, each val_i is evaluated with respect to the *portion* of the local environment that has been created so far. As a result, the expression val_i *may* depend on the values of the local variables $var_1, \ldots, var_{i-1}$ that precede it in the `let*` expression.

15.5.1 The Syntax of the `let*` Special Form

Each `let*` expression has the following form:

```
(let* ((var1 val1)
       (var2 val2)
        ...
       (varn valn))
  expr1
  expr2
   ...
  exprk)
```

You'll notice that the only difference is the asterisk in the name of the special form: `let*` instead of `let`.

15.5.2 The Semantics of the `let*` Special Form

A `let*` special form is evaluated as follows:

- An empty local environment is created.
- Each *var*/*val* pair is processed, in turn. In particular, an entry is created in the local environment that associates the value of val_i with the symbol var_i.

$\Longrightarrow$ Crucially, the i^{th} entry in the local environment is created *before* the $(i+1)^{\text{st}}$ value is computed. Thus, the expression, val_{i+1}, can refer to *any* of the *preceding* symbols, $var_1, \ldots, var_i$.

- Then the expressions in the body of the `let*` are evaluated, in turn.
- The value of the last expression in the body of the `let*` serves as the value of the entire `let*` expression.

Example 15.5.1

The following Interactions Window session demonstrates the kind of incremental computation that is characteristic of a `let*` *special form, but that is not possible with a (single)* `let` *special form:*

```
> (let* ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
    (printf "x: ~s, y: ~s, z: ~s, w: ~s~%" x y z w)
    (+ x y z w))
x: 4, y: 6, z: 24, w: 34
68
```

Notice that the expression, `(+ x 2)`*, that is used to compute the value for* `y` *refers to the local variable* `x`*. Similarly, the expression,* `(* x y)`*, that is used to compute the value for* `z` *refers to both* `x` *and* `y`*. Finally, the expression,* `(+ x y z)`*, that is used to compute the value for* `w` *refers to* `x`*,* `y` *and* `z`*. Trying to do this with a* `let` *expression causes DrScheme to complain.*

```
> (let ((x 4)
        (y (+ x 2))
        (z (* x y))
        (w (+ x y z)))
    (printf "x: ~s, y: ~s, z: ~s, w: ~s~%" x y z w)
    (+ x y z w))
... reference to undefined identifier: x
```

The reason is due to the difference in the way `let` *and* `let*` *expressions are evaluated (i.e., their semantics). In a* `let` *expression,* all *of the value expressions are evaluated* first, before *any entries are created in the local environment. Thus, none of the value expressions in a* `let` *can refer to any of the local variables being defined. In contrast, in a* `let*` *expression, the evaluation of the value expressions is interleaved with the creation of the entries in the local environment. Thus, each value expression can refer to symbols that* precede *it in the* `let*` *expression.*

15.5.3 Deriving a Single `let*` Expression from Nested `let` Expressions

In general, a `let*` expression of the form,

```
(let* ((var1 val1)
       (var2 val2)
        ...
       (varn valn))
  expr1
  expr2
   ...
  exprk)
```

is equivalent to n *nested* `let` expressions:

```
(let ((var1 val1))
  (let ((var2 val2))
    ...
      (let ((varn valn))
        expr1
        expr2
        ...
        exprk) ...))
```

So, a `let*` expression with n variable/value pairs effectively creates a sequence of n new local environments, where each new environment is nested inside its predecessor.

The following example demonstrates the equivalence.

Example 15.5.2

The following Interactions Window session evaluates a `let*` *expression and the equivalent* nested `let` *expression:*

```
> (let* ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
    (printf "x: ~s, y: ~s, z: ~s, w: ~s~%" x y z w)
    (+ x y z w))
x: 4, y: 6, z: 24, w: 34
68
> (let ((x 4))
    (let ((y (+ x 2)))
      (let ((z (* x y)))
        (let ((w (+ x y z)))
          (printf "x: ~s, y: ~s, z: ~s, w: ~s~%" x y z w)
          (+ x y z w)))))
x: 4, y: 6, z: 24, w: 34
68
```

Notice that the outermost `let` *expression (i.e., the one that specifies the local variable* `x`*) has a body that consists of a single* `let` *expression (i.e., the one that specifies the local variable* `y`*). Because the* `let` *expression for* `y` *is evaluated with respect to the local environment containing an entry for* `x`*, it is okay for the value expression,* `(+ x 2)`*, to refer to* `x`*. Similar remarks apply to the remaining variables.*

In general, `let*` provides a simpler syntax than the equivalent set of nested `let` expressions. Thus, if you ever need to do incremental computations where the value of each local variable depends of the values of the preceding local variables, then you should consider using `let*`.

15.6 The `letrec` Special Form

The `letrec` special form is provided to enable the specification of local recursive functions, something that cannot be done by `let` or `let*`. The specification of a local recursive function within a `letrec` special form is quite similar to the specification of a global recursive function within a `define` special form; however, the syntax of a `letrec` expression is much closer to that of `let` and `let*`. A common use of `letrec` is to embed an accumulator-based, tail-recursive helper function *within* the body of its wrapper function. In this way, the existence of the helper function (and access to it) can be hidden from the general programming public. As usual, in such scenarios, the wrapper function takes care of supplying appropriate inputs to the helper function, freeing the user to think about other things.

15.6.1 The Syntax of the `letrec` Special Form

The syntax of the `letrec` special form is identical to that of the `let` and `let*` special forms, except that the keyword is `letrec` instead of `let` or `let*`.

15.6.2 The Semantics of the `letrec` Special Form

In sharp contrast to how the `let` and `let*` special forms are evaluated, the evaluation of a `letrec` special form begins by creating the *entire* local environment, complete with entries for *all* of the local variables, before evaluating *any* of the value expressions. Because none of the value expressions have yet been evaluated, each local variable is initially given the dummy value, `#<undefined>`. However, since all of the local variables have corresponding entries in the local environment before any of the value expressions are evaluated, each value expression can refer to *any* or *all* of the local variables, whether they have values or not!

Example 15.6.1

The following interactions demonstrate that the `letrec` *special form sets up its local environment* before *evaluating any of the value expressions. Because the* `let` *and* `let*` *special forms do not do this, the corresponding instances generate errors.*

```
> (let ((x y)          ⟵ Evaluating y before an entry for y exists...
       (y x))
    (printf "x:~s, y:~s~%" x y))
ERROR:  reference to undefined identifier: y
> (let* ((x y)         ⟵ Ditto
        (y x))
    (printf "x:~s, y:~s~%" x y))
ERROR:  reference to undefined identifier: y
> (letrec ((x y)       ⟵ y has the value #<undefined>, so all is well
           (y x))      ⟵ x has the value #<undefined>, so all is well
    (printf "x:~s, y:~s~%" x y))
x:#<undefined>, y:#<undefined>
```

The preceding example is illustrative, but it ignores the primary purpose of the `letrec` special form: to create local recursive functions, similar to how the `define` special form can be used to create global recursive functions. For example, a `letrec` can be used to create a local variable `funky` whose value is a function whose body includes a recursive function call of the function named `funky`.

Example 15.6.2: Using `letrec` to create a local recursive function

The following interactions demonstrate that `letrec` *can be used to define a local recursive function, whereas* `let` *and* `let*` *cannot.*

```
> (let ((factyOne (lambda (n)
                    (if (<= n 1)
                        1
                        (* n (factyOne (- n 1)))))))
    (printf "No error up to this point, but ...~%")
    (factyOne 4))
No error up to this point, but ...
ERROR: reference to undefined identifier: factyOne
> (let* ((factyTwo (lambda (n)
                     (if (<= n 1)
                         1
                         (* n (factyTwo (- n 1)))))))
    (printf "No error up to this point, but ...~%")
    (factyTwo 4))
No error up to this point, but ...
ERROR: reference to undefined identifier: factyTwo
> (letrec ((factyThree (lambda (n)
                         (if (<= n 1)
                             1
                             ;; No problems here! :)
                             (* n (factyThree (- n 1)))))))
    (factyThree 4))
24
```

In the first example, the `let` *expression creates a local environment* $\mathcal{E}_1$ *that is nested inside the Global Environment. According to the semantics for a* `let`*, the* value *for its variable* `factyOne` *is evaluated with respect to the parent environment—in this case, the Global Environment. The result is a* `lambda` *function created with respect to the Global Environment. As the side-effect printing indicates, the evaluation of that* `lambda` *expression does not cause an error—because the expressions in the body are not evaluated when the function is created. However, attempting to apply the function to some numerical input requires evaluating the expressions in the function body—with respect to an automatically-created local environment* $\mathcal{E}_f$ *that is nested inside the Global Environment (i.e., the environment within which the function was created). Because there is no entry for* `factyOne` *in the Global Environment, this leads to an error.*

Similar remarks apply to the `let*` *expression because a* `let*` *that includes only one variable/value pair is equivalent to a* `let`*. However, for the* `letrec` *expression, there are no problems. It creates a local environment* $\mathcal{E}_1$ *that contains an entry for the variable* `factyThree`*, with a placeholder value of* undefined*, and then evaluates the value expression (i.e., the* `lambda` *expression) with respect to the environment* $\mathcal{E}_1$*. Thus, the* `lambda` *function is created with respect to the environment* $\mathcal{E}_1$*. Subsequently applying this function to a numerical input causes the body of the function to be evaluated with respect to the environment* $\mathcal{E}_1$*, because that is the environment within which the function was created. Since* $\mathcal{E}_1$ *contains an entry for* `factyThree`*, all is well.*

Although this example is also illustrative, it seems kind of silly to create a function like `factyThree` to use it only once. The following example highlights are more common, useful way of using `letrec`.

Example 15.6.3: Using `letrec` to create a local recursive function within a wrapper function

The following interactions demonstrate the use of the `letrec` *special form to create a local recursive (helper) function* within *the body of a wrapper function. In this case, the wrapper function is* `facty`*, and the local recursive (helper) function is the accumulator-based, tail-recursive* `facty-acc` *function. Aside from defining* `facty-acc`*, the only thing that* `facty` *does is to call* `facty-acc` *with appropriate inputs.*

```
> (define facty
    (lambda (n)
      ;; Body of FACTY starts here
      (letrec ((facty-acc (lambda (m acc)
                            ;; Body of FACTY-ACC starts here
                            (if (<= m 1)
                                acc
                                (facty-acc (- m 1) (* m acc))))))
        ;; Body of LETREC starts here
        (facty-acc n 1))))
> (facty 4)
24
> (facty 5)
120
```

This kind of application of `letrec` *is commonly used to hide the existence of a recursive helper function from users who may not understand what inputs to give it, or may not want to be bothered with thinking about what inputs to give it. The helper function only exists for use by the parent function; it is not visible to the general programming public. The parent function (*`facty`*) takes care of supplying the helper function (*`facty-acc`*) with appropriate inputs.*

- ⋆ *Take care when defining local recursive helper functions. For example, note the difference between the input* `n` *to* `facty` *and the input* `m` *to* `facty-acc`*. On successive recursive function calls,* `m` *takes on different values, while* `n` *never changes.*

In-Class Problem 15.6.1

Carefully draw a diagram that shows all of the relevant environments, and the variable/value pairs in those environments, for the evaluation of `(facty 4)` *from the preceding example.*

Special Forms Introduced in this Chapter

`let`	Create local environment
`let*`	Create local environment, supports incremental computations
`letrec`	Create local environment, supports recursive function definitions

Built-in Functions Introduced in this Chapter

`random`	Pseudo-random number generator (an impure function)