

9 Local Variables and Local Environments: `let`, `let*` and `letrec`

This section introduces the `let` special form and its somewhat more general variants, `let*` and `letrec`. The purpose of a `let` special form is to set up a *local environment*, much like those that exist inside a function-call box, and then to evaluate one or more expressions with respect to that local environment. The value of a `let` expression is the value of the last expression in its *body*.¹⁷ Once a `let` expression has been evaluated, its local environment vanishes.¹⁸

As will be seen, the `let*` special form can do everything that a `let` can do, plus a little bit more. Similarly, a `letrec` special form can do everything that a `let*` can do, plus a little bit more. Thus, the `let` special form is the most basic of the three.

9.1 The `let` Special Form

When introducing any special form, it is important to specify both the syntax and the semantics.

The syntax of a `let` expression. The syntax of a `let` expression is as follows:

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

where:

- `var1, ..., varn` are character sequences representing n distinct Scheme symbols, where $n \geq 0$;
- `val1, ..., valn` are n Scheme expressions; and
- `expr1, ..., exprk` are k Scheme expressions, where $k \geq 1$.

The expressions, `expr1, ..., exprk`, constitute the *body* of the `let` expression.

⇒ Notice that a `let` can include *zero or more* `var/val` pairs; however, the body of a `let` must include *at least one* expression.

Example 9.1 Some legal `let` expressions The following expressions are all legal `let` expressions:

- `(let () #t)`
- `(let ((x (+ 2 3))) (* x x))`
- `(let ((x (+ 2 3))
 (y 3)
 (z (* 2 2)))
 (printf "x: ~A, y: ~A, z: ~A~%" x y z)
 (+ x y z))`

The first `let` expression includes no `var/val` pairs, as indicated by the empty list. Its body consists of the single expression, `#t`. The second `let` expression includes a single `var/val` pair: `(x (+ 2 3))`. Its body consists of the single expression, `(* x x)`. The third `let` expression includes three `var/val` pairs: `(x (+ 2 3))`, `(y 3)` and `(z (* 2 2))`. Its body consists of two expressions: a `printf` expression and `(+ x y z)`. ■

¹⁷As in previous sections, we intentionally blur the distinction between expressions and the Scheme entities they represent for the sake of expositional convenience. Formally speaking, a `let` expression represents a list whose first entry is the `let` symbol. That list is a `let` special form.

¹⁸There are some exceptions whereby a local environment can outlast the evaluation of its `let`, but a discussion of these exceptions would take us too far afield.

The semantics of a let expression. As usual, we specify the semantics of a `let` special form by the special way in which it is evaluated. A `let` expression of the form

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

is evaluated as follows.

- First, the expressions, `val1`, ..., `valn`, are evaluated.
- Second, a local environment is created containing n entries—one for each of the `var/val` pairs in the `let` expression. In particular, each symbol `vari` is associated with the result of *evaluating* the corresponding `vali` expression.
- Third, the expressions, `expr1`, ..., `exprk`, in the *body* of the `let` special form are evaluated, in turn, *with respect to that newly created local environment*. Thus, in the process of evaluating these expressions, if any of the symbols `vari` ever needs to be evaluated, its value is drawn from the newly created local environment. For other symbols, the parent environment—which is often the global environment—is used.
- The value of the last expression, `exprk`, is the value of the entire `let` expression.

Example 9.2 Evaluating let expressions The following Interactions Window session demonstrates the evaluation of the sample `let` expressions seen earlier.

```
> (let ()
    #t)
#t
> (let ((x (+ 2 3)))
    (* x x))
25
> (let ((x (+ 2 3))
        (y 3)
        (z (* 2 2)))
    (printf "x: ~A, y: ~A, z: ~A~%" x y z)
    (+ x y z))
x: 5, y: 3, z: 4
12
```

In the first expression, the local environment contains no entries. Thus, when the body of the `let` is evaluated, the result is the same as if it were evaluated outside the `let`. In particular, the expression, `#t`, evaluates to `#t`, which is reported as the value of the entire `let` expression. Since the purpose of a `let` expression is to set up a local environment, it is rare to see a `let` expression that contains no `var/val` pairs.

In the second expression, the local environment contains a single entry that associates the value 5 with the symbol `x`. Notice the plethora of parentheses required to represent a list containing a single entry that is itself a list! Furthermore, the second entry in that subsidiary list is itself a list! The body of the `let` consists of the single expression, `(* x x)`, which evaluates to 25 in this context. Notice that 25 is reported as the value of the entire `let` expression.

In the third expression, the local environment contains three entries: one associating the value 5 with `x`, one associating the value 3 with `y`, and one associating the value 4 with `z`. The body contains two expressions. The `printf` expression causes information to be displayed in the Interactions Window; the expression `(+ x y z)` is then evaluated, resulting in the value 12, which is reported as the value for the entire `let` expression. ■

Example 9.3 Local vs. Global The following Interactions Window session demonstrates that the local environment supercedes the global environment when evaluating expressions in the body of a `let`.

```

> (define x 1000)
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
        (y 4))
      (+ x y z))
17

```

The first three expressions use the `define` special form to create three global variables, named `x`, `y` and `z`. The last expression uses a `let` to create a local environment containing two local variables, named `x` and `y`. When the single expression in the body of the `let` is evaluated, the values for `x` and `y` are drawn from the local environment, whereas the values for `+` and `z` are drawn from the global environment. ■

Example 9.4 Typically, a `let` expression is used to store a value in a local variable so that it can be referred to in subsequent expressions. The following Interactions Window session demonstrates how the result of a call to the `random` function can be stored in a local variable.

```

> (let ((rnd (random 10)))
      (printf "The random number is: ~A~%" rnd)
      (printf "When squared, it is: ~A~%" (* rnd rnd))
      (printf "We'll return its cube: ~%"
              (* rnd rnd rnd)))
The random number is: 6
When squared, it is: 36
We'll return its cube:
216
> rnd
ERROR: reference to undefined identifier: rnd

```

However, notice that after the `let` expression has been evaluated, its local environment ceases to exist. Thus, the subsequent attempt to evaluate `rnd` causes DrScheme to report an error. (This example assumes that there is no entry for `rnd` in the global environment.) ■

Deriving the `let` special form from the `lambda` special form. If you're thinking that the evaluation of a `let` expression seems awfully close to the evaluation of a function call, you're right. In fact, the `let` special form 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 `let` expressions with certain expressions involving the application of a *lambda function*.

Example 9.5 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: ~A, y: ~A~%" x y)
      (+ x y))
x: 5, y: 12
17
> ((lambda (x y)
      (printf "x: ~A, y: ~A~%" x y)
      (+ x y))
   (+ 2 3)
   (* 3 4))
x: 5, y: 12
17

```

⇒ 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 9.6 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)
```

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.

9.2 The `let*` Special Form

The syntax of a `let*` special form is nearly identical to that of a `let` special form. However, the semantics is substantially different. In particular, the construction of the local environment happens in a different way. This difference allows a certain kind of incremental computation that turns out to be quite useful.

The syntax of a `let*` expression. A `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`.

The semantics of a `let*` expression. A `let*` expression of the form given above is evaluated as follows:

- A 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*.
- ⇒ Crucially, the *i*th entry in the local environment is created *before* the (*i* + 1)st value is computed. Thus, the expression, *val_{i+1}*, can refer to *any* of the *preceding* symbols, *var₁*, ..., *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 9.7 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 `let`:

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

Notice that the expression, `(+ x 2)`, used to compute the value for `y` refers to the local variable `x`. Similarly, the expression, `(* x y)`, used to compute the value for `z` refers to both `x` and `y`. Finally, the expression, `(+ x y z)`, 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: ~A, y: ~A, z: ~A, w: ~A~%" 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. ■

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) ...))
```

The following example should help to verify this equivalence.

Example 9.8 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: ~A, y: ~A, z: ~A, w: ~A~%" 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: ~A, y: ~A, z: ~A, w: ~A~%" 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`. ■

In general, `let*` provides a simpler syntax than the corresponding 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 probably will want to use `let*`.

9.3 The `letrec` Special Form

To be continued!