

let, let* and letrec in Scheme

Luke Hunsberger

February 23, 2009

Scheme contains several varieties of the `let` special form: `let`, `let*` and `letrec`. Aside from their slightly different names, the syntax of these special forms is the same:

```
(kwd ((var1 val1)
      (var2 val2)
      ...
      (varn valn))
  body )
```

where:

- *kwd* is one of the keywords: `let`, `let*` or `letrec`;
- *var*₁, *var*₂, ..., *var*_{*n*} are *n* symbols representing local variables;
- *val*₁, *val*₂, ..., *val*_{*n*} are *n* expressions representing the values for the corresponding variables; and
- *body* is one or more expressions.

Whether `let`, `let*` or `letrec`, the purpose of the special form is to set up a local environment containing entries for the variables, *var*₁, ..., *var*_{*n*}, whose values are determined by evaluating the corresponding expressions, *val*₁, ..., *val*_{*n*}. In each case, the expressions in the *body* are evaluated with respect to that newly created local environment.

The differences between the `let`, `let*` and `letrec` special forms involve the order in which the steps are taken to create the local environment. For convenience, suppose that the special form is being evaluated in the Interactions Window (i.e., with respect to the global environment).

The `let` special form. The local environment for the `let` special form is set up by taking the following steps, in the following order:

- (1) All of the expressions, *val*₁, ..., *val*_{*n*}, are evaluated with respect to the global environment.
- (2) Entries for the variables, *var*₁, ..., *var*_{*n*}, are created in the local environment, using the values computed in step 1.

Since all of the value expressions are evaluated with respect to the *global environment*, *none* of the value expressions can refer to *any* of the variables in the local environment.¹

As has already been mentioned, a `let` special form can be translated into an equivalent expression involving a single `lambda` expression.

The *let special form.** The local environment for the `let*` special form is set up incrementally, by *interleaving* the evaluation of the value expressions with the creation of the corresponding entries in the local environment, as follows.

- (1.1) The expression, val_1 , is evaluated with respect to the global environment.
- (1.2) An entry for the variable, var_1 , is created in the local environment using the value computed in step 1.1.
- (2.1) The expression, val_2 , is evaluated with respect to the local environment—which contains an entry for the variable, var_1 .
- (2.2) An entry for the variable, var_2 , is created in the local environment using the value computed in step 2.1.
- (3.1) The expression, val_3 , is evaluated with respect to the local environment—which contains entries for the variables, var_1 and var_2 .
- (3.2) An entry for the variable, var_3 , is created in the local environment using the value computed in step 3.1.
- ...
- (n .1) The expression, val_n , is evaluated with respect to the local environment—which contains entries for the variables, var_1, \dots, var_{n-1} .
- (n .2) An entry for the variable, var_n , is created in the local environment using the value computed in step n .1.

Since each value expression, val_i , is evaluated with respect to the local environment that has so far been created, val_i can refer to any of the *preceding* variables (i.e., any of $val_1, val_2, \dots, val_{i-1}$).

As has already been mentioned, a `let*` expression involving n variables is equivalent to n nested `let` expressions.

¹Even if the value expressions were evaluated with respect to the local environment, they couldn't refer to any of the variables, var_1, \dots, var_n , because no entries for those variables exist in the local environment when the value expressions are being evaluated.

The letrec special form. The local environment for a `letrec` special form is created very differently, as follows:

- (1) Entries for all of the variables, var_1, \dots, var_n , are created in the local environment *before* any of the value expressions are evaluated. Because none of the value expressions have been evaluated yet, the value associated with each variable is initially set to a special *undefined* value, which drScheme reports as `#<undefined>`.
- (2) Each of the value expressions, val_1, \dots, val_n , are evaluated, in turn, with respect to the local environment as it currently exists.

Since every variable has a local environment entry prior to the evaluation of any of the value expressions, each value expression can refer to *any* of the variables, var_1, \dots, var_n —even if they don't yet have proper values associated with them. In particular, a value expression, val_i , can refer to the variable, var_i . Although this might seem funny at first, it is precisely this feature that allows local variables in a `letrec` expression to be used as names for recursive functions.

For comparison, consider the following `define` special form:

```
(define facty
  (lambda (n)
    (if (= n 0)
        1
        (* n (facty (- n 1))))))
```

Notice that the value expression (i.e., the `lambda` expression) refers to the variable, `facty`, even though that variable doesn't yet have a value at the time the `lambda` expression is being evaluated. Crucially, the *body* of the `lambda` expression does not get evaluated until the function is *called*—by which time, `facty` is guaranteed to have a value.

Now consider the `letrec` expression below, which creates a local recursive function, called `factie`, and then uses it in its body:

```
> (letrec ((factie (lambda (n)
                    (if (= n 0)
                        1
                        (* n (factie (- n 1))))))
          (map factie '(1 2 3 4 5 6)))
  (1 2 6 24 120 720))
```

Finally, just for fun, consider the following example, which demonstrates that variables have a special *undefined* value prior to the evaluation of their corresponding value expression:

```
> (letrec ((a b)
          (b 34)
          (c b))
  (list a b c))
(#<undefined> 34 34)
```

In this case, the variable **b** does not yet have a value when the value expression for **a** is computed. Thus, **a** receives the *undefined* value—even though **b** later receives a value, it's too late for **a**! However, when the value expression for **c** is evaluated, **b** already has a value; so, **c** receives that same value.