

Naming Values with Local Bindings

Summary: Algorithm designers regularly find it useful to *name* the values their algorithms process. We consider why and how to name new values within an algorithm.

Contents:

- Introduction
- Redundant Work
- Scheme's `let` Expressions
- Sequencing Bindings with `let*`
- Positioning `let` Relative to `lambda`
- Local Procedures

Introduction

When writing programs and algorithms, it is useful to *name* values we compute along the way. For example, in an algorithm that computes the roots of a quadratic polynomial, we often name the coefficient of the quadratic term, a , the coefficient of the linear term, b , and the constant term, c . In addition, since there is a common term (“the square root of b squared minus four $a c$ ”), we might want to name that term, too. When we associate a name with a value, we say that we *bind* that name to the value.

So far we've seen three ways in which Scheme permits the algorithm writer to bind a name to a value:

- The names of built-in procedures, such as `cons` and `quotient`, are *predefined*. When DrScheme starts up, these names are already bound to the procedures they denote.
- The programmer can introduce a new binding by means of a *definition*. A definition may introduce a new equivalent for an old name, or it may give a name to a newly constructed value.
- When a programmer-defined procedure is called, the *parameters* of the procedure are bound to the values of the corresponding arguments in the procedure call. Unlike the other two kinds of bindings, parameter bindings are *local* -- they apply only within the body of the procedure. Scheme discards these bindings when it leaves the procedure and returns to the point at which the procedure was called.

Redundant Work

There are often times when it seems that you repeat work that should only have to be done once. For example, consider again the problem of computing the roots of a quadratic (assuming, of course, that it has two roots). We can write the following:

```
(define roots
  (lambda (a b c)
    (list (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 4 a c))
          (/ (- (- b) (sqrt (- (* b b) (* 4 a c)))) (* 4 a c)))))
```

But that's inefficient because we repeat *lot* of work. (It's also a bit dangerous, since we might have gotten it wrong in one place or the other, and we have to look closely to tell.) How can we name the common computations so that we do them only once? If we rely only on the Scheme we know so far, we can write a helper function that takes the three common parts as a parameter

```
;;; Procedure:
;;; roots-helper
;;; Parameters:
;;;   negative-b, a number
;;;   square-root-of, a number
;;;   two-a, a number
;;; Purpose:
;;;   Compute the roots of a quadratic.
;;; Produces:
;;;   (root1 root2), a list of two numbers
;;; Preconditions:
;;;   negative-b must be (- b), where b is the coefficient of the
;;;   linear term
;;;   square-root-of must be (sqrt (- (* b b) (* 4 a c))), where
;;;   a is the coefficient of the quadratic term, b is the
;;;   coefficient of the linear term, and c is the constant term.
;;;   two-a must be (* 2 a), where a is as above.
;;; Postconditions:
;;;   root1 and root2 are the roots of a quadratic. That is,
;;;   (+ (* a root1 root1) (* b root1) c) = 0
;;;   (+ (* a root2 root2) (* b root2) c) = 0
(define roots-helper
  (lambda (negative-b square-root-of two-a)
    (list (/ (+ negative-b square-root-of) two-a)
          (/ (- negative-b square-root-of) two-a))))
```

Now, we can simply write

```
(define roots
  (lambda (a b c)
    (roots-helper (- b) (sqrt (- (* b b) (* 4 a c))) (* 2 a))))
```

But that's a lot of extra work. It's inconvenient to have to write (and document!) a procedure that we're just going to use once. It's also not particularly clear in the call.

Scheme's **let** Expressions

Scheme provides **let** expressions as an alternative way to create local bindings. A **let**-expression contains a *binding list* and a *body*. The body can be any expression, or sequence of expressions, to be evaluated with the help of the local name bindings. The binding list is a pair of structural parentheses enclosing zero or more *binding specifications*; a binding specification, in turn, is a pair of structural parentheses enclosing a name and an expression.

That precise definition may have been a bit confusing, so here's the general form of a **let** expression

```
(let
  ((name1 exp1)
   (name2 exp2)
   ...
   (namen expn))
  body1
  body2
  ...
  bodym)
```

When Scheme encounters a `let`-expression, it begins by evaluating all of the expressions inside its binding specifications. Then the names in the binding specifications are bound to those values. Next, the expressions making up the body of the `let`-expression are evaluated, in order. The value of the last expression in the body becomes the value of the entire `let`-expression. Finally, the local bindings of the names are cancelled. (Names that were unbound before the `let`-expression become unbound again; names that had different bindings before the `let`-expression resume those earlier bindings.)

Here's how we'd solve the earlier problem with `let`.

```
(define roots
  (lambda (a b c)
    (let ((negative-b (- b))
          (square-root-of (sqrt (- (* b b) (* 4 a c))))
          (two-a (* 2 a)))
      (list (/ (+ negative-b square-root-of) two-a)
            (/ (- negative-b square-root-of) two-a)))))
```

Here's another example of a binding list, taken from a `let`-expression in a real Scheme program:

```
(let ((next (car source))
      (stuff null))
  ...)
```

This binding list contains two binding specifications -- one in which the value of the expression `(car source)` is bound to the name `next`, and the other in which the empty list is bound to the name `stuff`. Notice that binding lists and binding specifications are *not* procedure calls; their role in a `let`-expression simply to give names to certain values while the body of the expression is being evaluated. The outer parentheses in a binding list are “structural,” like the outer parentheses in a `cond`-clause -- they are there to group the pieces of the binding list together.

Using a `let`-expression often simplifies an expression that contains two or more occurrences of the same subexpression. The programmer can compute the value of the subexpression just once, bind a name to it, and then use that name whenever the value is needed again. Sometimes this speeds things up by avoiding such redundancies as the re0computation of values.

In other cases, there is little difference in speed, but the code may be a little clearer. For instance, consider the following `remove-all` procedure that removes all copies of a value from a list. In the past, we might have written that procedure as follows.

```

;;; Procedure:
;;;   remove-all
;;; Parameters:
;;;   item, a value
;;;   ls, a list of values
;;; Purpose:
;;;   Removes all copies of item from ls and its sublists.
;;; Produces:
;;;   newls, a list
;;; Preconditions:
;;;   ls is a list. It may be empty.
;;; Postconditions:
;;;   No values equal to item appear in newls.
;;;   Every value not equal to item that appeared in ls also
;;;   appears in newls.
;;;   Every value that appears in newls also appears in ls.
;;;   If a preceded b in ls and neither a nor b equals item,
;;;   then a precedes b in newls.
(define remove-all
  (lambda (item ls)
    (cond
      ; If the list is empty, removing the element still gives
      ; us the empty list
      ((null? ls) null)
      ; If the first element of the list matches, skip over it.
      ((equal? item (car ls)) (remove-all item (cdr ls)))
      ; Otherwise, preserve the first element and remove item
      ; from the remainder of ls
      (else (cons (car ls) (remove-all item (cdr ls)))))))

```

Here is an alternative definition of the `remove-all` procedure which some people find clearer.

```

(define remove-all
  (lambda (item ls)
    ; If the list is empty, removing the element still gives
    ; us the empty list
    (if (null? ls) null
        (let (
            ; Name the car of the list first-element.
            (first-element (car ls))
            ; Recurse on the rest of the list and name it
            ; rest-of result.
            (rest-of-result (remove-all item (cdr ls))))
          ; If the first element of the list matches, skip over it.
          (if (equal? first-element item)
              rest-of-result
              ; Otherwise, preserve the first element and attach
              ; it to the rest.
              (cons first-element rest-of-result))))))

```

Sequencing Bindings with `let*`

Sometimes we may want to name a number of interrelated things. For example, suppose we wanted to square the average of a list of numbers (well, it's something that people do sometimes). Since computing the average involves summing values, we may want to name two different things: the total and the average (mean). We can nest one `let`-expression inside another to name both things.

```
(let ((total (+ 8 3 4 2 7)))
  (let ((mean (/ total 5)))
    (* mean mean)))
```

One might be tempted to try to combine the binding lists for the nested `let`-expressions, thus:

```
;; Combining the binding lists doesn't work!
(let ((total (+ 8 3 4 2 7))
      (mean (/ total 5)))
  (* mean mean))
```

This wouldn't work (try it and see!), and it's important to understand why not. The problem is that, within one binding list, *all* of the expressions are evaluated before *any* of the names are bound. Specifically, Scheme will try to evaluate both `(+ 8 3 4 2 7)` and `(/ total 5)` before binding either of the names `total` and `mean`; since `(/ total 5)` can't be computed until `total` has a value, an error occurs. You have to think of the local bindings coming into existence simultaneously rather than one at a time.

Because one often needs sequential rather than simultaneous binding, Scheme provides a variant of the `let`-expression that rearranges the order of events: If one writes `let*` rather than `let`, each binding specification in the binding list is completely processed before the next one is taken up:

```
;; Using let* instead of let works!
(let* ((total (+ 8 3 4 2 7))
       (mean (/ total 5)))
  (* mean mean))
```

The star in the keyword `let*` has nothing to do with multiplication. Just think of it as an oddly shaped letter. It means "do things in sequence, rather than all at once". I have no idea why they've chosen to do that.

Positioning `let` Relative to `lambda`

In the examples above, we've tended to do the naming within the body of the procedure. That is, we write

```
(define proc
  (lambda (params)
    (let (...
          exp)))
```

However, Scheme also lets us choose an alternate ordering. We can instead put the `let` before (outside of) the `lambda`.

```
(define proc
  (let (...)
    (lambda (params)
      exp)))
```

Why would we ever choose to do so? Let us consider an example. Suppose that we regularly need to convert years to seconds. (When you have sons between the ages of 5 and 12, you'll understand.) You might begin with

```
(define years-to-seconds
  (lambda (years)
    (return (* years 365.24 24 60 60))))
```

This is, of course, correct. However, it is a bit hard to read. You might want to name some of the values for clarity.

```
(define years-to-seconds
  (lambda (years)
    (let* ((days-per-year 365.24)
           (hours-per-day 24)
           (minutes-per-hour 60)
           (seconds-per-minute 60)
           (seconds-per-year (* days-per-year hours-per-day
                                minutes-per-hour seconds-per-minute)))
      (* years seconds-per-year))))
> (years-to-seconds 10)
315567360.0
```

We have clearly clarified the code, although we have also lengthened it a bit. However, as we noted before, a second goal of naming is to avoid recomputation of values. Unfortunately, even though the number of seconds per year never changes, we compute it *every* time that someone calls `years-to-seconds`. How can we avoid this recomputation? One strategy is to move the bindings to `define` statements.

```
(define days-per-year 365.24)
...
(define seconds-per-year (* days-per-year ... seconds-per-minute))
(define years-to-seconds
  (lambda (years)
    (* years seconds-per-year)))
```

However, such a strategy is a bit dangerous. After all, there is nothing to prevent someone else from changing the values here.

```
(define days-per-year 360) ; Some strange calendar, perhaps in Indiana
...
> (years-to-seconds 10)
311040000
```

What we'd like to do is to declare the values once, but keep them local to `years-to-seconds`. The strategy is to move the `let` outside the `lambda`.

```

(define years-to-seconds
  (let* ((days-per-year 365.24)
        (hours-per-day 24)
        (minutes-per-hour 60)
        (seconds-per-minute 60)
        (seconds-per-year (* days-per-year hours-per-day
                              minutes-per-hour seconds-per-minute)))
    (lambda (years)
      (* years seconds-per-year))))
> (years-to-seconds 10)
315567360.0

```

As you'll see in the lab, it is possible to empirically verify that the bindings occur only once in this case, and each time the procedure is called in the prior case.

So, one moral of this story is *whenever possible, move your bindings outside the `lambda`*. However, it is not always possible to do so. For example, if your `let`-bindings use parameters (as in the earlier quadratic example), then you need to keep them within the body of the `lambda`.

Local Procedures

As you may have noted, `let` behaves somewhat like `define` in that programmers can use it to name values. But we've used `define` to name more than values; we've also used it to name procedures. Can we also use `let` for procedures?

Yes, one can use a `let`- or `let`*-expression to create a local name for a procedure. And we name procedures locally for the same reason that we name values, because it speeds and clarifies the code.

```

(define hypotenuse-of-right-triangle
  (let ((square (lambda (n) (* n n))))
    (lambda (first-leg second-leg)
      (sqrt (+ (square first-leg) (square second-leg))))))

```

Regardless of whether `square` is also defined outside this definition, the local binding gives it the appropriate meaning within the `lambda`-expression that describes what `hypotenuse-of-right-triangle` does.

Note, once again, that there are two places one might define `square` locally. We can define it before the `lambda` (as above) or after the `lambda` (as below). In the first case, the definition is done only once. In the second case, it is done *every time* the procedure is executed.

```

(define hypotenuse-of-right-triangle
  (lambda (first-leg second-leg)
    (let ((square (lambda (n) (* n n))))
      (sqrt (+ (square first-leg) (square second-leg))))))

```

So, which we should you do it? If the helper procedure you're defining uses any of the parameters of the main procedure, it needs to come after the `lambda`. Otherwise, it is generally a better idea to do it before the `lambda`. As you practice more with `let`, you'll find times that each choice is appropriate.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.