

Naming Values with Local Bindings

- Redundant Work
- Let
- Sequencing Bindings with `let*`
- Local Procedures

So far we've seen three ways in which a value can be associated with a name in Scheme:

- 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 the problem of squaring the average of a list of numbers. We can write

```
(* (/ (sum lst) (length lst))
    (/ (sum lst) (length lst)))
```

But that's inefficient because we repeat the work of summing the list and computing the length of the list. How can we name the common computation so that we do it only once? If we rely only on the Scheme we know so far, we can write a helper function that takes the average as a parameter.

```
;;; Procedure:
;;; square
;;; Parameters:
;;; val, a number
;;; Purpose:
;;; Compute val*val
;;; Produces:
;;; The result of the computation
;;; Preconditions:
;;; val must be a number
;;; Postconditions:
;;; The result is the same "type" of number as val (e.g., if
;;; val is an integer, so is the result; if val is exact,
;;; so is the result).
;;; Citations:
;;; Based on code created by John David Stone dated March 17, 2000
;;; and contained in the Web page
```

```

;;; http://www.math.grin.edu/~stone/courses/scheme/procedure-definitions.xhtml
;;; Changes to
;;;   Parameter names
;;;   Formatting
;;;   Comments
(define square
  (lambda (val)
    (* value val)))

```

Now, we can simply write

```
(square (/ (sum lst) (length lst)))
```

In that case, `value` names the average of the sum of `lst` and the length of `lst`.

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.

Let

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.

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`.

```
(let ((average (/ (sum lst) (length lst))))
  (* average average))

```

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

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

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 recomputation of values.

In other cases, there is little difference in speed, but the code may be a little clearer. For instance, consider the `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, preseve 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-reulst
            (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.

Local Procedures

One can use a `let-` or `let*-` expression to create a local name for a procedure:

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