

## Naming Values with Local Bindings

**Summary:** In this laboratory, you will ground your understanding of the basic techniques for naming values and procedures in Scheme, `let` and `let*`.

### Contents:

- Exercises
  - Exercise 1: Evaluating `let`
  - Exercise 2: Nesting Lets
  - Exercise 3: Simplifying Nested Lets
  - Exercise 4: Viewing Bindings
  - Exercise 5: Ordering Bindings
  - Exercise 6: Ordering Bindings, Revisited
  - Exercise 7: Binding and I/O
  - Exercise 8: Access to Local Procedures
  - Exercise 9: Finding the Longest Element List
  - Exercise 10: Alternate Techniques
  - Exercise 11: Another Alternative
- For Those With Extra Time
  - Extra 1: Checking Preconditions
  - Extra 2: Timing Versions

## Exercises

### Exercise 1: Evaluating `let`

What are the values of the following `let`-expressions? You may use DrScheme to help you answer these questions, but be sure you can explain how it arrived at its answers.

a.

```
(let ((tone "fa")
      (call-me "al"))
  (list call-me tone "l" tone))
```

b.

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

c.

```
(let ((inches-per-foot 12)
      (feet-per-mile 5280))
  (let ((inches-per-mile (* inches-per-foot feet-per-mile))
        (* inches-per-mile inches-per-mile)))
```

## Exercise 2: Nesting Lets

Write a nested `let`-expression that binds a total of five names, `alpha`, `beta`, `gamma`, `delta`, and `epsilon`, with `alpha` bound to 9387 and each subsequent name bound to a value twice as large as the one before it. That is, `beta` should be twice as large as `alpha`, `gamma` twice as large as `beta`, and so on. The body of the innermost `let`-expression should then compute the sum of the values of the five names.

## Exercise 3: Simplifying Nested Lets

Write a `let*`-expression equivalent to the `let`-expression in the previous exercise.

## Exercise 4: Viewing Bindings

The file

`/home/rebelsky/Web/Courses/CS151/2006F/Examples/verbose-bindings.ss` contains alternative versions of `define`, `let`, and `let*` (named `verbose-define`, `verbose-let`, and `verbose-let*`).

- Load this file.
- Rewrite the examples from Exercise 1 to use `verbose-let`.
- Rewrite your code from Exercise 2 to use `vebose-let`.
- Rewrite your code from Exercise 3 to use `verbose-let*`.

## Exercise 5: Ordering Bindings

In the reading, we noted that it is possible to move bindings outside of the `lambda` in a procedure definition. In particular, we noted that the first of the two following versions of `years-to-seconds` required recomputation of `seconds-per-year` every time it was called while the second required that computation only once.

```
(define years-to-seconds-a
  (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))))
(define years-to-seconds-b
  (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))))

```

a. Confirm that `years-to-seconds-a` does, in fact, recompute the values each time it is called. (You might, for example, replace `let*` by `verbose-let*`.)

b. Confirm that `years-to-seconds-b` does not recompute the values each time it is called. (Again, replace the `let*` by `verbose-let*`.)

c. Given that `years-to-seconds-b` does not recompute each time, when does it do the computation? (Consider when you see the messages.)

## Exercise 6: Ordering Bindings, Revisited

You may recall that we defined a procedure to compute the roots of a quadratic polynomial as follows:

```

(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))))))

```

You might be tempted to move the `let` clause outside the `lambda`, just as we did in the previous exercise. However, as we noted in this reading, this reordering will fail.

a. Verify that it will not work to move the `let` before the `lambda`, as in

```

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

```

b. Explain, in your own words, why this fails (and why it should fail).

## Exercise 7: Binding and I/O

Local bindings are particularly useful when you are dealing with input (and, at times, output). Recall that we regularly had to call a helper procedure so that we could name values. For example, we had to call `sum-of-file-helper-helper` to name the value `read nextval`.

```
(define sum-of-file
  (lambda (filename)
    (sum-of-file-helper (open-input-file filename))))

(define sum-of-file-helper
  (lambda (source)
    (sum-of-file-helper-helper source (read source))))

(define sum-of-file-helper-helper
  (lambda (source nextval)
    (cond
      ((eof-object? nextval) (close-input-port source) 0)
      ((number? nextval) (+ nextval
                            (sum-of-file-helper source)))
      (else (sum-of-file-helper source)))))
```

Now, we can name that value without calling the helper-helper.

```
(define sum-of-file-helper
  (lambda (source)
    (let ((nextval (read source)))
      (cond
        ((eof-object? nextval) (close-input-port source) 0)
        ((number? nextval) (+ nextval (sum-of-file-helper source)))
        (else (sum-of-file-helper source))))))
```

Verify that this update works. Recall that the file `/home/rebelsky/Web/Courses/CS151/2006F/Examples/numbers.dat` contains a variety of numbers that sum to 258588.

## Exercise 8: Access to Local Procedures

Consider the following procedure that squares all the values in a list.

```
;;; Procedure:
;;; square-values
;;; Parameters:
;;; lst, a list of numbers of the form (num_1 num_2 ... num_n)
;;; Purpose:
;;; Squares all the values in lst.
;;; Produces:
;;; list-of-squares, a list of numbers
;;; Preconditions:
;;; [Standard]
;;; Postconditions:
;;; list-of-squares has the form (square_1 square_2 ... square_n)
;;; For all i, square_i is the square of num_i (that is num_i * num_i).
(define square-values
```

```

(lambda (lst)
  (let ((square (lambda (val) (* val val))))
    (if (null? lst)
        null
        (cons (square (car lst)) (square-values (cdr lst)))))))

```

a. Verify that `square-values` works correctly.

b. Try to execute `square` outside of `square-values`. Explain what happens.

## Exercise 9: Finding the Longest Element List

Here is a procedure that takes a non-empty list of lists as an argument and returns the longest list in the list (or one of the longest lists, if there is a tie).

```

;;; Procedure:
;;; longest-list-in-list
;;; Parameters:
;;; los, a list of lists
;;; Purpose:
;;; Finds one of the longest lists in los.
;;; Produces:
;;; longest, a list
;;; Preconditions:
;;; los is a nonempty list.
;;; every element of los is a list.
;;; Postconditions:
;;; Does not affect los.
;;; Returns an element of los.
;;; No element of los is longer than longest. That is,
;;; For each lst in los, (length los) >= (length lst).
(define longest-list-in-list
  (lambda (los)
    ; If there is only one list, that list must be the longest.
    (if (null? (cdr los))
        (car los)
        ; Otherwise, take the longer of the first list and the
        ; longest remaining list.
        (longer-list (car los) (longest-list-in-list (cdr los)))))

```

This definition of the `longest-list-in-list` procedure includes a call to the `longer-list` procedure, which returns the longer of two given lists:

```

;;; Procedure:
;;; longer-list
;;; Parameters:
;;; left, a list
;;; right, a list
;;; Purpose:
;;; Find the longer of left and right.
;;; Produces:
;;; longer, a list
;;; Preconditions:
;;; Both left and right are lists.
;;; Postconditions:

```

```

;;; longer is a list.
;;; longer is either equal to left or to right.
;;; (>= (length longer) (length left))
;;; (>= (length longer) (length right))
(define longer-list
  (lambda (left right)
    (if (<= (length right) (length left))
        left
        right)))

```

Revise the definition of `longest-list-in-list` so that the name `longer-list` is bound to the procedure that it denotes only locally, in a `let`-expression.

## Exercise 10: Alternate Techniques

Note that there are at least two possible ways to do the previous exercise: The definiens of `longest-list-in-list` can be a `lambda`-expression with a `let`-expression as its body, or it can be a `let`-expression with a `lambda`-expression as its body. That is, it can take the form

```

(define longest-list-in-list
  (let (...)
    (lambda (los)
      ...)))

```

or the form

```

(define longest-list-in-list
  (lambda (los)
    (let (...)
      ...)))

```

- Define `longest-list-in-list` in whichever way that you did not define it for the previous exercise.
- Does the order of nesting affect what happens when the procedure is invoked? You may want to use `verbose-let` to help you answer this question.
- If there is a difference, which arrangement is better? Why?

## Exercise 11: Another Alternative

The two definitions you came up with in the previous exercises are not the only alternatives you have in placing the `let`. Since `longer-list` is only needed in the recursive case, you can place the `let` there.

```

(define longest-list-in-list
  (lambda (los)
    ; If there is only one list, that list must be the longest.
    (if (null? (cdr los))
        (car los)
        ; Otherwise, take the longer of the first list and the
        ; longest remaining list.
        (let ((longer-list

```

```
(lambda (left right)
  (if (<= (length right) (length left))
      left
      right))))
(longer-list (car los) (longest-list -in-list (cdr los))))))
```

Including the original definition (in which `longer-list` is bound with `define`), you've now seen or written four variants of `longest-list-in-list`. Which do you prefer? Why?

## For Those With Extra Time

### Extra 1: Checking Preconditions

Extend your favorite version of `longest-list-in-list` so that it verifies its preconditions (i.e., that `los` only contains lists and that `los` is nonempty). If the preconditions are not met, your procedure should return `#f`.

It is perfectly acceptable for you to check each list element in turn to determine whether or not it is a list, rather than to check them all at once, in advance.

### Extra 2: Timing Versions

Dr. Scheme provides a keyword, `time`, that reports various metrics for the time it takes to execute an expression. Using DrScheme's `(time exp)` operation, determine which of the four versions of `longest-list-in-list` is indeed the fastest.

In doing this testing, you should build a fairly long outer list. The inner lists can be mostly 0- or 1-element lists..

---

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.