

Naming Values with Local Bindings

- Exercises
 - Exercise 0: Preparation
 - Exercise 1: Evaluating `let`
 - Exercise 2: Nesting Lets
 - Exercise 3: Simplifying Nested Lets
 - Exercise 4: Finding the Longest String
 - Exercise 5: Alternate Techniques
 - Exercise 6: Another Alternative
 - Exercise 7: Checking Preconditions
- Notes

Exercises

Exercise 0: Preparation

Start DrScheme

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.

```
;; Solutions to the quadratic equation  $x^2 - 5x + 4$ :
(let ((discriminant (- (* -5 -5) (* 4 1 4))))
  (list (/ (+ (- -5) (sqrt discriminant)) (* 2 1))
        (/ (- (- -5) (sqrt discriminant)) (* 2 1))))
```

c.

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

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 -- 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: Finding the Longest String

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

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

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

```
;;; Procedure:
;;; longer-string
;;; Parameters:
;;; left, a string
;;; right, a string
;;; Purpose:
;;; Find the longer of left and right.
;;; Produces:
;;; longer, a string
;;; Preconditions:
```

```

;;; Both left and right are strings.
;;; Postconditions:
;;; longer is a string.
;;; longer is either equal to left or to right.
;;; longer is at least as long as left.
;;; longer is at least as long as right.
(define longer-string
  (lambda (left right)
    (if (<= (string-length right) (string-length left))
        left
        right)))

```

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

Exercise 5: Alternate Techniques

Note that there are at least two possible ways to do the previous exercise: The definiens of `longest-string-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-string-in-list
  (let (...))
  (lambda (los)
    ...)))

```

or the form

```

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

```

a. Define `longest-list-in-list` in whichever way that you did not define it for the previous exercise.

b. Does the order of nesting affect what happens when the procedure is invoked? If so, which arrangement is better? Why?

Exercise 6: Another Alternative

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

```

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

```

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

Including the original definition, you've now seen or written four variants of `longest-string-in-list`. Which do you prefer? Why?

Exercise 7: Checking Preconditions

Extend your favorite version of `longest-string-in-list` so that it verifies its preconditions (i.e., that `los` only contains strings and that `los` is nonempty).

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

Notes