SYSC 3101 Winter 2019 Lab 6 Solutions

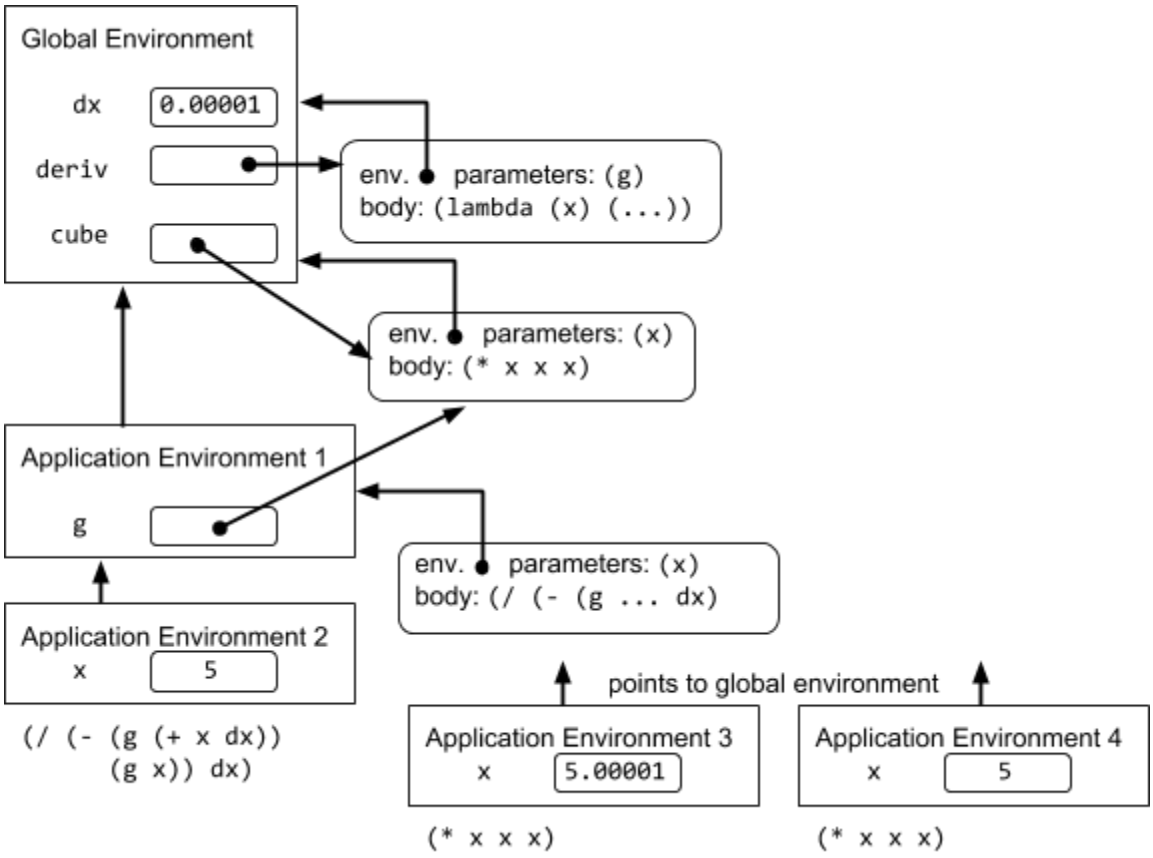
Exercise 1

This exercise deals with a solution to Question 6 from the midterm exam.

Draw a diagram that depicts the environment created when these expressions are evaluated:

```
(define dx 0.00001)

(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))

(define (cube x) (* x x x))
((deriv cube) 5)  ; Returns 75.00014999664018
```

Solution



Exercises 2 and 3 deal with a solution to Question 5 from the midterm exam.

Exercise 2

Recall that Racket's procedure `(build-list n f)` constructs a list by applying procedure `f` to the natural numbers between `0` and `n`-1, inclusive. In other words, `(build-list n f)` produces the same result as:

```
(list (f 0) (f 1) .. (f (- n 1)))
```

For example, when this expression is evaluated:

```
(build-list 5 (lambda (x) (* x x)))
```

`build-list` applies the `lambda` procedure to `0`, `1`, `2`, `3` and `4`, and returns `'(0 1 4 9 16)`.

Assume we are using a version of Racket that doesn't have `build-list`, and have therefore written our own implementation of this procedure:

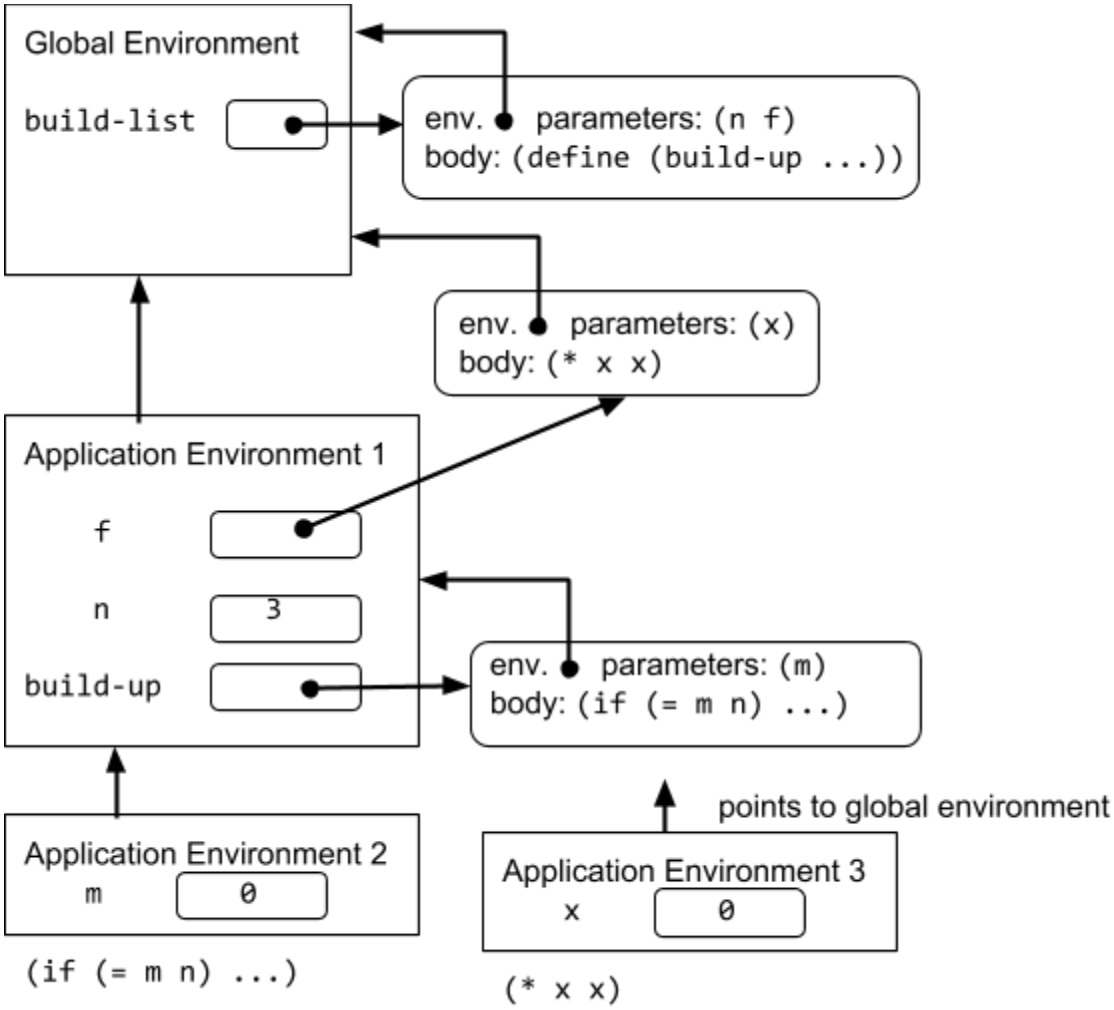
```
(define (build-list n f)

  ; build-up creates a list by applying procedure f to the integers
  ; from m to n-1, m < n, in order. The first list element is the
  ; value produced by (f m), and the last list element is the value
  ; produced by (f (- n 1)).

  (define (build-up m)
    (if (= m n)
        '()
        (cons (f m) (build-up (+ m 1)))))

  (build-up 0))
```

Draw a diagram that depicts the environment created by:

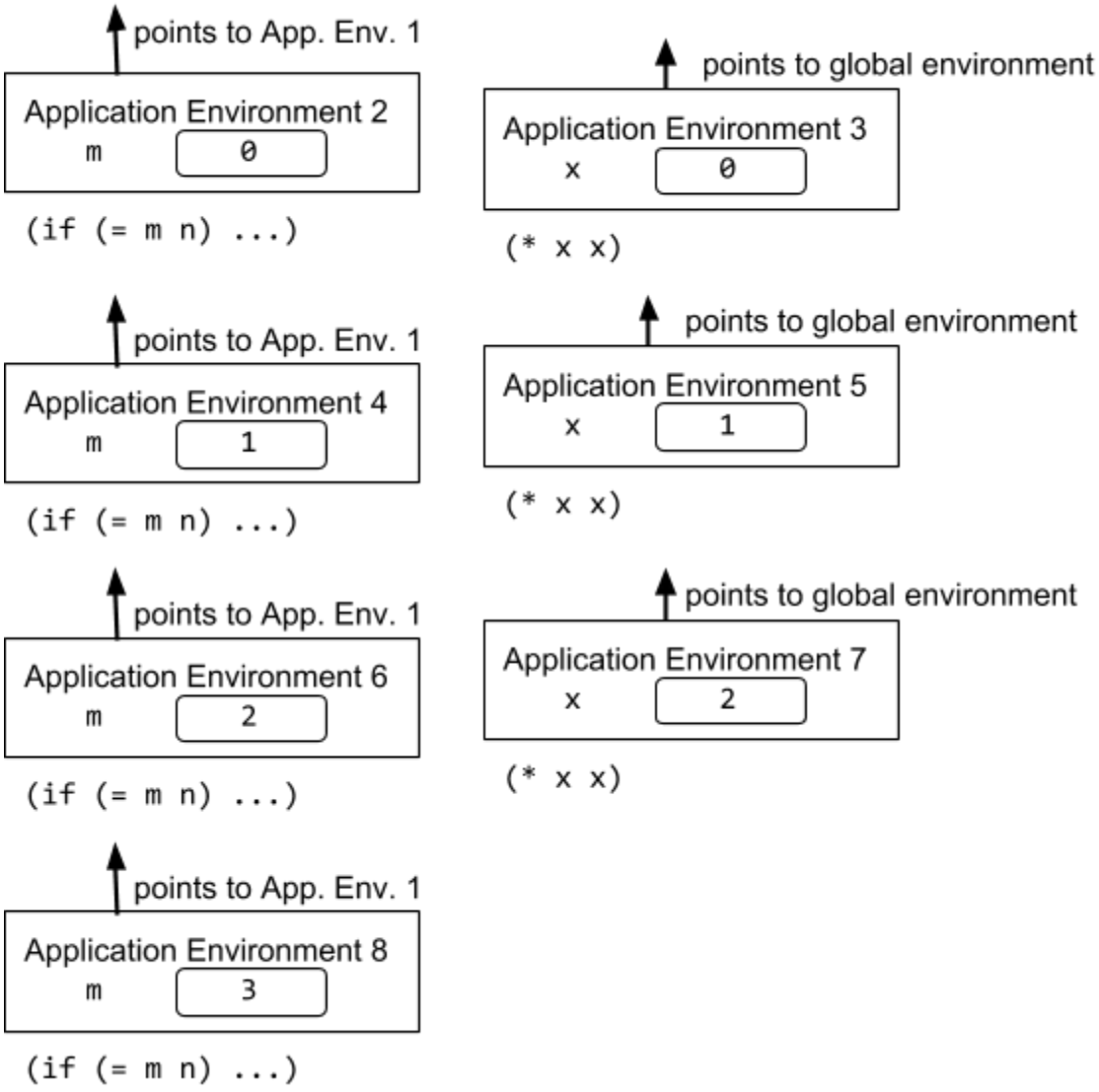
```
(build-list 3 (lambda (x) (* x x)))
```

Solution

This diagram shows the environment up to the point when the call expression `(build-up 0)` has been executed, and `build-up` has called the procedure that squares its argument (the procedure bound to `f`), with argument 0.



This diagram shows that `build-up` is called four times (with arguments 0, 1, 2 and 3) and the procedure bound to `f` is called three times (with arguments 0, 1 and 2).



Notice that the value returned by `(f m)` is `cons`'d with the list returned by `(build-up (+ m 1))`, and the resulting list is returned by `build-up`. Eventually, the call `(build-up 0)` returns the list `'(0 1 4)`, which is then returned by `build-list`.

Exercise 3

Procedure `multiply-all` takes a list of numbers and returns the product of the numbers in the list:

```
(define (multiply-all nums)
  (if (empty? nums)
      1
      (* (car nums) (multiply-all (cdr nums)))))

> (multiply-all '(2 3 4))    ; returns 24
> (multiply-all '(3))        ; returns 3
> (multiply-all '())         ; returns 1
```

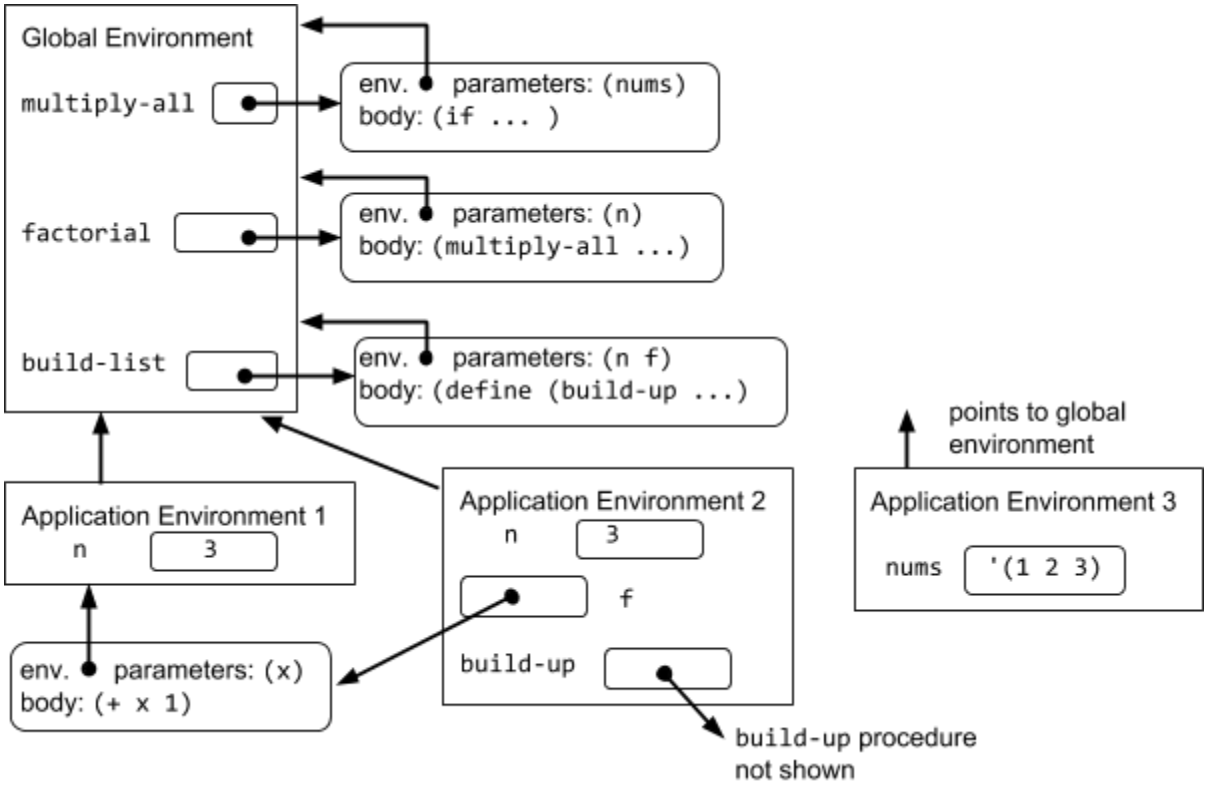
Procedure `factorial` takes a positive integer *n* and returns *n*!

```
(define (factorial n)
  (multiply-all (build-list n (lambda (x) (+ x 1)))))
```

Draw a diagram that depicts the environment created by: `(factorial 3)`

Don't duplicate your solution to Exercise 1. Your diagram should show the frames created when `factorial`, `multiply-all` and `build-list` are called, but you don't have to show the frames created when `build-up` and the `lambda` procedure are called.

Solution



Exercise 4

This example was presented in one of the lectures:

```
(define (make-upcounter counter)
  (lambda ()
    (set! counter (+ counter 1))
    counter))
```

(a) Draw a diagram that depicts the environment created by:

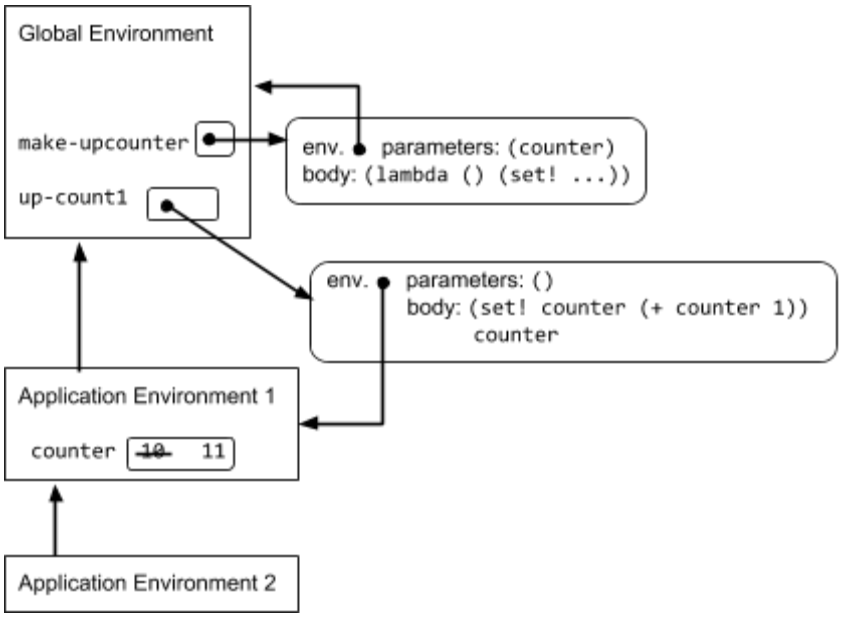
```
(define up-count1 (make-upcounter 10))
```

Use the notation presented in Evans' book.

(b) Draw a diagram that depicts the environment that is created the first time the expression `(up-count1)` is evaluated. Use the notation presented in Evans' book.

(c) Figure 10.1 in Evans' book depicts the environment created by a `make-counter` procedure that uses a `let` expression to initialize the variable that stores the counter value. Compare your diagrams from (a) and (b) to Figure 10.1. Ensure that you can explain the differences between that figure and your diagrams.

Solution



For a step-by-step discussion of this solution, see the document that summarizes the final lecture.