CSE 341 | Section 6

Racket: Basics, Lists, and Delayed Evaluation

Q1 (Scope Review): Consider the following Racket code:

```
(define x 3)
(define f1
  (lambda (x)
    (let ([y (+ x 1)])
      (+ y x))))
```

```
(define x 3)
(define f2
  (let ([y (+ x 1)])
    (lambda (x)
      (+ y x))))
```

What is `(f1 2)` bound to? **5**

What is `(f2 2)` bound to? **6**

Q2 (Functions): Define a function named `digit-sum` that accepts an integer `n` and returns the sum of its digits. For example, the call `(digit-sum 341)` should return 8 (since 3 + 4 + 1 = 8). If passed a negative parameter, the function should return the negative sum of the digits. For example, the call `(digit-sum -341)` should return -8 (since -(3 + 4 + 1) = -8). You may assume that only integers are passed as arguments. Use racket functions `remainder` and `quotient`.

```
(define (digit-sum n)
  (if (= n 0)
      0
      (+ (remainder n 10)
         (digit-sum (quotient n 10)))))
```

Q3 (Functions): Write a Racket function named star-string that accepts an integer argument n and returns a string of stars (asterisks) 2n long (i.e., 2 to the nth power). Use the racket `string-append` function. For example:

Call	Output	Reason
`(star-string 0)`	`"*"`	$2^0 = 1$
`(star-string 1)`	`"**"`	$2^1 = 2$
`(star-string 2)`	`"****"`	$2^2 = 4$
`(star-string 3)`	`"********"`	$2^3 = 8$

```
(define (star-string n)
  (if (= n 0)
      "*"
      (let
          ([s (star-string (- n 1))])
        (string-append s s))))
```

Q4 (Lists): Define a function `count-in-range` that takes a list of numbers and two numbers `lo` and `hi` and returns the count of elements in the list that are between `lo` and `hi` (inclusive).

```
(define (count-in-range xs lo hi)
  (if (null? xs)
      0
      (let ([x (car xs)]
            [cnt (count-in-range (cdr xs) lo hi)])
        (if (and (>= x lo) (<= x hi))
            (+ 1 cnt)
            cnt))))
```

Test: (= 4 (count-in-range (list 1 2 3 4 5) 2 5))

Q5 (Lists): Define a function `partition-parity` that takes a list as an argument and returns a pair of lists such that the first list holds all even values of the argument and the second list holds all odd (maintaining original order). Use racket function `even?` or `odd?`.

```
(define (partition-parity xs)
  (if (null? xs)
      (cons null null)
      (let ([x (car xs)]
            [rest (partition-parity (cdr xs))])
        (if (even? x)
            (cons (cons x (car rest)) (cdr rest))
            (cons (car rest) (cons x (cdr rest)))))))
```

Q6 (Streams): Define a function `zero-through-three` that returns a stream which cycles through the values 0, 1, 2, 3 every time it's called, starting with 0 (Racket has a function `remainder` that may be useful).

```
(define zero-through-three
  (letrec ([f (lambda (x)
                (cons (remainder x 4)
                      (lambda () (f (+ x 1)))))])
    (lambda () (f 0))))
```

Q7 (Streams): Define a function `zero-through-n` that takes a number `n` and returns a stream which cycles through the values 0, 1, 2, …, `n` every time it's called, starting with 0. You may assume `n` is non-negative.

```
(define (zero-through-n n)
  (letrec ([f (lambda (x)
                (cons (remainder x n)
                      (lambda () (f (+ x 1)))))])
    (lambda () (f 0))))
```