CS61A | Midterm 1 Review Solutions (v1.0)

Basic Info

Your login:
Your section number:
Your TA's name:
Your signature:

Midterm 1 is going to be held on Wednesday 6-830p, at 1 Pimentel. There is going to be a group question after the individual exam.

What will Scheme print?

What will the Scheme interpreter print in response to each of the following expressions?

1. `(butfirst '(help!))`
   ()
2. `(+ 27 '(word 1 0))`
   Error
3. `(and 127 ((lambda (y) #f)))`
   Error
4. `(cond (first '(1 2 3)) (else 'foo))`
   (1 2 3)
5. `(* (+ 5))`
   5
6. `(if (23) (23) (23))`
   Error

7.
   ```
   ((let ()
        (lambda () 2)))
   ```
   2

8.
   ```
   (every (lambda (x) (word (first x) '- x))
          '(waiting for my rocket to come))
   ```
   (w-waiting f-for m-my r-rocket t-to c-come)
9. `((lambda (x) (* 2 (x x))) (lambda (y) (* 4 (y y))))`
   infinite loop

Efficiency, Applicative/Normal Order, Invariants

1. Here are some definitions:

```
(define (square x) (* x x))
(define (foo x y) x)
(define (bar x y) (* x y y))    ;; treat as one call to *
```

   (a) How many times is * called when we evaluate `(foo (square (* 1 1)) (square (* 1 1)))` in normal order?
   3
   in applicative order?
   4

   (b) How many times is * called when we evaluate `(bar (square (* 1 1)) (square (* 1 1)))` in normal order?
   10
   in applicative order?
   5

2. Given a function `FOO`:

```
    (define (foo n)
       (if (< n 2)
           1
           (+ (baz (- n 1))
              (baz  (- n 2))))))
```

   For each of the following definitions of `BAZ`, state the order of growth of `FOO`.

   (a) `(define baz fib)`
   $\Theta(2^n)$ **or** $\Theta(n)$

   (b) `(define (baz n) (+ n (- n 1)))`
   $\Theta(1)$

   (c) `(define baz foo)`
   $\Theta(2^n)$

   (d) `(define baz factorial)`
   $\Theta(n)$

3. The function below, truncate, takes a sentence and cuts it off at the first occurrence of the word "end". If that word never appears, it returns the whole sentence.

```
(define (truncate sent)
  (define (helper part1 part2)
    (cond ((empty? part2) part1)
      ((eq? (first part2) 'end) part1)
      (else (helper (se part1 (first part2)) (bf part2)))))
  (helper '() sent))
```

   What is the most useful invariant of the helper function helper?
   (se part1 part2) = sent

Recursion

1. Write a procedure interleave, which takes two sentences as arguments and returns a sentence containing the first sentence, then the first element of the second sentence, then the second of the first sentence, then the second of the second sentence, etc. etc.

```
    > (interleave '(dont so to away) '(be quick walk))
    (dont be so quick to walk away)

    > (interleave '(i feel) '(cant the way i did before))
    (i cant feel the way i did before)
```

```
(define (interleave s1 s2)
  (cond ((empty? s1) s2)
        ((empty? s2) s1)
        (else (se (first s1) (first s2) (interleave (bf s1) (bf s2))))))
```

2. Write a function running-total, that takes in non- empty sentence of numbers and returns another sentence of the running totals. In other words, the first number in the resulting sentence should be the first number of the argument sentence; the second number in the resulting sentence should be the sum of the first and second numbers in the argument sentence, and so on.

```
> (running-total '(1 2 3 4))
(1 3 6 10)
> (running-total '(-5 0 -22 18 55))
(-5 -5 -27 -9 46)
> (running-total '(3))
(3)
```

```
(define (running-total sent)
  (define (helper sent sum)
    (if (empty? sent)
        '()
        (se (+ num (first sent))
            (helper (bf sent) (+ sum (first sent))))))
  (helper sent 0))
```

Higher Order Functions/Lambdas

1. Write a procedure named `constant-fn?` that takes two arguments: a function of one argument and a sentence of numbers. It should return *#t* if the value returned by the function is always the same for all numbers in the argument sentence.

   ```
   > (constant-fn? sqrt '(2 3 4 5 6))
   #f
   > (constant-fn? (lambda (x) 4) '(5 3 88 2 100 7 8 9))
   #t
   > (constant-fn? (lambda (a) (< a 10)) '(22 23 24 25 26 27))
   #t
   > (constant-fn? (lambda (a) (< a 10)) '(5 7 9 11))
   #f
   ```

   ```
   (define (constant-fn? fn sent)
     (empty? (keep (lambda (num) (not (equal? (fn num) (fn (first sent)))))
                   (bf sent))))
   ```

2. Define a procedure called make-keeper that takes a unary (one-argument) predicate function as an argument and returns a procedure that, when invoked on a sentence argument, will keep only those elements that satisfy the predicate.

   ```
   > ((make-keeper even?) '(1 2 3 4 5 6))
   (2 4 6)
   ```

   You may assume that the input to make-keeper will always be a unary function and that the input to the returned procedure will always be a sentence. **Use HOF, not recursion**

   ```
   (define (make-keeper pred)
     (lambda (sent) (keep pred sent)))
   ```