

CS 275

Sample Final Exam

This is more or less the final exam from Fall 2018. I changed the wording on a few problems so it will make sense in Fall 2019, and I added a question on call/cc.

1. **Write procedure (sizeOf L)** which returns the number of atoms in general list L. For example (sizeOf '(a b (c (d e) f (g)) h)) returns 8.

```
(define sizeOf (lambda (L)
  (cond
    [(null? L) 0]
    [(atom? L) 1]
    [else (apply + (map sizeOf L))])))
```

2. Write procedure (**sort vec**) where **vec** is a flat list of numbers. For example, (**sort '(7 4 5 9 3)**) returns (3 4 5 7 9).

```
(define sort
  (lambda (vec)
    (cond
      [(null? vec) null]
      [else (insert (car vec) (sort (cdr vec)))])))

(define insert
  (lambda (a vec)
    (cond
      [(null? vec) (list a)]
      [< a (car vec)] (cons a vec)
      [else (cons (car vec) (insert a (cdr vec)))])))
```


4. The *depth* of a flat list is 1; the depth of a list containing a flat list is 2; the depth of a general list is 1 more than the greatest depth of any of its elements. For example, the depth of '(a (b) c (d (e (f g))) h)' is 4 because the depth of (f g) is 1, the depth of (e (f g)) is 2, and the depth of (d (e (f g))) is 3. **Use procedures map and apply to write function (depth L) which returns the depth of general list L.**

```
(define depth
  (lambda (L)
    (cond
      [(atom? L) 0]
      [else (+ 1 (apply max (map depth L)))])))
```

5. **Explain in English what a closure is and why we need closures.** In particular, is there a property of the Scheme programming language that closures help us implement? You don't need a full essay here; I can answer this question in 2 sentences.

A closure is a triple that represents the value of a lambda expression. The three elements of a closure are its parameters, its body, and the environment at the time it is evaluated. We need closures to implement static binding.

6. In Lab 5 we made the following definitions for our environment datatypes:

```
(define empty-env (lambda () (list 'empty-env)))  
(define extended-env (lambda (syms vals old-env)  
  (list 'extended-env syms vals old-env)))
```

We also made recognizers `empty-env?` and `extended-env?` and for the `extended-env` type we made getters `syms`, `vals`, and `old-env`.

Using these tools, **write procedure (`lookup env symbol`)** that returns the value bound to `symbol` in the given environment, or the value `'oops` if `symbol` is not bound in the environment.

```
(define lookup  
  (lambda (env symbol)  
    (cond  
      [(empty-env? env) 'oops]  
      [else (find symbol (syms env) (vals env) (old-env env))])))
```

```
(define find  
  (lambda (symbol syms vals old)  
    (cond  
      [(null? syms) (lookup old symbol)]  
      [(eq? (car syms) symbol) (car vals)]  
      [else (find symbol (cdr syms) (cdr vals) old)])))
```

7. Here are 5 terms commonly used when we talk about programming languages. **Explain** (1 sentence is sufficient; use more if you must) **what each of them means:**

a. call-by-value

This is a procedure calling mechanism in which the values of the argument are passed to the procedure.

b. call-by-reference

This is a procedure calling mechanism in which references (such as the address) to the arguments are passed to the procedure.

c. call-by-name

This is a procedure calling mechanism in which the text of the arguments are passed to the procedure.

d. static binding

The values of free variables in a function body are found in the environment in place when the function is defined.

e. dynamic binding

The values of free variables in a function body are found in the environment in place when the function is called.

8. In class we produced the stream `Primes$` of prime numbers; you don't need to reproduce this. **Use this stream `Primes$` to produce the stream `TwinPrimes$`** that consists of pairs of primes that differ by 2. The first few elements of `TwinPrimes$` are (3 5) (5 7) (11 13) (17 19) and (29 31).

```
(define TwinPrimes$ (twinFilter Primes$))
```

```
(define TwinFilter  
  (lambda (s)  
    (cond  
      [(= (+ 2 (car$ s)) (car$ (cdr$ s)))  
        (cons$ (list (car$ s) (+ 2 (car$ s)))  
              (TwinFilter (cdr$ s)))]  
      [else (TwinFilter (cdr$ s))])))
```

9. Let's say that a "lop" is a list of pairs, where the first element is a symbol and the second element is a number, such as ((a 5) (b 3)). **Write a continuation-passing style function (increment-k sym amount lop k)** that increments the pair of lop whose first element is sym by the given amount. If there is no such pair, the pair (sym amount) is added to the list. k, of course, is the current continuation. For example
(increment-k 'b 3 '((a 4) (b 2) (c 5)) (lambda (x) x)) returns ((a 4) (b 5) (c 5)) while
(increment-k 'd 3 '((a 4) (b 2) (c 5)) (lambda (x) x)) returns '((a 4) (b 2) (c 5) (d 3))
You can assume that only one pair in a lop will start with any given symbol.

```
(define increment-k
  (lambda (sym amount lop k)
    (cond
      [(null? lop) (k (list sym amount))]
      [(eq? sym (caar lop)) (k (cons (list sym (+ amount (cadar lop))
                                     (cdr lop)))]
      [else (increment-k sym amount (cdr lop)
                          (lambda (t) (k (cons (car lop) t))))]))))
```

