CS 3 Final Review
Spring 2005

1. What does each of the following evaluate to in Scheme? If there is an error, write ERROR. For each which results in an error, what small change could you make to its arguments such that it does something reasonable?

a. `(se '(hello) 'there '())`
   ➔ `(hello there)`
b. `(list '(hello) 'there '())`
   ➔ `((hello) there ())`
c. `(append '(hello) 'there '())` ➔ ERROR
   `(append '(hello) '(there) '())` ➔ `(hello there)`
d. `((lambda (x y z) (cons x (cons y z))) 'z '(3) '())`
   ➔ `(z (3))`
e. `(member (quote (a)) (append (list '(a b c)) (quote (a))))`
   ➔ `#f`
f. `(apply map '(square (2 3 4)))` ➔ ERROR
   `(apply map (list square '(2 3 4)))` ➔ `(4 9 16)`
g. `(cadar (map reverse '(((1 2 3) 4) (5 6 7) (8 (9 10)))))`
   ➔ `(1 2 3)`
h. `(accumulate - (every square 1234))`
   ➔ `-10`

2. Write a procedure, `count-examples`, that inputs a sentence of words and returns the number of appearances of the phrase "for example" in the text.

```
(for example this sentence contains for example twice)
```

```
(define (count-examples s)
  (cond
     ((empty? s) 0)
     ((empty? (bf s)) 0)
     ((and (equal? (first s) 'for)
           (equal? (second s) 'example))
        (+ 1 (count-examples (bf (bf s)))))
     (else (count-examples (bf s)))))
```

Which recursive pattern does this procedure follow? Counting

3. Write a function, `list-average`, that uses higher-order functions to find the average of the numbers in a list that may contain anything.

```
(list-average '(1 2 6 7)) ➔ 4
(list-average '(1 2 abc (3 4) 6 7 #f)) ➔ 4
```

```
(define (list-average l)
   (/ (reduce + (filter number? l))
      (length (filter number? l))))
```

4. Describe what `mystery` does.

```
(define (mystery funct)
   (lambda (x) (if (funct x) 1 0)))
```

`mystery` returns a function that, given an input, returns 1 if applying that argument to `funct` results in a true value, and 0 if it does not. (Note that `funct` is the argument given to `mystery`).

b. Use `mystery` along with higher-order functions to write a procedure, `count-numbers`, that counts the number of items in a list that are numbers.

```
(define (count-numbers lst)
   (reduce + (map (mystery number?) lst)))
```

5. Sorting is a big deal in Computer Science. Earlier in the semester, we worked with an *insertion sort* procedure. It essentially starts with an empty sentence, and places each number from the input into its proper place until the input is empty. For reference, here is insertion sort:

```
(define (insert n l)
  (cond
    ((null? l) (list n))
    ((< n (car l)) (cons n l))
    (else (cons (car l) (insert n (cdr l))))))

(define (sort l)
  (if (null? l)
    l
    (insert (car l) (sort (cdr l)))))
```

Now we're going to work with a similar algorithm, called *selection sort*. It works, in a sense, opposite of insertion sort. It finds the smallest number in the input, removes it, places it in the output (after any other entries that have already been output), and repeats until the input is empty.

Suppose you are given a procedure called `remove`, that takes in a number and either a list or a sentence, removes the first occurrence of that number from that list/sentence, and returns the result. Now write `sort` according to the method described. You may use either lists or sentences, and you may use helper procedures.

```
(define (sort l)
   (if (null? l)
      l
      (cons (apply min l)    ;; you could use reduce too.
            (sort (remove (apply min l) l)))))

;; or

(define (sort s)
   (if (empty? s)
      s
      (se (accumulate min s)
          (sort (remove (accumulate min s) s)))))
```

6. Write a procedure, `deep-min`, that returns the least number in a general list that might consist of other lists.

```
(deep-min '((10 2 3) 5 (2 4 (1 8) 9) 4)) ➔ 1
```

```
(define (deep-min l)
   (cond
      ((null? l) X)
      ((list? (car l))
          (min (deep-min (car l)) (deep-min (cdr l))))
      (else
          (min (car l) (deep-min (cdr l))))))
```

Where `X` is some arbitrarily large number (or any number that is guaranteed to be larger than all numbers in your input). It will therefore never be returned by `min` (which is what we want).

Alternatively, `X` can be `#f`, and we can rewrite min:

```
(define (min x y)
   (cond
      ((not (and x y)) (or x y))
      ((< x y) x)
      (else y)))
```

The first case tests to see if either one of the inputs is `#f`. If so, then it returns the other input (`or` won't return `#f` unless both arguments are `#f`). The last two cases should be obvious.

7. Describe, qualitatively, what changes would be necessary to convert the *Difference Between Dates* code found in your reader to operate on a different calendar system, say a lunar calendar.

You must change the functions that deal with months, namely `days-preceding`. If the calendar alters the representation of dates (from `(month day)`), then you would also need to change the selectors `month-name` and `date-in-month`. If you were dealing with years (which the version in the reader does not do), then you would need to change any year calculations as well.

8. In a lab earlier this semester, we wrote a function that computed the n^{th} Fibonacci number. The Fibonacci sequence starts with two 1's, and each successive number is the sum of the previous two numbers in the sequence (so, it's 1 1 2 3 5 8 …). Here's the code for the function:

```
(define (fib n)
  (if (< n 2)
    1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

You may have noticed that it performed reasonably well for small inputs, but took a ridiculous amount of time (and slowed the lab machines to a halt) when you gave it a number larger than, say, 30.

a. Why does this happen?

It runs slowly because it uses a tree recursion that tries to recalculate every number from scratch, working all the way down the tree until it hits `(fib 0)` or `(fib 1)`. To better see this, let's consider the traces of `(fib 3)` and of `(fib 4)`:

```
STk> (fib 3)
.. -> fib with n = 3
.... -> fib with n = 2
...... -> fib with n = 1
...... <- fib returns 1
...... -> fib with n = 0
...... <- fib returns 1
.... <- fib returns 2
.... -> fib with n = 1
.... <- fib returns 1
.. <- fib returns 3
3
STk> (fib 4)
.. -> fib with n = 4
.... -> fib with n = 3
...... -> fib with n = 2
........ -> fib with n = 1
........ <- fib returns 1
........ -> fib with n = 0
........ <- fib returns 1
...... <- fib returns 2
...... -> fib with n = 1
...... <- fib returns 1
.... <- fib returns 3
.... -> fib with n = 2
...... -> fib with n = 1
```

```
...... <- fib returns 1
...... -> fib with n = 0
...... <- fib returns 1
.... <- fib returns 2
.. <- fib returns 5
5
```

Then, to compute `(fib 5)`, we must perform all of the work done by `(fib 4)`, plus all of the work done by `(fib 3)`. We can see that by adding 1 to the input, we have to do almost twice as much work! Since the amount of work (and therefore the amount of time it takes to compute) nearly doubles when the input goes up by 1, the amount of work required to compute `(fib 25)` gets to be rather ridiculous.

(Side note: it actually takes closer to 1.61x the work to compute the next Fibonacci number. Do you know why? Regardless, the function is exponentially complex, taking on the order of 1.61^n units of time to compute the n^{th} Fibonacci number).

b. We would like to improve on the performance of our program. Write a procedure that does this. You may use helpers. (Hint: compute the 10th Fibonacci number by hand. How did you do it differently from how the function listed above does it? Hint #2: Use an accumulating recursion)

The way we would compute `(fib 10)` is by writing out the sequence starting with the first two numbers. To get the next number, we simply look back at the last two we just wrote down, and add them together. At any step, we only need to know the last two numbers. Therefore, we write an accumulating recursion that keeps track of (at least) the last two numbers that were computed.

Here is one that just generates the Fibonacci sequence until it has enough terms:

```
(define (fib n)
   (if (< n 2)
      1
      (fib-help n '(1 1))))

(define (fib-help n s)
   (if (= n (- (count s) 1)) ; -1 needed because we start
      (last s)               ; counting the sequence from 0
      (fib-help n
                (se s (+ (last s) (last (bl s)))))))
```

It's easy to see that this one fares better:

```
STk> (fib 5)
.. -> fib-help with n = 5,  s = (1 1)
.... -> fib-help with n = 5,  s = (1 1 2)
...... -> fib-help with n = 5,  s = (1 1 2 3)
........ -> fib-help with n = 5,  s = (1 1 2 3 5)
.......... -> fib-help with n = 5,  s = (1 1 2 3 5 8)
.......... <- fib-help returns 8
........ <- fib-help returns 8
...... <- fib-help returns 8
.... <- fib-help returns 8
.. <- fib-help returns 8
8
```

Another variation keeps track of only the last two numbers that it's generated:

```
(define (fib n)
   (if (< n 2)
     1
     (fib-helper (- n 2) 1 1)))

(define (fib-helper n prev pre-prev)
   (if (zero? n)
     (+ prev pre-prev)
     (fib-helper (- n 1) (+ prev pre-prev) prev)))

STk> (fib 5)
.. -> fib-helper with n = 3,  prev = 1,  pre-prev = 1
.... -> fib-helper with n = 2,  prev = 2,  pre-prev = 1
...... -> fib-helper with n = 1,  prev = 3,  pre-prev = 2
........ -> fib-helper with n = 0,  prev = 5,  pre-prev = 3
........ <- fib-helper returns 8
...... <- fib-helper returns 8
.... <- fib-helper returns 8
.. <- fib-helper returns 8
8
```

(Another Side Note: Either of these functions can compute the 150th Fibonacci number without breaking a sweat, while the first function listed would take well over a trillion years!)