Read and fill in this page now

Your name: _____

Your login name (e.g., cs3-aa): _____

Your lab section day and time: _____

Your lab T.A.: _____

Name of the person sitting to your left: _____

Name of the person sitting to your right: _____

Prob 0: _____ Prob 1: _____ Prob 2: _____

Prob 3: _____ _____ _____ _____

Prob 4: _____ Prob 5: _____ _____

Subtotal: _____/32 Scaled Total: _____/28

You have 50 minutes to finish this test, which should be reasonable; there will be approximately 20 minutes of leeway given past one hour. Your exam should contain six problems (numbered 0-5) on 7 total pages. Note that the problems at the end of this exam may take longer than those at the front!

This is an open-book test. You may consult any books, notes, or other paper-based inanimate objects available to you. Read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

Restrict yourself to scheme constructs that we have seen in this classs. (Basically, this excludes chapters 16 and up in Simply Scheme).

Please write your answers in the spaces provided in the test; if you need to use the back of a page make sure to clearly tell us so on the front of the page. We believe we have provided more than enough space for your answers, so please don't try to fill it all up.

Partial credit will be awarded where we can, so do try to answer each question.

Relax!

Problem 0. (1 point)

Put your login name on each page. Also make sure you have provided the information requested on the first page.

Problem 1. Left-handed accumulate (4 points)

Consider the higher order procedure `l2r-accumulate`, which works much like the `accumulate` that you are familiar with. The main difference, as you might expect, is that `l2r-accumulate` starts at the left of its input sentence, moving to the right.

`(l2r-accumulate + '(1 2 3 4))`	➔	10
`(l2r-accumulate – '(1 2 3 4))`	➔	-8
`(l2r-accumulate word '(abra ca da bra))`	➔	abracadabra

Part A: Write `l2r-accumulate`. You can assume that it will only have to operate on sentences, rather than words. To make things easier, you can assume that the procedure that `l2r-accumulate` is given as the first parameter will only return words, rather than possibly sentences.

Problem 2. Certifying procedures (4 points)

`Every` can only take procedures (as its first argument) that return either a word or a sentence. Write the procedure `certify` which will ensure that the procedure given to an every will not cause an error due to its return value. `Certify` is used like this:

```
(every (certify a-proc) '(a sentence to be mapped))
(every (certify a-math-proc) '(1 2 3 4 5 6 7 8 9))
```

If the procedure passed to `certify` won't return a word or sentence, make it so that the word `"output-error"` is returned instead. For instance:

`(every number? '(friends 4 evah))`	➔	*ERROR*
`(every (certify number?) '(friends 4 evah))`	➔	`("certify-error" "certify-error" "certify-error")`

Problem 3. (Re)cursing the tree (1/2/1/6 points)

This question concerns a data representation for a tree (i.e., the thing that grows in the ground, has a trunk, etc.). The representation of a tree is a sentence of branches, where each branch is a word. The first branch in the sentence is the trunk.

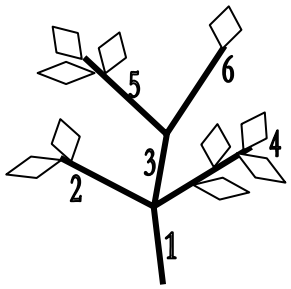
Each branch either has a certain number of leaves and no other branches coming off of it (called an "end-branch"), *or* has no leaves but connects to some number of other sub-branches. The other sub-branches are also contained in the tree data-structure.

An end-branch is represented by a word that starts with an `"e"` and ends with a number, which is the number of leaves on the end-branch. For instance, `"e12"` is a end-branch that has 12 leaves.

A non-end-branch starts with an `"x"`, and ends with a series of branch positions in the tree sentence, with each position separated by a `"-"`. For instance, the branch `"x3-4-5"` has three sub-branches which reside at position 3, 4, and 5 in the tree-sentence respectively.

Here is a "tree" that has 6 branches, including the trunk:

```
(x2-3-4 e2 x5-6 e4 e3 e1)
```



Part A: Write the predicate that tests whether a branch is an end-branch. Name this procedure and its parameter(s) properly.

Part B: Write the selector procedure that returns the number of leaves on a branch. Name the procedure and its parameter properly.

Part C: Write the selector procedure that returns the trunk for a given tree. Name the procedure and its parameter(s) properly.

Part D: Write `count-all-leaves`

Assume that the procedure `sub-branches` has been written, which takes two arguments: a branch and a tree that contains it. `sub-branches` returns the sentence containing the sub-branches connected to the branch given as the first argument.

`(sub-branches 'x2-3-4 '(x2-3-4 e2 x5-6 e4 e3 e1))`	➔	`(e2 x5-6 e4)`
`(sub-branches 'e3 '(x2-3-4 e2 x5-6 e4 e3 e1))`	➔	`()`

Fill in the blanks on the procedures below, such that the procedure `count-all-leaves` will return the total number of leaves on a tree (You only need to count leaves on end-branches). Note that you **must** use only the selectors and predicates that you have defined earlier, and `sub-branches,` when accessing a branch or tree. **For instance, using a higher-order function to access a tree as a sentence is a data abstraction violation**!

```
(define (count-all-leaves tree)

   _____________________________________________________


(define (count-all-leaves-helper branch tree)
   (if (end-branch? branch)

        _____________________________;end-branch base case

      (accumulate                    ;recursive case

         ____________________________________

         (every

              ____________________________________________

              ____________________________________________

              ))))
```

Problem 4. Rewrite recursion using higher-order functions. (6 points)

Consider the following procedure:

```
(define (recurring-mystery sent cut)
   (cond ((empty? sent)
          "")
         ((special? (first sent))
          (word (first sent)
                cut
                (recurring-mystery (bf sent) cut)))
         (else
          (recurring-mystery (bf sent) cut))))
```

(For this procedure to run without errors, it may need additional procedures to be defined first).

Write the procedure `hoffing-mystery`, without using any explicit recursion, such that it will return the same values as `recurring-mystery`, given the same input.

Problem 5. Rewrite a higher-order function as recursion (2/5 points)

Consider the following function:

```
(define (hof-of-horror sent)
   (every (lambda (v)
            (every (lambda (nv) (word v nv))
                   (keep (lambda (wd) (not (vowel? wd)))
                         sent)) )
          (keep vowel? sent)))
```

Part (A)

What does `(hof-of-horror '(a b c d e))` return?

Part (B)

Write the function `recursion-of-horror` (which you can abbreviate as `roh`) so that it will return the same values as `hof-of-horror`, given the same inputs. For `roh` and helpers, do not use `keep`, `every`, `accumulate`, `repeated`, or `lambda` forms.