CS 61A, Summer 2000
Midterm #1 (Sole midterm)
Professor Allison Coates

Problem #1 (5 points):

Give the commands necessary to perform the following tasks.

(a) Emacs

1. Load the file `~cs61a/lib/foo.scm` into an Emacs buffer.

2. Start a Scheme buffer and load `~cs61a/lib/foo.scm` into that buffer.

(b) Unix

1. Change into your home directory. Confirm you are in your home directory

2. Create the directory `sols`. Copy all the files from `~cs61a/sol` into `sols`.

3. Confirm the files are in the `sols` directory.

Problem #2 (10 points):

What will Scheme print in response to the following expressions? If an expression produces an error message or runs forever without producing a result, you may just say "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just say "procedure"; you don't have to show the form in which Scheme prints procedures. Assume that no global variables have been defined before entering these expressions (other than the predefined Scheme primitives), except `a` and `b`

```
(define a 'goodbye) (define b 'hello)

(let ((a 'hi)
      (b a))
  (list a b))

(let ((a 'hi)
```

```
  (list a (let ((b a)) b)))

(((lambda (one)
     (lambda (two)
        (list one two 'three))) 'a) 'b)

(let ((x 3)) (lambda () x))
```

Given each Scheme printout:

(a) Draw a box and pointer diagram representing the structure.

(b) Give an expression using only `cons` and `list` that would create the structure.

(c) Provide a scheme expression to retrieve the number `3` from this structure.

```
> foo
((1 . 2)((3) . 4) 5)

> bar
(1 (2 3) 4 ((5)))
```

Problem #3 (10 points):

For each of the following statements, indicate whether the statement is true or false by circling the appropriate response. Also, explain in one gramatically [sic] correct Enlish [sic] sentence why each statement is true or false.

(a) A function that returns a pair containing one answer in the car and one answer in the `cdr` is non-functional because it's returning two answers. **true / false**

(b) In general, when adding new functions to existing objects, message-passing style is more convenient than generic operations with explicit dispatch. **true / false**

(c) In general, an iterative process and recursive processes for the same algorithm share the same order of growth with respect to time complexity. **true / false**

(d) Given the following definitions:

```
(define (mystery1 n)
```

```
  (- n 100))
(define (mystery2 n)
  (+ n n))
(define (mystery3 n m)
  (+ n m))
```

Exactly two of the following expressions perform differently under normal order than under applicative order. **true / false**

```
> (mystery1 (rand 23))
> (mystery2 (rand 56))
> (mystery3 (rand 4) (rand 4))
```

Problem #4 (4 points):

Write a function `palindrome?` that takes a `word` and returns whether it is spelled the same backwards as forwards. (Note `""` is the empty word.)

```
> (palindrome? 'radar)
#t
> (palindrome? 'hellowthere)
#f
> (palindrome? 'amastergermregretsama)
#t
> (palindrome? "")
#t
```

Problem #5 (points):

Welcome to the University: a magical land of mystery and wonder. In the University live `students`, who are identified by their department and SID. The constructor `make-student` takes a department and an SID and creates a new student:

```
(define (make-student dept sid)  (attach-tag dept sid))
(define (getdept student)        (type-tag student))
(define (getsid student)         (contents student))

> (define rob (make-student 'psych 14000000))
rob
> (define lai (make-student 'cs 13000000))
lai
```

In the University also live a magical sect of people, the `professors`. The `professors` perform examples for all `students` who possess the correct department.

A `professor` is a procedure that takes three arguments:

1. a `student` asking for help
2. a procedure of one argument to perform
3. a list of examples to be computed

Professors return either a list of solutions or `'sorry` if the student is in the wrong department. The constructor `make-professor` takes a department and creates a new `professors`:

```
> (define cs-professor (make-professor 'cs))
cs-professor
> (cs-professor lai (lambda (x) (+ x 1)) '(2 3 4 5))
(3 4 5 6)
> (cs-professor lai square '(1 2 3 4))
(1 4 9 16)
> (cs-professor rob square '(1 2 3 4))
sorry
```

Write the procedure `make-professor`.

Problem #6 (points):

Sometimes when you download a file from the World Wide Web, it will come in compressed form. When you download such a file, you will have to decompress it if you want to make use of it.

We could also compress and decompress lists. For example, a list such as:

```
(vertigo vertigo psycho rearwindow rearwindow psycho)
```

is compressed into:

```
((2 vertigo) (1 psycho) (2 rearwindow) (1 psycho))
```

Each element of the compressed list is called a `run`. `(2 vertigo)` and `(1 psycho)` are examples of `runs`. Each `run` tells us how many times in a row a particular symbol appeared in the original list.

A `run` is a new abstract data type that we are inventing for this problem. A `run` consists of a symbol and a count of the number of times it appears consecutively. The constructor for a run is called

`makerun` and the selectors are `getsymbol` and `getcount`.

Here is how we could create a compressed list:

```
> (define mylist (list (makerun 'vertigo 2)
                       (makerun 'psycho 1)
                       (makerun 'rearwindow 2)
                       (makerun 'psycho 1)))
mylist
> mylist
((2 vertigo) (1 psycho) (2 rearwindow) (1 psycho))
```

(a) Define the following constructors and selectors for `runs`:

```
makerun     ; takes a symbol and a count, and returns a new run

getsymbol   ; takes a run and returns the symbol

getcount    ; takes a run and returns the count
```

(b) Write a function `decompress` that **decompresses** a compressed `list`. **Use your constructor and selectors from part (a) where appropriate**.

```
> (decompress mylist)
(vertigo vertigo psycho rearwindow rearwindow psycho)
```

(c) Suppose that we decide to change the way that a compressed list is represented. If a symbol, say `psycho`, only appears once, then it may be redundant to represent that run as `(1 psycho)`. Instead, we want to be even more economical, and represent it simply using the symbol `psycho` all by itself.

Your task is to modify the constructor and selectors from part (a) such that they are consistent with this new representation.

```
> (define mylist (list (makerun 'vertigo 2)
                       (makerun 'psycho 1)
                       (makerun 'rearwindow 2)
                       (makerun 'psycho 1)))
mylist
> mylist
((2 vertigo) psycho (2 rearwindow) psycho)

makerun     ; takes a symbol and a count, and returns a new run
```

```
getsymbol   ; takes a run and returns the symbol

getcount    ; takes a run and returns the count
```

If the `decompress` function you wrote in part (b) does not work with this new representation, you should return to part (b) and modify it so that it will work with both representations.

Problem #7 (points):

You've been abducted to the planet Smoo and asked to rescue Prince Froptoppit from the **killer trees**!

The killer trees of Smoo are implemented as list structures. To save the prince, you must search through the list structure and find his **depth** in the tree.

An element's depth is defined as the length of the path from the root to the element. For example:

```
      +
     / \
    +   d   ; depth 1
   / \
  +   c     ; depth 2
 / \
a   b       ; depth 3
```

Write a function `find` that will take a list structure and return either the depth of '`prince` in the structure or `#F` if `'prince` is not in the structure. You may assume that '`prince` does not appear more than once in the list structure.

```
> (find '(akiko (gax) (spuck (beeba))))
#F
> (find '(monster prince monster))
1
> (find '(monster akiko (spuck monster monster prince)))
2
> (find '(alia (poog (monster monster prince)) monster))
3
```
