CS61A, Spring 1995
Midterm #1

Question 1 (3 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, 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.

```
(first (butfirst '(yesterday)))

((lambda (x) x) (lambda (x) x))

((lambda (w) (sentence (word 'h w) (word 'th w))) 'ere)

(+ (* 3 5 0 7) (- 8 2))

(and (> 2 3) (/ 5 0))

(let ((me 'you)
      (you 'me))
  (sentence 'you 'love me))
```

Question 2 (3 points):

Consider the following code.

```
(define (all-vowels? wd)
  (cond ((empty? wd) #t)
        ((vowel? (first wd)) (all-vowels? (bf wd)))
        (else #f)))

(define (keep-all-vowels sent)
  (cond ((empty? sent) '())
        ((all-vowels? (first sent))
         (se (first sent) (keep-all-vowels (bf sent))))
        (else (keep-all-vowels (bf sent)))))
```

List all of the calls to `all-vowels?`, including recursive calls, during the evaluation of the following expression:

```
(keep-all-vowels '(eva ai xxx a))
```

Question 3 (5 points):

Recently the 61A staff has been playing with the ``four fours'' problem: How many different numbers can we make by combining four fours with various arithmetic operators? For example, we can make 4 * (4+4) - 4 which is 28. It turns out that some operators can produce very large results. For example, 4^{4^4} is a number with 155 digits, and 4^{4^{4^4}} is too large for scm to compute. It's therefore useful to write ``safe'' versions of the arithmetic operators, like this:

```
(define (safe-expt base power)
  (cond ((> base 30) #f)
        ((> power 8) #f)
        (else (expt base power))))
```

The trouble with this is that we can't use `safe-expt` to provide an argument to some other arithmetic function, as in this expression:

```
(+ (safe-expt 4 (safe-expt 4 4)) 4)
```

because the `+` procedure will complain if given `#f` as an argument. To make this system work, we need `false-ok` versions of all the arithmetic operators:

```
> (false-ok-+ 4 4)
8
> (false-ok-+ 4 #f)
#f
```

Your job is to write `make-false-ok`, a higher order procedure that takes as its argument an arithmetic operator like `+`, returning a version that checks its arguments for falsehood. If either argument is false, the new version should return false; if not, it should invoke the original operator. So we should be able to say

```
(define false-ok-+ (make-false-ok +))
```

You may assume that the argument to `make-false-ok` is a function of two arguments.

Read the problem again. Don't write `false-ok-+`!

Question 4 (3 points):

Given below is a solution to the change counting lab exercise

```
(define (cc amount coin-sent)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (empty? coin-sent)) 0)
        (else (+ (cc (- amount (first coin-sent)) coin-sent)
                 (cc amount (bf coin-sent)) ))))
```

Fill in the blanks in the framework below to produce a procedure that returns `#t` or `#f` according to whether change can be made for the given amount using the given coins.

```
(define (change-possible? amount coin-sent)

  (cond ((= amount 0) ________ )

        ((or (< amount 0) (empty? coin-sent)) ________ )

        (else (________ (change-possible? (- amount (first coin-sent))

                                                   coin-sent)
```

```
                (change-possible? amount (bf coin-sent)) ))))
```

Question 5 (5 points):

Write a predicate `two-twice?` that takes a word as its argument. It should return `#t` if and only if there is some sequence of two letters that appears twice in the word:

```
> (two-twice? 'banana)
#t
> (two-twice? 'alabaster)
#f
> (two-twice? 'mississippi)
#t
> (two-twice? 'decided)
#t
> (two-twice? 'laxxxor)
#t
```

Notice, in the last example above, that the two pairs overlap (`xx` twice).

You may use the following procedure to help:

```
(define (second wd)
  (first (butfirst wd)))
```

Take a peek at the solutions