CS61a, Spring 1999
Midterm #1

(Total: 20 points)

Question #1 [3 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).

```
(word 'for '(no one))
```

```
(if (first 'flying) 'yes 'no)
```

```
(every (lambda (w) (last (butlast w))) '(think for yourself))
```

```
(let ((first last)
      (last first))
   (first (last '(tomorrow never knows))))
```

```
((lambda (a b c) (word b a c)) '(i want you))
```

```
((lambda (a) (a 3)) (lambda (x) (word x x)))
```

Question #2 [4 points]

Write a procedure n-to-nth that takes a positive integer *n* as its argument, and returns the value of *n*^*n* (*n* to the *n*th power), **computed by multiplying *n* by itself *n* times**. Do not use exp, expt, or other mathematical primitives besides +, -, *, and /.

Question #3 [4 points]

Write the procedure *select* that takes four arguments: a predicate function of one argument, and three sentences, which will all be of equal length. (Reminder: Don't check for errors in the arguments.) *Select* must return a sentence of the same length as the arguments sentences, in which each word is chosen from the third argument or from the fourth argument depending on whether the predicate, applied to the corresponding word of the second argument, returns true or false. For example:

```
> (select even? '(5 14 32 9) '(she said she said) '(eight days a week))
(EIGHT SAID SHE WEEK)

> (select vowel?
          '(b i r c h)
          '(i saw her standing there)
          '(you like me too much))
(YOU SAW ME TOO MUCH)

(Historical note: Select is a primitive procedure in th
e APL programming language.)
```

Question #4 [4 points]

Consider the following procedures:

```
(define (cross a b)
        (if (empty? a)
            '()
            (se (cross1 (first a) b)
                (cross (bf a) b))))

(define (cross1 letter wd)
        (if (empty? wd)
            '()
            (se (word letter (first wd))
                (cross1 letter (bf wd)))))
```

(a) What is the value of *(cross 'get 'back)*?

(b) How many calls to cross1 are made in computing the result to part (a)?

(c) Does *cross* generate an iterative or a recursive process?

(d) Does *cross1* generate an iterative or a recursive process?

Question #5 [4 points]

In lecture you saw the example

```
(define (make-adder num)
        (lambda (x) (+ num x)))
```

We want to generalize this example, so that we can do to any two-argument procedure what **make-adder** does to +. You will write a procedure **maker-maker** for this purpose. Here are some examples of how it should work:

```
> (define (make-adder (maker-maker +))
> ((make-adder 3) 5)
8

> (define (make-subtracter (maker-maker -))
> (define ten-minus (make-subtracter 10))
> (ten-minus 3)
7

> (define make-sentencer (maker-ma
ker se))
> (define infinitive (make-sentencer 'to))
> (infinitive 'play)
(TO PLAY)
```

More generally, **maker-maker** takes as its argument a function of two arguments. It should return a **make-adder**-like function of one argument, such as **make-subtracter** or **make-sentencer** above. When that function is called, it returns another one-argument function such as **ten-minus** or **infinitive** above. When that new function is called, it call the original two-argument function with the remembered argument to **make-**whatever, and its own argument, as the two arguments.

(That's a complicated paragraph, but the function you have to write is pretty simple, once you understand it!)