CSCI 5582 Problem Set 1
Due: Sept. 22, 1997

Problem 1. (15 points)
Euclid's algorithm for finding the greatest common divisor of two positive integers m and n (assume m >= n) can be expressed as follows:

If m is divisible by n with no remainder, then the greatest common divisor is n.

Otherwise, find the remainder r that is left over when m is divided by n. Now find the greatest common divisor of n and r.

Write a recursive Scheme procedure, `euclid-gcd`, that takes two positive integers as arguments (assume that the first is greater than or equal to the second), and returns the greatest common divisor of the two integers. You will find the Scheme primitive remainder helpful: this takes two integer arguments and returns the remainder when the first is divided by the second.

Here are some examples of how the working procedure should behave:

```
>>> (euclid-gcd 20 4)
4

>>> (euclid-gcd 20 3)
1

>>> (euclid-gcd 35 14)
7
```

Problem 2. (20 points)
Consider the following peg-jumping game. There are 7 holes placed in a row. The starting position of the game has three black pegs in the three leftmost holes and three white pegs in the three rightmost holes (the center hole is empty). We could represent the starting position, then, as follows:

B B B _ W W W

A legal move is as follows:

a. A peg can move into an empty space on either side of it.
b. A peg can jump over a peg of an opposite color into the empty space.

Thus, from the starting position, there are two possible following positions:

B B _ B W W W
and
B B B W _ W W

From the first of these possibilities, the following moves are possible:

B _ B B W W W
B B B _ W W W
B B W B _ W W

Your job is to write a Scheme procedure named `find-peg-moves`. This procedure should take as its argument a position of the game (expressed as a list of 7 symbols: three B's, three W's, and a dash) and return a list of legal positions of the game following one move. Here are a couple of examples:

```
>>> (find-peg-moves '(b b b _ w w w))
((b b _ b w w w) (b b b w _ w w))
>>> (find-peg-moves '(b b _  b w w w))
((b _ b b w w w) (b b b _ w w w) (b b w b _ w w))
```

Problem 3. (20 points)
Write a procedure named `anagrams` that takes a list of symbols (we'll assume that these are single letters) as its argument, and returns a list of all distinct anagrams of the list as its result. Here are two examples of the procedure at work:

```
>>> (anagrams '(a n n))
((a n n) (n a n) (n n a))

>>> (anagrams '(n o o n))
((o o n n) (o n n o) (o n o n) (n n o o) (n o n o) (n o o n))
```

Problem 4. (15 points)
Write a procedure named `apply-till-predicate-is-met`. This procedure should take four arguments: a numerical function f, a numerical predicate pred, a starting value val, and a limiting iteration value limit. The procedure should call f on val as many times as necessary until the predicate is met, or until the number of calls has reached the limit value. In other words, our procedure looks at the sequence of values:

```
val
(f val)
(f (f val))
```

and so forth, until the predicate is true. When the predicate is met, the procedure should return the value for which the predicate is met, and the number of iterations of f required to meet the predicate (or the final achieved value and limit, if the predicate was never met). Here are two examples—in the first case, we needed 14 iterations of the square root procedure with a starting value of 5 to achieve a value within 0.0001 of 1. In the second case, after 20 iterations, we never saw an odd number:

```
>>> (apply-till-predicate-is-met sqrt
       (lambda (val) (< val 1.0001)) 5 20)
(1.00009823711941 14)

>>> (apply-till-predicate-is-met
       (lambda (x) (+ 2 x))
       (lambda (val) (not (even? val)))
       0 20)
(40 20)
```

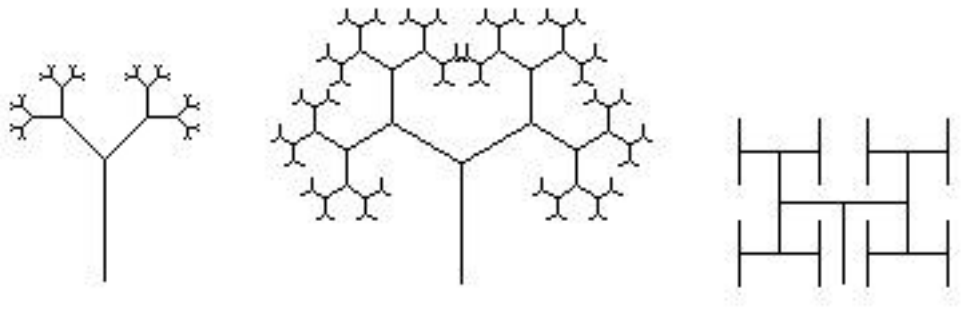
Problem 5. (15 points)
Write a procedure named `make-n-element-tester`. This procedure takes two arguments: a predicate pred and a number n. It returns a procedure which, when called on a list, will return #t if and only if the list contains at least n elements for which the predicate is true. Here are some examples of our procedure at work:

```
>>> (define at-least-three-evens?
      (make-n-element-tester even? 3))
at-least-three-evens?
>>> (define at-least-one-zero?
       (make-n-element-tester
         (lambda (x) (= x 0)) 1))
at-least-one-zero?
>>> (at-least-one-zero? '())
#f
>>> (at-least-one-zero? '(0))
#t
>>> (at-least-one-zero? '(1 2 3))
#f
>>> (at-least-three-evens? '(1 2 3 4 4 6))
#t
>>> (at-least-three-evens? '(2 4 6 8))
#t
```

Do just *one* of problem 6a or 6b.

Problem 6a. (15 points)

Modify the branch-making procedure shown in class so that it takes four arguments: a stem-length stem, a level-count level, a branch-angle theta, and a shrinking-ratio ratio. When we call our procedure with a level argument of 0, we get a straight line of length stem. Otherwise, we should see a straight line of length stem with two smaller trees (each smaller by a factor of ratio), branching off to the left and right from the vertical stem with angle theta. Here are some examples:



These three trees are produced by the following three calls to our new branch procedure.

```
(branch 30 5 45 2)
(branch 30 6 60 1.5)
(branch 20 4 90 1.25)
```

For instance, the middle tree is a six-level tree in which each sub-tree has a stem of length two-thirds the size of the previous-level stem, and in which the two subtrees branch off from the stem at an angle of 60 degrees.

Problem 6b. (15 points)
The following sequence of numbers is due to a puzzle by John Conway (the inventor of the "Game of Life"):

1
1 1
2 1
1 2 1 1
1 1 1 2 2 1
3 1 2 2 1 1
...

The rule for generating each new number is to read the previous one aloud, starting with 1. Thus, our numbers can be interpreted as follows:

```
1
1 1             "One one." (Read the previous value aloud!)
2 1             "Two ones."
1 2 1 1 "One two, one one."
1 1 1 2 1 1     "One one, one two, two ones."
```

Write a Scheme procedure named nth-conway-number that takes an argument n, and returns the nth element in this sequence. (The "zeroth" element is the list (1), the first element is the list (1 1), and so forth.)

Here's are some examples:

```
>>> (nth-conway-number 0)
(1)
>>> (nth-conway-number 1)
(1 1)
>>> (nth-conway-number 2)
(2 1)
>>> (nth-conway-number 3)
(1 2 1 1)
>>> (nth-conway-number 6)
(1 3 1 1 2 2 2 1)
```