All of the little functions that you write for this homework should be defined in the same source file, hw1.scm. Each question is worth 2 points, for a total of 20.

1. Write a predicate function `multiple?` that takes two integer arguments `x` and `y` and returns #t if `x` is an exact multiple of `y`, #f if `y` is 0 or `x` is not an exact multiple of `y`. You can assume that the arguments are in fact integers.

2. Write a simple function `area-of-triangle` that takes two numeric arguments `base` and `height` and calculates the area of the triangle. The function should always return an inexact number. You can assume that the arguments are in fact positive numbers.

3. An equation of the form Ax+By+C=0 in which A and B are not both zero has a straight line as its graph. The perpendicular distance from a point (x,y) to the line is calculated with $dist = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}}$. Write a simple function `dist-to-line` that takes 5 numeric input arguments `a`, `b`, `c`, `x`, and `y`, and returns the distance from the point to the line. You can assume that the arguments are in fact numeric and that A and B are not both zero.

4. Write a simple procedure `sosbig` that takes three numeric arguments `a`, `b`, and `c` and returns the sum of the squares of the two larger numbers. You can use a helper function `sum-of-squares` to simplify your code, but it should be local to `sosbig`, not a global definition.

5. In class I talked about how combinations are evaluated. Describe what this procedure does, and explain why it works the way it does.

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

6. Write a procedure `power-up` that takes as arguments two positive integers `b` and `n`, and returns the smallest power of `b` that is greater than `n`. That is, it should return the smallest integer *e* such that $b^e > n$. You may use the Scheme procedure `expt`, which raises a given number to a given power. You can assume that `b` is greater than 1. There is an iterative solution to this question. If you define a helper function, it should be local to `power-up`, not a global definition.

7. The following pattern of numbers is called Pascal's triangle.

```
            1
         1     1
      1     2     1
   1     3     3     1
1     4     6     4     1
            …
```

The numbers at the left and right edges of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure `pascal` that takes two integer parameters `row` and `col` and returns the value at that point in the triangle. Your procedure should implement a recursive process. Row and column numbers start at 0. You can assume that the inputs are valid locations, that is, both `row` and `col` are ≥ 0 and `col` ≤ `row`.

8. Write a procedure `clamp` that takes three numeric arguments `p`, `a`, and `b`, and returns a value that is clamped to the interval [a,b]. That is: $clamp = \begin{cases} a, p < a \\ p, a \le p \le b \\ b, p > b \end{cases}$.

You can assume that the arguments are in fact numeric and that `a` ≤ `b`.

9. In lecture I gave an example of a square root procedure. There is a very similar procedure for cube roots based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value $y_{new} = \dfrac{\frac{x}{y^2} + 2y}{3}$. Using this formula, write a procedure `root3` that takes a single numeric value `x` and returns an approximation to the cube root of `x`. You can start from the code in `sqrtd2` given in lecture. You can assume that `x` is in fact a numeric value.

10. Write a procedure `sum-powers` that takes two integer arguments `b` and `n` and returns the sum $\sum_0^n b^i = b^0 + b^1 + b^2 + \cdots + b^n$. Your procedure can be either linear recursive or iterative.

Note that there is a closed form summation formula for this: $\sum_0^n b^i = \dfrac{b^{n+1} - 1}{b - 1}$, but your function should calculate the value by adding up each of the individual b^i values. You can assume that both b and n are positive integer values.