CSE 1729 - Introduction to Principles of Programming Due September 17, 2018

Problem Set 2

For this assignment, you will submit 1 document, a Racket file, in Mimir. You can (and should) submit multiple times as you are working, but your grade will be based on your final submission. Be aware that the test set in Mimir may not be complete until later in the week. You will use the same labeling convention as in lab 2.

This convention for problem sets is as follows: for questions that ask you to write code, identify which problem it is, and for each test case put in a string identifying the test case. For problems that ask a question, put in a string identifying the question, then put your answer in successive strings, 1 per line. Here is a simple example: question 3a asks you to write a square function in SCHEME 3b asks you whether the square function always returns a value greater than zero. What you put in the Racket definitions will be something like the following:

```
"problem 3a."
(define (square x)
  (* x x))
"tests"
"(square 4)"
(square 4)
"(square -5)"
(square -5)
"(square (square 3))"
(square (square 3))
"problem 3b."
"If you give the square function a zero, the result will be zero."
"Also, if you give it a complex number the result might be negative."
```

What you (and we) will see in the lower window when you hit the run button is:

```
"problem 3a."
"tests"
"(square 4)"
16
"(square -5)"
25
"(square (square 3))"
81
"problem 3b."
"If you give the square function a zero, the result will be zero."
"Also, if you give it a complex number the result might be negative."
```

Now it is easy to see what your test results are. Your definitions should be in the same order as in the assignment, with tests within each code question, and answers in strings so easy to read in interactions pane in Racket.

Remember: code questions should always include illustrative examples.

1. Recall from class the definition of number-sum, which computes the sum of the first n numbers:

   ```
   (define (number-sum n)
     (if (= n 0)
         0
         (+ n (number-sum (- n 1)))))
   ```

   (a) Adapt the function to one named (odd-sum n) that computes the sum of the first n odd numbers. (So (odd-sum 4) should return 16, the sum of the first 4 odd numbers: $1+3+5+7$.)

   (b) Evaluate your function at 1, 2, 3, 4, 5, 6, and 7. What does this sequence of numbers look like, and does that make sense?

   (c) Adapt the function into (sum-from-to a b) that it computes the sum of the all integers from a to b, so (sum-from-to 3 5) should evaluate to 12. If a is greater that b, it should evaluate to 0.

2. Write a recursive function, named (k-product k), that, given a positive integer $k > 1$, computes the product

$$\underbrace{\left(1-\frac{1}{2^2}\right)\left(1-\frac{1}{3^2}\right)\cdots\left(1-\frac{1}{k^2}\right)}_{k-1}.$$

   (Experiment with the results for some various values of k; this might suggest a simple non-recursive way to formulate this function.)
   Note: The under brace and $k-1$ indicate that there are $k-1$ terms in the product your function computes.

3. (a) Consider the *Babylonian method* for computing square roots: given a (positive) number $x \geq 1$, the Babylonian method starts with a "guess" $s_0 = x/2$. It then improves this guess by the iterative rule

$$s_k = \begin{cases} \frac{x}{2} & \text{if } k = 0, \\ \frac{1}{2}\cdot\left(s_{k-1} + \frac{x}{s_{k-1}}\right) & \text{otherwise} \end{cases}$$

   Give a SCHEME function, named (babylonian x k), that computes roots using the Babylonian method and evaluates to the k^{th} approximation (s_k).
   Incidentally, the Bablylonian method converges quite quickly to the root: each subsequent guess has twice as many digits of accuracy as the previous guess.

   Remark 1. *Notice that the parameter* k *does not appear in the expression calculating the next approximation. It only serves as a counter of how many iterations of this algorithm to perform. Therefore, your function may take a parameter that either counts the number of recursive calls from* $1\ldots k$ *or* $k\ldots 1$, *as long as the number of iterations (function calls) is* k.

(b) We will now measure how large `k` needs to be in the above function to provide a good approximation of the square root. You will write a SCHEME function `(terms-needed x tol)` that will evaluate to the number of terms in the infinite sum needed to be within `tol`, that is, the smallest `k` such that the difference between `x` and `(square (babylonian x k))` is less than `tol`.

Remark 2. *At first glance, the problem of defining* `(terms-needed x tol)` *appears a little challenging, because it's not at all obvious how to express it in terms of a smaller problem. But you might consider writing a helper function* `(first-value-k-or-higher x tol k)` *that evaluates to* `k` *if* `(square (bablyonian x k))` *is within* `tol` *of the argument* `x`*, otherwise calls itself recursively with larger* `k`.

4. Recall from high-school trigonometry the cos function: If T is a right triangle whose hypotenuse has length 1 and interior angles x and $\pi/2 - x$, $\cos(x)$ denotes the length of the edge adjacent to the angle x (here x is measured in radians). You won't need any fancy trigonometry to solve this problem.

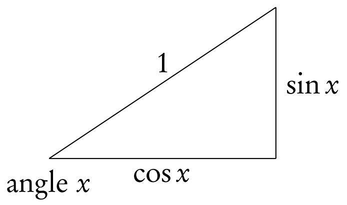

Figure 1: A right triangle.

It is a remarkable fact that for all real x,

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Write a SCHEME function `new-cos` so that `(new-cos x n)` returns the sum of the first $(n+1)$ terms of this power series evaluated at x. Specifically, `(new-cos x 3)`, should return

$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$

and, in general, `(new-cos x n)` should return

$$\sum_{k=0}^{n}(-1)^k \frac{x^{2k}}{(2k)!}.$$

You may use the built-in function `(expt x k)`, which returns x^k. It might make sense, also, to define `factorial` as a separate function for use inside your `new-cos` function. (Aesthetic hint: Note that the value $2k$ is used multiple times in the definition of the k^{th} term of this sum. Perhaps you can use a `let` statement to avoid computing this quantity more than once?)

Once you have finished:

1. Save your work to a file using the appropriate filename, problemSet2.rkt.
2. Submit your solution file for grading via the MIMIR site.