CSE 1729 - Introduction to Principles of Programming — Due: September 25, 2018

Problem Set 3

Honor code. As a student in 1729, you have pledged to uphold the **1729 honor code**. Specifically, you have pledged that any work you hand in for this assignment must represent your individual intellectual effort.

Remark: The Racket interpreter can maintain two different representations of numeric quantities: fractions and decimal representations. While fractions always represent exact numeric quantities, decimal expansions maintain a finite number of digits to the right of the decimal point. The Racket interpreter will attempt to infer, when dealing with numbers, whether to maintain them as fractions or as decimal expansions. For example

```
> (/ 1 2)
1/2
> (/ 1 2.0)
0.5
> (+ (/ 1 2) (/ 1 3) (/ 1 6))
1
> (+ (/ 1 2) (/ 1 3.0))
0.8333333333333333
>
```

In general, the interpreter will maintain exact expressions for numeric quantities "as long as possible," expressing them as fractions. You can instruct the interpreter that a number is to be treated as a decimal by including a decimal point: thus `1` is treated as an exact numeric quantity, whereas `1.0` is treated as a decimal expansion. The interpreter knows how to convert fractions to decimals (you can configure the number of digits of accuracy you wish), but will never convert decimals back to fractions (or integers). (So you know, this process is called *type-casting*.)

Arithmetic expressions like `(+ 1 1.0)` pose a problem because the two arguments are of different "types." In this case, the interpreter will transform the exact argument (`1`) into a decimal value, and then proceed as though all arguments were decimals (returning a decimal result). Other arithmetic operations are treated similarly.

1. Define $H_n = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$; these are referred to as the *harmonic numbers*. A remarkable fact about these numbers is that as n increases, they turn out to be very close to $\ln n$. ($\ln n$ is the *natural logarithm* of n.) In particular, as n increases the difference $|H_n - \ln n|$ converges to a constant (Euler's constant).
   (a) Show how to use Scheme to compute H_n by writing a Scheme function named `(harmonic n)`.
   (b) Using your `harmonic` function, write a function `(euler-approx n)` that gives an estimate of Euler's constant based on H_n. (You may wish to use the Scheme function `(log x)` which returns the natural logarithm of x.) So you know you are in the ballpark, Euler's constant is a little over a half.

2. A integer $n > 1$ is prime if its only positive divisors are 1 and n. (The convention is not to call 1 prime.) The following scheme procedure determines if a number is prime.

```
(define (prime? n)
  (define (divisor? k) (= 0 (modulo n k)))
  (define (divisors-upto k)
    (and (> k 1)
         (or (divisor? k) (divisors-upto (- k 1)))))
  (not (divisors-upto (- n 1))))
```

(So, it returns `#t` for prime numbers like 2, 3, 5, 7, and 11 and `#f` for composite (that is, non-prime) numbers like 4, 6, 8, and 9.) We will use this function to develop a function `(nth-prime n)` which evaluates to the n^{th} prime in the sequence $2, 3, 5, 7, \ldots$.

(a) The key to this problem is to break it down into smaller parts. First, write a function `(nth-prime-from n k)` which finds the nth prime starting from the number k. `(nth-prime-from 3 7)` should evaluate to 13, and `(nth-prime-from 2 6)` would evaluate to 11.

(b) Given `(nth-prime-from n k)` , write the function `nth-prime` so that `(nth-prime n)` returns the nth prime number. For example, `(nth-prime 1)` evaluates to 2, `(nth-prime 5)` evaluates to 11, and so forth. This is the easy part!

3. The Lucas numbers are a sequence of integers, named after Édouard Lucas, which are closely related to the Fibonacci sequence. In fact, they are defined in very much the same way:

$$L_n = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ L_{n-1} + L_{n-2} & \text{if } n > 1. \end{cases}$$

(a) Using the recursive description above, define a Scheme function named `(lucas n)` which takes one parameter, n, and computes the n^{th} Lucas number L_n.

(b) The small change in the "base" case when $n = 0$ seems to make a big difference: compare the first few Lucas numbers with the first few Fibonacci numbers. As you can see, the Lucas numbers are larger, which makes sense, and—in fact—the difference between the nth Lucas number and the nth Fibonacci number grows as a function of n.

Consider, however, the ratio of two adjacent Lucas numbers; specifically, define

$$\ell_n = \frac{L_n}{L_{n-1}}.$$

Write a Scheme function `(lucas-ratio n)` that computes ℓ_n (given n as a parameter). Compute a few ratios like ℓ_{20}, ℓ_{21}, ℓ_{22}, ...; what do you notice? (It might be helpful to convince Scheme to print out the numbers as regular decimal expansions. One way to do that is to add `0.0` to the numbers.)

Now define

$$f_n = \frac{F_n}{F_{n-1}}$$

where F_n are the Fibonacci numbers. As above, write a Scheme function `fibonacci-ratio` to compute f_n and use it to compute a few ratios like f_{20}, f_{21}, f_{22}, ...; what do you notice?

4. In lecture we presented code for `sqrt-converge`, which approximates a square root by starting with an interval that contains the square root, testing whether the midpoint of that interval is greater than or less than the actual square root, and then recursively calling itself on either the upper or lower half of the interval. In this problem you will generalize that code so it produces the `nth` root of a number for any positive integer `n`.

   (a) As part of this, we will need to be able to raise a number to the `nth` power where `n` is a positive integer – to be used analogously to `square` in the lecture code. We could use the following function, `(power base exp)`, that raises a number (`base`) to a power (`exp`):

```
(define (power base exp)
   (cond ((= exp 0) 1)
         (else
          (* base (power base (- exp 1))))))
```

   But we will be using this a lot, so we want to be more efficient. Design a Scheme function `fast-expt` which calculates b^e for any integer $e \geq 0$ by the rule:

$$b^e = \begin{cases} 1 & \text{if } e = 0, \\ (b^2)^{\frac{e}{2}} & \text{if } e \text{ is even}, \\ b * (b^2)^{\frac{e-1}{2}} & \text{if } e \text{ is odd}. \end{cases}$$

   You may want to try the `even?` and `odd?` functions defined in Scheme. Notice that this is *not* the rule used for `fastexp` in Lab.

   (b) Once `fast-expt` is working, you can define your `(nth-root-approx x n tol` function, which approximates the `nth` root of `x`, $\sqrt[n]{x}$. The approximation should be close enough so your approximation is within `tol` of `x`. Use the structure of `sqrt-converge` as a guide. Do not make `fast-expt` a local function of `nth-root-approx`, because it will be tested separately.

   (c) In his *Fundamental Algorithms*, computer scientist Donald Knuth asks the question whether $\sqrt[n]{n}$ has a limit as $n \to \infty$. Use your `nth-root-approx` function to write `(nth-root-of-n-approx n tol)`, which evaluates to an approximation (within `tol` of $\sqrt[n]{n}$). Test it on some `n` values – does it seem to have a limit?

5. In mathematics and other fields, two quantities $a > b$ are said to have the *golden ratio* if

$$\frac{a+b}{a} = \frac{a}{b}.$$

   For example, the heights of alternate floors of Notre Dame cathedral are in this proportion, as are the spacings of the turrets; the side-lengths of the rectangular area of the Parthenon have this ratio.

   Mathematically, it is easy to check that if a and b have this relationship then the ratio $\phi = a/b$ is the unique positive root of the equation $\phi + 1 = \phi^2$, which is approximately 1.618.

   (a) An alternate form for the golden ratio is given by $\phi = 1 + \frac{1}{\phi}$. This formula can be expanded recursively to the *continued fraction*:

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}$$

Define a recursive Scheme function named `(golden n)` which takes one parameter, n, and computes an approximation to the value of this repeated fraction by expanding it to depth n. To be precise, define

$$\Phi_1 = 1 + \frac{1}{1} = 2\,,$$
$$\Phi_2 = 1 + \frac{1}{1 + \frac{1}{1}} = \frac{3}{2}\,,$$
$$\Phi_3 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = 1\frac{2}{3}\,,$$
$$\ldots$$

or, more elegantly,

$$\Phi_1 = 2\,,$$
$$\Phi_n = 1 + \frac{1}{\Phi_{n-1}} \quad \text{for } n > 1.$$

Your function, given n, should return Φ_n.

(b) Another form of the golden ratio is given by the formula $\phi^2 = 1 + \phi$. This gives a recursive formula for a *continued square root*:

$$\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \cdots}}}}$$

Define a Scheme function named `(golden-sqrt n)`that takes one parameter, n, and computes the n^{th} convergent of the golden ratio using the continued square root. (Use the Scheme function `(sqrt x)` which returns the square root of x.)

The next two questions require the use of a function that generates a random number. I have supplied Scheme code with this assignment in file `r5rs-random.rkt`, so you can conveniently copy and paste it into your `problemSet3.rkt` file. This code has a number of features that we have not yet seen in Scheme; use it as it is and trust that we will learn those features later. It includes a function `random` that produces a uniformly-distributed random value between 0.0 and 1.0; that is the probability that `(random)` is less than a value x (where x is between 0 and 1) is x; for example, the probability that `(random)` is less than .5 is .5, the probability that `(random)` is less that .7 is .7, and so forth.

6. One way to spend time with a friend is to flip coins as a game. Consider the following: Players take turns flipping one coin at a time. If the toss results in Heads, the flipper gets a coin from his or her opponent, if it results in Tails, the flipper gives a coin to his or her opponent. Unless the players both have an infinite number of coins, eventually one player will have all of the coins (possibly a very long time if there are a lot of coins).

Suppose I want to simulate this game between two players A and B, where A has n coins and B has k coins. I can simulate a single toss by generating a random number between 0 and 1, giving one of A's coins to B if that number is less than .5, and giving one of B's coins to A otherwise. I run the game by repeating tosses until one player runs out of coins, counting the number of tosses in the game. To cleverly encode extra information, I report the number of tosses taken if A runs out of coins, and the negation of the number of tosses if B runs out of coins.

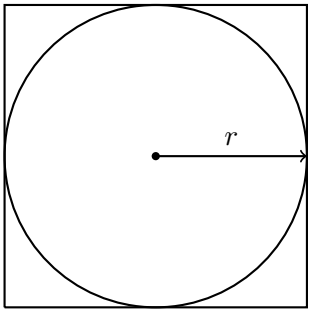
(a) First, you will write a function `(tosses-taken a b)` that evaluates to the number of tosses taken if A runs out of coins, or the negation of the number of tosses taken if B runs out of coins; `a` and `b` are the number of coins that player A and B have at the start of the game. As you need to count things, it may be useful to define a helper function.

(b) Second, you will write a function `(count-wins a b n)` that reports the number of times that player A wins when you run `n` games that start with A having `a` coins and B having `b` coins.

Testing these functions can be tricky; since they are based on random events they will return different values each time. Try them with a range of values (keep `a` and `b` relatively small) and see if the results match your intuitions.

7. Consider the task of approximating the number π. There are many ways to proceed; in this problem you will explore one method based on a process known as *monte-carlo sampling*.

We begin by inscribing a circle (of some radius r) inside a square, as shown in the picture.



Then, compute the ratio of the area of the circle to the area of the square. This ratio ρ is

$$\rho = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}\,.$$

Of course, if we could compute ρ exactly that would be great, because π is exactly $4 \cdot \rho$.

Our strategy will be to *approximate* ρ by throwing darts randomly into the square. Consider a dart thrown into the square by selecting each of its x and y coordinates by drawing randomly (and uniformly) from the interval $[-r, r]$. It follows that the probability that the dart lies in the circle is precisely ρ. If we throw n random darts into the square according to this rule, we would expect that the fraction that fall in the circle is close to ρ:

$$\rho \approx \frac{\text{number of darts in the circle}}{n}\,.$$

The algorithm you are required to write computes an estimate of π via this method. It takes as input an integer n indicating how many darts should be thrown. It then proceeds by throwing the darts uniformly at random at a square of side 2 centered at the origin (so both the x and y coordinates of the spot where the dart lands are random numbers between -1 and 1). The algorithm must determine, for each dart, where it lands (inside or outside the circle?) and tally the darts that fall inside. The estimate for ρ and therefore π directly follow.

Of course, you will need a mechanism for generating random numbers; the `random` function used above would work fine: `random` produces a "random" floating point value between 0 and 1. To obtain a random number in the range $[-1, 1]$, for example, you can evaluate the Scheme expression `(- (* 2 (random)) 1)`. (Why is this syntax correct?)

(a) First write a function `(one-sample)` which throws one sample into the square and returns `#t` if the sample fell in the circle, `#f` otherwise. One easy way to structure this function is to use a `let` statement to first bind two variables x and y to two random values. Note that if you have the x and y coordinates of a vector in the plane, it is easy to tell if it fell in the circle: this happens exactly when $x^2 + y^2 \leq 1$.

(b) Next, write a function `(pi-samples k)` which throws `k` samples into the unit square and returns the number of them that fell into the circle.

(c) Finally, write a function `(pi-approx k)` which evaluates to an approximation of π based on k samples.

8. Here is an extra practice problem that will not affect your grade – give it a shot!

Consider the problem of defining a function `interval-sum` so that `(interval-sum m n)` returns the sum of all the integers between m and n. (So, for example `(interval-sum 10 12)` should return the value $33 = 10 + 11 + 12$.) One strategy is

```
(define (interval-sum m n)
  (if (= m n)
      m
      (+ n (interval-sum m (- n 1)))))
```

and another solution, which recurses the "other direction" is

```
(define (interval-sum m n)
  (if (= m n)
      m
      (+ m (interval-sum (+ m 1) n))))
```

These both work. It seems like one should be able to combine these to produce another version:

```
(define (interval-sum2 m n)
  (if (= m n)
      m
      (+ m
         (interval-sum2 (+ m 1) (- n 1))
         n)))
```

But this only works for certain pairs of input numbers. What's going on?

Write a Scheme function `(ok-for-interval-sum2 m n)` that evaluates to `#t` if `(interval-sum2 m n)` will work correctly, `#f` otherwise.

Once you have finished:

1. Save your work to a file using the appropriate filename, problemSet3.rkt.
2. Submit your solution file for grading via the MIMIR site.