CSE1729 - Introduction to Programming February 16, 2017

Problem Set 3 Solutions

Remark 1. *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). Show how to use SCHEME to compute H_n and, using this, give an estimate of Euler's constant. (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.

```
(define (harmonic n)
  (if (= n 1)
      1
      (+ (/ 1 n) (harmonic (- n 1)))))

(abs (- (harmonic 1000) (log 1000)))
```

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.) Using this procedure, write a function `count-primes` so that `(count-primes t)` returns the number of prime numbers between 1 and t.

```
(define (count-primes t)
  (cond ((= t 1) 0)
        ((prime? t) (+ 1 (count-primes (- t 1))))
        (else (count-primes (- t 1)))))
```

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 `Lucas` which takes one parameter, n, and computes the n^{th} Lucas number L_n.

```
  (define (lucas n)
  (cond ((= n 0) 2)
    ((= n 1) 1)
    ((> n 1) (+ (lucas (- n 1))
                (lucas (- n 2))))))
```

(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.

Index	F_n	L_n
0	0	2
1	1	1
2	1	3
3	2	4
4	3	7
5	5	11
6	8	18
7	13	29
8	21	47
9	34	76
10	55	123

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` 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.)

```
  (define (l-ratio n) (/ (lucas n)
                         (lucas (- n 1))))
  (exact->inexact (l-ratio 20))
  (exact->inexact (l-ratio 21))
  (exact->inexact (l-ratio 22))
  (exact->inexact (l-ratio 23))
  (exact->inexact (l-ratio 24))
  (exact->inexact (l-ratio 25))
```

Index	L_n/L_{n-1}
20	1.6180340143330838
21	1.6180339789779863
22	1.6180339924824318
23	1.6180339873241927
24	1.6180339892944646
25	1.6180339885418877

Gee; they're all pretty close. It seems that the ratio between adjacent Lucas numbers converges to some particular value about 1.618…

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 $f_{20}, f_{21}, f_{22}, \ldots$; what do you notice?

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        ((> n 1) (+ (fib (- n 1))
                    (fib (- n 2))))))
(define (f-ratio k) (/ (fib k)
                       (fib (- k 1))))
(exact->inexact (f-ratio 20))
(exact->inexact (f-ratio 21))
(exact->inexact (f-ratio 22))
(exact->inexact (f-ratio 23))
(exact->inexact (f-ratio 24))
(exact->inexact (f-ratio 25))
```

This yields

Index	F_n/F_{n-1}
20	1.6180339631667064
21	1.6180339985218033
22	1.618033985017358
23	1.6180339901755971
24	1.618033988205325
25	1.618033988957902

Whoa...the ratios of adjacent large Fibonacci numbers also seem to converge...to the same value!

(c) *Computing with a promise.* Ask your SCHEME interpreter to compute L_{30}, then L_{35}, then L_{40}. What would you suspect to happen if you asked it to compute L_{50}?

Consider the following SCHEME code for a function of four parameters called `fast-Lucas-help`. The function call

```
(fast-Lucas-help n k lucas-a lucas-b)
```

is supposed to return the nth Lucas number *under the promise that it is provided with any pair of previous Lucas numbers*. Specifically, if it is given a number $k \leq n$ and the two Lucas number L_k and L_{k-1} (in the parameters `lucas-a` and `lucas-b`), it will compute L_n. The idea is this: If it was given L_n and L_{n-1} (so that $k = n$), then it simply returns L_n, which is what is was supposed to compute. Otherwise assume $k < n$, in which case it knows L_k and L_{k-1} and wishes to make some "progress" towards the previous case; to do that, it calls `fast-Lucas-help`, but provides L_{k+1} and L_k (which it can compute easily from L_k and L_{k-1}). The code itself:

```
(define (fast-Lucas-help n k luc-a luc-b)
  (if (= n k)
      luc-a
      (fast-Lucas-help n (+ k 1) (+ luc-a luc-b) luc-a)))
```

With this, you can define the function `fast-Lucas` as follows:

```
(define (fast-Lucas n) (fast-Lucas-help n 1 1 2))
```

(After all, $L_0 = 2$ and $L_1 = 1$.)

Enter this code into your SCHEME interpreter. First check that `fast-Lucas` agrees with your previous recursive implementation (Lucas) of the Lucas numbers (on, say, $n = 3, 4, 5, 6$). Now evaluate `(fast-Lucas 50)` or `(fast-Lucas 50000)`.

There seems to be something qualitatively different between these two implementations. To explain it, consider a call to `(Lucas k)`; how many total recursive calls does this generate to the function `Lucas` for $k = 3, 4, 5, 6$? Now consider the call to `(fast-Lucas-help k 1 1 2)`; how many recursive calls does this generate to `fast-Lucas-help` for $k = 3, 4, 5, 6$? Specifically, populate the following table (values for $k = 1, 2$ have been filled-in):

	Recursive calls made by `(Lucas k)`	Recursive calls made by `(fast-Lucas-help k 1 1 2)`
$k = 1$	0	0
$k = 2$	2	1
$k = 3$	4	2
$k = 4$	8	3
$k = 5$	14	4
$k = 6$	24	5

4. 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 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

$$\begin{aligned}
\Phi_1 &= 1 + \frac{1}{1} = 2,\\
\Phi_2 &= 1 + \frac{1}{1 + \frac{1}{1}} = \frac{3}{2},\\
\Phi_3 &= 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1}}} = 1\frac{2}{3},\\
&\dots
\end{aligned}$$

or, more elegantly,

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

Your function, given n, should return Φ_n.

```
(define (golden n)
 (if (= n 1)
     2
     (+ 1 (/ 1 (golden (- n 1))))))

(exact->inexact (golden 20))
```

(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*:

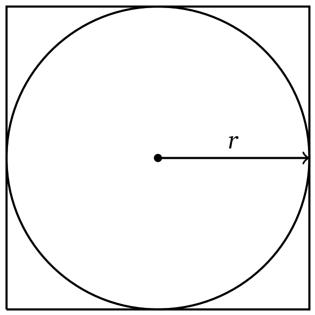
$$\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \cdots}}}}$$

Define a SCHEME function 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.)

```
(define (golden-sqrt n)
  (if (= n 0)
      1
      (sqrt (+ 1 (golden-sqrt (- n 1))))))
```

5. 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 acording 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 SCHEME function called `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?)

As for structuring your code, I suggest that you first write a function

```
(define (one-sample) ...)
```

which throws one sample into the square and returns a 1 if the sample fell in the circle, and a 0 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 $x^2 + y^2 \leq 1$. Then, write a function

```
(define (pi-samples k) ...)
```

which throws k (random) samples into the unit square and returns the number of them that fell into the circle. Now what?

```
(define (one-sample)
  (define (transform z) (- (* 2 z) 1))
  (let ((x (transform (random)))
        (y (transform (random))))
```

```
    (< (sqrt (+ (* x x) (* y y))) 1)))

(define (pi-samples k)
  (if (= k 0)
      0
      (+ (if (one-sample) 1 0)
         (pi-samples (- k 1)))))

(define (pi-approx k)
  (exact->inexact (* 4 (/ (pi-samples k) k))))
```

Here's another more compact solution.

```
(define (pi-approx n)
  (define (pi-approx-aux i acc)
    (let ((dist (sqrt (+ (expt (- (* 2 (random)) 1) 2)
                         (expt (- (* 2 (random)) 1) 2)))))
    (if (= i 0)
        acc
        (pi-approx-aux (- i 1) (if (<= dist 1) (+ acc 1) acc)))))
  (* (/ (pi-approx-aux n 0) n) 4.0))
```

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

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

Indeed, this will only work if $n - m$ is even. Notice that the difference between m and n is reduced by two during each recursive call. If the difference reaches zero, the program will gracefully terminate with the correct value; it follows that if the two values start with a positive even difference that the program will work. However, if the program is called in a setting where $n = m + 1$, the recursive call will be made with $n = m - 1$ and will never return.