CSE1729 - Introduction to Programming | September 18, 2020

Problem Set 3

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) (/ 1 6)
0.9999999999999999
>
```

Clearly, the use of fractions (rationals) is more precise because of the rounding errors that occur with decimals. 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.*). You can "force" a conversion from a fraction to a decimal easily. Indeed, writing* `(+ (/ 1 3) 0.0)` *or even* `(exact->inexact (/ 1 3))` *will produce a decimal from the fraction* $\frac{1}{3}$.

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) Give a SCHEME function, called `harmonic` so that `(harmonic n)` returns H_n.

   (b) Using your function from the previous part, give an estimate of Euler's constant. Specifically, write a SCHEME function
   mintinlineschemeEulerest so that `(Eulerest n)` returns the absolute value of the difference between H_n and $\ln n$. (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. Note how it uses a nested definition `divisor?` (see class) to keep the code clean and tidy.

```
(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 m)` returns the number of prime numbers between 1 and m.

3. A pair of integers (a, b) are *relatively prime* if there is no integer $d > 1$ that evenly divides both of them. The following SCHEME procedure determines if a given pair of numbers are relatively prime:

```
(define (rel-prime a b)
  (define (divides-both d)
    (and (= 0 (modulo a d))
         (= 0 (modulo b d))))
  (define (divisor-upto k)
    (and (> k 1)
         (or (divides-both k)
             (divisor-upto (- k 1)))))
  (not (divisor-upto (min a b))))
```

(So, it returns `#f` for pairs like $(4,6)$ and $(3,21)$ which have common divisors, and `#t` otherwise.) Write a SCHEME function `count-rel-prime` so that `(count-rel-prime n)` returns the number of pairs of integers (a, b) so that $1 \leq a \leq n$ and $1 \leq b \leq n$ and a and b are relatively prime.

4. 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` that computes ℓ_n (given n as a parameter). Compute a few ratios like $\ell_{20}, \ell_{21}, \ell_{22}, \ldots$; 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 $f_{20}, f_{21}, f_{22}, \ldots$; what do you notice?

(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
$k = 3$		
$k = 4$		
$k = 5$		
$k = 6$		

5. The half-companion Pell numbers H_n and the Pell numbers P_n can be defined by the following paired recurrence relations. **Notice H_n is defined in terms of both H_n and P_n. So is P_n.**

$$H_n = \begin{cases} 1 & \text{if } n = 0 \\ H_{n-1} + 2P_{n-1} & \text{otherwise} \end{cases}$$

$$P_n = \begin{cases} 0 & \text{if } n = 0 \\ H_{n-1} + P_{n-1} & \text{otherwise} \end{cases}$$

(a) Provide a SCHEME function, named `(H n)` which takes one parameter, n, and returns the half-companion Pell number at index n using the recurrence relation above.

(b) Provide a SCHEME function, named `(P n)` which takes one parameter, n, and returns the Pell number at index n using the recurrence relation above.

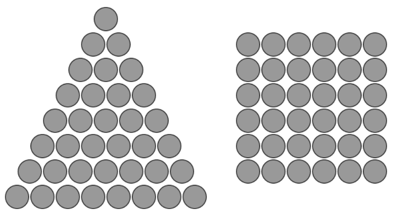

Figure 1: 36 circles arranged in a triangle as well as a square.

A triangular number counts objects arranged in an equilateral triangle. The n^{th} triangular number is the number of objects arranged in an equilateral triangle with n items along each side.
A *square triangular* number is a triangular number that is also a perfect square. The problem of finding square triangular numbers reduces to Pell's equation in that every triangular number is of the form $\frac{t(t+1)}{2}$. Therefore we seek integers t and s such that $\frac{t(t+1)}{2} = s^2$.

(c) Define a SCHEME function named `(t n)` to compute the t value for the n^{th} triangular square. Use the following definition to write your function:

$$t_n = \begin{cases} 2P_n^2 & \text{if } n \text{ is even} \\ H_n^2 & \text{if } n \text{ is odd} \end{cases}$$

(d) Define a SCHEME function named `(s n)` to compute the s value for the n^{th} triangular square. Use the following definition to write your function:

$$s_n = H_n P_n$$

(e) Finding the n^{th} triangular square becomes computing

$$\frac{t_n(t_n+1)}{2}$$

Define a SCHEME function named `(tri-square n)` which uses this expression and your `(t n)` function to find the n^{th} triangular square.

(f) The n^{th} triangular is also equivalent to s_n^2. Define a SCHEME function, named `(square-tri n)`, which uses this expression and your `(s n)` function to compute the n^{th} triangular square.

6. 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 adjacent 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) The golden ratio also satisfies the equation $\phi = 1 + \frac{1}{\phi}$. 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

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

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, called `golden`, when given n, should return Φ_n.

(b) As mentioned above, the golden ratio satisfies the equation $\phi^2 = 1 + \phi$. This can be expanded into a different recursive formula that yields 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. To be more precise, define:

$$\Phi_n = \begin{cases} 1 & \text{if } n = 0,\\ \sqrt{1 + \Phi_{n-1}} & \text{if } n > 0.\end{cases}$$

(Note that these Φ_i are different from those in the previous part of the problem.) Now define a SCHEME function `golden-sqrt` so that `(golden-sqrt n)` returns the value Φ_n. (Use the SCHEME function `(sqrt x)` which returns the square root of x.)

7. 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?

8. The famous Ackermann Function, named for Wilhelm Ackermann whose doctoral advisor was David Hilbert (a famous mathematician) is generally expressed today as:

$$\operatorname{ack}(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ \operatorname{ack}(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ \operatorname{ack}(m-1,\operatorname{ack}(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

What makes this function absolutely recursive is the second parameter to the recursive call to ack in the last case. The second parameter to ack in this case is itself a recursive call to ack. Notice, that in each recursive call, either m or n are decreased and neither are ever increased. Therefore, this algorithm will always terminate, eventually. Because of the diabolical recursive second parameter to the recursive call in the last case, computing this function for any values other than fairly small values of m and n takes more time to compute than any reasonable human being would wait. In fact, computing $\operatorname{ack}(4,1)$ took longer than three minutes on my laptop. Note, however, that even though we can not realistically compute Ackermann's function for fairly large values for m and n, we do know that if we could build a computer that would last long enough and could store extremely large integer values, the algorithm would eventually terminate with the solution.

Define a SCHEME function, named `(ack m n)`, that computes the Ackermann function as defined above. For testing, your function should be able to compute the following:

```
>(ack 3 4)
125
>(ack 4 0)
13
```

9. In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively defined objects. For instance, C_n counts the number of expressions containing n pairs of parentheses which are correctly matched:

```
((()))  ()(())  ()()()  (())()  (()())
```

The n^{th} Catalan number is given directly in terms of binomial coefficients by

$$C_n = \prod_{k=2}^{n} \frac{n+k}{k} \text{ for } n \geq 0$$

The first Catalan numbers for $n = 0, 1, 2, 3, \ldots$ are

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$$

Define a SCHEME function, named `(catalan n)`, computes the n^{th} Catalan number using the definition above.