CSE 1729	Introduction to Principles of Programming Problem Set 10	December 6, 2018

Note that this problem set is due on a Wednesday night/Thursday morning. It is the last one of the semester, so have a ball!

Steams, streams, and more streams.

Remember to use the stream code that was supplied for the last lab. A solution to the lab problems – useful stream functions – will be provided after the lab is due. I stream, you stream, we all stream...

1. *Mersenne primes* are prime numbers that are one less than a power of t, that is

$$p = 2^k - 1$$

for some integer k. They are pretty common for small k: $2^2 - 1, 2^3 - 1, 2^5 - 1, 2^7 - 1$ are all prime numbers for example, but then they thin out a bit. In this problem you will produce a stream of the Mersenne primes. For all of these you should use a signal-processing coding style.

   (a) First, write a function `(primes)` that evaluates to the stream of all primes. Hint: we if we have a function `prime?` we could use it to filter all of the primes from the stream of integers starting at 2.

   (b) One of the interesting features of Mersenne primes is that k in the above formula is also a prime number, that is, if 2^k-1 is prime, then k is prime (sadly, k being prime does not mean that 2^k-1 is prime). Using this and your `(primes)` function, write a function `(mersenne-candidates)` that evaluates to a stream of all numbers of the form $2^k - 1$ such that k is a prime number. This stream should be in increasing order.

   (c) Finally, write a function `(mersenne-primes)` that evaluates to a stream of all of the Mersenne primes in increasing order. As you look at these in order by doing successive `stream-cdrs` of the stream you will see that it becomes computationally difficult as these numbers get large quickly, but you should be able to see the first seven of these (using `str-to-list`, for example) fairly quickly.

2. We can define some streams implicitly by using a form that includes the name in the definition. Recall, for example, the implicit definitions for a streams of ones and for the non-negative integers:

```
(define ones  (cons-stream 1 ones))
(define ints (cons-stream 1 (add-streams ones ints)))
```

   (a) Implicitly define `facts`, the stream of factorial numbers: $1, 1, 2, 6, 24, \ldots$.

   (b) Implicitly define `pow-seven`, the stream of the powers of 7, i.e. 1, 7, 49, 343,….

3. (a) e can be expressed as a power series as follows:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots$$

Define a function `(e-terms)` that returns the the stream corresponding the terms of this sum, that is, the stream containing

$$\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \frac{1}{3!}, \ldots$$

(b) Use your `partial-sums` function from lab and the sequence in 3a to write a function `(e-approx)` that returns a stream containing successive approximations to e:

$$\sum_{k=0}^{n} \frac{1}{i!} \text{for } n = 0, 1, 2, \ldots$$

.

4. (a) Write function `(stream-merge str1 str2)` that takes 2 streams of numbers, each in increasing order, and returns the stream that contains all of those values in sorted (increasing) order without repeats. Note that each input stream is sorted without repeats, but the same element could occur in both lists.

   (b) Use your merge function to produce `235-stream`, the sorted stream of positive integers whose prime factors are 2, 3 and 5 only. i.e. all numbers of the form $2^i 3^j 5^k$, for $i \geq 0, j \geq 0, k \geq 0$. Your definition can be implicit or not, but the function `(235-stream)` should evaluate to that stream. Hint: if x is an element of this stream, then 1) $2x$ is a member of this stream, 2) $3x$ is a member of this stream, and 3) $5x$ is a member of this stream.

5. Suppose we want to generate all (i, j) pairs, such that i and j are positive integers.

   If `(enumerate-integers-from 0)` from lab produces the stream of integers $0, 1, 2, 3, \ldots$, then we could use the following code:

```
(define (interleave s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))

(define (pairs s t)
  (cons-stream
   (cons (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (cons (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s)  t))))

(define (pairs-stream)
  (let ((naturals (stream-cdr (enumerate-integers-from 0))))
    (pairs naturals naturals)))
```

   Try this code, and examine the order of the pairs in `(pairs-stream)`. What sort of pattern does this order exhibit? (More about this later.)

It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an ad hoc interleaving process. We can use a technique similar to the merge procedure from the last lab, if we define a way to say that one pair of integers is "less than" another. One way to do this is to define a "weighting function" W and stipulate that $x < y$ if if $W(x) < W(y)$. To compare pairs, W must be a function that takes one pair as a parameter and returns some numeric value.

(a) Write a procedure `merge-weighted` that is like merge, except that merge-weighted takes an additional argument weight, which is a procedure that computes the weight of a stream element, and is used to determine the order in which elements should appear in the resulting merged stream. Notice that although `merge-weighted` should not allow duplicates (defined as two things that are `equal?`), it may be the case that $W(x) = W(y)$ for some x and y values, even though x and y are not duplicates. As with merge, you can assume that the inputs are sorted with respect to the weighting function.

(b) Using `merge-weighted`, generalize `pairs` to a function `weighted-pairs` that takes two streams, together with a procedure that computes a weighting function, and generates the stream of pairs, ordered according to weight.

(c) Use your `weighted-pairs` procedure to write the function `(weighted-pairs-stream)` , which evaluates to the stream of all pairs of positive integers (i,j) with the pairs ordered according to the sum i + j.

(d) Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called Ramanujan numbers, in honor of the mathematician Srinivasa Ramanujan. Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$, then search the stream for two consecutive pairs with the same weight. (We need to ensure that we do not include (i, i) pairs, and only one of (i, j) or (j, i) for $i \neq j$), but that should be easy). Write a function `(ramanujan n)` that returns the nth Ramanujan number. The first such number is 1729 *(What a coincidence!)*. What are the next five?

Hint: an incremental approach is best for this last one – first write code to generate the stream of pairs ordered by sum of cubes, and work from there. The most elegant way to write `(ramanujan n)` would be to use `stream-nth` on the stream of Ramanujan numbers, but alternative approaches are possible.

6. If the stream `pairs-stream` (generated using the code at the start of problem 5) contains all of the pairs of integers without repeats in some order such that any pair can be found after a finite number of `stream-cdrs`, then it defines a pairing between the positive integers $1, 2, 3, \ldots$ and all of the pairs of positive integers. In that case, there exist two functions: `(encode-ps pair)` which maps from the pair `(i . j)` to `n`, its position in `pairs-stream`, and `(decode-ps n)` which maps from the integer `n` to the pair `(i . j)` which is at the `n`th position in `pairs-stream`.

(a) Write the function `(encode-ps pair)` that takes a pair of positive integers and returns its position in `pairs-stream`.

(b) Write the function `(decode-ps n)` that takes a positive integer and returns the pair that would be at that position in `pairs-stream`.

Neither of these functions may use `pairs-stream` *or any other stream to calculate its result.*

It is not completely clear how to write these functions – there are many possible approaches. My suggestion is that you play around with `pairs-stream` and try to deduce the pattern. Although these functions cannot use `pairs-stream` in their computation, you can use `pairs-stream` and various stream functions to test whether your functions are correct.

These are the last two problems on the last problem set, and are meant to be challenging. Good luck!