COMP 360, Fall 2015, Project 4

In this assignment, you will explore some fun and exciting streams and ways of using them.

Getting Started

Download `proj4-start.rkt` from the class website to your own computer. This file defines all the stream-related function you need to get started, including `stream-cons`, `stream-car`, and so on.

Functions to Define

1. Define a function `make-recursive-stream` that serves as a general-purpose stream-creation function that creates a stream where the next element of the stream is always calculated from the previous element.

   `make-recursive-stream` takes two arguments: a function of one argument called `f`, and an initial value for the stream called `init`. `make-recursive-stream` returns a stream consisting of the elements `init`, `(f init)`, `(f (f init))`, and so on.

   ```
   Example: (make-recursive-stream (lambda (x) (+ x 1)) 1)
            ==> stream of 1, 2, 3, 4, ...
   Example: (make-recursive-stream (lambda (s) (string-append s "a")) "")
            ==> stream of "", "a", "aa", "aaa", ...
   ```

   Hint: This function might seem more complicated than it really is. Just think about the code that creates an infinite stream of integers:

   ```
   (define (integers-from n)
     (stream-cons n (integers-from (+ n 1))))
   ```

   Think about how you would generalize this.

2. Pascal's triangle is a (infinitely large) triangular structure of numbers that looks like this:

   ```
             1
           1   1
         1   2   1
       1   3   3   1
     1   4   6   4   1
   1   5  10  10   5   1
  . . . . . etc . . . . .
   ```

   One way to define the triangle is to say that the first two rows of the triangle consist of one 1 and two 1's, respectively. Each further row is begins and ends with a 1, and each "interior" number in a row is the sum of the two numbers in the preceding row that are to the left and right of the number in question. For example, the 2 in the third row is the sum of the two 1's in the preceding row.

   Define a stream `pt` that consists of each row of Pascal's triangle as a list. Hint: Use `make-recursive-stream` by defining a function that takes a row of Pascal's triangle and generates the next row.

Do not do this problem using the binomial theorem or using some other definition of Pascal's triangle.

```
Example: (define pt . . . whatever you decide . . . )
         (stream-enumerate pt 6)
         ==> '((1) (1 1) (1 2 1) (1 3 3 1) (1 4 6 4 1) (1 5 10 10 5 1))
```

3. Define a function `stream-flatten` that takes an infinite stream of lists and returns a new stream consisting of all the elements from the first list, followed by all the elements from the second list, and so on.

   ```
   [This assumes you've already written the Pascal's triangle part from above.]
   Example: (define flatpt (stream-flatten pt))
            (stream-enumerate flatpt 20)
            ==> '(1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5)
   ```

4. Define a function `merge` that takes two infinite streams of numbers where each individual stream is in sorted order. `merge` returns a new stream consisting of all the items from the two streams combined in sorted order.

   ```
   Example: (define pow2 (make-recursive-stream (lambda (x) (* x 2)) 1))
            (define pow3 (make-recursive-stream (lambda (x) (* x 3)) 1))
            (stream-enumerate (merge pow2 pow3) 20)
            ==> '(1 1 2 3 4 8 9 16 27 32 64 81 128 243 256 512 729 1024 2048 2187)
   ```

The next few questions all use streams to produce musical tones. This is a great use of streams because we can represent a sound by a stream of samples.

A little about how sound generation works:

Sounds are transmitted through the air (or other mediums) via sound waves. The sound for a "pure" musical tone can be represented by a sine wave; that is, a sine function with a specific amplitude and period. In sound, the amplitude of a wave controls how loud the sound is (larger amplitudes lead to higher volumes) and the period of the wave (how fast the sine wave oscillates between the maximum and minimum values) controls the pitch of the tone (faster oscillations lead to higher pitches).
As an example, a sine wave consisting of 440 oscillations per second (Hertz) produces the musical note A (specifically, the A above middle C).
Because it is very hard to represent more sophisticated sounds (like speech) with sine waves, computers represent sound by *sampling* sound waves many times per second to try to approximate the true sound wave as closely as possible. For instance, CDs use an audio format that samples the sound wave 44,100 times every second.
You will be producing streams that represent sounds using this same sampling procedure. There is a sound library that comes with Racket that will let you listen to your streams as sounds.
The only two functions that you need to know about are:

- `(play-stream stream)` is a function that takes an infinite stream and plays the stream as a sound. It assumes that the stream consists of floating point numbers between -1 and 1 that

represent a sound wave sampled 44,100 times per second. In other words, playing one second of sound uses the first 44,100 numbers from the stream.

I would recommend restricting your amplitude values to between about +0.2 and -0.2 because 1 and -1 correspond to the loudest possible sound your computer can make. So keep your volume low to prevent blowing out your speakers.

- `(stop)` is a function that stops all sounds playing. You will need to call this because when you play an infinite stream, it naturally plays forever.

5. Define a function `make-sine-stream` that takes a frequency as an argument. This function returns an infinite stream of floating point numbers representing a sine wave that ranges between -0.2 and +0.2 that has a period of 1/freq, and is sampled 44,100 times per second.

   As an example, say we call `(define s (make-sine-stream 441))`. This gives us a stream of floating point numbers representing a sine wave that oscillates 441 times per second, sampled at 44,100 times per second. The first 100 numbers in the sequence will therefore represent one cycle of the wave.

   Note that the `sin` function in Racket works in radians.

   ```
   Example:  (define s (make-sine-stream 441))
             (stream-ref s 0) ==> 0
             (stream-ref s 25) ==> 0.2
             (stream-ref s 50) ==> 0
             (stream-ref s 75) ==> -0.2
             (stream-ref s 100) ==> 0
             (play-stream s)  ; plays the tone
   ```

6. `make-sine-stream` returns a stream that wastes a lot of memory because the sine function is periodic but your solution to the previous problem (probably) isn't taking advantage of that. In other words, your function for question 5 probably creates an infinite number of cons cells, which is wasteful because the the infinite stream you are creating eventually repeats the sequence of floating point numbers over and over again. It would be a lot more memory efficient (constant memory versus infinite memory) if we could create a stream that re-used cons cells to create a true "circular" stream in memory, rather than having to continuously allocate new cons cells to repeat the same sequence over and over.

   This is analogous to the two ways we looked at to create an infinite stream of 1s: one way actually made a recursively-linked list with one cons cell, and the other way made a linked list of lots of cons cells.

   Define a new function `make-circular-stream` that takes a single **list** argument. What this function will do is create a circular stream consisting of all the items in the list, such that the cdr of the last element in the stream is a promise to point back to the first cons cell in the stream.

   This function is tricky to get right because you have to save the first cons cell in the stream that you create with stream-cons so that when you reach the end of the list, you can point back to it. You can use mutation to solve this problem.

Here's how you'll know you did it right. These lines of code create two infinite streams, each one alternating between 1 and -1. However, you can see by the output below that the "good" one is circular, and the "bad" one is not. (Note how the output for the good one labels the whole stream as "#0" and then the last cons cell contains a promise in the cdr that references #0 again, so the whole thing is circular.)

```
> (define good (make-circular-stream '(1 -1)))
> (define bad (make-recursive-stream (lambda (x) (- x)) 1))
> (stream-enumerate good 4)
'(1 -1 1 -1)
> (stream-enumerate bad 4)
'(1 -1 1 -1)
> good
#0='(1 . #<promise!(-1 . #<promise!#0#>)>)
> bad
'(1 . #<promise!(-1 . #<promise!(1 . #<promise!(-1 . #<promise!(1 .
     #<promise:...>)>)>)>)>)
```

Assessment

Solutions should be:

- Correct
- In good style, including indentation and line breaks
- Written using features discussed in class.

Turn-in Instructions

- Put all your solutions in one file, `proj4_lastname_firstname.rkt`, where `lastname` is replaced with your last name, and `firstname` is replaced with your first name.
- Upload your file to Moodle before the project deadline.