CSE4102: Programming Languages **Spring 2017**

Homework 3: Streams

Out Date: 02/07/2017
Due Date: 02/14/2017

Objectives

All you will be doing in this homework is get some practice with streams and stream computing. By now, you should be already familiar with the SML environment. Submit a single text file (`.sml` extension) with your answers. Do respect the name of the functions to ease grading. The problem statements are pretty short and the answers are even shorter. But it will take you some time to absorb this way of thinking. The questions are scaffolded, therefore I strongly suggest that you handle them *in order*.

1 Introduction

Consider the SML infinite stream data type as defined in class, namely:

```
datatype 'a Stream = Nil
           | Cons of 'a * (unit -> 'a Stream);
exception Bad of string;
fun from seed next = Cons(seed,fn () => from (next seed) next);
fun head (Nil) = raise Bad("got nil in head")
  | head (Cons(a,b)) = a;

fun tail (Nil) = raise Bad("got nil in tail")
  | tail(Cons(a,b)) = b();

fun take 0 stream       = nil
  | take n (Nil)        = raise Bad("got nil in take")
  | take n (Cons(h,t)) = h::(take (n-1) (t()));
```

Note: do not copy-paste from the above into a text file as you are likely to pick up non-ascii characters –e.g., single quotes – that will throw off the SML interpreter. You are far better off simply retyping that!
Recall that `Stream` is merely an SML abstract data type featuring two constructors. The `Nil` constructor is useful to create an empty stream (in case something goes horribly wrong and you wish to return a dummy stream for instance). The `Cons` constructor is useful to encapsulate a stream state based on a *seed* and a delayed expansion function (the sleeping beauty!) that contains a generator. The `from` function is the user-visible stream creation utility. It takes two arguments:

- a numerical seed: `seed`
- a generator function `next` which, given the k^{th} value produces the $(k+1)^{th}$ value of the stream.

Closely observe how the implementation of `from` packages the seed with an anonymous SML function that effectively delays the construction of a new stream tail. The new stream is simply the suffix of the current stream obtained from an invocation of `next` generator on the current seed. The construction of `from` is not obvious and I enjoin you to carefully read the code and experiment with the SML interpreter before attempting the questions below.

The `head` and `tail` functions are helper functions to retrieve the current seed of a stream, or the suffix (another stream) of the current stream. Note how `tail` triggers an evaluation of the delayed construction

embedded in `from` by sending the special "no value" (a double parenthesis) to `b`. Finally, `take` is a utility function that extracts the first n elements from a stream s. In a sense, it boils down to a repeated application of tail. Once again, please, do study the code before considering the questions below.

The questions below (8 of them) are *highly* scaffolded to help you complete the homework. I *strongly* suggest doing them in the order they appear.

Question 1

Solve the following exercises

1. Write an SML statement that binds the infinite stream of the non-negative naturals (as reals), i.e., $[1.0, 2.0, 3.0, ...$ to *nat* (In class we did a stream of natural numbers!). Namely it should look like the following

   ```
   val nat = ...
   ```

2. Write an SML statement that binds the infinite stream of ones (as reals),namely, $[1.0, 1.0, 1.0, 1.0, ...$ to the SML variable *one*. Namely it should look like the following

   ```
   val one = ...
   ```

3. Write an SML statement that binds the infinite stream of zeroes (as reals), namely, $[0.0, 0.0, 0.0, 0.0, ...$ to the SML variable *zero*. Namely it should look like the following

   ```
   val zeroes = ...
   ```

4. Write an SML statement that binds the infinite stream $[1.0, -1.0, 1.0, -1.0, 1.0, ...$ (i.e., a stream of 1.0 with alternating signs) to the SML variable *alt*. Recall that in SML the unary minus is written ~. Namely it should look like the following

   ```
   val alt = ...
   ```

   Note how the *take* function must works fine with all your streams. For instance, a call

   ```
   take 5 nat
   ```

should produce $[1.0, 2.0, 3.0, 4.0, 5.0]$

Question 2

Write an SML function which, given two streams a and b produces the stream $[a_0 \cdot b_0, a_1 \cdot b_1, ...$ where $\cdot$ denotes the multiplication. Your function should have the following API

```
fun mul a b = ...
```

And it should, of course, return a value of type *real Stream* when passed two inputs of the same type. Note how the function is specified in curried style.

Question 3

Write an SML statement that produces a stream named fs holding the infinite factorial stream. Namely, the stream $[0!, 1!, 2!, 3!, 4!....$ is none other than $[1.0, 1.0, 2.0, 6.0, 24.0, 120.0, 720.0, 5040.0, 40320.0, 362880.0, ...$ Note that your implementation cannot make use of any traditional implementation of the factorial function. Instead, it must rely on the following observation:

Multiplying the two streams:

$$\begin{array}{llllllll} [1 & ,2 & ,3 & ,4 & ,5 & ,6 & ,.... \\ [0! & ,1! & ,2! & ,3! & ,4! & ,5! & ,.... \end{array}$$

produces the stream

$$[1! \quad ,2! \quad ,3! \quad ,4! \quad ,5! \quad ,6! \quad ,....$$

Namely, it uses the well-known identity $n! = n \cdot (n-1)!$ but does not recompute every term from scratch each time! As expected, you know the base case, namely, $0! = 1$.

Question 4

Write an SML function *weave* that takes two streams as input and produces a new stream that interleaves the two source streams. For instance, for the inputs:

$$\begin{array}{llllllll} [1 & ,2 & ,3 & ,4 & ,5 & ,6 & ,.... \\ [0 & ,0 & ,0 & ,0 & ,0 & ,0 & ,.... \end{array}$$

The output should be the stream:

$$[1 \quad ,0 \quad ,2 \quad ,0 \quad ,3 \quad ,0 \quad ,....$$

The definition should start as below

---

fun weave a b = ...

Question 5

Write an SML function px that takes a single argument x (type real) and produces the stream of all powers of x. Namely, it produces $[1, x, x^2, x^3, ...$ (Recall that $x^0 = 1$). Note that once again you are **not** supposed to recompute each term from scratch (and you should not be using the *power* function built into the SML basis library). The API would be

---

fun px x = ...

Question 6

Write an SML function $frac$ which, given a stream s computes the stream of the inverses of s. Clearly, for the input $[s_0, s_1, s_2, s_3, ...$ it produces

$$[\frac{1.0}{s_0}, \frac{1.0}{s_1}, \frac{1.0}{s_2}, \frac{1.0}{s_3}, ...$$

Question 7

Consider the infinite Taylor expansion for e^x. Namely

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ...$$

Write an SML statement that binds *coefs* to an infinite stream representing the coefficients of the expansion of e^x. Clearly, the stream should be $[1, 1, \frac{1}{2!}, \frac{1}{3!}, ...$ Finally, use your auxiliary functions defined earlier to write an SML function *eval* which, given a stream s of coefficients, a value x and an *order* n computes the expansion of e^x to the n^{th} term **using only infinite series defined earlier, the function `take` and possibly a higher-order like `foldl`.**
Note that with $coef = [c_0, c_1, c_2, ...$ and x, you can define the infinite streams

$$p(x) = [1, x, x^2, x^3, ...$$

as well as

$$terms(x) = coef \cdot p(x)$$

The API of eval should be

---

fun eval s x order = ...

---

where s is the stream of coefficients, x is the value at which to evaluate the polynomial and *order* is the number of terms to include. Finally, armed with your *eval* function write an SML function *ex* that computes the approximation of e^x to the 10^{th} order.

Question 8

Repeat the process of question 7 for the Taylor expansion of

$$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - ...$$

Note that your *eval* function is completely reusable, as well as the vast majority of auxiliary functions you wrote.

Have fun!