CS61A – Homework 5.2 **Kurt Meinz**
University of California, Berkeley Summer 2003

Topic: Streams

Lectures: Wednesday July 23, Thursday July 24

Reading: Abelson & Sussman, Section 3.5.1–3, 3.5.5

This assignment explores infinite streams. Use `show-stream` (abbreviated as `ss`) to print a stream; it takes an optional second argument specifying the number of elements to print. This homework is due at **8PM on Sunday, July 27**. Please put your solutions into a file called `hw6-1.scm` and submit it electronically with `submit hw6-1`. As always, include test cases in your file but be sure to comment them out so the file loads smoothly.

Question 1. Write a procedure `list->stream` that takes a list as its argument and returns an infinite stream of the elements of the list, re-starting at the begging once the end of the list is reached:

```
STk> (ss (list->stream '(there is no spoon)))
(there is no spoon there is no spoon there is ...)
```

Question 2. In this question, you'll write a more general `stream-map` procedure and use it to define a stream *implicitly.*

A. We'd like to generalize the two-argument `stream-map` function defined on Page 320 so that it behaves as follows (Assume `ones` and `integers` are both infinite streams.):

```
STk> (ss (stream-map list ones integers) 5)
((1 1) (1 2) (1 3) (1 4) (1 5) ...)
```

As you can see, the new `stream-map` takes n streams and a procedure that can take n arguments. The procedure is applied to the corresponding elements of each stream. You may assume that the streams given to `stream-map` will be infinite. Hence, a base case is not needed. Complete this definition of `stream-map`:

```
(define (stream-map proc . streams)
   ( ??
     (apply proc (map  ?? streams))
     (apply stream-map (map  ?? streams))))
```

B. We'd now like to create an infinite stream of factorials:

```
STk> (ss factorials)
(1 2 6 24 120 720 5040 40320 362880 3628800 ...)
```

The nth element of this stream is $n + 1$ factorial. Complete the following implicit definition of this stream:

```
(define factorials (cons-stream 1 (stream-map *  ??  ??)))
```

Notice that unlike `list->stream` from Question 1, you're not writing a function that returns a stream; instead, you're defining the variable `factorials` to be the *stream itself.* Yet, because of the delayed evaluation afforded by streams, you may refer to the stream you're defining as you're defining it! See Page 328 for a more complete discussion of *implicit* stream definitions. Do not define any helper functions for this problem. You may, however, use the `integers` stream.

The adventure continues on the next page.

Question 3. Create an infinite stream called `runs` that looks like this:

```
STk> (ss runs 15)
(1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 ... )
```

You'll probably want to use the generator approach to creating streams by defining an auxiliary function, say, `runs-generator` and calling it with some initial values. Then use it to define `runs`:

```
STk> (define runs (runs-generator parameters))
```

Question 4. Write a procedure `chocolate` that takes the name of someone who likes chocolate a lot and creates an infinite stream that says so:

```
STk> (ss (chocolate 'greg) 25) (greg likes chocolate greg really
likes chocolate greg really really likes chocolate greg really
really really likes chocolate greg really really really really
likes chocolate ... )
```

If you have trouble with this problem, try to first define a version of `chocolate` for lists that takes an additional argument: the maximum number of "really"s. Then gradually change list operations like `cons` and `append` to stream operations like `cons-stream` and `stream-append`. You'll need a helper function.

Question 5. The `pairs` procedure defined on Page 341 seems more complicated than needed. In the book's version, the first pair, represented by (S_0, T_0) on the diagram on Page 339, is formed explicitly. The `stream-map` handles the subsequent pairings of S_0. Why is the first pair a special case? Why can't `stream-map` take care of the entire row? Here is a simpler version of `pairs`:

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

Does this work? Explain what happens when we attempt to evaluate the following with the new definition:

```
STk> (pairs integers ones)
```