CSE 1729 - Introduction to Principles of Programming November 27, 2018

Laboratory Assignment 12

Objectives

- Program in "signal-processing" style
- Work with streams
- Write and use some stream tools

Activities

To effectively write stream code you will need the Scheme primitives to deal with the stream data type. Add the following code to the beginning of your Racket file. You should be working in R5RS Scheme.

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream head tail)
     (cons head (delay tail)))))

(define (stream-car x)
  (car x))

(define (stream-cdr x)
  (force (cdr x)))

(define empty-stream? null?)
```

(This code, plus code for `tracer`, a useful thing, is available in the attached file `stream-utils.rkt`).

To make sure that your stream utilities work, and so you have a stream to manipulate, try out the following code from the lecture:

```
(define (enumerate-integers-from a)
  (cons-stream
   a
   (enumerate-integers-from (+ a 1))))

(define test2 (enumerate-integers-from 1))
(stream-car test2)
(stream-car (stream-cdr test2))
test2
(stream-cdr test2)
```

1. To make your debugging easier, write the Scheme function `(str-to-list str k)` which returns a list containing the first `k` elements of stream `str`. If `k` is greater than the number of elements of the stream `str`, you should return a list containing all of the stream elements, i.e. your code shouldn't give an error for any $\texttt{k} \geq 0$ as long as `str` is a stream.

2. To use streams in signal-processing-style code we need to define the primitives. Here are a few to implement that have analogues in list programming. All should be ordered consistent with their stream parameter, and all should take advantage of delayed evaluation by using `cons-stream`, `stream-car`, and `stream-cdr` instead of `cons`, `car`, and `cdr`. Note: some of these were shown on the slides, but be careful as the slides may have had small bugs and should be cleaned up.

   (a) `(stream-filter p str)` returns a stream made up of all of the elements of `str` for which the function `p` returns true.

   (b) `(stream-map f str)` returns a stream made up of the results of applying function `f` to each element of `str`. Unlike the `map` function in Scheme this will only work for functions of 1 argument.

   (c) `(stream-nth index str)` returns the nth element of a stream, where an index of 1 corresponds to the first element.

3. Using your functions above where appropriate, write the following stream functions:

   (a) `(scale-stream k str)` returns a stream made up of the results of multiplying each element of `str` by `k`.

   (b) `(add-streams str1 str2)` returns a stream made up by adding the elements of the two streams pairwise. E.g., if `str1` has the elements `(s1 s2 s3)` and `str2` has the elements `(t1 t2 t3)`, then `(add-streams str1 str2)` would have the elements `(s1+t1 s2+t2 s3+t3)`. If one stream is shorter than the other, act as if the shorter stream is extended by zeroes.

   (c) `(mult-streams str1 str2)` returns a stream made up by multiplying the elements of the two streams pairwise. E.g., if `str1` has the elements `(s1 s2 s3)` and `str2` has the elements `(t1 t2 t3)`, then `(mult-streams str1 str2)` would have the elements `(s1*t1 s2*t2 s3*t3)`. If one stream is shorter than the other, act as if the shorter stream is extended by zeroes.

   (d) `(append-streams str1 str2)` returns a stream containing all of the elements of `str1` followed by all of the elements of `str2`. (This should still work if `str1` is an infinite stream.)

4. Using your functions above where appropriate, write the following stream functions:

   (a) `(odd-factors-of k)` returns a stream made up all odd non-negative numbers that are evenly divisible by `k`, in ascending order. Write your solution to this in terms of your stream functions in signal-processing style.

   (b) `(partial-sums str)` returns a stream containing the partial sums of `str`, that is, if s1, s2, s3,s4,... are the elements of `str`, the result stream contains s1, s1+s2, s1+s2+s3, s1+s2+s3+s4, ... This function should produce a finite stream when given a finite stream of numbers.

   (c) `(square-stream)` returns a stream made up of the squares of the non-negative integers in ascending order.

Question for your consideration: This could be done as a stream-map, a stream-filter, or a stream-mult. What is the best (or worst) choice?