Streams

Recall that we have created a stream abstraction using `cons-stream`, `stream-car` and `stream-cdr`.  

```scheme
(define (cons-stream x (y lazy-memo))
  (lambda (msg)
    (cond  ((eq? msg 'stream-car) x)
        ((eq? msg 'stream-cdr) y)
                 (else (error "unknown message" msg)))))
```

```scheme
(define (stream-car s) (s 'stream-car))
(define (stream-cdr s) (s 'stream-cdr))
```

```scheme
(define (add-streams s1 s2)
  (cond ((null? s1) '())
        ((null? s2) '())
        (else (cons-stream (+ (stream-car s1) (stream-car s2))
                          (add-streams (stream-cdr s1) (stream-cdr s2)))))))
```

```scheme
(define (stream-filter pred str)
  (if (pred (stream-car str))
      (cons-stream (stream-car str) (stream-filter pred (stream-cdr str)))
      (stream-filter pred (stream-cdr str)))))
```

Lazy evaluation

What is the potential problem with the following implementation of `cons-stream`?

```scheme
(define (cons-stream x (y lazy-memo))
  (cons x y))
```

`cons` is a built-in procedure, it is not lazy and will force the evaluation of `y`.

Integers

Recall that we have defined a stream full of ones using `(define ones (cons-stream 1 ones))`

Can you remember how to define the stream of integers?

```scheme
(define integers (cons-stream 0 (add-stream integers ones)))
```

Use filter to define `odd-integers`, the stream of odd integers.

```scheme
(define odd-integers (stream-filter odd? integers))
```

Note that `(stream-filter even? odd-integers)` will result in an infinite loop.

Use mutually-recursive definitions to extract odd- and even-indexed elements of a stream

```scheme
(define (extract-even s) (cons-stream (stream-car s) (extract-odd (stream-cdr s))))
(define (extract-odd s) (extract-even (stream-cdr s)))
```

Write a procedure `(stream-find <key> <stream>)` that returns `<key>` when it finds it in the stream.

```scheme
(define (stream-find key s)
  (if (equal? key (stream-car s))
      (stream-find key (stream-cdr s)))
           
```

Draw a box-and-arrow diagram to trace a call to `(stream-find 5 odd-integers)`.

Do the same for the Eratosthenes sieve… ;)

Basic streams

Create a stream of Fibonacci numbers using `add-streams`.

```scheme
(define Fib (cons-stream 1 (cons-stream 1 (add-stream Fib (stream-cdr Fib)))))
```
Stream manipulation

Write a \((\text{stream-map2} \ <\text{operation}>\ <\text{stream1}>\ <\text{stream2}>)\) higher-order procedure.

\[
\begin{align*}
\text{(define \(\text{stream-map2} \ op \ s1 \ s2\))} \\
\quad \text{if \(\text{or} (\text{null?} \ s1) (\text{null?} \ s2)\) \(\'()\)} \\
\quad \text{cons-stream \(\text{op} (\text{stream-car} \ s1) (\text{stream-car} \ s2)\) \(\text{stream-map2} \ op (\text{stream-cdr} \ s1) (\text{stream-cdr} \ s2)\))}
\end{align*}
\]

Write a function \((\text{merge} \ <\text{s1}>\ <\text{s2}>)\) that merges two ordered streams of integers.

\[
\begin{align*}
\text{(define \(\text{merge} \ s1 \ s2\))} \\
\quad \text{cond \((\text{null?} \ s1) \ s2\) \(\text{null?} \ s2\) \s1} \\
\quad \text{if \(\text{<} (\text{stream-car} \ s1) (\text{stream-car} \ s2)\))} \\
\quad \text{cons-stream \(\text{stream-car} \ s1\) \(\text{merge} \ (\text{stream-cdr} \ s1) \ s2\)\)} \\
\quad \text{else} \\
\quad \text{cons-stream \(\text{stream-car} \ s2\) \(\text{merge}\ s1 \ (\text{stream-cdr} \ s2)\)\))}
\end{align*}
\]

Random stream

We want a stream of random number for some simulation software. What do these implementations do? Does it matter if evaluation is memoized or not?

\[
\begin{align*}
\text{(define \(\text{random-stream} \ (\text{cons-stream} \ (\text{random} \ 100) \ \text{random-stream})\))} \\
\text{We obtain a stream with always the same number. This is similar to the definition of ones.}
\end{align*}
\]

\[
\begin{align*}
\text{(define \(\text{make-random-stream} \ (\text{cons-stream} \ (\text{random} \ 100) \ \text{make-random-stream})\))} \\
\text{We obtain a stream of different random numbers. If the stream is memoized, the sequence of numbers stays the same. If we do not memoize, then the value of a particular element in the stream will be different each time we read it with stream-car.}
\end{align*}
\]

Power series

Recall from lecture that we can represent an infinite Power Series as a stream.

\[
\begin{align*}
\text{(define \(\text{powers} \ x\))} \ (\text{cons-stream} \ 1 \ (\text{scale-stream} \ x \ \text{powers} \ x))\)) \\
\text{(define \(\text{facts} \ (\text{cons-stream} \ 1 \ (\text{mult-streams} \ (\text{stream-cdr} \ ints) \ \text{facts})\))} \\
\text{(define \(\text{series-approx} \ \text{coeffs}\)) \ (\lambda (x)) \\
\quad \text{(\mult-streams} \ (\text{div-streams} \ \text{powers} \ x \ (\text{cons-stream} \ 1 \ \text{facts}) \ \text{coeffs}))) \\
\text{(define \(\text{power-series} \ \text{g}\))} \ (\lambda (x)) \ (\text{add-streams} \ \text{stream-accum} \ ((\text{series-approx} \ \text{g}) \ x))) \\
\text{(define \(\text{sine-coeffs} \ (\text{cons-stream} \ 0 \ (\text{cons-stream} \ 1 \ (\text{cons-stream} \ 0 \ (\text{cons-stream} \ -1 \ \text{sine-coeffs}))))\))} \\
\text{(define \(\text{cos-coeffs} \ (\text{stream-cdr} \ \text{sine-coeffs})\))}
\end{align*}
\]

Warm up: exponential

Define the power series for exponential

\[
\begin{align*}
\text{(define \(\text{exp-coef}\))} \ \text{ones}
\end{align*}
\]

Differentiation and Integration

Our power-series streams represent functions, so we can perform usual operations on functions. In particular, we can compute the derivative of the corresponding function. Use the fact that the stream is a polynomial.

\[
\begin{align*}
\text{(define \(\text{differentiate-series} \ \text{s}\))} \ \text{stream-cdr} \ s\)) \\
\text{You can similarly define the integral of a power series}
\end{align*}
\]

\[
\begin{align*}
\text{(define \(\text{integral-series} \ \text{init-value} \ \text{s}\))} \ \text{cons-stream} \ \text{init-value} \ \text{s}\)) \\
\text{Use this to define \(e^x\). Hint, what is the derivative (or integral) of \(e^x\)?}
\end{align*}
\]

\[
\begin{align*}
\text{(define \(\text{exp-coef}\))} \ \text{cons-stream} \ 1 \ \text{integral-series} \ \text{exp-coef})\))
\end{align*}
\]
You can use a similar strategy to define cosine and sine using mutually-recursive definitions.

```scheme
(define cos-coef (integral-series 1 (scale-stream -1 sine-coef)))
(define sine-coef (integral-series 0 cos-coef))
```

**Function multiplication**

Another operation is function multiplication. This involves multiplying two infinite polynomials, which is not the same as mul-streams, as that only does element-wise multiplication.

**Hint:** reduce the problem to that of polynomial by folding in the factorial coefficients.

```scheme
(define (mul-poly p1 p2)
  (add-stream (stream-scale (stream-car p1) p2)
              (cons-stream 0 (mul-poly (stream-cdr p1) p2)))))

(define (mul-series s1 s2)
  (mult-stream fact
               (mul-poly (div-stream s1 fact) (div-stream s2 fact))))
```

Then this should look interestingly simple:

```scheme
(add-streams (mul-series sine sine) (mul-series cosine cosine))
```