Lazy Evaluation & Streams MIT 6.001 Recitation

Streams

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

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

(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))))))
(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`?

```
(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?

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

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

```
(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

```
(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.

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

```
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 ...)
(define Fib (cons-stream 1 (cons-stream 1 (add-stream Fib (stream-cdr Fib)))))
```

Stream manipulation

Write a `(stream-map2 <operation> <stream1> <stream2>)` higher-order procedure.

```
(define (stream-map2 op s1 s2)
  (if (or (null? s1) (null? s2)) '()
    (cons-stream (op (stream-car s1) (stream-car s2))
      (stream-map2 op (stream-cdr s1) (stream-cdr s2)))
```

Write a function `(merge <s1> <s2>)` that merges two ordered streams of integers.

```
(define (merge s1 s2)
  (cond ((null? s1) s2)
        ((null? s2) s1)
        ((< (stream-car s1) (stream-car s2))
          (cons-stream (stream-car s1) (merge (stream-cdr s1) s2)))
        (else
          (cons-stream (stream-car s2) (merge s1 (stream-cdr s2))))))
```

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?

```
(define random-stream (cons-stream (random 100) random-stream))
```

We obtain a stream with always the same number. This is similar to the definition of ones.

```
(define (make-random-stream) (cons-stream  (random 100) (make-random-stream)))
```

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.

Power series

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

```
(define (powers x) (cons-stream 1 (scale-stream x (powers x))))
(define facts (cons-stream 1 (mult-streams (stream-cdr ints) facts)))
(define (series-approx coeffs) (lambda (x)
  (mult-streams (div-streams (powers x) (cons-stream 1 facts)) coeffs)))
(define (stream-accum str)
  (cons-stream (stream-car str) (add-streams (stream-accum str) (stream-cdr str))))
(define (power-series g) (lambda (x) (stream-accum ((series-approx g) x))))

(define sine-coeffs (cons-stream 0 (cons-stream 1 (cons-stream 0 (cons-stream –1 sine-coeffs)))))
(define cos-coeffs (stream-cdr sine-coeffs))
```

Warm up: exponential

Define the power series for exponential

```
(define exp-coef ones)
```

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.

```
(define (differentiate-series s) (stream-cdr s))
```

You can similarly define the integral of a power series

```
(define (integral-series init-value s) (cons-stream init-value s))
```

Use this to define e^x. Hint, what is the derivative (or integral) of e^x?

```
(define exp-coef (cons-stream 1 (integral-series exp-coef)))
```

You can use a similar strategy to define cosine and sine using mutually-recursive definitions.

```
(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.

```
(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 p1 fact) (div-stream p2 fact))
```

Then this should look interestingly simple:

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