MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 24 Solutions — 5/12/2006
Streams

Streams

In a lazy evaluator:

1.
   ```
   (define (cons-stream x (y lazy-memo))
     (cons x y))
   ```
2. `(define stream-car car)`
3. `(define stream-cdr cdr)`

In an applicative order evaluator (DrScheme) using the `delay` special form:

1. `(cons-stream a b)` - equivalent to `(cons a (delay b))`[1]
2. `(stream-car c)` - equivalent to `(car c)`
3. `(stream-cdr c)` - equivalent to `(force (cdr c))`

Simple Streams:

Zeros: (0 0 0 0 0 0 ....

```
(define zeros (cons-stream 0 zeros))
```

Ones: (1 1 1 1 1 1 ....

```
(define ones (cons-stream 1 ones))
```

Natural numbers (called ints): (1 2 3 4 5 6 ....

```
(define ints (cons-stream 1 (add-streams ones ints)))
```

[1]Since cons-stream must be a special form, you can't `define` it, but the following will work in DrScheme if you want to try these without using l-eval: `(define-macro cons-stream (lambda (car cdr) (list 'cons car (list 'delay cdr))))`

Stream operators

We'd like to be able to operate on streams to modify them and combine them with other streams. For example, to do element-wise addition or multiplication:

```
(define (add-streams s1 s2) (map2-stream + s1 s2))
(define (mul-streams s1 s2) (map2-stream * s1 s2))
(define (div-streams s1 s2) (map2-stream / s1 s2))
```

Write map2-stream:

```
(define (map2-stream op s1 s2)
  (cons-stream (op (stream-car s1) (stream-car s2))
               (map2-stream op (stream-cdr s1) (stream-cdr s2))))
```

Another possible operation is multiplying every element of the stream by a constant factor:

```
(define (scale-stream x s)
  (cons-stream (* x (stream-car s))
               (scale-stream x (stream-cdr s))))
```

Define the stream of fibonacci numbers:

```
(define fibs (cons-stream 0
               (cons-stream 1
   (add-streams fibs
     (stream-cdr fibs)))))
```

Implement the stream of factorials, which goes `(1 1 2 6 24 120 ...`:

```
(define facts (cons-stream 1 (mul-streams ints facts)))
```

Power Series

We can approximate functions by summing terms of an appropriate power series. A power series has the form:

$$\sum a_n x^n = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

By selecting appropriate a_n, the series converges to the value of a function. One particularly useful function for which this is the case is e^x which has the following power series:

$$e^x = 0! + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Since power series involve an infinite summation, of which we might only care about the first couple terms, they are an excellent problem to tackle with streams. The stream will encode the coefficients a_n. For example, to represent the function $f(x) = 3$, we'd use a stream whose first element was 3, and the rest are zeros, so that $a_0 = 3$, and $a_1, a_2, \ldots = 0$. The following two procedures come in handy:

```
(define (powers x)
  (cons-stream x (scale-stream x (powers x))))

(define (sum-series s x n)
  (define (sum-helper s sum n)
    (if (= n 0)
        sum
        (sum-helper (stream-cdr s) (+ sum (stream-car s)) (- n 1))))
  (sum-helper (mul-streams s (powers x)) 0 n))
```

Write an expression that computes a stream to represent the power series that converges to $f(x) = 2x + 5$:

```
(define two-x-plus-five (cons-stream 5 (cons-stream 2 zeros)))
```

Write an expression that computes the stream for e^x:

```
(define e-to-the-x (div-streams ones facts))
```

To compute e^5 using 20 terms, we'd call `(sum-series e-to-the-x 5 20)`.

Since the stream represents a function, we can write operations which work on functions and try to implement them in terms of the coefficients of the series. One such operation is integration. The integral of an infinite polynomial is also an infinite polynomial, but the coefficients will be different. In particular, we'll want our integration function to return a stream whose constant term (first element) is missing, as it can't actually compute it from the series.

```
(define (integrate-series s)
  (div-streams s ints))
```

Write a new definition of e^x using `integrate-series` (Hint: what is the integral of e^x?)

```
(define e-to-the-x (cons-stream 1 (integrate-series e-to-the-x)))
```

Given that we can build e^x this way, implement sin and cos in a similar fashion:

```
(define sine (cons-stream 0 (integrate-series cosine)))
(define cosine (cons-stream 1 (scale-stream -1
                                 (integrate-series sine))))
```

Bonus Round Problem: Another operation is function multiplication. This involves multiplying two infinite polynomials, which is not the same as `mul-streams`, as that only does elementwise multiplication.

```
(define (mul-series s1 s2)
  (cons-stream (* (stream-car s1) (stream-car s2))
     (add-streams (scale-stream (stream-car s1)
                                (stream-cdr s2))
                  (mul-series (stream-cdr s1) s2))))
```

Then this should look interestingly simple:

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