MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 16 — 10/31/2007 Solutions
Streams

Delay & Force

- `(delay` *expr*`)`: returns a *promise* to evaluate *expr* sometime later if asked. Special Form.
- `(force` *promise*`)`: evaulate the promise created earlier with `delay`.

One possible implementation for delay would be to turn a call to delay into a thunk – a procedure of no arguments. Force then applies this procedure to no arguments.

Thunks may be memoized: rather then evaluate the promise more than once, remember the value after the first evaluation and return it again if asked.

For example, the following definition of `memoize` will take in one thunk, and return another that is memoized.

```
(define (memoize thunk)
  (let ((need-val #t)
        (val 'whatever))
    (lambda ()
      (if need-val
          (begin
            (set! val (thunk))
            (set! need-val #f)))
      val)))
```

Problem: Write an expression that will return true if DrScheme's implementation of `delay` and `force` use memoization, and false otherwise.

```
(let* ((a 0)
       (p (delay (set! a (+ a 1)))))
  (force p)
  (force p)
  (= a 1))
```

Infinite Streams

Delay and Force can be used to build streams with no determined end – since the elements don't exist until they're needed, there's no reason to define a length on construction:

1. `(cons-stream a b)` - Special form 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)))`

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))))
```

[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 examples:

```
(define-macro cons-stream (lambda (car cdr) (list 'cons car (list 'delay cdr))))
(define (stream-car c) (car c))
(define (stream-cdr c) (force (cdr c)))
```

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

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

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.

We will construct a stream that consists of successively improved approximations of e^x, in several steps.

To begin with, construct a stream that consists of the coefficients a_0, a_1, a_2 and so on, for the expansion of e^x:

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

Next, we need a stream that consists of powers of x, which can be defined as:

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

We also need a procedure which takes in a stream of coefficients, and produces a stream of partial sums:

```
(define (sum-series s x)
  (sum-stream (mul-streams s (powers x))))
```

The one missing piece here is `sum-stream`, which takes a single strean, and retuns a stream that consists of just the first element, followed by the sum of the first two, then the sum of the first three, and so on.

Define $\texttt{sum} - \texttt{stream}$:

```
(define (sum-stream s)
  (let ((a (stream-car s)))
    (cons-stream
     a
     (add-streams (scale-stream a ones )
                  (sum-stream (stream-cdr s))))))
```

With sum-streams defined, we can define e^x as follows:

```
(define (e-to-the-x x)
  (sum-series
   e-to-the-x-coeffs
   x))
```

In DrScheme, printing out the first several elements of `(e-to-the-x 1)` converted to decimal notation results in:

```
(print-stream (scale-stream 1.0 (e-to-the-x 1)) 20)
(1.0                 2.0
 2.5                 2.6666666666666665
 2.7083333333333335  2.716666666666667
 2.7180555555555554  2.7182539682539684
 2.71827876984127    2.7182815255731922
 2.7182818011463845  2.718281826198493
 2.7182818282861687  2.718281828446759
 2.7182818284582297  2.7182818284589945
 2.7182818284590424  2.718281828459045
 2.718281828459045   2.718281828459045)
```