Delay & Force

- (delay expr): returns a promise to evaluate expr sometime later if asked. Special Form.
- (force promise): evaluate 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.

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. \((\text{cons-stream } a \ b)\) - Special form equivalent to \((\text{cons } a \ (\text{delay } b))\) \(^1\)

2. \((\text{stream-car } c)\) - equivalent to \((\text{car } c)\)

3. \((\text{stream-cdr } c)\) - equivalent to \((\text{force } (\text{cdr } c))\)

Simple Streams:

Zeros: \((0 \ 0 \ 0 \ 0 \ 0 \ ...\)

\(\text{(define zeros}\)

Ones: \((1 \ 1 \ 1 \ 1 \ 1 \ ...\)

\(\text{(define ones}\)

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

\(\text{(define ints}\)

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:

\(\text{(define (add-streams s1 s2) (map2-stream + s1 s2))}\)
\(\text{(define (mul-streams s1 s2) (map2-stream * s1 s2))}\)
\(\text{(define (div-streams s1 s2) (map2-stream / s1 s2))}\)

Write map2-stream:

\(\text{(define (map2-stream op s1 s2)}\)

---

\(^1\)Since \text{cons-stream} must be a special form, you can’t \text{define} it, but the following will work in DrScheme if you want to try these examples:

\(\text{(define-macro cons-stream (lambda (car cdr) (list 'cons car (list 'delay cdr)))}}\)
\(\text{(define (stream-car c) (car c))}\)
\(\text{(define (stream-cdr c) (force (cdr c)))}\)
Another possible operation is multiplying every element of the stream by a constant factor $c$:

$$\text{(define (scale-stream c s)}$$

Implement the stream of factorials, which goes $1 \ 1 \ 2 \ 6 \ 24 \ 120 \ldots$:

$$\text{(define facts}$$

**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 $e^x$:

$$\text{(define e-to-the-x-coeffs}$$

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

$$\text{(define (powers x)}$$

$$\text{(cons-stream x (scale-stream x (powers x))))}$$
We also need a procedure which takes in a stream of coefficients, and produces a stream of partial sums:

\[
\text{(define (sum-series s x)} \\nonumber \\
\quad \quad \text{(sum-stream (mul-streams s (powers x)))}) \nonumber
\]

The one missing piece here is \texttt{sum-stream}, which takes a single stream, and returns 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-stream}:

\[
\text{(define (sum-stream s)} \nonumber \\
\]

With \texttt{sum-streams} defined, we can define \(e^x\) as follows:

\[
\text{(define (e-to-the-x x)} \nonumber \\
\quad \text{(sum-series)} \nonumber \\
\quad \quad \text{e-to-the-x-coeffs \nonumber} \\
\quad \quad \text{x))} \nonumber
\]

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

\[
\text{(print-stream (scale-stream 1.0 (e-to-the-x 1)) 20)} \nonumber \\
\text{(1.0 2.0 \nonumber} \\
\text{2.5 2.6666666666666665 \nonumber} \\
\text{2.7083333333333335 2.7166666666666667 \nonumber} \\
\text{2.7180555555555554 2.7182539682539684 \nonumber} \\
\text{2.71827876984127 2.718281557319220} \nonumber \\
\text{2.7182818011463845 2.7182818261998493} \nonumber \\
\text{2.7182818286061687 2.718281828466759} \nonumber \\
\text{2.7182818284582297 2.7182818284589945} \nonumber \\
\text{2.7182818284590424 2.718281828459045} \nonumber \\
\text{2.718281828459045 2.718281828459045)} \nonumber
\]