Lazy Evaluation

Here are some new Scheme expressions:

- (delay exp) returns an object called a *promise,* without evaluating exp*.*
- (force promise) evaluates the promised expression and returns its value.

If a promised expression has been evaluated once, forcing it again returns its value without re-evaluating it.

For example

```
> (define foo
        (delay (begin
                (display "Oh, goody I'm being evaluated!\n")
                2)))
> (force foo)  => Oh goody I'm being evaluated!
                  2
> (force foo) => 2
> (force foo) => 2
```

Another example:

```
> (define d  (+ z 3)) => error: z is undefined
> (define d (delay (+ z 3)))
> (define z 5)
> (force d) => 8
> (define z 23)
> (force d) => 8
```

Now, how could we implement delay and force?

The only place in standard Scheme where we can give an expression without immediately evaluating it is in the body of a lambda expression.

Try this:

```
> (define d (lambda () (+ x 5)))
> (set! x 23)
> (d)
```

A lambda expression with no arguments is a wrapper that delays evaluation; such a lambda expression is sometimes called a *thunk*.

To avoid re-evaluating the delayed expression, we can store the expression's value in an internal environment and just return it when we need it --

```
(delay exp)
```

is equivalent to

```
(let ( [thunk (lambda () exp)]
       [value 0]
       [evaluated? #f] )
    (lambda ( )
        (if (not evaluated)
            (begin
                (display "evaluating\n")
                (set! value (thunk))
                (set! evaluated? #t)
                value)
            value)))
```

We can then define (force promise) as (promise)

Note that (delay exp) cannot be defined as a procedure, since
(f exp)
always evaluates exp.

delay is created as a type of expression, just like if and lambda.

Streams

A stream is a list-like structure, with some of the values actually present and the rest contained in a promise. This can be used to represent infinite sequences. For example, we might have a stream of positive integers:

(1 2 3 <promise>)

Each time we evaluate the promise we get the next value of the stream, and another promise to produce the rest:

(1 2 3 4 <new promise>)

Streams are built from three primitive operators:

- (cons$ x y) This is the same as (cons x (delay y)). Of course, like delay it can't be written as a lambda expression.
- (car$ s) This is

```
(define car$ (lambda (s)
                (if (promise? s)
                    (car$ (force s))
                    (force (car s)))))
```

- (cdr$ s) This is

```
(define cdr$ (lambda (s)
                (if (promise? s)
                    (cdr$ (force s))
                    (force (cdr s)))))
```

These three are contained in file "stream.ss", which we will use for all of our work with streams.

delay and force are native to Scheme, car$, cdr$ and cons$ are not. All of our work with streams will use the header

```
(require "stream.ss")
```

and this file will need to be in the same directory as our programs.

In addition to car$, cdr$ and cons$, stream.ss defines a number of helper procedures for working with streams:

- (nth$ n s) returns the nth cdr of stream s
- (show$ n s) returns a list (a real list) of the first n elements of stream s, followed by a promise for the rest of s.
- (printn$ s n) prints the first n elements of s
- (print$ s) prints the first 10 elements of s, asks if you want more, and responds.
- (+$ s1 s1) returns a new stream that adds the corresponding entries of s1 and s2 together.
- (filter$ p s) takes a predicate and a stream and returns a new stream consisting of the elements of s that satisfy the predicate.
- (map$ f s) maps procedure f (a function of 1 variable) to stream s

- (append$ s1 s2) appends stream s2 after s1; this doesn't have much effect unless s1 is finite.
- (fold$ rec-case base-case base-test s) does fold with stream s.

All of these helper functions are easy to define. For example

```
(define map$ (lambda (f s)
        (cons$ (f (car$ s)) (map$ f (cdr$ s))))))

(define nth$ (lambda (n s)
        (cond
                [(= 0 n) s]
                [else (nth$ (- n 1) (cdr$ s))])))
```

Example:

```
(define IntsFrom$ (lambda (n)
        (cons$ n (IntsFrom$ (+ n 1)))))

(define Ints$ (IntsFrom$ 0))
(define Evens$ (map$ (lambda (x) (* 2 x))  Ints$)
(define Ones$ (cons$ 1 (Ones$)))
(define Odds$ (+$ Evens Ones$))
```