Streams

Suppose X is a list. What is the first element of (cons 1 X)?

What is the second element of (cons 1 X)?

What happens if you type this int Dr. Racket:

```
(define Ones  (cons 1 Ones))
```

Why is this different from

```
(define fact (lambda (x)
                         (if (= x 0) 1(* x  (fact (- x 1)))))
```

Suppose this was a legitimate definition:

```
        (define Ones (cons 1 Ones))
```

What would be the first element of Ones?

```
(define Ones (cons 1 Ones))
```

What would be the second element of Ones?

What would be the third element of Ones?

If this definition was allowed:

```
(define Ones (cons 1 Ones))
```

what would Ones be?

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 "streams.rkt", 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 "streams.rkt")
```

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

In addition to car$, cdr$ and cons$, streams.rkt 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$))
```