Chapter 3a
Streams

Topics of Chapters 1 & 2

	data	procedures
primitive	X	X
combinations	X	X
abstraction	X	X

But this is not sufficient for organizing large systems. Now we study modularity.

Chapter 3: Forms of Modularity

Organize a system
in a modular way

Raises the linguistic issue of "state"

According to the objects that live in the system

or

According to the streams of information that flow in the system

Raises the linguistic issue of "delayed evaluation"

Remember Lists as Standard Interfaces

Compute sum of all prime numbers in an interval

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

Standard iterative style (efficient)

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

Using list operations (elegant but painfully inefficient)

And a second list is created!

All integers are actually stored

Overhead Can Be Outrageous

Find the second prime in the interval [10.000, 1.000.000]

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

A million integers are stored.
Most of them ignored

Streams to the Rescue

Streams are lazy lists

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

make and use streams in the same way as lists (sequences)

```
the-empty-stream
stream-null?
```

∃

c.f. make-fraction

The difference is the time at which the elements are evaluated. With ordinary lists, both the car and the cdr are evaluated at construction time. With streams, the cdr is evaluated at selection time.

Very Similar to Lists

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred
                                     (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream))))
```

Delayed Objects in Scheme

special form

```
(delay <exp>)
```

```
(force <exp>)
```

procedure

syntactic sugar

special form

```
(cons-stream <a> <b>)
```

⇔

```
(cons <a> (delay <b>))
```

```
(stream-cdr <exp>)
```

⇔

```
(force (cdr <exp>))
```

procedure

Back to the Example

Find the second prime in the interval [10.000, 1.000.000]

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

only when needed

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                 (stream-enumerate-interval 10000 1000000))))
```

"original expression"

```
(stream-enumerate-interval 10000 1000000)

                ⟹ (cons 10000
                        (delay (stream-enumerate-interval 10001 1000000)))


(stream-filter prime?                    (prime? 10000) ⟹ #f
                 (cons 10000
                       (delay (stream-enumerate-interval 10001 1000000))) )

(stream-filter prime?                    (prime? 10001) ⟹ #f
                 (cons 10001
                       (delay (stream-enumerate-interval 10002 1000000))) )

...

(stream-filter prime?                    (prime? 10007) ⟹ #t
                 (cons 10007
                       (delay (stream-enumerate-interval 10008 1000000))) )
```

```
(stream-filter prime?                    (prime? 10007) ⟹ #t
                (cons 10007
                      (delay (stream-enumerate-interval 10008 1000000))) )
```

```
⟹ (cons 10007
        (delay
          (stream-filter
           prime?
           (cons 10008
                 (delay
                   (stream-enumerate-interval 10009
                                              1000000))))))
```

```
(stream-cdr (cons 10007
                  (delay
                    (stream-filter
                     prime?
                     (cons 10008
                           (delay
                             (stream-enumerate-interval 10009
                                                        1000000))))) )
```

from the original expression

```
||
```

```
(stream-filter prime?                    (prime? 10009) ⟹ #t
                (cons 10009
                      (delay (stream-enumerate-interval 10010 1000000))) )
```

```
⟹ (cons 10009
        (delay
          (stream-filter
           prime?
           (cons 10010
                 (delay
                   (stream-enumerate-interval 10011
                                              1000000))))))
```

```
(stream-car (cons 10009
                  (delay
                    (stream-filter
                     prime?
                     (cons 10010
                           (delay
                             (stream-enumerate-interval 10011
                                                        1000000))))) )
⟹ 10009
```

Implementing Delay&Force

syntactic sygar

```
(delay <exp>)
```

⇔

```
(lambda () <exp>)
```

Called 'a thunk'

delay evaluation + capture environment!

```
(force <exp>)
```

⇔

```
(<exp>)
```

eval in correct environment!

Infinite Streams

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))

(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))

(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

Example: The Sieve of Eratosthenes

```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

Defining Streams Implicitly

without a "generating" procedure

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

(define (add-streams s1 s2)
  (stream-map + s1 s2))

(define integers (cons-stream 1 (add-streams ones integers)))

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

```
(define (stream-map proc . argstreams)
  (if (stream-null? (car argstreams))
      the-empty-stream
      (cons-stream
        (apply proc (map stream-car argstreams))
        (apply stream-map
               (cons proc (map stream-cdr argstreams))))))
```

generalized stream-map

Formulating Iterations as Stream Processes

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))

(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
                 (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses)))
  guesses)
```

represent state as a “timeless” stream of values rather than as a set of variables to be updated

Newton's method revisited

```
> (display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

Chapter 3a