The Joys of Scheme Cont'd

Daniel P. Friedman
Computer Science Departmert
Indiana University

XX Conferencia Latinoamericana de Informatica.
PANEL 1994.
Mexico City

Goals

- Demonstrate how to build infinite lists.
  - Show that working with infinite lists is simple.
  - Present nearly unbelievable programs.

Outline

- Identifying Characteristics of Streams
- Operations on Streams
  - stream-cons, stream-car, stream-cdr
- "Promises Promises"
- Implementation of Call-by-Name Streams
- General Stream Utilities
  - stream->list, show
  - stream-ref, stream-filter-out
  - stream-map, stream-ormap
- Programs Using Call-by-Name Streams
  - positives, multiples-of, powers-of
- Implementation of Memoized Streams
- Watch the Stream Grow

- Programs Using Memoized Streams
  - odds-from-5
  - factorials, fibonaccis
  - primes (the easy way)
- Data Recursion
  - positives, evens, odds, factorials, fibonaccis
  - primes (a harder way)
  - primes (a clever way)
  - pi and The Prime Number Theorem
  - prime? and an unforgetable prime number
- Closing Remarks

An Overview of Streams

Identifying Characteristics

- Streams are Infinite Lists
  - Items can be added to the front.
  - Items cannot be added at the "rear."
  - Searching can lead to infinite loops.

Operations on Streams

- (stream-car stream)
  - The first element of a stream
- (stream-cdr stream)
  - All but the first element of a stream
- (stream-cons item stream)
  - Extends the stream by one item.

“Promises Promises”

... A promise is a record with one field: thunk.

```
(define promise-tag "promise")

(define promise
  (lambda (thunk)
    (vector promise-tag thunk)))

(define promise?
  (lambda (p)
    (and (vector? p)
         (= (vector-length p) 2)
         (eq? (vector-ref p 0) promise-tag))))

(define promise->thunk
  (lambda (p)
    (vector-ref p 1)))
```

Implementation of Call-by-Name Streams

... In terms of the basic syntax.

(stream-cons item s) ⇒ (cons item (promise (lambda () s)))

```
(define stream-car
  (lambda (s)
    (car s)))

(define stream-cdr
  (lambda (s)
    ((promise->thunk (cdr s)))))
```

—————— End of Implementation ——————-

```
(define random-digits-generator
  (lambda ()
    (stream-cons (random 10) (random-digits-generator))))
(define random-digits (random-digits-generator))
```

(show random-digits 10) → (7 4 9 7 0 4 8 6 6 3 . . .)
(show random-digits 10) → (7 5 4 2 1 3 6 1 5 3 . . .)

General Stream Utilities

```
(define stream->list
  (lambda (s n)
    (cond
      ((zero? n) '())
      (else (cons
              (stream-car s)
              (stream->list (stream-cdr s) (sub1 n)))))))

(define show
  (lambda (s n)
    (display "(")
    (for-each
      (lambda (v) (write v) (display " "))
      (stream->list s n))
    (display "... )")
    (newline)))
```

```
(define stream-ref
  (lambda (s n)
    (cond
      ((zero? n) (stream-car s))
      (else (stream-ref (stream-cdr s) (sub1 n))))))

(define stream-filter-out
  (lambda (f s)
    (if (f (stream-car s))
        (stream-filter-out f (stream-cdr s))
        (stream-cons (stream-car s)
          (stream-filter-out f (stream-cdr s))))))
```

```
(define stream-map
  (lambda (f . s*)
    (stream-cons
      (apply f (map stream-car s*))
      (apply stream-map f (map stream-cdr s*)))))

(define stream-ormap
  (lambda (f quit s)
    (and (not (quit (stream-car s)))
         (or (f (stream-car s))
             (stream-ormap f quit (stream-cdr s))))))
```

Programs Using Call-by-Name Streams

```
(define integers-generator
  (lambda (n)
    (stream-cons n (integers-generator (add1 n)))))

(define positives (integers-generator 1))
```

(stream->list positives 5) → (1 2 3 4 5)

(show positives 5) → (1 2 3 4 5 . . .)

(stream-ref positives 5) → 6

(show (stream-filter-out even? positives) 4) → (1 3 5 7 . . .)

(show (stream-map add1 positives) 5) → (2 3 4 5 6 . . .)

```
(define multiples-of
  (lambda (n)
    (letrec
      ((loop
         (lambda (k)
           (stream-cons k (loop (+ k n))))))
      (loop n))))
(show (multiples-of 2) 5) → (2 4 6 8 10 ... )

(define powers-of
  (lambda (n)
    (letrec
      ((loop
         (lambda (k)
           (stream-cons k (loop (* k n))))))
      (loop 1))))
(stream-ref (powers-of 2) 50) → 1125899906842624
```

Implementation of Memoized Streams

```
(define stream-cdr
  (lambda (s)
    (if (promise? (cdr s))
        (set-cdr! s ((promise->thunk (cdr s)))))
    (cdr s)))
```

——————— End of Implementation ———————-

```
(define random-digits (random-digits-generator))
```

(show random-digits 10) → (6 4 5 0 8 6 3 7 6 . . .)
(show random-digits 10) → (6 4 5 0 8 6 3 7 6 . . .)

Watch the Stream Grow

(define random-digits (random-digits-generator))

random-digits → (3 . #("promise" `<procedure>`))

(show random-digits 2) → (3 6 . . .)

random-digits → (3 6 1 . #("promise" `<procedure>`))

(show random-digits 5) → (3 6 1 5 3 . . .)

random-digits → (3 6 1 5 3 0 . #("promise" `<procedure>`))

Programs Using Memoized Streams

```
(define odds-from-5
  (stream-cdr
    (stream-cdr
      (stream-filter-out even? positives))))
(show odds-from-5 10) → (5 7 9 11 13 15 17 19 21 23 ... )

(define factorials-generator
  (lambda (n a)
    (stream-cons a (factorials-generator (add1 n) (* n a)))))
(define factorials (factorials-generator 1 1))
(show factorials 7) → (1 1 2 6 24 120 720 ... )

(define fibonaccis-generator
  (lambda (a b)
    (stream-cons a (fibonaccis-generator b (+ a b)))))
(define fibonaccis (fibonaccis-generator 1 1))
(show fibonaccis 7) → (1 1 2 3 5 8 13 ... )
```

All the Prime Numbers (The Easy Way)

```
(define primes-generator
  (lambda (s)
    (stream-cons
      (stream-car s)
      (primes-generator
        ((sieve (stream-car s)) (stream-cdr s))))))

(define sieve
  (lambda (n)
    (lambda (s)
      (stream-filter-out (evenly-divides-by n) s))))

(define evenly-divides-by
  (lambda (n)
    (lambda (k)
      (zero? (remainder k n)))))

(define primes
  (primes-generator (stream-cdr positives)))

(show primes 10) → (2 3 5 7 11 13 17 19 23 29 ... )
```

Data Recursion

```
(define positives
  (stream-cons 1 (stream-map add1 positives)))

(define evens
  (stream-cons 0 (stream-map (lambda (x) (+ x 2)) evens)))

(define odds
  (stream-cons 1 (stream-map (lambda (x) (+ x 2)) odds))))

(define factorials
  (stream-cons 1 (stream-map * positives factorials)))

(define fibonaccis
  (stream-cons 1
    (stream-cons 1
      (stream-map + fibonaccis (stream-cdr fibonaccis)))))
```

All the Prime Numbers (A Harder Way)

```
(define primes
  (stream-cons 2
    (stream-cons 3
      (stream-filter-out
        (lambda (n)
          (has-prime-divisor? n (sqrt n)))
        odds-from-5))))

(define has-prime-divisor?
  (lambda (n bound)
    (stream-ormap
      (lambda (p) ((evenly-divides-by p) n))
      (lambda (p) (> p bound))
      (stream-cdr primes))))
```

All the Prime Numbers (A Clever Way)

```
(define primes
  (stream-cons 2
    (stream-cons 3
      (letrec
        ((builder
            (lambda (n bound s)
              (let* ((next (stream-car s)) (max (* next next)))
                (letrec
                  ((loop
                     (lambda (n)
                       (cond
                         ((= n max)
                          (builder (+ n 2) next (stream-cdr s)))
                         ((has-prime-divisor? n bound)
                          (loop (+ n 2)))
                         (else (stream-cons n (loop (+ n 2))))))))
                  (loop n))))))
        (builder 5 2 (stream-cdr primes))))))
```

Prime Number Theorem

$$\lim_{n\to\infty} \pi(n)\frac{\log n}{n} = 1$$

```
(define pi
  (lambda (n)
    (letrec
      ((count (lambda (s)
                (cond
                  ((> (stream-car s) n) 0)
                  (else (add1 (count (stream-cdr s))))))))
      (count primes))))
(* (pi 1000) (/ (log 1000) 1000)))) → 1.160502886868999

(define prime?
  (lambda (n)
    (or (= n 2)
        (and (> n 2)
             (odd? n)
             (not (has-prime-divisor? n (floor (sqrt n))))))))
(prime? 1234567891) → #t
```