函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第十讲

时间和流

●前面用状态和赋值做模拟，看到了赋值带来的复杂性
- 本节考虑用另一种方式模拟状态变化
- 希望避免赋值带来的一些复杂性问题

●本质上说，基于状态模拟的复杂性来自现实世界中被模拟的现象前面的考虑是：
- 用有局部状态的计算对象模拟现实中状态可能变化的对象
- 用计算机系统随时间的变化模拟现实世界的变化
- 通过给具有局部状态的对象赋值，实现计算机系统里的状态变化

●现在考虑另一可能性：把随时间变化的量表示为一个随时间变化的函数x(t)。这样可得到两种（角度的）观察：
- 看x 在一系列具体时刻的值，它是一个随时间变化的变量
- 看x 的整个历史，它就是一个时间t 的函数，并没有变化

时间和流

- 下面考虑离散时间上的函数
  - 离散时间的函数可以用无穷长的序列模拟。称其为流（stream）
  - 研究如何用流模拟状态变化，模拟一个系统随时间变化的历史
- 为做这种模拟，需要引进一种也称为流的数据结构
  - 流可以看作无穷长的表（序列）
  - 不能用表（list）**来表示流，因为流可能无穷长**
  - 下面将采用延时求值技术表示任意（潜无穷）长的序列
- 下面将用流模拟一些包含状态的系统，构造一些有趣的模型
  - 不需要赋值和变动数据结构，因此可以避免赋值带来的问题
  - 流也不是万能的，使用上也有本质性的困难
  - 实际上，流不可能解决随时间变化中的复杂问题

序列和表

●求区间[a,b]中的素数之和的迭代写法：

```
(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))
```

●求区间[a,b]中的素数之和的列表写法：

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
;要生成区间[a,b]，和该区间素数的列表，效率太低
```

序列和表

●求区间[10000,1000000]中的第二个素数：

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

要生成[10000,1000000]整个序列，效率无法接受

能否[10000,1000000]中的元素，用到的时候才去生成？

流

●流是一种列表，但是其在构造时，cdr并没有被求出，只有真正用到其cdr时，才会对其cdr求值。

●表的两个成分（car/cdr）都在构造表的时候求值,流的cdr 部分将推迟到选取该部分时再求值

●流的构造函数和选择函数：

构造函数: **cons-stream**

选择函数:

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

只有执行 **stream-cdr**时，流的cdr才会被求值

●空流

```
(define the-empty-stream '())
(define (stream-null? stream)  (null? stream))
```

流

●流的其他函数：

```
(define (stream-ref s n) ;取流中第n个元素
  (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-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

流的实现

- **scheme的特殊形式：delay**

求值(delay <e>) 时不求值<e>，而是返回一个延时对象，延时对象是对未来计算<e>的一个承诺。

```
(delay (+ 5 3))
=> #<promise:unsaved-editor7983:2:0>
```

- 用基本过程force对延时对象进行求值：

```
(define x (delay (+ 5 3)))
(force x)
=> 8
```

流的实现

- delay 是否能实现为普通过程?

```
(define (delay p)
        (lambda () p))

(define (force x)
        (x))

(delay (display "ok"))
```

=> ??

流的实现

- delay 是否能实现为普通过程?

```
(define (delay p)
        (lambda () p))

(define (force x)
        (x))

(delay (display "ok"))

=> ok#<procedure>
```

delay不能实现为普通过程，因为普通过程求值时，会先对参数求值。应实现为特殊形式。force是普通过程。

流的实现

- cons-stream 是否能实现为普通过程?(此处delay为特殊形式)

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s =>?
(cdr s) =>?
(force (cdr s)) =>?
(define x (delay (display 'hh)))
x =>?
(force x) =>?
```

流的实现

- cons-stream 是否能实现为普通过程?

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s => hey'(1 . #<promise:unsaved-editor9165:3:10>)
(cdr s) =>?
(force (cdr s)) =>?
(define x (delay (display 'hh)))
x =>?
(force x) =>?
```

流的实现

- cons-stream 是否能实现为普通过程?

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s => hey'(1 . #<promise:unsaved-editor9165:3:10>)
(cdr s) => #<promise:unsaved-editor9165:3:10>
(force (cdr s)) => ?
(define x (delay (display 'hh)))
x =>?
(force x) =>?
```

流的实现

- cons-stream 是否能实现为普通过程?

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s => hey'(1 . #<promise:unsaved-editor9165:3:10>)
(cdr s) => #<promise:unsaved-editor9165:3:10>
(force (cdr s)) =>
(define x (delay (display 'hh)))
x =>?
(force x) =>?
```

流的实现

- cons-stream 是否能实现为普通过程?

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s => hey'(1 . #<promise:unsaved-editor9165:3:10>)
(cdr s) =>
(force (cdr s)) =>
(define x (delay (display 'hh)))
x => #<promise:x>
(force x) =>?
```

流的实现

- cons-stream 是否能实现为普通过程?

```
(define (cons-stream a b)
  (cons a (delay b)))

(define s (cons-stream 1 (display 'hey)))
s => hey'(1 . #<promise:unsaved-editor9165:3:10>)
(cdr s) =>
(force (cdr s)) =>
(define x (delay (display 'hh)))
x => #<promise:x>
(force x) =>hh
(force x) (无输出)
```

流的实现

- cons-stream 应实现为特殊形式！用“宏”实现

```
(define-syntax cons-stream
  (syntax-rules ()
    [(cons-stream x y) (cons x (delay y))]))
```

解释器要对宏进行预处理，进行形式上的等价替换

(cons-stream a b)　被预处理成：

```
(cons a (delay b))
```

宏

- 又一例子：

```
(define-syntax when
  (syntax-rules ()
    ((when pred exp exps ...)
      (if pred (begin exp exps ...)
          void
          ))))


(when (> 3 2) (display "ok") (newline) (display "bad"))
=>
ok
bad
```

特殊形式delay

- 防止delay重复求值

在确保 delay是特殊形式的前提下：

```
(define (delay exp);在什么情况下用这个delay不合适?
  (memo-proc (lambda ()
   exp)))

(define (memo-proc proc);记住proc是否执行过
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

特殊形式delay

- 防止delay重复求值

在确保 delay是特殊形式的前提下：

```
(define (delay exp);在需要exp有副作用,比如输出的情况下不合适!
  (memo-proc (lambda ()
   exp)))

(define (memo-proc proc);记住proc是否执行过
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

流的实现

- stream-car 和 stream-cdr

```
(define (stream-car stream) (car stream))

(define (stream-cdr stream) (force (cdr stream)))
```

重写求区间的第二个素数

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

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

即:

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

car为10000, cdr并没有被计算

重写求区间的第二个素数

```
(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)))))
```

求区间[10000,1000000]第二个素数:

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

重写求区间的第二个素数

```
(stream-filter prime?
        (stream-enumerate-interval 10000 1000000))) 执行的经过:
10000不是素数，所以:
(stream-cdr stream) ->
(stream-enumerate-interval 10001 1000000) ->
(cons-stream
    10001
   (stream-enumerate-interval 10002 1000000))) ->
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

```
(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)))))
```

重写求区间的第二个素数

直到碰上10007,是个素数,则返回:

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

即:

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

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

无穷流

- 包含所有正整数的序列integers:

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

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

integers可以用来构造其他无穷流

无穷流

●所有不被7整出的正整数序列：

```
(define (divisible? x y) (= (remainder x y) 0))
(define no-sevens ;所有不被7整除的正整数
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))

(stream-ref no-sevens 100) ;求第100个不被7整除的正整数

=>117
```

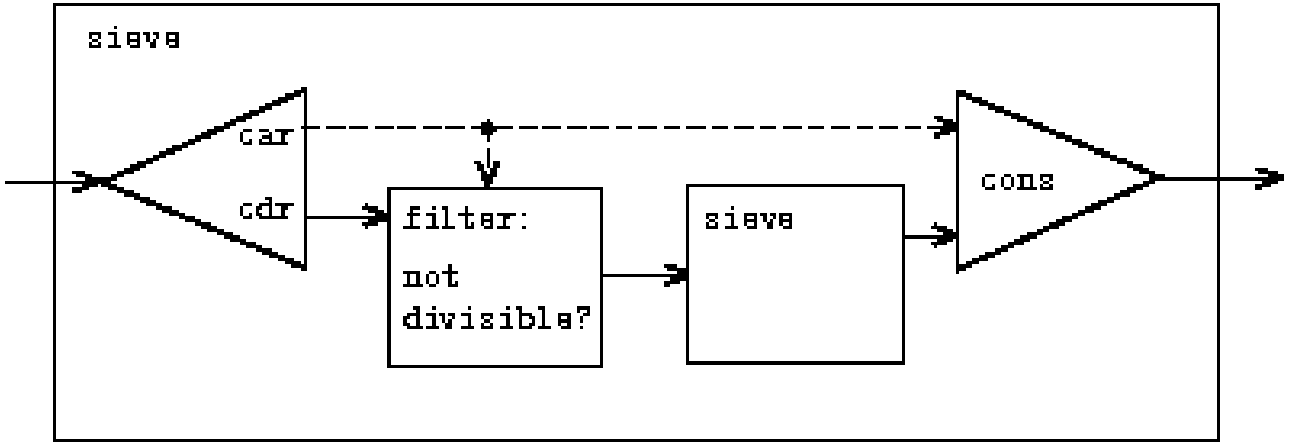
无穷流

●筛法构造所有素数的无穷序列：

```
(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)))

(stream-ref primes 50);求第50个素数
=>233
```

流形成的信号处理系统

●筛法构造所有素数的无穷序列：



虚线表示传输的是简单数据，实线表示传输的是流（序列）

流的**car 部分构造过滤器，而且作为结果流的car**

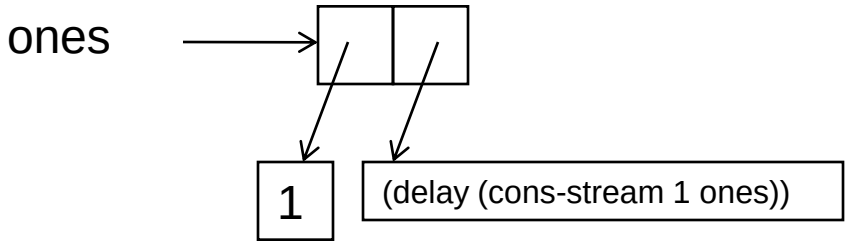
sieve 内嵌一个同样的信号处理系统sieve，所以，实际上这是一个无穷递归定义的系统

隐式定义流

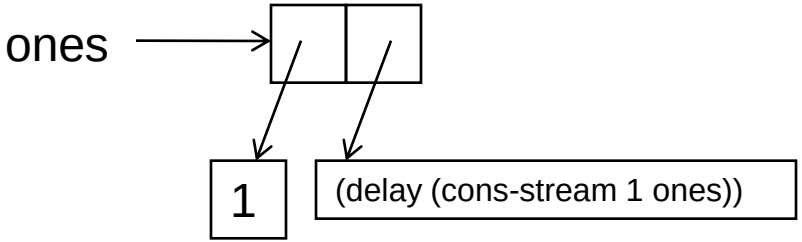
- 元素为1的无穷流：

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

ones是一个序对，car是1，cdr是一个求值ones的承诺。对cdr的求值则又给了我们一个1和求值ones的一个承诺.....

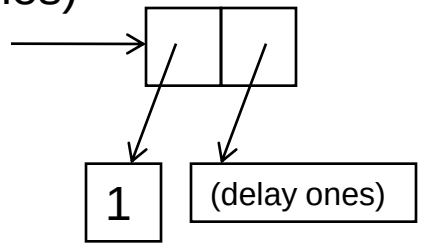


隐式定义流

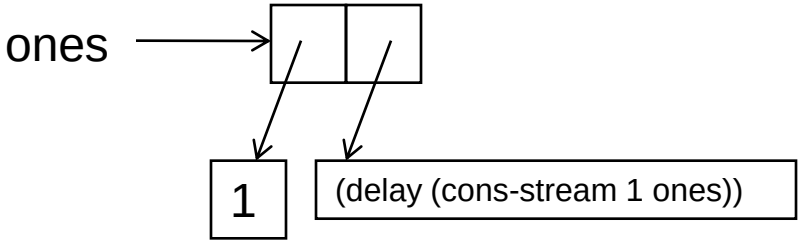


(stream-cdr ones) => (force (delay (cons-stream 1 ones)) =>

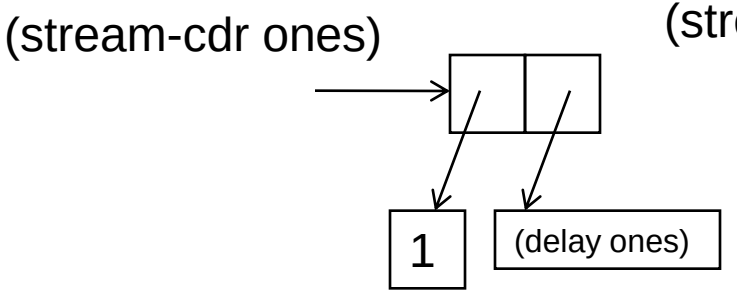
(stream-cdr ones)

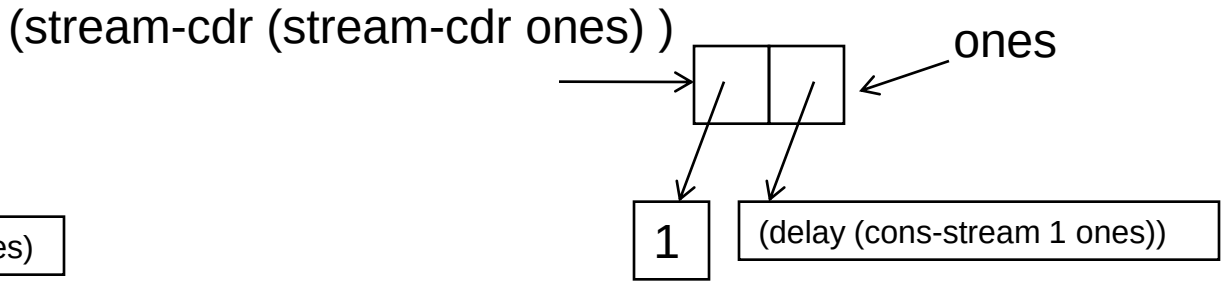


隐式定义流



(stream-cdr ones) => (force (delay (cons-stream 1 ones)) =>





隐式定义流

- 流的加法：

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

生成一个新流，元素为s1,s2对应元素之和

隐式定义流

- 流的加法：

```
(define (stream-map proc . argstreams) ;argstreams是若干个参数，每个参数都是一个stream ;stream-map 选出每个流的第一个元素，以他们作为参数调用proc形成一个结果，再以每个流的第二个元素作为参数，调用proc形成一个结果...然后把所有结果弄成一个流。
  (if (stream-null? argstreams)
    the-empty-stream
    (cons-stream (apply proc (map stream-car argstreams))
                 (apply stream-map
                        (cons proc (map stream-cdr
                                        argstreams))))))
```

隐式定义流

- 用流加法实现整数流：

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

在任意时刻，总有足够数量的integers的元素被生成，可以将其反馈到integters的定义，以便生成下一个元素。

隐式定义流

- 缩放流中的数值：

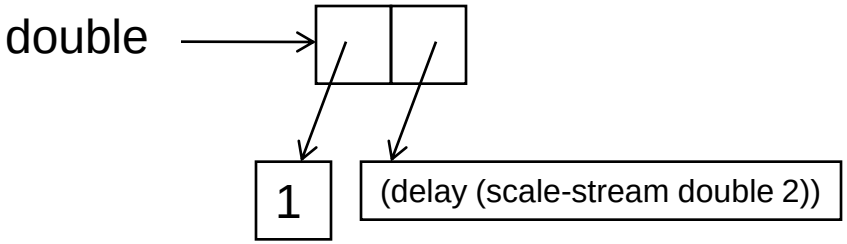
```
(define proc (lambda (x) (* x factor)))

(define (scale-stream stream factor)
        (stream-map proc stream))


(define double (cons-stream 1 (scale-stream double 2)))
```

生成序列：1, 2, 4, 8, 16, 32, ...

隐式定义流

- 求 `(stream-car (stream-cdr double))`:

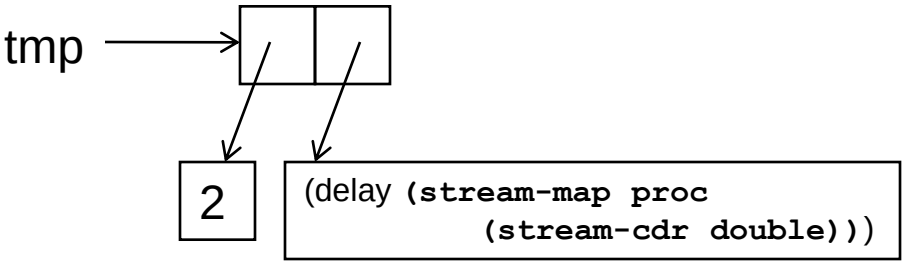
开始:




```
(define proc (lambda (x) (* x factor)))
(define (scale-stream stream factor)
         (stream-map proc stream))
(define double (cons-stream 1
                  (scale-stream double 2)))
```

隐式定义流

- 求 (stream-cdr double)，即：
  (force (delay (scale-stream double 2))),记结果为tmp



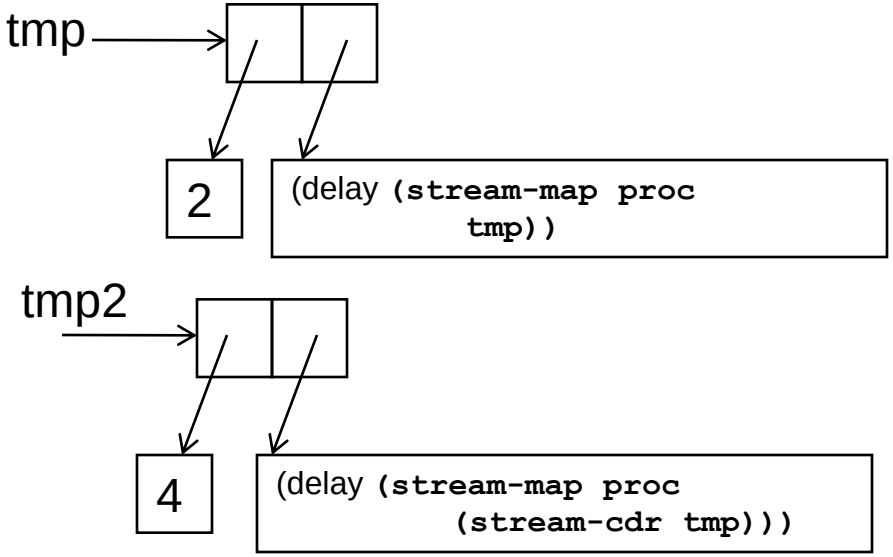

```
(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc
                               (stream-cdr s)))))
```

```
(define proc (lambda (x) (* x factor)))
(define (scale-stream stream factor)
         (stream-map proc stream))
(define double (cons-stream 1
                  (scale-stream double 2)))
```

(car tmp) = 2

隐式定义流

●求 (cdr tmp)，即：
(force (delay (stream-map proc (stream-cdr double)))),
即 (force (delay (stream-map proc tmp))),记结果为tmp2




```
(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 proc (lambda (x) (* x factor)))
(define (scale-stream stream factor)
         (stream-map proc stream))
(define double (cons-stream 1
                  (scale-stream double 2)))
```

隐式定义流

- 斐波那契数列：

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

```
    0 1 1 2 3  5  8 13 ... = fibs
    1 1 2 3 5  8 13 21 ... = (stream-cdr fibs)
0 1 1 2 3 5 8 13 21 34 ... = fibs
```

隐式定义流

- 素数流另一定义：

prime? 用素数流去判断一个数是否素数
primes 用**primes** 做流的过滤，删除不是素数的元素
primes 和**prime?** 相互递归引用，其结构和计算都很复杂

```
(define primes
  (cons-stream
   2
   (stream-filter prime? (integers-starting-from 3))))

(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

迭代表示为流过程

- 求平方根

```
(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)

(display-stream (sqrt-stream 2))
```

```
1.
1.5
1.4166666666666665
1.4142156862745097
1.414213562373899
...
1.414213562373095
1.414213562373095
1.414213562373095
......
```

迭代表示为流过程

- 求 PI的值

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

```
4.
2.666666666666667
3.466666666666667
2.895238095238095 6
3.339682539682540 3
2.976046176046176 5
3.283738483738484 4
3.017071817071818
...
```

```
(define (pi-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (pi-summands (+ n 2)))))
(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream)
```

迭代表示为流过程

- partial-sums

```
(define (partial-sums s)
;以s为参数，返回的流是 s0,s0+s1,s0+s1+s2, ....
  (define tmps
    (cons-stream (stream-car s) (add-stream tmps
                                            (stream-cdr s))))
  tmps)
```

流的加速收敛

- 加速求PI

欧拉提出的加速技术，适用于交错级数。对于项为Sn的级数(Sn = a1 + a2 + ...+ an)，加速级数的通项为：

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

新级数和原级数收敛到相同的值，但是快得多

流的加速收敛

- 加速求PI

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0))           ; Sn-1
        (s1 (stream-ref s 1))           ; Sn
        (s2 (stream-ref s 2)))          ; Sn+1
    (cons-stream (- s2 (/ (square (- s2 s1))
                          (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

```
3.166666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

```
(display-stream (euler-transform pi-stream))
```

加速流的再加速

- 构造一个流的流，其中每个流都是前一个流的变换的结果

```
(define (make-tableau transform s)
  (cons-stream s
               (make-tableau transform
                             (transform s))))
```

$$
\begin{array}{ccccccc}
s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \cdots \\
 & s_{10} & s_{11} & s_{12} & s_{13} & \cdots \\
 & & s_{20} & s_{21} & s_{22} & \cdots \\
 & & & \cdots & &
\end{array}
$$

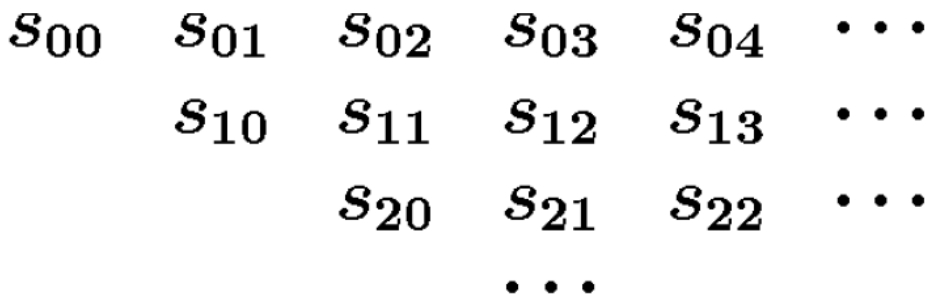
取出表列中每个序列的第一项，就得到了所需的序列：

```
(define (accelerated-sequence transform s)
        (stream-map stream-car (make-tableau transform s)))
```

加速流的再加速

- 极速求PI：

```
(display-stream (accelerated-sequence euler-transform
                                      pi-stream))
```



4.
3.166666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...

```
(display-stream (accelerated-sequence euler-transform
                                      pi-stream))
```

计算**8** 项就得到**14** 位有效数字。原序列需要计算10^{13} 项才能得到同精度的近似值

序对的无穷流

- 生成所有整数序对(i,j)，i<=j且i+j是素数。

若有一个 int-pairs,是所有满足i<=j的整数序对(i,j)的序列，则问题变为:

```
(stream-filter (lambda (pair)
                 (prime? (+ (car pair) (cadr pair))))
               int-pairs)
```

序对的无穷流

考虑**int-pairs** 的生成。一般说，假定有流**S = {S_i}** 和**T = {T_j}**，从它们可以得到无穷阵列，以及其对角线之上的序列集合：

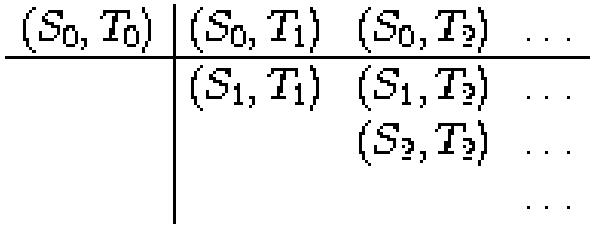
$$
\begin{array}{llll}
(S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\
(S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \ldots \\
(S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \ldots \\
\ldots & & &
\end{array}
$$

$$
\begin{array}{llll}
(S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\
 & (S_1, T_1) & (S_1, T_2) & \ldots \\
 & & (S_2, T_2) & \ldots \\
 & & & \ldots
\end{array}
$$

如果**S 和T 是整数的流，对角线序对集合就是所需的int-pairs**

序对的无穷流

- 将这个无穷流称为**(pairs S T)**，它由三个部分构成：

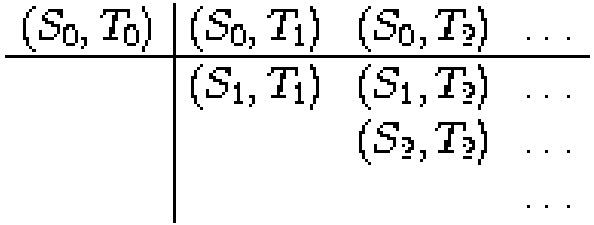


- 第**3**部分是由**(stream-cdr S)** 和**(stream-cdr T)** 递归构造的序对
- 第**2**部分也容易构造：

```
(stream-map (lambda (x) (list (stream-car s) x)) (stream-cdr t))
```

序对的无穷流

●将这个无穷流称为**(pairs S T)**，它由三个部分构成：



●所需的序对流：

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (<combine-in-some-way>
       (stream-map (lambda (x) (list (stream-car s) x))
                   (stream-cdr t))
       (pairs (stream-cdr s) (stream-cdr t)))))
```

还需要将两个无穷流组合起来

序对的无穷流

●模仿表的组合操作**append**：

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

是否可行？

序对的无穷流

●模仿表的组合操作**append**：

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

不行！因为第一个流无穷长，第二个流的元素永远不出现

序对的无穷流

●交错组合：

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

即使第一个流无穷长，第二个流的元素也有同等机会出现

序对的无穷流

- 最终结果：

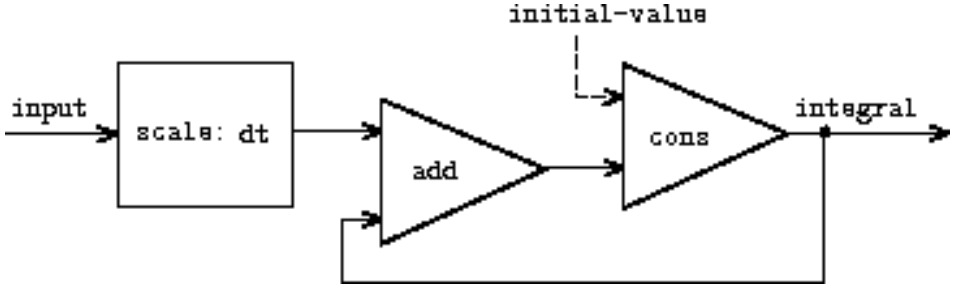
```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

流作为信号

- 可以用流表示一个信号在一系列时间点的值：

- 积分器（或称求和器）。对输入流$x = (x_i)$，初始值C 和一个小增量dt，它累积和S_i 并返回$S = (S_i)$：

$$S_i = C + \sum_{j=1}^{i} x_j dt$$



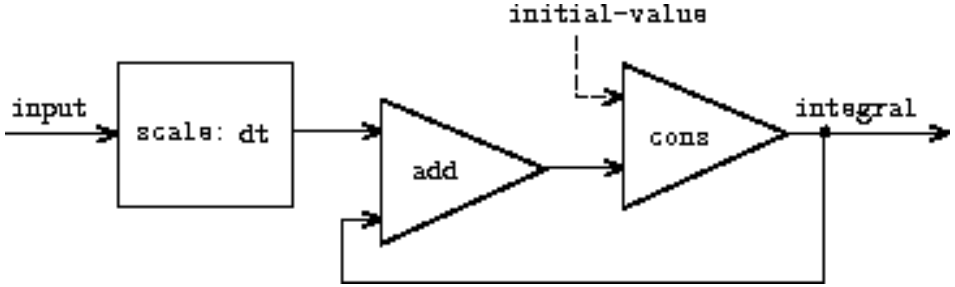

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int)  ;Si = Si-1 + xi * dt   S0 = C
```

流作为信号

- 可以用流表示一个信号在一系列时间点的值：

- 积分器（或称求和器）。对输入流$x = (x_i)$，初始值C 和一个小增量dt，它累积和S_i 并返回$S = (S_i)$：

$$S_i = C + \sum_{j=1}^{i} x_j dt$$




```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int);能成功是因为cons-stream里有 delay,
```

没有delay 无法构造带有反馈循环的系统（构造int 时用到它自身）

流的延时求值

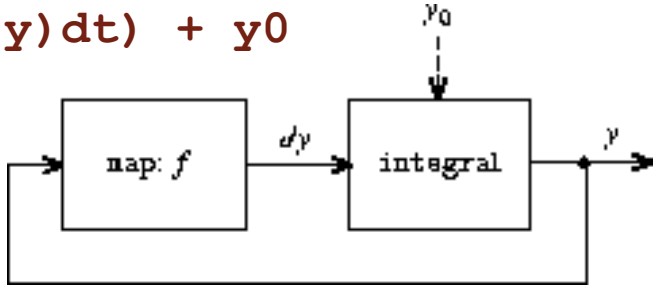
- 有时需要显式使用**delay**

- 假设要定义一个求解微分方程**dy/dt = f(y)** 的信号处理系统， **f** 是给定的函数。如图：

$$y = \int f(y)dt + y_0$$

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int)
```

```
(define (solve f y0 dt) ;y=integral(f(y)dt) + y0
 (define y (integral dy y0 dt))
 (define dy (stream-map f y))
 y) ;不可行！因为要求dy已经定义！
```



流的延时求值

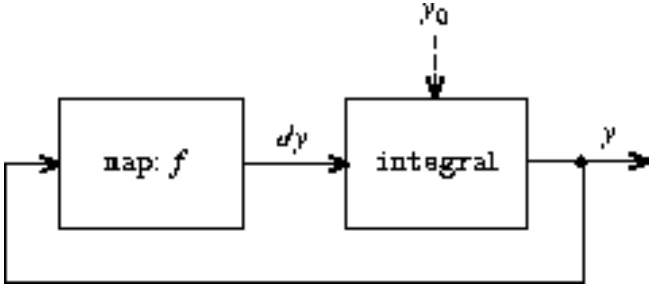
```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000);dt=0.0001
```

- 重新定义integral

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (let ((integrand (force delayed-integrand)))
                   (add-streams (scale-stream integrand dt)
                                int))))
  int) ;为何要force?
```

- 重新定义solve

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```



;integral在不知道 (delay dy)的情况下就能生成y的第一个元素initial-value。有了y的第一个元素，就能求出dy的第一个元素。

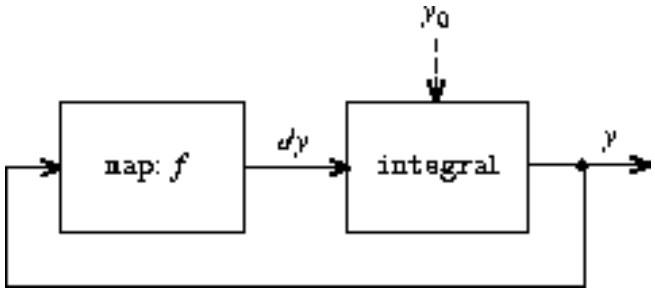
流的延时求值

调用这个**integral 时需要delay 被积参数**

例：求***dy/dt = y ，初始条件为y(0) = 1，在t = 1 的值：***

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000);dt=0.0001
```

=>2.716924



流用于模块化

- 引进赋值得到了新的模块化手段
  - 可以把系统状态的一些部分封装起来，隐藏到局部变量里

- 流模型可以提供类似的模块化，而且不需要赋值
  - 重新考虑前面蒙特卡罗模拟的例子
  - 两个随机数互素的概率是$6/\pi^2$
- 最关键的模块化需要是隐藏随机数生成器内部状态，达到内部状态与使用随机数的程序隔离
- 对基于赋值的技术，模块化的技术基础是利用过程rand-update 实现一个随机数生成器，其中封装一个局部状态：

```
(define rand
   (let ((x random-init))
         (lambda () (set! x (rand-update x)) x)))
```

rand-update 根据当前随机数生成下一个随机数

流用于模块化

- 流技术可以实现类似的模块化，描述中只看到一个随机数流：

```
(define random-numbers
        (cons-stream random-init
                     (stream-map rand-update random-numbers)))
```

这个流可以用作随机数生成器

- 基于random-numbers 做蒙特卡罗模拟的数对序列（流）：

```
(define cesaro-stream
  (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1))
                        random-numbers))
(define (map-successive-pairs f s)
  (cons-stream
   (f (stream-car s) (stream-car (stream-cdr s)))
   (map-successive-pairs f (stream-cdr (stream-cdr s)))))
```

cecaro-stream 是个布尔值流，流中真假值表示实验的成功与失败

流用于模块化

●试验时只需把**cecaro-stream 送给过程monte-carlo，它生成一个流S，**S_i就是考查到第i对随机数时，所有随机数对中，互素的比例

```
(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
     (/ passed (+ passed failed))
     (monte-carlo
      (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1))))

(define pi
  (stream-map (lambda (p) (sqrt (/ 6 p)))
              (monte-carlo cesaro-stream 0 0)))
```

可以支持任何蒙特卡罗试验换试验
只需要定义好相应的流
这个程序里没有状态也没有赋值