3。模块化，对象和状态(4)

本次课讨论：

- 流式计算
  - 流与序列
  - 延时求值
  - 无穷流
  - 流的应用
- 时间的函数式观点
- 不同系统模拟方法（函数式、对象、流）的比较

时间和流

- 在用状态和赋值模拟的工作中，我们看到了赋值带来的复杂性。本节考虑能否用其他方式模拟状态变化，避免其中一些复杂性问题
- 复杂性的来源是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真实中的状态可能变化的对象
  - 用计算机李的系统随着时间的变化模拟现实世界中的变化
  - 通过对有局部状态的对象的赋值，实现计算机系统里的状态变化
- 现在考虑另一可能性：把一个随时间变化的量表示为一个随时间变化的函数 x(t)。这样可得到两个层面上的观察：
  - 如果看具体时刻 x 的值，它是一个随时间变化的变量
  - 如果看 x 的整个历史，这里没有变化。作为函数的 x 不变
- 下面考虑离散时间上的函数，可以用无穷长的序列模拟它们。这种序列称为流，用于模拟状态变化，模拟一个系统随时间变化的历史

流、序列和表

- 为做这种模拟，需要引进一种（也）称为流（stream）的数据结构
  - 不能直接用表结构表示流，因为一般说流可能是无穷的结构
  - 下面采用延时求值技术，用流表示任意长的序列（潜无穷长）
- 流技术可用于模拟一些包含状态的系统，构造一些有趣模型。但这里不需要赋值和变动数据结构，因此可以避免了赋值带来的问题
- 流不是万能灵药，使用上有本质性困难。下面也讨论这方面问题

序列和表

- 前面讨论过用序列作为组合程序的标准接口，构造了许多有用的序列操作抽象，如 map、filter、accumulate。这些抽象用起来很漂亮
- 但是用表表示序列，得到好结果的同时可能付出严重的效率代价（空间和/或时间），因为每步操作中都可能构造出很大的数据结构

表处理的效率问题

- 用迭代风格计算一个区间中所有素数之和的过程：

```
(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))
```

- 用序列操作的组合写同样程序：

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

- 第一个程序计算中只维持部分积累和，而第二个程序里：
  - enumerate-interval 构造出区间 [a,b] 中所有整数的表
  - filter 基于产生过滤后的表并将其送给 accumulate
  - 两个表都可能很大（由区间长度确定）

表和流

- 清晰性和模块化的代价是效率和资源消耗
- 求 10000 到 1000000 区间的第二个素数（极端低效）：

  (car (cdr (filter prime? (enumerate-interval 10000 1000000))))
- 流是一种有趣想法，支持序列操作同时又能避免用表表示序列的额外代价：程序像操作表一样工作，又有递增计算的高效率。基本想法：
  - 做好一种安排，工作中只构造出序列的一部分。仅当程序需要访问序列的尚未构造出来的部分时才去构造它
  - 程序可以认为整个序列都存在，就像是处理和使用完整的序列
- 下面将设计一种实现，使序列构造和使用之间的交替完全透明化
- 从表面看流就像是表。构造函数 cons-stream，选择函数 stream-car 和 stream-cdr。the-empty-stream 是空流，它不是 cons-stream 的结果，可以用 stream-null? 判断。这些操作满足：

  (stream-car (cons-stream *x y*)) = *x*

  (stream-cdr (cons-stream *x y*)) = *y*

流和基于流的序列操作

- 基于它们可以定义各种序列操作（与基于表的操作类似）：

```
(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-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

- 检查流的内容的输出函数：

```
(define (display-stream s) (stream-for-each display-line s))
(define (display-line x) (newline) (display x))
```

- 为使流的构造和使用能自动而透明地交替进行，流的实现中需要做适当安排，使对其 cdr 部分的求值等到实际做 stream-cdr 时再做，而不是在求值 cons-stream 时做
- 讨论有理数时提出化简可以在构造时做或在取分子和分母时做。两种方式产生的数据抽象等价，但可能影响效率。这里的情况与之类似
- 作为数据抽象，流和常规表一样，不同点只是元素的求值时间。表的两个成分都在构造时求值，而流的 cdr 部分推迟到选取时求值
- 流的实现基于特殊形式 delay：求值 (delay <*e*>) 时并不求值 <*e*>，而是返回一个延时对象，它允诺可以由它得到 <*e*> 的值
- 与 delay 配套的过程 force 以延时对象为参数，执行对它的求值

流的实现

- cons-stream 是特殊形式，(cons-stream <a> <b>) 等价于表达式 (cons <a> (delay <b>))，两个选择函数定义为：

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

- 为理解流计算，看前面提的一个实例。用流重写求第二个素数的例子：

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

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

对stream-enumerate-interval 的调用返回：

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

流的计算

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

- 对流的过滤过程：

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

- stream-filter 检查 car 后丢掉 10000 并迫使流求出序列的下一元素，这次 stream-enumerate-interval 返回

```
(cons 10001 (delay (stream-enumerate-interval 10002 1000000)))
```

- 这样丢掉一个一个数直到 stream-enumerate-interval 返回

```
(cons 10007 (delay (stream-enumerate-interval 10008 1000000)))
```

最外层表达式的 stream-cdr 丢掉这第一个素数后继续，得到第二个素数 10009 之后计算结束。整个序列只展开了前面几项

流的实现

- 延迟求值可看作
  - “按需计算”，或
  - “需要驱动的程序”

  其中流成员的计算只做到足够满足需要的那一步为止
- 这样就消解了计算中事件的实际发生顺序和过程表面结构之间的紧密关系，达到模块化和效率方面的双赢

delay 和 force 的实现

- 两个基本操作的实现很简单：

  (delay <exp>) 就是 (lambda () <exp>) 的语法包装

  force 简单调用由 delay 产生的无参过程

  (define (force delayed-object) (delayed-object))
- PLT Scheme 定义了另一套延迟求值功能，和这里不兼容

流和记忆器

- 实际计算中可能会多次强迫求值同一个延时对象。每次都去重复求值会浪费很多资源（包括空间和时间）
- 一个解决办法是改造延时对象，让它在第一次求值中记录求出的值，再次求值时直接给出记录的值。这样的对象称为带记忆的延时对象
- 带记忆的延时对象的实现：

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if already-run?
          result
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)))))
```

现在应该用 (memo-proc (lambda () <exp>)) 作为 (delay <exp>)，force 的定义不变

无穷的流（流的应用）

- 流给使用者造成一种假相，以序列的一部分（已访问部分）扮演完整序列。这一技术可用于表示很长的序列，甚至无穷长的序列
- 考虑包含所有整数的序列：

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

序列的 car 是 1，cdr 是生成从 2 开始的序列的允诺。由于任何程序只可能用到有限个整数，它不会发现这里实际上没有无穷序列

- 可以基于 integer 定义其他许多无穷流

无穷的流

- 所有不能被 7 整除的整数流：

```
(define (divisible? x y) (= (remainder x y) 0))
(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7))) integers))

(stream-ref no-sevens 100)
```

117

- 所有斐波纳契数的流：

```
(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

其 car 是 0，其 cdr 是求值 (fibgen 1 1) 的允诺。对 (fibgen 1 1) 求值得到其 car 为 1，其 cdr 是求值 (fibgen 1 2) 的允诺，等等

无穷流

- 用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-filter 从去掉 2（当时的 car）的流中筛掉 2 整除的数，把生成的流送给 sieve；sieve 调用 stream-filter 从去掉 3（当时 car）的流中筛掉所有 3 整除的数，把剩下的流送给 sieve；... ...。

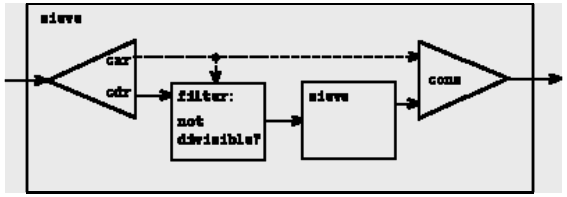
- 找第50个素数，只需写：

```
(stream-ref primes 50)
```

233

流形成的信号处理系统

- 流可看作信号处理系统。下图是 sieve 形成的信号处理系统图示：



- 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 部分用于构造过滤器，且作为结果的流的 car
  - 信号处理系统 sieve 内嵌一个同样的信号处理系统 sieve，所以这实际上是一个无穷递归定义的系统

前面技术中的做法是显式描述如何生成流中各个元素

另一种技术是利用延时求值隐式地描述流的构造。下面是一些例子

隐式地定义流

- 元素均为 1 的无穷流：

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

- 通过流运算，可以方便地构造出各种流。如加法运算和整数流：

```
(define (add-streams s1 s2) (stream-map + s1 s2))
(define integers (cons-stream 1 (add-streams ones integers)))
```

看看这个 integer 如何顺序地生成一个个整数

- 可以定义更多流运算，如缩放流中的各个数值：

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

生成序列：1, 2, 4, 8, 16, 32, ...

隐式地定义流

- 用这种风格生成斐波纳契数的流：

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

从斐波纳契序列的前两个数出发求序列 fibs 和 (stream-cdr fibs) 的逐项和。构造出的部分序列再用于随后的构造

```
   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
```


隐式地定义流

- 素数流的另一定义：

```
(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))
```

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

流计算的应用

- 基于延时求值的流也是很强大的模拟工具
  - 可以在许多问题上代替局部状态和赋值
  - 与此同时避免引入状态和赋值带来的麻烦
- 在模拟实际的系统时，流支持的模块划分方式和基于赋值和局部状态的方式不同。现在可以把整个时间序列或信号序列（而不是各时刻的值）作为关注的目标，这样做更容易组合来自不同时刻的状态成分
- 下面通过一些实例研究这种模拟

  应该更多关注其中的一般性想法和一般性技术，而不是具体问题

把迭代表示为流过程

- 迭代就是不断更新一些状态变量。可以考虑把这样的过程表示为流
- 前面的求平方根过程中生成一系列逐步改善的猜测值：

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

换种方式，可以生成这种猜测值的无穷序列：

```
(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
1.4142135623746899
...
```

生成更多的项，可以得到更好的猜测。可基于 sqrt-stream 定义一个过程，让它不断生成猜测值，直到得到足够好的答案为止

把迭代表示为流过程

- 考虑研究过的生成 π 的近似值的过程，基于交错级数

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

- 生成级数的几项和的流（partial-sums 求流的前缀段之和的流）：

```
(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)
4.
2.666666666666667
3.466666666666667
2.8952380952380956
3.3396825396825403
2.9760461760461765
3.2837384837384844
3.017071817071818
...
```

- 这个流能收敛到 π，但收敛太慢
- 下面考虑一些有趣技术，专门针对流计算模式，不是模拟状态变量更新

流的加速收敛

- 考虑如何加速级数收敛。欧拉提出了一种加速技术，特别适合用于交错级数。对于项为 S_n 的级数，加速序列的项是：

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

- 对于流 s，下面过程给出其加速后的流：

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

```
(display-stream (euler-transform pi-stream))
3.166666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

- 还可以考虑加速这样得到的序列
- 或者递归地加速下去，得到一个流的流（下面称为表列，tableau），其中每个流是前一个流的加速结果

流的加速收敛

- 用一个个加速的方式生成流的流（表列）：

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

表列的形式：

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

- 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)))
```

- 加速试验，生成逼近 π 的序列：

```
(display-stream (accelerated-sequence euler-transform pi-stream))
4.
3.166666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...
```

- 计算 8 项就得到 14 位有效数字。原序列计算 10^{13} 项才能得到同精度的近似值
- 虽然不用流模型也能实现这种加速，但这里整个流可以像序列一样用，描述这种加速技术特别方便

序对的无穷流

- 前面讨论过如何用序列把常规程序里用嵌套循环处理的问题表示为序列操作。该技术可推广到无穷流，写出一些不容易用循环表示的程序（直接做时，需要对无穷集合做循环）
- 推广 2.2.3 节的 prime-sum-pair，生成所有满足条件的整数序对 (i,j)，其中 i<=j 且 i+j 是素数

  若 int-pairs 是所有满足 i<=j 的序对 (i,j) 的序列，立刻可得

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

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

$$\begin{matrix} (S_0,T_0) & (S_0,T_1) & (S_0,T_2) & \cdots \\ (S_1,T_0) & (S_1,T_1) & (S_1,T_2) & \cdots \\ (S_2,T_0) & (S_2,T_1) & (S_2,T_2) & \cdots \\ \cdots & & & \end{matrix} \qquad \begin{matrix} (S_0,T_0) & (S_0,T_1) & (S_0,T_2) & \cdots \\ & (S_1,T_1) & (S_1,T_2) & \cdots \\ & & (S_2,T_2) & \cdots \\ & & & \cdots \end{matrix}$$

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

- 称这一无穷流为 (pairs S T)，它由三个部分构成：

$$\begin{array}{l|lll} (S_0,T_0) & (S_0,T_1) & (S_0,T_2) & \cdots \\ & (S_1,T_1) & (S_1,T_2) & \cdots \\ & & (S_2,T_2) & \cdots \\ & & & \cdots \end{array}$$

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

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

- 所需的序对流很简单：

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

剩下的问题是如何把两个无穷流组合起来

序对的无穷流

- 最简单的想法是模仿表的组合操作 append：

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

这样做不行！因为第一个流无穷长，第二个流的元素永远不出现

- 要考虑更巧妙的组合方法，如交错组合：

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

这样，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

- 最终得到生成所需的流的过程：

```
(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)))))

(define int-pairs (pairs integers integers))
```

- 交错是研究并发系统行为的一种重要工具
  - 这里用的交错是确定性的交错（一边一个）
  - 许多时候需要考虑非确定性的交错

流作为信号

- 前面讨论流时用信号处理作为背景。实际上可以直接用流为信号处理建模，其中用流的元素表示一个信号在顺序的一系列时间点上的值
- 考虑一个例子：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 $S = (S_i)$：

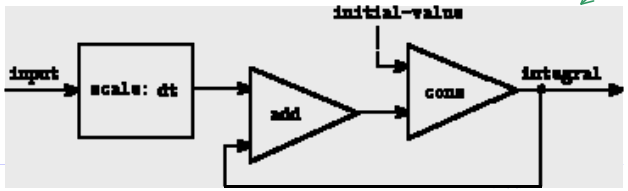
$$S_i = C + \sum_{j=1}^{i} x_j dt$$

输入流 integrand 经 dt 缩放送入加法器，加法器输出反馈回来送入同一个加法器，形成一个反馈循环

- 过程定义的形式类似于前面隐式定义的整数流：

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

图示：



流和延时求值

- 过程 integral 说明了如何用流模拟包含反馈循环的信号处理系统，其中加法器的反馈通过内部流 int 模拟：

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

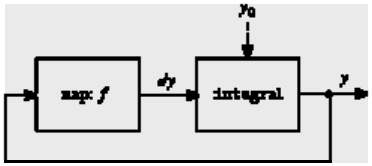
- 能处理这种定义，因为在 cons-stream 里有 delay。否则就需要用先构造出 cons-stream 的参数，而后用它去定义 int

  因此就无法构造带有反馈循环的系统（构造 int 时用到它自身）

- 一般而言，构造带有反馈循环的处理系统时，其中反馈流的定义都是递归的。必须有 delay，才能用没有完全构造好的流去定义它自身
- 对更复杂的情况，仅有隐藏在 cons-stream 里的 delay 可能不够，可能需要显式地明确使用 delay

流和延时求值

- 现在要定义一个求解微分方程 $dy/dx = f(x)$ 的信号处理系统，其中的 f 是给定的函数。如下图：



- 这里用一个部件实现应用 f 的映射，处理中存在一个反馈循环，循环中还包括一个积分器。模拟计算机用类似的电路求这种微分方程
- 如果用下面过程模拟这个信号处理系统：

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

这个过程将无法工作：定义 y 用到 dy，而当时 dy 还没有定义

流和延时求值

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)      ; 前面的代码
```

- 有时需要在还不知道 dy 的情况下开始生成 y

  integral 要在知道流的部分信息（第一个元素）的情况下开始使用

- 为处理这种情况，必须保证要使用的流元素能及时生成出来
  - 对 integral，流 int 的第一个元素由 initial-value 给出
  - 对于现在要定义的流，流 y 的第一个元素来自参数 y0，并不需要知道 dy 就能得到这个元素
  - 有了 y 的第一个元素就可以开始构造 dy 了，而后就可以用 dy 的元素去构造流 y 的元素
- 要实现这种想法，需要修改 integral，让它把被积分流看作延时参数

流和延时求值

- integral 里需要用 force 去强迫对积分对象的求值：

```
(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)
```

- 现在在 y 里把 dy 定义为延时求值的参数，就可以实现所需过程了：

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

- 调用这个 integral 的过程时需要 delay 被积参数（使用变麻烦了）
- 求 $dy/dx = y$ 在 $y = 1$，初始条件为 $y(0) = 1$ 的值：

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000)
2.716924
```

求值的正则序

- 显式使用 delay 和 force 扩大了应用范围，也使工作变复杂。新的 integral 能模拟更多信号处理系统，但要求用延时流作为被积对象
- 解决同一个问题定义了两个 integral：一个用常规参数，一个用延时参数，可以适应更多问题。处理其他问题时也会遇到类似情况
- 要避免写两个过程，一个办法是让所有参数都是延时的。例如换一种求值模型，其中所有的参数都不求值。也就是采用正则序求值
- 采用正则序可以得到统一的参数使用方式。很适合流处理的需要

  下章将讨论如何构造一个统一使用正则序的语言
- 采用正则序也有问题：它与局部状态的变动不协调
  - 应用序要求参数的求值总在过程体执行前完成，求值时间很清楚
  - 正则序将参数求值延迟到使用时，延时期间发生的赋值可能改变相关对象的状态，影响参数的值。这可能使程序语义变得不清晰

函数式程序和对象的模块化

- 前面说过，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方式，同时又不需要赋值
- 用蒙特卡罗模拟的例子说明。这里最关键的模块化是隐藏随机数生成器的内部状态，使之与使用程序隔离。技术上是基于过程 rand-update 实现随机数生成器：

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

- 采用流技术，描述中只看到一个随机数的流：

```
(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

函数式程序和对象的模块化

- 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 是一个布尔值流，流中的真假值记录了蒙特卡罗测试的成功与失败

函数式程序和对象的模块化

- 把 cecaro-stream 馈入过程 monte-carlo，这个过程生成顺序的一系列 π 估计值的流：

```
(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)))
```

模块结构良好，概念清晰，可以支持任何蒙特卡罗试验

与此同时，这个程序里没有状态，也没有赋值

时间的函数式观点

- 回到本章开始提出的对象和状态问题，换个角度去看它们
  - 赋值和变动对象是模块化手段，用它们模拟复杂系统的方式是：构造有局部状态的对象，用赋值改变状态，用这种对象在执行中的变化去模拟现实世界中各种对象的行为
  - 可以用流做类似模拟，模拟的是变化的量随时间的变化史。直接表现时间缓和了模型中的事件和被模拟世界的时间的联系。**delay** 使事件的发生顺序与被模拟世界里的时间脱了钩
- 下面比较这两种模拟。考虑取款处理器的例子：

```
(define (make-simplified-withdraw balance)
    (lambda (amount)
      (set! balance (- balance amount))
      balance))
```

生成的处理器实例有局部状态，每次调用减少 **balance** 值

一系列调用就是送给它一个提款额序列，解释器显示一个余额序列

时间的函数式观点

- 用一个流来模拟，它以初始余额和提款流作为参数，生成余额流：

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
     balance
     (stream-withdraw (- balance (stream-car amount-stream))
                      (stream-cdr amount-stream))))
```

这个过程是一个良好定义的数学函数，其输出完全由输入确定

- 从输入提款额并观看余额的角度，这个流的行为就像前面的对象

  但这里没有内部状态和变动，因此没有前面提到的各种麻烦
- 这里好像有悖论： **stream-withdraw** 是一个数学函数，其行为不变，而用户却像是与一个状态不断变化的系统打交道。怎么理解？
- 解释：是用户行为的时态特性赋予系统时态性质。如果用户只从交互流和余额流的角度（不看具体交互），就看不到系统的状态特征

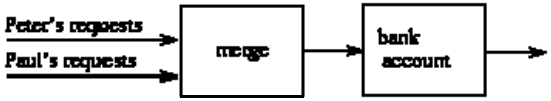
时间的函数式观点

- 一个复杂的系统有许多部分
  - 从一个部分，可以看到其余部分都在随时间变化，有变化的状态
  - 要写程序去模拟真实世界的这种分解，最自然的方式就是定义一些有状态的对象，让它们根据需要变化（对象和状态途径）
  - 这样做，就是用计算机执行的时间去模拟真实世界的时间，把真实世界的对象“塞进”计算机
- 有局部状态的对象很有用，也很直观，比较符合人对所处并与之交流的世界的看法（看成一些不断变化的对象）。但这种模型本质地依赖于事件发生的顺序，还带来并发同步等很麻烦的问题
- 纯函数式语言里没有赋值和变动对象，所有过程实现的都是定义良好的数学函数，其行为永远不变，因此特别适合描述并发系统
- 进一步观察，可以看到时间在函数式模型里的影子

  特别麻烦的是设计交互式程序，模拟独立对象之间的交互时

时间的函数式观点

- 例：考虑共享账户问题。在常见系统里 **Peter** 和 **Paul** 的共享账户是用状态和赋值模拟的，他们的请求被送到这个计算对象
- 按照流的观点没有对象，要模拟 **Peter** 和 **Paul** 的共享账户，就需要把两人的交易请求流合并后送给表示他们的共享账户的过程：



- 这里的归并有问题：怎么反映真实世界里两人的行为效果（两人碰头时应看到以前所有交易的效果）。不能用交替（两人交替操作，不合理）
- 要解决这个问题，就必须引入显式同步（前面想避免的事情）
- 本章提出了两种技术模拟不断变化的世界，：
  - 用一集相互分离、受时间约束、有内部状态且相互交流的对象
  - 用一个没有时间也没有状态的统一体（流）

  两种途径各有优势和劣势，但都不能完全令人满意