プログラミング言語(SICP)

3. Modularity, Objects, and State
3.5.3 ～ 3.5.5 streams
6月13日 8号館3階大講義室 中間試験

奥乃 博（3章）
大学院情報学研究科知能情報学専攻
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/12/ProgLang/
{okuno, igarashi}@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃1研，2研，ソ基分野
糸原 遼彦(M2) 奥乃研・音楽ロボットG
柳楽 浩平(M2) 奥乃研・ロボット聴覚G

NoStudent Left Behind

5月23日・本日のメニュー

6月13日(木) 中間テスト
範囲は第3章

3.5 Streams
3-5-1 Streams are Delayed Lists
3-5-2 Infinite Streams
3-5-3 Exploiting the Stream Paradigm
3-5-4 Streams and Delayed Evaluation
3-5-5 Modularity of Funcitional Programs and Modularity of Objects

NoStudent Left Behind

Perfect numbers, empty stream

NoStudent Left Behind

- 完全数 (perfect number) すべての約数の和が自分の倍

```
(define (perfect? n) (= n (accumulate + 0 (factors n))))
(define (factors n)
  (filter (lambda (factor) (divisible? n factor))
          (enumerate-interval 1 (/ n 2)) ))
(define perfect-numbers
  (stream-filter perfect? Integers) )
(stream-ref perfect-numbers 0)
```

6

- Infinite stream の問題点

```
(stream-filter odd? (stream-filter even? integers))
(stream-null?
   (stream-filter odd? (stream-filter even? integers) )
```

3.5.3 Exploiting the Stream Paradigm

ストリームパラダイムの追求

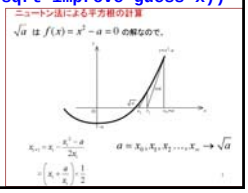
ストリームによる問題解決は，状態変数への代入の周囲に組織化されたシステムとは異なる部品化の境界を持つシステムが構築できるため，照り輝く．

- The stream approach can be illuminating because it allows us to build systems with different module boundaries than systems organized around assignment to state variables.
- **ストリームアプローチが燦然と輝けるのは，システム構築で状態変数への代入を中心に構成されたシステムとは違うモジュール境界を使えるからである．**

Formulating iterations as stream processes

- 1.2.1節：状態変数を更新して「繰返し型プロセス」を実現．
- 本節：Timeless stream of values instead of a set of variables to be updated

```
(define (integers-starting-from n)
   (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
(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
1.4142135623746899
...
```



円周率を求める

$$\frac{\pi}{4}=\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots$$

```
(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
...
```

Exercise 3.55（宿題）
Define a procedure partial-sums that takes as argument a stream S and returns the stream whose elements are S0, S0 + S1, S0 + S1 + S2, ...
For example, (partial-sums integers) should be the stream 1, 3, 6, 10, 15, ...

収束を加速化

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

NoStudent Left Behind

```
(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
...
```

euler-transform が呼ばれる時点での s であることに注意

Tableau: Euler 変換を更に加速化

NoStudent Left Behind

- 加速化を再帰的に加速化 Tableau: A stream of streams を利用

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

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

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

pi-stream s_{00} s_{01} s_{02} s_{03} s_{04} ...

Euler-transform s_{10} s_{11} s_{12} s_{13} ...

Euler-transform2回 s_{20} s_{21} s_{22} ...

...

Exercise 3.64（宿題）

NoStudent Left Behind

Write a procedure stream-limit that takes as arguments a stream and a number (the tolerance). It should examine the stream until it finds two successive elements that differ in absolute value by less than the tolerance, and return the second of the two elements. Using this, we could compute square roots up to a given tolerance by

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance) )
```

Exercise 3.65

Use the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for. How rapidly do these sequences converge?

Infinite streams of pairs

すべての整数のペア(2.2.3節)を無限ストリームに展開

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
               (flatmap
                (lambda (i)
                  (map (lambda (j) (list i j))
                       (enumerate-interval 1 (- i 1))))
                (enumerate-interval 1 n)))))

(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))) )
```

(S_0,T_0) (S_0,T_1) (S_0,T_2) ...
(S_1,T_0) (S_1,T_1) (S_1,T_2) ...
(S_2,T_0) (S_2,T_1) (S_2,T_2) ...

Int-pairs が全整数i, j (i ≦ j) に対する整数のすべての対とすれば,

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

(S_0,T_0) (S_0,T_1) (S_0,T_2)
(S_1,T_1) (S_1,T_2)
(S_2,T_2)

無限ストリームを3つの部分に分けて考えると,

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

(S_0,T_0) | (S_0,T_1) (S_0,T_2)
(S_1,T_1) (S_1,T_2)
(S_2,T_2)

Infinite streams of pairs

すべての整数の対の無限ストリームは3つの部分から構成

(S_0,T_0) (S_0,T_1) (S_0,T_2) ...
(S_1,T_1) (S_1,T_2) ...
(S_2,T_2) ...
...

(S_0,T_0) (S_0,T_1) (S_0,T_2) ...
(S_1,T_1) (S_1,T_2) ...
(S_2,T_2) ...
...

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

(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)))))
```

stream同士の結合

- 単純にappendで繋ぐと，無限ストリームには???

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

- interleaveで繋ぐと，

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

Infinite stream の結合

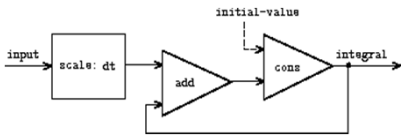
```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream
          (stream-car s1)
          (interleave s2 (stream-cdr s1)) )))

(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)) )))
```

Streams as signals

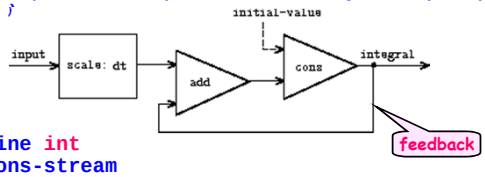
- アナログ計算機のアナロジー：積分器 $S_i = C + \sum_{j=1}^{i} x_j\,dt$
- Input stream: X =(Xj)， 初期値 C，増分 dt

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



3.5.4 Streams and Delayed Evaluation

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



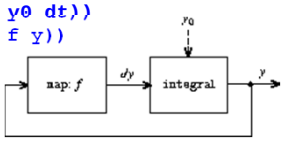

```
(define int
  (cons-stream
    initial-value
    (add-streams (scale-stream integrand dt)
                 int )))
```

add-stream 中の int は定義されていないと, loopは機能せず

⇒ delay は本質的

微分方程式 dy/dt = f(y) のアナログ解法

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

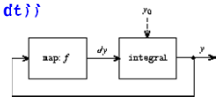


- このプログラムは動かない！
- 定義は正しい.
- 通常feedbackの1回目は未定義なので初期値を使用.
- ⇒ argument を delay (delayed argument)

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

Delayed argument の導入

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

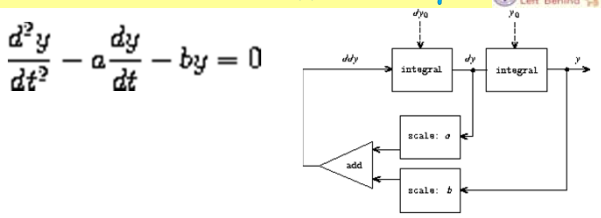


- dy/dt = y ⇒ y = C×et
- y(0) = 1

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

2.716924 (≒ e)

Ex.3.78 2nd order linear differential equati



The output stream, modeling y, is generated by a network that contains a loop. This is because the value of d^2y/dt^2 depends upon the values of y and dy/dt and both of these are determined by integrating d^2y/dt^2. The diagram we would like to encode is shown in figure 3.35. Write a procedure solve-2nd that takes as arguments the constants a, b, and dt and the initial values y_0 and dy_0 for y and dy/dt and generates the stream of successive values of y.

Normal-order evaluation

- delay, force を明示的に使えば, プログラミングの自由度大
- ループ有プログラムでは, 実行プロセスが複雑化
  - 通常の手続きと, delayed argumentsの手続きに分類
  - 2つのクラスの手続きを作成 ⇐ 綺麗な体系ではない

1. すべての手続が delayed arguments を取るものとする
   この意味は normal-order evaluation
2. Normal-order evaluation を使用すれば, delayed arguments は簡単化できる
3. ただし, これはstreamだけを使う場合.
4. 通常のプログラミングスタイル(assignment, mutable data, input/output)とは 相性が悪い. See Ex.3.51, 3.52.
5. Mutability と delayed evaluation をどう共存させるかはまだオープン問題

3.5.5 Modularity of Functional Programs and Modularity of Objects

- 局所状態変数の使用によりmodularity が高まる
- stream でも modularity が高まるはず.
- stream を使った Monte Carlo 法(πを求める)で検証

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

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

stream を使った Monte Carlo 法(π)

```
(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) ))
```

局所変数への代入はない.

A functional-programming view of ti

- 変数代入による銀行口座に引出し

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

- 関数型プログラミング(stream)による銀行口座引出し

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

- 共同口座での引出しはうまくいくか？

宿題: 5月30日午前8時 締切

1. Ex3.55, 3.64, 3.74.
2. プログラムの説明, 実行例をつけて, 設問に応えること.
3. レポート(PDF)とプログラムファイルを送付

   PROG-7@zeus.kuis.kyoto-u.ac.jp

   file 名は 学籍番号-名前-7.pdf

- 友達に教えてもらったら, その人の名前を明記すること. Webは出展を明記. (otherwise『同じ』回答は減点)