

工学部専門科目

「プログラミング言語」 SICP  
第3章 ~ Modularity, Objects, State ~  
その5

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科  
通信情報システム専攻

May 27, 2014

# 今日のメニュー

- 中間テストについて

## 3.5 ストリーム

### 3.5.1 ストリーム = 遅延リスト

### 3.5.2 無限ストリーム

## 3.5: ストリーム

- 3.1 ~ 3.3 (3.4 も) の内容: 代入 (assignment, set!, set-car!, set-cdr!) を使った「状態」のモデリング
  - ▶ 暗黙に決定したこと: モデル化される世界の時間の流れ = 計算機上の時間の流れ
- 3.5 の内容: 「状態」の別のモデリング手法
  - ▶ 状態 = 時刻上の関数
    - ★ c.f. 物理学での状態モデリング: 時刻  $t$  における物体の位置  $x$  を関数  $x(t)$  で表現
  - ▶ 時刻が離散的      時刻上の関数 = 値の (無限) 列
  - ▶ 列を表現するための (新たな) データ構造: ストリーム

## 3.5.1 遅延リストとしてのストリーム

- 2.2.3: リストを使ったデータ列の表現とその操作
  - ▶ `map`, `filter` などで複雑な処理を簡潔に表現!
  - ▶ 一方で, リストは非効率なこともある
- ストリーム: 必要になったら初めて要素を計算するような列
  - ▶ `cons-stream`
  - ▶ `stream-car`, `stream-cdr`

リストと何が違うの!?

# map, filter の非効率性 (1/2)

例題: a 以上 b 以下の素数の和を求める

解1: 素直な iterative 版

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
           ;; See 1.2.6 for def. of prime?
           (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

計算に要するのは基本的に count, accum の値だけ  
⇒ Good!

## map, filter の非効率性 (2/2)

解2: 汎用リスト操作関数を使った簡潔版

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

各リスト操作後に中間データ

- `enumerate-interval` の結果: `(a a+1 ... b)`
  - `filter` の結果: `(p1 p2 ... pn)`
- が発生!  $\implies$  余分な(?)メモリ消費

# 極端な例

10,000 以上で 2 番目の素数を求める

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

3 番目以降の素数の計算は全くの無駄!  $\implies$  やっぱり  
いちいち解 1 のように問題に特化した手続きをいち  
いち書くべき?

# 両方のええとこどり: ストリーム

- ストリーム = データの列 (リストと同様)
- = 部分的にデータが計算されたリスト
- = 最初の方の要素の集まり  
+ 未計算の「残りの要素」

「残りの要素」の計算は、必要になった時に自動的にやってくれる!



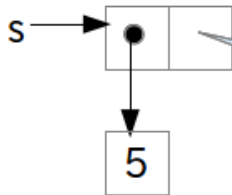
# ストリームに関する基本値と手続き

- 値: `the-empty-stream`
- 手続き: `stream-null?`
- 手続き: `stream-car`
- 手続き: `stream-cdr`
- 特殊形式: `cons-stream`

名前が違うだけでリストと同じ!?  $\implies$  ストリーム版  
`map`, `filter` などもリスト版と同じように定義できる

# リストとストリームの違い

- リスト: car 部, cdr 部の計算はペアを作る時に行う  
(define l (cons (+ 2 3) (cons (+ 4 5) '())))  
;; ふたつの足し算はこの時点で行われる
- ストリーム:  
cdr 部の計算は, ペアから**選択された時**に遅延される  
(define s (cons-stream (+ 2 3)  
 (cons-stream (+ 4 5)  
 the-empty-stream)))  
;; 5 と (+ 4 5) のリスト (!?)  
(stream-car (stream-cdr s))  
;; ここで黄色部分の計算が発生して 9 になる



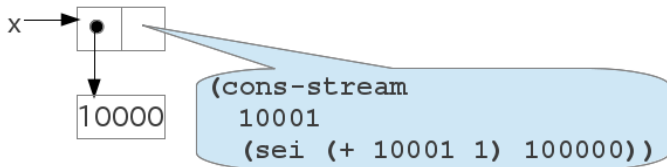
```
(cons-stream  
  (+ 4 5)  
  the empty-stream)
```

吹き出し部が stream-cdr が適用されるまで遅延される計算

# 10,000 以上で二番目の素数，再び

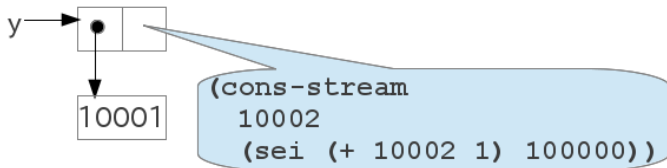
```
(stream-car ;; 以下 s-car と略
 (stream-cdr ;; 以下 s-cdr と略
  (stream-filter prime?
   (stream-enumerate-interval 10000 1000000))))
```

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



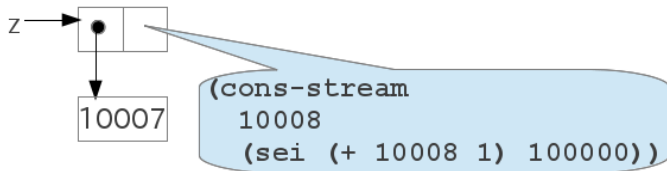
```
(s-car (s-cdr (stream-filter prime? x)))
```

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



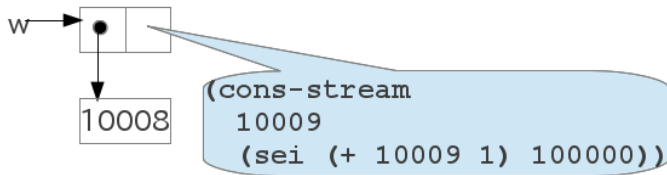
```
(s-car (s-cdr (stream-filter prime? y)))
```

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



```
(s-car (s-cdr (stream-filter prime? z)))
```

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



```
(s-car (s-cdr
  (cons-stream 10007 (stream-filter prime? w))))
```

⇓

```
(s-car (stream-filter prime? w))
```

⇓

```
(s-car (cons-stream 10009 ...))
```

⇓

10009 ; ; 余計な素数のリストを生成せず計算完了!



# ストリームの実装

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

- 新たな特殊形式 delay:

```
(delay b) = (lambda () b) ;; 引数のない関数
```

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

```
(define (stream-null? s) (null? s))
```

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

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

- (force delayed-obj) = (delayed-obj)

## jakld について注意

jakld には delay, force, cons-stream だけ用意されています。他は自分で定義してください。

## delay & force (メモ化つき)

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

を使って,

- `(delay x) = (memo-proc (lambda () x))`
- `force` はそのまま

## 3.5.2: 無限ストリーム

- ストリームを使うと「ながーい」列も無駄な計算をすることなく効率的に表現可能
- 無限に「ながーい...」くても OK!

### 1 から始まる整数の無限列

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

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

- `stream-enumerate-interval` と比較してみよう!

```
(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
```

# フィボナッチ数列

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

# エラトステネスのふるいによる素数列

```
(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)))
```

# 生成関数を明示的に定義しないストリーム定義

```
(define ones (cons-stream 1 ones)) ;; 1 の列
(define (add-streams s1 s2)
  (stream-map + s1 s2))    ;; See Ex. 3.50

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

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



```
(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))
```

# 宿題：6/3(火) 10:30 締切

- Ex. 3.51, 3.54, 3,58
- レポートには
  - ▶ 考え方の説明
  - ▶ プログラムリストと考え方の対応
  - ▶ 実行例を示すこと
- レポート (pdf) とプログラムファイルを提出システムを通じて提出
- 友達に教えてもらったなら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)

# 来週の予定範囲 3.5.3 ~ 3.5.5