プログラミング言語-4 2012年5月2日

プログラミング言語(SICP)
3. Modularity, Objects, and State
3.5 streams

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　達彦(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

1

5月16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

3.5 Streams

実世界での課題: 変化 (change) のモデル化

1. 実世界での時間変化を計算オブジェクトの局所状態の時間変化 (time variation) をモデル化
   ⇒ モデル化したオブジェクトの局所変数への代入 (assignment) で時間変化をとらえる
2. 代替案: stream を利用 ⇒ 一部の問題が軽減化
   瞬間での値変化ではなく，値の全履歴x(t) で考える
   離散時間変化とみなすと，x(t) はsequence となる．
   Stream はsequence だが単なる list ではない．
   Delayed evaluation technique と組み合わせると，stream は大規模な(無限長の)sequence が表現可能

3.5.1 Streams are delayed lists

2.2.3章：map, filter, accumulate, enumerate, …

```
(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))
```

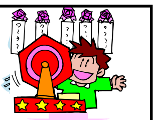
簡潔な記述だが無駄な処理：処理を順に適用，リストのコピーを繰り返す

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

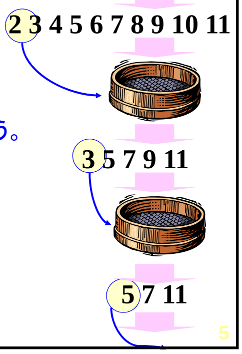
- (enumerate-interval 10000 1000000) **は完全なリストを作る**
- (car (cdr (filter prime?
  (enumerate-interval 10000 1000000))))

⇒ 10000以上の2番目の素数を返す，ただし，全ての素数を求めてから.

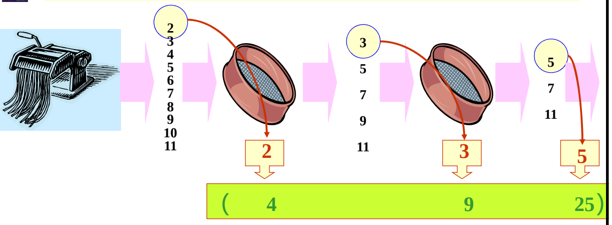
seq: 慣用インタフェース



- 処理間のインタフェース
- API（Application Program Interface）
- Parameterでの受け渡し
- データ構造をインタフェースに使う。
- sequence を活用
- 例： 素数を求めるための The Sieve of Eratosthenes（エラトステネスの篩）

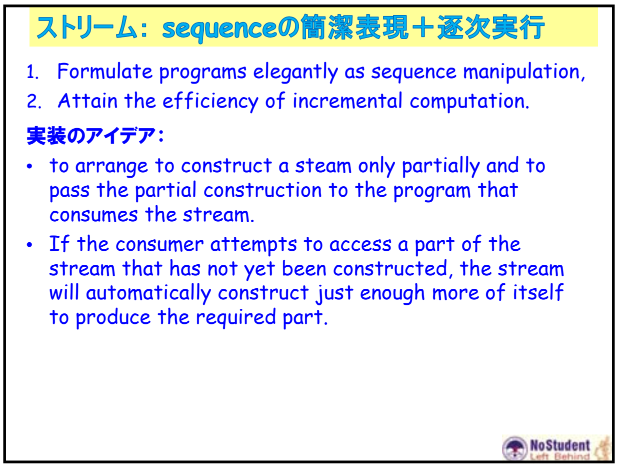


共通性の視点：素数の2乗を求める



共通点を見る4つの基本手続き

- 数え上げ（enumerate）
- フィルタ（filter）
- 写像（map）
- 集約（accumulate）

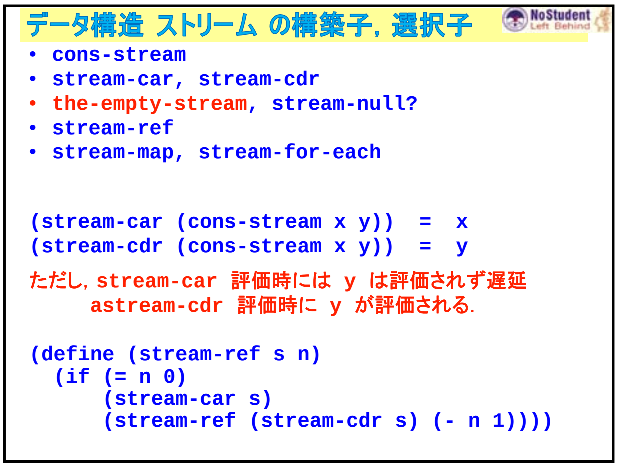


ストリーム：sequenceの簡潔表現＋逐次実行

1. Formulate programs elegantly as sequence manipulation,
2. Attain the efficiency of incremental computation.

実装のアイデア：

- to arrange to construct a steam only partially and to pass the partial construction to the program that consumes the stream.
- If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part.



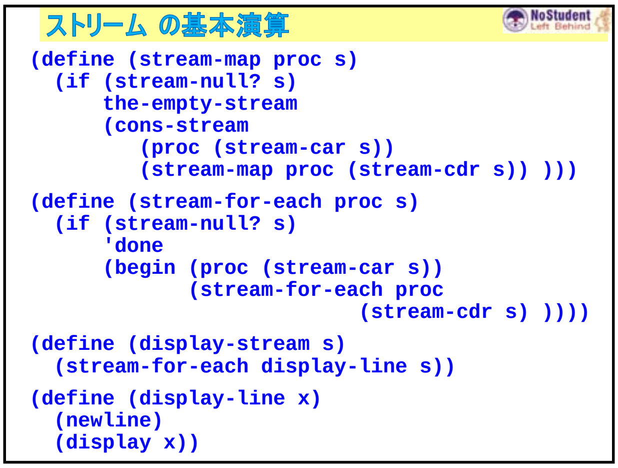
データ構造 ストリーム の構築子，選択子

- cons-stream
- stream-car, stream-cdr
- the-empty-stream, stream-null?
- stream-ref
- stream-map, stream-for-each

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

ただし，stream-car 評価時には y は評価されず遅延
　　astream-cdr 評価時に y が評価される.

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
```



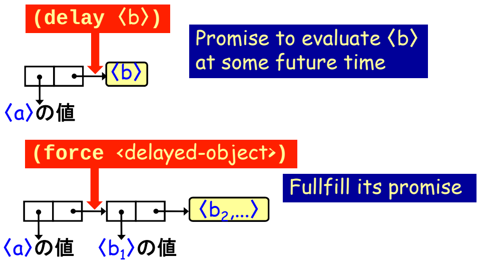
ストリーム の基本演算

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

delay ストリーム 実装上のspecial form

- (delay 〈exp〉): delayed object を生成して返.
- (force 〈obj〉): delayed object を評価して結果を返す(fulfill its promise).

(cons-stream 〈a〉 〈b〉) = (cons 〈a〉(delay 〈b〉))



(delay 〈b〉)

Promise to evaluate 〈b〉 at some future time

(force <delayed-object>)

Fullfill its promise

delay ストリーム 実装上のspecial form

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

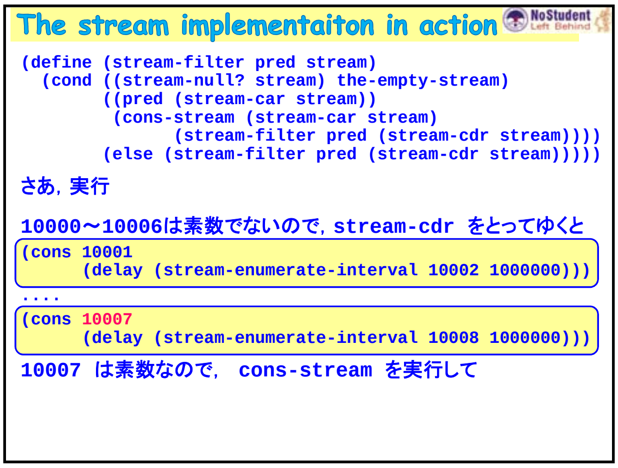
The stream implementaiton in action

2番目の素数の例の再考

```
(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) )))
```

実際の動きは

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



The stream implementaiton in action

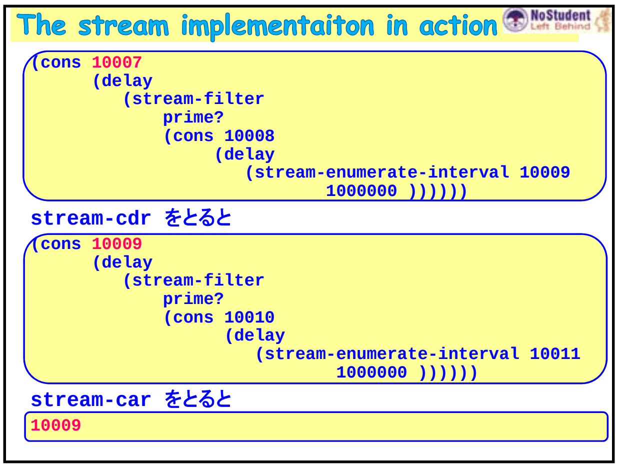
```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                (stream-filter pred (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

さあ，実行

10000～10006は素数でないので，stream-cdr をとってゆくと

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
....
(cons 10007
      (delay (stream-enumerate-interval 10008 1000000)))
```

10007 は素数なので， cons-stream を実行して



The stream implementaiton in action

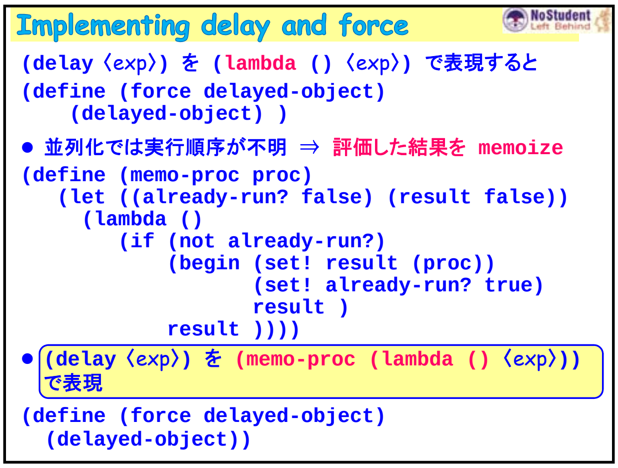
```
(cons 10007
      (delay
         (stream-filter
             prime?
             (cons 10008
                   (delay
                      (stream-enumerate-interval 10009
                              1000000 ))))))
```

stream-cdr をとると

```
(cons 10009
      (delay
         (stream-filter
             prime?
             (cons 10010
                    (delay
                       (stream-enumerate-interval 10011
                               1000000 ))))))
```

stream-car をとると

```
10009
```



Implementing delay and force

(delay ⟨exp⟩) を (lambda () ⟨exp⟩) で表現すると

```
(define (force delayed-object)
    (delayed-object) )
```

- 並列化では実行順序が不明 ⇒ 評価した結果を memoize

```
(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 ))))
```

- (delay ⟨exp⟩) を (memo-proc (lambda () ⟨exp⟩)) で表現

```
(define (force delayed-object)
  (delayed-object))
```

Exercise 3.50

Complete the following definition, which generalizes stream-map to allow procedures that take multiple arguments, analogous to map in section 2.2.3.

```
(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-map proc . argstreams)
  (if (⟨??⟩ (car argstreams))
      the-empty-stream
      (⟨??⟩
        (apply proc (map ⟨??⟩ argstreams))
        (apply stream-map
                   (cons proc (map ⟨??⟩ argstreams)) ))))
```

2引数を取る例を考えること

Exercise 3.51

In order to take a closer look at delayed evaluation, we will use the following procedure, which simply returns its argument after printing it:

```
(define (show x)
   (display-line x)
   x)
```

What does the interpreter print in response to evaluating each expression in the following sequence?

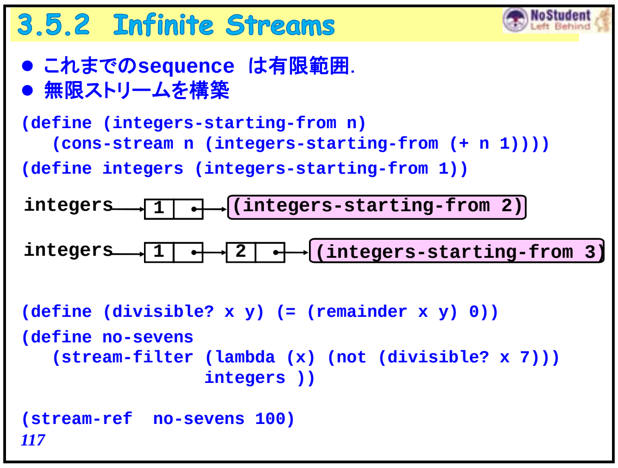
```
(define x (stream-map show
                (stream-enumerate-interval 0 10)))
(stream-ref x 5)
(stream-ref x 7)
```

Exercise 3.52

Consider the sequence of expressions

```
(define sum 0)
(define (accum x)
   (set! sum (+ x sum))
   sum)
(define seq (stream-map accum
             (stream-enumerate-interval 1 20)))
(define y (stream-filter even? seq))
(define z (stream-filter
            (lambda (x) (= (remainder x 5) 0))  seq))
(stream-ref y 7)
(display-stream z)
```

What is the value of sum after each of the above expressions is evaluated? What is the printed response to evaluating the stream-ref and display-stream expressions? Would these responses differ if we had implemented (delay <exp>) simply as (lambda () <exp>) without using the optimization provided by memo-proc? Explain.



3.5.2 Infinite Streams

- これまでのsequence は有限範囲.
- 無限ストリームを構築

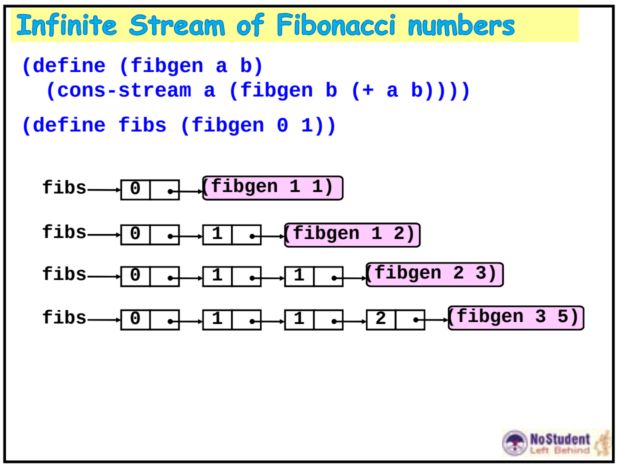
```
(define (integers-starting-from n)
   (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
```

integers → 1 | • → (integers-starting-from 2)

integers → 1 | • → 2 | • → (integers-starting-from 3)

```
(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



Infinite Stream of Fibonacci numbers

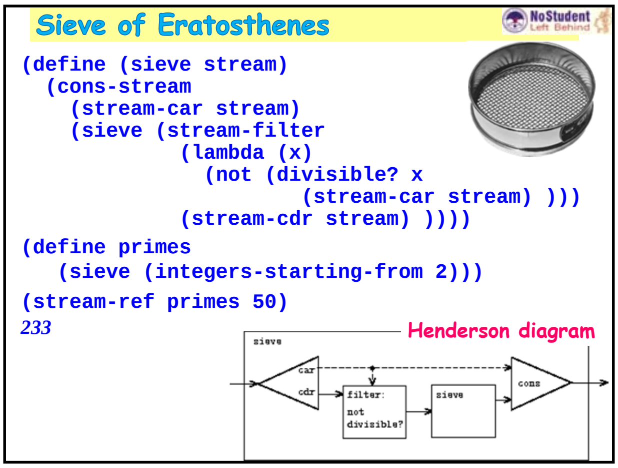
```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

fibs → 0 | • → (fibgen 1 1)

fibs → 0 | • → 1 | • → (fibgen 1 2)

fibs → 0 | • → 1 | • → 1 | • → (fibgen 2 3)

fibs → 0 | • → 1 | • → 1 | • → 2 | • → (fibgen 3 5)



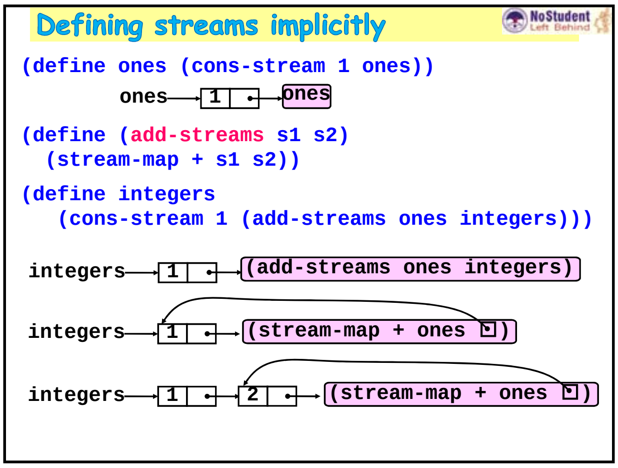
Sieve of Eratosthenes

```
(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)
```

233

Henderson diagram

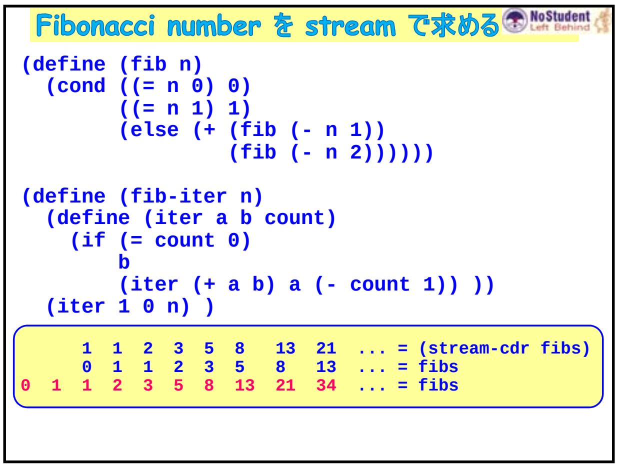
sieve
car
cdr
filter:
not
divisible?
sieve
cons



Defining streams implicitly

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

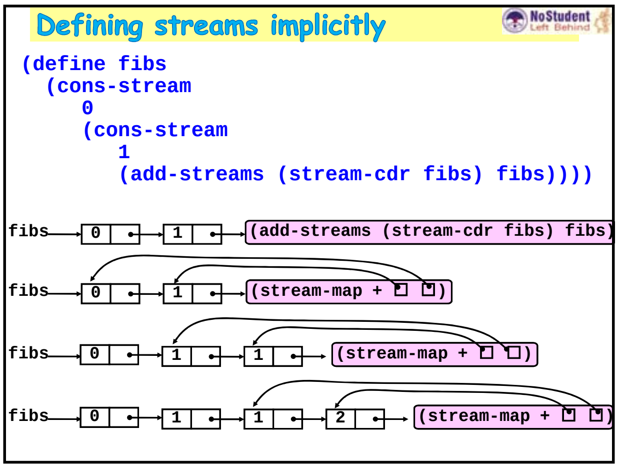
```
(define (add-streams s1 s2)
  (stream-map + s1 s2))
(define integers
   (cons-stream 1 (add-streams ones integers)))
```



Fibonacci number を stream で求める

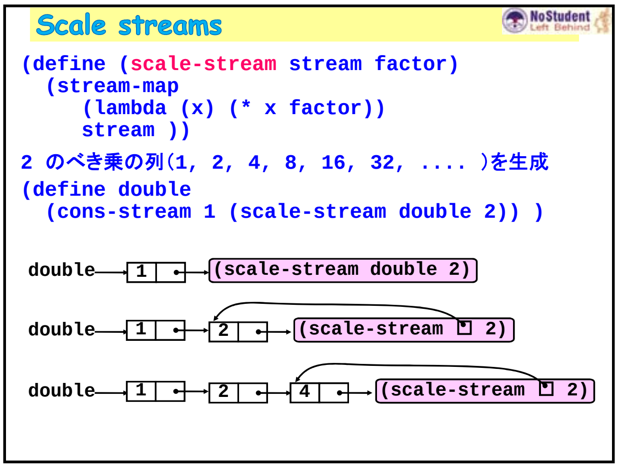
```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))

(define (fib-iter n)
  (define (iter a b count)
    (if (= count 0)
        b
        (iter (+ a b) a (- count 1)) ))
  (iter 1 0 n) )
```

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



Defining streams implicitly

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



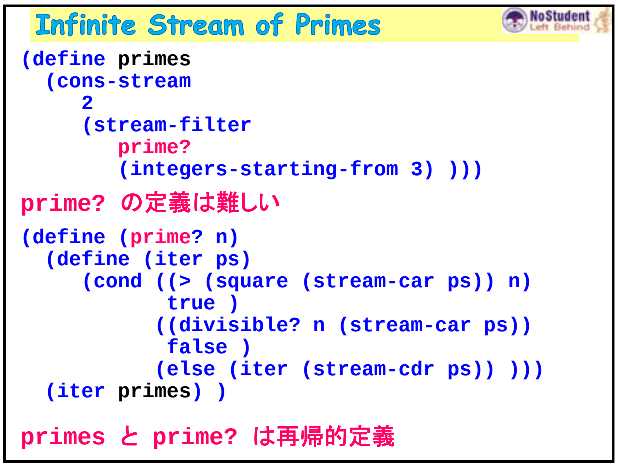
Scale streams

```
(define (scale-stream stream factor)
  (stream-map
     (lambda (x) (* x factor))
     stream ))
```

2 のべき乗の列(1, 2, 4, 8, 16, 32, ...)を生成

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

```
double→[1|•]→(scale-stream double 2)

double→[1|•]→[2|•]→(scale-stream □ 2)

double→[1|•]→[2|•]→[4|•]→(scale-stream □ 2)
```



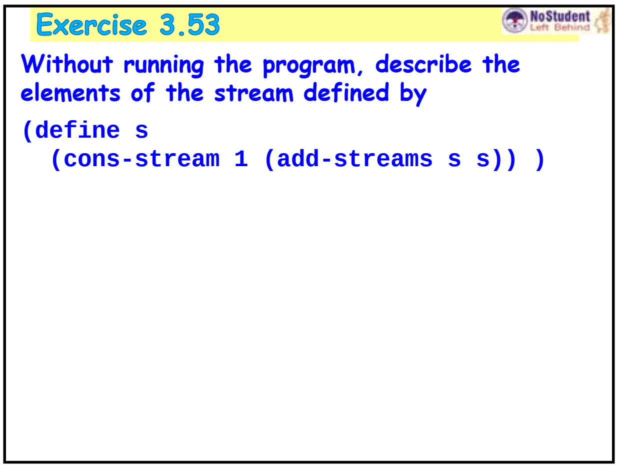
Infinite Stream of Primes

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

prime? の定義は難しい

```
(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) )
```

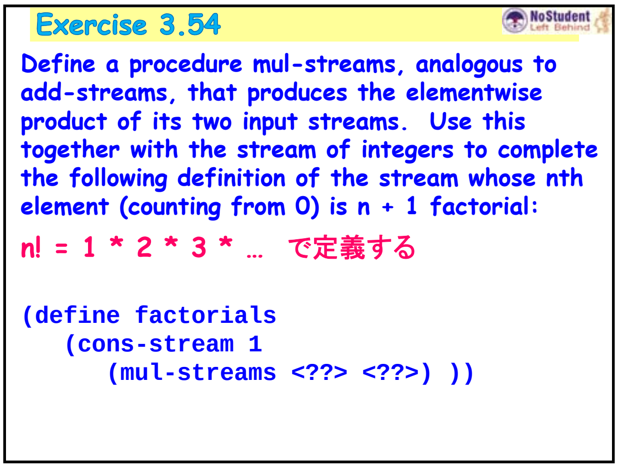
primes と prime? は再帰的定義



Exercise 3.53

Without running the program, describe the elements of the stream defined by

```
(define s
  (cons-stream 1 (add-streams s s)) )
```

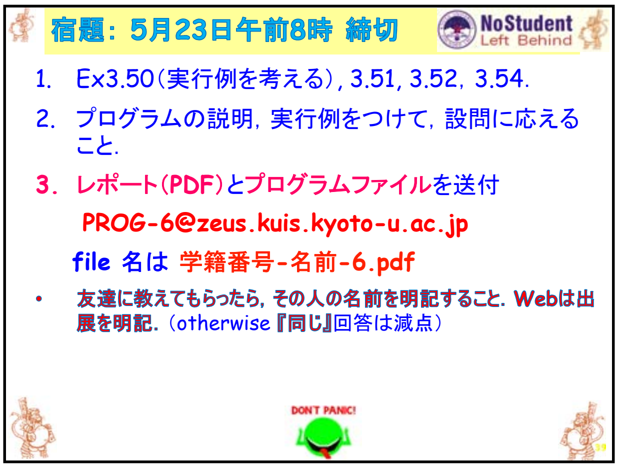


Exercise 3.54

Define a procedure mul-streams, analogous to add-streams, that produces the elementwise product of its two input streams. Use this together with the stream of integers to complete the following definition of the stream whose nth element (counting from 0) is n + 1 factorial:

n! = 1 * 2 * 3 * … で定義する

```
(define factorials
   (cons-stream 1
      (mul-streams <??> <??>) ))
```



宿題：5月23日午前8時 締切

1. Ex3.50（実行例を考える），3.51, 3.52, 3.54.
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-6@zeus.kuis.kyoto-u.ac.jp

   file 名は 学籍番号-名前-6.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）