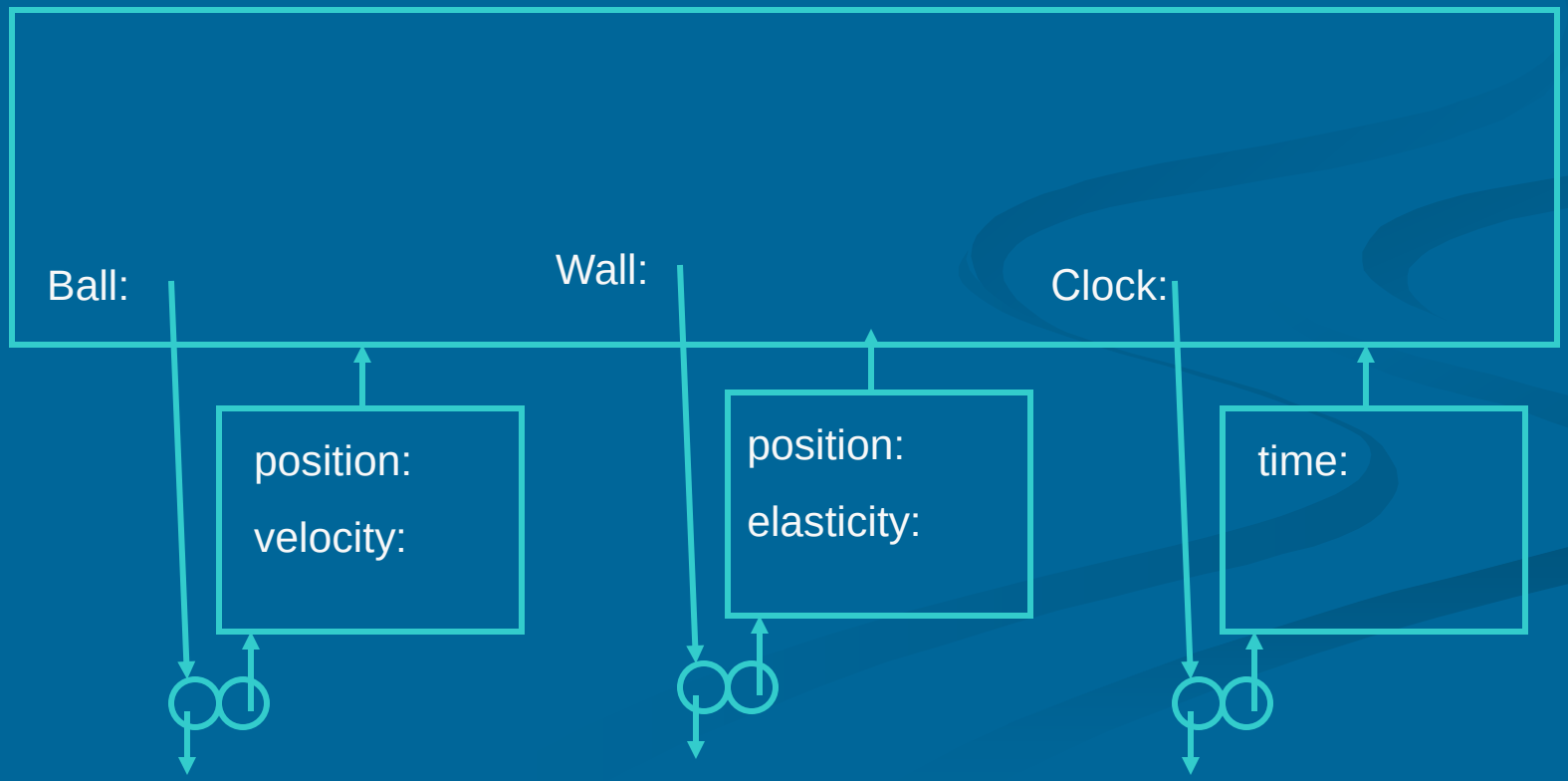
ЛЕКЦИЯ 8
Потоки

Потоки. Зачем?

- Предположим, что нужно промоделировать поведение системы мяч + стена
- Можно использовать объекты, часы, уравнения движения



Потоки. Зачем?

- Состояния системы описываются значениями атрибутов объектов в разные моменты времени

Clock: 1	Ball: (x1 y1)	Wall: e1
Clock: 2	Ball: (x2 y2)	Wall: e2
Clock: 3	Ball: (x3 y3)	Wall: e2
Clock: 4	Ball: (x4 y4)	Wall: e2
Clock: 5	Ball: (x5 y5)	Wall: e3
	…	

Потоки. Зачем?

- Те же данные можно рассматривать с другой точки зрения

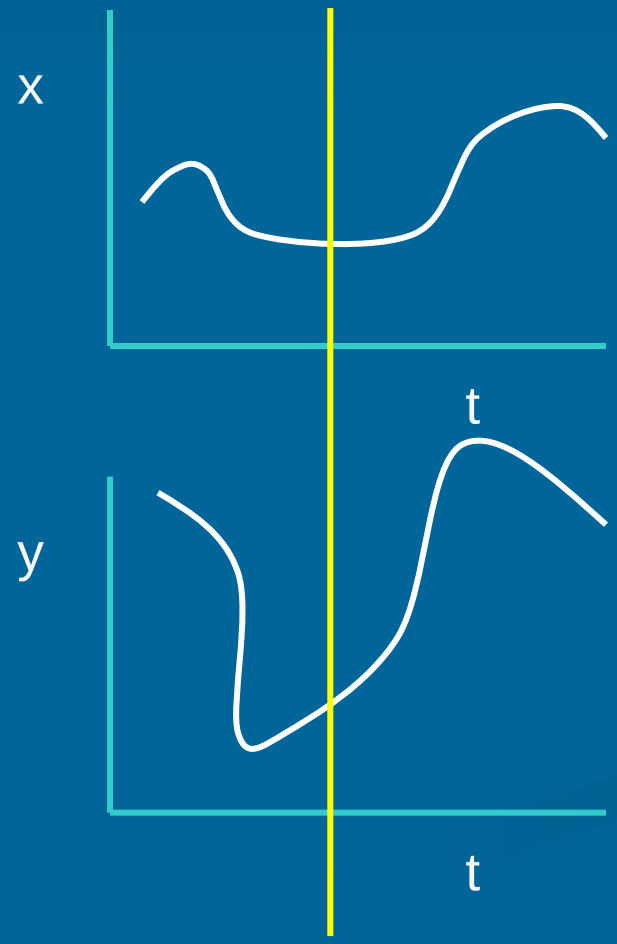
Clock:
1
2
3
4
5
…

Ball:
(x1 y1)
(x2 y2)
(x3 y3)
(x4 y4)
(x5 y5)
…

Wall:
e1
e2
e2
e2
e3
…

Потоки как альтернатива присваиванию

- Каждый объект – источник потока данных
- История изменения состояний модели представлена потоками данных



Ленивые вычисления

- Нормальный порядок вычислений:
  - вычисление нестрогой функции стартует до вычисления аргументов
  - подвыражение вычисляется только, если оно *необходимо*
    - для вывода
    - для функции строгой по этому аргументу (элементарные функции обычно строгие)
    - для проверки (в if, cond, case …)
    - для применения (1й элемент комбинации)
- Мемоизация – экономим вычисления за счёт запоминания вычисленных ранее результатов.
- Порядок, определяемый программистом: объявляя функцию указываем строгость по аргументам.

Как узнать порядок вычислений?

- Функция для визуализации:

```
(define (notice x)
    (display x)
    (newline)
    x)

(+ (notice 2) (notice 3)) ==>        2
                                     3
                                  5
```

Поток как ленивый список

- Ленивый cons, car, cdr

(require racket/stream)

(stream-cons first rest) -- формирует поток из головы и хвоста

(stream-first str) -- возвращает голову потока

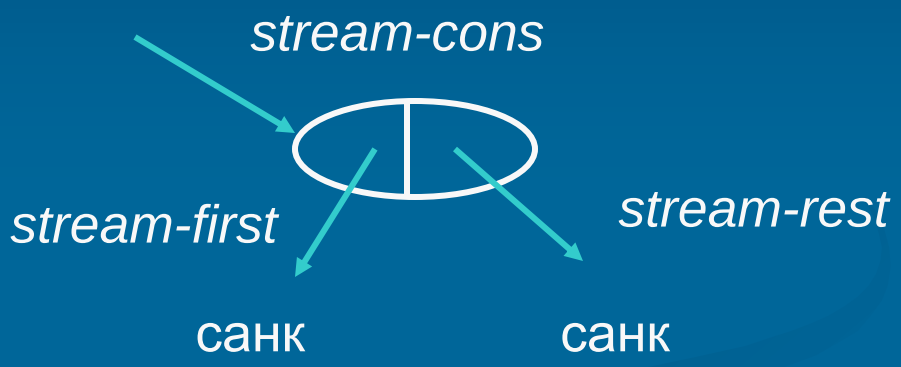
(stream-rest str) -- возвращает хвост потока

- Мы будем использовать потоки Racket, основная разница в названиях функций:

Racket	Scheme
stream-cons	cons-stream
stream-first	stream-car
stream-rest	stream-cdr
stream	list
empty-stream	null
stream-empty? ...	null? ...

Устройство потока

- Похож на пару, но *обе* части *ленивые*



- Санк (thunk) – обещание вычислить значение, когда оно будет *необходимо*.
- Санки мемоизированные (другие не рассматриваем).
- Пример

```
(define x (stream 99 (/ 1 0)))
(stream-first x) ==> 99
(stream-first (stream-rest x)) ==> /: division by zero
```

Визуализируем вычисление потока

- Рассмотрим пример

```
(define s (stream-map notice (stream 1 2 3 4 5 6 7 8 9 10)))
(stream-ref s 5) ==>      6
                          6
(stream-ref s 7) ==>      8
                          8
```

- остальные элементы потока s лишь санки

```
(stream-ref s 7) ==>  8
(stream-ref s 5) ==>  6
```

- мемоизация в действии!

Стиль программирования с потоками

- Простые описания в привычных терминах fold, map, filter ... + эффективные вычисления

со списком (не эффективно)

```
(list-ref (filter (lambda (x) (prime? x)) (enumerate-interval 2 100000000))  100)
```

с потоком (эффективно)

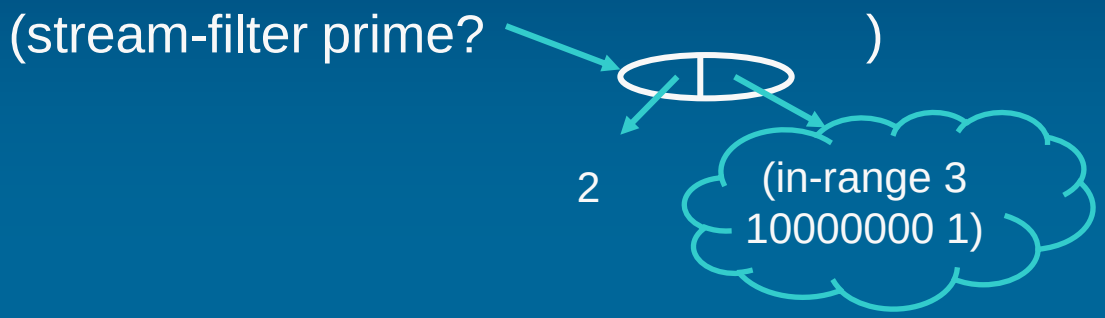
```
(stream-ref
  (stream-filter (lambda (x) (prime? x))
                 (in-range 2 100000000 1))
  100)
```

Аналог in-range можно реализовать самим:

```
(define (stream-interval a b step)
        (if (> a b) empty-stream
                    (stream-cons a (stream-interval (+ a step) b step))))
```

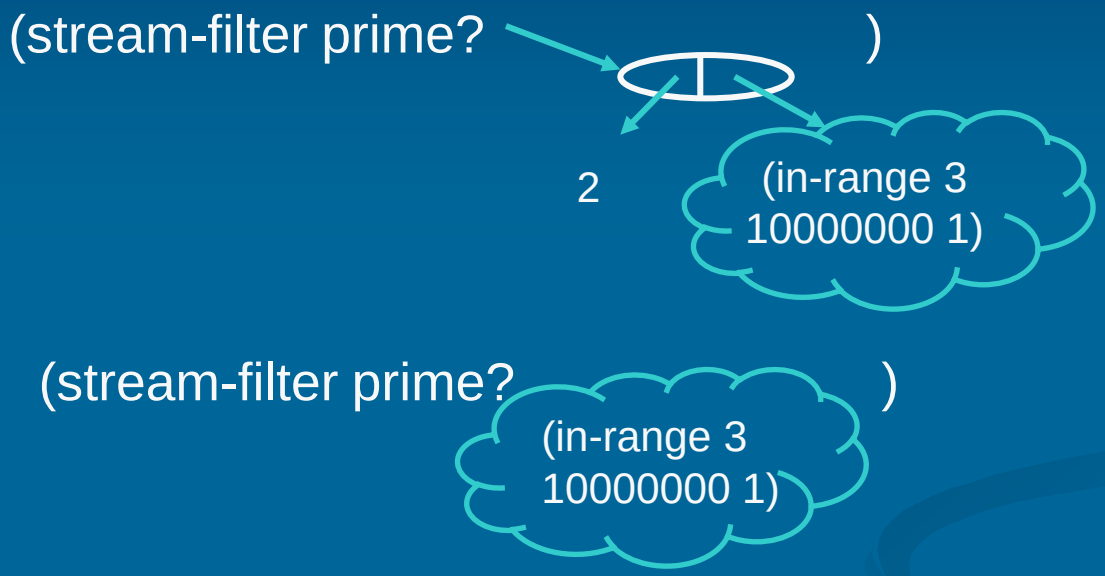
Вычисления в примере

(stream-filter prime? (in-range 2 100000000 1))



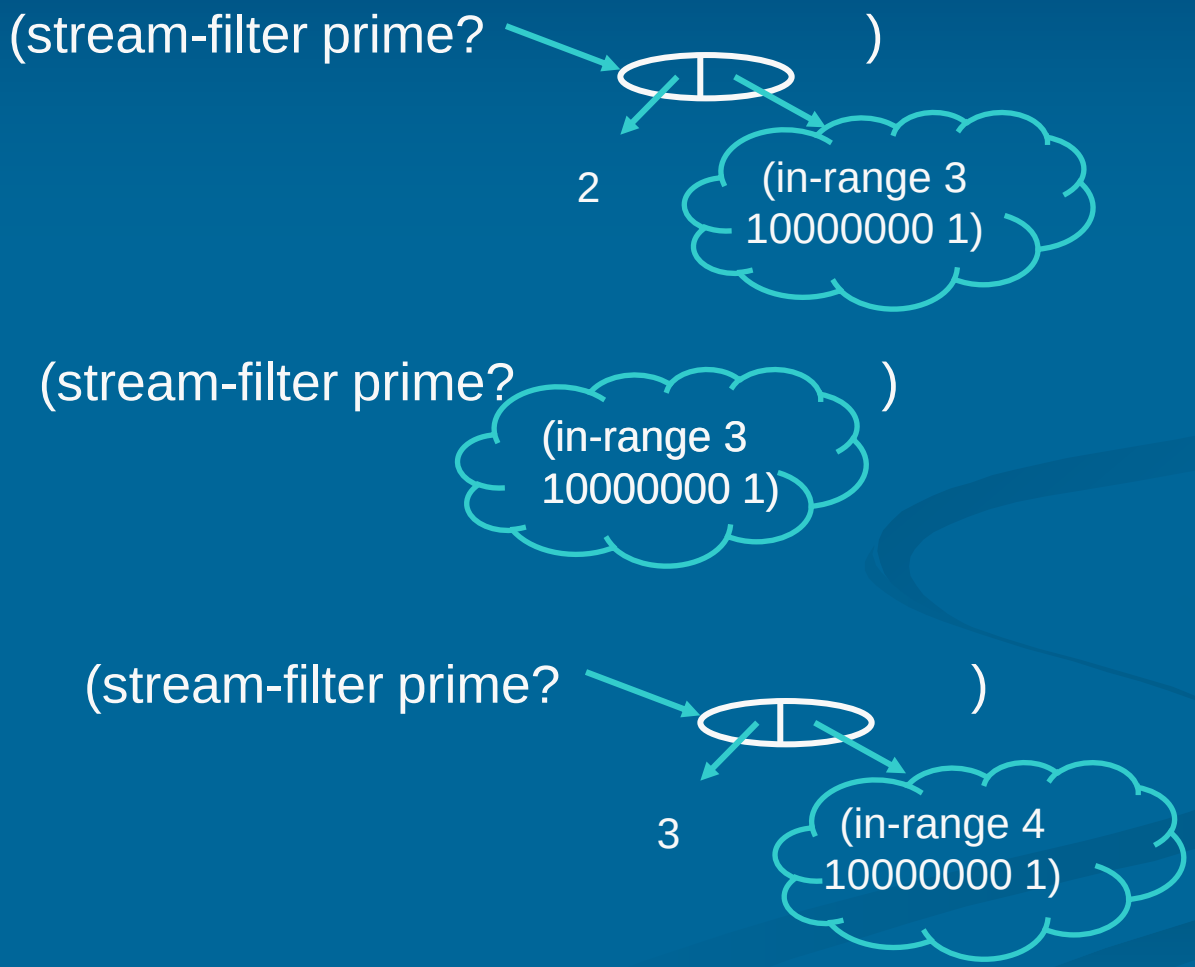
Вычисления в примере

(stream-filter prime? (in-range 2 100000000 1))



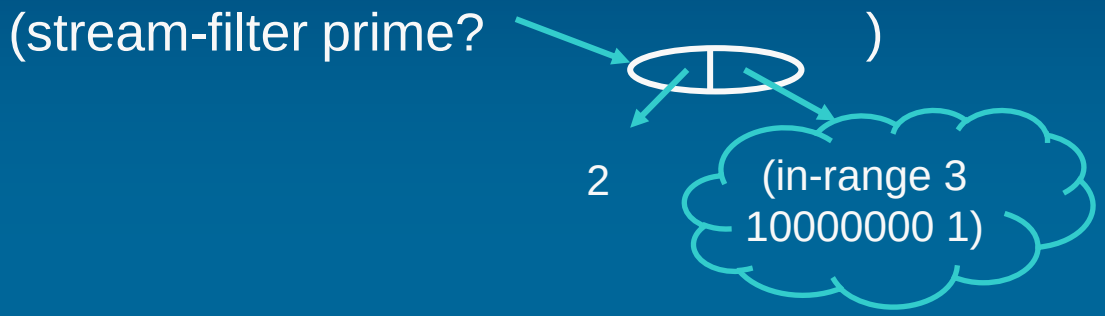
Вычисления в примере

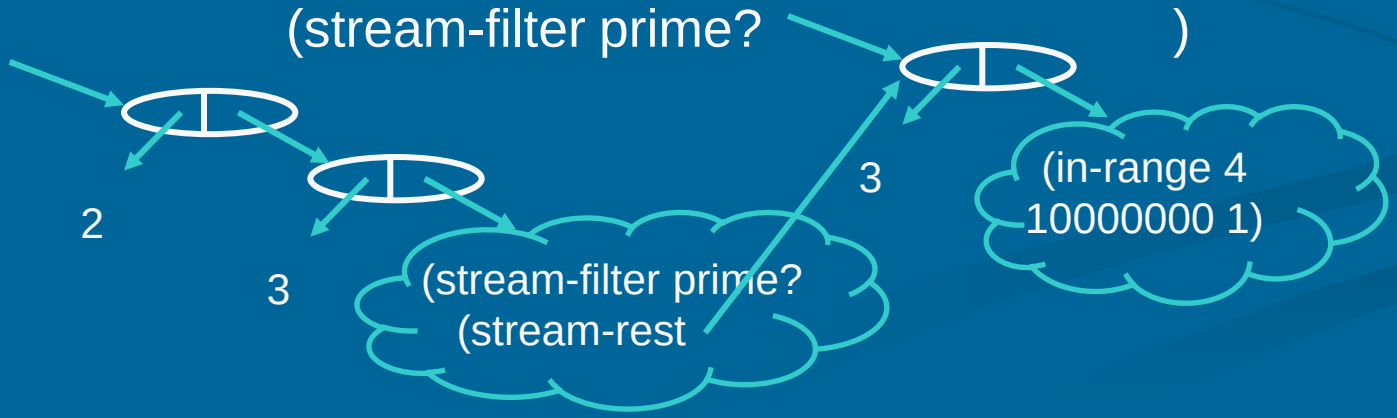
(stream-filter prime? (in-range 2 100000000 1))



Вычисления в примере

(stream-filter prime? (in-range 2 100000000 1))



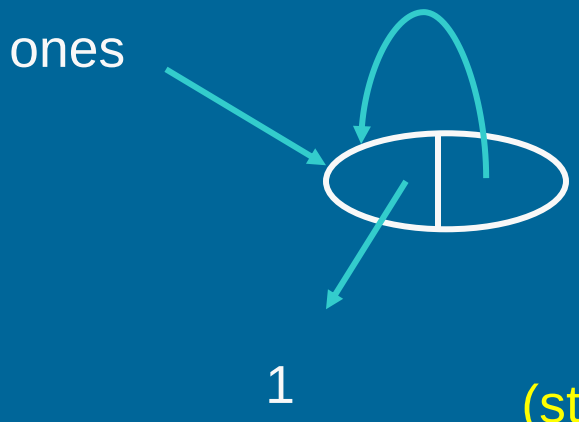


Ещё одна возможность: бесконечная структура данных!

- рассмотрим такое определение

(define ones (stream-cons 1 ones))

(stream-first (stream-rest ones)) ==> 1



Бесконечный ряд единиц!

ones: 1 1 1 1 1 1 ....

(stream-ref ones 1) ==>1

(stream-ref ones 1000) ==>1

(stream-ref ones 10000000) ==> 1

От конечных списков к бесконечным потокам

```
(define (streams-add s1 s2)
  (cond ((stream-empty? s1) empty-stream)
        ((stream-empty? s2) empty-stream)
        (else (stream-cons
            (+ (stream-first s1) (stream-first s2))
            (streams-add (stream-rest s1) (stream-rest s2))))))
(define ints
  (stream-cons 1 (streams-add ones ints)))
```

неявное описание ints

(+ (stream-first ones)
(stream-first ints))

(streams-add (stream-rest ones)
(stream-rest ints))

Явный способ задать ints

```
(define (ints-from n)   ; порождающая функция
  (stream-cons n (ints-from (+ n 1))))
(define ints (ints-from 1))
```

Поток чисел Фибоначчи

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

(stream-rest fibs):		1	1	2	3	5	8	...
fibs:		0	1	1	2	3	5	...
fibs:	0	1	1	2	3	5	8	13 ...

или:

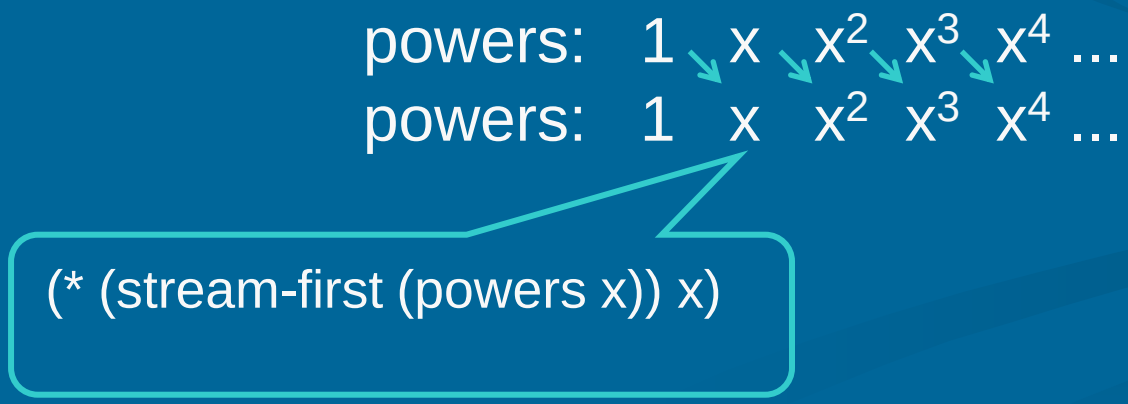
```
(define (fibgen a b) ; порождающая функция
  (stream-cons a (fib-gen (b (+ a b))))
(define fibs (fibgen 0 1))
```

Ещё одна функция для неявного порождения

```
(define (stream-scale s f)
  (stream-map (lambda (x) (* x f)) s))
```

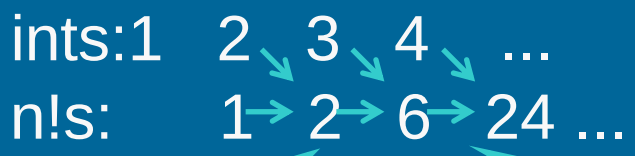
неявное описание потока степеней (1 x x^2 x^3 ...)

```
(define (powers x)
  (stream-cons 1 (stream-scale (powers x) x)))
```



Поток факториалов

```
(define (streams-mul s1 s2)
  (cond ((stream-empty? s1) empty-stream)
        ((stream-empty? s2) empty-stream)
        (else (stream-cons
           (* (stream-first s1) (stream-first s2))
           (streams-mul (stream-rest s1) (stream-rest s2))))))
(define n!s  (stream-cons 1 (streams-mul n!s
                                         (stream-rest ints))))
```




```
(* (stream-first n!s)
   (stream-ref ints 1))
```

```
(streams-mul (stream-rest n!s)
 (stream-rest (stream-rest ints)))
```

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли первые числа от 2 до 100

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли 2 в ответ и вычеркнули всё, что делится на 2

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли 3 в ответ и вычеркнули всё, что делится на 3

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли 5 в ответ и вычеркнули всё, что делится на 5

Поиск всех простых чисел

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

взяли 7 в ответ и вычеркнули всё, что делится на 7

Поиск всех простых чисел

	2	3		5		7			
11		13				17		19	
		23						29	
31						37			
41		43				47			
		53						59	
61						67			
71		73						79	
		83						89	
						97			

получили список всех простых чисел, меньших 100

Метод просеивания

```
(define (sieve str)
  (stream-cons
    (stream-first str)
    (sieve (stream-filter
            (lambda (x)
              (not (= 0 (remainder x (stream-first str)))))
            (stream-rest str)))))

(define primes
    (sieve (stream-rest ints)))
```

(2 (sieve (stream-filter (lambda ... 2...) (stream-rest ints))))

(2 3 (sieve (stream-filter (lambda ... 3 ...)
(sieve (stream-filter (lambda ... 2 ...) (stream-rest ints)))
))

Слияние бесконечных потоков

```
(define (interleave str1 str2)
  (stream-cons (stream-first str1)
               (interleave str2 (stream-rest str1))))
(interleave ones ints)
```

```
ones:  1  1  1  1  1  1 ...
ints:     1  2  3  4  5  ...
       ---------------------
       1 1 1 2 1 3 1 4 1 5 1 ...
```

Поток рациональных чисел

```
(define (div-by-stream n s)
    (stream-cons (/ n (stream-first s))
                 (div-by-stream n (stream-rest s))))
(define (make-rats n)
    (stream-cons n
        (interleave (div-by-stream n (stream-rest ints))
            (make-rats (+ n 1)))))
(define rats (make-rats 1))
```

```
1/1   1/2   1/3   1/4   1/5   ...
2/1   2/2   2/3   2/4   2/5   ...
3/1   3/2   3/3   3/4   3/5   ...
4/1   4/2   4/3   4/4   4/5   ...
...
1/1  1/2  2/1  1/3  2/2  1/4  3/1  1/5  2/3  ...
```

Степенные ряды как потоки

$g(x) = g(0) + x\,g'(0) + x^2/2\,g''(0) + x^3/3!\,g'''(0) + \ldots$

Например:

$\exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24 + \ldots$

$\cos(x) = 1 - x^2/2 + x^4/24 - \ldots$

$\sin(x) = x - x^3/6 + x^5/120 - \ldots$

Степенные ряды как потоки

```
(define (series-approx coeffs) ; результат – функция конструктор ряда
  (lambda (x)
    (streams-mul
      (streams-div (powers x) (stream-cons 1 n!s))
      coeffs)))
```

$g(x) = g(0) + x\,g'(0) + x^2/2\,g''(0) + x^3/3!\,g'''(0) + \ldots$

```
(define (stream-accum str) ; результат – ряд частичных сумм
  (stream-cons (stream-first str)
               (streams-add (stream-accum str)
                            (stream-rest str))))
```

- $\Rightarrow$ $g(0) = S_0$
- $\Rightarrow$ $g(0) + x\,g'(0) = S_1 = S_0 + x\,g'(0)$
- $\Rightarrow$ $g(0) + x\,g'(0) + x^2/2\,g''(0) = S_2 = S_1 + x^2/2\,g''(0)$
- $\Rightarrow$ $g(0) + x\,g'(0) + x^2/2\,g''(0) + x^3/3!\,g'''(0) = S_3 = S_2 + x^3/3!\,g'''(0)$

Степенные ряды как потоки

```
(define (power-series g) ; даст функцию для ряда Sn
  (lambda (x)
    (stream-accum ((series-approx g) x))))

(define  exp-coeffs ones)  ; поток коэффициентов exp(x)
(define sine-coeffs  ; поток коэффициентов sin(x)
  (stream 0 1 0 (stream-cons -1 sine-coeffs)))
(define cos-coeffs (stream-rest sine-coeffs))
; поток коэффициентов cos(x)

(define (exp-approx x) ; ряд приближений exp(x)
   ((power-series exp-coeffs) x))
(define (sine-approx x) ; ряд приближений sin(x)
   ((power-series sine-coeffs) x))
(define (cos-approx x) ; ряд приближений cos(x)
   ((power-series cos-coeffs) x))
```

Потоковый поиск квадратного корня

- Создадим поток приближений к корню

```
(define (sqrt-improve guess x); получение следующего
                  (average guess (/ x guess)))
(define (sqrt-stream x)
  (stream-cons 1.0
     (stream-map (lambda (g) (sqrt-improve g x))
                 (sqrt-stream x))))

(define sqrt2 (sqrt-stream 2))
(stream-ref sqrt2 0) ==> 1.0
(stream-ref sqrt2 1) ==> 1.5
(stream-ref sqrt2 2) ==> 1.4166666666666665
(stream-ref sqrt2 3) ==> 1.4142156862745097
(stream-ref sqrt2 4) ==> 1.4142135623745899 ...
```

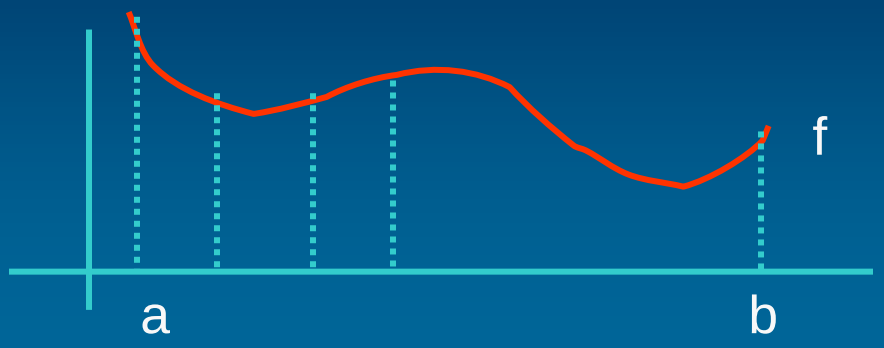
Потоковый поиск квадратного корня

- К генерирующей части добавим проверяющую.

```
(define (stream-limit s tol)
  (define (iter s)
    (let ((f1 (stream-first s))
          (f2 (stream-first (stream-rest s))))
      (if (close-enough? f1 f2 tol)
          f2
          (iter (stream-rest s)))))
   (iter s))
(stream-limit (sqrt-stream 2) 1.0e-5) ==> 1.412135623746899
```

- Вычисление описано как генерация приближений и их проверка.

Потоковое интегрирование



метод трапеций

```
(define (trapezoid f a b h) ; сумма метода трапеций ∫ab f(x)dx  с шагом h
  (let ((dx (* (- b a) h))
        (n (/ 1 h)))
    (define (iter j sum)
      (if (>= j n)
          sum
          (iter (+ j 1) (+ sum (f (+ a (* j dx)))))))
    (* dx (iter 1 (+ (/ (f a) 2)
                     (/ (f b) 2))))))
```

Потоковое интегрирование

$\pi = 4*\mathrm{arctg}(1) = \int_0^1 4dx/(1+x^2)$

```
(define (witch x) (/ 4 (+ 1 (* x x))))
```

```
(trapezoid witch 0 1 0.1) ==> 3.1399259889071587
```

```
(trapezoid witch 0 1 0.01) ==> 3.141575986923129
```

- Получили приближения к π, которые тем лучше, чем меньше высота трапеций (шаг разбиения).

Потоковое интегрирование

из прака 1-го курса известно, что шаг нужно мельчить пополам

```
(define (keep-halving r h) ; генерация потока
   (stream-cons  (r h) (keep-halving r (/ h 2))))
```

для проверки подходит stream-limit

```
(stream-limit
  (keep-halving (lambda (h) (trapezoid witch 0 1 h)) 0.5)
  1.0e-9)
```

==> 3.14159265343456

Итоги лекции 8

- Ленивые вычисления полезны.
- Мемоизация полезна при ленивых вычислениях.
- Программирование с потоками – интересный альтернативный метод описания вычислений.