ЛЕКЦИЯ 6
Присваивание.
Модель окружений

Потребность в присваивании

- Рассмотрим пример: денежный счёт
- Пусть операция withdraw снимает деньги со счёта

(withdraw 25) → 75

(withdraw 25) → 50

(withdraw 55) → “Not enough money!”

(withdraw 15) → 35

- Пусть текущий баланс является значением balance

(define balance 100)

- В теле withdraw должно быть что-то, меняющее значение balance

…

(set! balance (- balance amount))

…

Специальная форма set!

- (set! <имя> <новое значение>)
- как работает:
  - <имя> должно быть определено!!!
  - вычисляет <новое значение>
  - связывает его с переменной <имя>
  - не вычисляет <имя>!!!
  - значение формы зависит от реализации (#<void>)
- определение withdraw

```
(define (withdraw amount)
   (if   (>= balance amount)
         (begin (set!  balance (- balance amount))
                balance)
         “Not enough money”))
```

Последствия присваивания

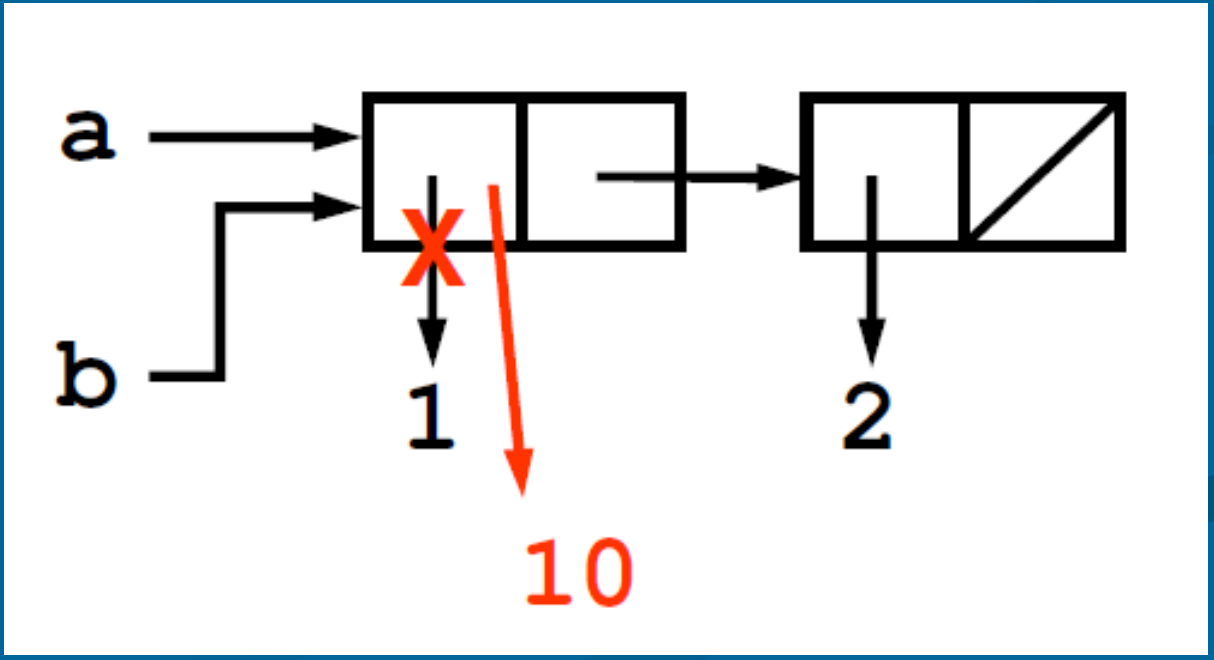
- Присваивание делает неактуальной подстановочную модель!
- Рассмотрим withdraw без присваивания

```
(define (withdraw2 amount)
    (if    (>= balance amount) (- balance amount)
           “Not enough money”))
(withdraw2 20)          → 80
(withdraw2 20)          → 80
(withdraw 20)           →
(if (>= 100 20)
           (begin (set! balance (- 100 20))
                  100)
           “Not enough money”))          → 100?
```

Присваивание и точечные пары

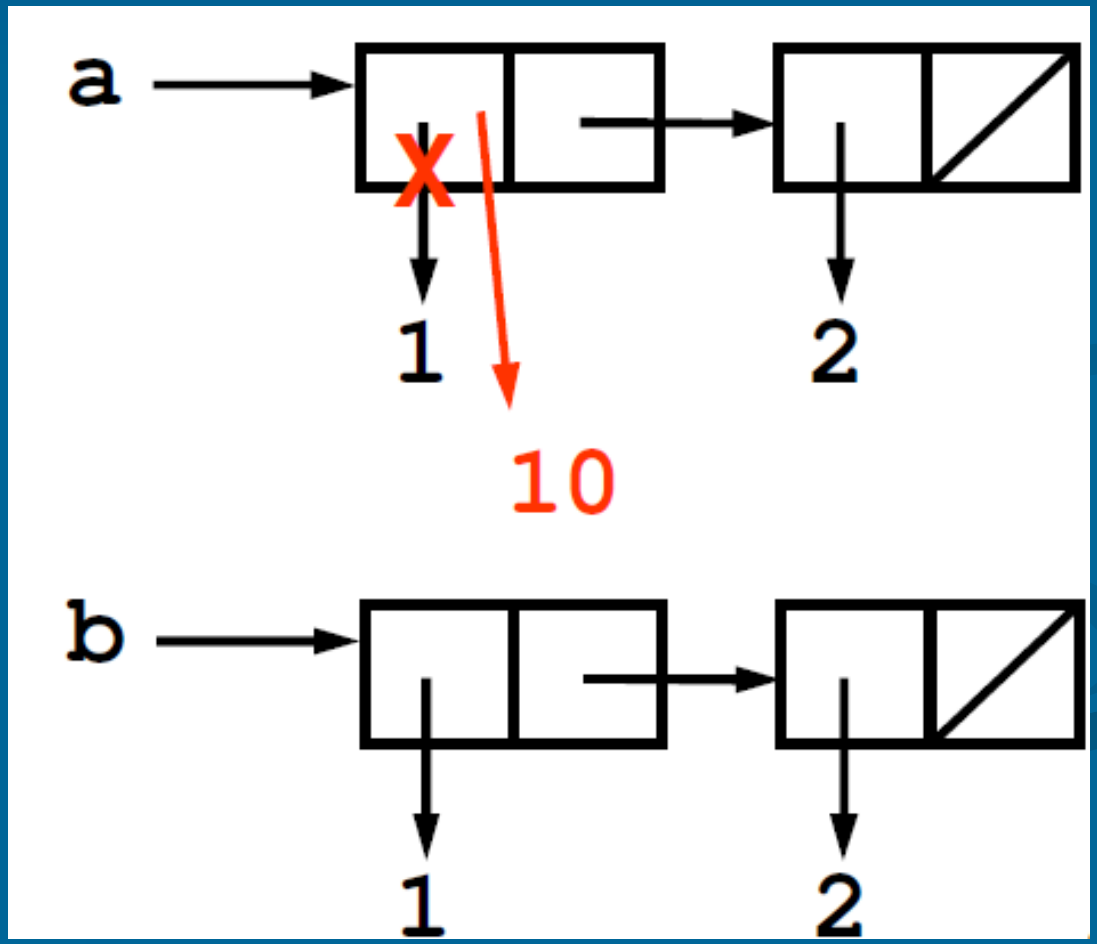
- (set-car! <пара> <новый car>) меняет 1й элемент точечной пары
- (set-cdr! <пара> <новый cdr>) меняет 2й элемент точечной пары

(define a (list 1 2))
(define b a)
a → (1 2)
b → (1 2)
(set-car! a 10)
b → (10 2)



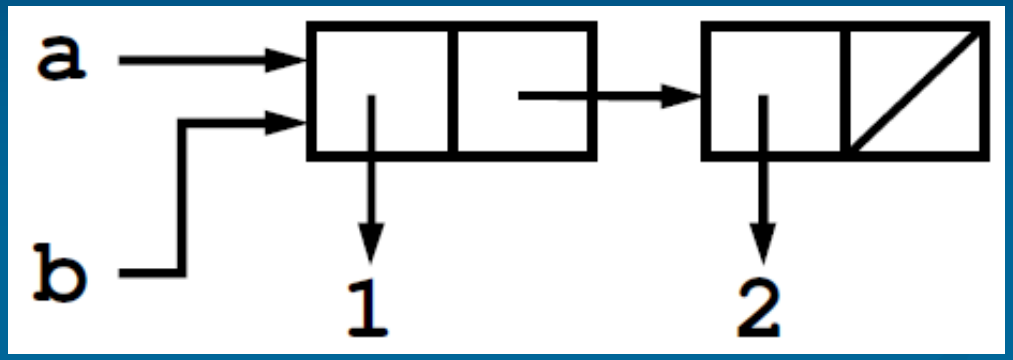
Что будет, если b определить иначе

(define a (list 1 2))
(define b (list 1 2))
a → (1 2)
b → (1 2)
(set-car! a 10)
a → (10 2)
b → (1 2)



Тождественность, эквивалентность

- Два объекта могут совпадать
  - (eq? a b) → #t



- Два объекта могут одинаково выглядеть
  - (equal? (list 1 2) (list 1 2)) → #t
  - (eq? (list 1 2) (list 1 2)) → #f

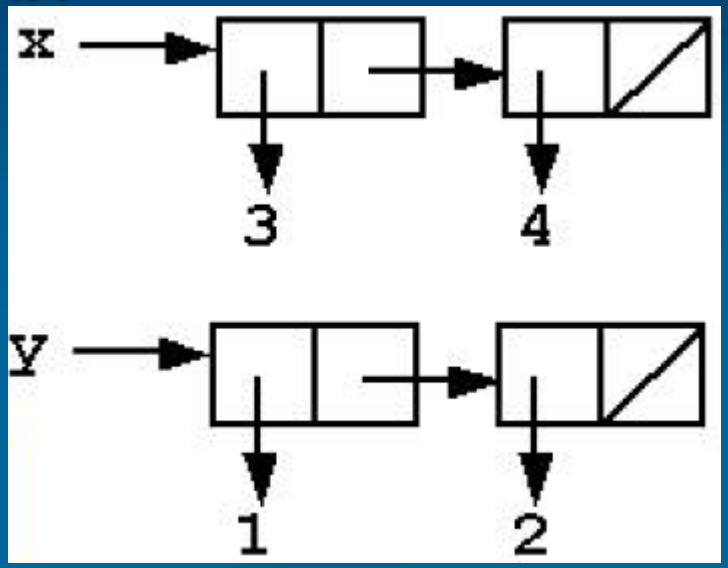
  но не совпадать

Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

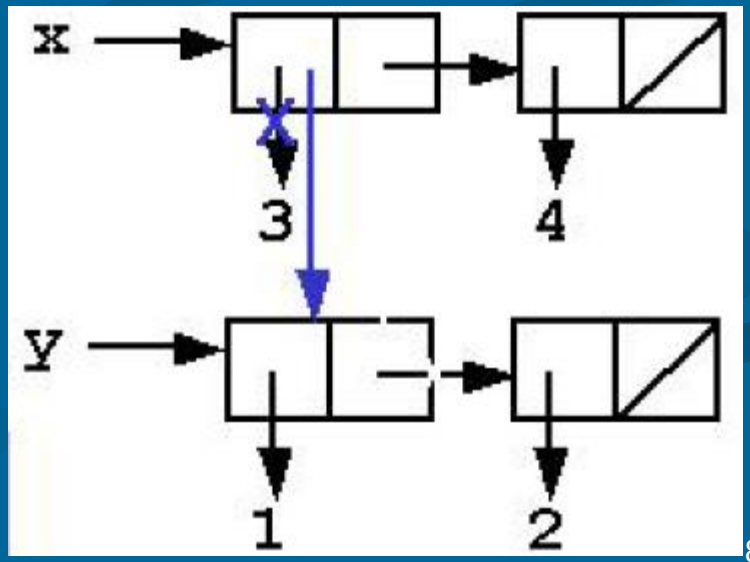
(define x ‘(3 4))

(define y ‘(1 2))



(set-car! x y)

x → ((1 2) 4)



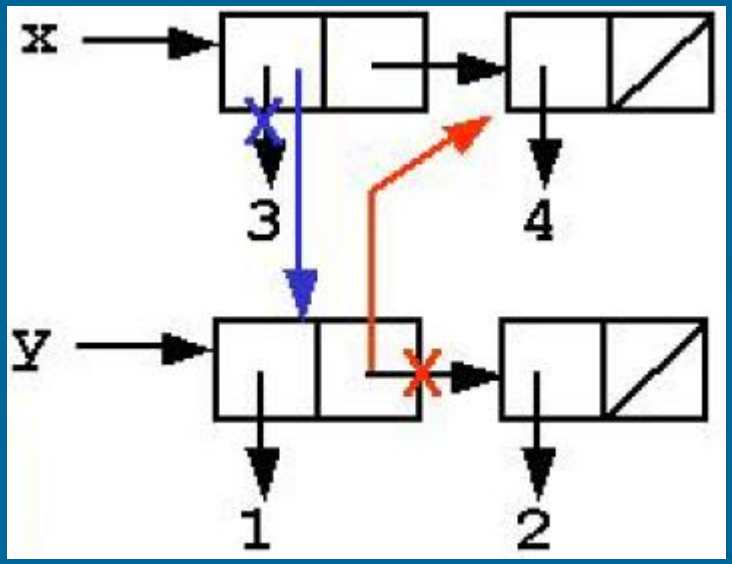
Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

…

(set-cdr! y (cdr x))

x → (1 4)



- Порядок вычислений влияет на результат

(* (withdraw 10) (withdraw 40)) → (* 90 50) → 4500

(* (withdraw 40) (withdraw 10)) → (* 60 50) → 3000

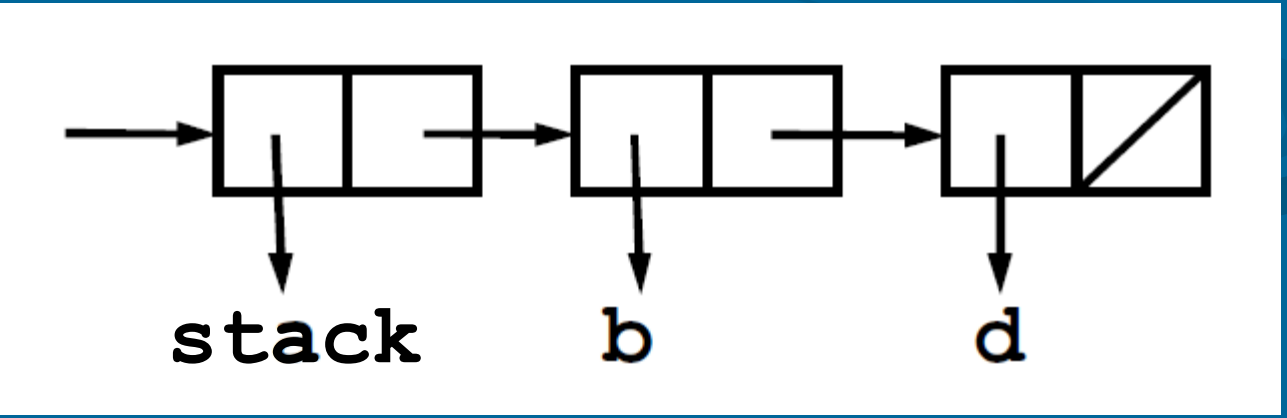
Промежуточный итог

- Встроенные функции Scheme для присваивания (мутаторы):
  - set! (есть в scheme/base)
  - set-car! set-cdr! (нет в scheme/base)
  - есть дополнительный модуль для mutable-структур scheme/mpair со своим набором функций mcons, mcar, mcdr, set-mcar!, set-mcdr!, mlist

- Присваивание создаёт дополнительные трудности:
  - возникают сторонние эффекты;
  - подстановочная модель больше не подходит для описания работы программ

Стек

- Конструктор (make-stack)
- Селектор (top-stack s)
- Операции:
  - (insert-stack! s e)
  - (delete-stack! s)
  - (stack? s)
  - (empty-stack? s)
- Стек легко реализовать как список с заглавным звеном.

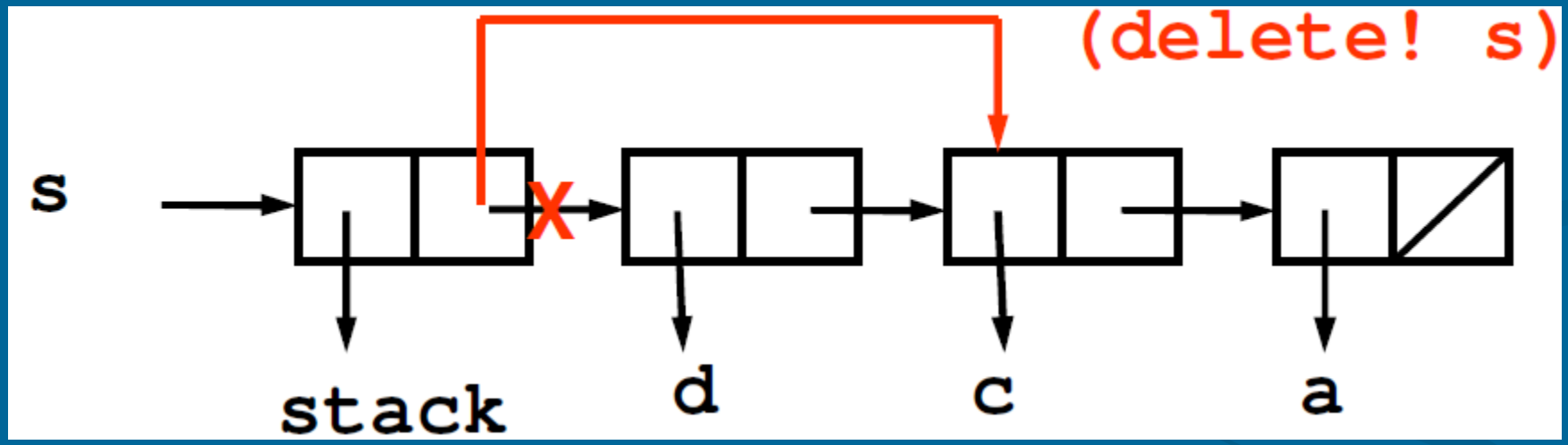


Стек. Реализация

```
(require scheme/mpair)
(define (make-stack)
        (mcons 'stack '()))

(define (stack? s) ; anytype -> boolean
        (and (mpair? s) (eq? 'stack (mcar s))))

(define (empty-stack? s) ; Stack<A> -> boolean
        (and    (stack? s)
                (null? (mcdr s))))

(define (insert-stack! s e) ; Stack<A>, A -> Stack<A>
        (if     (stack? s)
                (set-mcdr! s (mcons elt (mcdr s)))
                s))
```

Стек. Продолжение реализации

```
(define (delete-stack! s) ; Stack<A> -> Stack<A>
        (if     (and (stack? s) (not (empty-stack? s)))
                (set-mcdr! s (mcdr (mcdr s)))
                s))
```




```
(define (top-stack s) ; Stack<A> -> A
        (if     (and (stack? s) (not (empty-stack? s)))
                (mcar (mcdr s))
                “empty stack"))
```

Д/з Реализуйте очередь

- Конструктор (make-queue)
- Селектор (front-queue q)
- Операции:
  - (insert-queue! q e)
  - (delete-queue! q)
  - (queue? q)
  - (empty-queue? q)
- Сложность вставки должна быть константой!

Вернёмся к счетам

```
(define (make-withdraw balance)
    (lambda (amount)
        (if     (>= balance amount)
                (begin (set! balance (- balance amount))
                       balance)
                “Not enough money”)))
(define w1 (make-withdraw 100))
(define w2 (make-withdraw 100))
```

(w1 50) → 50

(w2 70) → 30

(w2 40) → “Not enough money”

(w1 40) → 10

Счета-объекты

```
(define (make-account balance)
    (define (withdraw amount)
        (if     (>= balance amount)
                (begin (set! balance (- balance amount))
                       balance)
                “Not enough money”))
    (define (deposit amount)
        (set! balance (+ balance amount)) balance)
    (define (dispatch m)
        (cond  ((eq? m ‘withdraw) withdraw)
               ((eq? m ‘deposit) deposit)
               (else display)))
    dispatch)
```

Счета-объекты. Продолжение

(define acc (make-account 100))

((acc ‘withdraw) 50) → 50

((acc ‘withdraw) 60) → “Not enough money”

((acc ‘deposit) 60) → 110

↑ ↑ ↑ программирование с передачей сообщений

(define acc2 (make-account 100))

↑ ↑ ↑ другой объект-счёт

(define acc3 acc2)

↑ ↑ ↑ разделяемый объект-счёт

((acc2 'deposit) 15) → 115

((acc3 'deposit) 15) → 130

Модель вычислений с окружениями (МВО)

- позволяет описать, как работает программа с присваиваниями

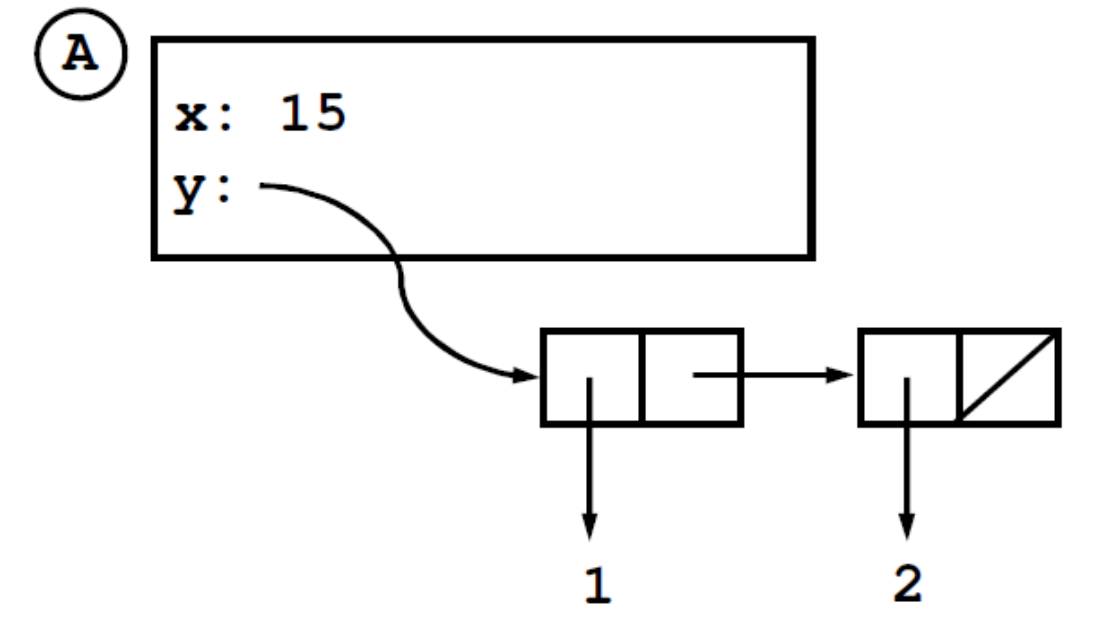
- в подстановочной модели имя – метка для значения
- в МВО имя – место для хранения значения

- в ПМ функция – описание аргументов и тела
- в МВО функция – объект со своим окружением

- Значение выражения зависит от окружения, в котором вычисляется выражение

Элементы МВО

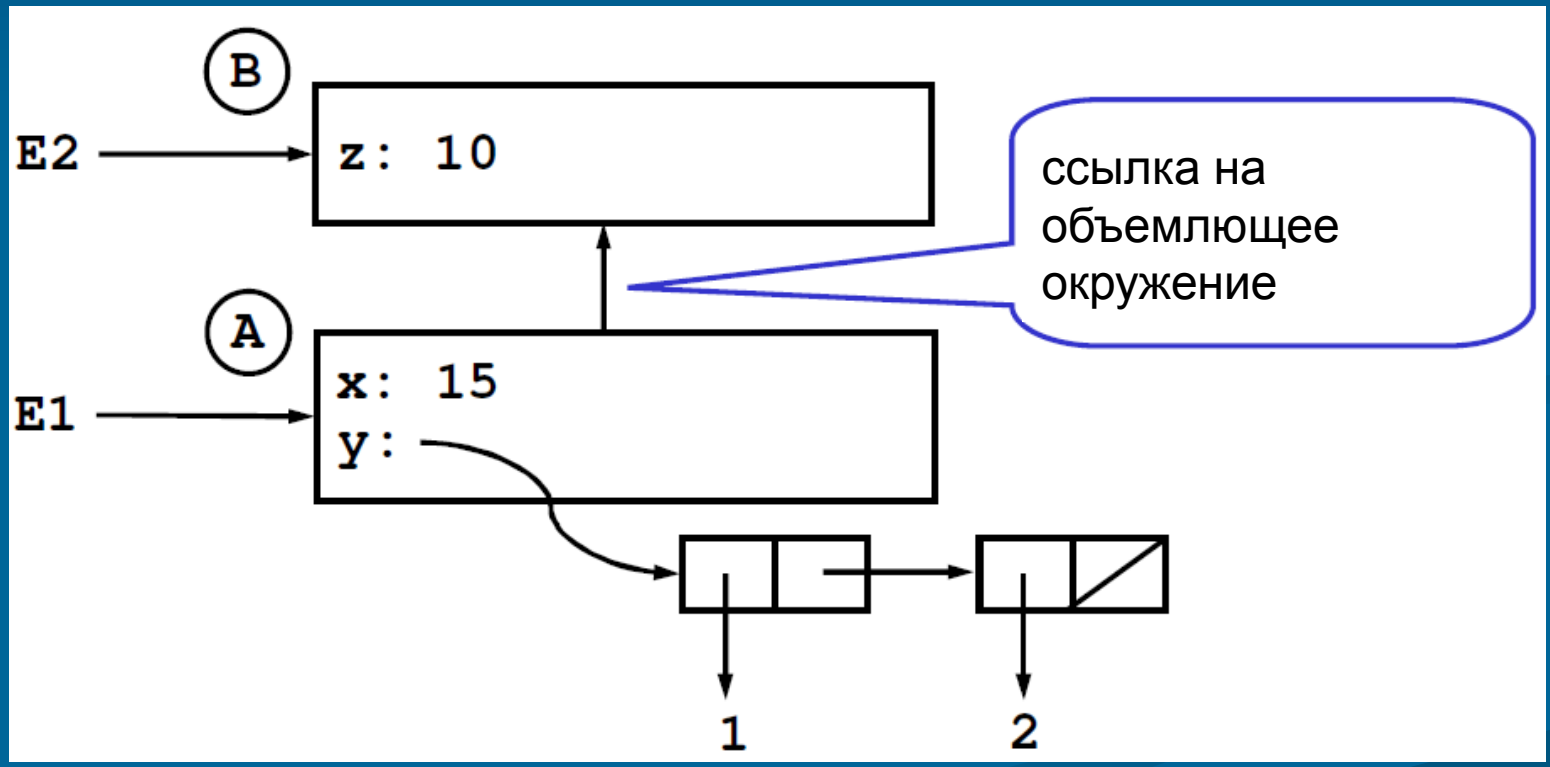
- Кадр – таблица связываний
- Связывание – пара имя : значение
- Пример:



- В кадре А содержатся два связывания

Элементы МВО

- Окружение – последовательность кадров
- Пример:



- Окружения E1 и E2 разделяют общий кадр B

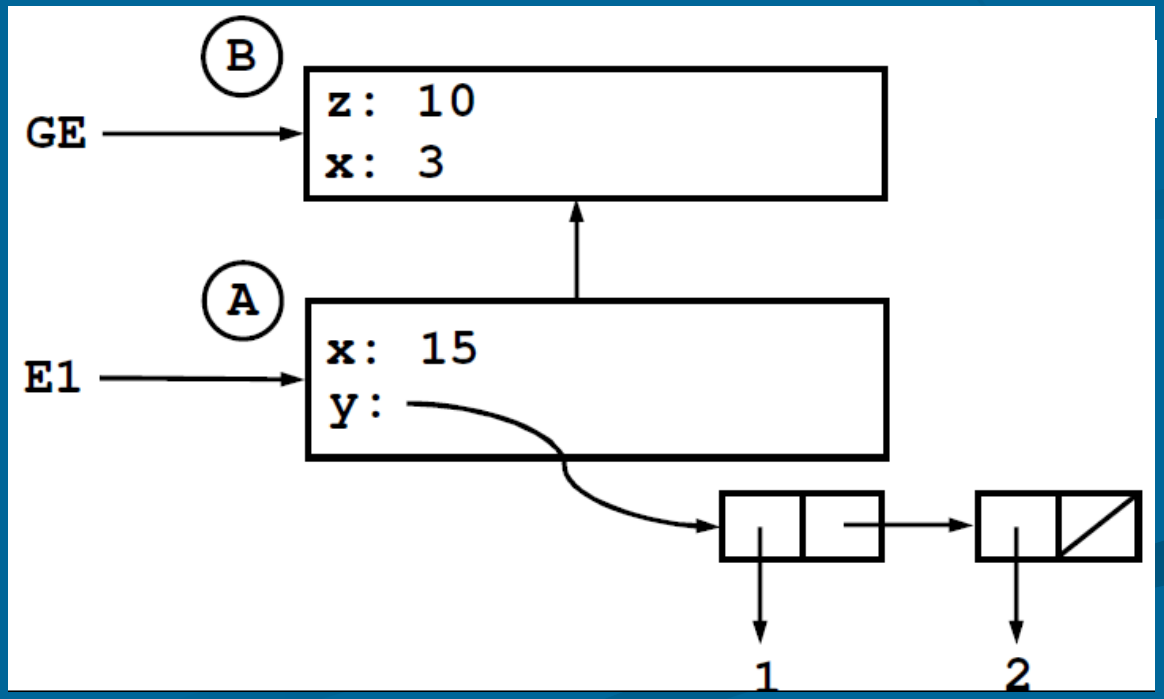
МВО

- Кадр может не иметь ссылки на объемлющее окружение, если это кадр глобального окружения.
- Все вычисления осуществляются внутри окружений.
- Текущее окружение меняется всякий раз при вычислении функции.
- Правила вычислений:
  - Чтобы вычислить комбинацию, следует вычислить все её части и применить первую часть к остальным.

    ↑ ↑ ↑ порядок вычисления частей неопределён!

МВО. Правила вычислений

- Вычисление имени X в окружении E:
  - Найти первый кадр, в котором есть связывание для X.
  - Взять в качестве результата значение из этого связывания.

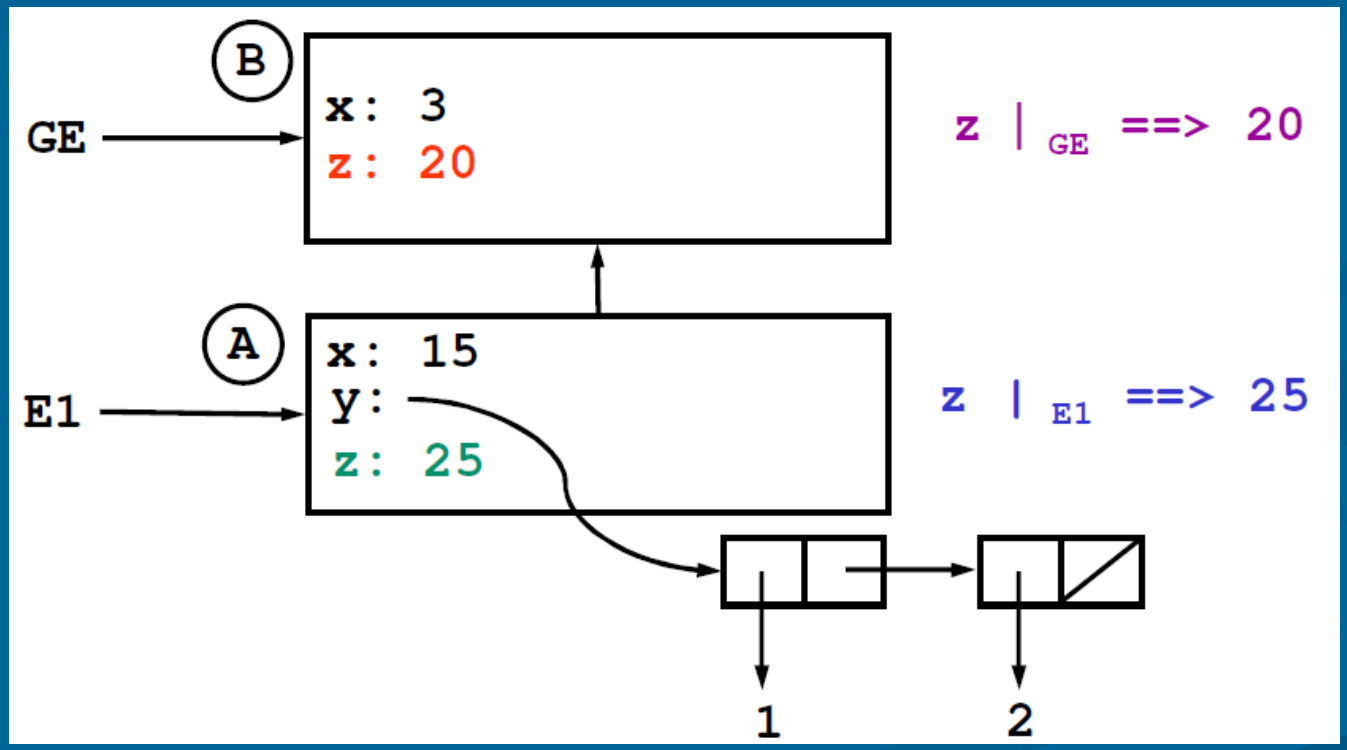


  - Остальные связывания X *скрыты*.

МВО. Правила вычислений

- Вычисление define в окружении:

  добавить новое связывание в первый кадр окружения.



define в глобальном окружении и в окружении E1

Если в кадре уже было связывание – изменить его.

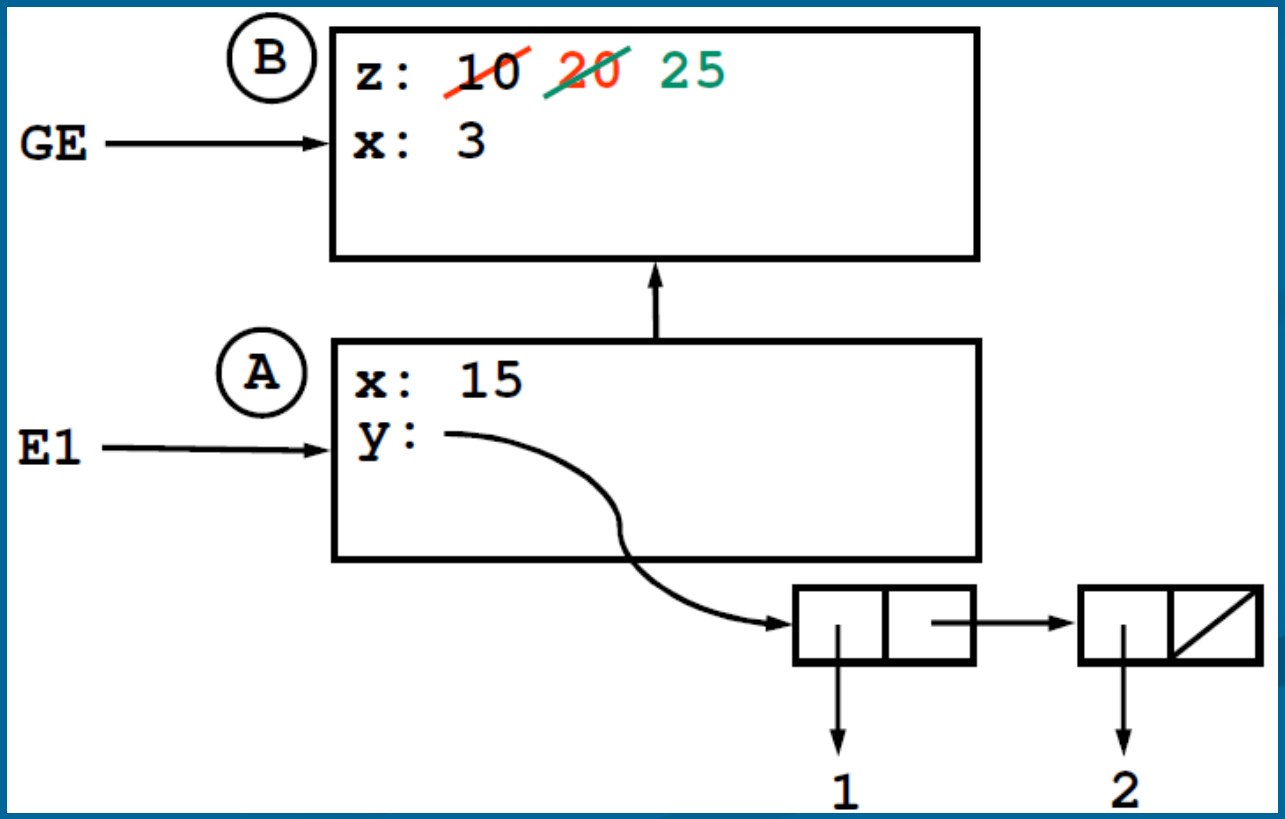
МВО. Правила вычислений

- Вычисление set! в окружении:

  изменить связывание имени в том кадре окружения, где оно впервые встретится.

(set! z 20) $|_{GE}$

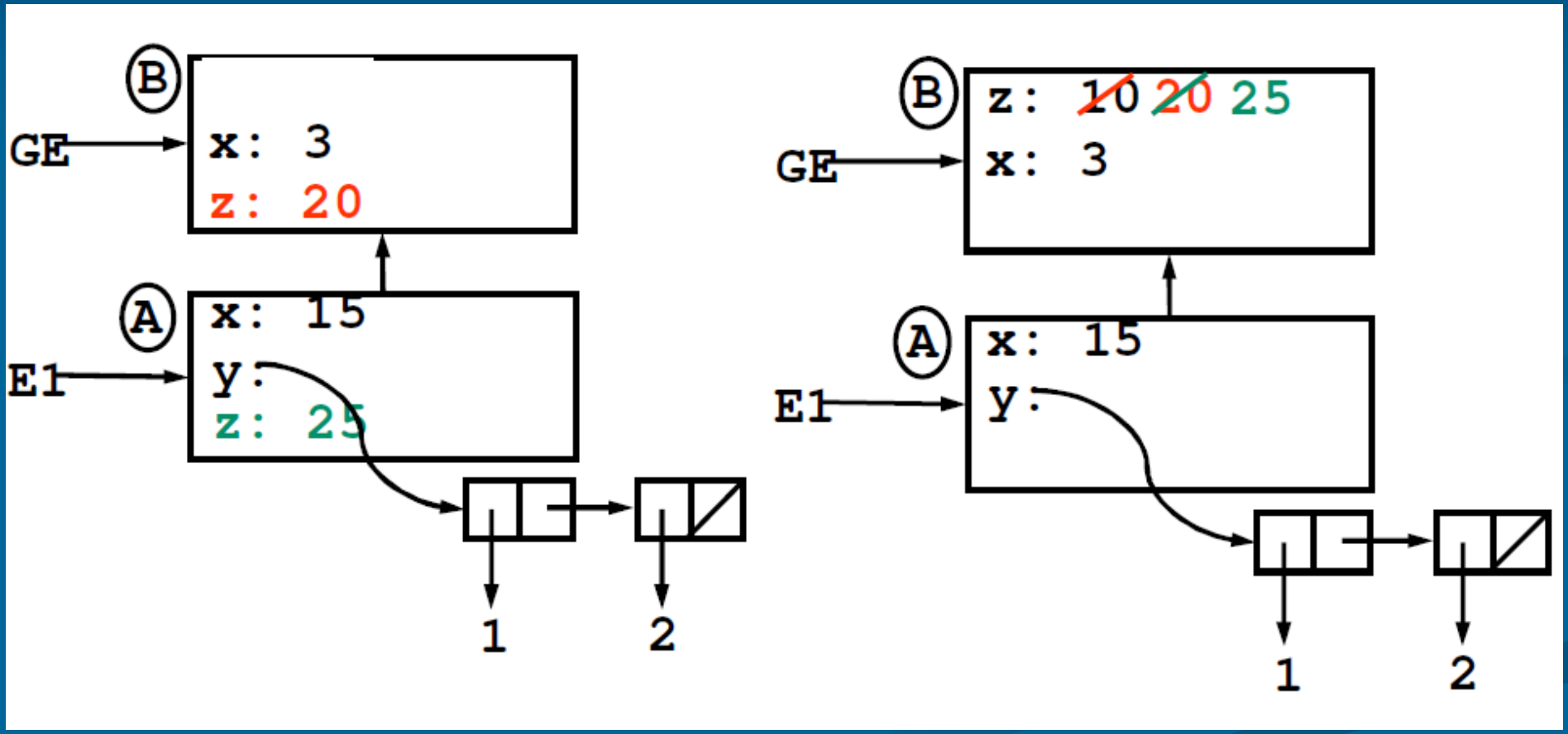
(set! z 25) $|_{E1}$



Если имя нигде не встретилось выдать ошибку!

define и set! вычисляются не одинаково

(define z 25) $|_{E1}$ (set! z 25) $|_{E1}$

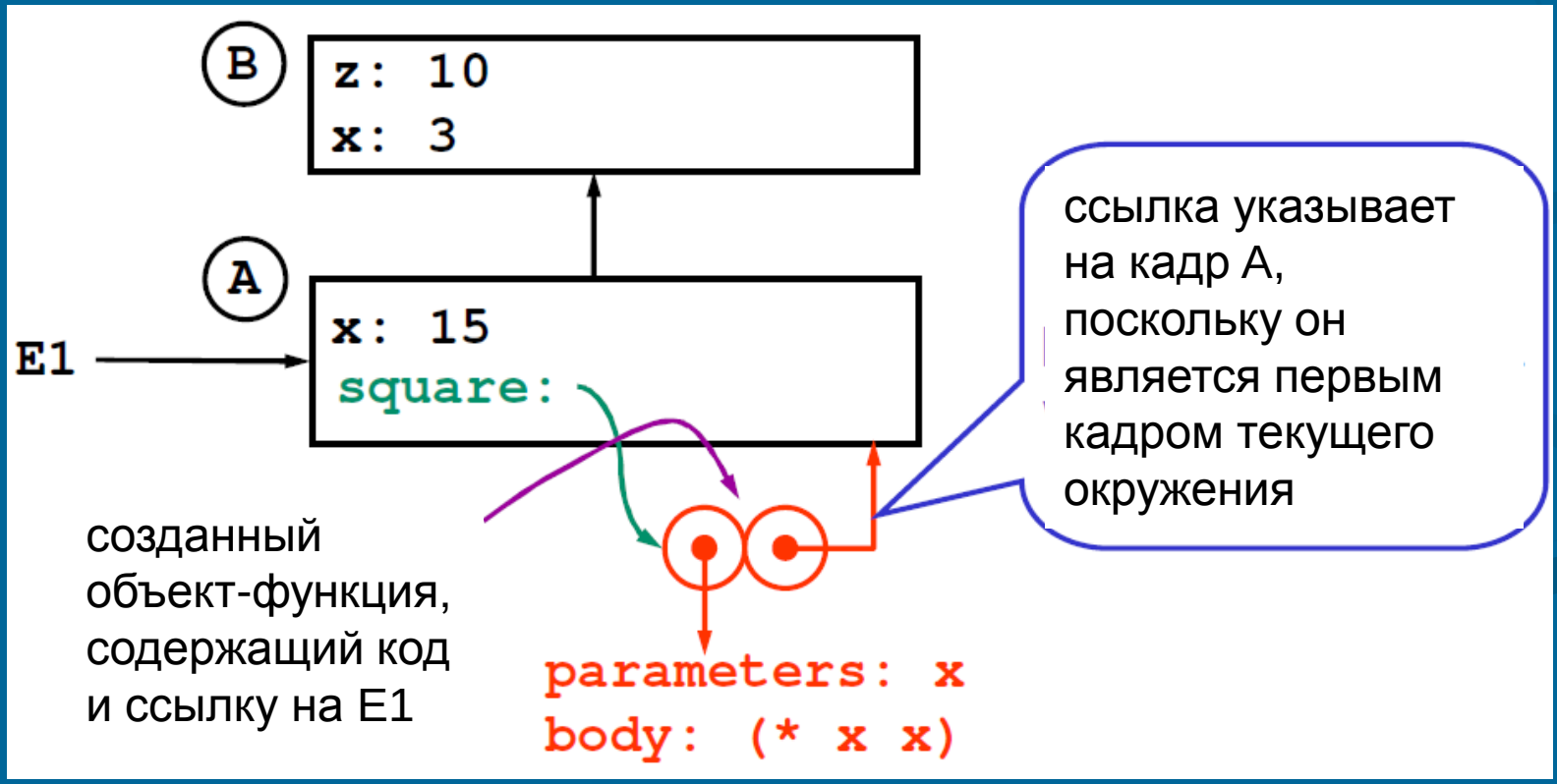


МВО. Правила вычислений

- Вычисление lambda в окружении:

  создать функцию с телом из lambda и ссылкой на текущее окружение

(define square (lambda (x) (* x x))) |$_{E1}$

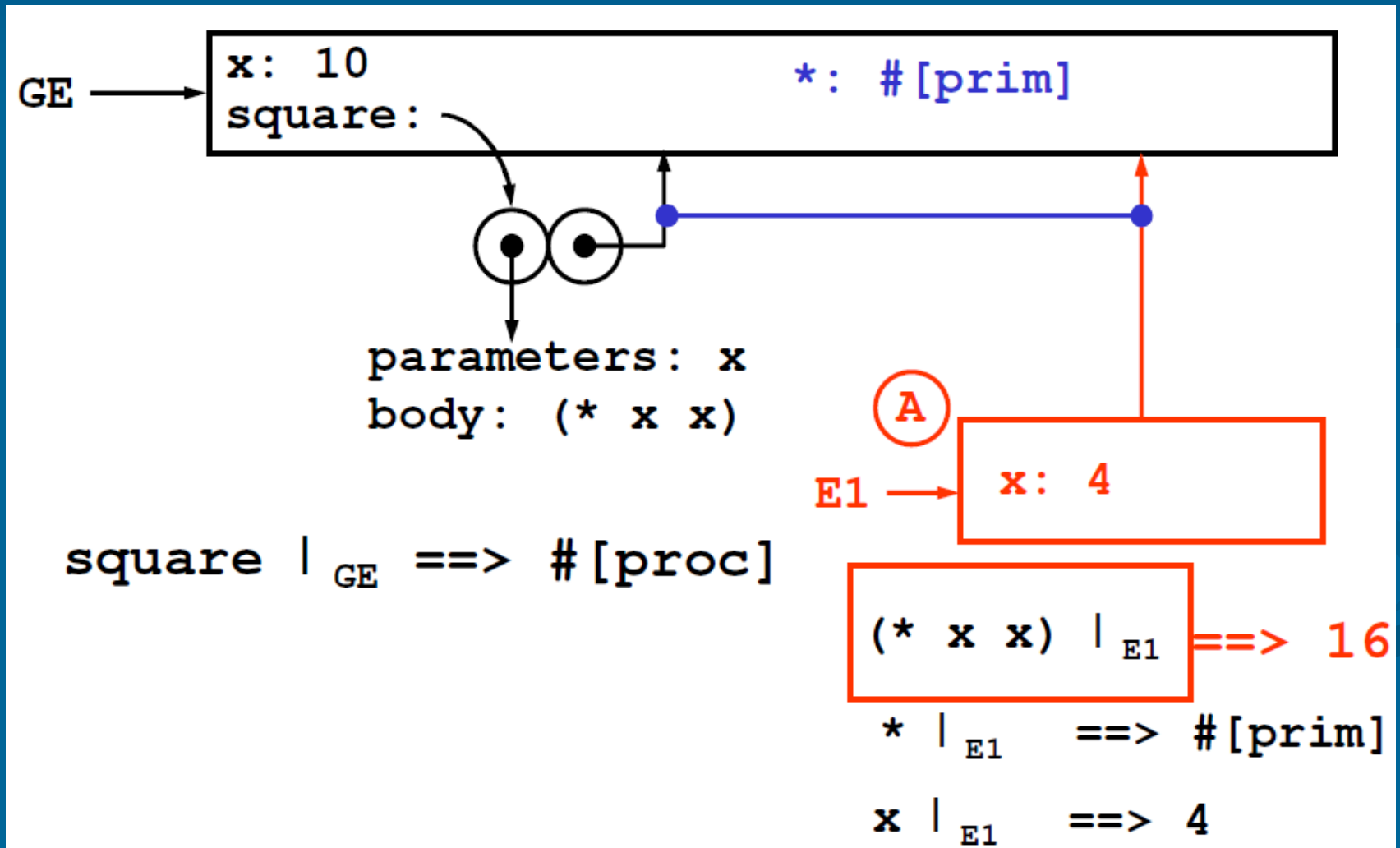


МВО. Правила вычислений

- Применение функции в окружении:
  - создать новый кадр A
  - сделать его первым кадром нового окружения E1
  - провести ссылку на объемлющее окружение от кадра A к окружению, на которое указывает ссылка объекта-функции
  - в кадр A добавить связывания всех аргументов функции со значениями из вызова функции
  - вычислить тело функции в построенном окружении E1.

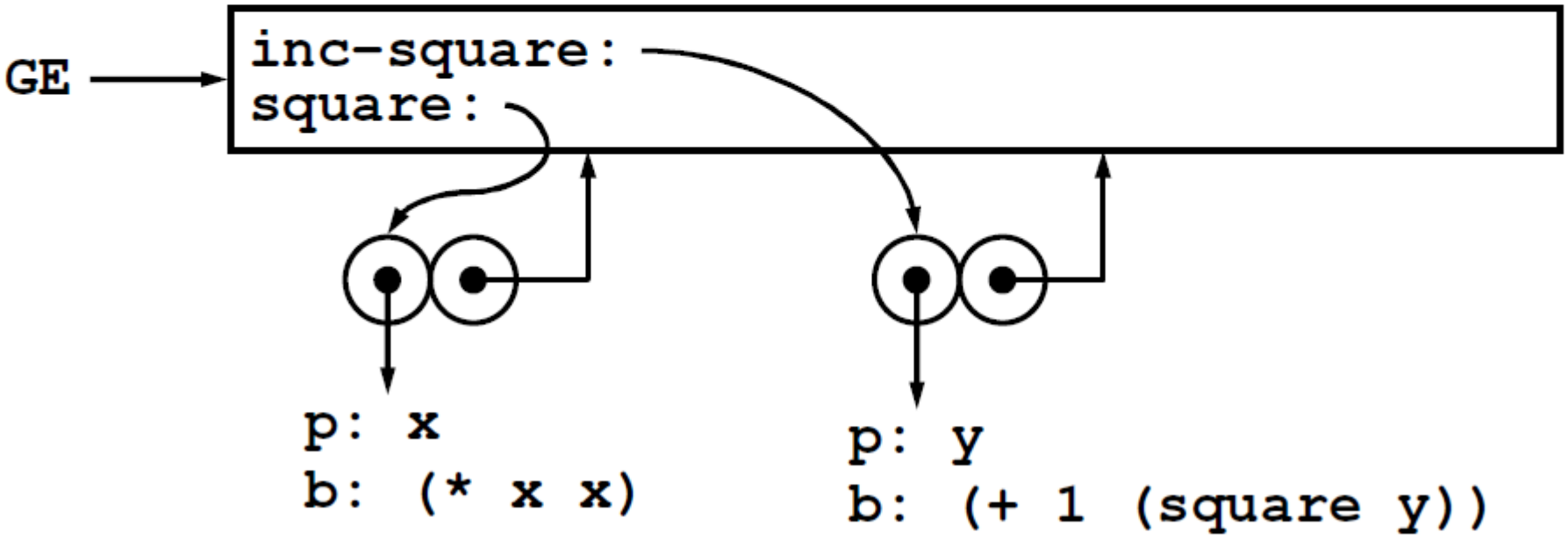
Пример применения функции

- Пусть square описана и вызвана в глобальном окружении: (square 4) I_{GE}



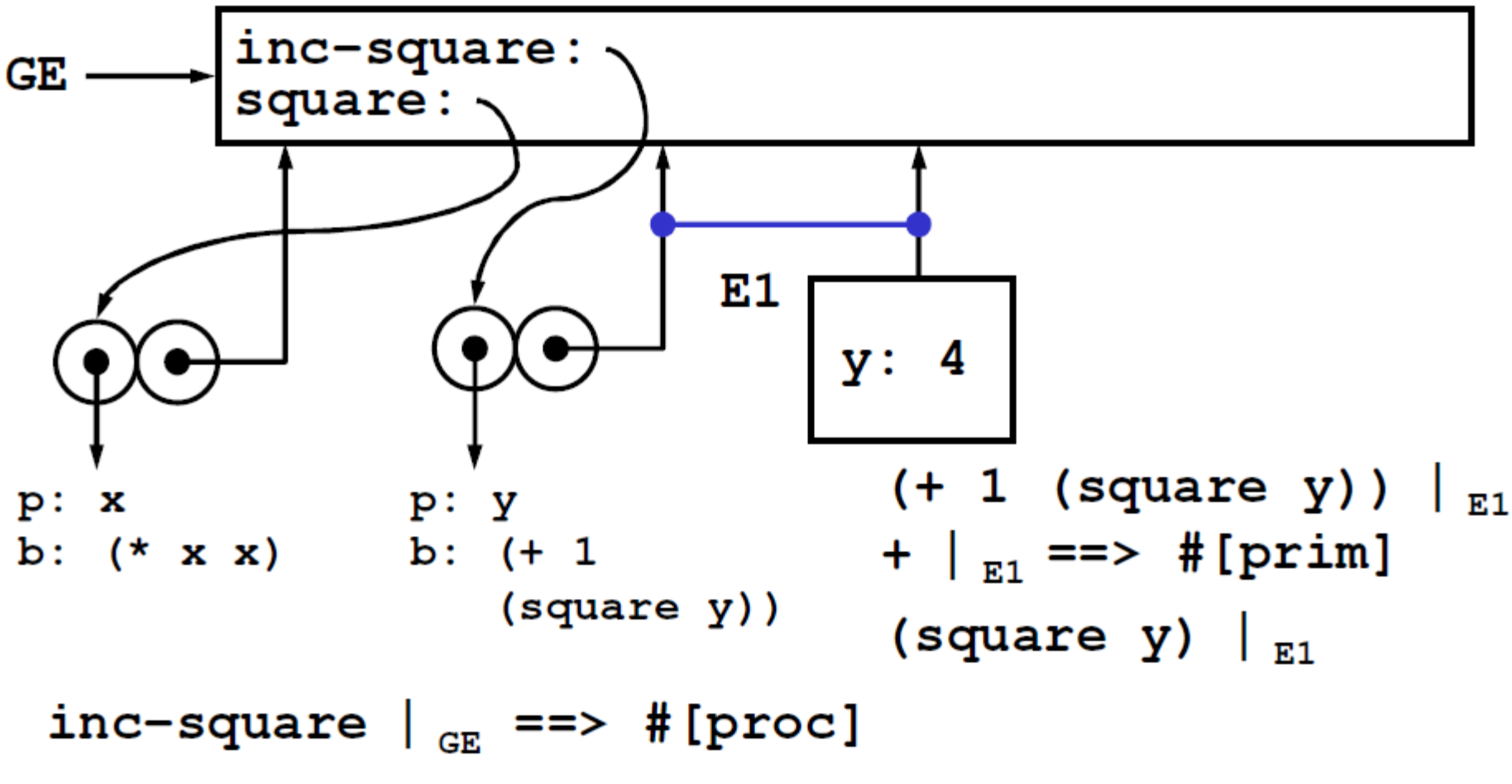
Ещё пример

```
(define square (lambda (x) (* x x))) | GE
(define inc-square
        (lambda (y) (+ 1 (square y))) | GE
```



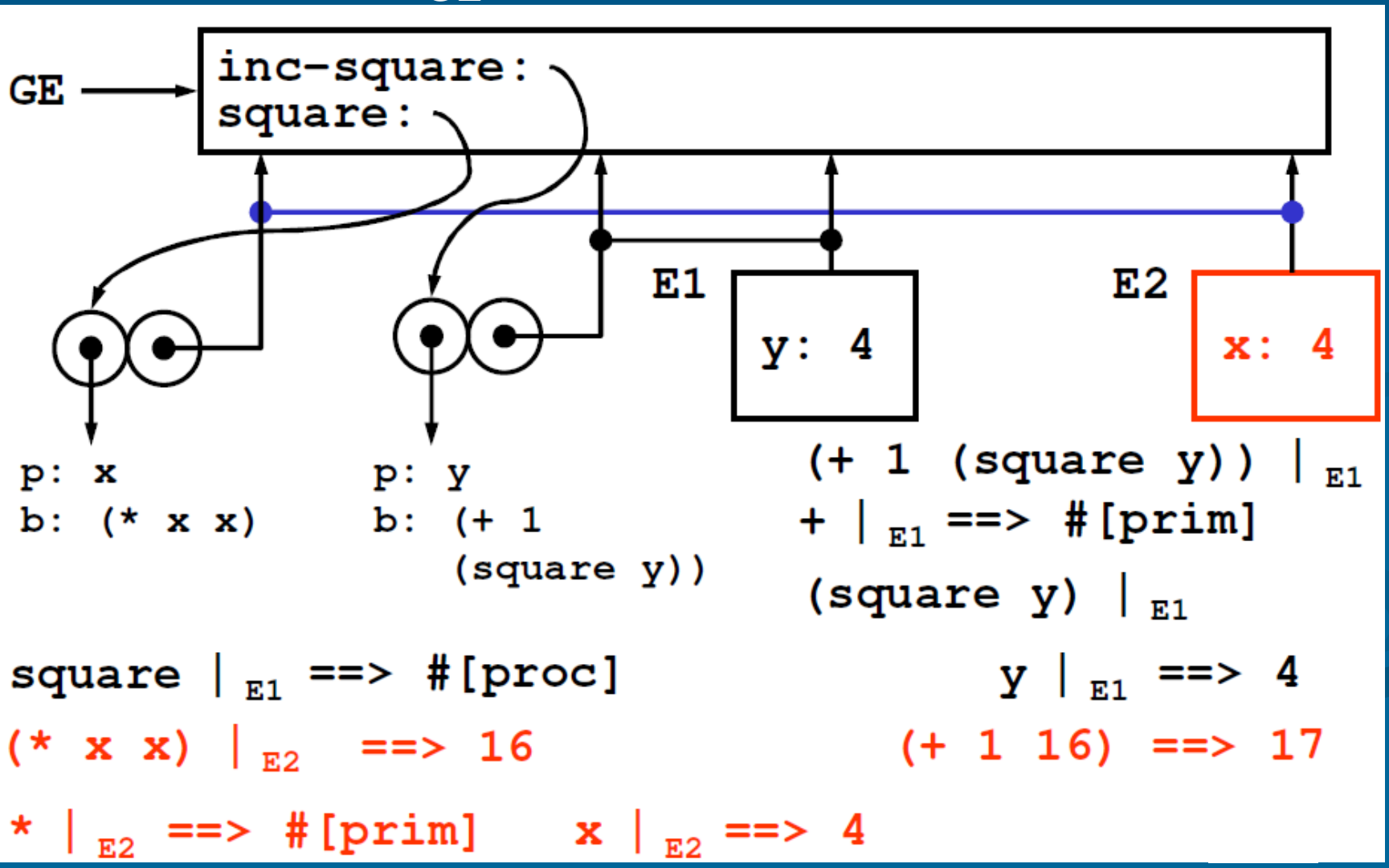
Продолжение примера

(inc-square 4) $|_{GE}$



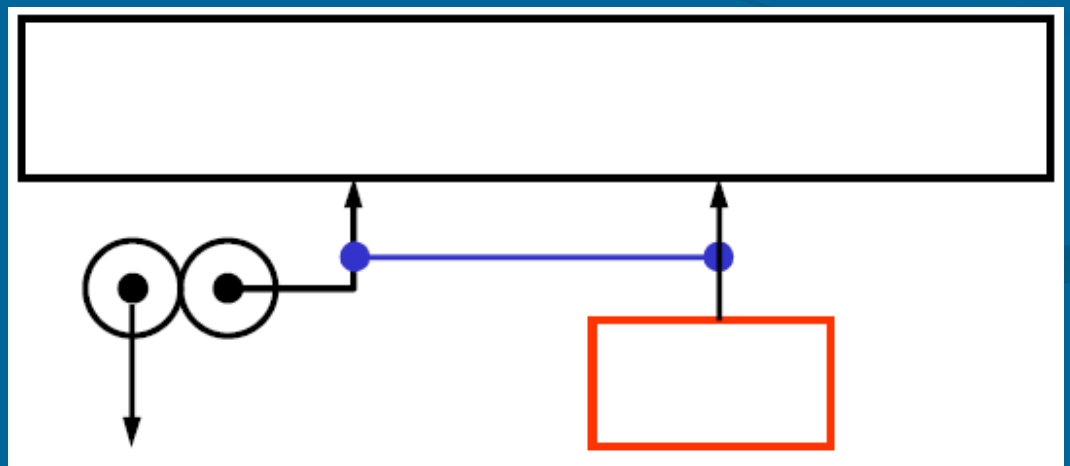
Продолжение примера

(inc-square 4) $|_{GE}$



Итоги примера

- МВО не описывает полностью работу интерпретатора, но позволяет находить те же ответы, какие находит интерпретатор.
- Стандартные связывания из глобального окружения (*, cons, ...) рисовать не надо.
- Во всех созданных кадрах ссылка на объемлющее окружение указывала на глобальное окружение, так как inc-square и square описаны в нём.
- Полезно запомнить шаблон:

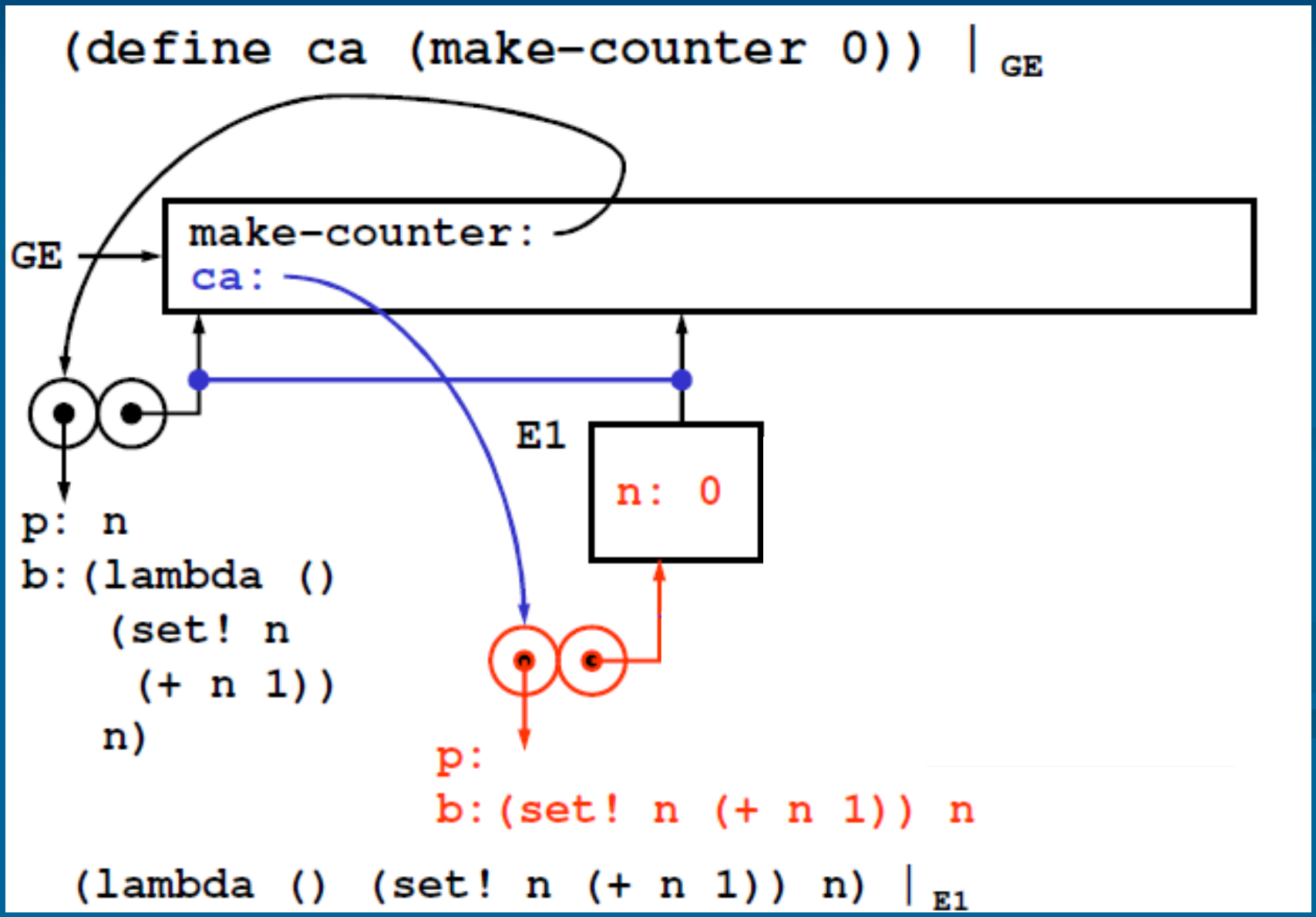


Снова пример

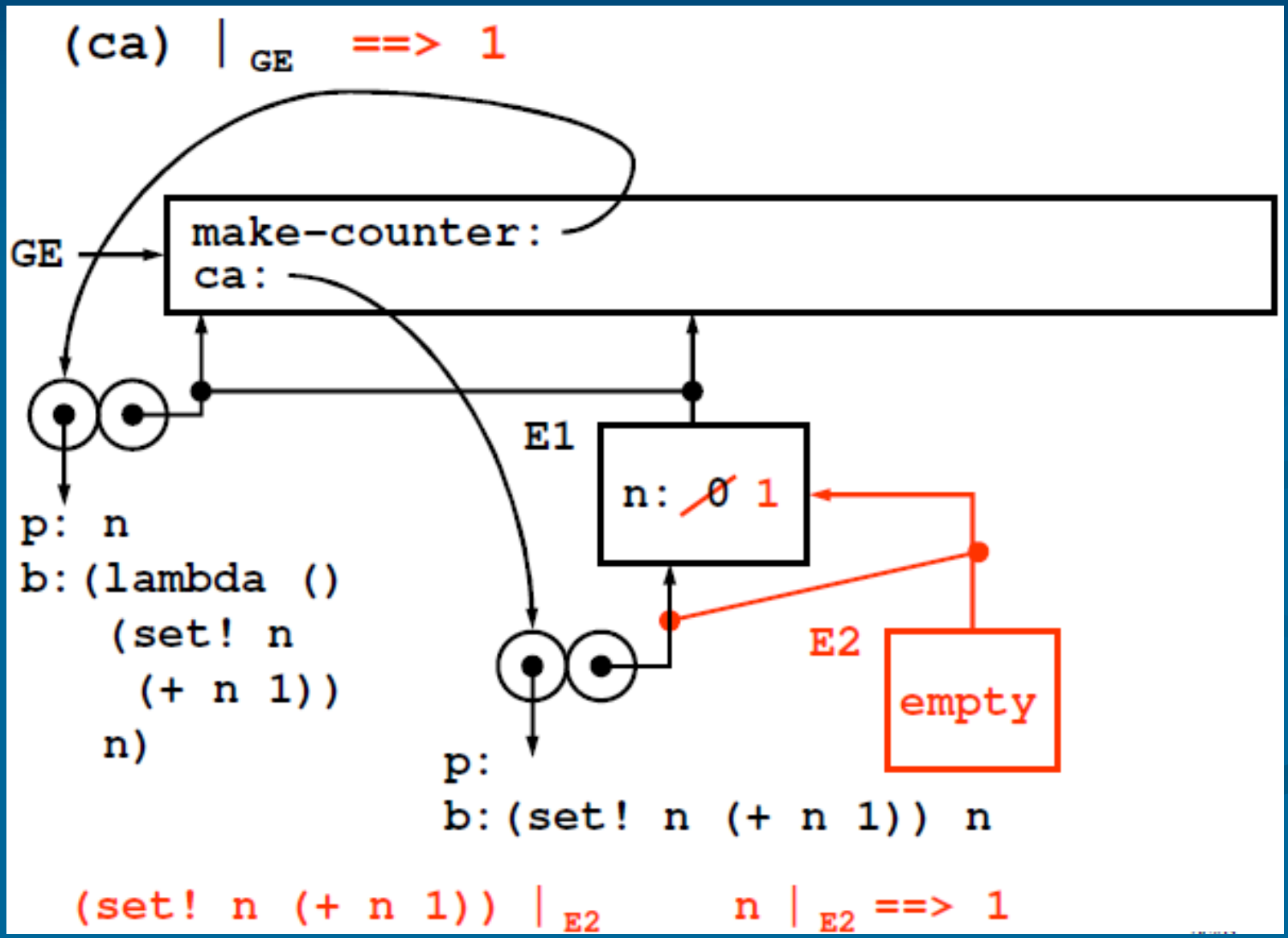
- Счётчик

```
(define (make-counter n)
        (lambda () (set! n (+ n 1)) n))
(define ca (make-counter 0))
(ca) ==> 1
(ca) ==> 2
(define cb (make-counter 0))
(cb) ==> 1
(ca) ==> 3
(cb) ==> 2 ; ca и cb независимы
```

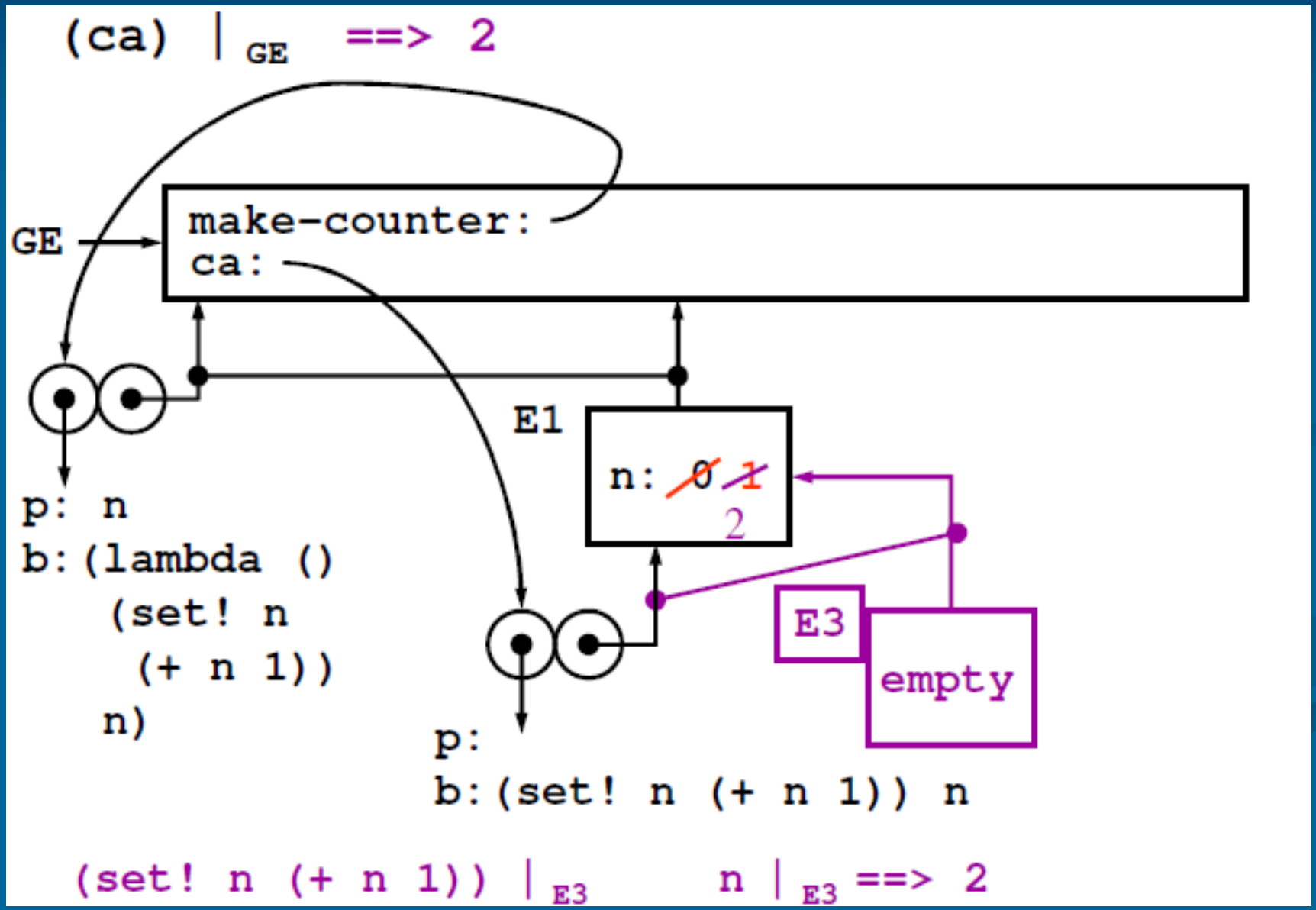
Счётчик. Продолжение



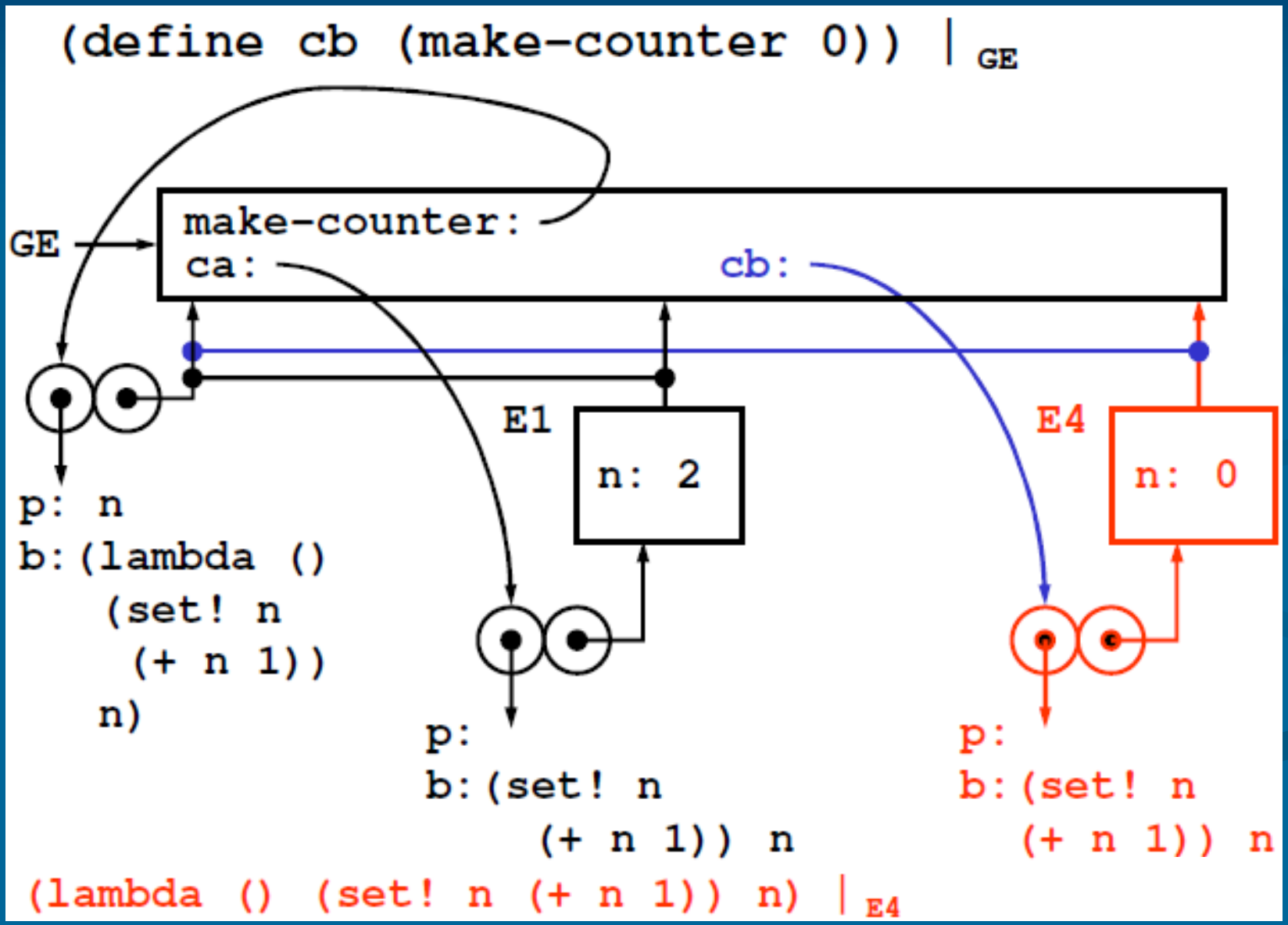
Счетчик. Продолжение



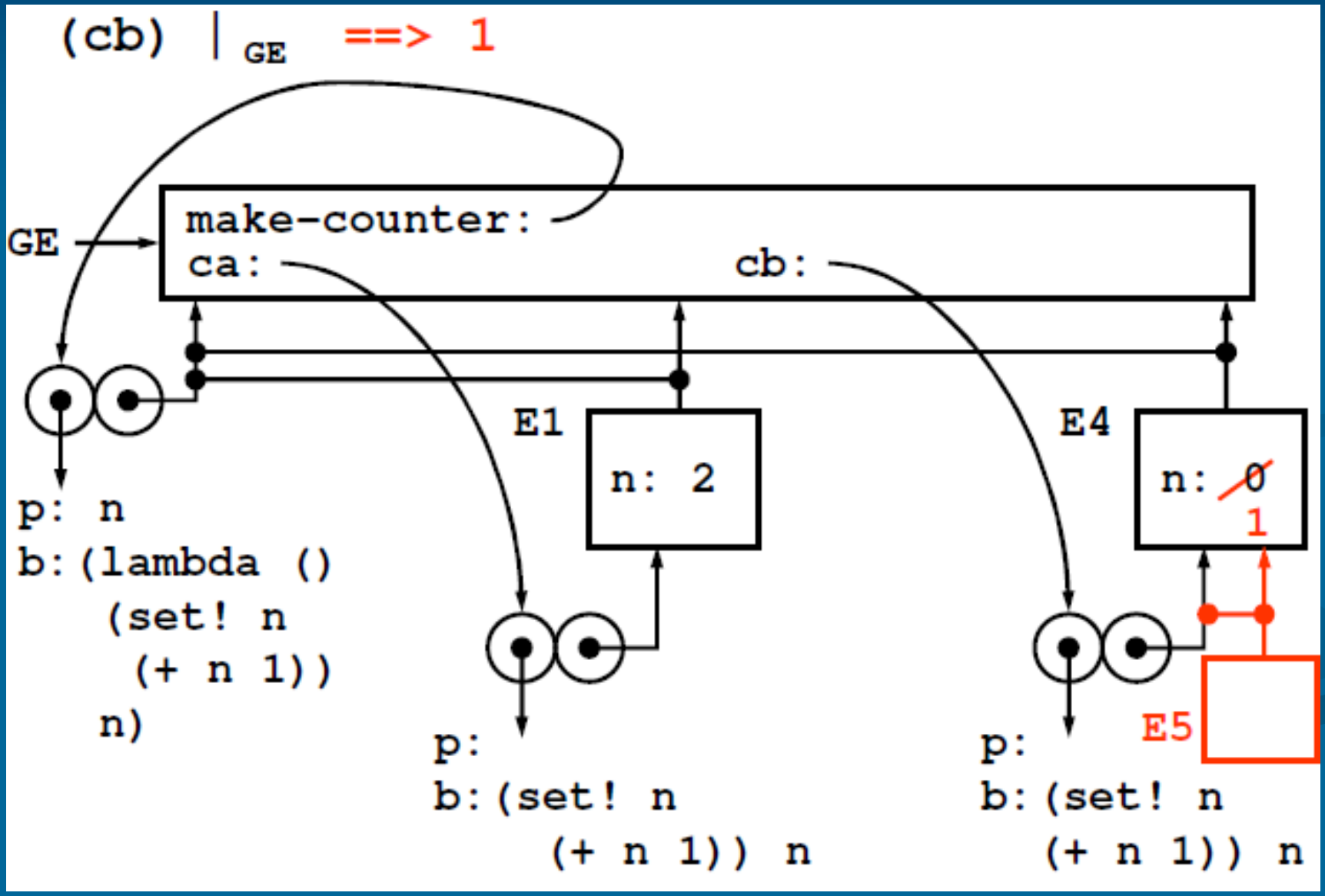
Счетчик. Продолжение



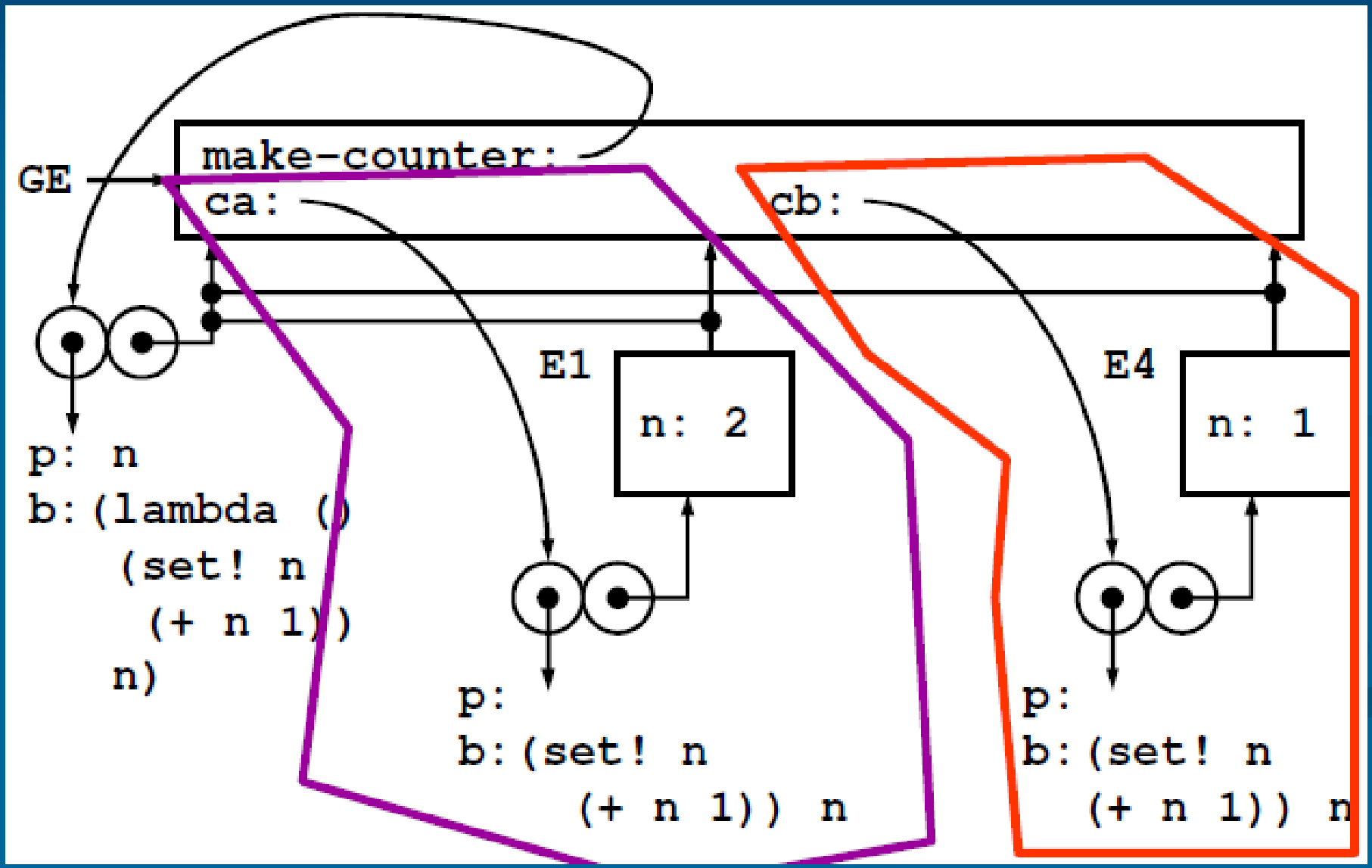
Счетчик. Продолжение



Счетчик. Продолжение



Счетчик. Продолжение



Таблица

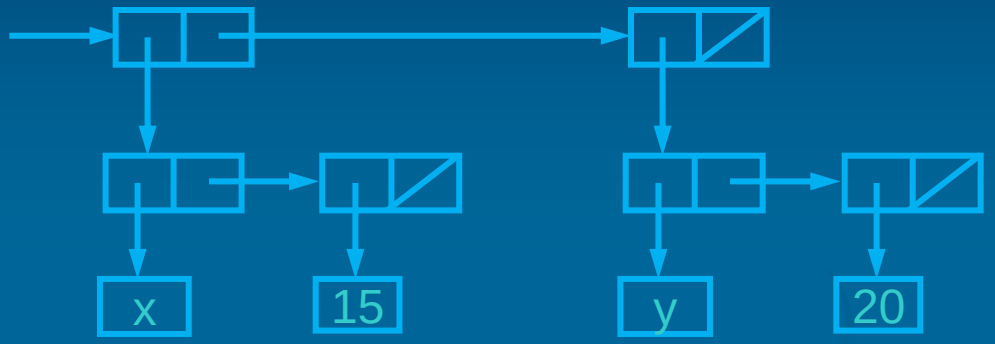
- конструктор (make-table)
- селектор (lookup <table> <key>)
- мутатор (insert! <table> <key> <value>)
- Список пар: (*table* (key1 . val1) (key2 . val2)…)

```
(define (make-table)(mlist '*table*))
(define (lookup table key)
    (let ((rec (massoc key (mcdr table))))
          (if (mpair? rec) (mcdr rec) #f)))
(define (insert! table key value)
     (let ((rec (massoc key (mcdr table))))
        (if (mpair? rec) (set-mcdr! rec value)
         (set-mcdr! table (mcons (mcons key value)
                                 (mcdr table))))))
```

Assoc? Ассоциативные списки

- списки пар, в которых первый элемент используется для поиска

‘((x 15) (y 20))



- поиск пары по equal? (assoc <key> <alist>)

(assoc ‘x ‘((w 1) (x 2) (y 3) (z 4))) ==> (x 2)

- искать по eq? (assq <key> <alist>)

(assq ‘x ‘((w 1) (x 2) (y 3) (z 4))) ==> (x 2)

(assq ‘(x 1) ‘(((x 1) 2) ((y 1) 3)))) ==> #f

(assoc ‘(x 1) ‘(((x 1) 2) ((y 1) 3)))) ==> ((x 1) 2)

- искать по своему сравнению (assoc <key> <alist> <fun>)
- искать по предикату (assf <pred> <alist>)

Таблица

(define table (make-table))

(insert! table ‘c 1)

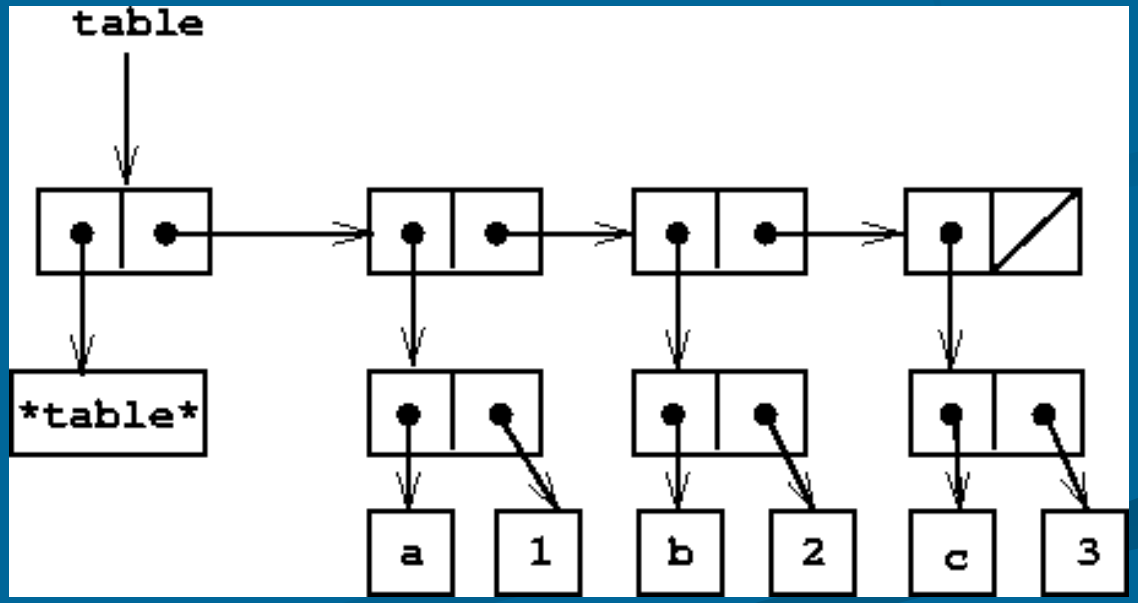
(insert! table ‘b 2)

(insert! table ‘a 1)

(insert! table ‘c 3)

(lookup table ‘c) ==> 3

(lookup table ‘d) ==> #f



Мемоизация (табуляризация)

- Идея: функция будет запоминать вычисленные результаты.
- Перед тем как вычислить функция проверит, нет ли результата среди запомненных.
- Хранить можно в таблице.
- (memo-fib 2) ==> 1 и запомнить пару (2 1)

```
(define fib-table (make-table))
(define (memo-fib n)
   (cond ((= n 0) 0) ((= n 1) 1)
    (else (let ((res (lookup fib-table n)))
       (if res res
           (let ((val (+ (memo-fib (- n 1)) (memo-fib (- n 2)))))
             (insert! fib-table n val) val))))))
```

Мемоизация (табуляризация)

```
> (memo-fib 7) ==> 13
> (display fib-table)
(*table* (7 . 13) (6 . 8) (5 . 5) (4 . 3) (3 . 2) (2 . 1))
> (memo-fib 11) ==> 89
> (display fib-table)
(*table* (11 . 89) (10 . 55) (9 . 34) (8 . 21) (7 . 13) (6 . 8)
    (5 . 5) (4 . 3) (3 . 2) (2 . 1))
```

Если вычисляем для n, которое было – O(n)

Если вычисляем для n, которого не было – $O(n^2)$

Мемоизация (табуляризация)

- Нужно ли для каждой функции заводить явно свою таблицу и писать в теле lookup, insert!?

```
(define (memoize func)
    (let ((table (make-table)))
          (lambda (x)
             (let ((prev-result (lookup table x)))
                (if prev-result prev-result
                    (let ((result (func x)))
                                 (insert! table x result)
                                 result))))))
(define memo-fib2 (memoize (lambda (n)
    (cond ((= n 0) 0) ((= n 1) 1)
          (else (+ (memo-fib2 (- n 1)) (memo-fib2 (- n 2)))))))))
```

Векторы

- Вектор – массив (фиксированного размера коллекция значений с доступом по индексу)
- конструктор (make-vector <size> <value>)
- селектор (vector-ref <vector> <index>)
- мутатор (vector-set! <vector> <index> <value>)
- длина (vector-length <vector>)
- внешнее представление ‘#(val_0 val_1 val_2 … val_N) или (vector val_0 val_1 val_2 … val_N)

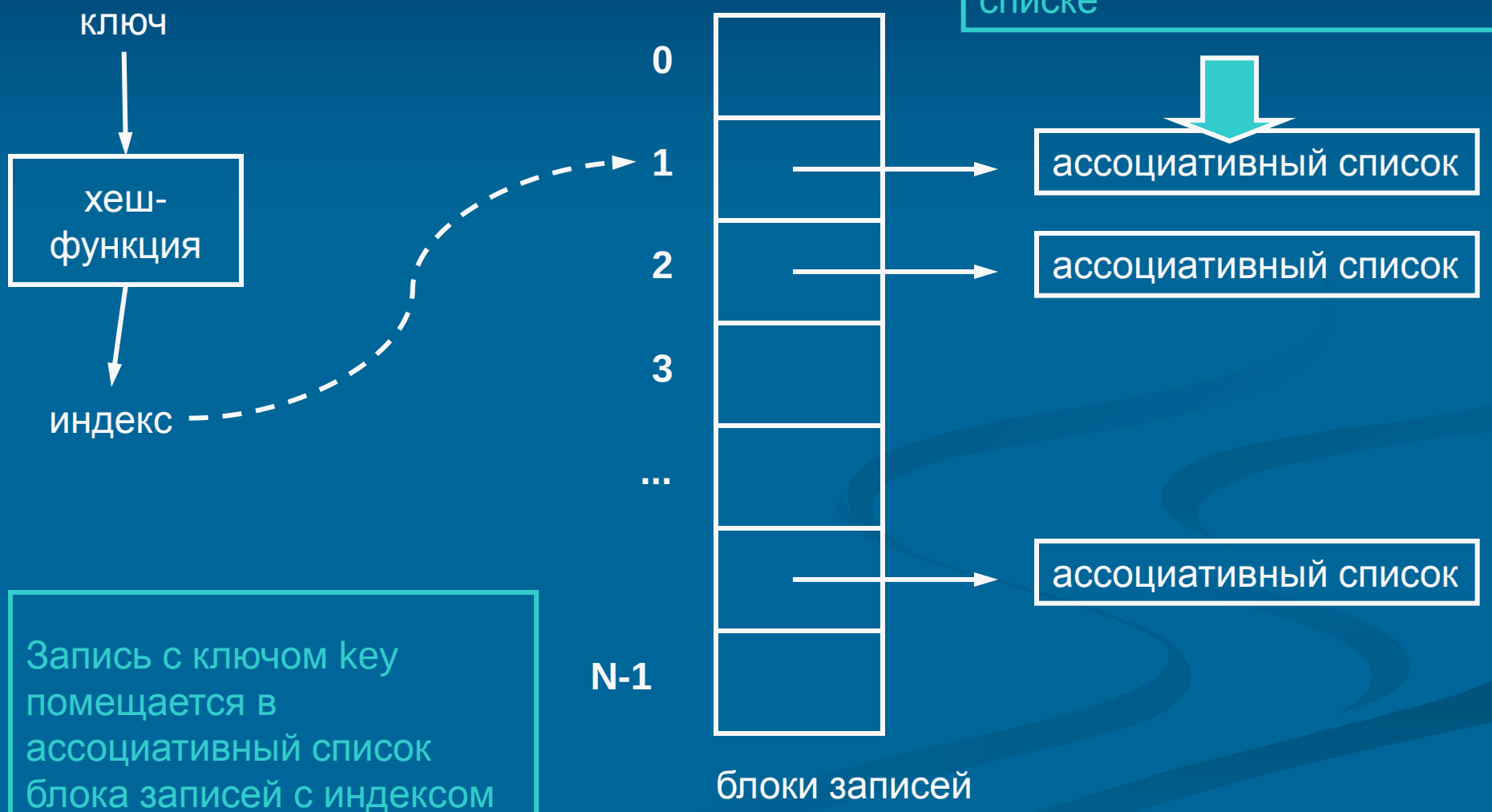
```
> (define beatles (vector 'john 'paul 'george 'pete))
> (vector-set! beatles 3 'ringo)
> beatles ==> ‘#(john paul george ringo)
> (vector-length beatles) ==> 4
```

Векторы

- преобразование (list->vector <list>)
- преобразование (vector->list <vector>)
- если скрестить таблицы и векторы, можно получить хеш-таблицу

- конструктор (make-hash-table <size> <hash-func>)
- селектор (lookup-hash-table <table> <key>)
- мутатор (insert-hash-table! <table> <key> <value>)

Хеш-таблица



Запись с ключом key помещается в ассоциативный список блока записей с индексом index = hash(key)

Реализация хеш-таблиц

```
(define (make-hash-table size hashfunc)
  (let ((buckets (make-vector size)))
    (define (helper n) (if (< n size) (begin (vector-set! buckets n
     (make-table)) (helper (+ n 1))) #t))
    (helper 0) (list '*hash-table* size hashfunc buckets)))
(define (lookup-hash-table tbl key)
 (let ((index ((caddr tbl) key (cadr tbl))))
  (lookup (vector-ref (cadddr tbl) index) key)))
(define (insert-hash-table! tbl key val)
  (let ((index ((caddr tbl) key (cadr tbl)))
        (buckets (cadddr tbl)))
      (insert! (vector-ref buckets index) key val)))
```

Реализация хеш-таблиц

```
> (define t7 (make-hash-table 7 (lambda (x y) (remainder x y))))
> (insert-hash-table! t7 2 'a)
> (insert-hash-table! t7 1 'ab)
> (insert-hash-table! t7 100 'abc)
> (display t7)
(*hash-table* 7 #<procedure> #((*table*) (*table* (1 . ab))
    (*table* (100 . abc) (2 . a)) (*table*) (*table*) (*table*)
    (*table*)))
> (lookup-hash-table t7 9) ==> #f
> (lookup-hash-table t7 100) ==> 'abc
> (lookup-hash-table t7 1) ==> 'ab
> (lookup-hash-table t7 2) ==> 'a
```

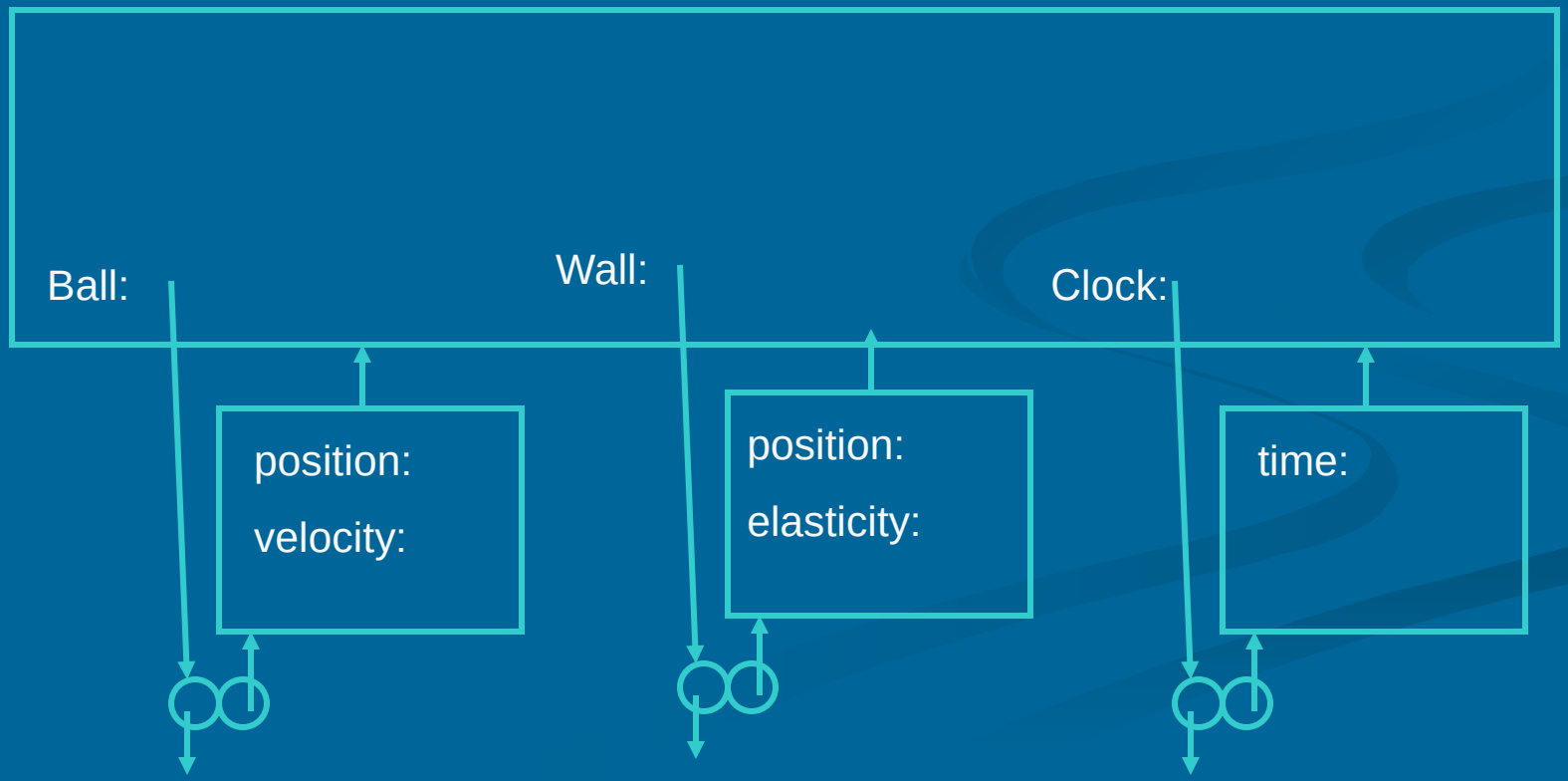
Итоги лекции 6

- Присваивание даёт дополнительные возможности.
- Присваивание создаёт дополнительные трудности:
  - требует новую модель описания вычислений вместо подстановочной модели;
  - приводит к возникновению сторонних эффектов;
  - порядок вычислений начинает играть роль.

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

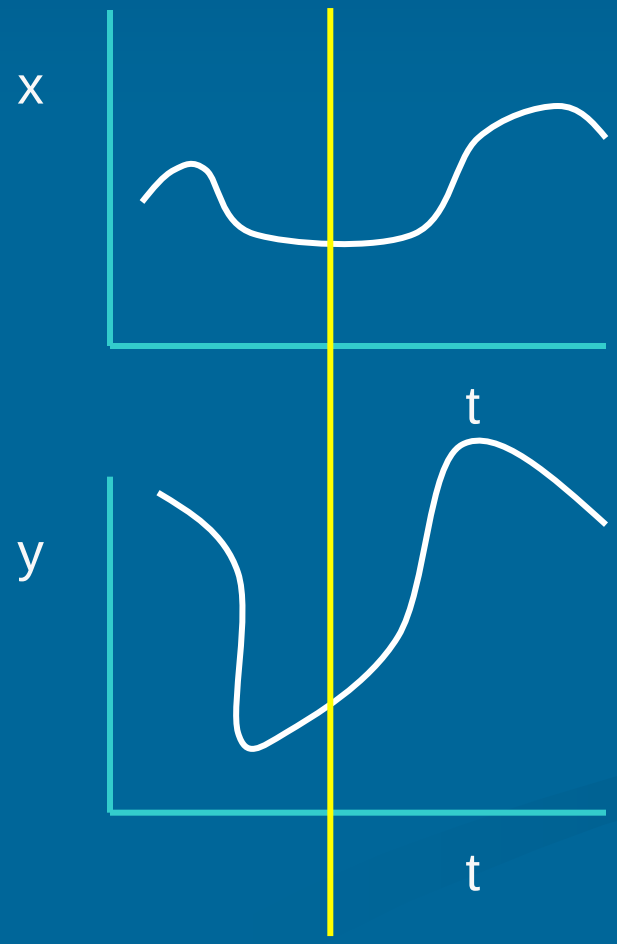
Clock:
1
2
3
4
5
…

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

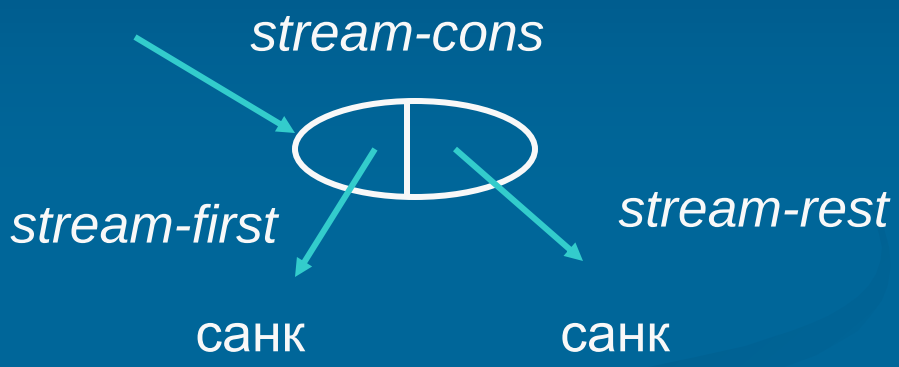
(stream-rest str) -- возвращает хвост потока

- 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)

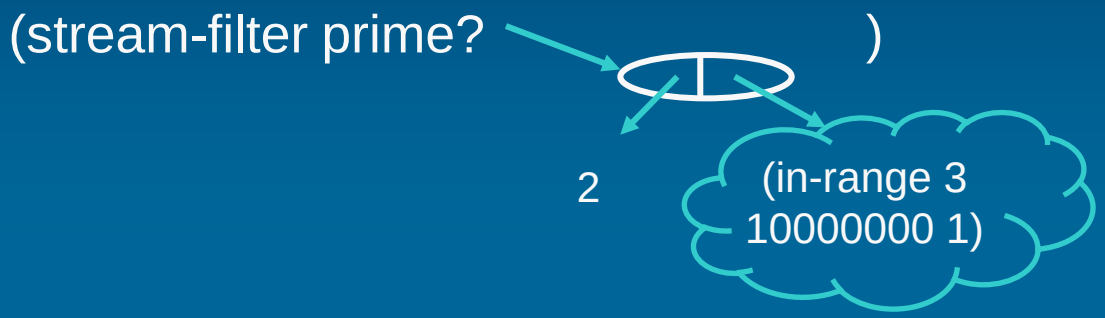
```
(stream-ref
  (stream-filter (lambda (x) (prime? x))
                 (in-range 2 100000000 1))
  100)
```

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))))
```

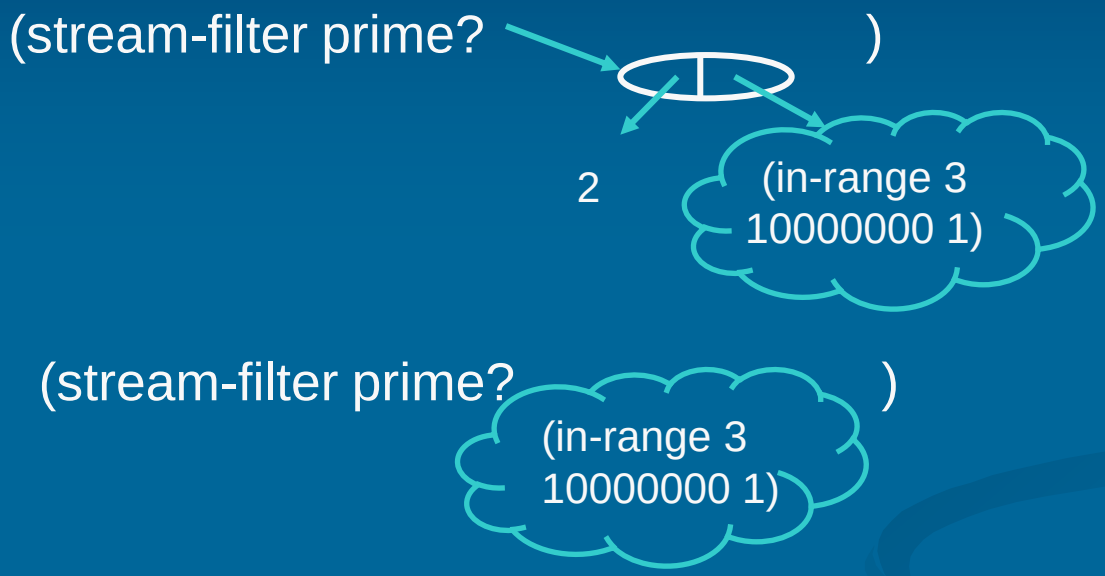
Вычисления в примере

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



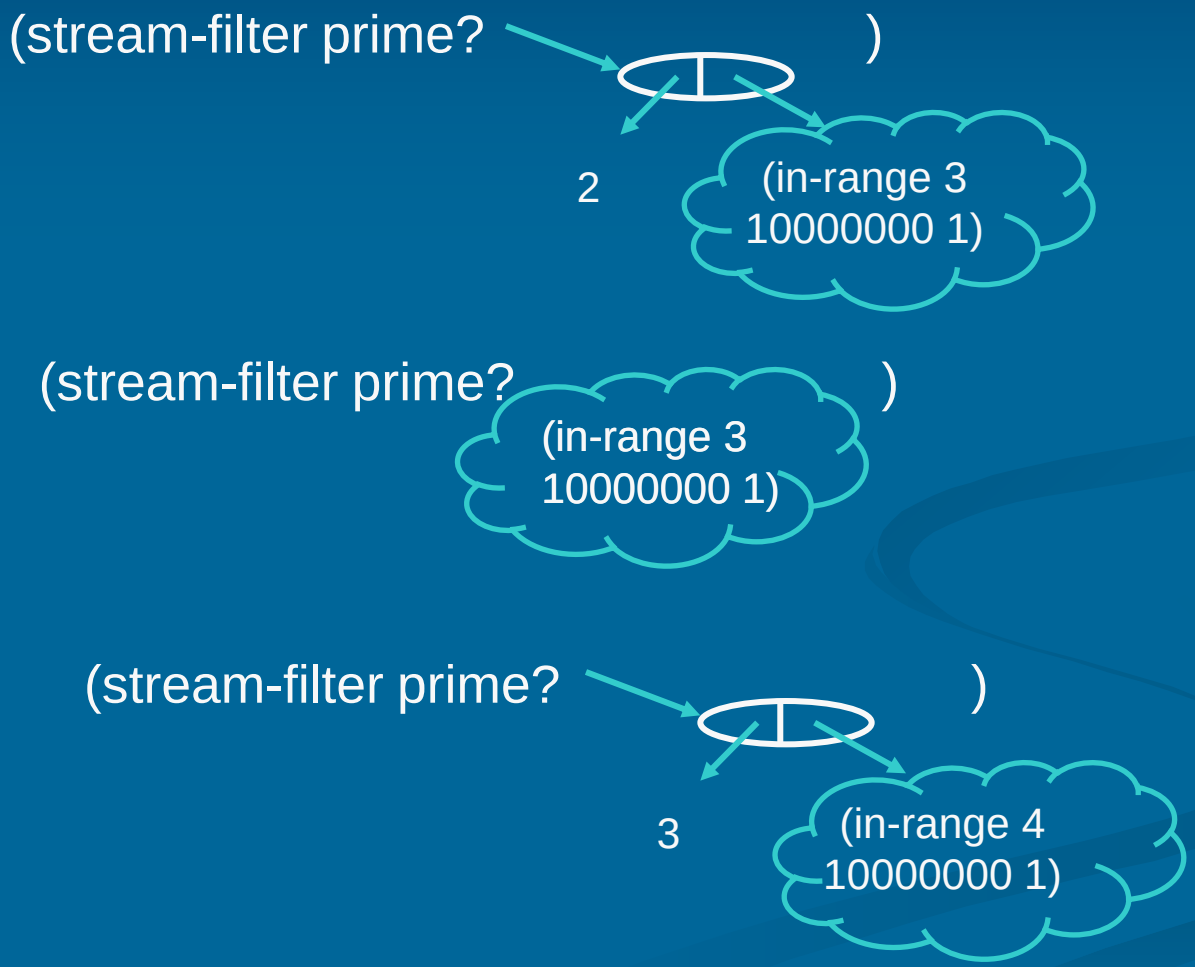
Вычисления в примере

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



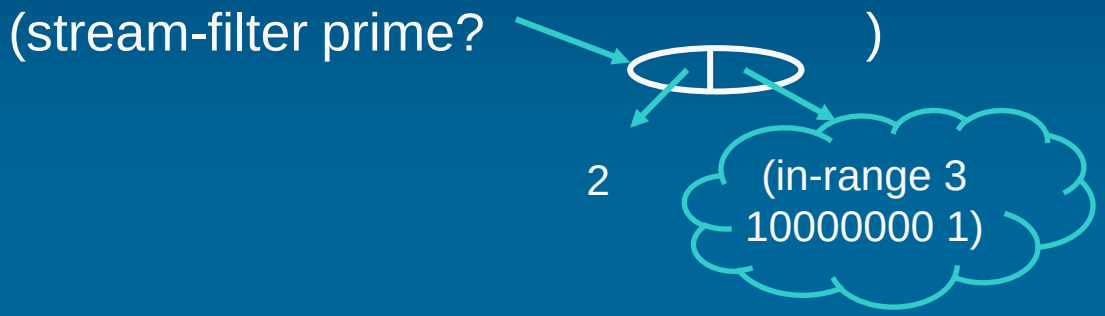
Вычисления в примере

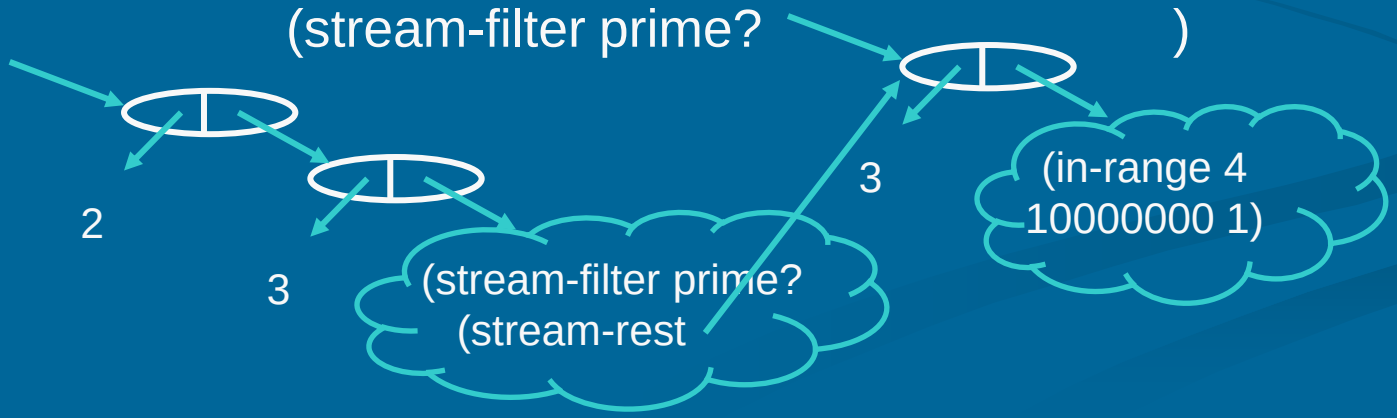
(stream-filter prime? (in-range 2 100000000 1))



Вычисления в примере

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



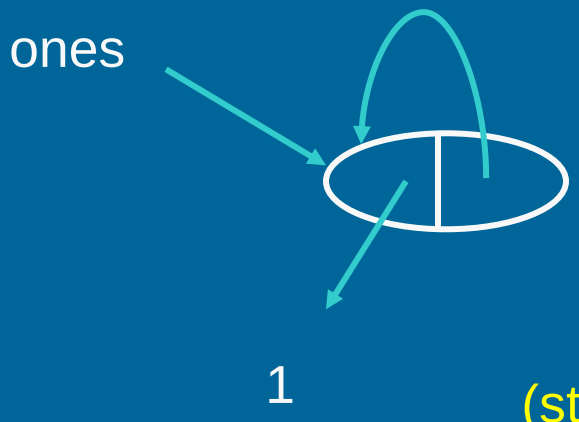


Ещё одна возможность: бесконечная структура данных!

- 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:

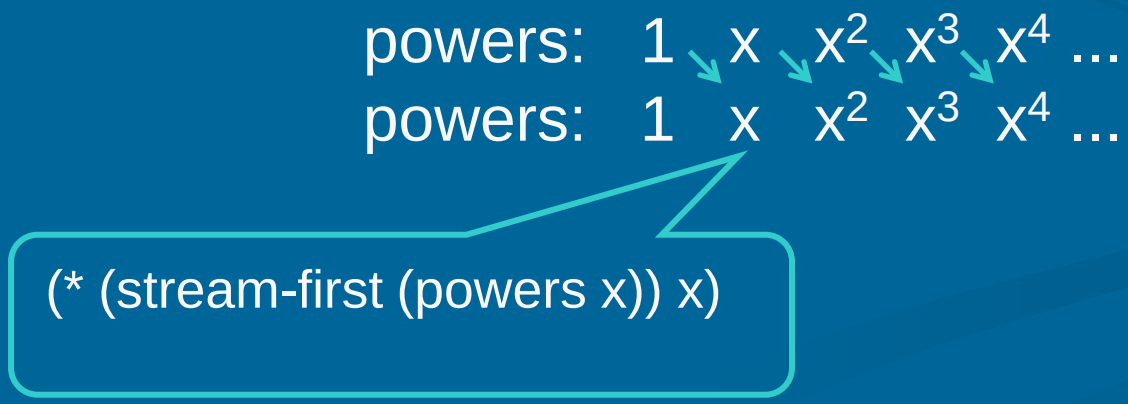
```
(define (fibgen a b) ; порождающая функция
  (stream-cons a (fib-gen (b (+ a b))))
(define fibs (fibgen 0 1))
```

Ещё одна функция для неявного порождения

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

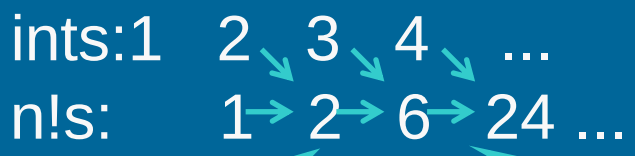
неявное описание потока степеней (1 x x^2 x^3 ...)

```
(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 ...
```

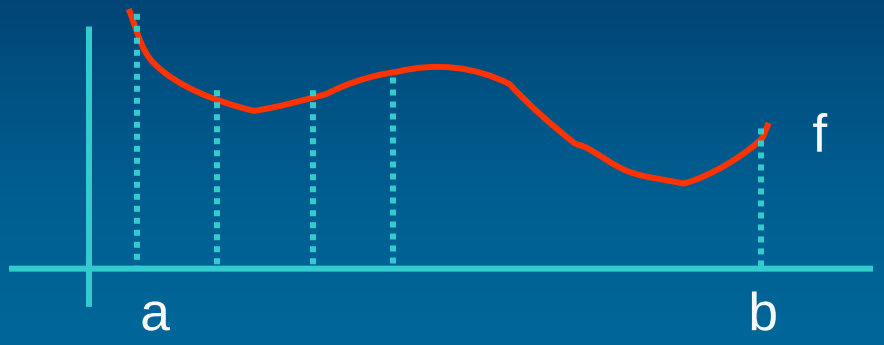
Потоковый поиск квадратного корня

- 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

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

```
(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.

ЛЕКЦИЯ 9
Макросы

Пример макроса. Swap

- чтобы поменять местами значения переменных a и b нужно:

```
(let ((c b))
   (set! b a)
   (set! a c))
```

- чтобы не писать подобный фрагмент каждый раз, определим макрос swap:

```
(define-syntax swap
   (syntax-rules ()
         ((swap a b)
          (let ((c b))
             (set! b a)
             (set! a c))
         )))
```

Пример макроса. Swap

- как работает swap?

> (define first 5)

> (define second 'a)

> (swap first second)

> first ==> 'a

> second ==> 5

> c ==> ошибка!

- при подстановке c заменяется на временное имя:

> (define c 10)

> (swap first c)

> first ==> 10

> c ==> 'a

- работает!

Пример макроса. Swap

- возможный результат подстановки (swap first c):

```
(let ((__generated_symbol_1 c))
  (set! c first)
  (set! first __generated_symbol_1))
```

- описывая макрос, мы указали имя:

```
(define-syntax swap
```

- указали тип применяемого преобразования:

```
(syntax-rules ()
```

- указали правило, содержащее образец:

```
((swap a b)
```

- и шаблон:

```
(let ((c b)) (set! b a) (set! a c))
        )))
```

Swap c define

- перепишем swap без let, чтобы убедиться, что гигиену обеспечивает syntax-rules, а не let:

```
(define-syntax swap2
    (syntax-rules ()
          ((swap2 a b)
           (begin (define c b)
              (set! b a)
              (set! a c))
          )))
> (define c 5)
> (define x 1)
> (define y 2)
> (swap2 x y)
> (swap2 x c)
> (display (list x y c)) ==> (5 1 2)
```

Итог примера

- Макрос это:
  - специальным образом описанная трансформация кода
  - трансформация, осуществляемая без вычисления кода
  - трансформация, не использующая данные времени выполнения
- Другой пример cond-set!

```
(cond-set! (> test 4) var 15)
```

- при подстановке даёт:

```
(cond ((> test 4) (set! var 15)))
```

- не вычисляет свои аргументы!

Макроподстановка

- Приблизительно подстановка происходит так
  1) сопоставитель находит вхождение имени макроса в тексте
  2) сопоставитель перебирает образцы, описанные в макросе
  3) если удаётся сопоставить с образцом, в шаблон подставляются параметры макровызова
  4) конфликтующие имена из шаблона заменяются на сгенерированные символы
  5) получившийся код вставляется на место макровызова

Когда уместно использовать макрос?

- Используйте макросы для изменения порядка вычислений (определения собственных спецформ). Например:
  1) условные вычисления (cond, case, and, or)
  2) циклы (do)
  3) связывания (let, let*)
  4) не вычисляемые имена (=>)

Когда неуместно использовать макрос?

- Всегда, когда без него можно обойтись!
  1) Макросы в отличие от функций не являются объектами первого класса.
  2) Макросы затрудняют отладку.

cond-set!

- Вернёмся к примеру:

```
(cond-set! (> test 4) var 15)
```

- Можно ли описать cond-set! функцией?

```
(define (cond-set! test variable value)
        (cond (test (set! variable value))))
```

- не работает!

```
> (define a 5)
> (define b 10)
> (cond-set! (< a b) a b)
> a ==> 5
```

- меняется значение локальной переменной функции!
- Макрос уместен.

Специальные формы и макросы

- Примитивных (настоящих) спецформ мало:
  - lambda
  - if
  - quote
  - set!

- Все остальные можно описать макросами.
- Хотя в трансляторе они могут быть реализованы напрямую.

Специальные формы и макросы

- and как макрос:

```
(define-syntax and
  (syntax-rules ()
        ((and) #t)
        ((and test) test)
        ((and test1 test2 ...)
              (if test1 (and test2 ...) #f))))
```

Системы макросов в Scheme

- Разные реализации Scheme поддерживают разные системы макросов
  - defmacro (унаследована от Лиспа)
  - syntax-rules (стандартная R5RS, «гигиеничная»)
  - syntax-case
  - syntax-table
  - …
- Рассмотрим syntax-rules (она есть в Racket!)

Характеристики системы syntax-rules

- Гигиена. Макрос не портит окружения, в которых происходит подстановка.
- Прозрачность ссылок. Окружение, в котором происходит подстановка не портит макрос.
- Язык образцов используется для описания структуры макрокоманд.
- Закрытость. Макросистема отделена от Scheme. Во время подстановки нельзя запустить какой-либо код.

Гигиеничные макросы

- Все локальные переменные макроса автоматически переименовываются перед подстановкой, для того чтобы предотвратить конфликт имен.
- Пример: swap. Локальное имя c не конфликтует с c из окружения, в котором происходит подстановка.
- Гигиеничность означает, что вызов макроса не может неожиданно повлиять на связывания. Ожидаемые изменения возможны:

```
(define-syntax shadow
   (syntax-rules () ((shadow a b)
                     (begin (define a 5) b))))
> (define test 7)
> (shadow test test) ==> (begin (define test 5) test) ==> 5
> test ==> 5      ожидаемое изменение с таким вызовом
```

Прозрачность ссылок

- Все свободные переменные из шаблона макроса имеют связывания из окружения, в котором описан макрос.
- Пример:

```
(define-syntax dec
   (syntax-rules () ((dec arg)
                     (set! arg (- arg 1)))))
> (define a 1)
> (define b 1)
> (let ((- +)) (dec a) (set! b (- b 1)))
> a ==> 0  минус «старый»!
> b ==> 2  минус «новый»!
```

Язык образцов

- Язык образцов в syntax-rule прост:
  - образец – это комбинация (список)
  - первый элемент – имя макроса
  - литералы, а также списки, векторы представляют сами себя
  - бывают спецсимволы и многоточия
  - остальное – переменные образца.
- Образец сопоставится:
  - если литералы, списки, векторы совпадают точно
  - если каждая переменная образца сопоставима одному подвыражению макрокоманды.

Язык образцов. Пример

- Дан образец:

```
(let1 (name value) body)
```

- Макрокоманда:

```
(let1    (x (read))
         (cond ((not x) (display "you said no"))))
```

- Образец сопоставится:
  - name = x
  - value = (read)
  - body = (cond ((not x) (display "you said no")))

Язык образцов. Ещё пример

- Дан образец:

```
(contrived #((first . rest) #(3 any)))
```

- Макрокоманда:

```
(contrived #((1 2 3 4 5) #(3 ’(foo))))
```

- Образец сопоставится:
  - first = 1
  - rest = (2 3 4 5)
  - any = ’(foo)

Язык шаблонов

- Шаблон – это код на Scheme, определяющий вместе с образцом, что будет подставлено.
  - литералы, а также списки, векторы представляют сами себя
  - символы, не встречающиеся в образце, представляют сами себя
  - остальное – переменные образца.
- При подстановке происходит замена внутри шаблона переменных образца, на то, с чем они сопоставились.

Язык шаблонов. Пример

- Образец:

```
(let1 (name value) body)
```

- Шаблон:

```
(let ((name value)) body)
```

- Макрокоманда:

```
(let1   (x (read))
        (cond ((not x) (display "you said no"))))
```

- Подстановка:

```
(let    ((x (read)))
        (cond ((not x) (display "you said no"))))
```

Многоточие

- Если за переменной образца следует … она сопоставляется с несколькими подвыражениями макрокоманды.
- Пример образца:

```
(dotimes count body ...)
```

- Пример макрокоманды:

```
(dotimes 5 (set! x (+ x 1)) (display x))
```

- Сопоставление:

```
count = 5
body ... = (set! x (+ x 1)) (display x)
```

Многоточие. Пример

- В шаблоне вхождение переменной образца с многоточием заменяется на все сопоставленные ей подвыражения
- Пример:

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (cond ((> counter 0) (begin body ...
         (loop (- counter 1)))))))))
```

```
> (define x 5)
> (dotimes 5 (set! x (+ x 1)) (display x))
```

- в результате код, печатающий 678910

Многоточие. Пример

- В шаблоне вхождение переменной образца с многоточием заменяется на все сопоставленные ей подвыражения
- в результате код, печатающий 678910

```
(let loop ((counter 5))
  (cond ((> counter 0)
    (begin (set! x (+ x 1)) (display x)
      (loop (- counter 1))))))
```

Многоточие. Пример

- Многоточие в шаблоне после подвыражения с переменной образца с многоточием вызывает многократную подстановку подвыражения с каждым сопоставлением переменной.
- Пример:

```
(define-syntax thunkify
  (syntax-rules ()
        ((thunkify body ...)
         (list (lambda () body) ...))))
```

- Макрокоманда:

```
(thunkify 5 (* x x))
```

- Постановка

```
(list (lambda () 5) (lambda () (* x x)))
```

Многоточие. Ещё пример

```
(define-syntax update-if-true!
   (syntax-rules ()
         ((update-if-true! (condition variable) ...)
          (begin (let ((test condition))
                   (cond (test (set! variable test)))) ...))))
```

- Макрокоманда:

```
(update-if-true! ((> x 5) x-is-big) ((zero? y) y-is-zero))
```

- Постановка

```
(begin
    (let ((test (> x 5)))
     (cond (test (set! x-is-big test))))
    (let ((test (zero? y)))
     (cond (test (set! y-is-zero test)))))
```

Многоточия. Пример

```
(define-syntax quoted-append
   (syntax-rules ()
         ((quoted-append (arg ...) ...)
          (quote (arg ... ...)))))
```

- Макрокоманда:

```
(quoted-append (1 2 3) (a b c) (+ x y))
```

- Постановка

```
’(1 2 3 a b c + x y)
```

Группировка пар образец-шаблон

- Одно макроопределение может содержать несколько описаний пар образец-шаблон:

```
(define-syntax and
  (syntax-rules ()
        ((and) #t)
        ((and test) test)
        ((and test1 test2 ...)
              (if test1 (and test2 ...) #f))))
```

пары просматриваются сверху вниз!

Спецсимволы в образцах

- Пусть мы ходим, чтобы макрокоманда:

```
(implications (a => b) (c =>d) (e =>f))
```

- давала подстановку:

```
(begin (cond (a b)) (cond (c d)) (cond (e f)))
```

- проблема в том, как указать, что => -- не переменная!

```
(syntax-rules ()
((implications (condition => consequent) ...)
(begin (cond (condition consequent)) ...)))
```

- приводит к тому, что при макровызове

```
(implications (test =. (set! testp #t)))
```

- получается сопоставление

```
condition = test        => = =.
consequent =  (set! testp #t)
```

Спецсимволы в образцах

- Спецсимволы следует указывать в списке после syntax-rules:

```
(syntax-rules (=>)
((implications (condition => consequent) ...)
(begin (cond (condition consequent)) ...)))
```

- теперь при макровызове

```
(implications (test =. (set! testp #t)))
```

- не будет сопоставления, так как спецсимвол сопоставляется только сам с собой
- другой случай, когда сопоставление невозможно:

```
(let ((=> 5)) (implications (foo => bar)))
```

=> разные!

Итоги по спецсимволам в образцах

- Спецсимволы из макроопределения относятся к окружению макроопределения.
- Символы из макрокоманды относятся к окружению, в котором встретилась команда.
- Спецсимвол сопоставится, только если он не перекрыт в окружении макрокоманды.

Итоги по syntax-rules

- Сочетание гигиеничности с прозрачностью ссылок делает макросы системы syntax-rules лексически безопасными.
  - макрос не портит неожиданно окружение, в котором происходит подстановка
  - окружение, в котором происходит подстановка, не портит макрос
  - действует «правило наименьшего удивления»

Вспомним dotimes

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (cond ((> counter 0) (begin body ...
         (loop (- counter 1))))))))))
```

- что если?

```
> (define counter 5)
> (dotimes 5 (set! counter (+ counter 1)) (display counter))
```

- в результате код, печатающий 678910
- макрос безопасный – counter’ы разные.

Рекомендации по стилю макроопределений

- сортируйте пары образец-шаблон (сначала короткие, как в and)
- имя макроса в образце можно не писать, а ставить прочерк (так проще переименовывать макросы)

```
(define-syntax dec
   (syntax-rules () (( _ arg)
                     (set! arg (- arg 1)))))
```

Итоги лекции 9

- Мы рассмотрели не всё. Бывают:
  - cps-style макросы
  - обход гигиеничности (Petrofsky’s find-identifier)
  - синтезирование символов
- Того, что рассмотрено, достаточно.

ЛЕКЦИЯ 10
λ-Исчисление. Теоретические основы функционального программирования

Лямбда-исчисление

λ-исчисление – это набор формальных систем, основанных на нотации, которую придумал Алонзо Черч (A. Church) в 1930 г.

Исчисление анонимных функций

Лямбда-исчисление

x−y можно рассматривать, как функцию f(x) и функцию g(y)

f(x) = x−y g(y) = x−y

f: x -> x−y g: y -> x−y

Нотация Черча:

f ≡ λx. x−y g ≡ λy. x−y

Настоящая нотация Чёрча

На самом деле Чёрч писал с «крышкой»: ^

ŷ. x−y

Есть версия, что при наборе его статьи в типографии не нашлось нужной литеры, поэтому напечатали так:

Λy. x−y

Дальше при перепечатке заменили большую лямбду на маленькую:

λy. x−y

Аппликация

Аппликация (apply, композиция):

$f \equiv \lambda x.\ x - y$

$f(0) = 0 - y$ в нотации Чёрча: $(\lambda x.\ x - y)\ 0 = 0 - y$

$f(1) = 1 - y$ в нотации Чёрча: $(\lambda x.\ x - y)\ 1 = 1 - y$

Каждая λ-функция – это функция от одного аргумента!

Аппликация

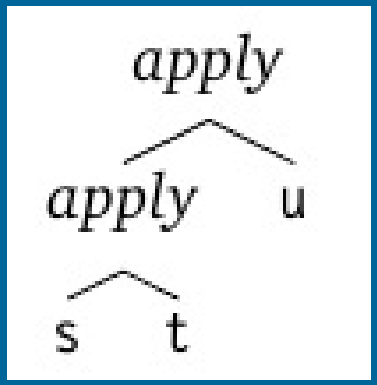
Функции от двух и более аргументов в λ-нотации:

f(x,y) = x−y ==> λx. λy. x−y

Аппликация левоассоциативная!

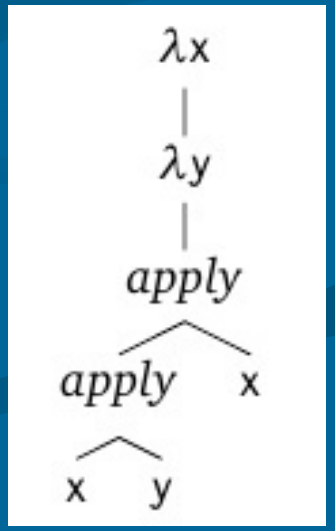
(λx. λy. x−y) 7 2 => (λy. 7−y) 2 ==> 7−2

s t u ≡ ((s t) u)



Аппликация «заберёт всё, до чего дотянется»

λx. λy. x y x ≡ λx. (λy. ((x y) x))



λ-нотация

Свободные переменные и связанные переменные

$$\lambda x.\ x - y$$

x – связанная переменная

y – свободная переменная

Формальное определение

λ-терм (λ-выражение) – это:

переменная (например, x);

константа (например, c1);

комбинация или аппликация s t функции s к аргументу t,
где s и t – λ-термы;

абстракция λx. s λ-терма s по переменной x

БНФ для λ-термов:

<λ-терм> ::= <переменная> | <константа> |
<λ-терм> <λ-терм> | λ <переменная>. <λ-терм>

Свободные и связанные переменные

Вхождение переменной x в λ-терм t является свободным, если оно лежит вне области действия соответствующей абстракции.

Формально FV(t) – множество свободных переменных терма t, BV(t) – связанных:

$FV(x) = \{x\}$	$BV(x) = \varnothing$
$FV(c1) = \varnothing$	$BV(c1) = \varnothing$
$FV(s\ t) = FV(s) \cup FV(t)$	$BV(s\ t) = BV(s) \cup BV(t)$
$FV(\lambda x.\ s) = FV(s) - \{x\}$	$BV(\lambda x.\ s) = BV(s) \cup \{x\}$

Пример: $s = (\lambda x.\ \lambda y.\ x)\ (\lambda x.\ z\ x)$

$FV(s) = \{z\}$	$BV(s) = \{x, y\}$

Подстановка

Подстановка $t[s/x]$ – это терм, получаемый из терма t заменой переменной x на терм s.

Пример: $(\lambda y.\ x+y)[5/x] = \lambda y.\ 5+y$

Ещё пример: $(\lambda y.\ x+y)[y/x] = \lambda y.\ y+y$??? $= \lambda w.\ y+w$!!!

При подстановке следует учитывать свободные/связанные переменные.

$x[t/x] = t$ $\quad y[t/x] = y$, если $x \neq y$ $\quad c[t/x] = c$

$(s\ u)[t/x] = s[t/x]\ u[t/x]$ $\quad (\lambda x.\ s)[t/x] = \lambda x.\ s$

$(\lambda y.\ s)[t/x] = \lambda y.\ (s[t/x])$, если $x \neq y$, либо $x \notin FV(s)$, либо $y \notin FV(t)$

$(\lambda y.\ s)[t/x] = \lambda z.\ (s[z/y][t/x])$, иначе, причём $z \notin FV(s) \cup FV(t)$

Преобразования λ-выражений

- α-редукция «переименование связанной переменной»
  - если v и w – переменные, а t – λ-терм, то

  λv. t -α-> λw. t[v/w], если w $\notin$ FV(t)
- Пример: λx. λy. x−y -α-> λx. λv. x−v

Преобразования λ-выражений

- β-редукция «вычисление функции для заданного аргумента»
  - (λx. s) t -β-> s[t/x]
- Пример:

(λx. (λy. x−y)) 7 2 -β-> (λy. 7−y) 2 -β-> 7−2

- Бывает ещё η-редукция, но мы её не рассматриваем.

Примеры β-редукции

- (λx. (λy. y x) z) v -β-> (λy. y v) z -β-> z v
- (λx. x x) (λx. x x) -β-> (λx. x x) (λx. x x) -β-> (λx. x x) (λx. x x) -β-> … -β-> (λx. x x) (λx. x x)
- (λx. x x y) (λx. x x y) -β-> (λx. x x y) (λx. x x y) y -β-> (λx. x x y) (λx. x x y) y y -β-> …

Эквивалентность

- Термы t и v эквивалентны (записывается t = v), если есть цепочка термов s, r, ..., такая что t ---> s ---> r ---> ... ---> v, где каждая ---> является либо -α->, либо -β->, либо «обратной β-редукцией» <-β-
- Справедливо, что

t = t

если s = t, то t = s

если s = t и t = v, то s = v

если s = t, то s u = t u

если s = t, то u s = u t

если s = t, то λx. s = λx. t

- Эквивалентность – не тождественность:

  λx. x = λy. y но они не тождественны

λ-Редукция

- λ-редукция, это «эквивалентность в одну сторону» (без «обратной β-редукции» <-β-)

t --> t

~~если s --> t, то t --> s~~

если s --> t и t --> v, то s --> v

если s --> t, то s u --> t u

если s --> t, то u s --> u t

если s --> t, то λx. s --> λx. t

- термин λ-редукция соотносится с понятием «вычисление функ. программы»

Нормальная форма

λ-выражение находится в нормальной форме, если ни одна β-редукция не может быть применена.

- Для выражения (λx. (λy. y x) z) v нормальная форма z v
- Для выражения (λx. x x y) (λx. x x y) нормальной формы нет

Нормальная форма

- В каком порядке делать редукции?
- Например, обозначим L= (λx. x x y) (λx. x x y)
- Найдем н. ф. выражения (λu. v) L
  - (λu. v) L = v, если начинаем слева
  - но можно всегда делать справа и зациклиться:

  (λu. v) L -β-> (λu. v) (L y) -β-> (λu. v) (L y y)…
- Выражение (λx. x x) (λx. x x) нормальной формы не имеет (выбор стратегии не спасает).

Стратегии редукции

- Теорема: Если s --> t, где t имеет нормальную форму, по последовательность редукций, в которой всегда выбирается самое левое редуцируемое выражение приводит к терму в нормальной форме.
- Это нормальный порядок редукции

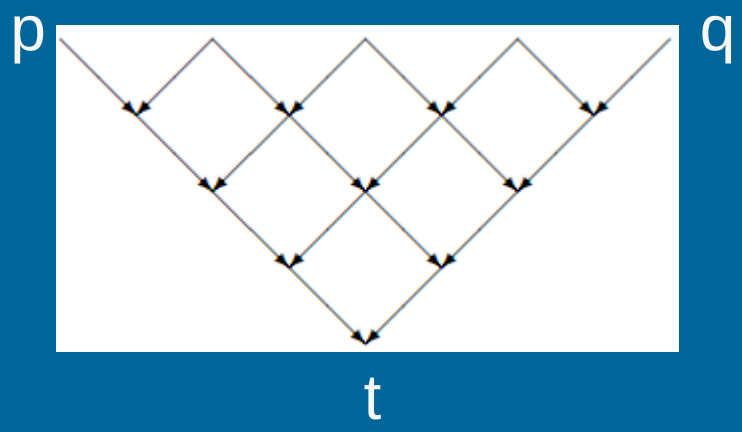
Теорема Черча-Россера

Если t --> u и t --> w, то существует терм v:

u --> v и w --> v.

Теорема об эквивалентности

- Если p = q, то существует выражение t, такое что p --> t и q -->t
- Схема доказательства:



Верхний «зигзаг» существует, так как p = q. Теорема Ч.-Р. позволяет достроить низ картинки.

Единственность нормальной формы

Следствие: Если t = v и t = w, причём v и w имеют нормальную форму, то v = w, причём цепочка редукций состоит только из α-редукций.

Значит, если нормальная форма существует, то она единственна с точность до α-редукций!

Комбинаторы

Комбинатор – λ-терм без свободных переменных.

Примеры:

- I ≡ λx. x (тождественность)
- S ≡ λf λg λx. (f x) (g x) (выделение)
- K ≡ λx λy. x (константа: K a --> λy. a)

Утверждается, что для любого λ-терма существует его эквивалент без λ-абстракций, являющийся композицией I, S, K и переменных.

I = S K K

Комбинаторы можно рассматривать как аналог машинного кода для λ-выражений.

Комбинатор неподвижной точки

- Y – комбинатор неподвижной точки, если, применив его к функции f , мы получим неподвижную точку x для f , то есть такое x, что f (x) = x. Для всех f имеем: f (Y f) = Y f
- Y-комбинатор Карри λf. (λx. f (x x))(λx. f (x x))
- Другой Y-комбинатор придумал А. Тьюринг
- Y-комбинаторы – основа для рекурсивных функций. Он даёт возможность описать λ-выражением анонимную рекурсивную функцию.

Арифметика в λ-исчислении

- $0 := \lambda f.\ \lambda x.\ x$
- $1 := \lambda f.\ \lambda x.\ f\ x$
- $2 := \lambda f.\ \lambda x.\ f\ (f\ x)$
- $3 := \lambda f.\ \lambda x.\ f\ (f\ (f\ x))$
- …

Арифметика в λ-исчислении

- SUCC := λ*n*. λ*f*. λ*x*. *f* (*n f x*)
- PLUS := λ*m*. λ*n*. λ*f*. λ*x*. *m f* (*n f x*)

т. е. PLUS := λ*n*. λ*m*. *m* SUCC *n*

- MULT := λ*m*. λ*f*. *m* (*n f*)

т. е. MULT := λ*m*. λ*n*. *m* (PLUS *n*) 0

Пример

PLUS 2 3 == (λ*m*. λ*n*. λ*f*. λ*x*. *m f* (*n f x*))

(λ*f*. λ*x*. *f* (*f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) --->

(λ*n*. λ*f*. λ*x*. (λ*f*. λ*x*. *f* (*f x*)) *f* (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) --->

(λ*n*. λ*f*. λ*x*. (λ*a*. λ*b*. *a* (*a b*)) *f* (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*)))--->

(λ*n*. λ*f*. λ*x*. (λ*b*. *f* (*f b*)) (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) --->

(λ*n*. λ*f*. λ*x*. *f* (*f* (*n f x*))) (λ*f*. λ*x*. *f* (*f* (*f x*))) --->

(λ*f*. λ*x*. *f* (*f* ((λ*f*. λ*x*. *f* (*f* (*f x*))) *f x*))) -->

(λ*f*. λ*x*. *f* (*f* ((λ*a*. λ*b*. *a*(*a* (*a b*))) *f x*))) =>

(λ*f*. λ*x*. *f* (*f* (*f*(*f* (*f x*))))) -- это 5

Логика в λ-исчислении

- TRUE := λ*x*. λ*y*. *x*
- FALSE := λ*x* . λ*y*. *y*
- AND := λ*p*. λ*q*. *p q p*
- OR := λ*p*. λ*q*. *p p q*
- NOT := λ*p*. λ*a*. λ*b*. *p b a*
- IFTHENELSE := λ*p*. λ*a*. λ*b*. *p a b*