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

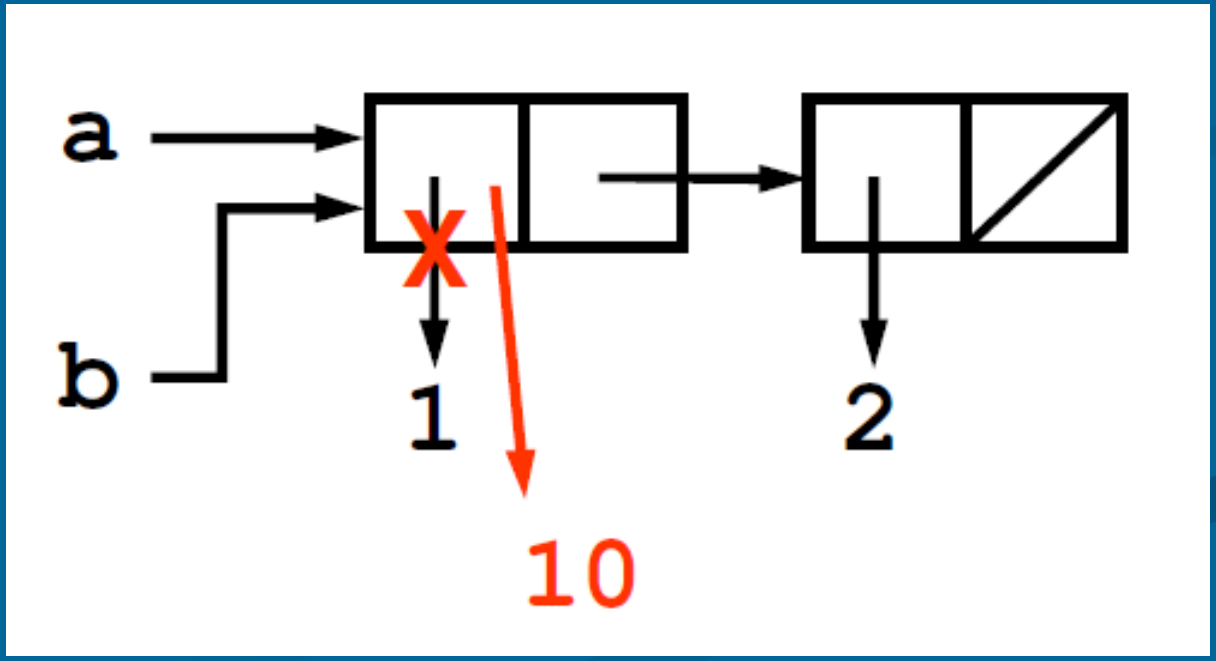
- Присваивание делает неактуальной подстановочную модель!
- Рассмотрим withdraw без присваивания

```
(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

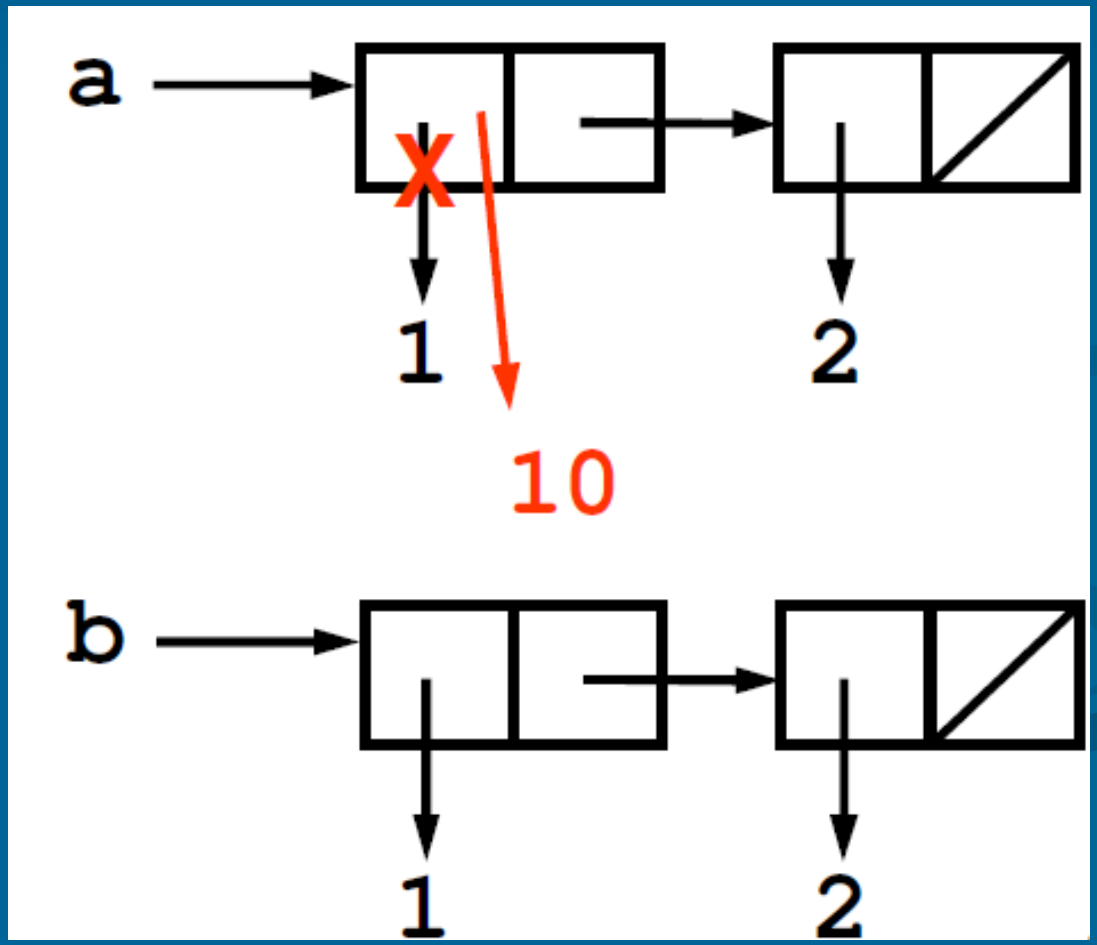
- (set-car! <пара> <новый car>) меняет 1й элемент точечной пары
- (set-cdr! <пара> <новый cdr>) меняет 2й элемент точечной пары

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



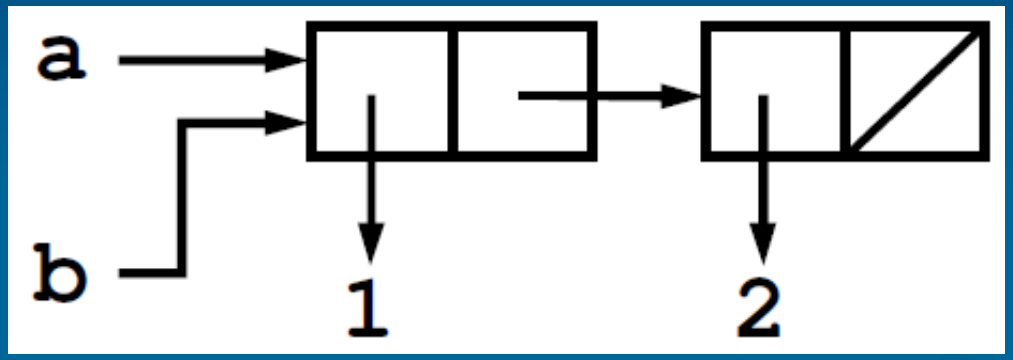
Что будет, если b определить иначе

(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

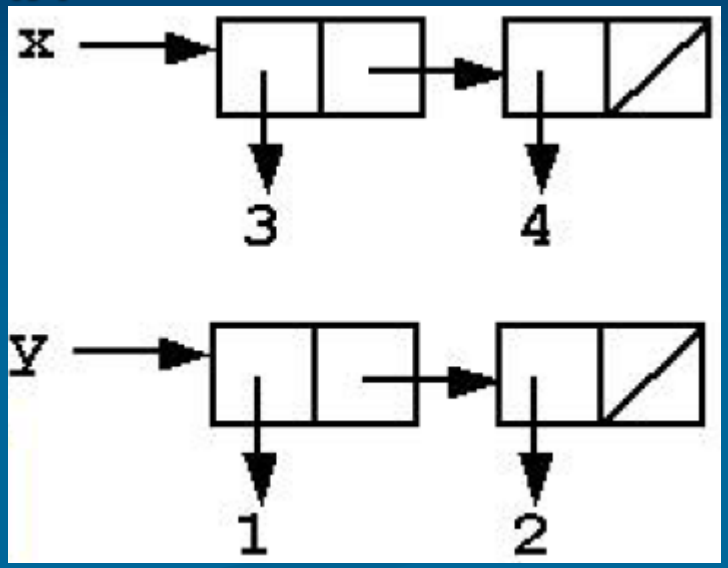
  но не совпадать

Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

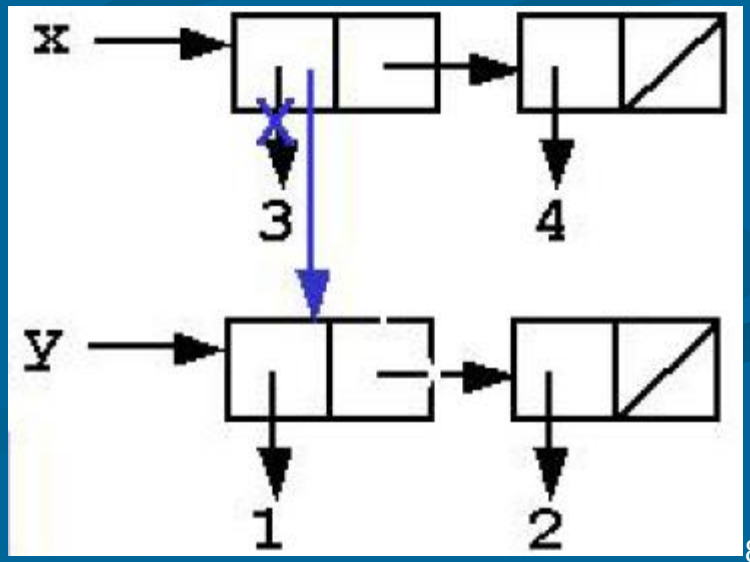
(define x ‘(3 4))

(define y ‘(1 2))



(set-car! x y)

x → ((1 2) 4)



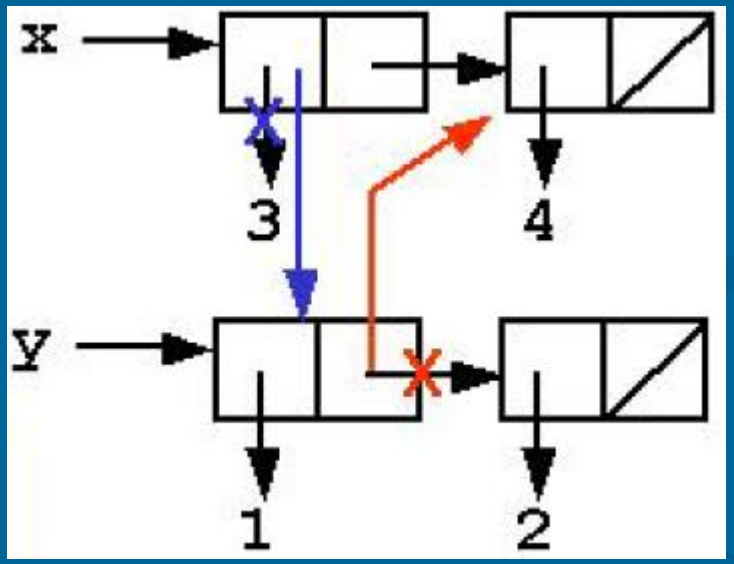
Последствия присваивания

- 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

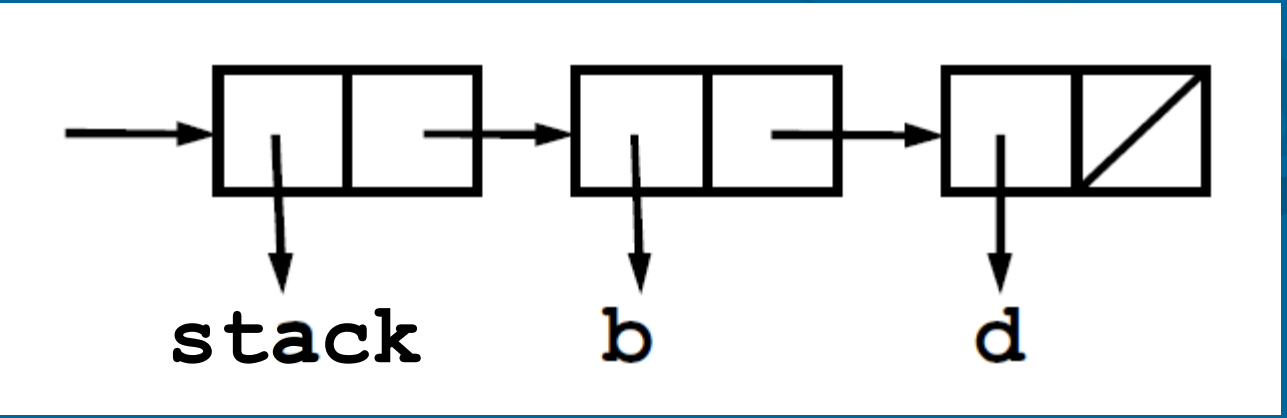
Промежуточный итог

- 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.

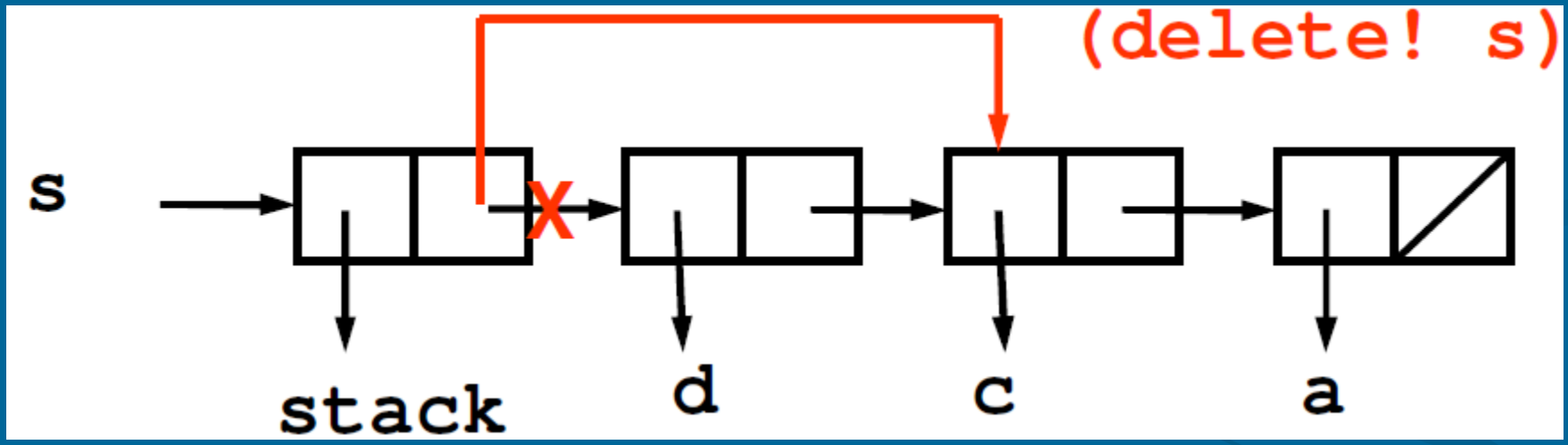


Стек. Реализация

```
(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

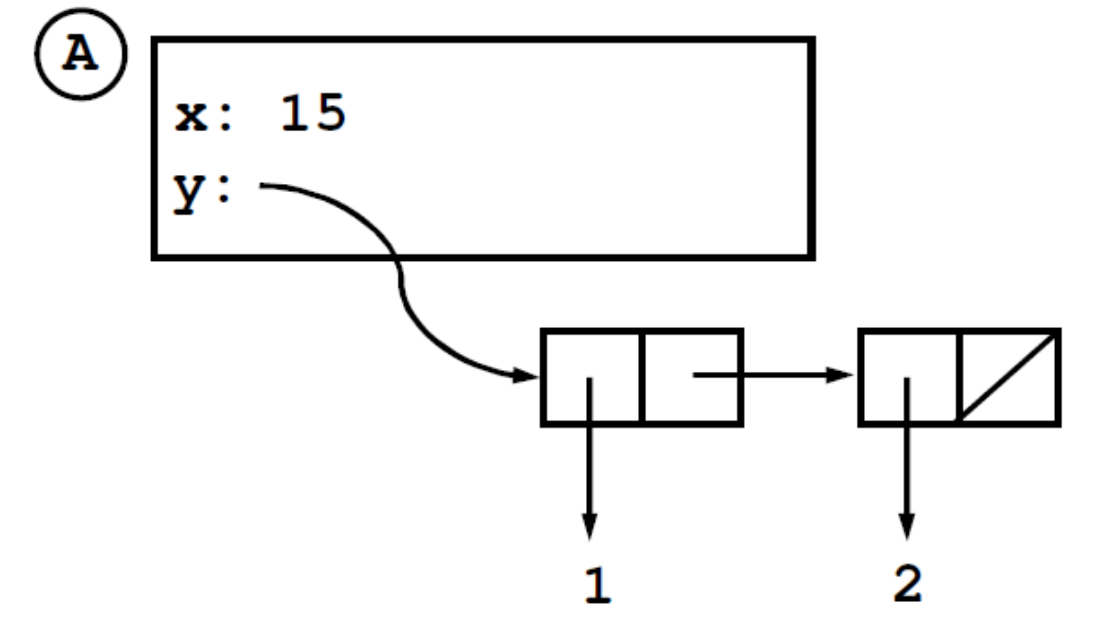
- в подстановочной модели имя – метка для значения
- в МВО имя – место для хранения значения

- 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

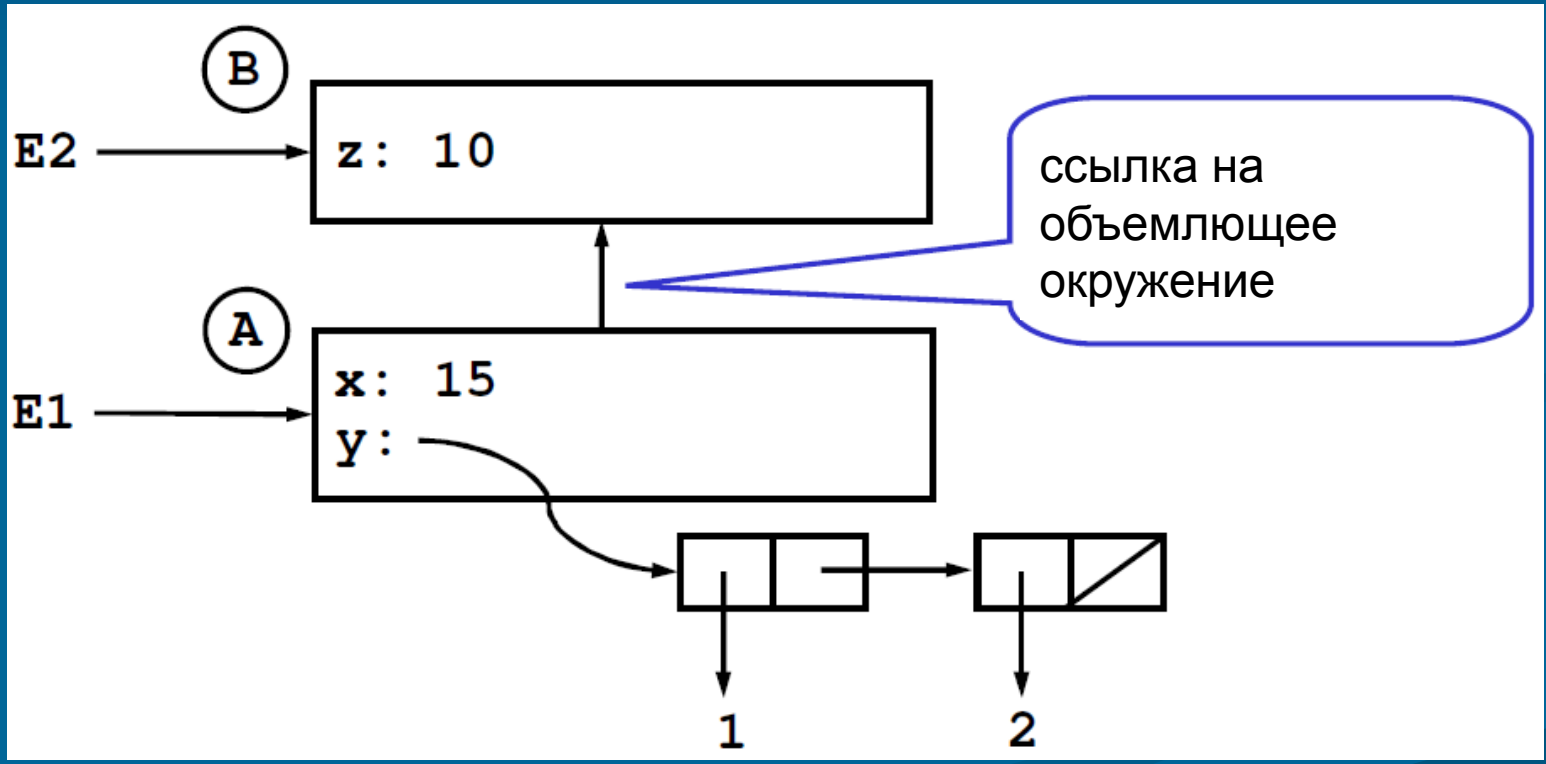
- Кадр – таблица связываний
- Связывание – пара имя : значение
- Пример:



- В кадре A содержатся два связывания

Элементы МВО

- Окружение – последовательность кадров
- Пример:



- Окружения E1 и E2 разделяют общий кадр B

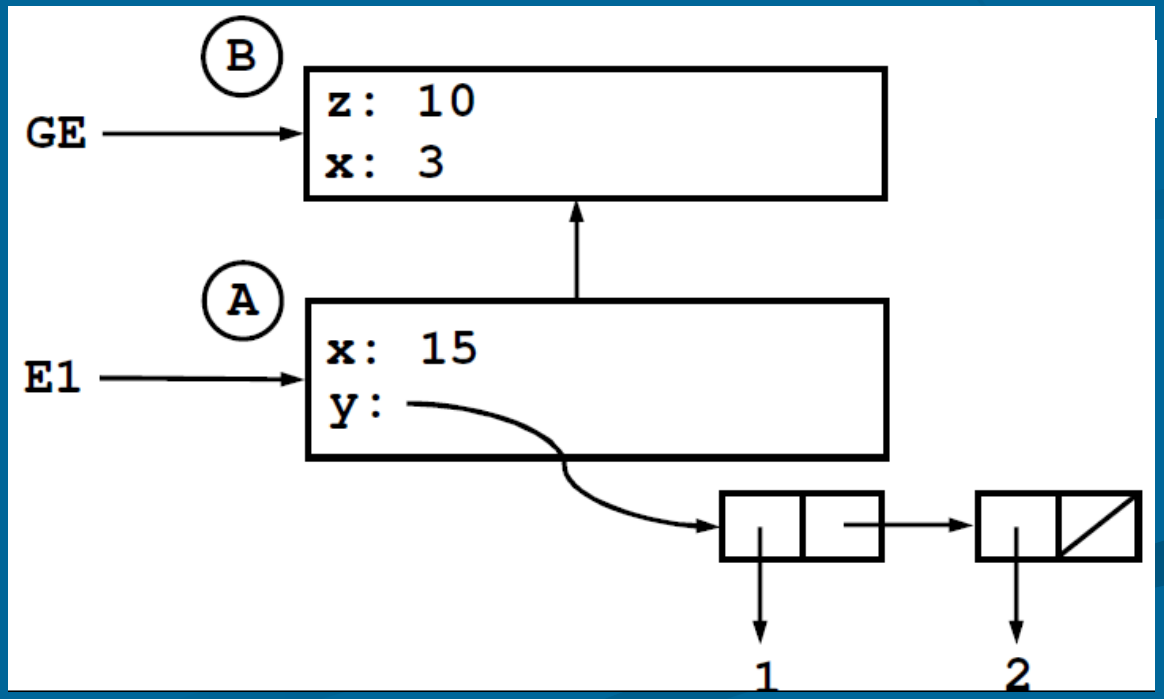
МВО

- 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.

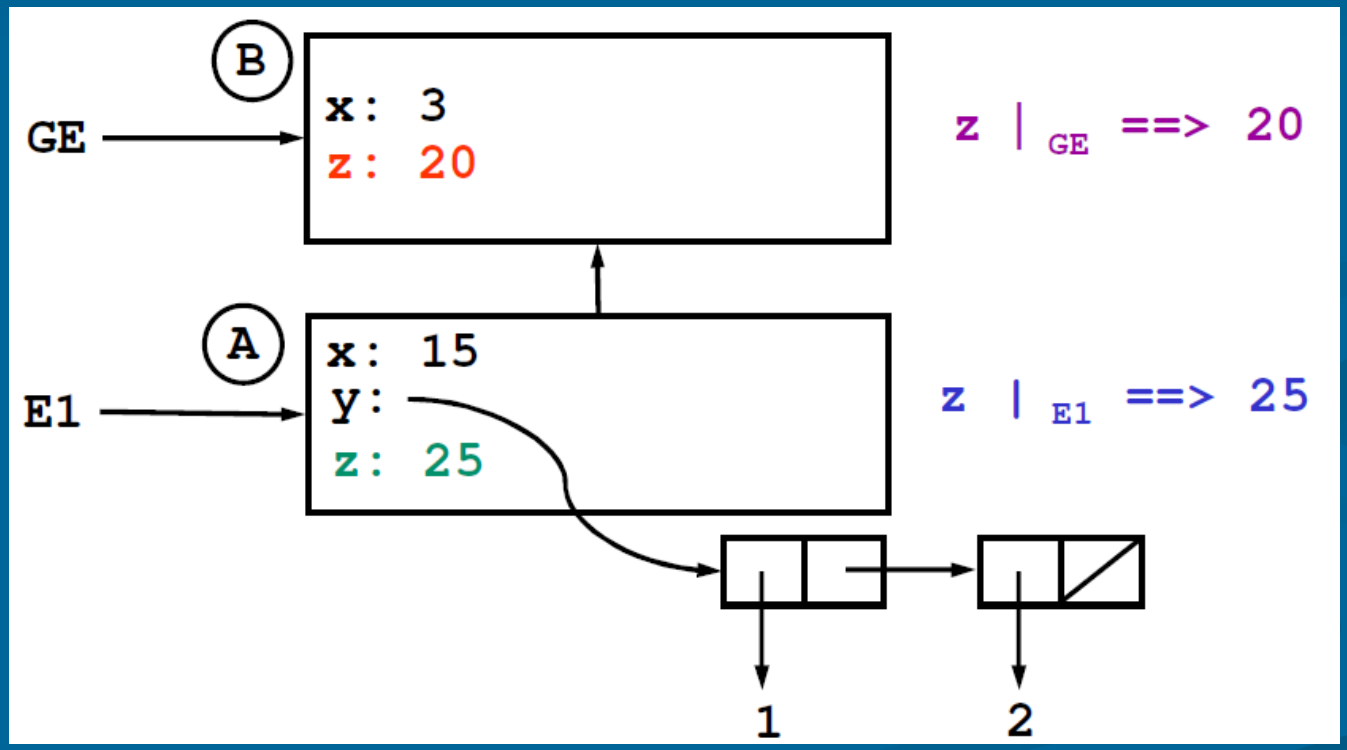


  - Остальные связывания X *скрыты*.

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.

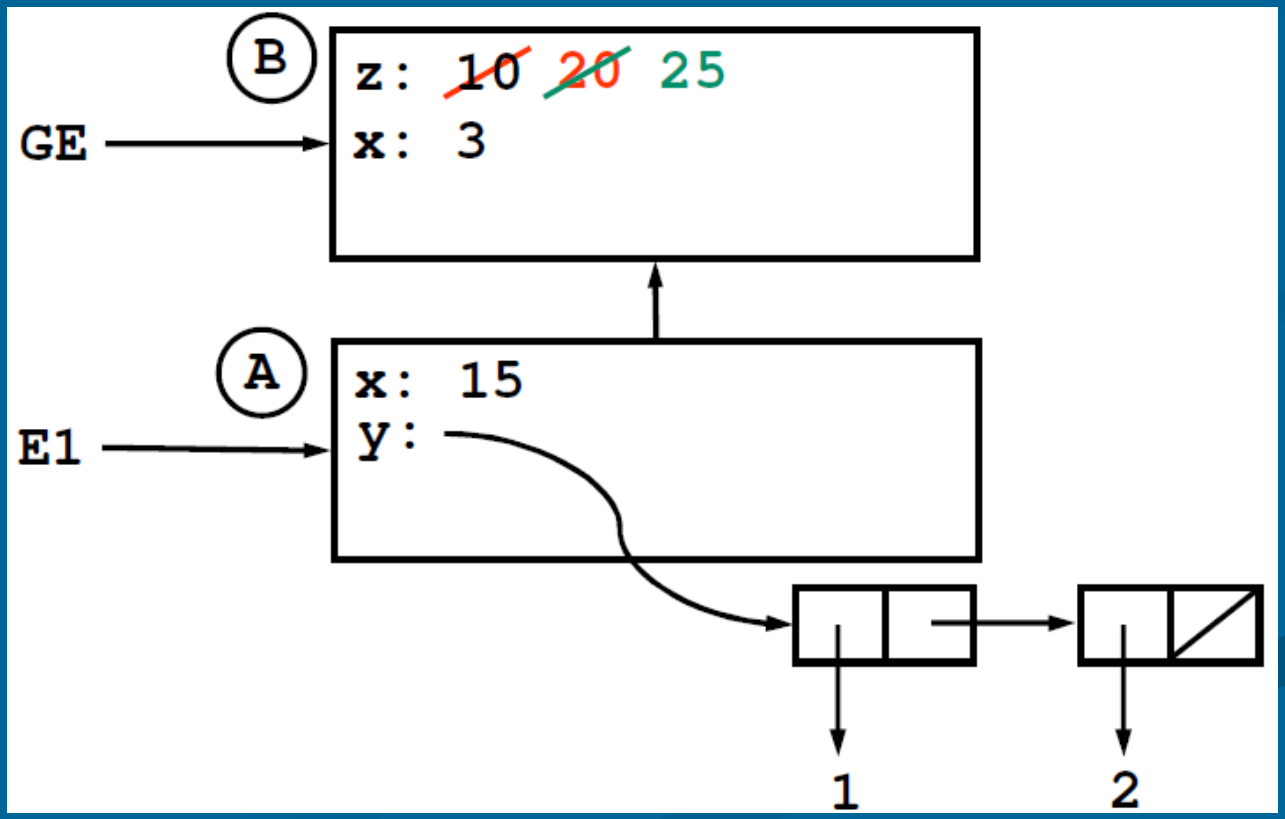
МВО. Правила вычислений

- Вычисление set! в окружении:

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

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

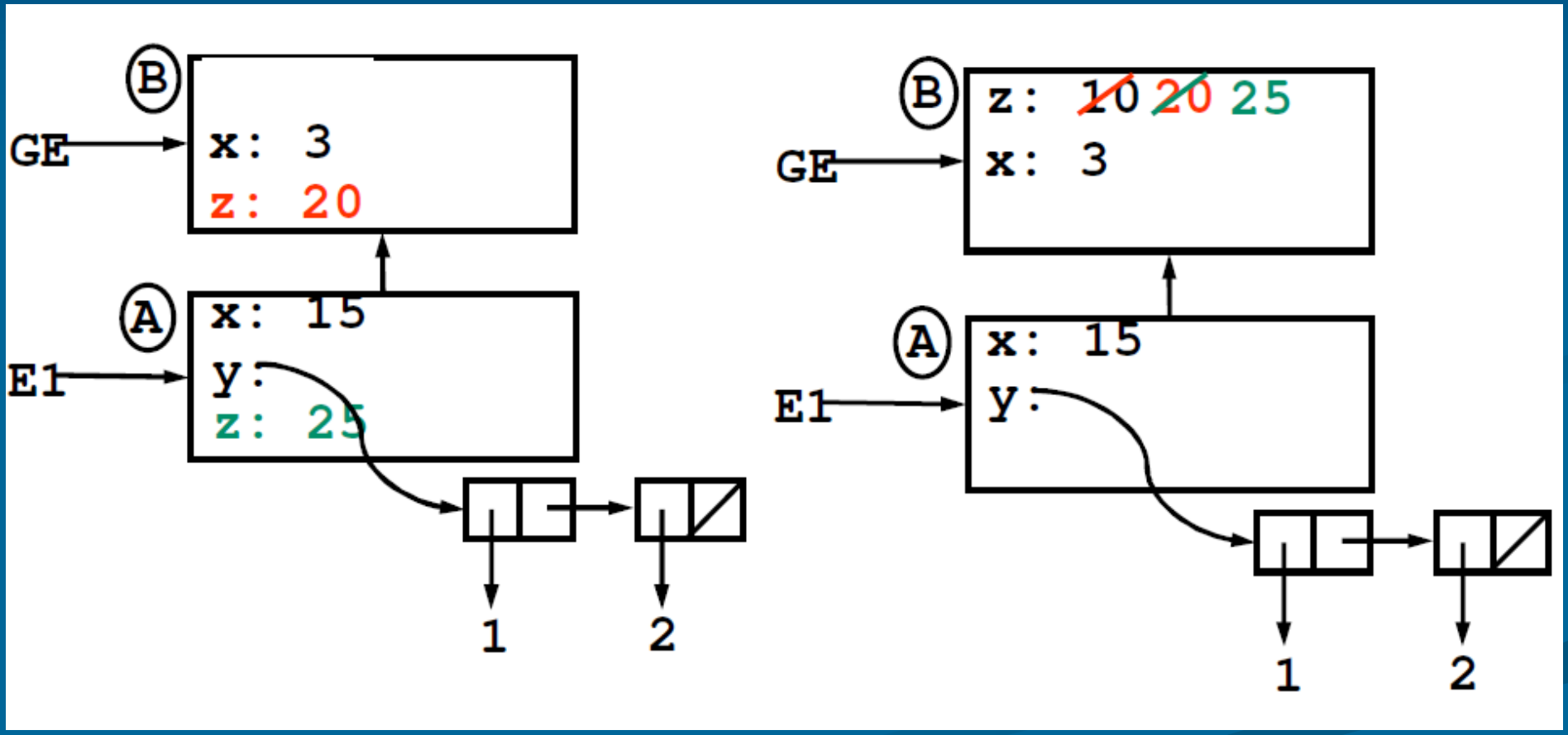
(set! z 25) $|_{E1}$



Если имя нигде не встретилось выдать ошибку!

define и set! вычисляются не одинаково

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

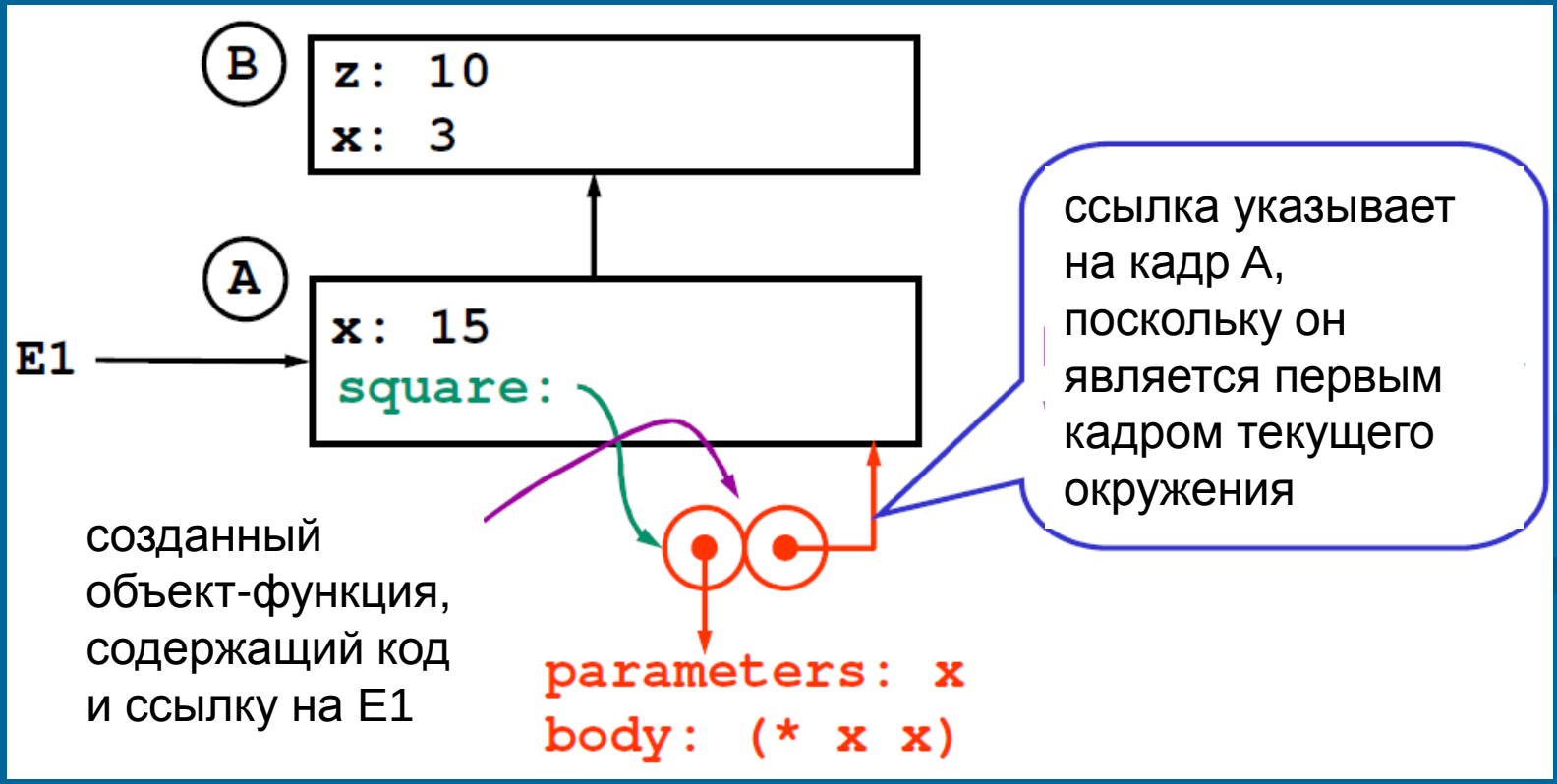


МВО. Правила вычислений

- Вычисление lambda в окружении:

  создать функцию с телом из lambda и ссылкой на текущее окружение

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

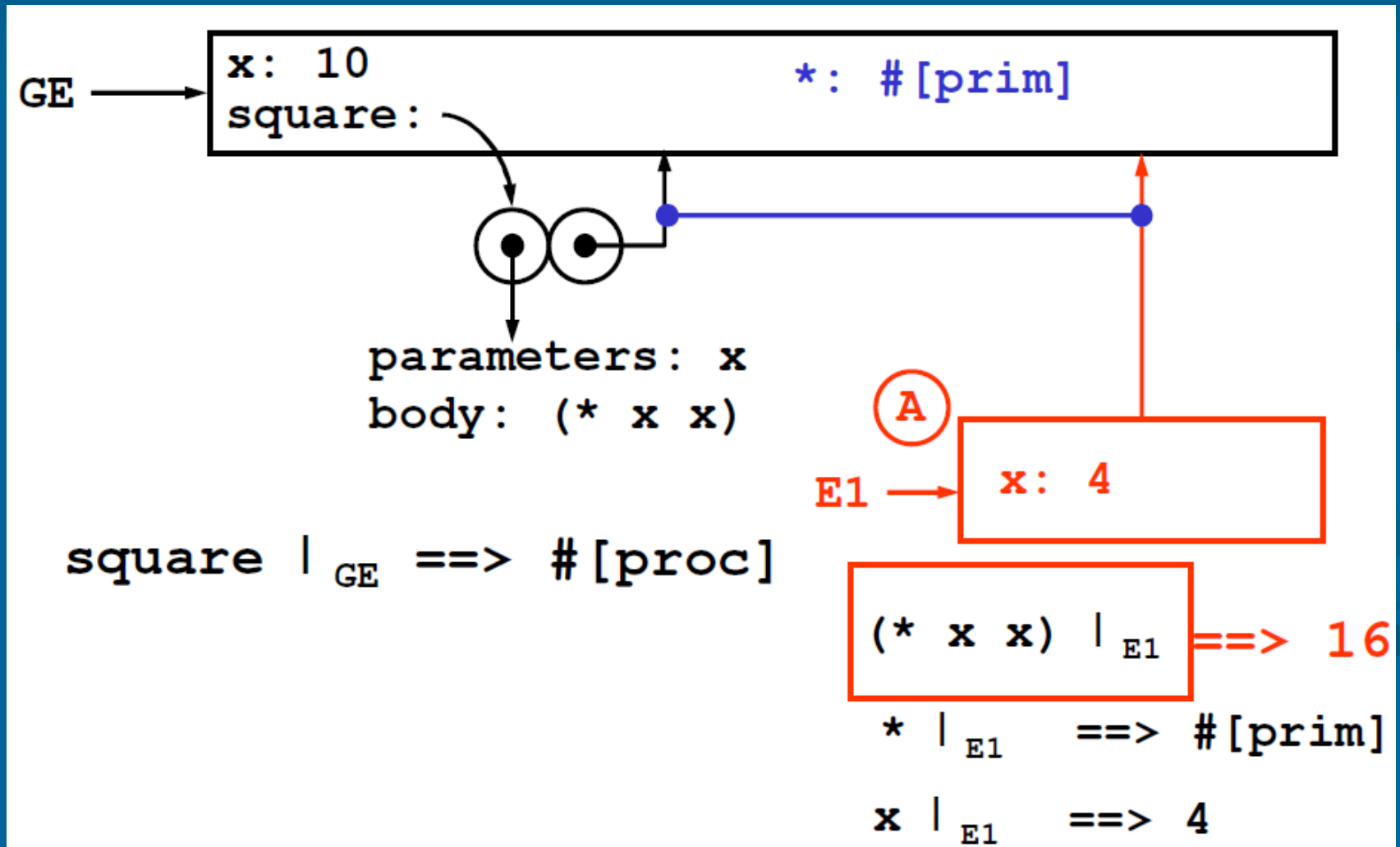


МВО. Правила вычислений

- 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.

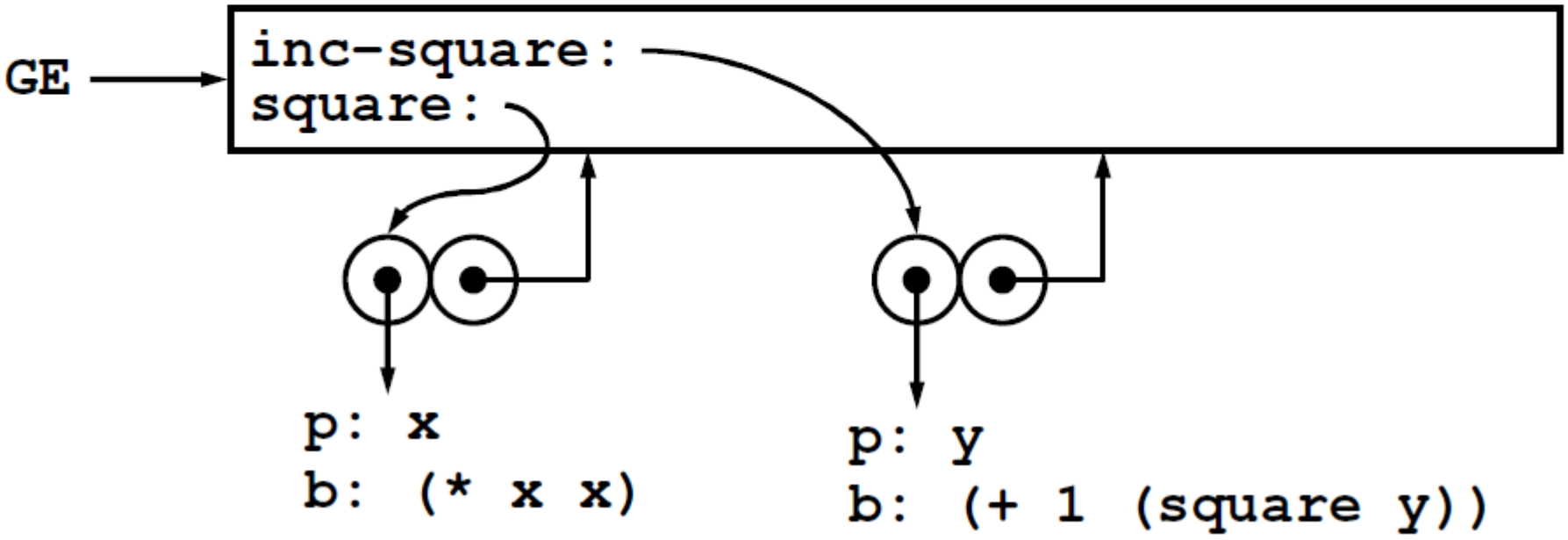
Пример применения функции

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



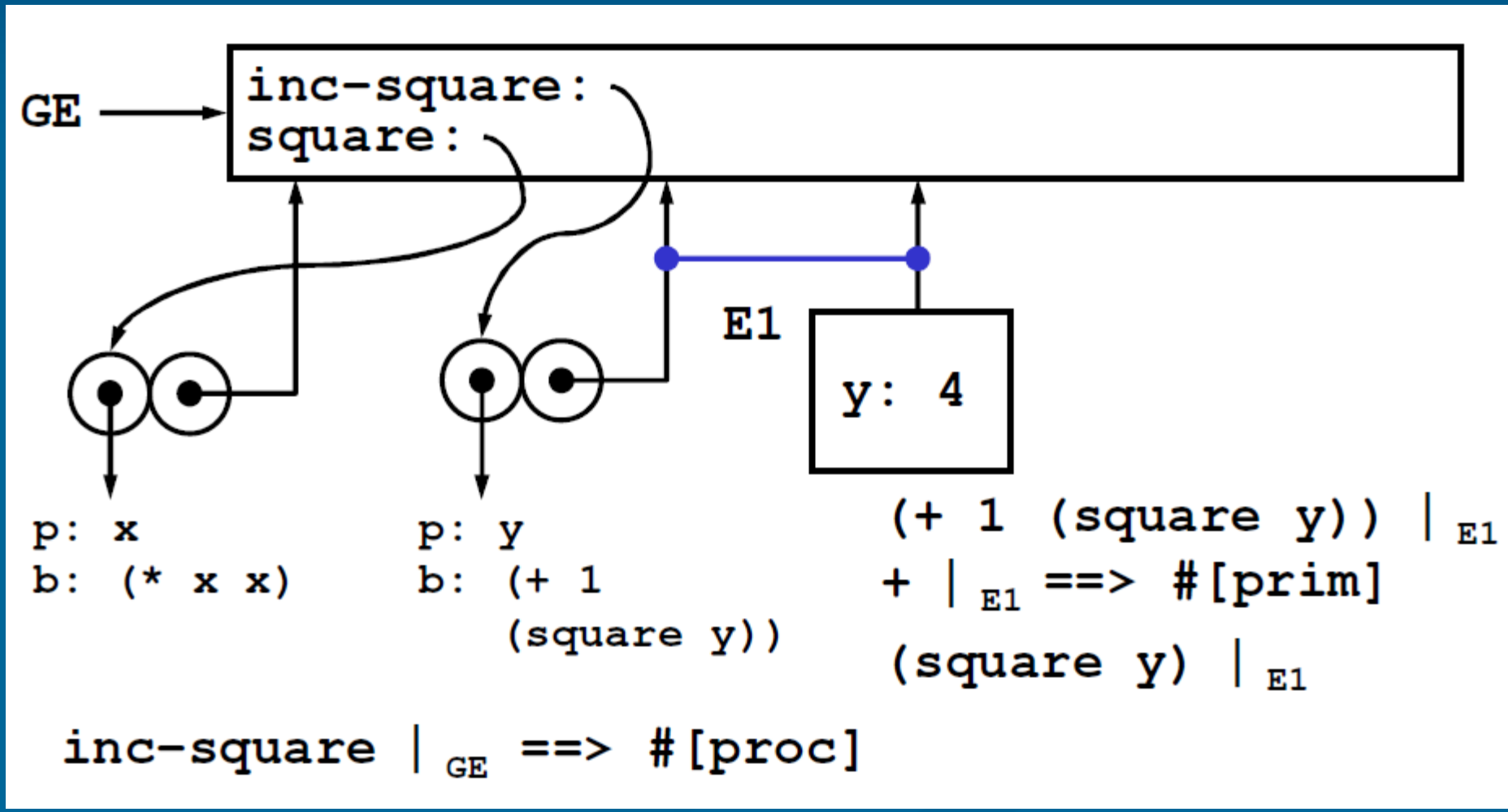
Ещё пример

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



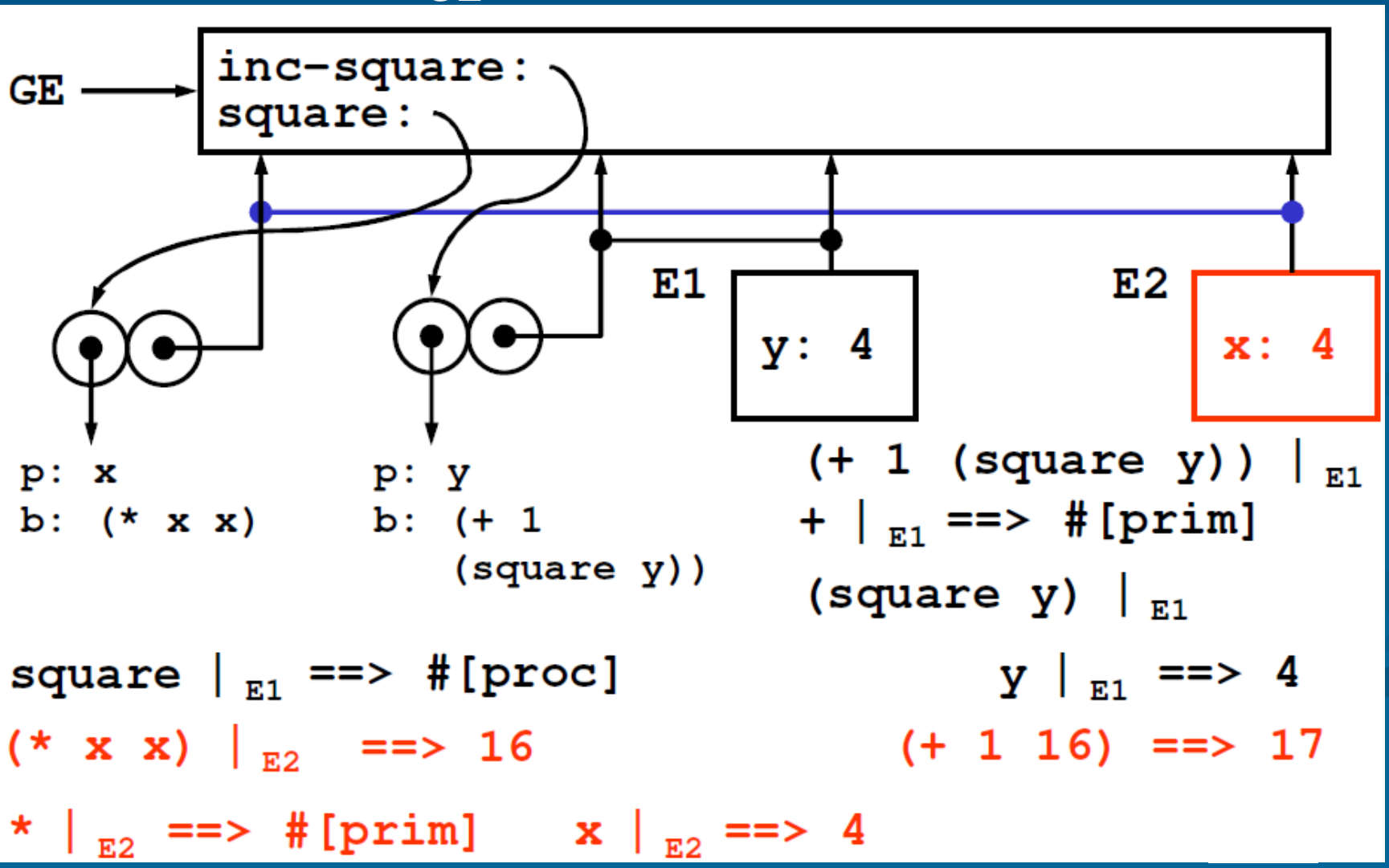
Продолжение примера

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



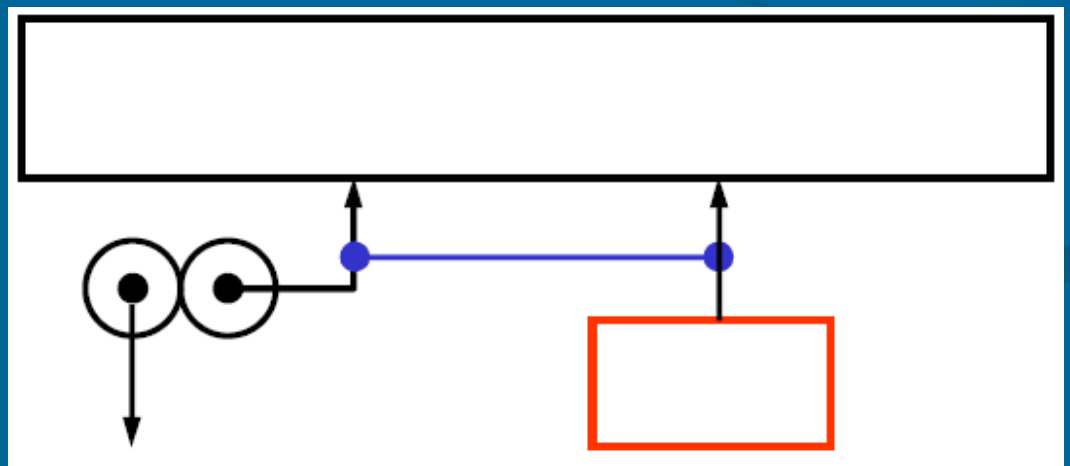
Продолжение примера

(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:

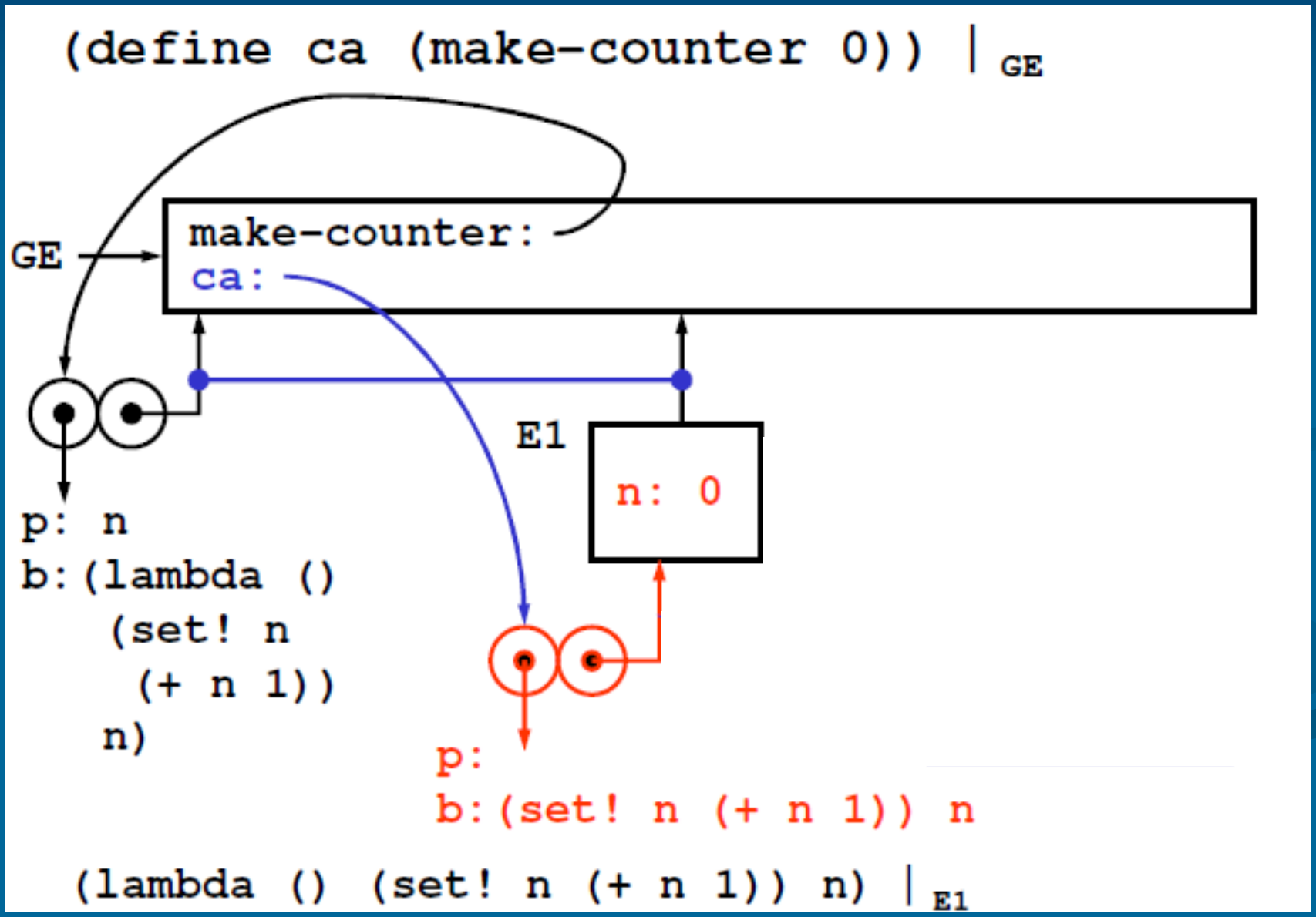


Снова пример

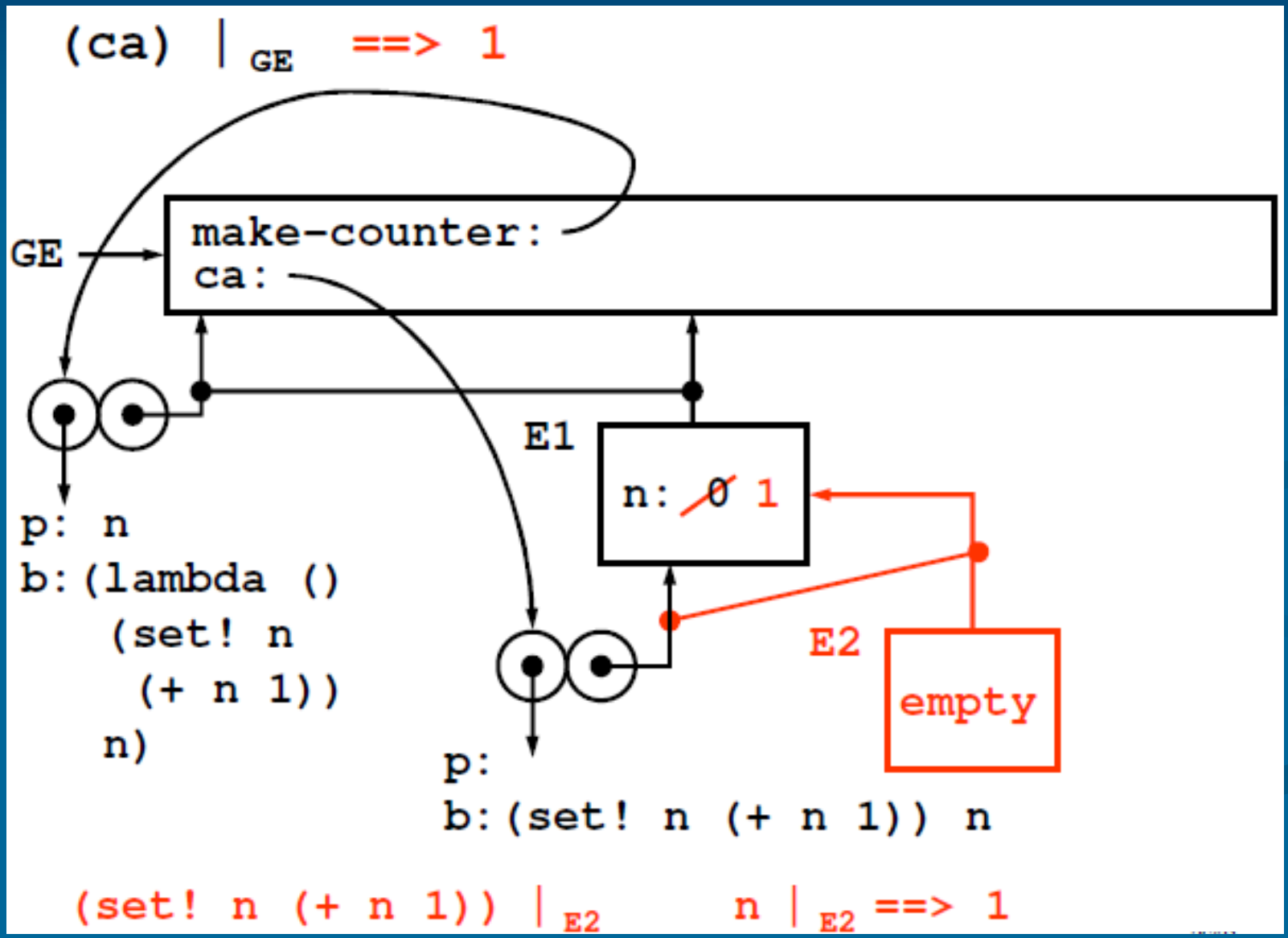
- Счётчик

```
(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
```

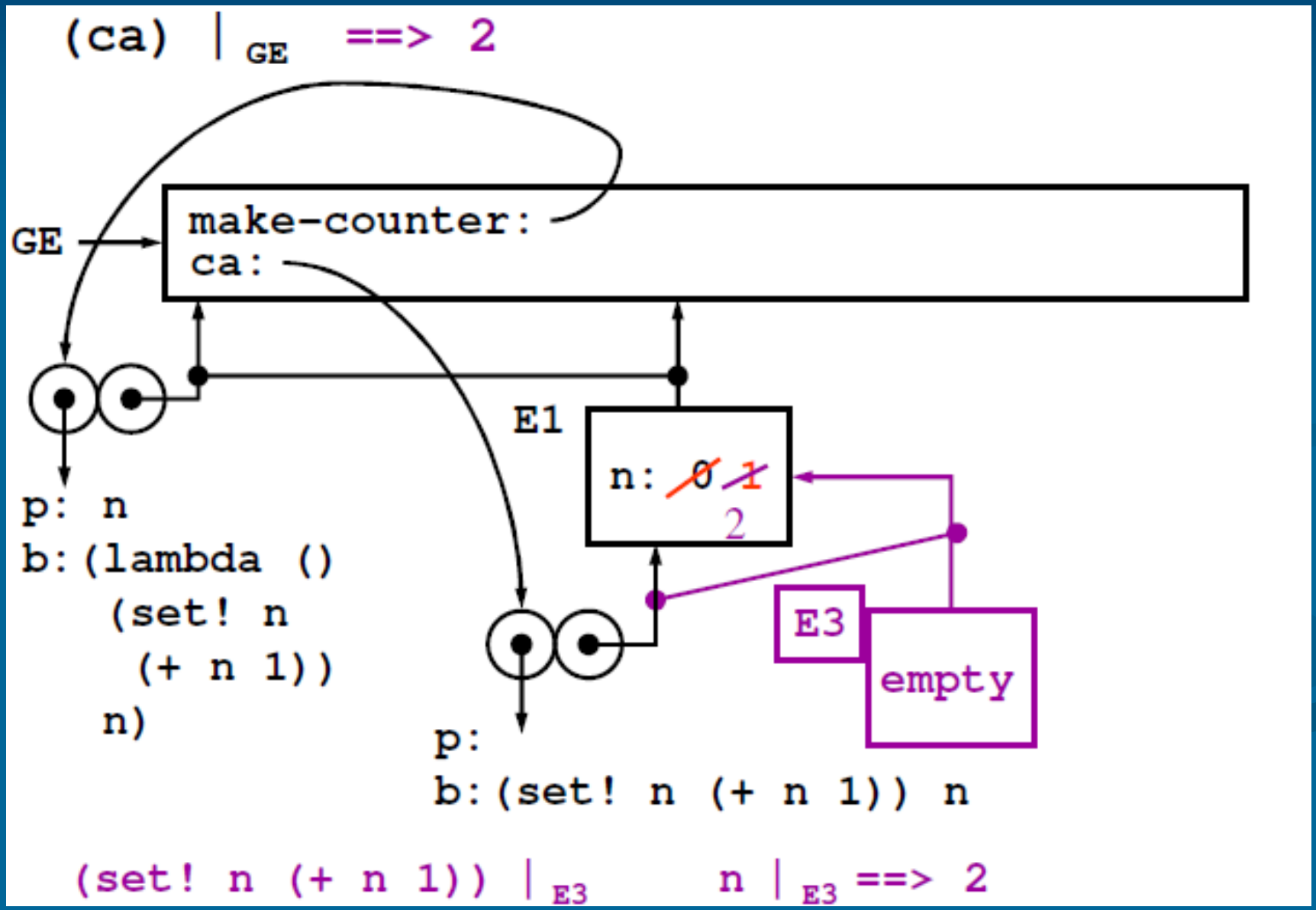
Счётчик. Продолжение



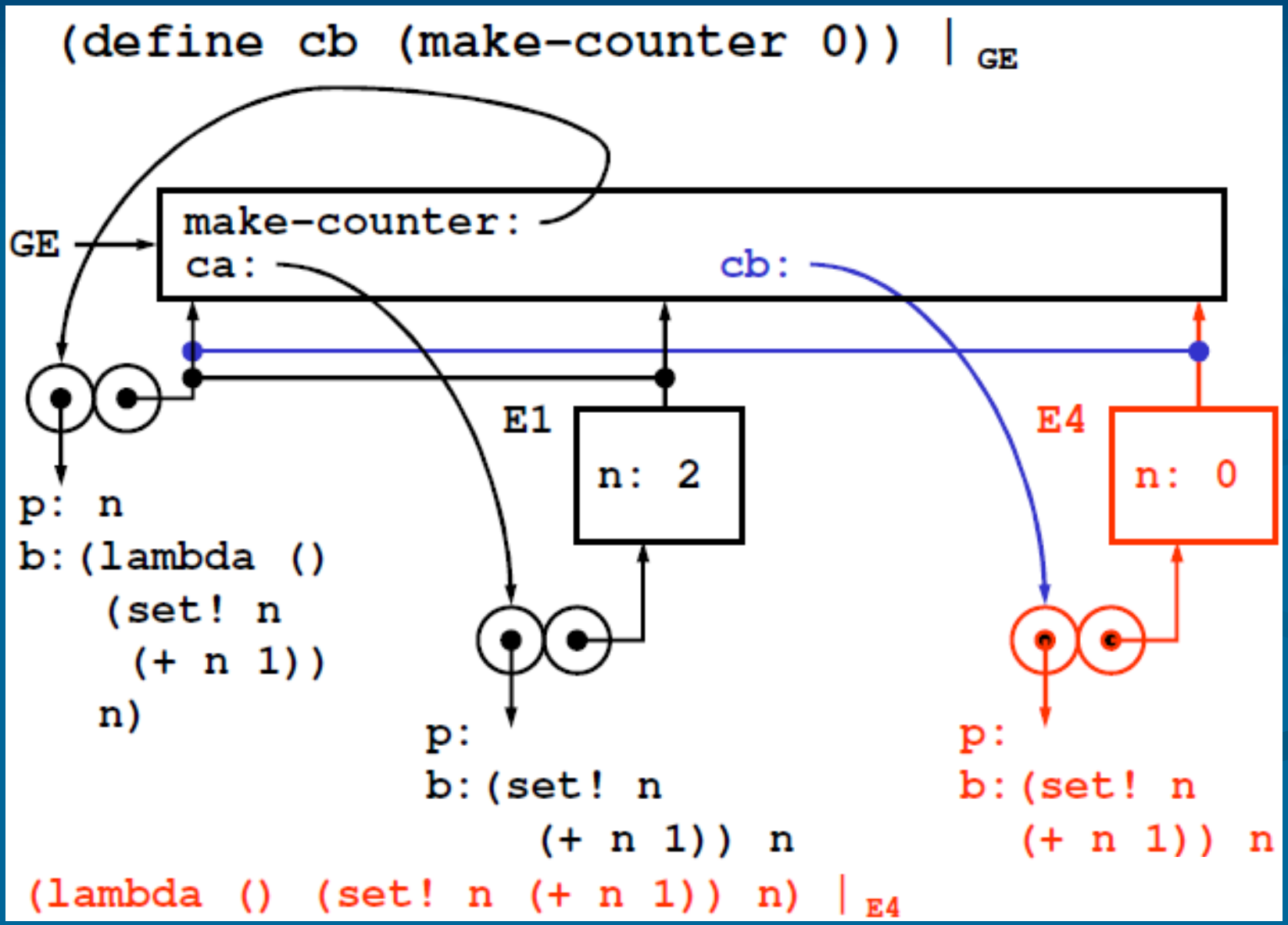
Счетчик. Продолжение



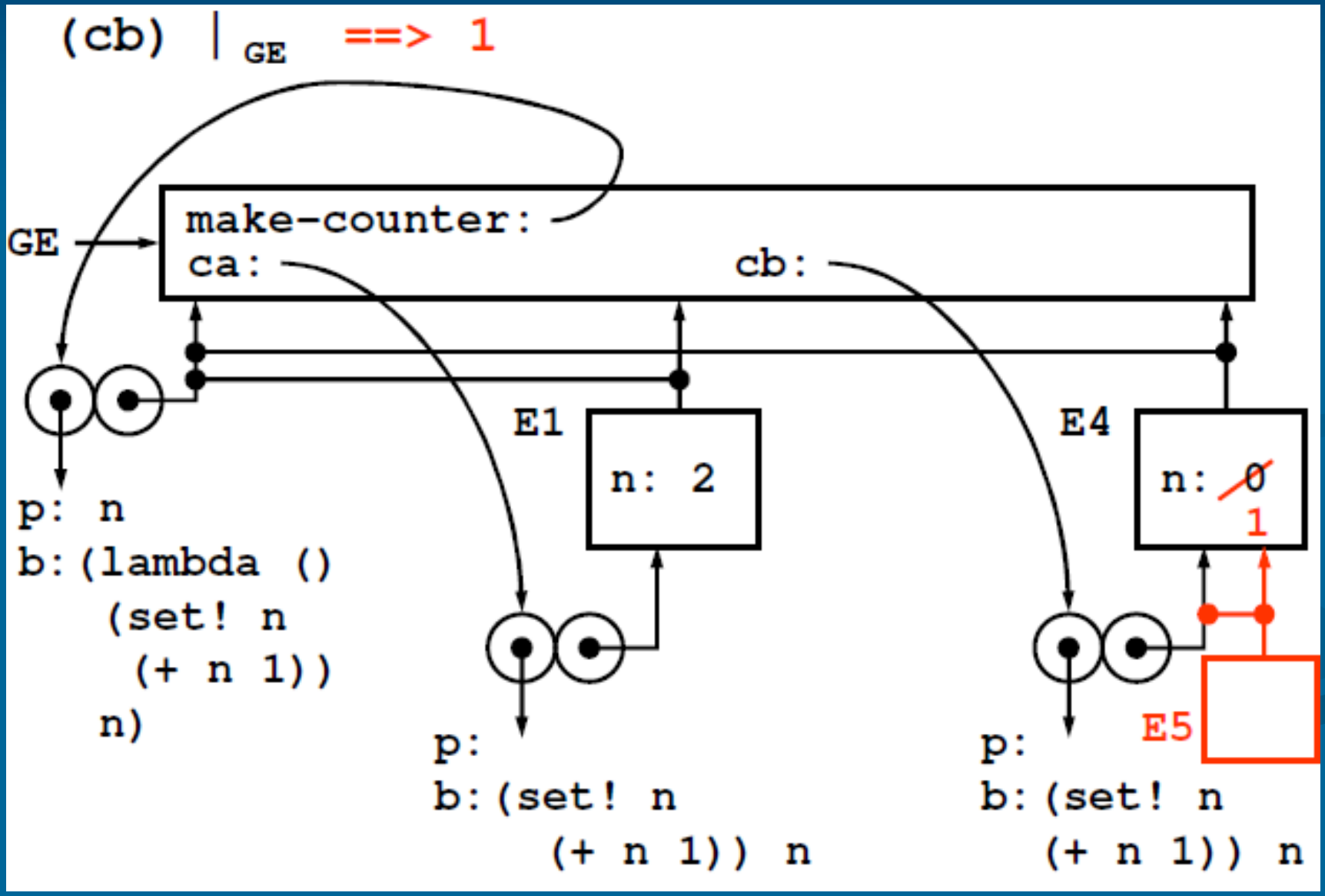
Счетчик. Продолжение



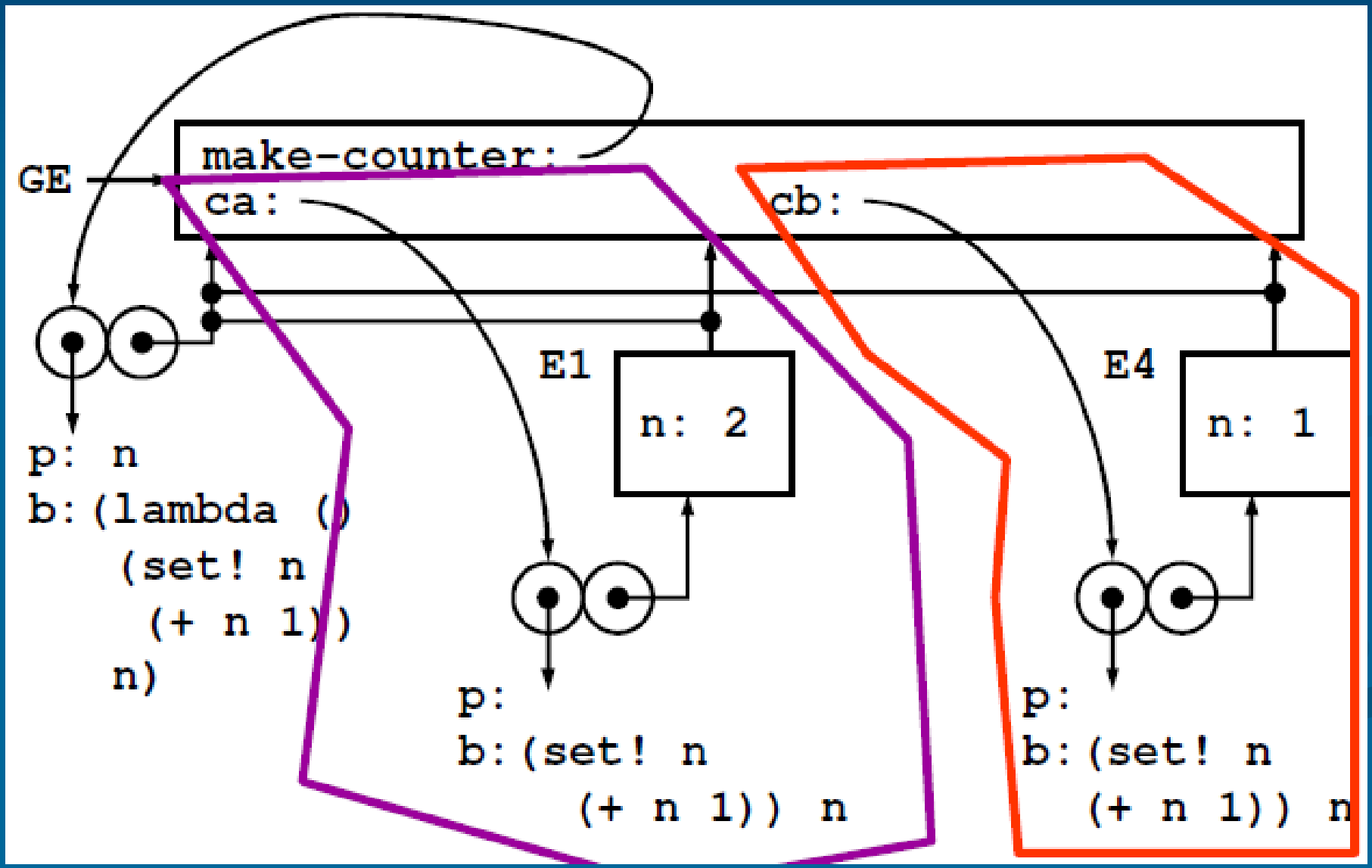
Счетчик. Продолжение



Счетчик. Продолжение



Счетчик. Продолжение



Таблица

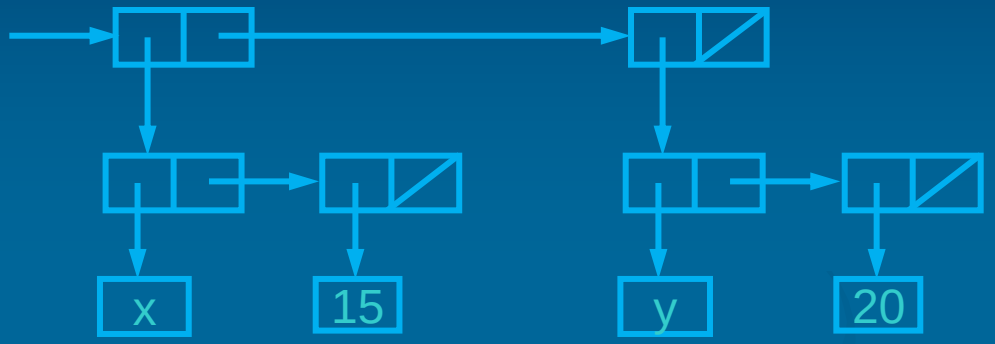
- конструктор (make-table)
- селектор (lookup <table> <key>)
- мутатор (insert! <table> <key> <value>)
- Список пар: (*table* (key1 . val1) (key2 . val2)…)

```
(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)

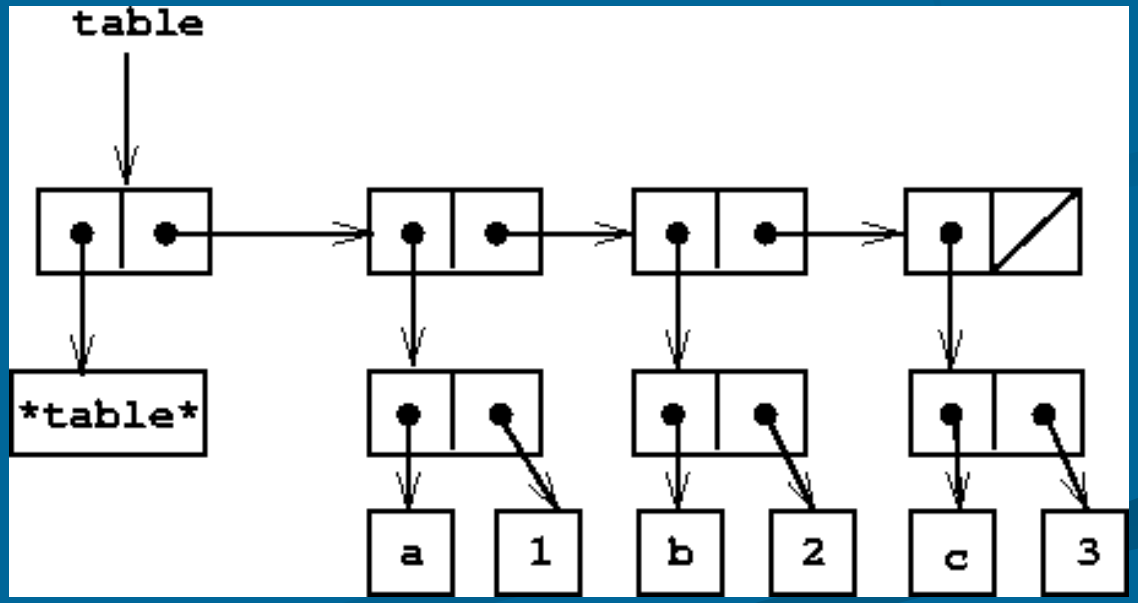
(assq ‘(x 1) ‘(((x 1) 2) ((y 1) 3)))) ==> #f

(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)

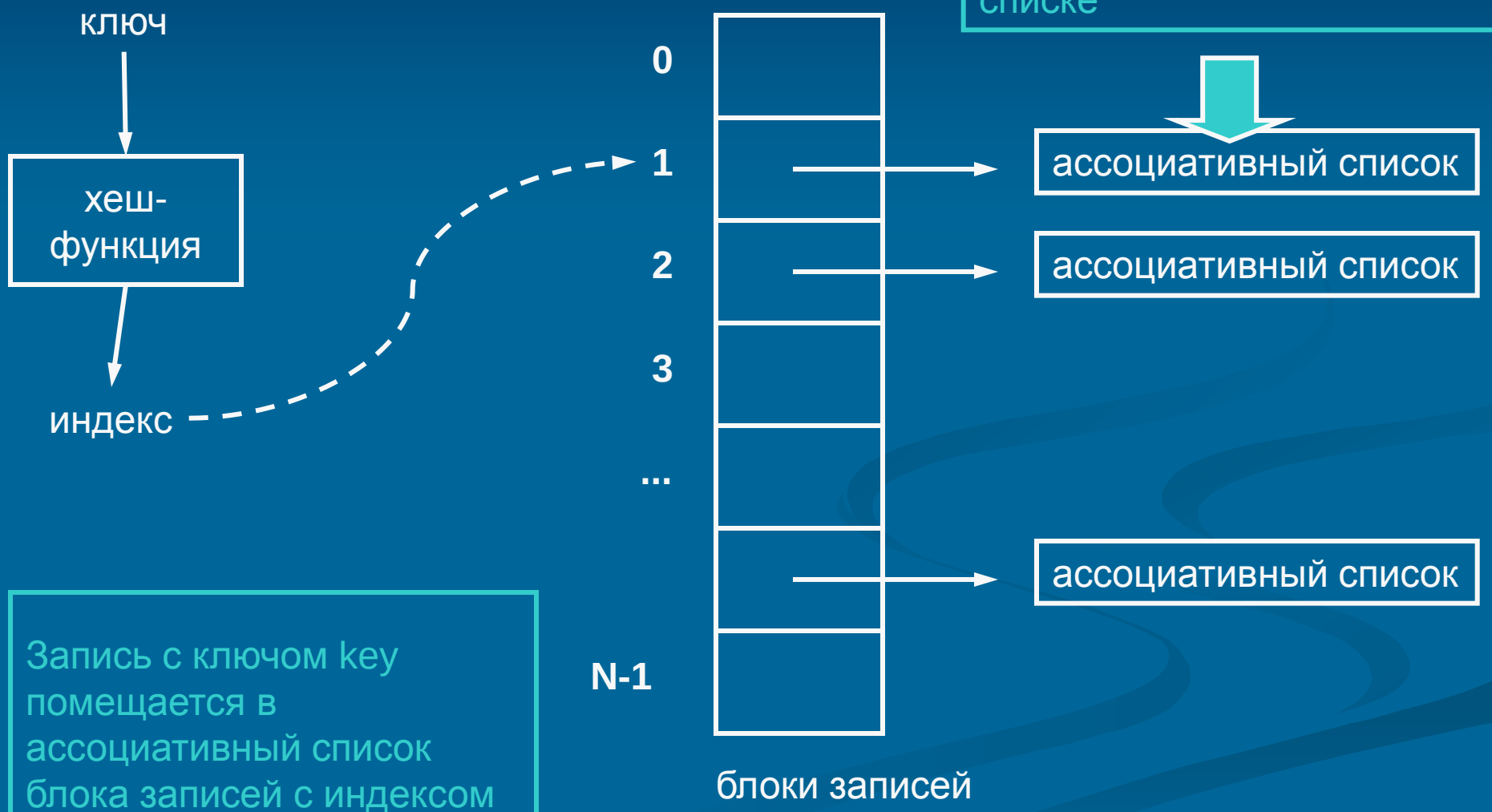
```
> (define beatles (vector 'john 'paul 'george 'pete))
> (vector-set! beatles 3 'ringo)
> beatles ==> ‘#(john paul george ringo)
> (vector-length beatles) ==> 4
```

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.