Лекция 6. Присваивание.
Модель вычислений с окружениями

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> <выражение>)
- как работает:
  - <имя> должно быть определено до set! !!!
  - вычисляет <выражение>
  - связывает значение <выражения> 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)
             (error ”Not enough money”)))
```

Последствия присваивания

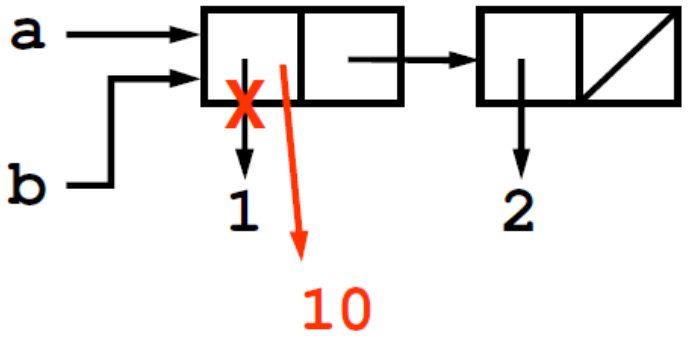
- Присваивание делает неактуальной подстановочную модель!
- Рассмотрим withdraw без присваивания

```
> (define (withdraw2 amount)
    (if     (>= balance amount) (- balance amount)
            (error ”Not enough money”)))
> (withdraw2 20) -> 80
> (withdraw2 20) -> 80
> (withdraw 20)   -> ? (в подстановочной модели)
(if  (>= 100 20)
           (begin   (set! balance (- 100 20))
                       100)
           (error ”Not enough money”)))     -> 100?!
```

Присваивание и точечные пары

- (require scheme/mpair) разрешает мутируемые пары
- (mcons <голова> <хвост>) создаёт мутируемую точечную пару
- (mlist $<e_0>$ $<e_1>$ … $<e_n>$) создаёт мутируемый список
- (set-mcar! <пара> <новый car>) меняет car мутируемой пары
- (set-mcdr! <пара> <новый cdr>) меняет cdr мутируемой пары

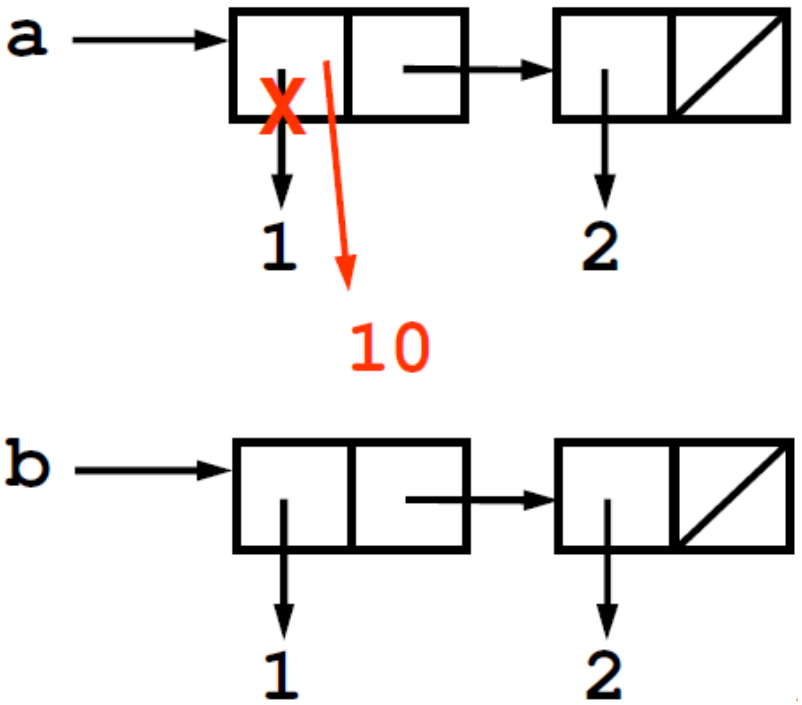
```
> (define a (mlist 1 2))
> (define b a)
> a -> {1 2}
> b -> {1 2}
> (set-mcar! a 10)
> b -> {10 2}
```



- в учебнике иначе: вместо set-mcar! пишут set-car!, вместо set-mcdr! – set-cdr!, вместо mlist – list, mcons – cons, mcar – car, …

Что будет, если b определить иначе?

```
> (define a (mlist 1 2))
> (define b (mlist 1 2))
> a -> {1 2}
> b -> {1 2}
> (set-mcar! a 10)
> a -> {10 2}
> b -> {1 2}
```

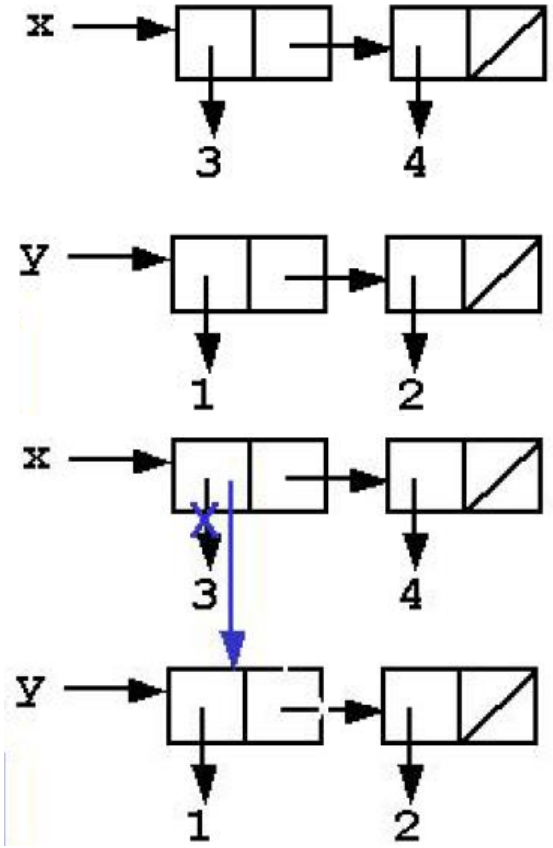


Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

> (define x (mlist 3 4))

> (define y (mlist 1 2))



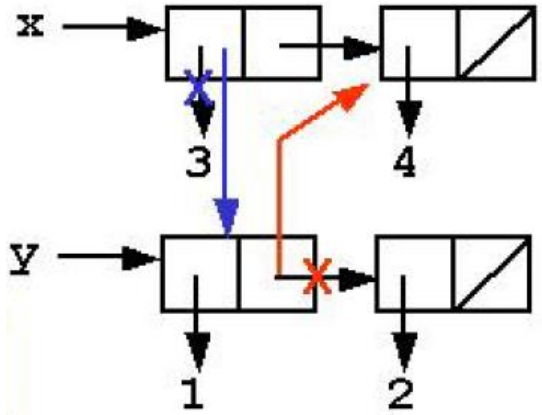
> (set-mcar! x y)

> x -> {{1 2} 4}

Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

...
> (set-mcdr! y (mcdr x))
> y -> {1 4}
> x -> {{1 4} 4}



- Порядок вычислений влияет на результат

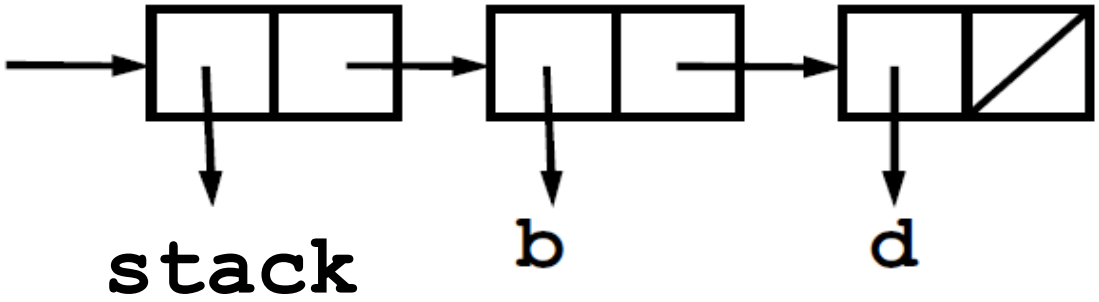
> (* (withdraw 10) (withdraw 40)) --> (* 90 50) -> 4500
> ((lambda (x) (* (withdraw 10) x)) (withdraw 40)) --> (* 50 60) -> 3000

Промежуточный итог

- Встроенные возможности Scheme для присваивания (мутаторы):
  - спецформа set! (есть в scheme/base)
  - спецформы set-car!, set-cdr! (нет в scheme/base)
  - есть дополнительный модуль для mutable-структур scheme/mpair со своим набором функций mcons, mcar, mcdr, mlist, … и набором спецформ set-mcar!, set-mcdr!
  - векторы мутируемые (см. доки Racket)
- Присваивание создаёт дополнительные трудности:
  - возникают сторонние эффекты;
  - подстановочная модель больше не подходит для описания работы программ

Мутируемые структуры данных. Стек

- Конструктор (make-stack)
- Селектор (top-stack s)
- Операции:
  - (insert-stack! s e)
  - (delete-stack! s)
  - (stack? s)
  - (empty-stack? s)



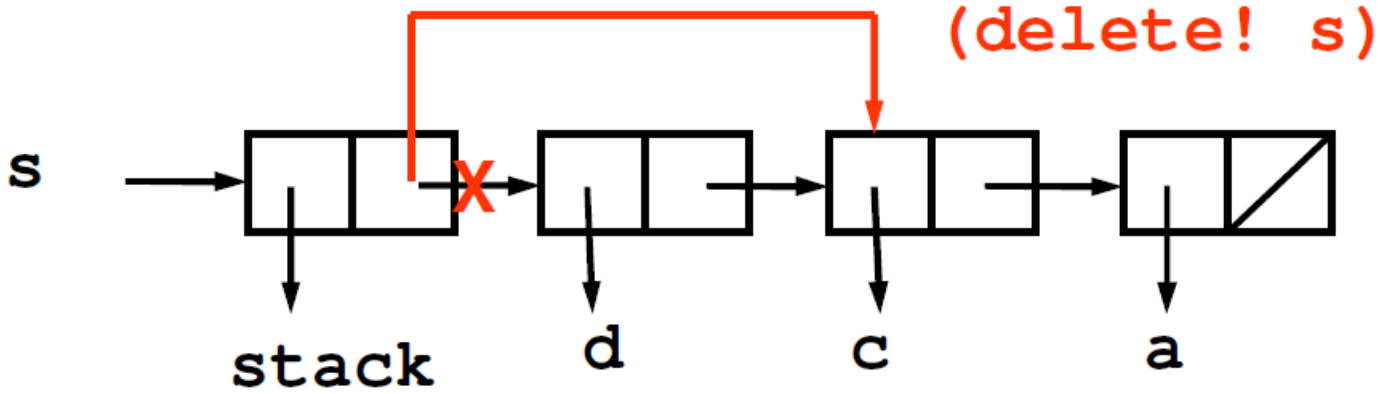
- Стек легко реализовать как мутируемый список с заглавным звеном.

Стек. Реализация

```
(require scheme/mpair) ; обязательно для мутируемых пар
(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 e (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"))
```

Вернёмся к счетам

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

> (define w1 (make-withdraw 100))
> (define w2 (make-withdraw 100))
> (w1 50) -> 50
> (w2 70) -> 30
> (w1 40) -> 10
> (w2 40) -> ”Not enough money”

Счета-объекты

```
(define (make-account balance)
    (define (withdraw amount)
        (if     (>= balance amount)
                (begin  (set! balance (- balance amount))
                        balance)
                (error "Not enough money")))
    (define (deposit amount) (begin
        (set! balance (+ balance amount)) balance)
    (define (dispatch m)
        (cond   ((eq? m 'withdraw) withdraw)
                ((eq? m 'deposit) deposit)
                (else (error "Wrong message"))))
    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

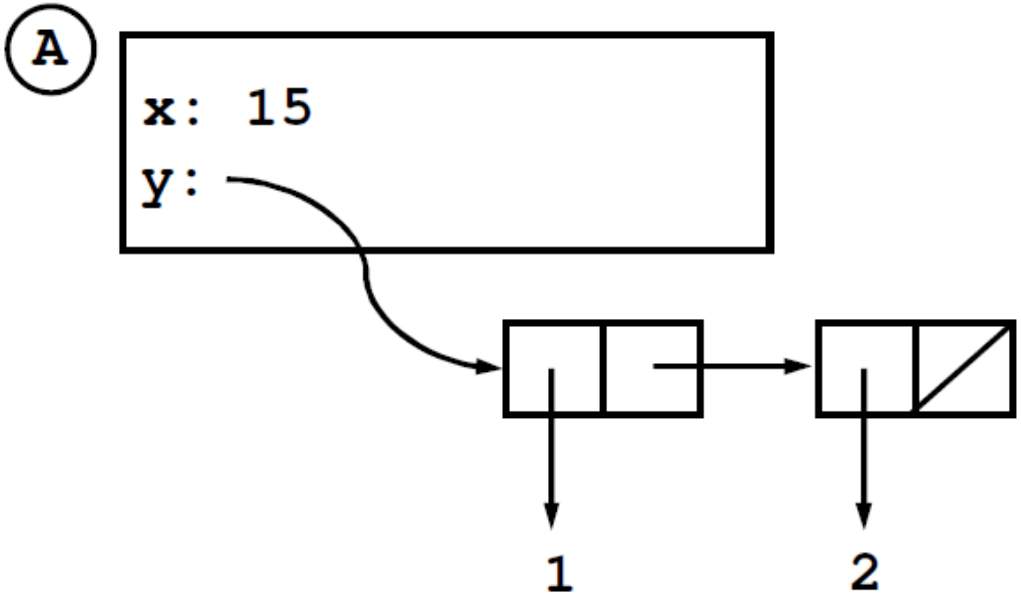
- в подстановочной модели имя – метка для значения
- в МВО имя – место для хранения значения

- 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

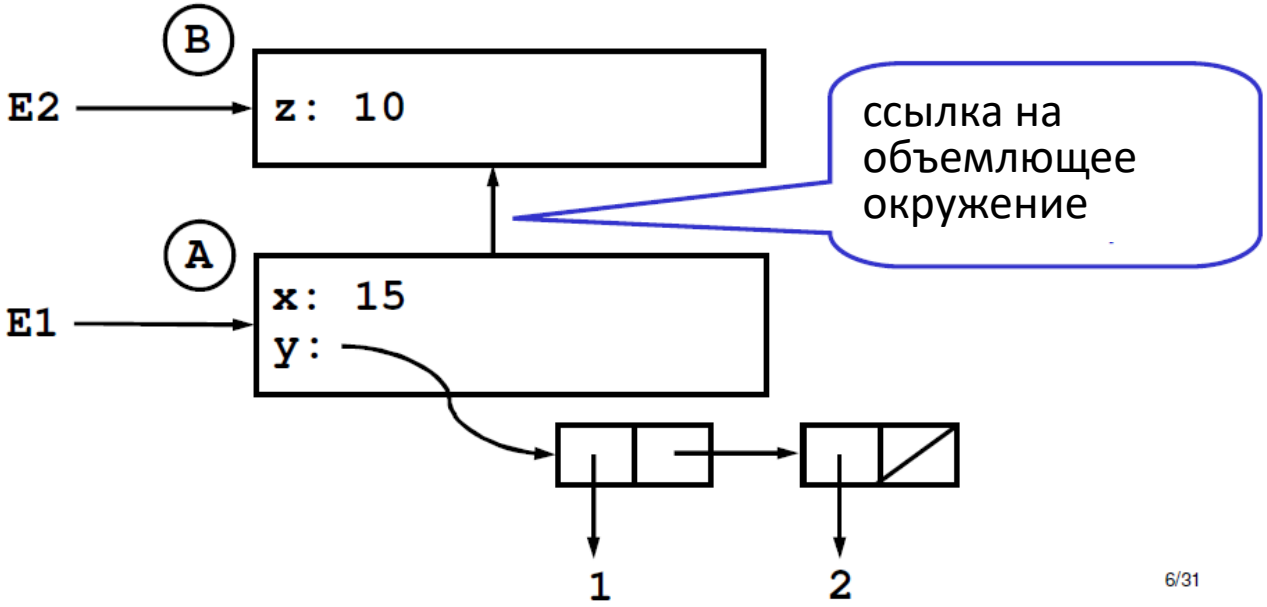
- Кадр – таблица связываний
- Связывание – пара имя : значение
- Пример:



- В кадре А содержатся два связывания
- Звенья списка нарисованы снаружи, поскольку так удобнее.

Элементы МВО

- Окружение – последовательность кадров
- Пример:



- Окружения E1 и E2 разделяют общий кадр B

MBO

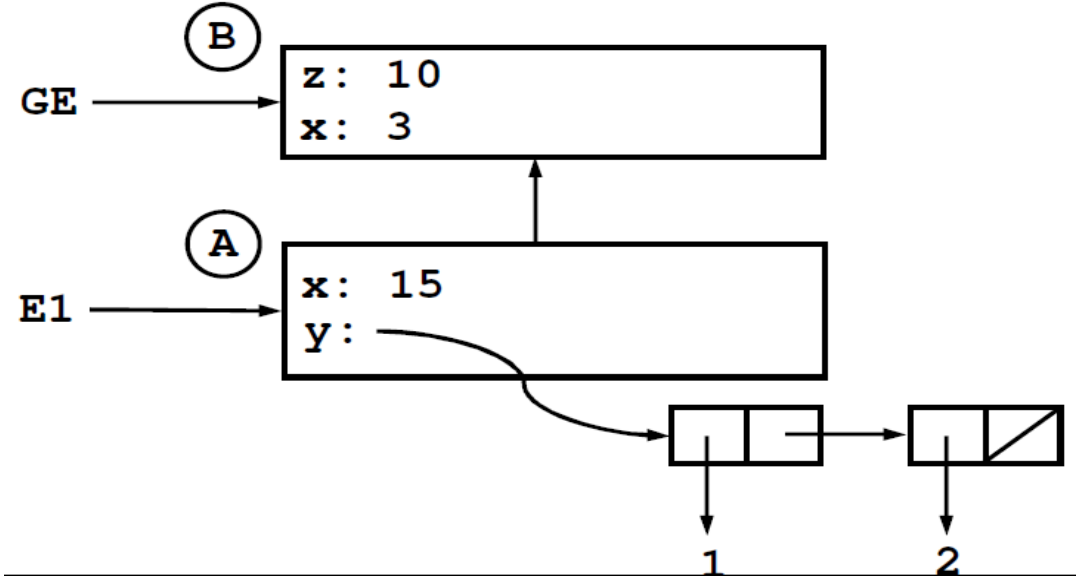
- Кадр может не иметь ссылки на объемлющее окружение, если это кадр глобального окружения.
- Любое вычисление происходит в некотором окружении.
- Для вычисления вызова функции, определённой в коде (пользовательской), строится новое окружение на базе текущего.
- Правила вычислений:
  - Чтобы вычислить комбинацию, где на первом месте примитивная (не пользовательская) функция, следует вычислить все её части и применить первую часть к остальным.

    ↑ ↑ ↑ порядок вычисления частей не определён!

МВО. Правила вычислений

Вычисление имени x в окружении E1:

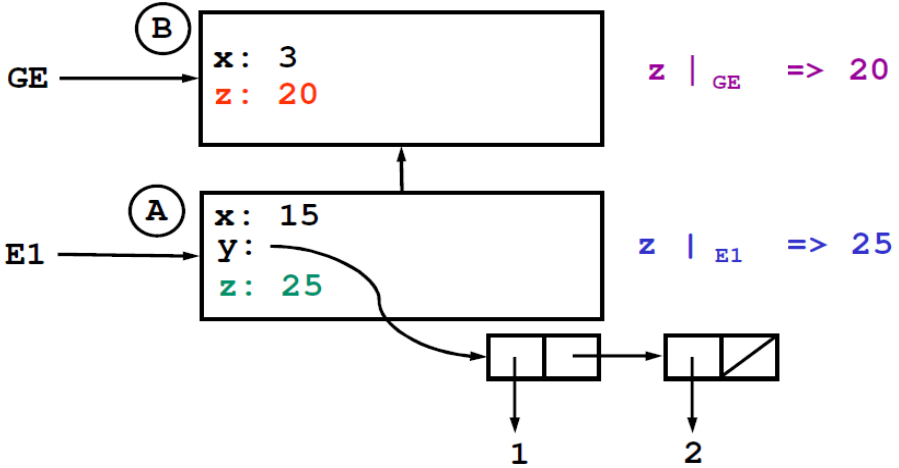
- Найти первый кадр, в котором есть связывание для x.
- Взять в качестве результата значение из этого связывания.



- Остальные связывания x *скрыты*.

МВО. Правила вычислений

Вычисление (define <имя> <выражение>) в окружении:
добавить новое связывание для <имя> в первый кадр окружения,
если связываний имени там не было;
если в кадре было связывание, то ...*выдать ошибку*.



(define z 20) $|_{GE}$

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

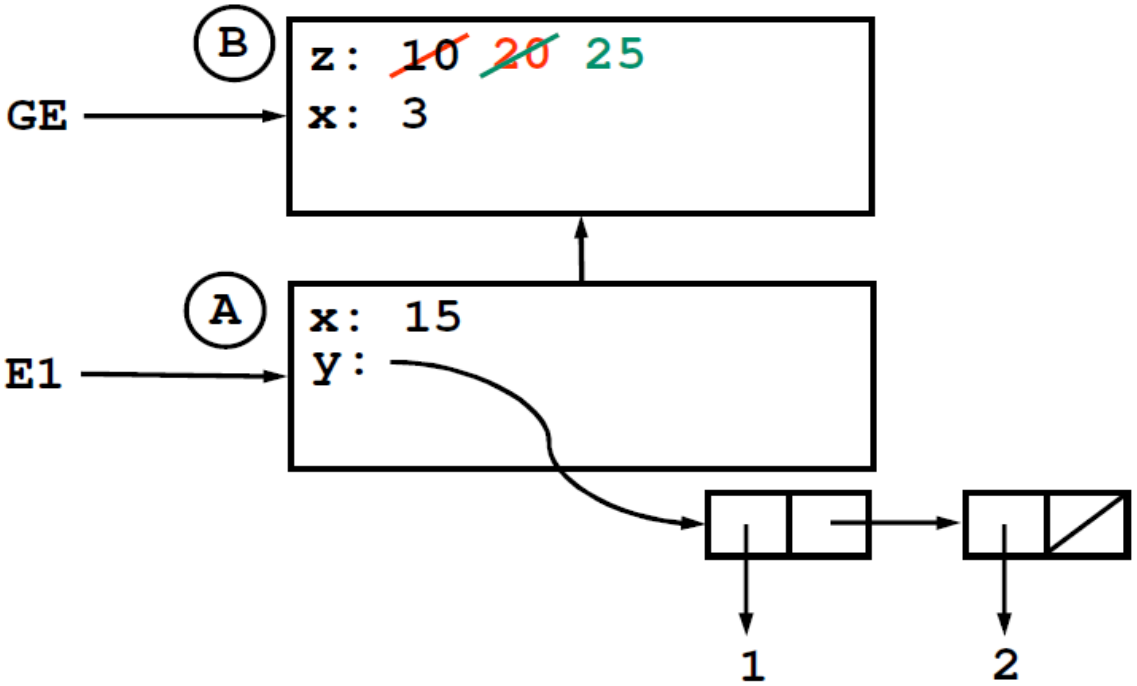
define в глобальном окружении и в окружении E1

МВО. Правила вычислений

Вычисление (set! <имя> <выражение>) в окружении:
изменить связывание имени в том кадре окружения, где оно впервые встретится;
если <имя> нигде не встретилось, *выдать ошибку*!

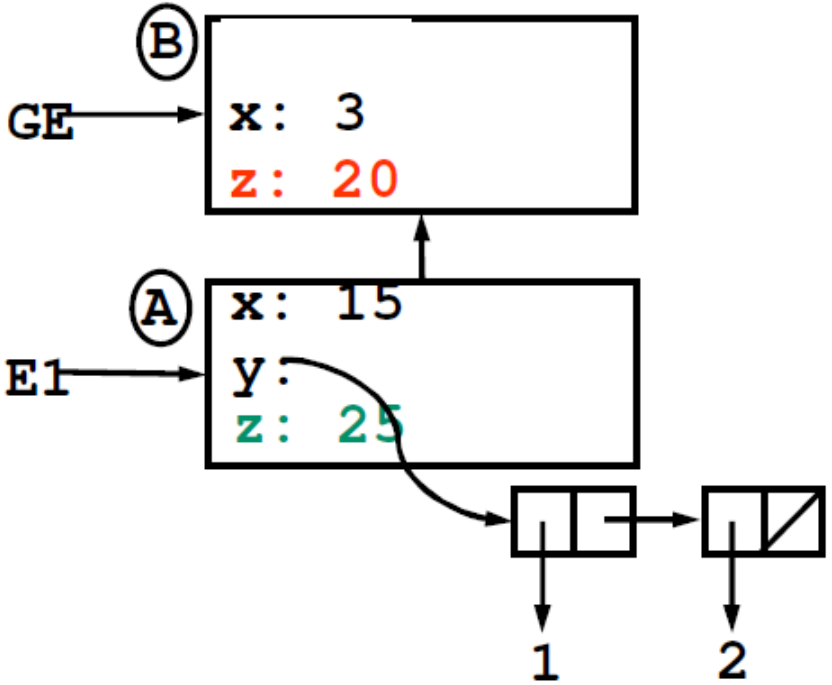
(set! z 20) |$_{GE}$

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

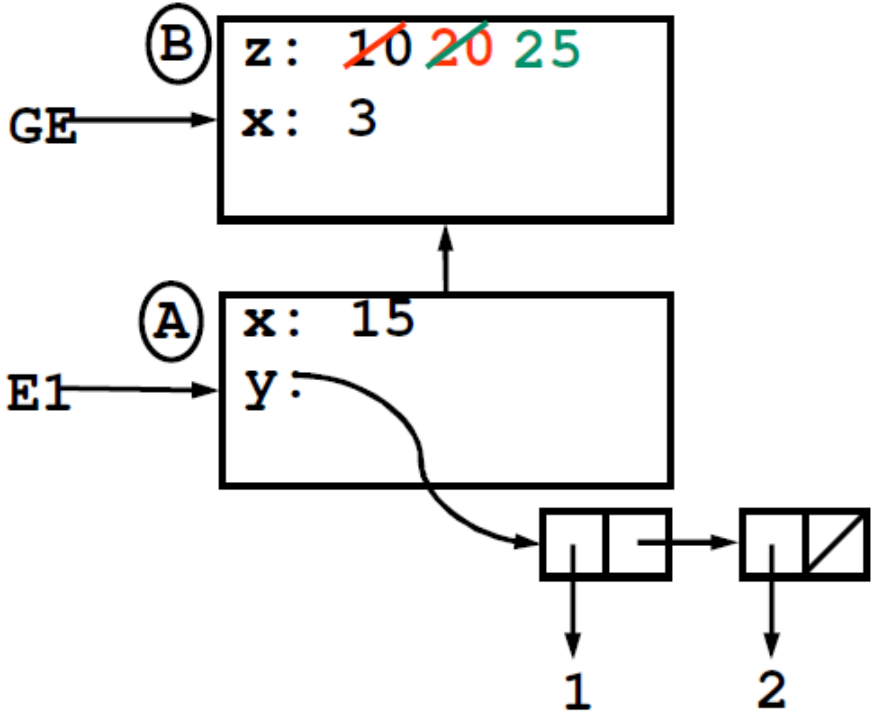


define и set! вычисляются не одинаково

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



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

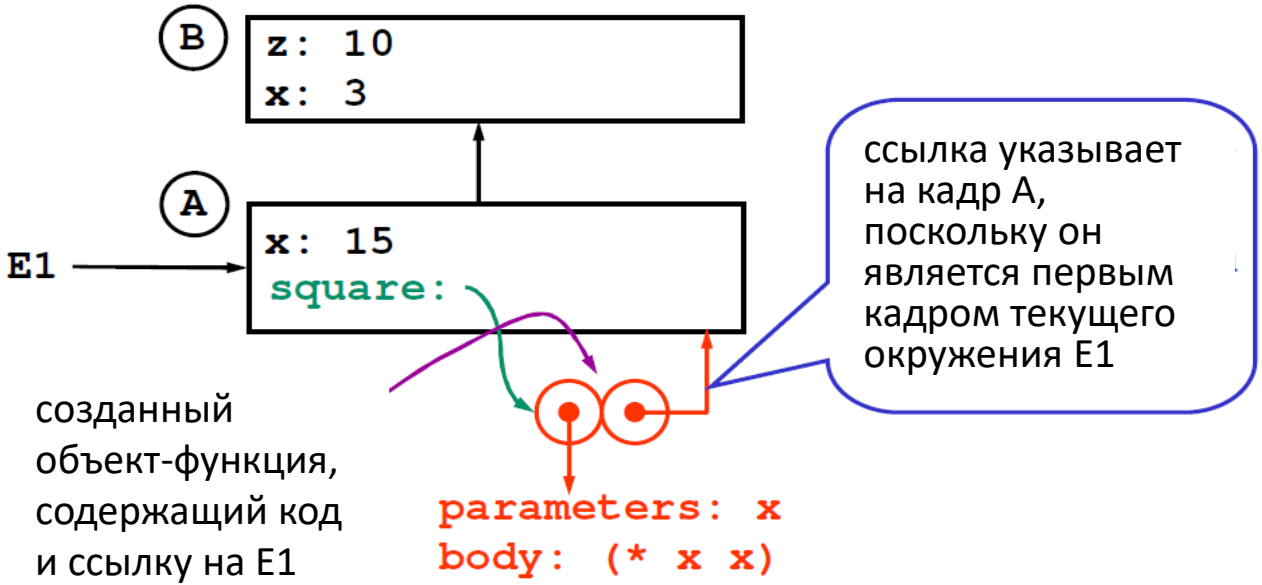


МВО. Правила вычислений

Вычисление lambda в окружении:

создать функцию с телом из lambda и ссылкой на 1й кадр текущего окружения

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



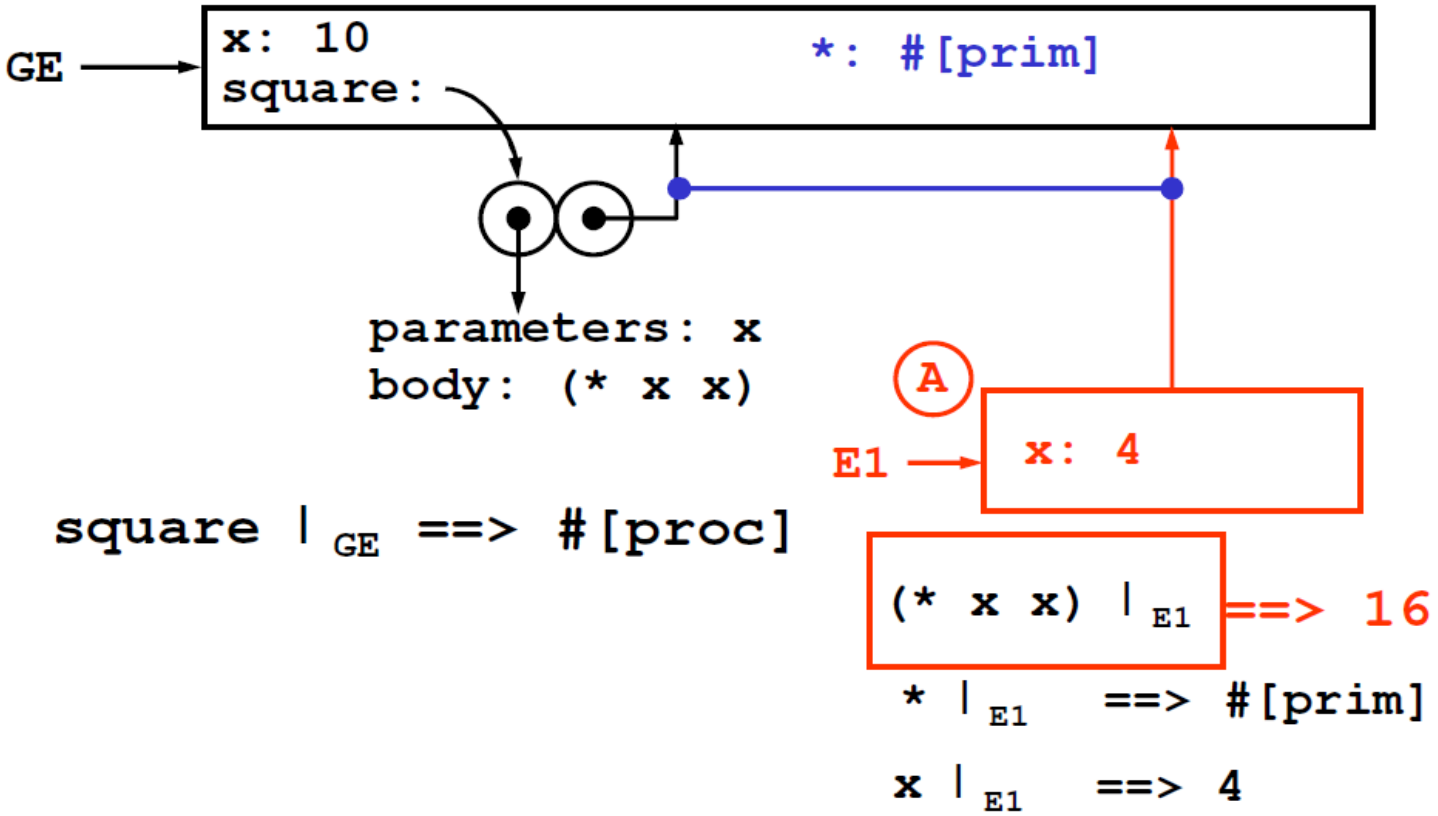
МВО. Правила вычислений

Вычисление вызова непримитивной (определённой в коде, пользовательской) 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.

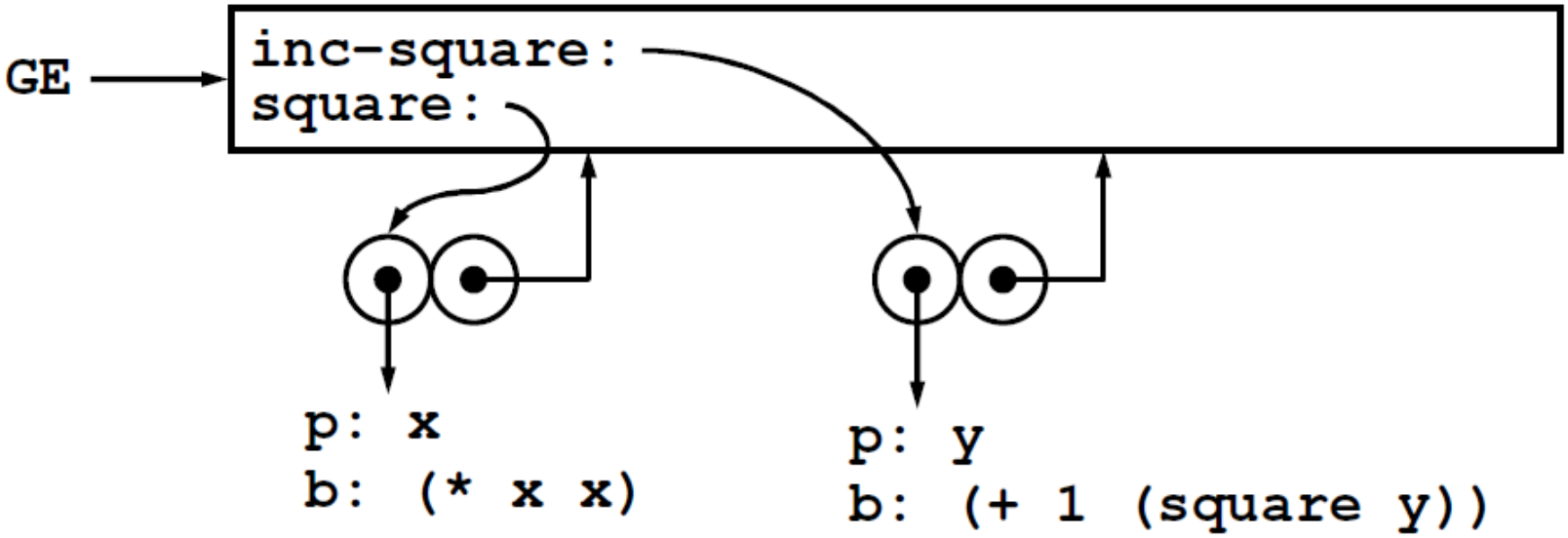
Пример применения функции

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



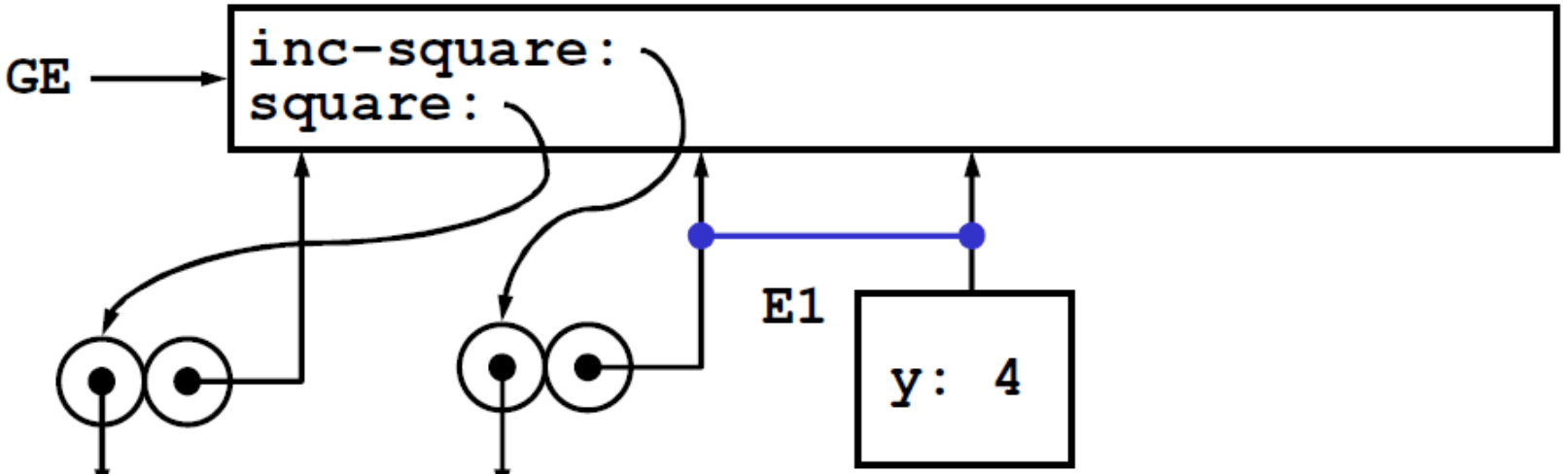
Ещё пример

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



Продолжение примера

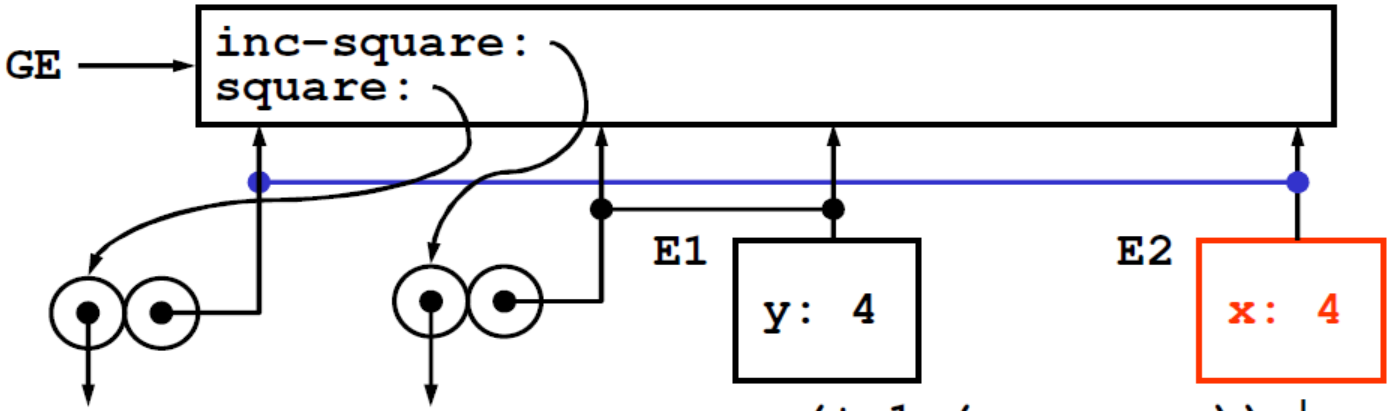
(inc-square 4) $|_{GE}$



inc-square $|_{GE}$ ==> #[proc]

Окончание примера

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



square $|_{E1}$ ==> #[proc]

y $|_{E1}$ ==> 4

(* x x) $|_{E2}$ ==> 16

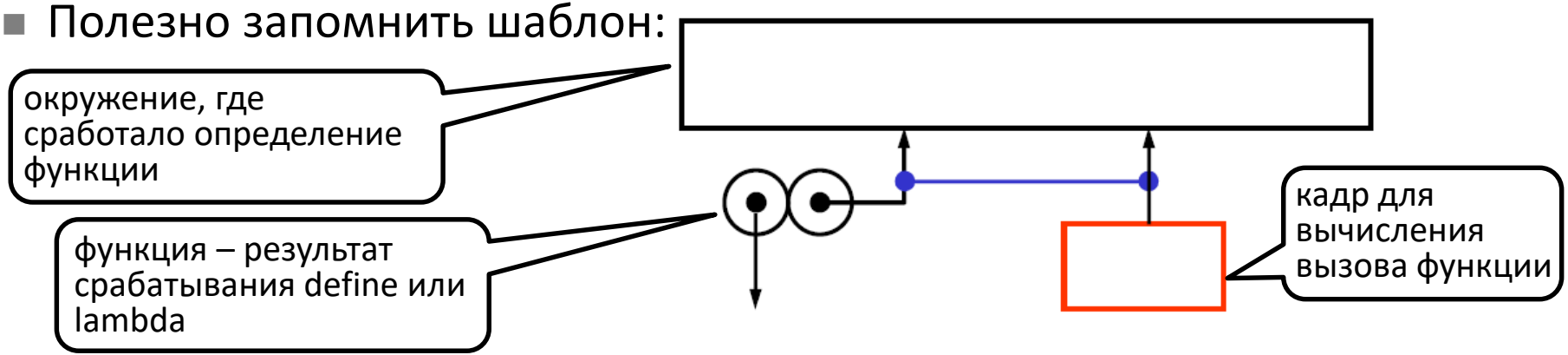
(+ 1 16) ==> 17

* $|_{E2}$ ==> #[prim]

x $|_{E2}$ ==> 4

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:



Забыли про let?

let вычисляется также как lambda, поскольку

```
(let ((<var1> <exp1>)
      (<var2> <exp2>) ...
      (<varn> <expn>)) <body>)
```

это то же, что

```
((lambda (<var1> ...<varn>)
                              <body>)
      <exp1> ...
      <expn>)
```

Забыли про define для функций?

define для функции вычисляется по правилу define для имени и правилу lambda, поскольку

(define (<name> <var1> ...<varn>) <body>)

это то же, что

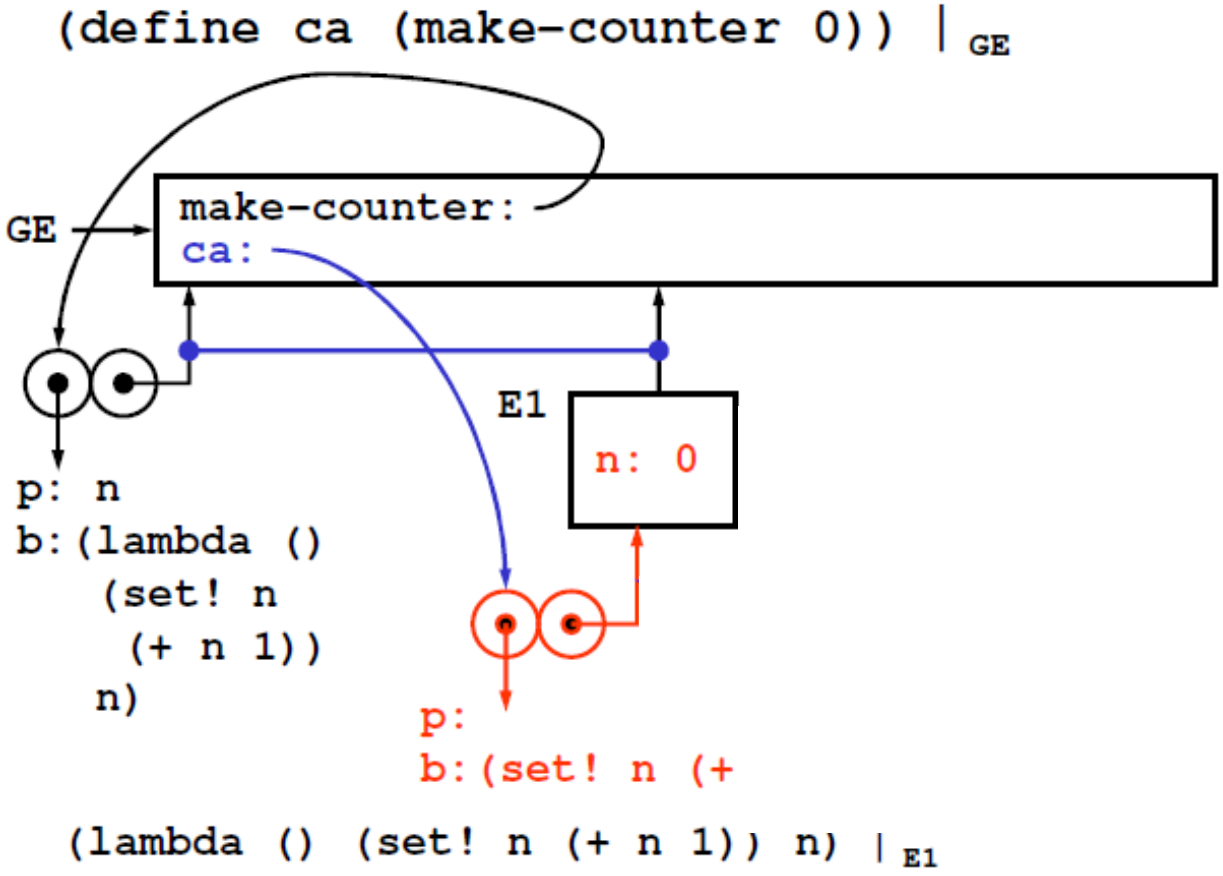
```
(define <name>
        (lambda (<var1> ...<varn>) <body>)
)
```

Снова пример

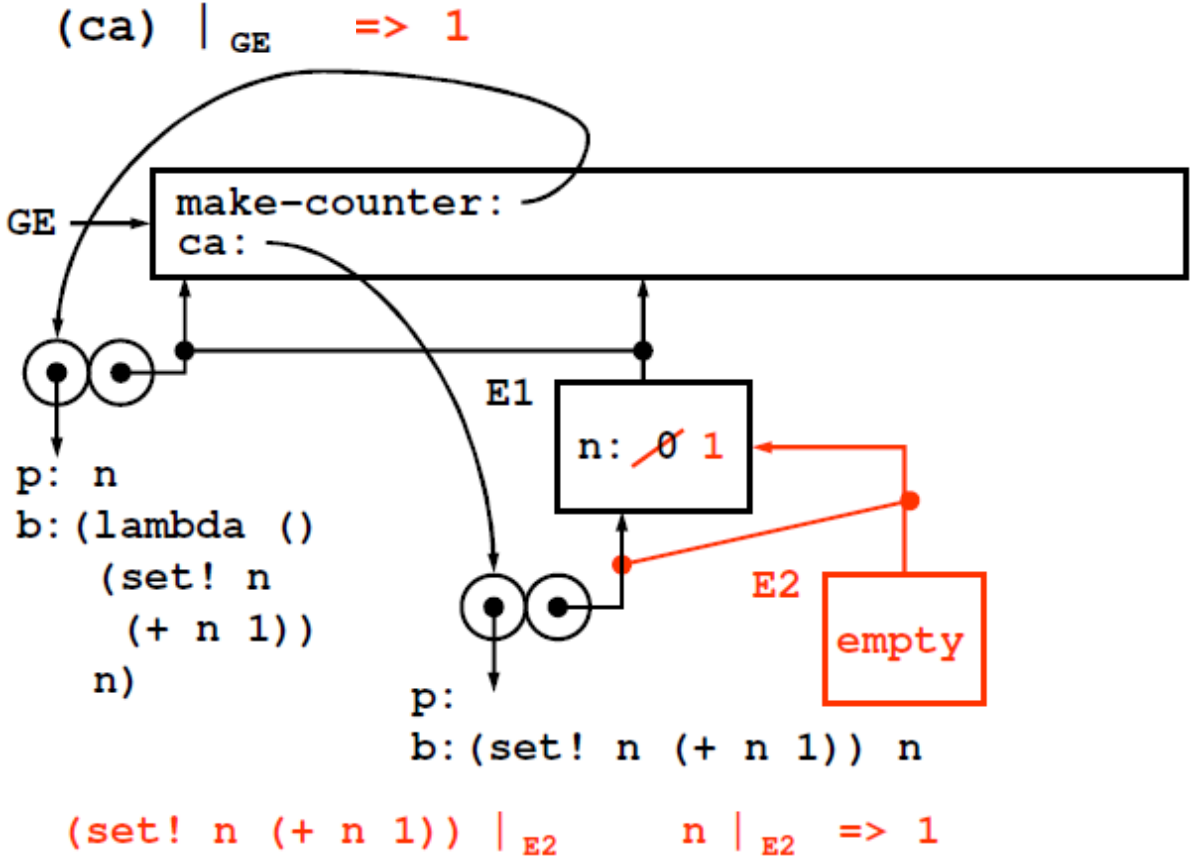
- Счётчик

```
(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
```

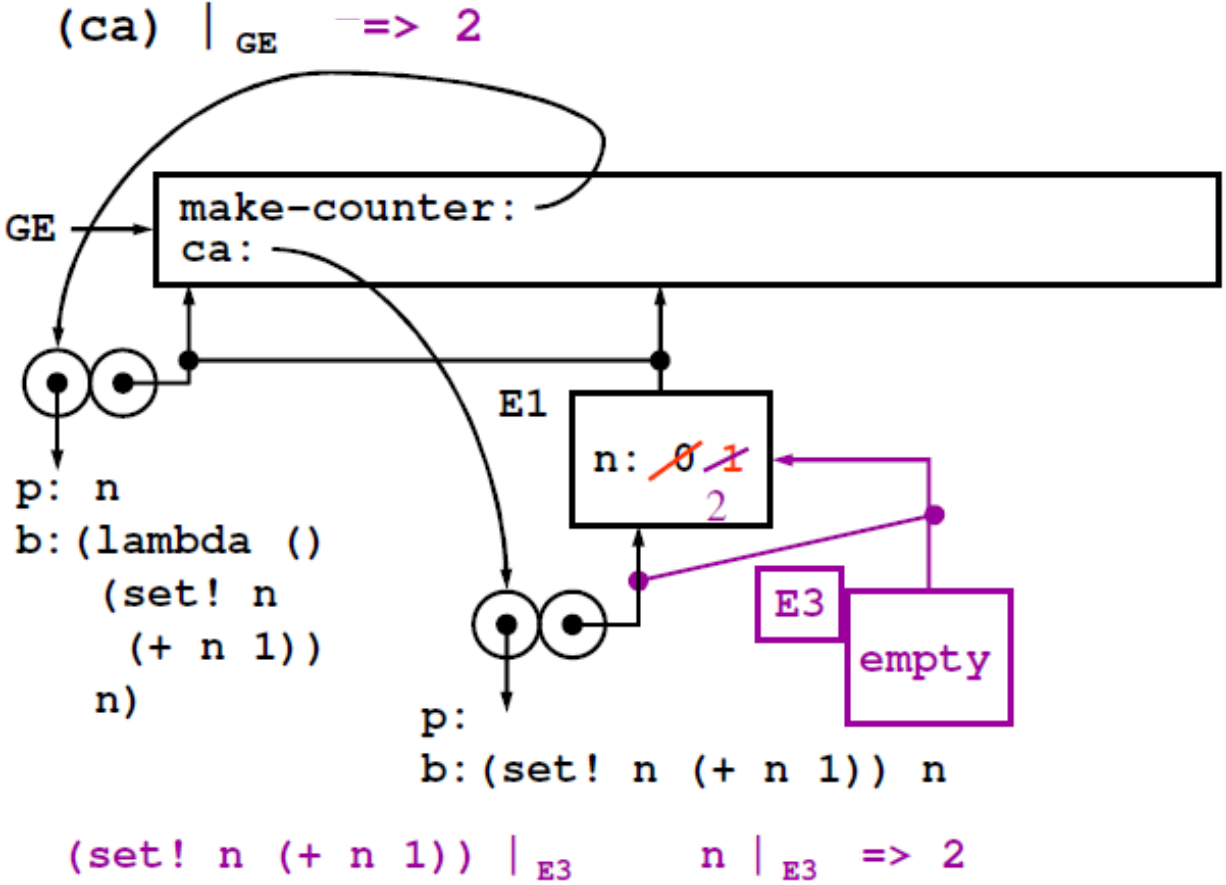
Счётчик. Продолжение



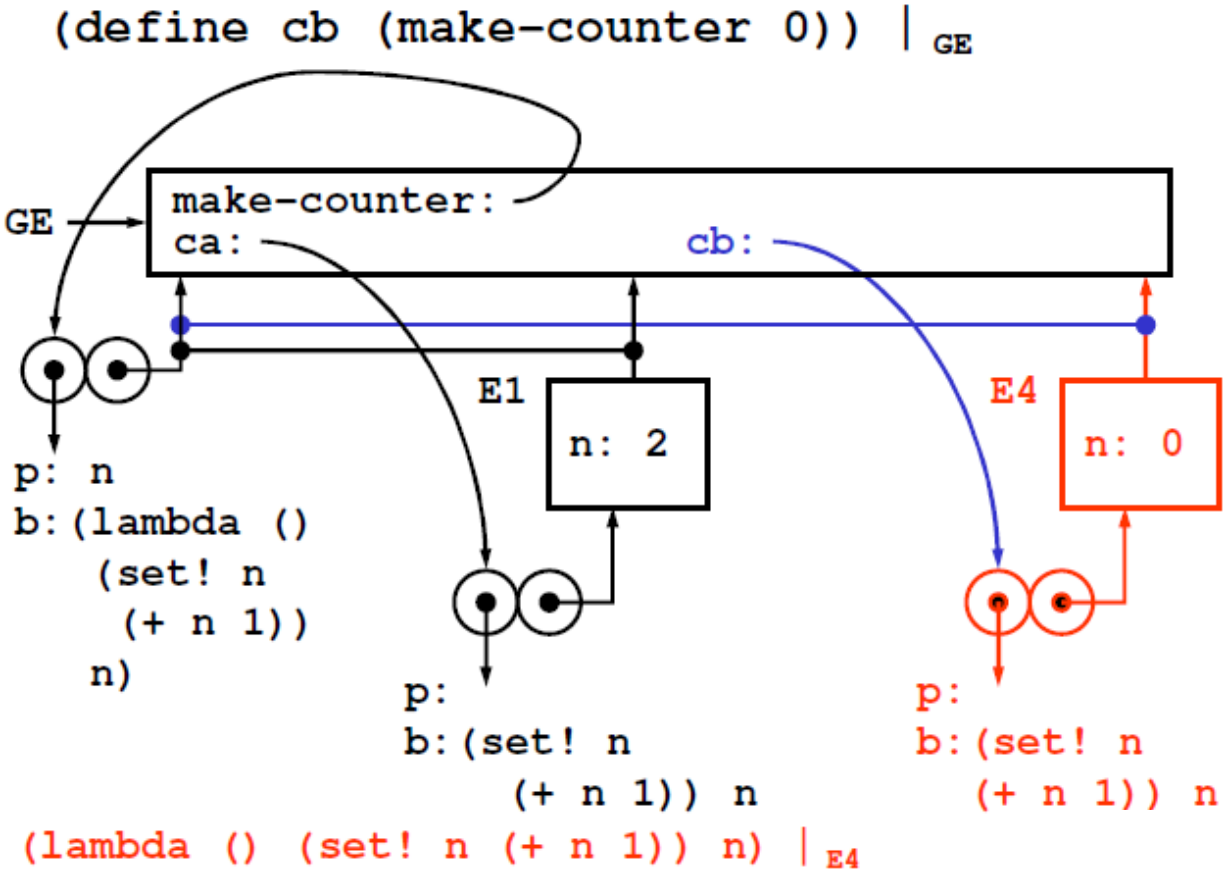
Счетчик. Продолжение



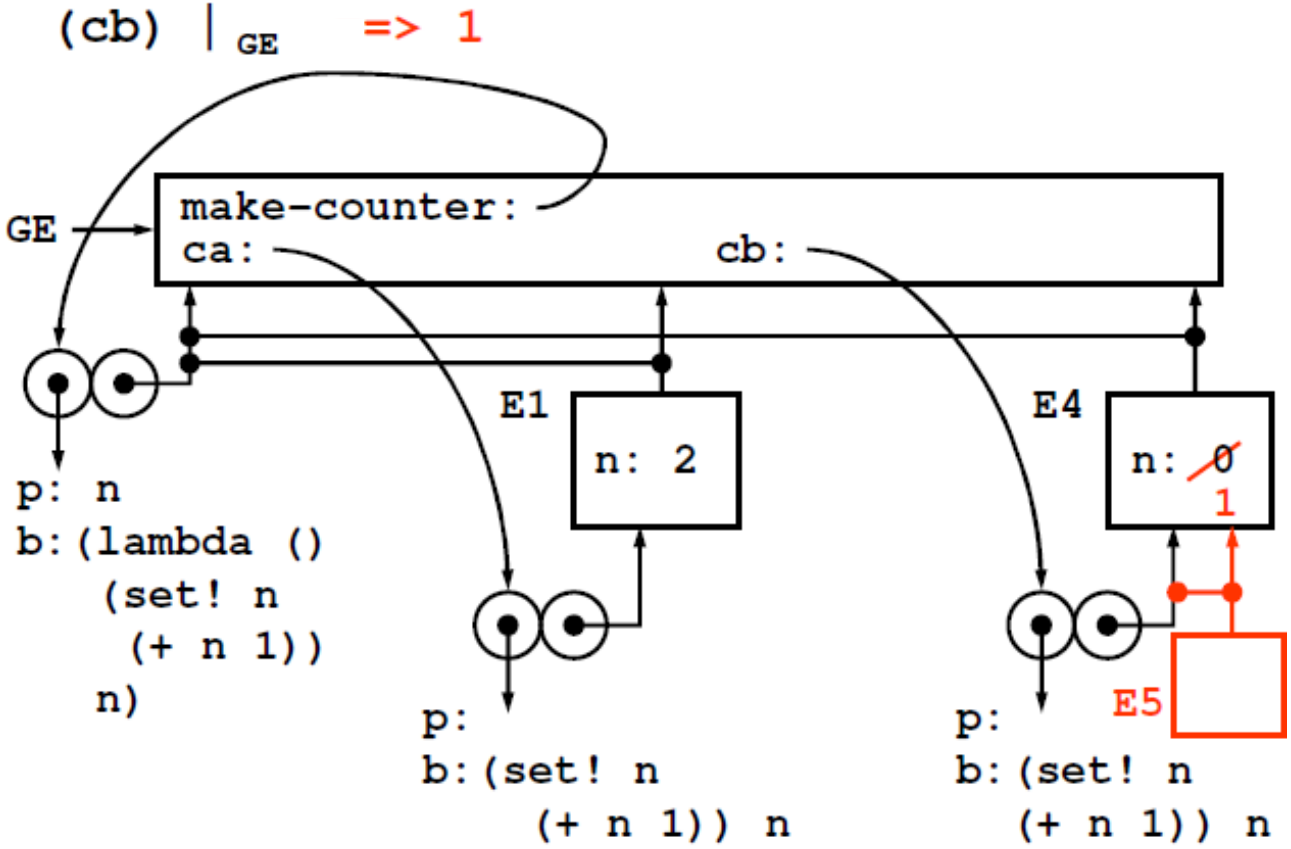
Счетчик. Продолжение



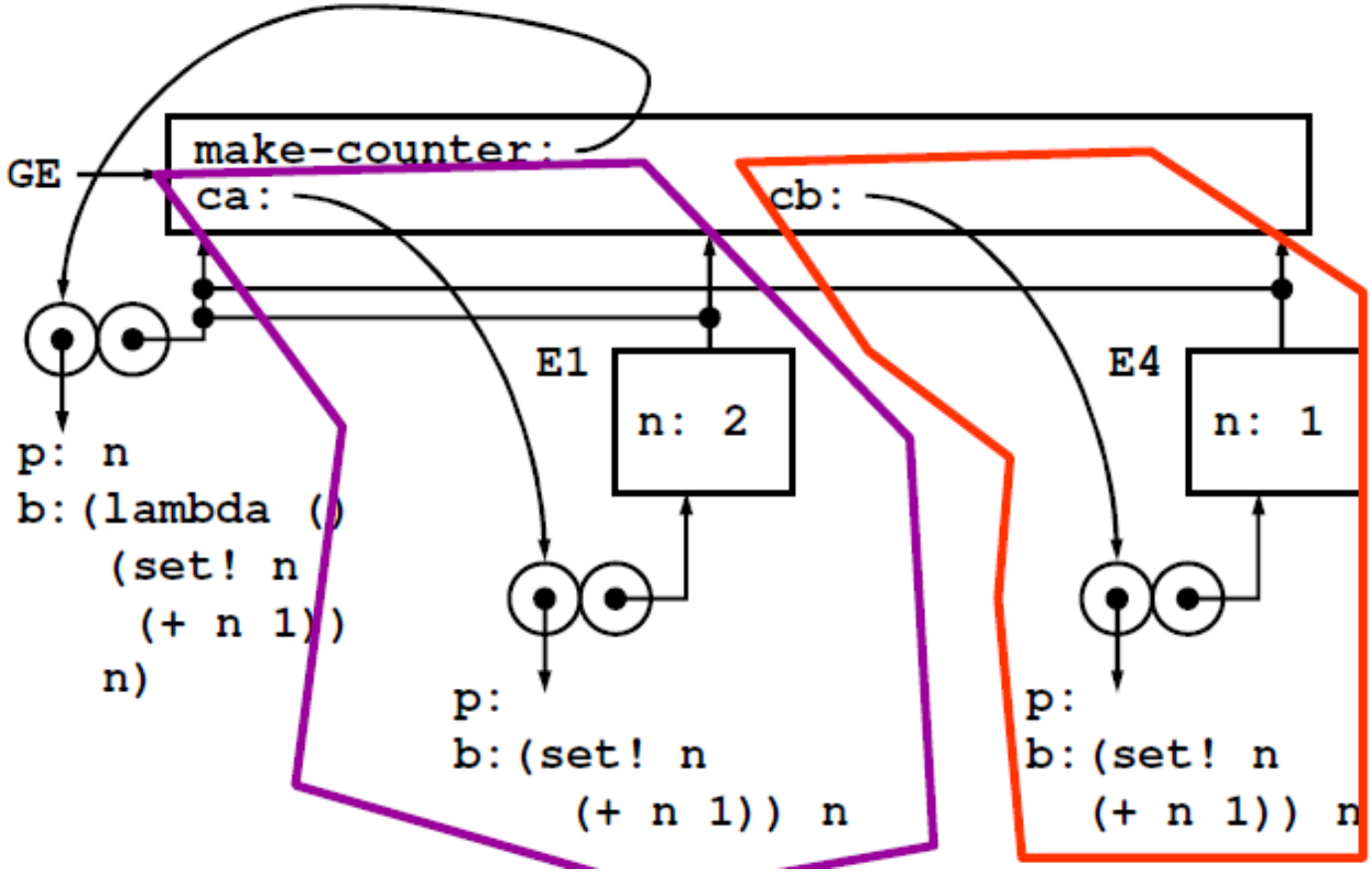
Счетчик. Продолжение



Счетчик. Продолжение



Счетчик. Продолжение



Мутируемые списки

- подключаем модуль (require scheme/mpair)
- конструктор мутируемого списка (mlist <e1> … <eN>)
- внешнее представление (mlist 1 2 3) -> {1 2 3}
- ' (апостроф) и quote «не работают» (mlist? '{1 2 3}) -> #f
- селектор (mlist-ref <lst> <pos>)
- длина (mlength <lst>)
- манипуляции mappend и mappend! / mreverse и mreverse!
- метафункции mmap и mfor-each
- поиск элемента mmember
- поиск в ассоц. списке по equal? (massoc <key> <alist>)

(massoc 'x (mlist '(w 1) '(x 2) '(y 3) '(z 4))) -> (x 2)

- поиск в ассоц. списке по eq? (massq <key> <alist>)
- поиск в ассоц. списке по предикату (massf <pred> <alist>)
- конвертеры mlist->list и list->mlist

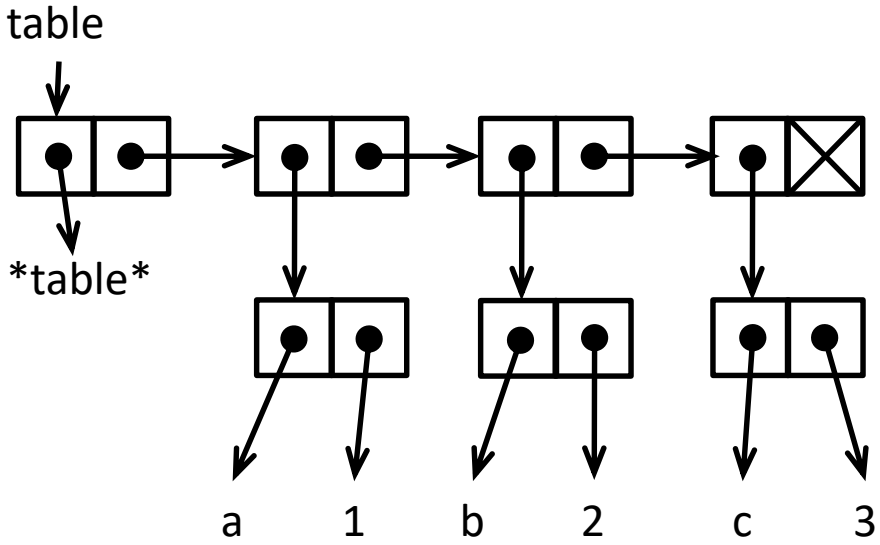
Таблица

- конструктор (make-table)
- селектор (lookup <table> <key>)
- мутатор (insert! <table> <key> <value>)
- внутреннее представление м-списком пар с загл. звеном:
  {*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))))))
```

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
> table -> {*table* {a . 1} {b . 2} {c . 3}}
```



Мемоизация (табуляризация)

- Идея: функция будет запоминать вычисленные результаты.
- Перед тем как вычислить функция проверит, нет ли результата среди запомненных.
- Хранить можно в таблице.
- Вычислить > (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. Пример

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

Выигрыша по сравнению с эффективным итеративным расчётом *нет*.
Есть *проигрыш по памяти*. Выигрыш есть по сравнению с экспоненциальным рекурсивным расчётом.
Есть готовые таблицы в Racket:
конструктор (make-hash <alist>)
> (make-hash '((a . 1) (b . 2) (c . 3))) -> #hash((a . 1) (b . 2) (c . 3))
селектор (hash-ref <hash> <key> <failval>)
мутатор (hash-set! <hash> <key> <val>)

Мемоизация. Пример оправданного использования

Допустим, необходимо получать значения прайморialов не превышающих $2 \bullet 10^{22}$. Известно, что последовательность растёт экспоненциально. В диапазон попадают лишь 18 чисел 1, 2, 6, 30, 210, 2310, 30030, 510510, 9699690, 223092870, 6469693230, 200560490130, 7420738134810, 304250263527210, 13082761331670030, 614889782588491410, 32589158477190044730, 1922760350154212639070.

```
(define primorial-vect (vector 1 2 6 30 210 2310 30030 510510 …
1922760350154212639070)
(define (primorial n) (vector-ref primorial-vect n))
> (primorial 5) -> 210
```

результат получаем за O(1). Это быстрее, чем считать каждый раз с самого начала. Расходы на память не велики.

«Мемоизирующая» функция

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))))))))
```

Мутируемые векторы

- Вектор – массив (фиксированного размера коллекция значений с доступом по индексу)
- конструктор (make-vector <size> <value>)
- ещё конструктор (vector val_0 val_1 val_2 … val_N)
- 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)

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

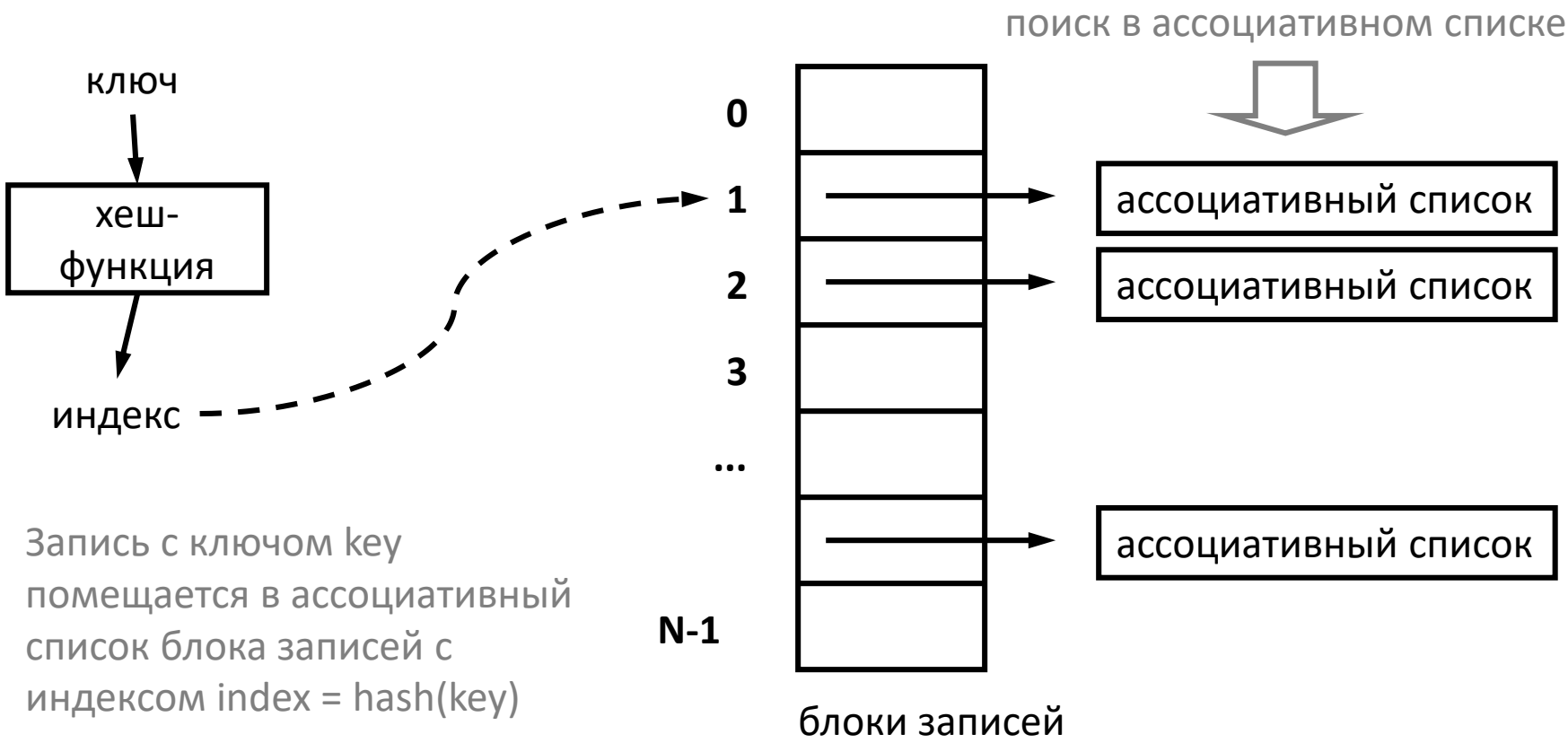
Мутируемые векторы

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

С помощью вектора реализуем хеш-таблицу:

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

Хеш-таблица «на коленке»



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)))
  (begin (let loop ((i (sub1 size))) (if (> i -1)
     (begin (vector-set! buckets i (make-table)) (helper (sub1 i))) #t))
     (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)))
```

Пример работы хеш-таблицы

```
> (define t7 (make-hash-table 7 remainder))
> (insert-hash-table! t7 2 'a)
> (insert-hash-table! t7 1 'ab)
> (insert-hash-table! t7 100 'abc)
> 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:
  - оно требует новую модель описания вычислений вместо подстановочной модели – *модель вычислений с окружениями*;
  - оно приводит к возникновению сторонних эффектов;
  - оно делает значимым порядок вычисления подвыражений.
- Мемоизация иногда позволяет улучшить работу функции без её существенного переписывания. Иногда – *не позволяет*, а лишь напрасно тратит память. Выгода от мемоизации получается лишь тогда, когда честное вычисление стоит дороже, чем поиск в таблице.