ВВЕДЕНИЕ В ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Малышко Виктор Васильевич
e-mail: victor@sp.cs.msu.ru
vvmalyshko@gmail.com

Содержание курса

- Раздел 1. Построение абстракций на основе функций и структур данных
- Раздел 2. Дополнительные возможности языка программирования Scheme
- Раздел 3. Математические основы функционального программирования

Раздел 1:

- Тема 1. Основные сведения о языке Scheme
- Тема 2. Рекурсия и итерация
- Тема 3. Функции высшего порядка
- Тема 4. Структуры данных
- Письменная контрольная работа

Раздел 2:

- Тема 5. Присваивание. Модель вычислений с окружениями
- Тема 6. Объектно-ориентированное программирование в Scheme
- Тема 7. Макросы в Scheme

Раздел 3:

- Тема 8. Основные сведения о λ-исчислении

Основная литература

- Абельсон Х., Сассман Дж. Структура и интерпретация компьютерных программ. – М.: Добросвет, КДУ. 2010 newstar.rinet.ru/~goga/sicp/sicp.pdf
- Харрисон Дж. Введение в функциональное программирование. – Новосибирск. 2009. https://goo.gl/g7E6pl
- Чернов А. В. Учебное пособие по заданию «Доктор». – М.: ВМК МГУ. 2006 ejudge.ru/study/5sem

Дополнительно

- Веб-страница книги SICP на сайте MITPress: mitpress.mit.edu/sicp/
- Веб-страница курса SICP на сайте MITOpenCourseware: goo.gl/hO3nFs
- Курс Е. П. Кирпичева compsciclub.ru/courses/fprog
- Видеозапись лекций Е. П. Кирпичева www.lektorium.tv/course/22779
- Felleisen M., Findler R. et al. How to Design Programs: An Introduction to Computing and Programming. 2003 www.htdp.org

ЛЕКЦИЯ 1
Основные сведения о языке Scheme

Аргументы в пользу функционального программирования

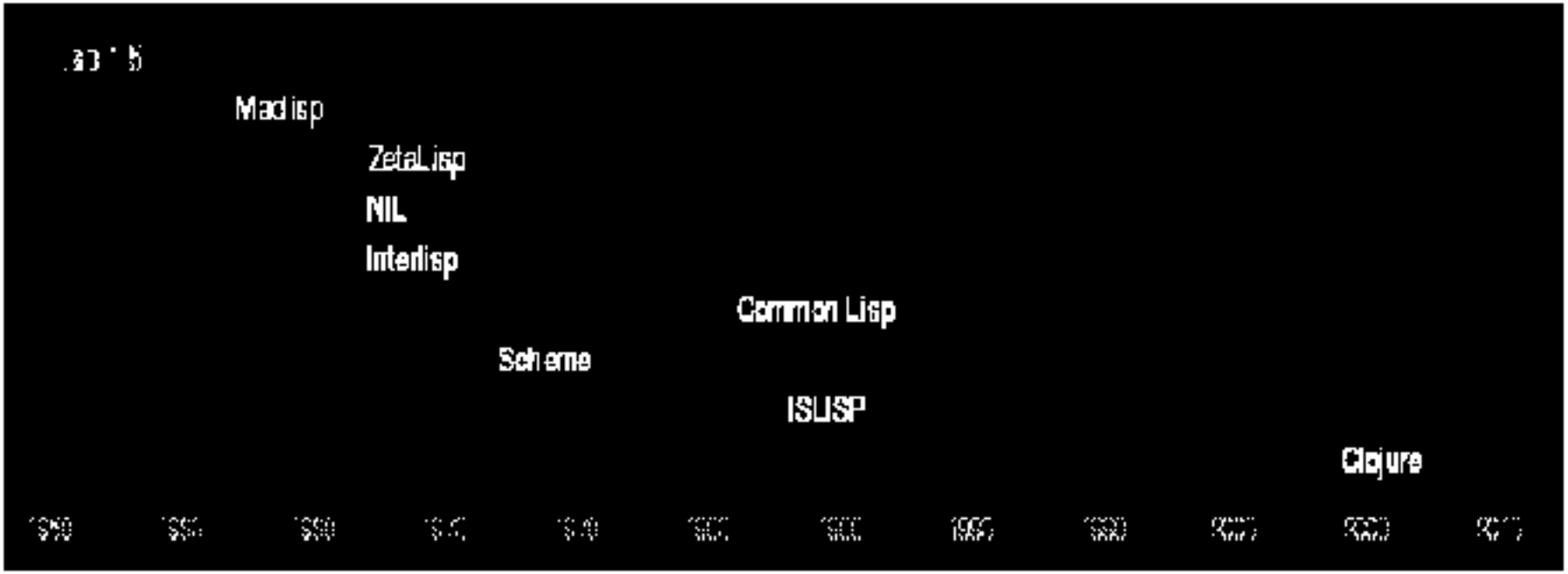
- Функциональные программы легко писать.
- Функциональные программы короче императивных.
- Функциональные программы легче понимать и анализировать.
- Модульность – естественное свойство функциональных программ.
- Функциональные языки удобны при решении задач ИИ.

Исторический экскурс

Джон Маккарти (1927-2011) MIT

В 1958 году создал язык LISP (*LIS*t *P*rocessing language)

Диалекты языка Lisp



среди основных диалектов выделим

- Common Lisp -- ANSI INCITS 226-1994
- Scheme -- IEEE 1178-1990

Scheme

- Язык создавался в MIT в период 1975-1980 гг.
- Авторы:

Джеральд Сассман

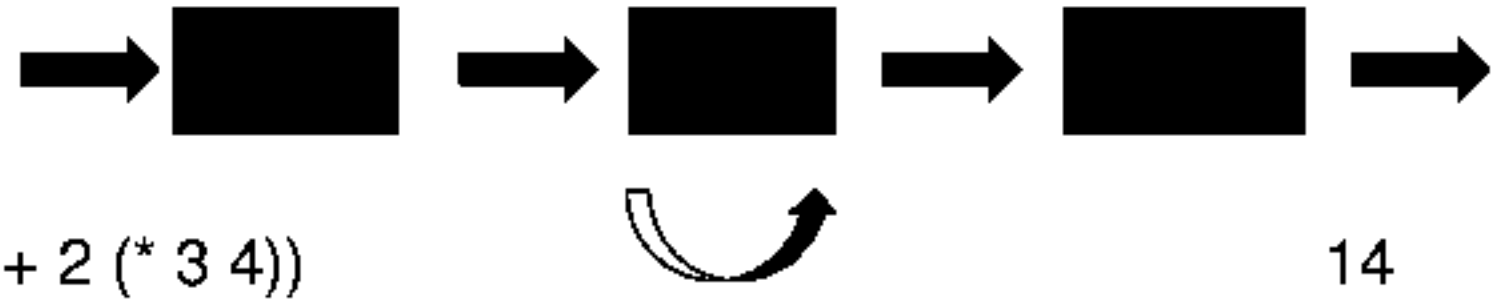
Гай Стил

Отличия Scheme от обычных (императивных) языков программирования

- Основой является не фон-Неймановская архитектура, а λ-исчисление.
- Программирование в декларативном стиле: не «как программа должна делать», а «что программа должна делать».
- Скобочные выражения, польская [неинверсная] запись + скобки (+ 2 (* 3 4))
- Программа является последовательностью вызовов функций друг из друга.

Отличия Scheme от обычных (императивных) языков программирования

- Данные и функции представляются одинаково.
- Функции – «объекты первого класса» (могут передаваться как параметры, возвращаться как результаты, быть значением или частью сложного значения).
- Выполнение программы –
  прочитать -> вычислить -> вывести



(+ 2 (* 3 4))

14

Отличия Scheme от обычных (императивных) языков программирования

- Автоматическое управление памятью.
- Управляющая структура программы -- рекурсия.
  - нет циклов
  - нет переменных
  - нет присваиваний [почти]
- Динамическая типизация.

Отличия Scheme от диалектов Lisp

- Минималистичный язык
- Точная арифметика
- Ленивые вычисления
- Не различает «регистр» вне строк и char'ов ...

Знакомство со Scheme
Имена и окружение

- Идентификаторы

  x x->y name#

  не могут включать в себя разделители ( ) ; " ' ` | [ ] { } или начинаться с # или ,
- Связать имя и значение позволяет define

  (define size 2)

  (define dblsize (+ size size))
- define – специальная форма, *вычисляющая* значение 2-го аргумента и связывающая его с 1-м
- Окружение – место, где хранятся связывания. Новое окружение создается из старого при добавлении связываний. Если добавляется новое связывание имени, то старое связывание *затеняется*.

Знакомство со Scheme
Внешнее представление

- Любое значение имеет внешнее представление, то есть, запись в виде последовательности символов. Интерпретатор выводит внешние представления значений в ответ на запросы.

```
(define size 2)     =>
size                =>    2
(* size 5)          =>    10
(= size 2)          =>    #t
```

- У функций *нестандартное* внешнее представление.

```
+                   =>    #<procedure:+>
```

- Внешнее представление считывается read и выводится print или display

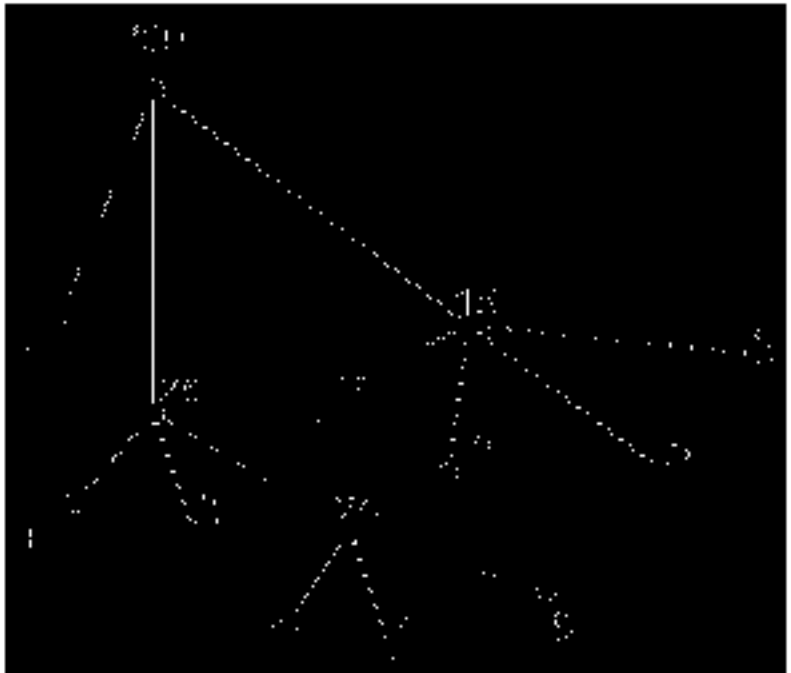
Знакомство со Scheme
Выражения

- Литералы
  - Литеры #\a #\A #\newline #\space
  - Числа -1 1/6 10.005 #o777 2-i
  - Булевы значения #t #f
  - Строки "Hello \"world#\\"
  - Символы, «цитаты» (quote (+ 1 2)) или 'xyz
  - Пустой список '() или null
- Имена xyz x->y
- Спецформы (define и т. п.)
- Вызовы функций

Выражения (продолжение)
Вызовы функций (или комбинации)

- Запись комбинации: (c_1 c_2 … c_n)
- Вычисление комбинации:
  a) найти значения всех c_i *(стандарт не определяет порядок вычисления c_i)*
  b) применить функцию, являющуюся значением c_1 к значениям остальных c_i
- Правило вычисления комбинаций рекурсивно.
- Комбинация в виде дерева

(* (+ 2 (* 4 6)) (+ 3 5 7))



- Спецформы – не комбинации!

Знакомство со Scheme

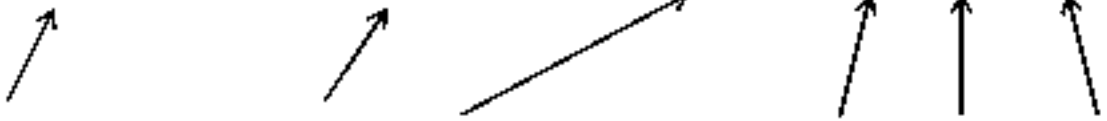
«Свои» функции

- Определение своей функции даётся спецформой define

  (define (<имя> <параметры>) <тело>)

- Пример

```
(define (square x) (* x x))
```



определяем квадрат x как умножение x на x

- После определения функцию можно использовать

```
(square 10)                          =>   100
(+ (square 3) (square 4))            =>   25
(define (sum-of-squares x y) (+ (square x) (square y)))
```

«Свои» функции (продолжение)

```
(define (f a)
      (sum-of-squares (+ a 1) (* a 2)))
(f 5)                    => 136
```

- Представить, как идут вычисления помогает *подстановочная модель* (замена вызова телом)

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

- Результат тот же, но интерпретатор может работать *иначе*.

«Свои» функции (продолжение)

- Рассмотрим способ вычисления в *нормальном порядке* (полная подстановка, затем редукция)

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

подстановка

редукция

- При *нормальном порядке* пока аргумент не понадобится, он не вычисляется.
- (+ 5 1) и (* 5 2) считали дважды
- Ранее считали в *аппликативном порядке* («вычисли аргументы, примени функцию»)

«Свои» функции (продолжение)

- Анонимные функции задаются спецформой lambda

(lambda (<параметры>) <тело>)

- Значением спецформы lambda является функция
- Пример:

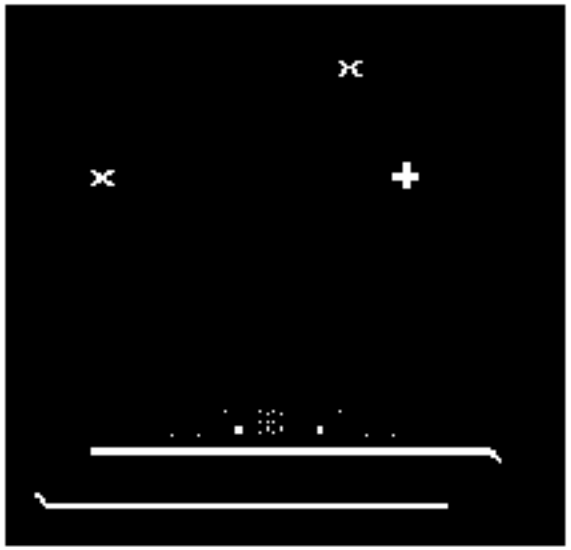
(lambda (x) (+ x x)) => #<procedure>

((lambda (x) (+ x x)) 4) => 8

- (define (<имя> <параметры>) <тело>) на самом деле сокращённо от

  (define <имя>

    (lambda (<параметры>) <тело>))



Условные спецформы

- «Разбор случаев» – cond

  (cond (<p_1> <e_1>) ; p_i – булева функция (предикат)
  (<p_2> <e_2>) ; (<p_i> <e_i>) – i-ая ветвь
  … (<p_n> <e_n>)) ; e_i – выражение-следствие

- Вычисляем предикаты по порядку, начиная с 1-го, до тех пор пока не получим p_i = #t.
- Вычисляем e_i. Его значение и будет значением cond.
- В заключительной ветви полезно вместо предиката писать else.
- Если все предикаты ложны, значение cond не определено.
- Пример:

```
(define (new-abs x)
        (cond ((< x 0) (- x))
              (else x)))
```

Условные спецформы (продолжение)

- Условное выражение – if

(if <предикат> ; сначала вычисляется предикат
<следствие> ; если он истенен, считаем следствие
<альтернатива>) ; иначе – альтернативу

- Пример: (define (new-if-abs x)
(if (< x 0)
(- x)
x))

- Для записи предикатов полезны спецформы and и or

(and $<e_1>$ … $<e_n>$) ; вычисляет e_i по порядку, начиная с 1-го, пока не найдёт e_i = #f и не вернёт #f. Иначе, если все Подвыражения истины, то значение and = #t. (and) => #t

(or $<e_1>$ … $<e_n>$) ; вычисляет e_i по порядку, начиная с 1-го, пока не найдёт e_i = #t и не вернёт #t. Иначе – #f.

(or) => #f Можно использовать функцию not

Условные спецформы (продолжение)

- Пример, демонстрирующий разницу между нормальным и аппликативным порядком выполнения:

```
(define (p) (p))                    ; зацикливающаяся функция
(define (test x y)
        (if (= x 0)
            0
            y))
```

- При нормальном порядке вызов (test 0 (p)) вернёт 0

```
(test 0 (p))
(if (= 0 0)  0  (p))
0
```

- При аппликативном порядке получаем зацикливание при вычислении второго параметра (test 0 (p))
- Вопрос: Можно ли if не делать спецформой, а реализовать через cond?

Условные спецформы (продолжение)

- Выражение с вариантами – спецформа case

(case <ключ>

(<$vars_1$> <e_1>) ; <$vars_i$> – (o_{1i}, o_{2i}, … o_{ki})

(<$vars_2$> <e_2>) ; o_{ji} – внешние представления

… (<$vars_n$> <e_n>))

- Вместо <$vars_n$> может быть else.
- Вычисляем <ключ>.
- Сравниваем значение ключа с вариантами первой ветви. Если есть совпадение, вычисляем <e_1> и возвращаем его значение. Иначе берем следующую ветвь и т. д.
- Пример: (case (* 2 3)

((2 3 5 7) 'prime)

((1 4 6 8 9) 'composite)) => composite

Знакомство со Scheme
Спецформа begin

(begin <exp_1>
<exp_2>
... <exp_n>)

- Вычисляет все подвыражения по порядку.
- Значением формы является значение последнего подвыражения.
- Помогает, если нужно сделать ввод/вывод.
- Пример:

```
(begin (println "Input N:")
       (read))
```

Знакомство со Scheme

Числа

- Башня числовых типов:

  number

  complex 1+i

  real 0.001

  rational 1/3

  integer -1 #xff

- Функции проверки типа number? real? ...
- Функции = < > <= >= принимают ≥2 аргументов. То же +-*/
- Деление нацело: quotient, remainder, modulo

(modulo 13 4) => 1 (remainder 13 4) => 1

(modulo -13 4) => 3 (remainder -13 4) => -1

(modulo 13 -4) => -3 (remainder 13 -4) => 1

(modulo -13 -4) => -1 (remainder -13 -4) => -1

Числа (продолжение)

- gcd, lcm (неотрицательный результат)
- floor (ближайшее из не превосходящих)
- ceiling (ближайшее из не меньших)
- truncate (ближайшее из не превосходящих по модулю)
- round («обычное» округление)

(floor -4.3) => -5.0 (ceiling -4.3) => -4.0

(truncate -4.3) => -4.0 (round -4.3) => -4.0

(floor 3.5) => 3.0 (ceiling 3.5) => 4.0

(truncate 3.5) => 3.0 (round 3.5) => 4.0

- exp, log, sin, cos, tan, asin, acos, atan, sqrt, sqr
- (expt x y) – x^y
- (random x) – псевдослучайное целое число из [0, x), целое x > 0

Числа (продолжение)

- Напишем функцию two-of-three, которая принимает 3 значения и выдает сумму квадратов наибольших двух из них.

```
(define (two-of-three x y z)
   (cond      ((or (<= x y z) (<= x z y)) (sum-of-squares y z))
              ((or (<= y x z) (<= y z x)) (sum-of-squares x z))
              (else (sum-of-squares x y))))
```

Числа (продолжение)

- Напишем факториал

```
(define (factorial n)
   (if  (= n 1)
        1
        (* n (factorial (- n 1)))))
```

- С помощью подстановочной модели найдём 3!

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

расширение

сжатие

Числа (продолжение)

- Опишем нахождение √x методом Ньютона
- Чтобы приблизительно найти √x нужно:
  1. Выбрать начальное приближение g (= 1).
  2. Получить текущее (улучшенное) значение приближения g := ½ (g + x / g).
  3. Продолжать улучшать приближение, пока g не станет достаточно хорошим.

x = 2	g =1
x / g = 2	g = ½ (1 + 2) = 3/2 = 1,5
x / g = 4/3	g = ½ (3/2 + 4/3) = 17/12 = 1,4166666666666
x / g = 24/17	g = ½ (17/12 + 24/17) = 577/408 = 1,4142156

Числа (продолжение)
Метод Ньютона

```
(define (sqrt-iter guess x)           ; рекурсивная функция
   (if (is-good-enough? guess x)
       guess                          ; выход из рекурсии
       (sqrt-iter (improve guess x) x)  ; рекурсивный вызов
   ))
(define (improve guess x)             ; улучшение приближения
   (average guess (/ x guess)))
(define (average x y)                 ; среднее арифметическое
   (/ (+ x y) 2))
(define (is-good-enough? guess x)  ; проверка приближения
   (< (abs (- (* guess guess) x)) 0.0001))
(define (my-sqrt x) (sqrt-iter 1.0 x)) ; функция для вызова
```

Метод Ньютона (продолжение)

- Улучшим стиль

```
(define (my-sqrt x)
   (define (is-good-enough? guess x)
        (< (abs (- (* guess guess) x)) 0.0001))
   (define (improve guess x)
        (average guess (/ x guess)))
   (define (sqrt-iter guess x)
        (if (is-good-enough? guess x)
                guess
                (sqrt-iter (improve guess x) x)))
   (sqrt-iter 1.0 x))
(define (average x y) (/ (+ x y) 2))
```

- С помощью блочной структуры мы скрыли «лишние» функции

Метод Ньютона (продолжение)

- Перепишем, учитывая, что x внутри my-sqrt один и тот же

```
(define (my-sqrt x)
   (define (is-good-enough? guess)
        (< (abs (- (sqr guess) x)) 0.0001))
   (define (improve guess)
        (average guess (/ x guess)))
   (define (sqrt-iter guess)
        (if (is-good-enough? guess)
                guess
                (sqrt-iter (improve guess))))
   (sqrt-iter 1.0))
(define (average x y) (/ (+ x y) 2))
```

- Избавились от хранения лишних связываний имени x!

Знакомство со Scheme

Литеры

- Внешнее представление: #\a #\A #\newline #\space
- Функции сравнения char=? char>? ...
- Проверка типа char?
- Установка регистра char-upcase char-downcase

Знакомство со Scheme

Строки

- Внешнее представление "Hello \"world#\\"
- Чтобы записать внутри " ставим перед ним \
- Чтобы записать внутри \ ставим перед ним #\
- string аргументы-литеры собирает в строку
- Сравнение строк string=? и т. п.
- Слияние строк string-append
- Выделение частей строки string-head substring string-tail
- Поиск подстроки substring?

Знакомство со Scheme

Списки

- Внешнее представление () (a b c) (a . (b . (c . ())))
- Конструирование списка из головы и хвоста: cons

(cons 'a '(b)) => (a b)

- Взять голову непустого списка car

(car '(1 2 3 4 5 6)) => 1

(car '((1 2 3 4) 5 6)) => (1 2 3 4)

- Взять хвост непустого списка cdr

(cdr '(1 2 3 4 5 6)) => (2 3 4 5 6)

(cdr '((1 2 3 4) 5 6)) => (5 6)

Списки (продолжение)

- Задание перечислением элементов list

(list '+ 1 2) => (+ 1 2)

- Проверка на список list?

(list? '(a b)) => #t (list? '()) => #t

(list? 'a) => #f

- Длина списка length
- Проверка на пустой список null?
- Получить n-ый элемент list-ref

(list-ref '(1 2 3) 2) => 3

- Слить списки (два или больше) append

(append '(1 2 3) '(4 5) '(6)) => '(1 2 3 4 5 6)

- Отзеркалить reverse
- Проверить вхождение элемента member

Списки (продолжение)

- Напишем свою версию list-ref

```
(define (my-list-ref list n)
   (if  ( = n 0)
        (car list)
        (my-list-ref (cdr list) (- n 1))))
```

- Своя версия length

```
(define (my-length list)
   (if  (null? list)
        0
        (+ 1 (my-length (cdr list)))))
```

Списки (продолжение)

- Свой append

```
(define (my-append list1 list2)
   (if  (null? list1)
        list2
        (cons (car list1) (my-append (cdr list1) list2))))
```

- Свой reverse

```
(define (my-reverse lst)
   (if  (null? lst)
        '()
        (append (my-reverse (cdr lst)) (list (car lst)))))
```

Списки (продолжение)

- Функция (apply <функция> <список>)

```
(apply + '(1 2 3))      => 6
(apply max '(1 2 3)) => 3
(apply < '(1 2 3))      => #t
```

Ещё о вычислениях

- Функция (eval <выражение>)

```
(eval (+ 5 7))                  => 12
(eval 12)                       => 12
(eval '(+ 5 7))                 => 12
'(+ 5 7)                        => (+ 5 7)
(define a (list '+ 5 7))
(eval a)                        => 12
(eval 'a)                       => (+ 5 7)
(eval '(eval 'a))               =>
(eval (eval '(eval 'a)))        =>
```

Локальные имена

- Спец. форма (let ((<имя$_1$> <выражение$_1$>)
  (<имя$_2$> <выражение$_2$>) …
  (<имя$_N$> <выражение$_N$>))
  <тело>)
- То же, что ((lambda (<имя$_1$> … <имя$_N$>)
  <тело>)
  <выражение$_1$> … <выражение$_N$>)
- Пример:
  (let ((x (read))) (+ (* (+ x 1) x) 1))
- Спец. форма let* гарантирует правильный порядок

Итоги лекции 1

- Процесс вычисления программы: **read-eval-print.**
- Правила записи имён.
- Связывание. Окружение.
- Классификация выражений.
- Комбинация. Правило вычисления комбинаций.
- Спец. формы (define, lambda, cond, if, case, begin, and, or, quote). Правила их вычисления.
- Числа. Литеры. Строки. Функции работы с ними.
- Блочная структура программы.
- Списки и списочные функции.
- Функции eval и apply. Спец. форма let.

ЛЕКЦИЯ 2
Рекурсия и итерация. Хвостовая рекурсия

Что было раньше

- Подстановочная модель. Правила:

1. Если выражение – литерал, то вернуть само выражение.
2. Если выражение – имя, вернуть его значение в текущем окружении.
3. Если выражение – спец. форма, то вычислить его по правилам этой формы.
4. Если выражение – комбинация, то:
   4.1. Вычислить его подвыражения в произвольном порядке.
   4.2. Если первое подвыражение – встроенная процедура, то применить её к операндам.
   4.3. Если оно – пользовательская функция, подставить в её тело аргументы и вычислить.

Функции и процессы

- Вспомним факториал

```
(define (fact1 n)
        (if (= n 1) 1
            (* n (fact1 (- n 1)))))
```

- Процесс, порождаемый (fact1 3)

```
(if (= 3 1) 1 (* 3 (fact1 (- 3 1))))
(* 3 (fact1 2))
(* 3 (if (= 2 1) 1 (* 2 (fact1 (- 2 1)))))
(* 3 (* 2 (fact1 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact1 (- 1 1))))))
(* 3 (* 2 (* 1))
(* 3 2)
6
```

расширение

сжатие

Функции и процессы

(fact1 6)

(* 6 (fact1 5))

(* 6 (* 5 (fact1 4)))

(* 6 (* 5 (* 4 (fact1 3))))

(* 6 (* 5 (* 4 (* 3 (fact1 2))))) ...

отложенные операции

- Количество шагов линейно. Память линейна.

Функции и процессы

- Итеративный факториал

```
(define (fact2 n)
   (define (loop i result)
         (if (= i 1) result
                  (loop (- i 1) (* i result))))
   (loop n 1))
```

- Процесс порождаемый (fact2 3)

```
(loop 3 1)
(if (= 3 1) 1 (loop (- 3 1) (* 3 1)))
(loop 2 3)
(if (= 2 1) 3 (loop (- 2 1) (* 2 3)))
(loop 1 6)
(if (= 1 1) 6 (loop (- 1 1) (* 1 6)))          ==> 6
```

Функции и процессы

(fact2 6)
(loop 6 1)
(loop 5 6)
(loop 4 30)
(loop 3 120)
(loop 2 360)
(loop 1 720)
720

отложенных операций нет!

- Количество шагов линейно. Память постоянна.

Рекурсивные и итеративные процессы

- Если в ходе процесса возникает цепочка отложенных операций, то процесс рекурсивный.
- Если в ходе процесса отложенных операций нет, то процесс итеративный.
- Рекурсивный процесс, в котором число шагов линейно – линейно рекурсивный.
- Итеративный процесс с линейным количеством шагов – линейно итеративный.

Нелинейный рекурсивный процесс

- Рекурсивное вычисление чисел Фибоначчи

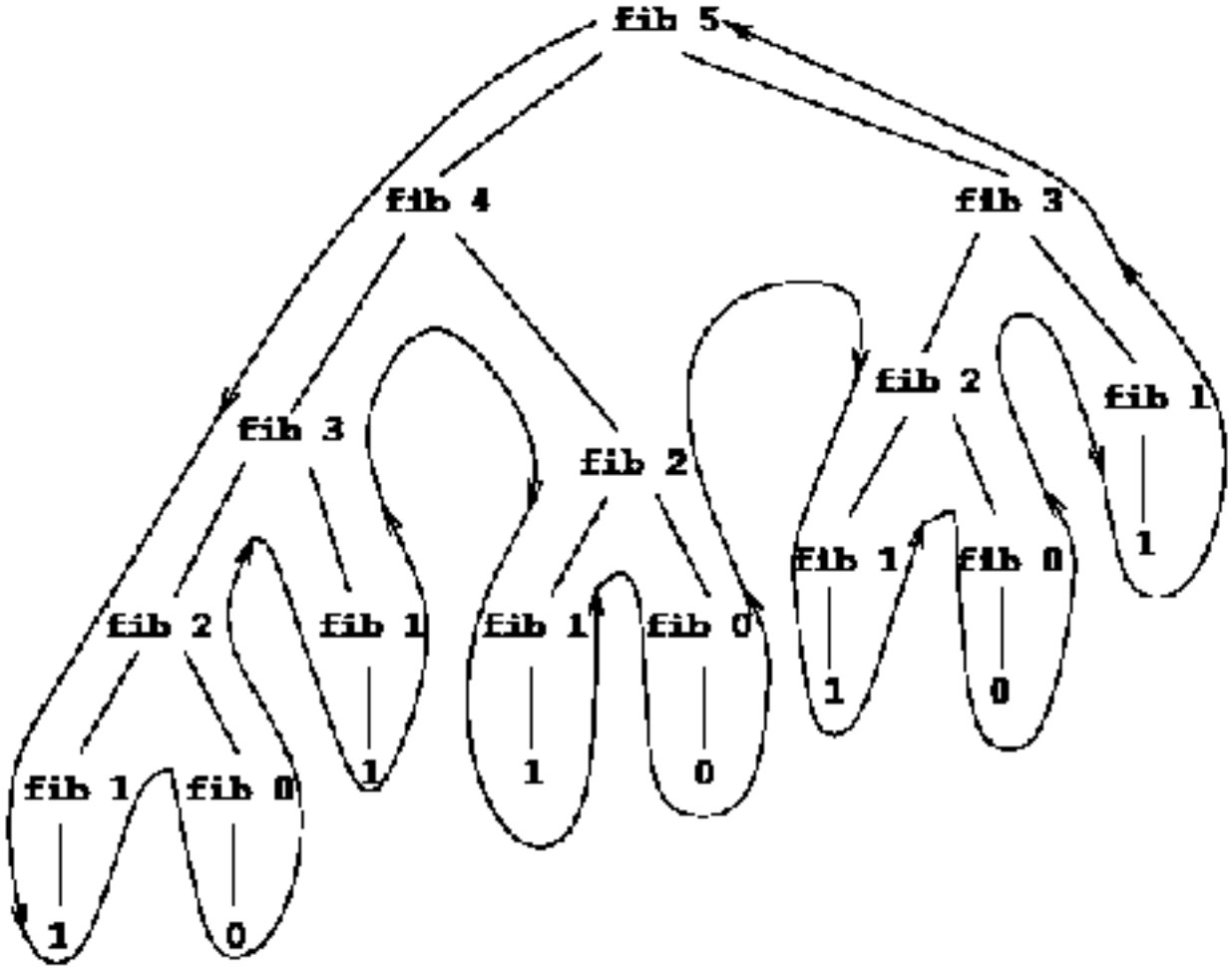
```
(define (fib n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (fib (- n 1)) (fib (- n 2))))))
```

- Вычисление (fib 5)

```
(fib 5)
(+ (fib 4) (fib 3))
(+ (fib 4) (+ (fib 2) (fib 1)))
(+ (fib 4) (+ (+ (fib 1) (fib 0)) 1))
(+ (fib 4) (+ 1 1))
(+ (fib 4) 2)
(+ (+ (fib 3) (fib 2)) 2) ... 5
```

Нелинейный рекурсивный процесс

- Дерево рекурсивного вычисления чисел Фибоначчи



- Количество шагов экспоненциально. Память линейна

Итеративное вычисление чисел Фибоначчи

```
(define (fib2 n)
   (define (loop i fib-n-1 fib-n-2)
        (if (= i 0) fib-n-2
                    (loop (- i 1) (+ fib-n-1 fib-n-2) fib-n-1)))
   (loop n 1 0))
```

- (fib2 5)

  (loop 5 1 0)

  (loop 4 1 1)

  (loop 3 2 1)

  (loop 2 3 2)

  (loop 1 5 3)

  (loop 0 8 5)

- Количество шагов линейно, память константа

Вспомним оборачивание списка

```
(define (my-reverse lst)
   (if  (null? lst)
        '()
        (append (my-reverse (cdr lst)) (list (car lst)))))
(my-reverse '(1 2 3))
   (append (my-reverse '(2 3)) (list 1))
   (append (append (my-reverse '(3)) (list 2)) '(1))
   (append (append (append (my-reverse '()) (list 3)) '(2))
   '(1))
   (append (append (append '() '(3)) '(2)) '(1))
   (append (append '(3) '(2)) '(1))
   (append '(3 2) '(1))     ==> (3 2 1)
```

Отложенные операции! Рекурсивный процесс

Итеративное оборачивание списка

```
(define (reverse2 lst)
    (define (loop lst result)
        (if (null? lst) result
            (loop (cdr lst) (conc (car lst) result))))
    (loop lst '()))
```

- (reverse2 '(1 2 3))

  (loop '(1 2 3) '())

  (loop '(2 3) '(1))

  (loop '(3) '(2 1))

  (loop '() '(3 2 1))

Линейно итеративный процесс

Возведение в степень

- $a^n = a \cdot a \cdots a = a \cdot a^{n-1}$

```
(define (my-expt a n)
    (if (= n 0)
        1
        (* a (my-expt a (- n 1)))))
```

- (my-expt 10 4)

```
(if (= 4 0) 1 (* 10 (my-expt 10 (- 4 1))))
```

отложенные операции!

- Количество шагов линейно. Память линейна

Итеративное возведение в степень

- перепишем с накоплением

```
(define (my-expt1 a n)
   (define (loop i result)
     (if (= i 0)
         result
         (loop (- i 1) (* a result))))
   (loop n 1))
```

- (my-expt1 10 4)

(loop 4 1)

(loop (- 4 1) (* 10 1))

(loop (- 3 1) (* 10 10))

(loop (- 2 1) (* 10 100)) ...

отложенные операции отсутствуют!

- Количество шагов линейно. Память постоянна

Нелинейное итеративное возведение в степень

```
(define (my-expt2 a n)
   (define (loop a i result)
     (cond ((= i 0) result)
           ((even? i) (loop (* a a) (/ i 2) result))
           (else (loop a (- i 1) (* a result)))))
   (loop a n 1))
```

- (my-expt2 10 4)

(loop 10 4 1)

(loop (* 10 10) (/ 4 2) 1)

(loop (* 100 100) (/ 2 1) 1)

(loop 10000 1 1)

- Количество шагов O(log n). Память постоянна

Итеративный НОД с логарифмическим ростом

```
(define (my-gcd a b)
   (if (= b 0) a
       (my-gcd b (remainder a b))))
```

(my-gcd 4 12)

(my-gcd 12 4)

(my-gcd 4 0)

4

- Количество шагов O(log n). Память постоянна

Итоги лекции 2

- Рекурсивное описание функции может породить рекурсивный процесс или итеративный процесс.
- Писать программу надо, оценивая сложность по шагам и по памяти.
- Хвостовая рекурсия – способ экономить память. В рекурсивном вызове можно не создавать новое окружение.

ЛЕКЦИЯ 3
Функции высшего порядка

Функция высшего порядка

- Функция, манипулирующая другими функциями, называется функцией высшего порядка.
- Пример -- суммирование:

```
(define (sum-integers a b)
      (if (> a b)
            0
            (+ a (sum-integers (+ a 1) b))))
(define (sum-cubes a b)
      (if (> a b)
            0
            (+ (* a a a) (sum-cubes (+ a 1) b))))
```

Функция высшего порядка

- Суммирование (продолжение) $\sum 1/((4i-3)*(4i-1))$:

```
(define (pi-sum a b)
      (if (> a b)
            0
            (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

- Проглядывается общая схема

```
(define (<имя> a b)
      (if (> a b)
            0
            (+ (<терм> a) (<имя> (<следующий> a) b))))
```

Функция высшего порядка

- Опишем функцию суммирования:

```
(define (sum term a next b)
    (if (> a b) 0
        (+ (term a) (sum term (next a) next b))))
```

- Сумма кубов через sum:

```
(define (sum-cubes a b)
    (sum (lambda (x) (* x x x)) a (lambda (x) (+ x 1)) b))
```

- Сумма целых через sum

```
(define (sum-integers a b)
    (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
```

- (define (pi-sum a b)

(sum (lambda (x) (/ 1.0 (* x (+ x 2)))) a (lambda (x) (+ x 1)) b))

«Лайфхак»: можно использовать λ-выражения!

Итог по суммированию

- Описали схему суммирования функцией высшего порядка.
- Повысили уровень абстракции.
- Избавились от дублирования кода.
- Явно выразили идею в программе.

- sum порождает рекурсивный процесс, перепишем:

```
(define (sum term a next b)
   (define (loop a result)
         (if (> a b)
               result
               (loop (next a) (+ (term a) result))))
   (loop a 0))
```

Перемножение

- По аналогии с суммированием можно умножать:

```
(define (product term a next b)
   (define (loop a result)
         (if (> a b)
                result
                (loop (next a) (* (term a) result))))
   (loop a 1))
(define (sum term a next b)          ; есть что-то общее
   (define (loop a result)
         (if (> a b)
                result
                (loop (next a) (+ (term a) result))))
   (loop a 0))
```

Накопление

- Повысим уровень абстракции:

```
(define (accumulate combiner null-val term a next b)
   (define (loop a result)
         (if (> a b)
                result
                (loop (next a) (combiner (term a) result))))
   (loop a null-val))

(define (sum term a next b)
   (accumulate + 0 term a next b))

(define (product term a next b)
   (accumulate * 1 term a next b))
```

Накопление

- Пример использования accumulate:

```
(define (enumerate-interval a b)
   (accumulate (lambda (x y) (append y x))
        '() (lambda (x) (list x)) a (lambda (x) (+ x 1)) b))

(enumerate-interval 1 10)   => (1 2 3 4 5 6 7 8 9 10)
```

- Если в обратном порядке, то проще

```
(accumulate cons '() (lambda (x) x) 10 (lambda (x) (+ x 1)) 20)
=>      (20 19 18 17 16 15 14 13 12 11 10)
```

- если всё ещё нужно в прямом, то считаем в обратном и переворачиваем reverse, который линейный.

Накопление с фильтром

- Опишем filtered-accumulate:

(define (filtered-accumulate predicate combiner null-val term a next b) ...)

если очередное a удовлетворяет фильтру, происходит накопление, иначе пропускаем и берем следующее a.

```
(define (filtered-accumulate predicate combiner null-val term a next b)
  (define (loop a result)
      (if (> a b)      result
                       (loop (next a)
                       (if (predicate a) (combiner (term a) result)
                           result))))
  (loop a null-val))
```

Функция высшего порядка возвращает функцию

- Определим взятие производной от функции-аргумента:

```
(define (derive f)
      (lambda (x) (/ (- (f (+ x 0.00001)) (f x)) 0.00001)))
```

- Вызовем: ((derive (lambda (x) (* x x))) 10)
- По подстановочной модели:

```
((lambda (x) (/ (- ((lambda (x) (* x x)) (+ x 0.00001))
   ((lambda (x) (* x x)) x)) 0.00001)) 10)

(/ (- ((lambda (x) (* x x)) (+ 10 0.00001)) ((lambda (x) (* x
   x)) 10)) 0.00001))

...
```

20.0000099942131 ~ 2x в точке x=10

Функции высшего порядка и списки

- Умножение элементов списка

```
(define (scale-list lst factor)
   (if (null? lst)
        '()
        (cons (* (car lst) factor) (scale-list (cdr lst) factor))))
```

- Приращение, возведение в степень ...
- Отображение списка (map <функция> <список>)

```
(define (map func lst)
   (if (null? lst)
        '()
        (cons (func (car lst)) (map func (cdr lst)))))
(define (scale-list lst factor)
   (map (lambda (x) (* x factor)) lst))
```

Функции высшего порядка и списки

- Стандартный **map**

(map <функция> <список1> <список2> … <списокN>)

(map + '(1 2 3) '(10 20 30) '(100 200 300))

=> (111 222 333)

- «Близнец» **map** – for-each

(for-each (lambda (x) (println x)) '(1 2 3))

=> 1

2

3

Функции высшего порядка и списки

- Просеивание filter

(filter <предикат> <список>)

(filter odd? '(1 2 3 4 5))

=> (1 3 5)

```
(define (filter predicate lst)
   (cond ((null? lst) '())
         ((predicate (car lst))
               (cons (car lst) (filter predicate (cdr lst))))
         (else (filter predicate (cdr lst)))))
```

Функции высшего порядка и списки

- Накопление

(accum <функция> <нач значение> <список>)

(accum + 0 '(1 2 3 4 5))

=> 15

(accum * 1 '(1 2 3 4 5))

=> 120

```
(define (accum func init lst)
   (if (null? lst) init)
         (func (car lst) (accum func init (cdr lst)))))
```

Стандартный синоним foldr

Правоассоциативная свертка

(func e1 (func e2 (... (func eN init)...)

Функции высшего порядка и списки

- Левоассоциативная свёртка

(foldl <функция> <нач значение> <список>)

```
(define (foldl func init lst)
   (if (null? lst) init
         (foldl func (func (car lst) init) (cdr lst))))
```

(func eN (func eN-1 (... (func e1 init)...)))

```
(foldl cons '() '(1 2 3 4))       =>    (4 3 2 1)
Но (foldr cons '() '(1 2 3 4)) =>    (1 2 3 4)

(foldl list '() '(1 2 3 4))       =>    (4 (3 (2 (1 ()))))
(foldr list '() '(1 2 3 4))       =>    (1 (2 (3 (4 ()))))
```

Лево- и право- ассоциативные свёртки

- сумма списка

(foldl + 0 lst) ; найдём сумму подсписка без последнего элемента и прибавим последний элемент

(foldr + 0 lst) ; найдём сумму хвоста списка и прибавим первый элемент

- минимум списка неотрицательных чисел

(foldl min 0 lst) ; найдём min в подсписке без последнего элемента и выберем минимум из него и последнего элемента

(foldr min 0 lst) ; найдём min в хвосте списка и выберем минимум из него и первого элемента

Лево- и право- ассоциативные свёртки

- (map func lst)

(foldl (lambda (x y) (append y (list (func x)))) '() lst)

найдём map подсписка без последнего элемента и допишем к нему список из func от последнего элемента

(foldr (lambda (x y) (cons (func x) y)) '() lst)

найдём map от хвоста списка припишем его справа после func от первого элемента

Функции высшего порядка и списки

- map, filter и свертки позволяют легко реализовывать обработку списков

- произведение квадратов нечетных чисел списка

```
(define (product-of-squares-of-odd-elems lst)
   (foldr * 1
        (map (lambda (x) (* x x))
             (filter odd? lst))))

(product-of-squares-of-odd-elems '(1 2 3 4 5))  => 225
```

Функции высшего порядка и списки

- список квадратов чисел Фибоначчи

```
(define (list-fib-squares n)
        (map (lambda (x)
                    (let ((temp  (fib x))) (* temp temp))
                       (enumerate-interval 1 n)))
```

Д/з написать линейно-итерационно.

Пример

- Вычислить значение многочлена в точке: коэффициенты многочлена задаются списком.

```
(define (gorner-l lst x)
; a_n, a_n-1, ... a_1, a_0 по убыванию степеней
  (foldl (lambda (a b) (+ (* b x) a)) 0 lst))

(define (gorner-r lst x)
; a_0, a_1, ... a_n-1, a_n по возрастанию степеней
  (foldr (lambda (a b) (+ (* b x) a)) 0 lst))
```

Д/з

- На вход подается список списков, на выходе требуется получить только те списки, сумма элементов которых больше произведения элементов первого списка.

Итоги лекции 3

- Функции высшего порядка – это полезно.
- Абстракция позволяет программисту контролировать сложность программы и явно выражать в тексте свои идеи.
- В Scheme функция может быть как аргументом вызова функции, так и результатом.
- map, filter, foldl, foldr помогают в работе со списками.
- Вообще говоря, бывают свёртки для деревьев.

Разбор задач анкеты от 21 сентября

- Функция (sqlist n) для натурального n возвращает список квадратов первых n натуральных чисел (1 4 9 … n^2), а для остальных целых n – пустой список.

```
(define (sqlist-r1 n)
  (if (< n 1) '()
      (append (sqlist-r1 (- n 1)) (list (* n n)))))
```

вспомним, что append – линейный ☻

Разбор задач анкеты от 21 сентября

- Перепишем с cons

```
(define (sqlist-r2 n)
  (define (helper i)
    (if (> i n) '()
        (cons (* i i) (helper (+ i 1)))))
  (helper 1))
```

- теперь итеративный

```
(define (sqlist-i n)
  (define (loop i result)
    (if (< i 1) result
        (loop (- i 1) (cons (* i i) result))))
  (loop n '()))
```

Разбор задач анкеты от 21 сентября

- Функция (powlist n) для натурального n возвращает список первых n+1 степеней 2 (1 2 4 … 2^n), а для остальных целых n – пустой список.

```
(define (powlist-r n)
  (define (helper i pow-i)
    (if (> i n) '()
        (cons pow-i (helper (+ i 1) (* pow-i 2)))))
  (if (< n 1) '() (helper 0 1)))
```

Разбор задач анкеты от 21 сентября

- теперь итеративный

```
(define (powlist-i n)
  (define (loop i pow-i result)
    (if (< i 0) result
        (loop (- i 1) (* pow-i 2) (cons pow-i result))))
  (if (< n 1) '() (reverse (loop n 1 '()))))
```

ЛЕКЦИЯ 4
Структуры данных

Потребность в структурах данных

- Рассмотрим пример: арифметика рациональных чисел
- Реализация без структуры данных

```
(define (num-of-sum num1 den1 num2 den2)
  (+ (* num1 den2) (* num2 den1)))
(define (den-of-sum num1 den1 num2 den2)
  (* den1 den2))
```

- недостатки:
  - отслеживание какие числители соответствуют каким знаменателям
  - большое количество аргументов функций
  - все операции из двух частей -- вычисление числителя и вычисление знаменателя

Проектирование структуры данных

- Придумаем способ работы с рациональными числами.
- Основные операции (конструктор и селекторы):
  - (make-rat num den) – создать рац. число
  - (num r) – получить числитель числа r
  - (den r) – получить знаменатель числа r
- Полагая операции реализованными можно определить сумму, произведение ...

```
(define (sum-rat a b)
    (make-rat (+ (* (num a) (den b)) (* (den a) (num b)))
              (* (den a) (den b))))
(define (mul-rat a b)
    (make-rat (* (num a) (num b)) (* (den a) (den b))))
```

Проектирование структуры данных

- sum-rat такова, что 1/2 + 1/6 = 8/12, а не 2/3
- Можно переписать sum-rat, используя gcd

```
(define (sum-rat a b)
   (let* ((n (+ (* (num a) (den b)) (* (den a) (num b))))
          (d (* (den a) (den b)))
          (g (gcd n d)))
   (make-rat (/ n g) (/ d g))))
```

- Можно специальным образом определить make-rat (или селекторы):

```
(define (make-rat a b)
    (let ((g (gcd a b))) (cons (/ a g) (/ b g))))
(define num car)
(define den cdr)
```

Что дала структура данных?

- Получилась слоистая система:

программы, работающие
с рациональными числами

---

sum-rat, mul-rat,..

---

make-rat, num, den

---

cons, car, cdr

реализация пар

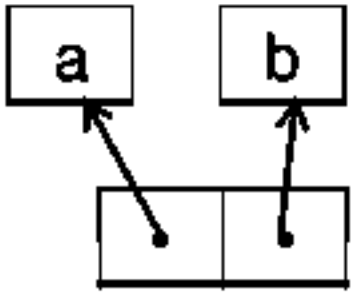
Что дала слоистая система?

- Изменения локализованы на уровне
- Переопределим make-rat и селекторы:

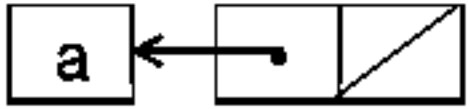
```
(define make-rat cons)
(define (num x)
   (let ((g (gdc (car x) (cdr x))))
   (/ (car x) g)))
(define (den x)
   (let ((g (gdc (car x) (cdr x))))
   (/ (cdr x) g)))
```

- sum-rat, mul-rat работают, их не надо переписывать

cons



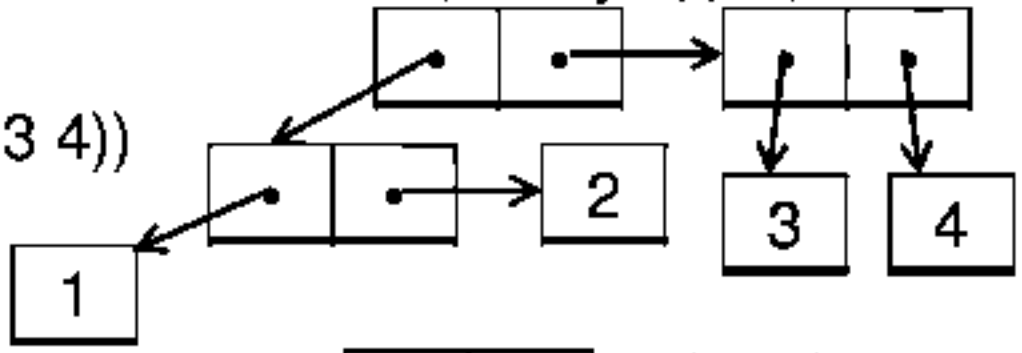
- cons создаёт точечную пару
- внешнее представление (cons a b) => (a . b)
- пара – не всегда список!



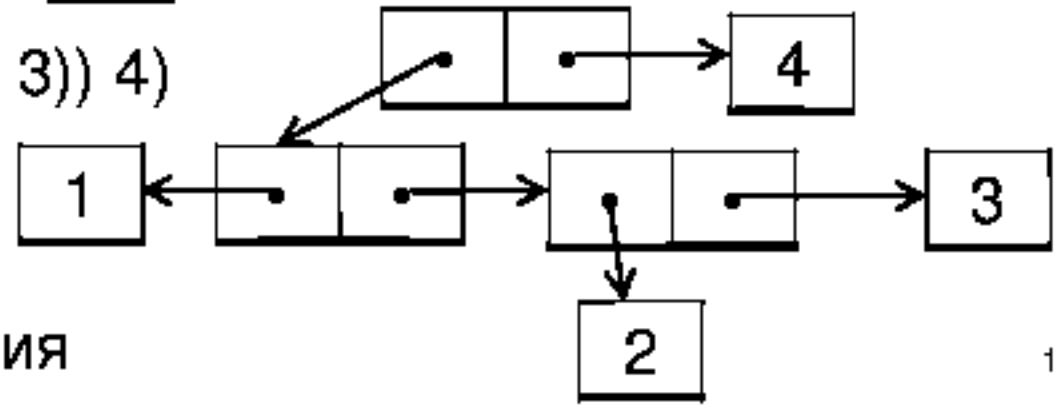
(cons a '())

- элементами пары может быть всё, что угодно, в том числе другие пары:

(cons (cons 1 2) (cons 3 4))



(cons (cons 1 (cons 2 3)) 4)



- свойство замыкания

Отступление. Точечные пары и функции

- С помощью нотации точечной пары можно определить функцию с нефиксированным количеством параметров:

```
(define (f . params)
   (display params))
(f) =>                  ()
(f 1 2 3 4 5) =>        (1 2 3 4 5)

(define (first+ x . tail)
   (map (lambda (y) (+ x y)) tail))

(first+ 10 1 2 3 4) =>        (11 12 13 14)
(first+ 10)         =>        ()
```

Отступление. Cons, car, cdr и функции

- Рассмотрим код

```
(define (cons x y)
   (define (dispatch m)
     (cond ((= m 0) x)
           ((= m 1) y)
           (else (println "ошибка cons"))))
   dispatch)
(define (car x) (x 0))
(define (cdr x) (x 1))
```

- Получили «функциональную» реализацию пар.
- Вообще говоря, можно числа реализовать функциями
- А значит -- и все данные.

Отступление. Cons, car, cdr и функции

- Другая «функциональная» реализация пар

```
(define (cons x y)
   (lambda (m) (m x y)))
(define (car pair) (pair (lambda (p q) p)))
(define (cdr pair) (pair (lambda (p q) q)))
```

(car (cons 1 2)) ==> (car (lambda (m) (m 1 2))) ==>
((lambda (m) (m 1 2)) (lambda (p q) p)) ==>
((lambda (p q) p) 1 2) ==> 1

аналогично cdr

Деревья

- Будем рассматривать бинарные деревья с данными как в узлах, так и в листьях.
- Дерево, это такая абстракция, для которой верно:
  - Пустое дерево – это дерево. empty-tree
  - Непустое дерево состоит из корня и двух поддеревьев. (make-tree data left right)
  - Дерево можно проверить на пустоту. empty-tree?
  - Селекторы дерева:
    - получить корень tree-data
    - получить левое поддерево tree-left
    - получить правое поддерево tree-right

Деревья

- Полагая базовые операции реализованными, можно определять более сложные:
  - проверить, все ли узлы удовлетворяют определенному условию:

```
(define (all-tree p t)
  (if (empty-tree? t) #t
      (and (p (tree-data t)) (all-tree p (tree-left t))
                             (all-tree p (tree-right t))))))
```

  - существует ли узел, удовлетворяющий условию:

```
(define (any-tree p t)
  (not (all-tree (lambda (x) (not (p x))) t))
```

Деревья

- Получить список всех узлов:

```
(define (flatten-tree t)
  (if (empty-tree? t) '()
      (append (flatten-tree (tree-left t))
              (cons (tree-data t) (flatten-tree (tree-right t))))))
```

- Получить список листьев:

```
(define (fringe-tree t)
  (cond ((empty-tree? t) '())
    ((and (empty-tree? (tree-left t))
          (empty-tree? (tree-right t)))
      (list (tree-data t)))
    (else (append (fringe-tree (tree-left t))
                  (fringe-tree (tree-right t)))))))
```

Деревья

- Получить список узлов, удовлетворяющих условию:

```
(define (collect-tree p t)
  (cond ((empty-tree? t) ())
        ((p (tree-data t))
         (append (collect-tree p (tree-left t))
                 (cons (tree-data t) (collect-tree p (tree-right t)))))
        (else
         (append (collect-tree p (tree-left t))
                 (collect-tree p (tree-right t))))))
```

Деревья

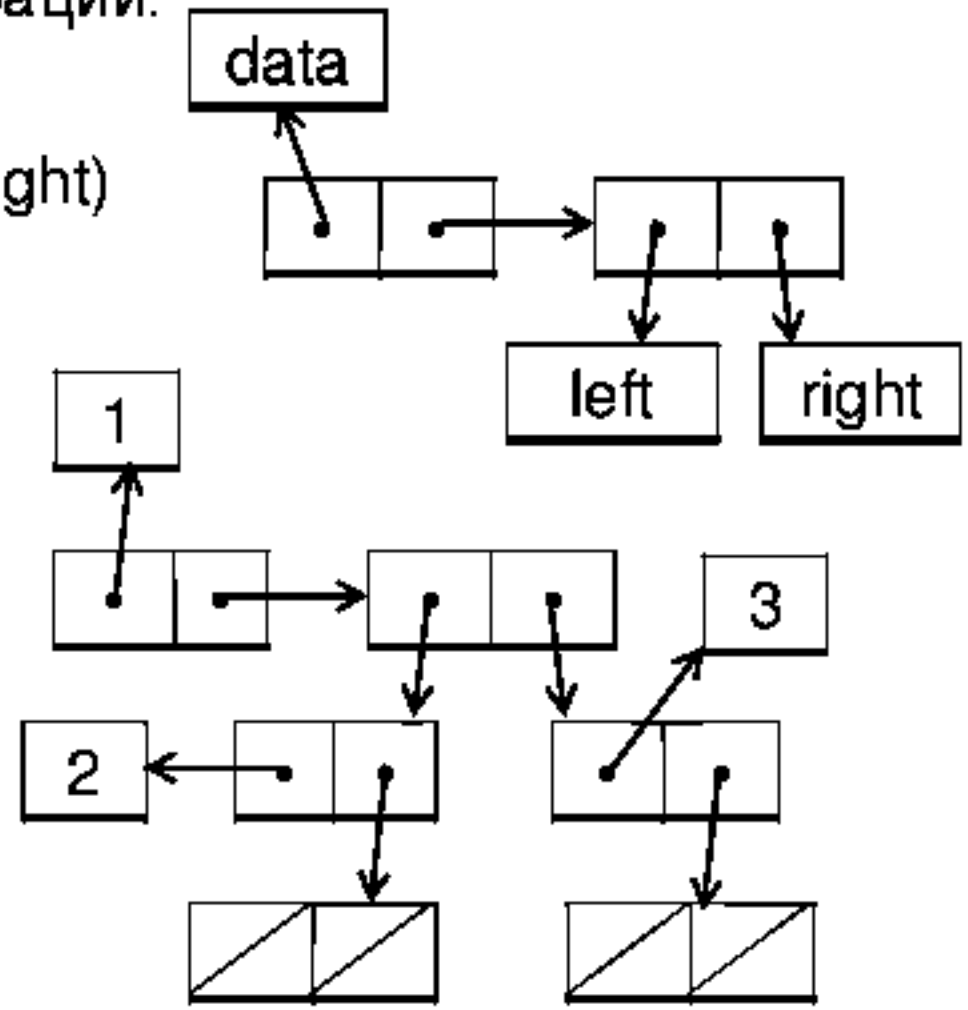
- Выполнить действие для каждого узла, удовлетворяющего условию:

```
(define (with-all-satisfying p k t)
  (define (rec t)
    (if (empty-tree? t)
        #f
        (begin
          (if (p (tree-data t)) (k (tree-data t)) #f)
          (rec (tree-left t))
          (rec (tree-right t)))))
  (rec t))
```

Деревья

- Реализуем базовые операции:

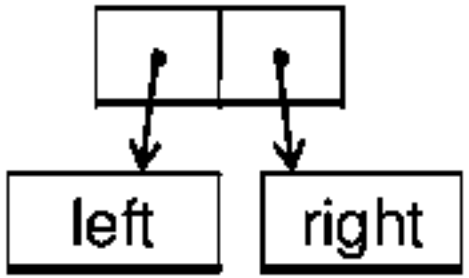
```
(define empty-tree '())
(define (make-tree data left right)
  (cons data (cons left right)))

(define tree-data  car)
(define tree-left  cadr)
(define tree-right cddr)

(define (empty-tree? t)
  (eq? t '()))
```

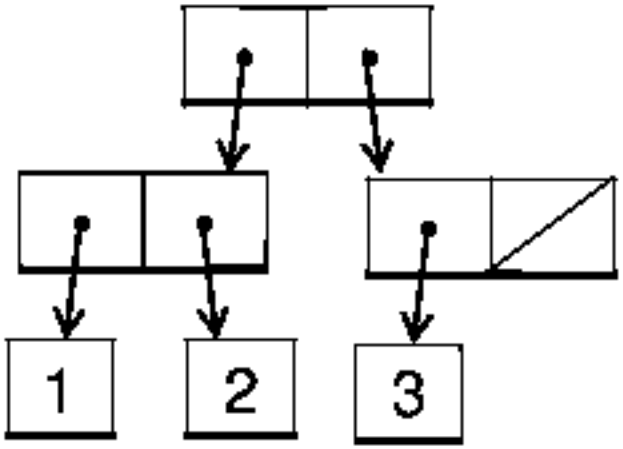


Деревья

- Иногда рассматривают деревья с данными только в листьях:

```
(define empty-tree '())
(define (make-tree left right)
  (cons left right))
(define tree-left  car)
(define tree-right cdr)
(define (empty-tree? t)
  (eq? t '()))

(define (leaf-tree? x) (not (pair? x)))
(define (tree-data t)
        (if (leaf-tree? t) t '()))
```





Бинарные деревья поиска

- В дереве поиска данные упорядочены (меньшие – слева от корня, большие -- справа).
- Базовые операции дерева поиска:
  - empty-bst
  - (empty-bst? t)
  - (insert-bst key t)
  - (member-bst? key t)

Бинарные деревья поиска

- Реализации

```
(define empty-bst empty-tree)

(define (empty-bst? t) (empty-tree? t))

(define (member-bst? key t)
   (cond ((empty-bst? t) #f)
      ((= key (tree-data t)) #t)
      ((< key (tree-data t))
       (member-bst? key (tree-left t)))
      (else
       (member-bst? key (tree-right t)))))
```

Бинарные деревья поиска

- Вставка (без балансировки)

```
(define (insert-bst key t)
  (cond ((empty-bst? t)
         (make-tree key empty-bst empty-bst))
        ((= key (tree-data t))
         t)
        ((< key (tree-data t))
         (make-tree (tree-data t)
                    (insert-bst key (tree-left t))
                    (tree-right t)))
        (else
         (make-tree (tree-data t)
                    (tree-left t)
                    (insert-bst key (tree-right t))))))
```

Итоги лекции 5

- Структуры данных упрощают программы
- При проектировании структур данных полезно использовать слоистые системы.
- Основа иерархических структур данных в Scheme – точечная пара (cons, car, cdr).

Разбор задач анкеты от 28 сентября

- С помощью функций высшего порядка map, foldl, foldr, filter опишите функцию (task28a lst) принимающую непустой список списков чисел. Функция находит среднее арифметическое максимумов модулей элементов вложенных списков.

```
(define (task22a lst)
  (* 1.0 (/ (foldl + 0 (map (lambda (ls)
                              (foldl max 0 (map abs ls))) lst))
            (length lst))))
```

Разбор задач анкеты от 28 сентября

- С помощью функций высшего порядка map, foldl, foldr, filter опишите функцию (task28b lst) принимающую непустой список списков чисел. Функция находит максимальное среди всех средних арифметических модулей элементов вложенных списков.

```
(define (task22b lst)
   (foldl max 0 (map (lambda (ls)
                       (if (null? ls) 0
                           (* 1.0 (/ (foldl + 0
                                            (map abs ls))
                                     (length ls)))
                       )) lst)))
```

ЛЕКЦИЯ 5
Остаточные вычисления. Стиль передачи остаточных вычислений

Остаточные вычисления

- Остаточные вычисления (continuations) -- что-то, что ждёт значение. С каждым промежуточным значением связано остаточное вычисление -- вычисления, которые должны быть выполнены, как только значение станет известным.
- Например: (sqrt (+ (read) 1)) ждёт, пока пользователь не введет число.
- Остаточные вычисления для функции foo в выражении (* (foo 1) (+ 5 1)) можно описать через lambda:
- (lambda (x) (* x (+ 5 1)))

Остаточные вычисления

- Остаточные вычисления – динамические объекты. На каждом шаге программы существуют текущие остаточные вычисления
- Например:

```
(define (fact n) (if (= n 0) 1 ( * n (fact (- n 1)))))

(fact 3)
 (* 3 (fact 2))
 (* 3 (* 2 (fact 1)))
 (* 3 (* 2 (* 1 (fact 0))))
...
=> 6
```

Факториал в стиле передачи о. в.

Перепишем факториал в виде функции с дополнительным параметром -- функцией, представляющей собой остаточные вычисления:

```
(define (fact-cc n cc)
   (if (= n 0) (cc 1)
       (fact-cc (- n 1) (lambda (x) (cc (* n x))))))
```

Пример вызова:

```
(fact-cc 2 (lambda (x) x))
```

(fact-cc 1 (**lambda** (x_1) ((lambda (x) x) (* 2 x1))))

(fact-cc 0 (**lambda** (x_2) ((lambda (x_1)((lambda (x)x)(* 2 x_1))) (* 1 x_2))))

((lambda (x_2) ((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 x_2))) 1)

Факториал в стиле передачи о. в. Продолжение

((lambda (x_2) ((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 x_2))) 1)
((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 1))
((lambda (x) x) (* 2 1))
2

reverse в стиле передачи о. в.

- Опишем reverse в continuation-passing style
- Сначала определим схему вычислений:
  - Результат можно получить, вставляя элементы исходного списка по одному от начала к хвосту в голову списка-результата.
  - reverse в обычном стиле:

```
(define (reverse-simple lst)
   (define (loop l res)
            (if (null? l) res (loop (cdr l) (cons (car l) res))))
   (loop lst '()))
```

  - пример: (loop (list 1 2 3) '())
    (loop (list 2 3) (cons 1 '()))
    (loop (list 3) (cons 2 (cons 1 '()))) ...

reverse в стиле передачи о. в.

- Пишем в continuation-passing style

```
(define (reverse-cps lst cc)
        (if (null? lst) (cc '())
            (reverse-cps (cdr lst) (lambda (x) (cons (car lst) (cc x))))))
```

- что общего с обычным описанием?

```
        (define (loop l res)
                (if (null? l) res (loop (cdr l) (cons (car l) res))))
```

- пример работы

(reverse-cps '(1 2) (lambda (x) x))

(reverse-cps '(2) (lambda (x_1) (cons 1 ((lambda (x) x) x_1))))

(reverse-cps '() (lambda (x_2) (cons 2 ((lambda (x_1) (cons 1 ((lambda (x) x) x_1))) x_2))))

reverse в стиле передачи о. в.

(reverse-cps '() (lambda (x_2) (cons 2 ((lambda (x_1) (cons 1 ((lambda (x) x) x_1))) x_2))))

((lambda (x_2) (cons 2 ((lambda (x_1) (cons 1 ((lambda (x) x) x_1))) x_2))) '())

(cons 2 ((lambda (x_1) (cons 1 ((lambda (x) x) x_1))) '()))

(cons 2 (cons 1 ((lambda (x) x) '())))

(cons 2 (cons 1 '()))

(cons 2 '(1))

'(2 1)

Вставка в хвост в стиле передачи о. в.

- Опишем app-cps, добавляющую элемент в хвост списка:
- Сначала определим схему вычислений:
  - Результат можно получить, приписав голову исходного списка к результату вставки в хвост (рекурсивной). Вставка в пустой список получается как список из нового элемента.
  - пишем в обычном стиле:

```
(define (app-simple lst e)
   (if (null? lst) (list e)
            (cons (car lst) (app-simple (cdr lst) e))))
```

  - пример работы

```
(app-simple '(1 2) 3)
(cons 1 (app-simple '(2) 3))
(cons 1 (cons 2 (app-simple '() 3))) ...
```

Вставка в хвост в стиле передачи о. в.

- Опишем app-cps, добавляющую элемент в хвост списка:

```
(define (app-cps lst e cc)
   (if (null? lst) (cc (list e))
   (app-cps (cdr lst) e (lambda (k) (cc (cons (car lst) k))))))
```

- Что общего с обычной версией?

```
(define (app-simple lst e)
   (if (null? lst) (list e)
          (cons (car lst) (app-simple (cdr lst) e))))
```

- Пример:

(app-cps '(1 2) 3 (lambda (x) x))

(app-cps '(2) 3 (lambda (x_1) ((lambda (x) x) (cons 1 x_1))))

(app-cps '() 3 (lambda (x_2) ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 x_2))))

Вставка в хвост в стиле передачи о. в.

(app-cps '() 3 (lambda (x_2) ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 x_2))))

((lambda (x_2) ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 x_2))) '(3))

(((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 '(3)))

((lambda (x) x) (cons 1 (cons 2 '(3))))

(cons 1 (cons 2 '(3)))

(cons 1 '(2 3))

'(1 2 3)

Итоги лекции 5

- Преобразуя программу согласно cps мы достигаем двух целей:
  - Всю рекурсию делаем хвостовой.
  - В интерпретаторе, вычисляющем в аппликативном порядке, реализуем вычисления в нормальном порядке.

Немного повторения перед анкетой

- length-cps
- сначала обычный

```
(define (length-simple lst)
   (if (null? lst) 0
      (+ 1 (length-simple (cdr lst)))))
```

- пример

```
(length-simple '(1 2 3))
(+ 1 (length-simple '(2 3)))
(+ 1 (+ 1 (length-simple '(3))))
(+ 1 (+ 1 (+ 1 (length-simple '()))))
(+ 1 (+ 1 (+ 1 0)))
(+ 1 (+ 1 1))
(+ 1 2)          3
```

Немного повторения перед анкетой

- готовы писать length-cps

```
(define (length-cps lst cc)
  (if (null? lst) (cc 0)
      (length-cps (cdr lst) (lambda (x) (+ 1 (cc x))))))
```

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

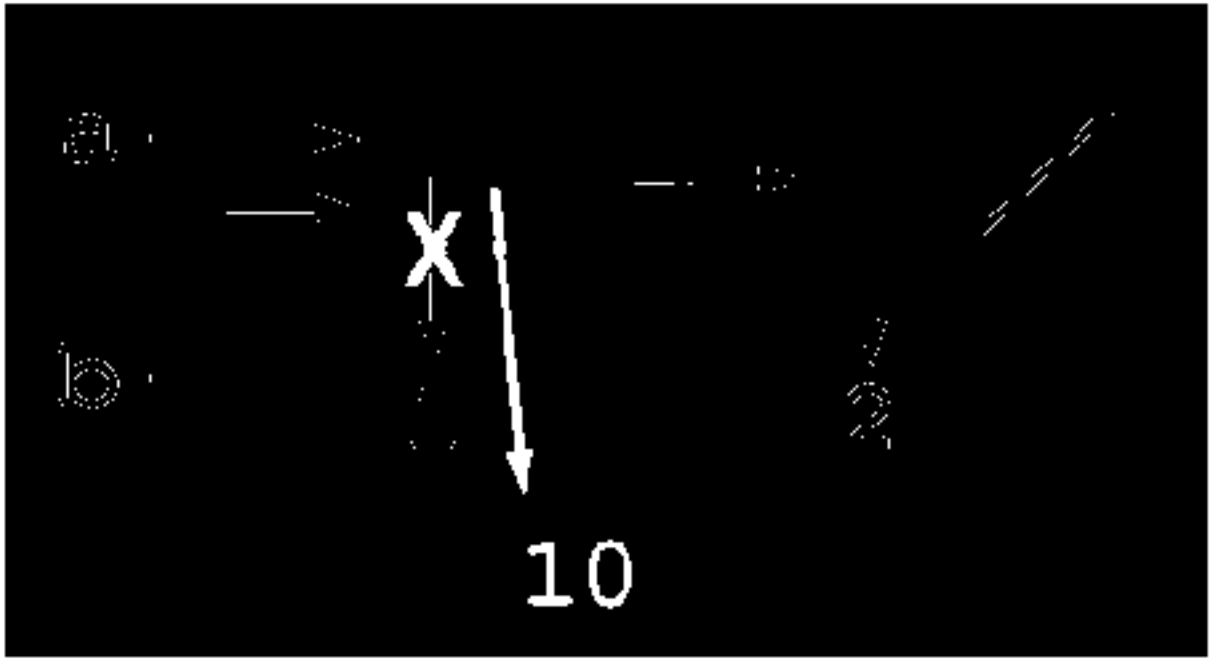
- Присваивание делает неактуальной подстановочную модель!
- Рассмотрим withdraw без присваивания

```
(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

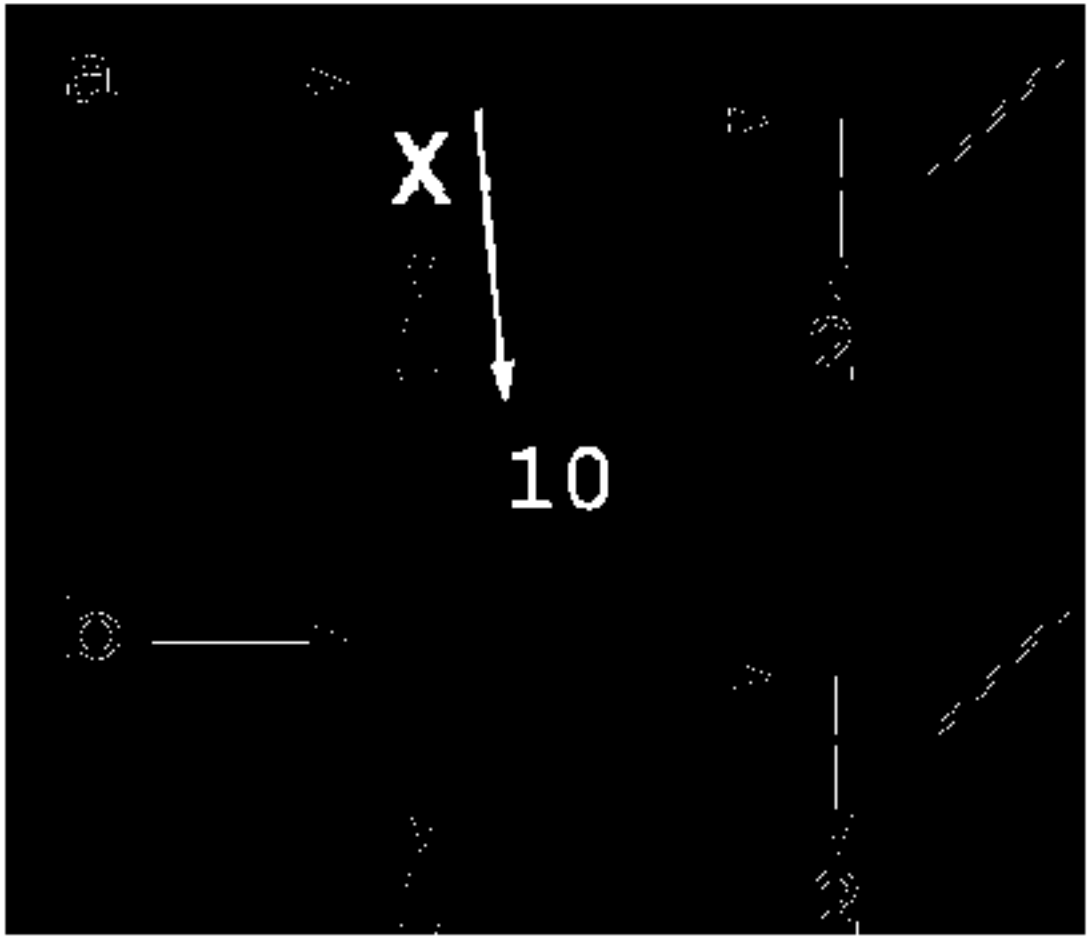
- (set-car! <пара> <новый car>) меняет 1й элемент точечной пары
- (set-cdr! <пара> <новый cdr>) меняет 2й элемент точечной пары

(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

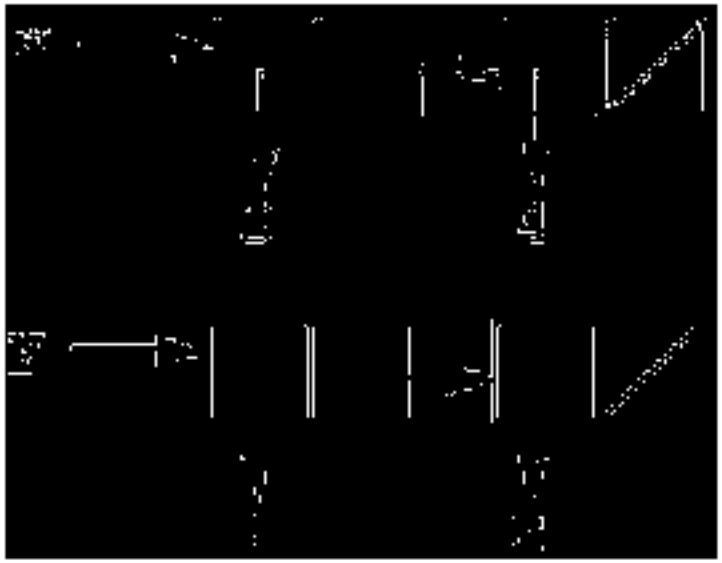
  но не совпадать

Последствия присваивания

- Часть одного объекта может быть общей с другим объектом:

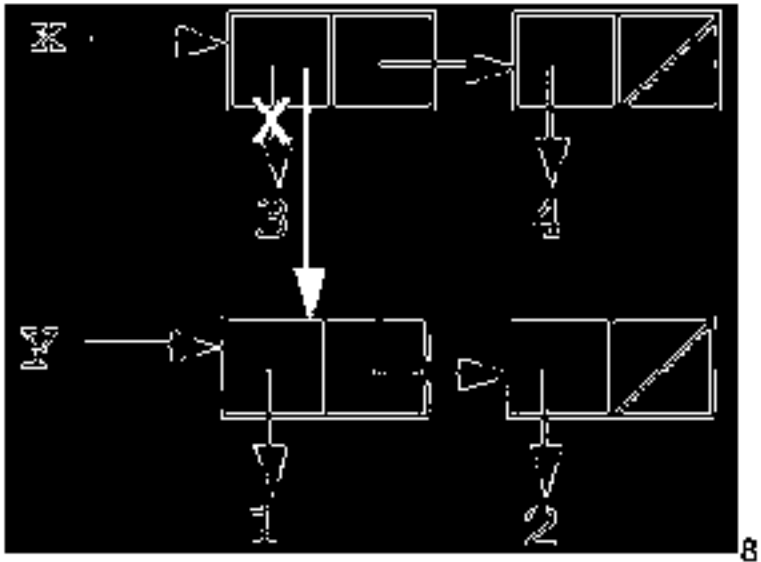
(define x '(3 4))

(define y '(1 2))



(set-car! x y)

x → ((1 2) 4)



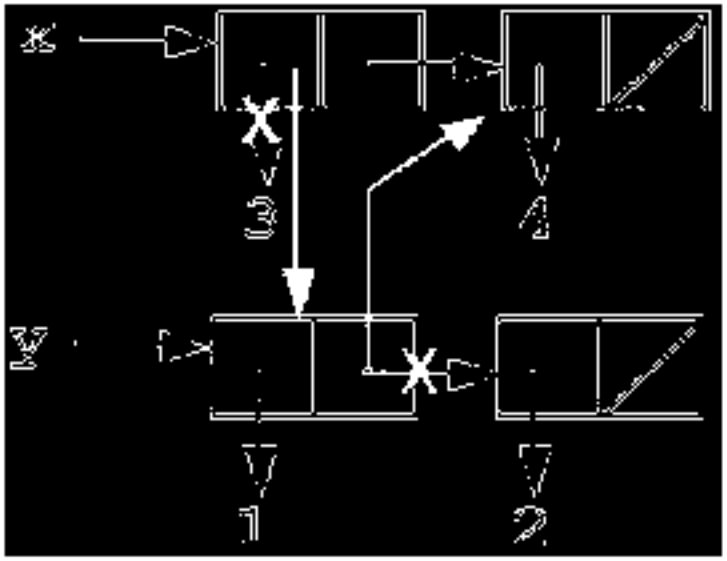
Последствия присваивания

- 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

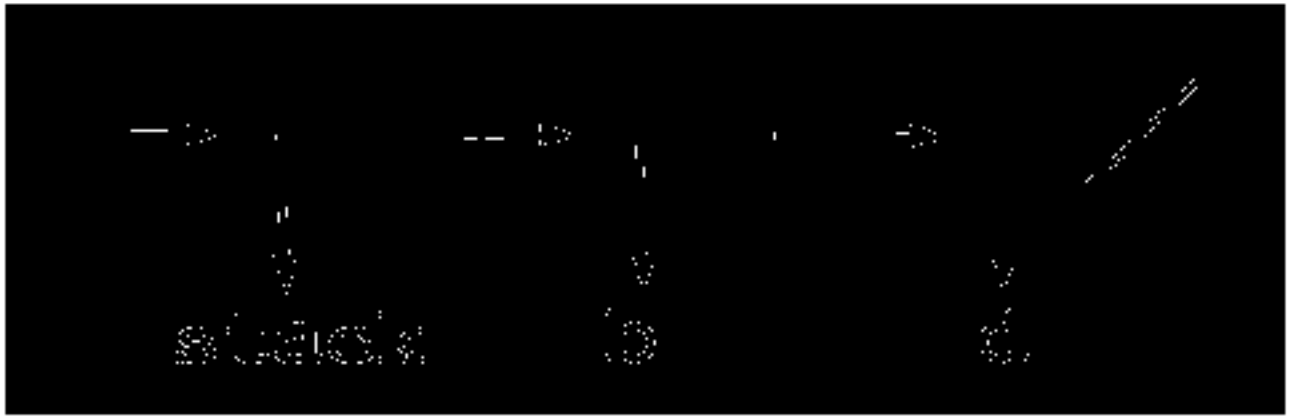
Промежуточный итог

- 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.

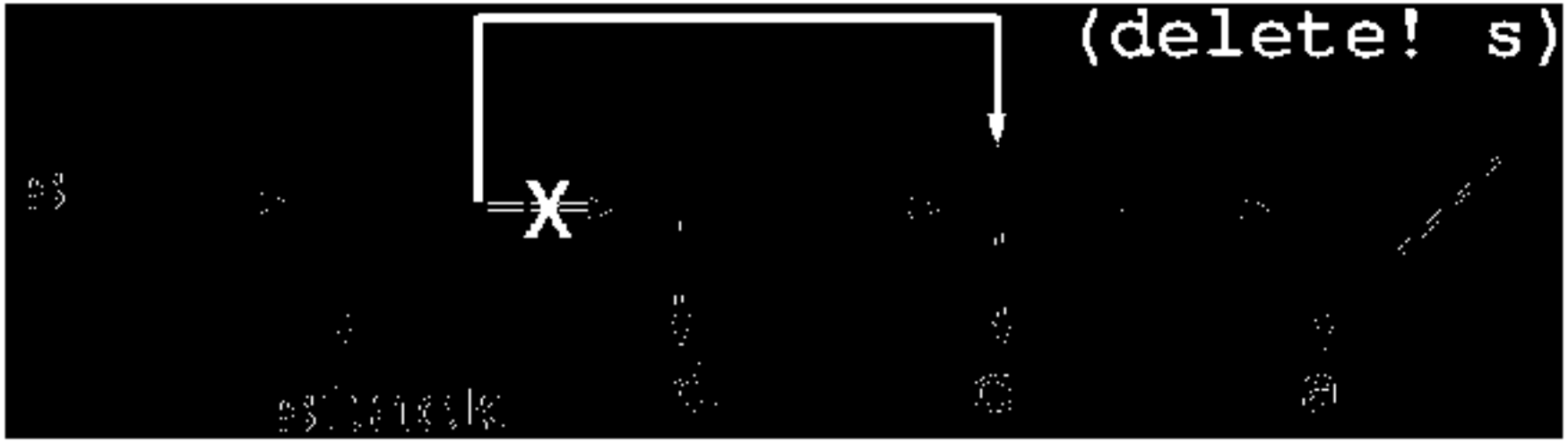


Стек. Реализация

```
(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

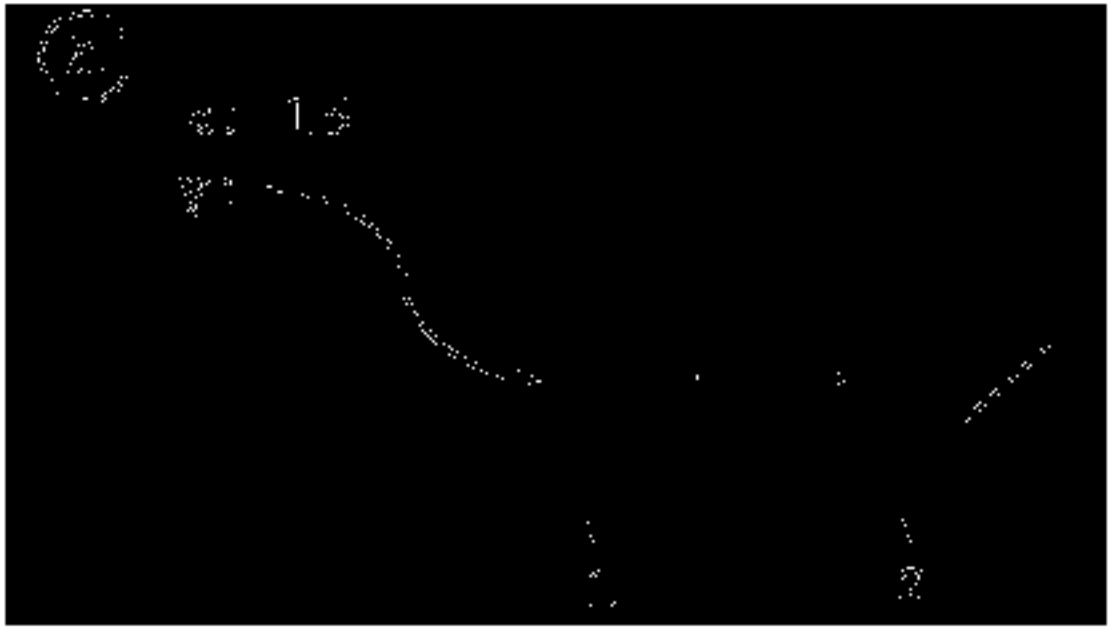
- в подстановочной модели имя – метка для значения
- в МВО имя – место для хранения значения

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

- Значение выражения зависит от окружения, в котором вычисляется выражение

Элементы MBO

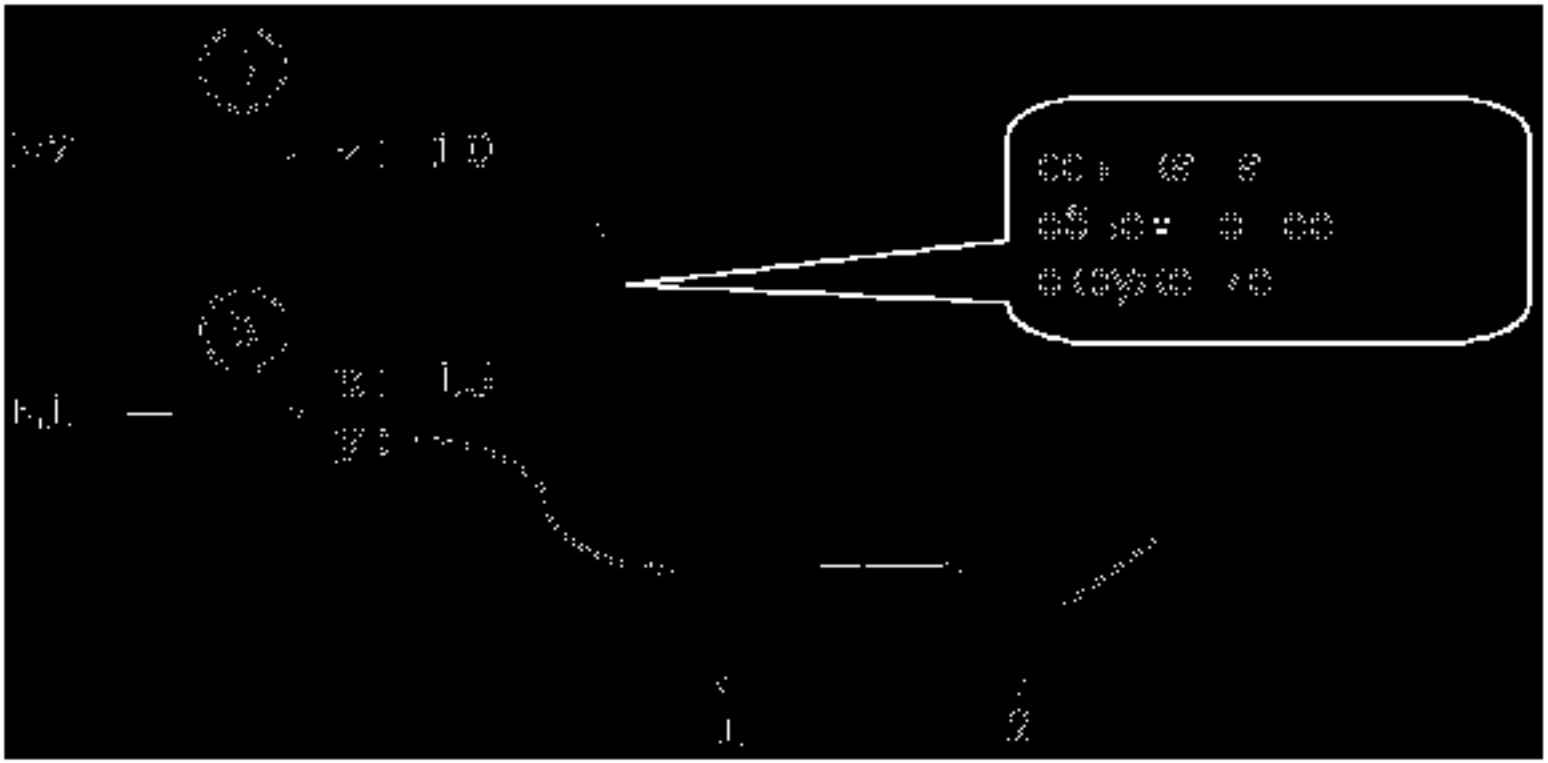
- Кадр – таблица связываний
- Связывание – пара имя : значение
- Пример:



- В кадре A содержатся два связывания

Элементы MBO

- 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.
- 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

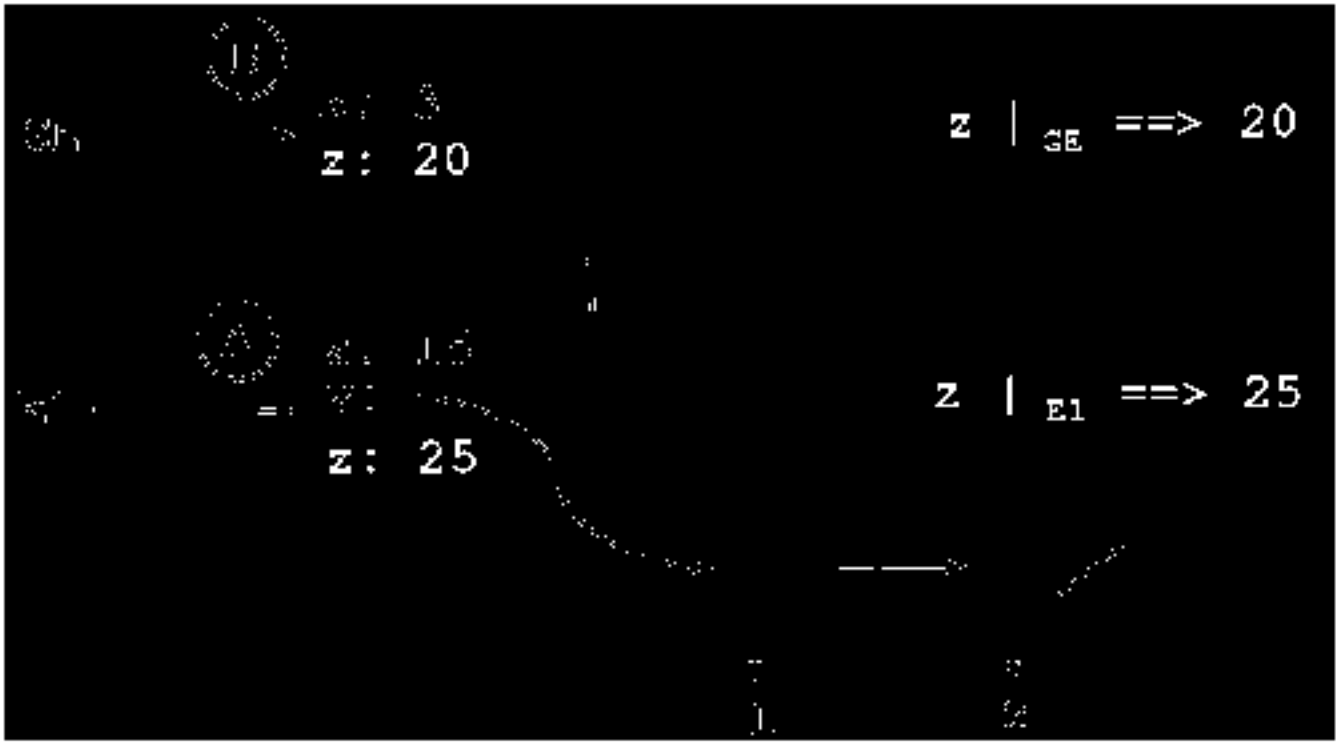
- Вычисление имени X в окружении E:
  - Найти первый кадр, в котором есть связывание для X.
  - Взять в качестве результата значение из этого связывания.

  - 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.

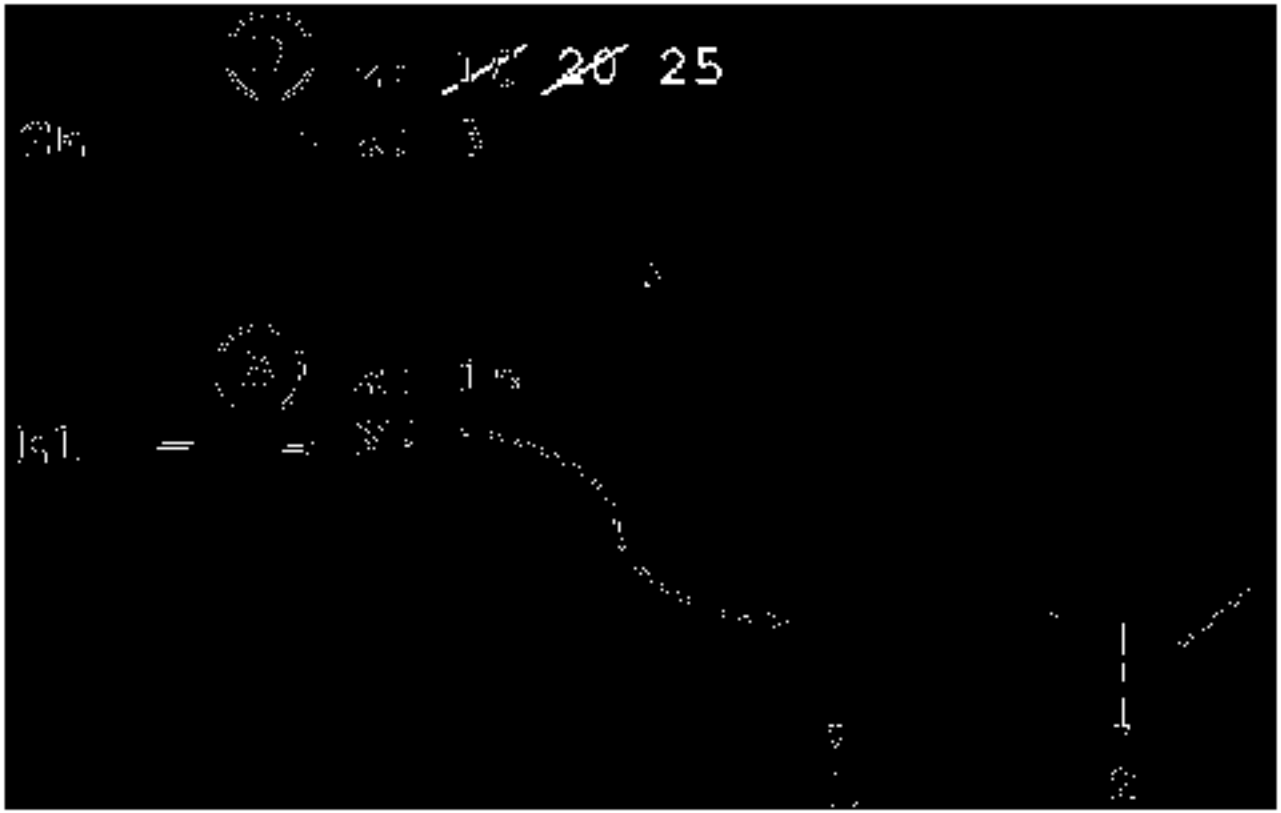
МВО. Правила вычислений

- Вычисление set! в окружении:

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

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

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

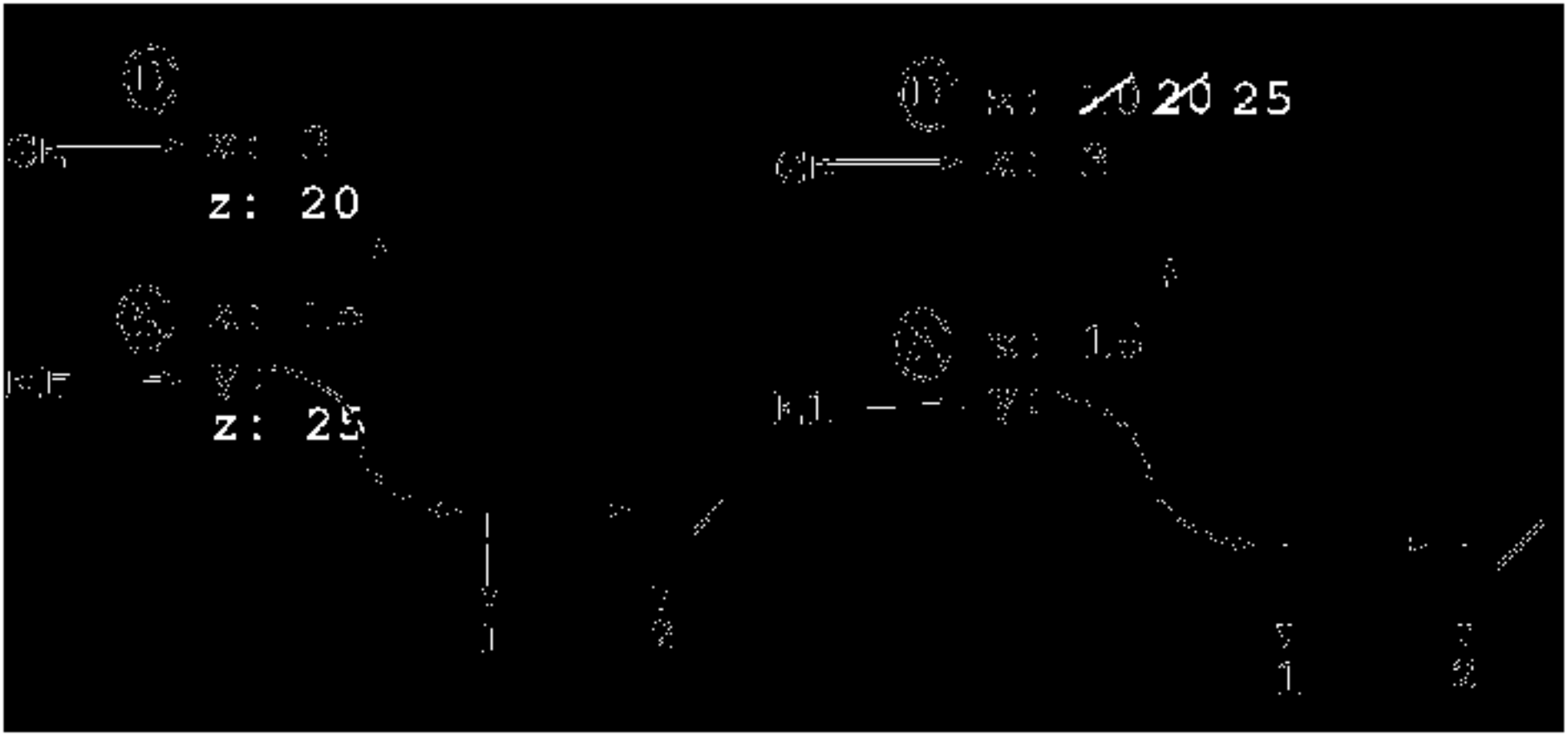


Если имя нигде не встретилось выдать ошибку!

define и set! вычисляются не одинаково

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

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

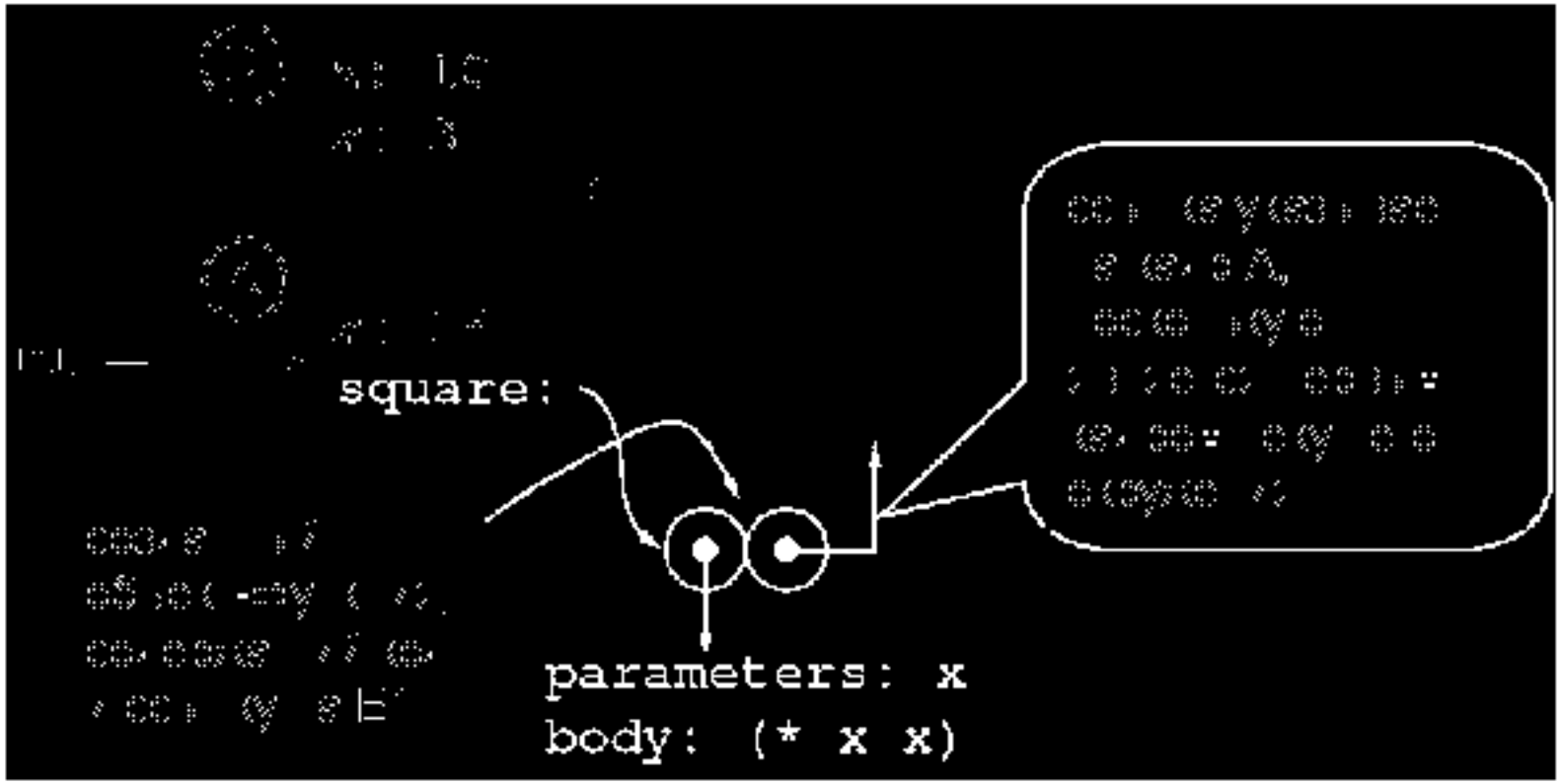


МВО. Правила вычислений

- Вычисление lambda в окружении:

  создать функцию с телом из lambda и ссылкой на текущее окружение

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

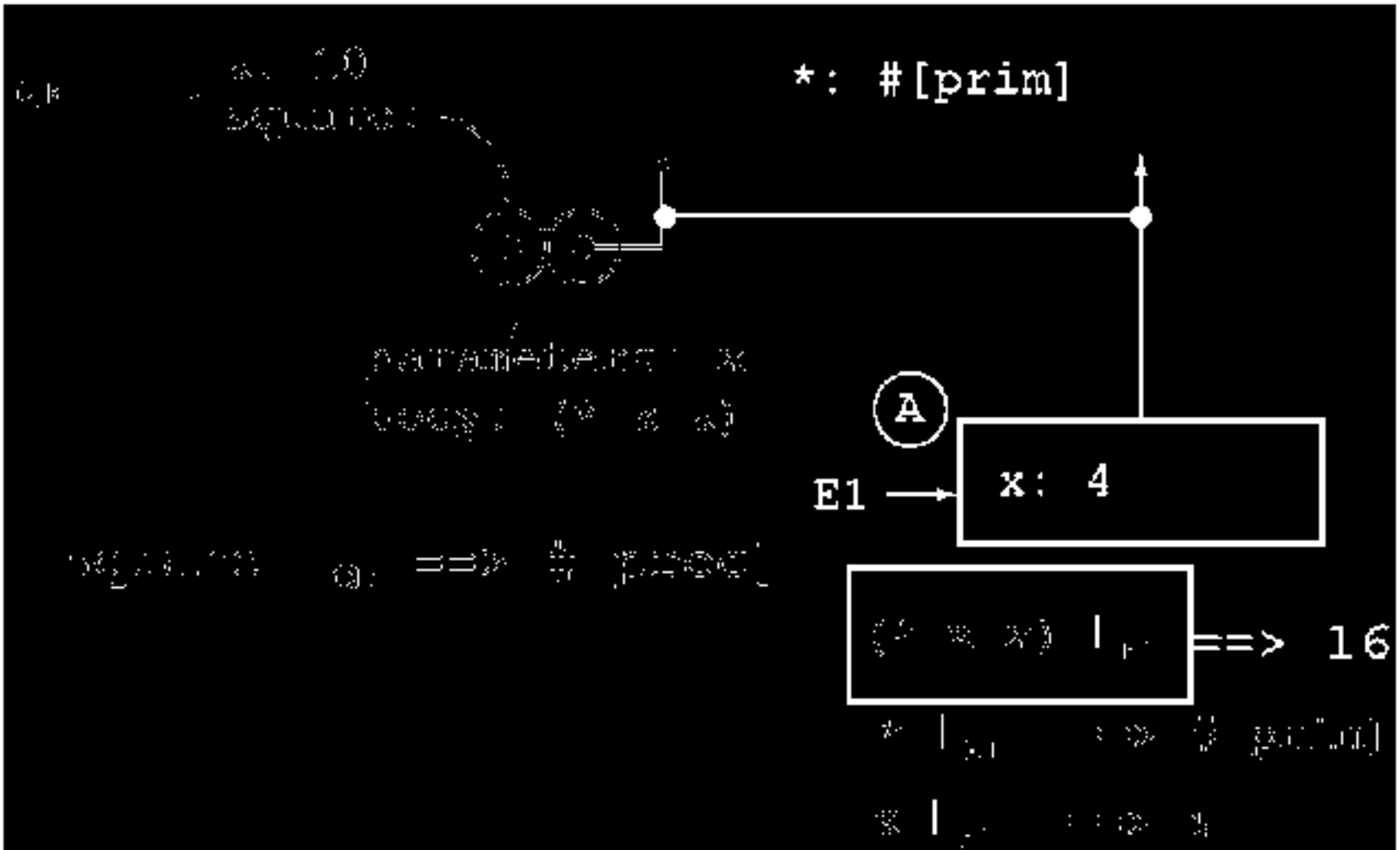


МВО. Правила вычислений

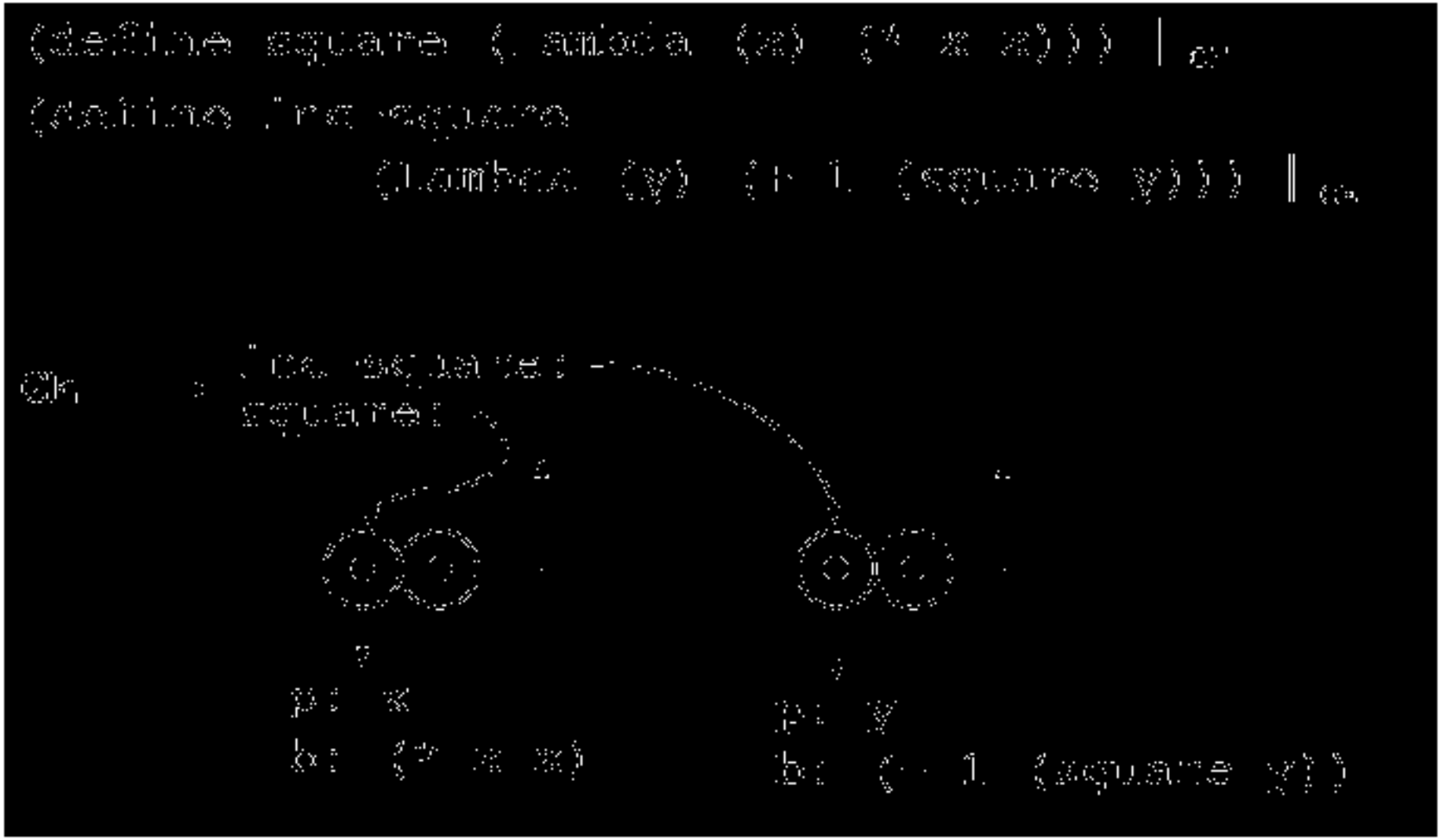
- Применение функции в окружении:
  - создать новый кадр A
  - сделать его первым кадром нового окружения E1
  - провести ссылку на объемлющее окружение от кадра A к окружению, на которое указывает ссылка объекта-функции
  - в кадр A добавить связывания всех аргументов функции со значениями из вызова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

Пример применения функции

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

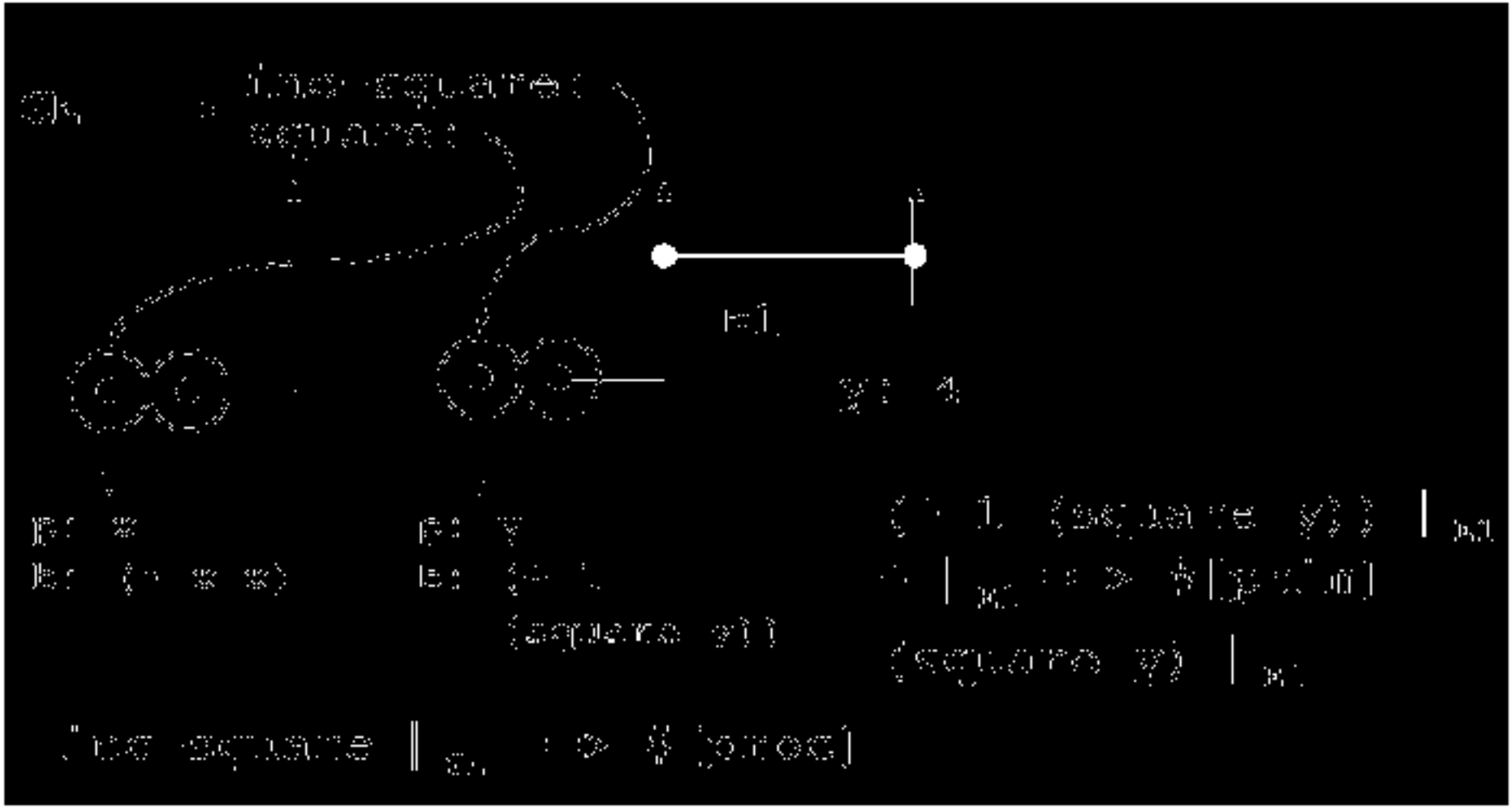


Ещё пример


(define square (lambda (x) (* x x)))
(define (re-square
(lambda (y) (+ 1 (square y))))
x: (* x x)
y: (+ 1 (square y))

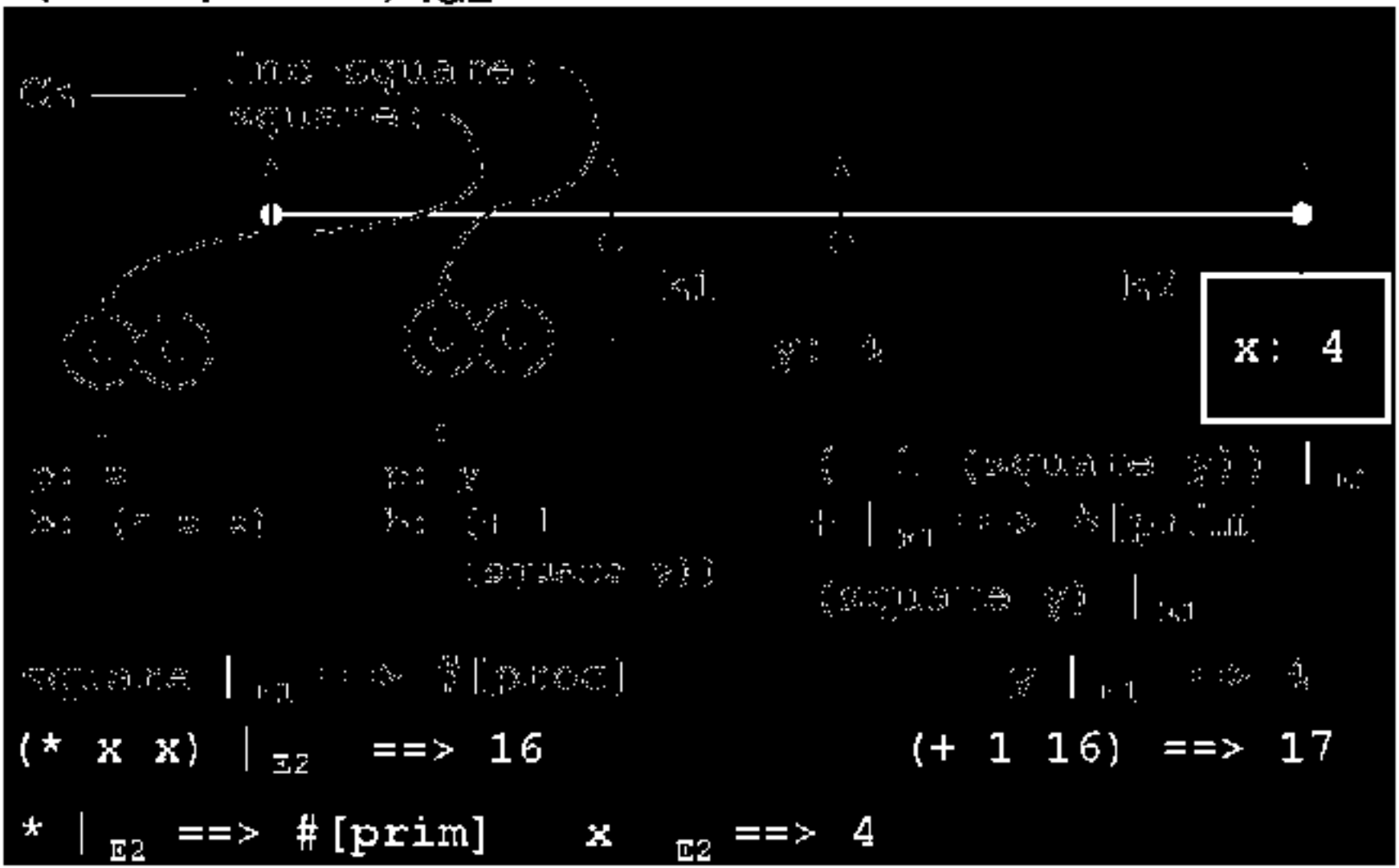
Продолжение примера

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



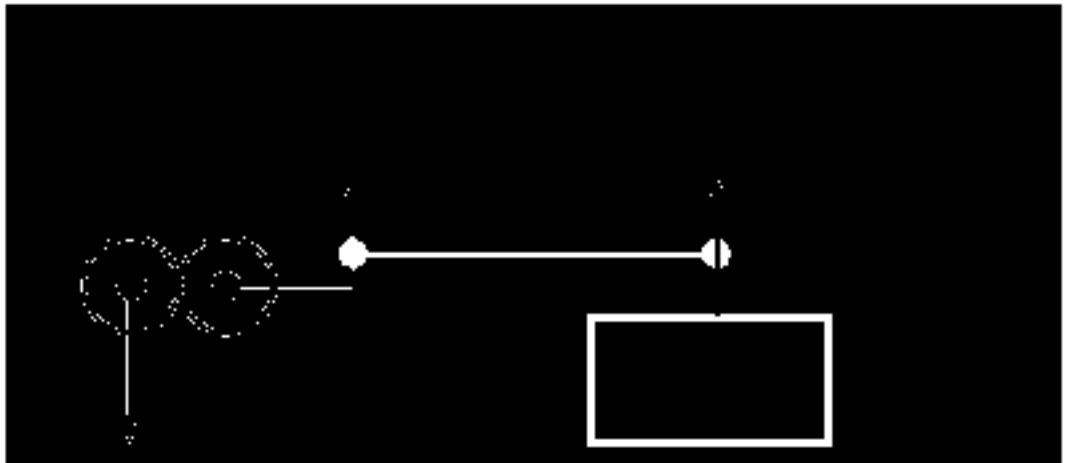
Продолжение примера

(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:

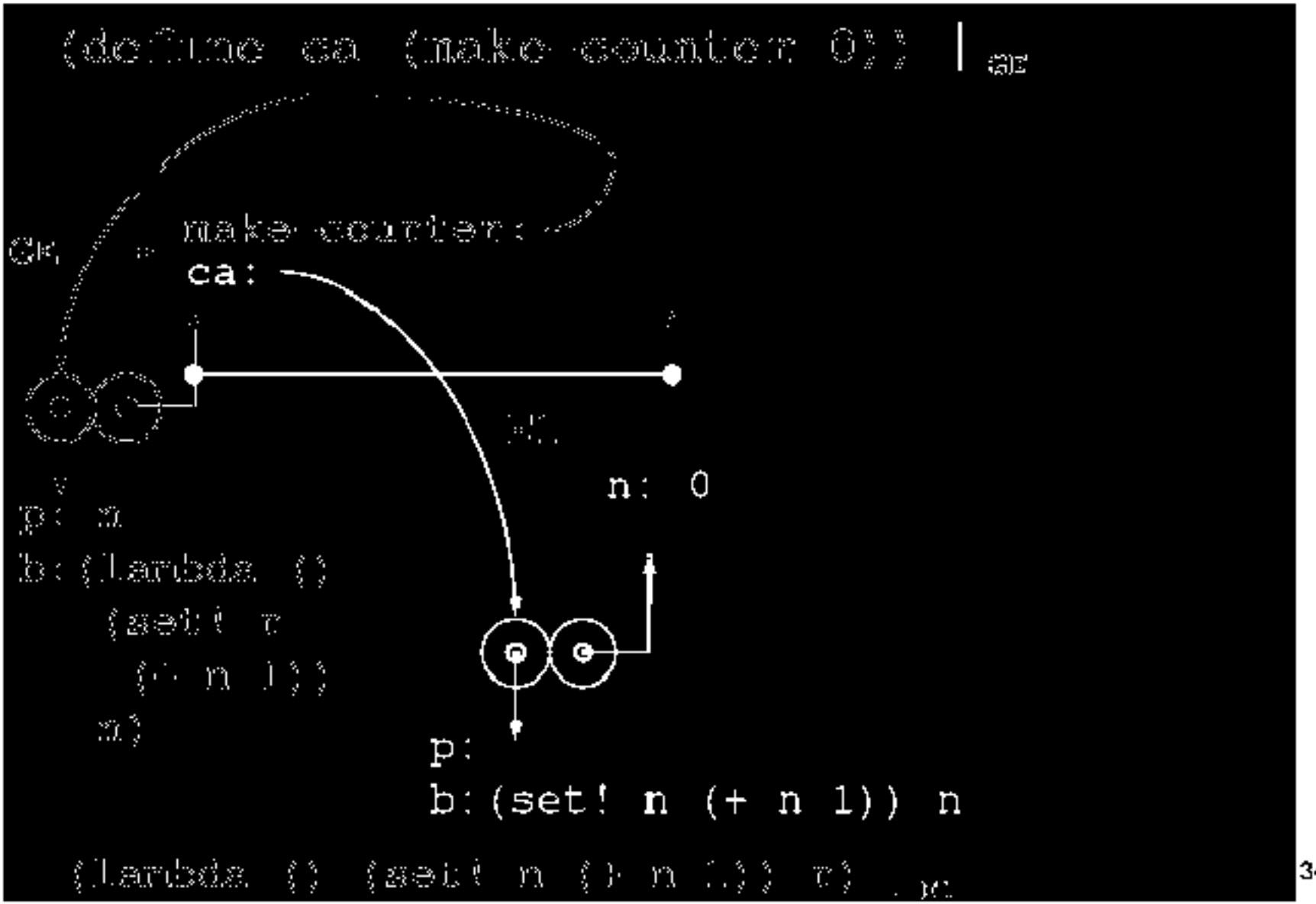


Снова пример

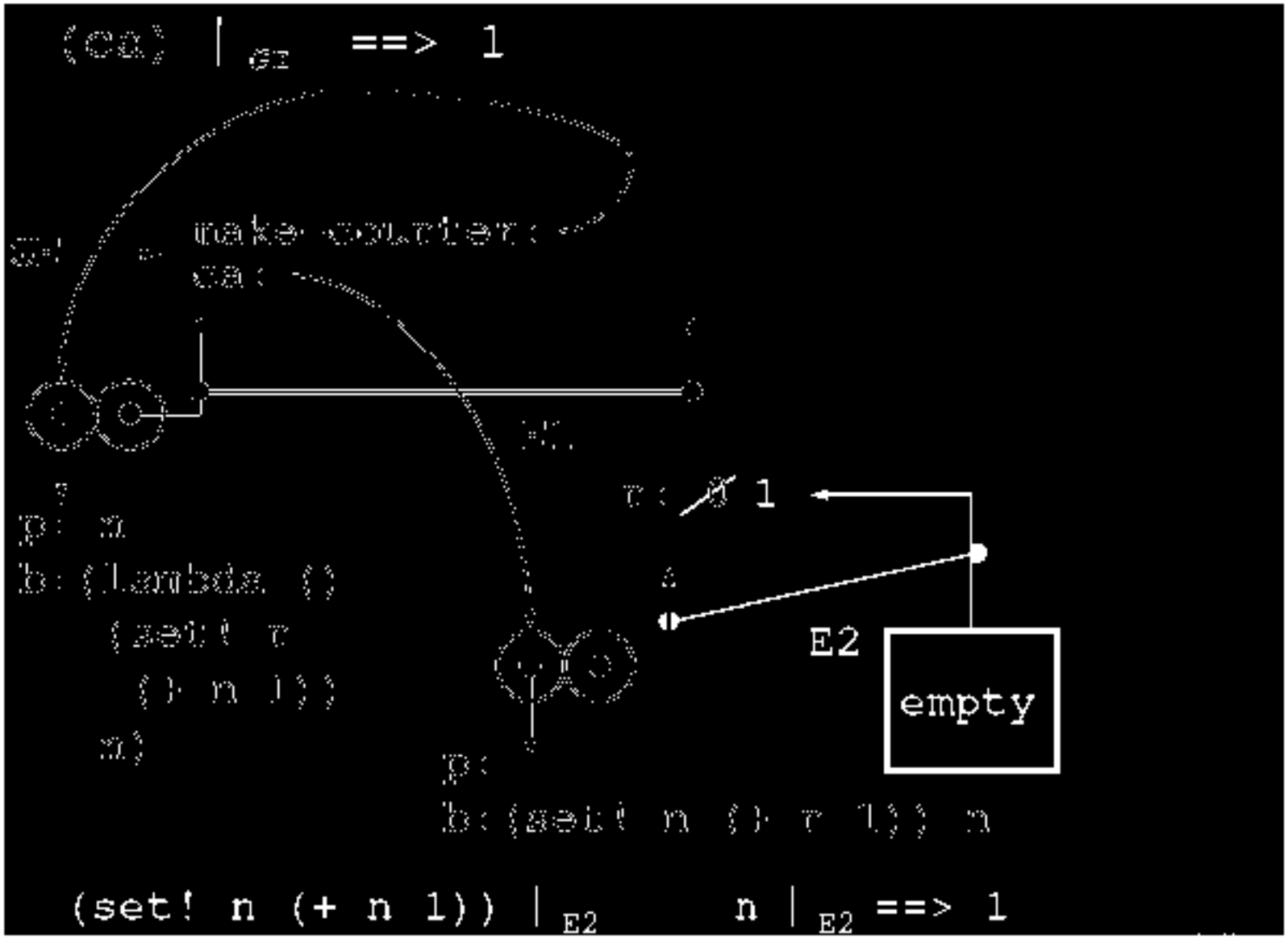
- Счётчик

```
(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
```

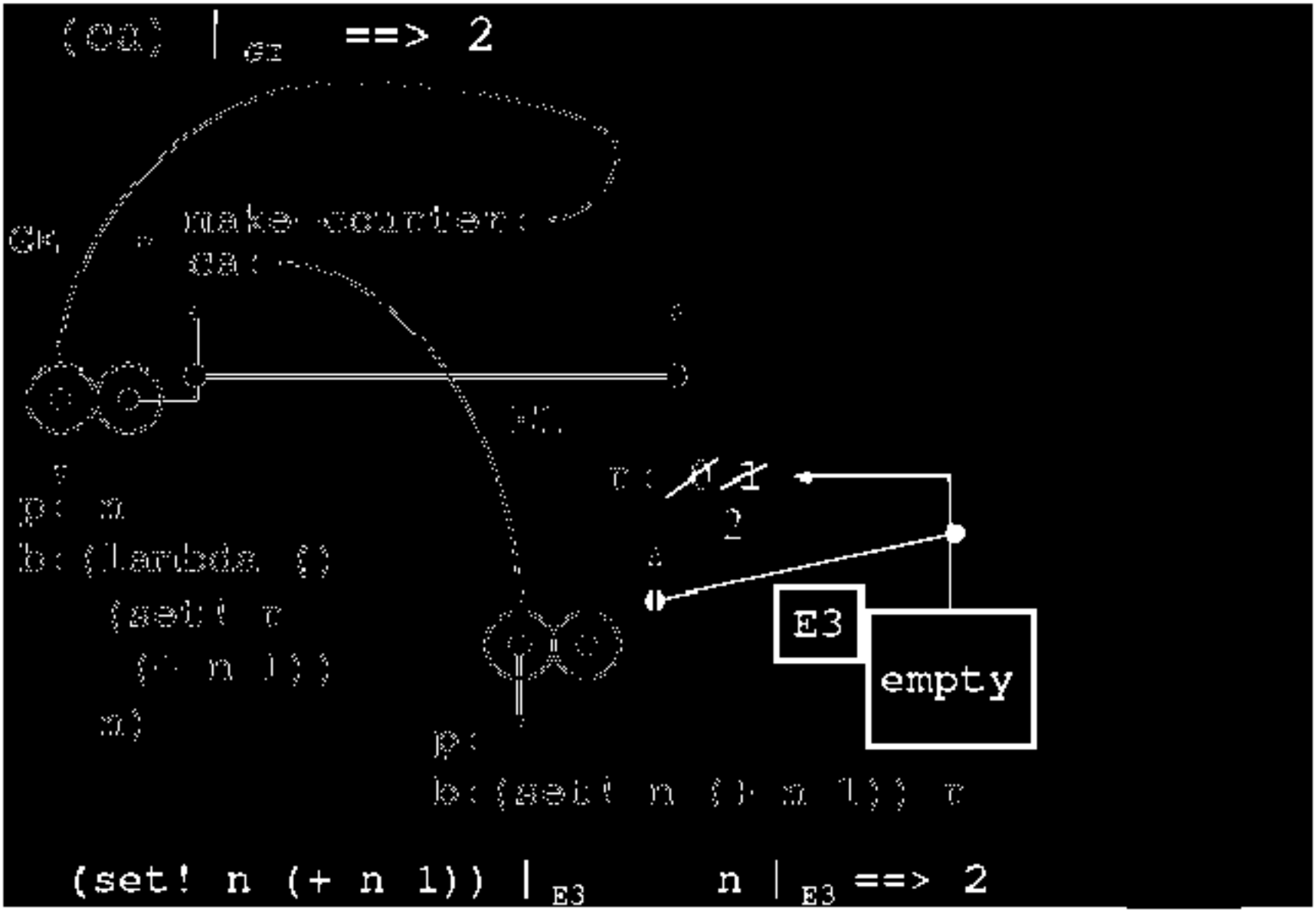
Счётчик. Продолжение



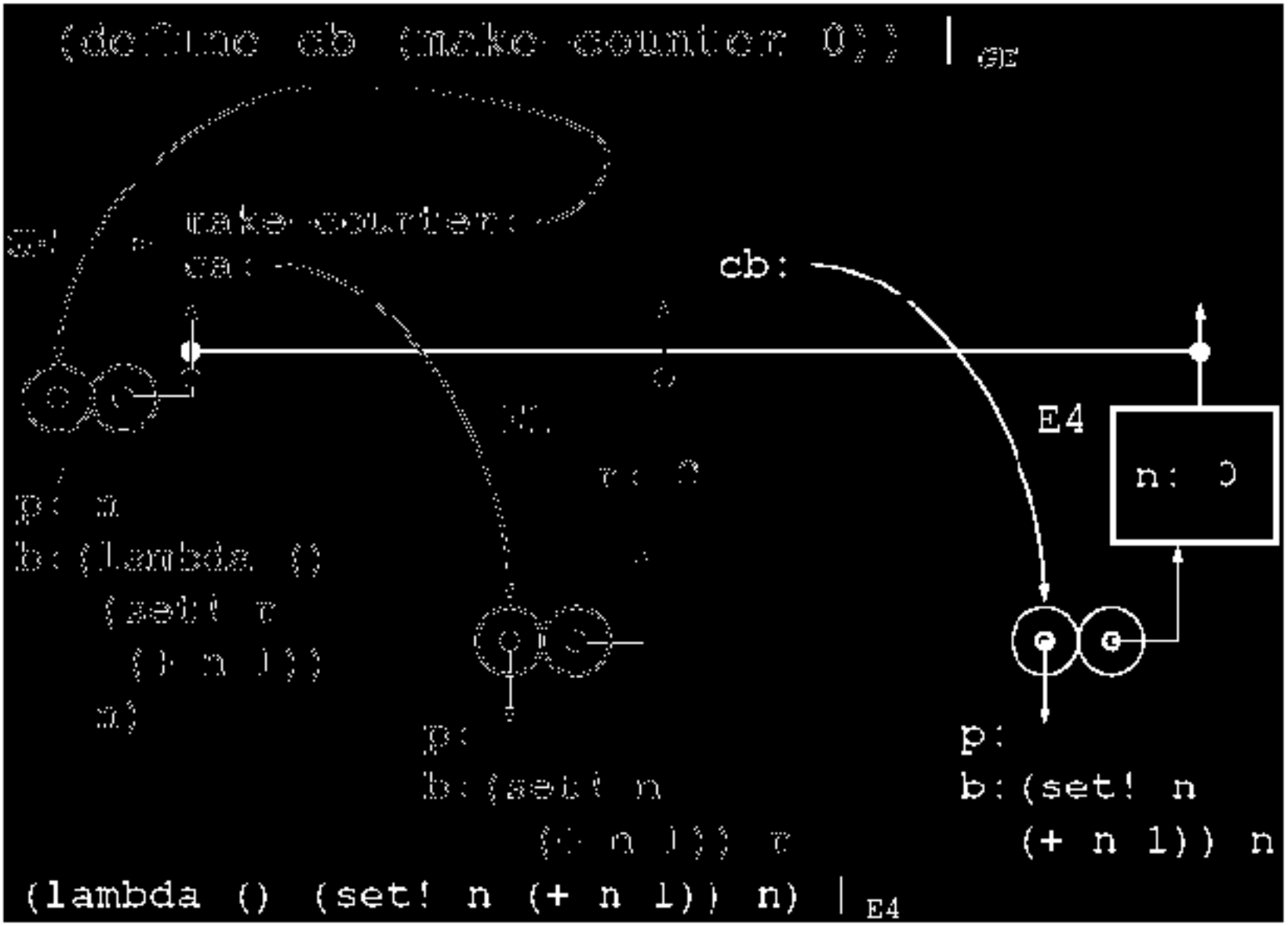
Счетчик. Продолжение



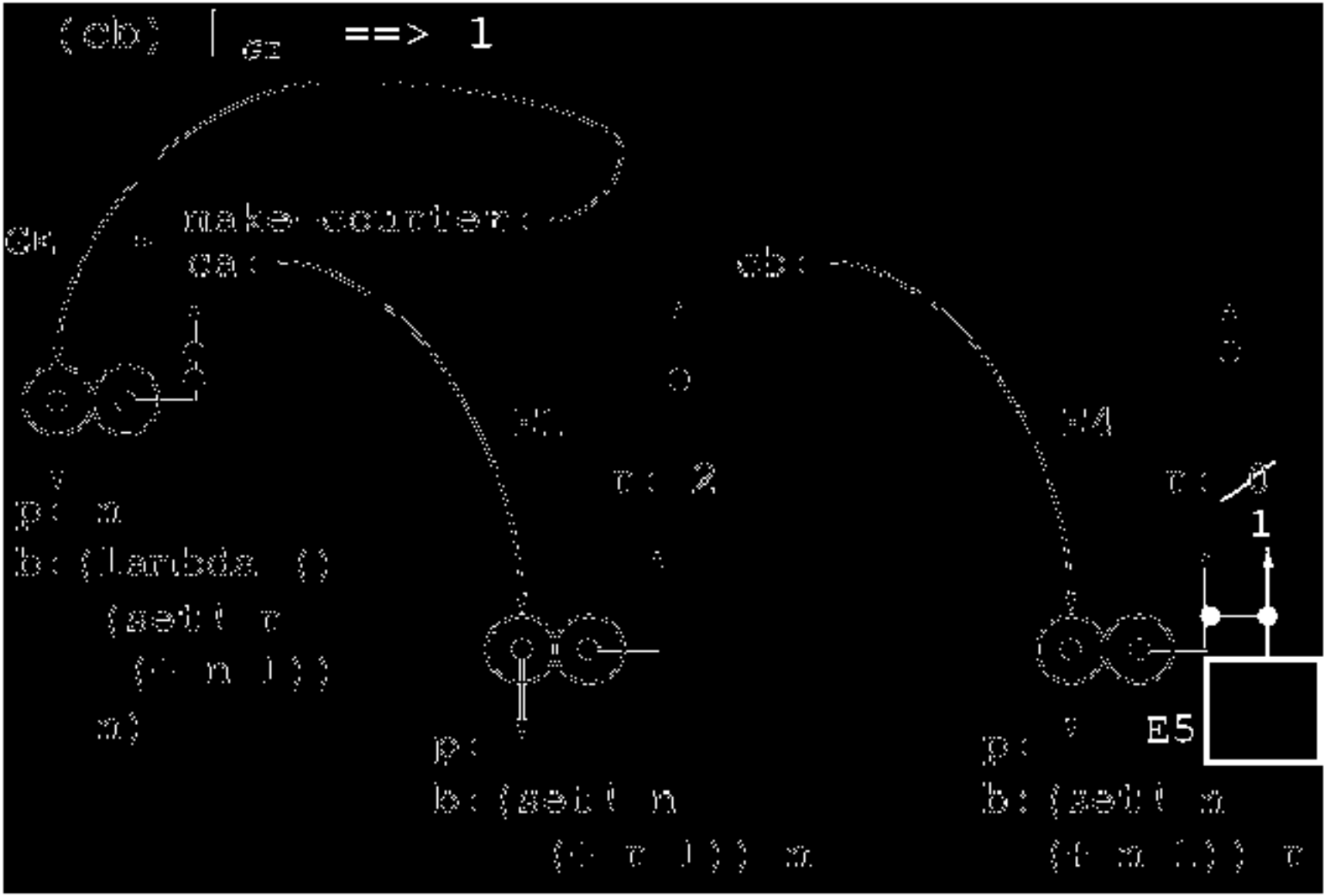
Счетчик. Продолжение



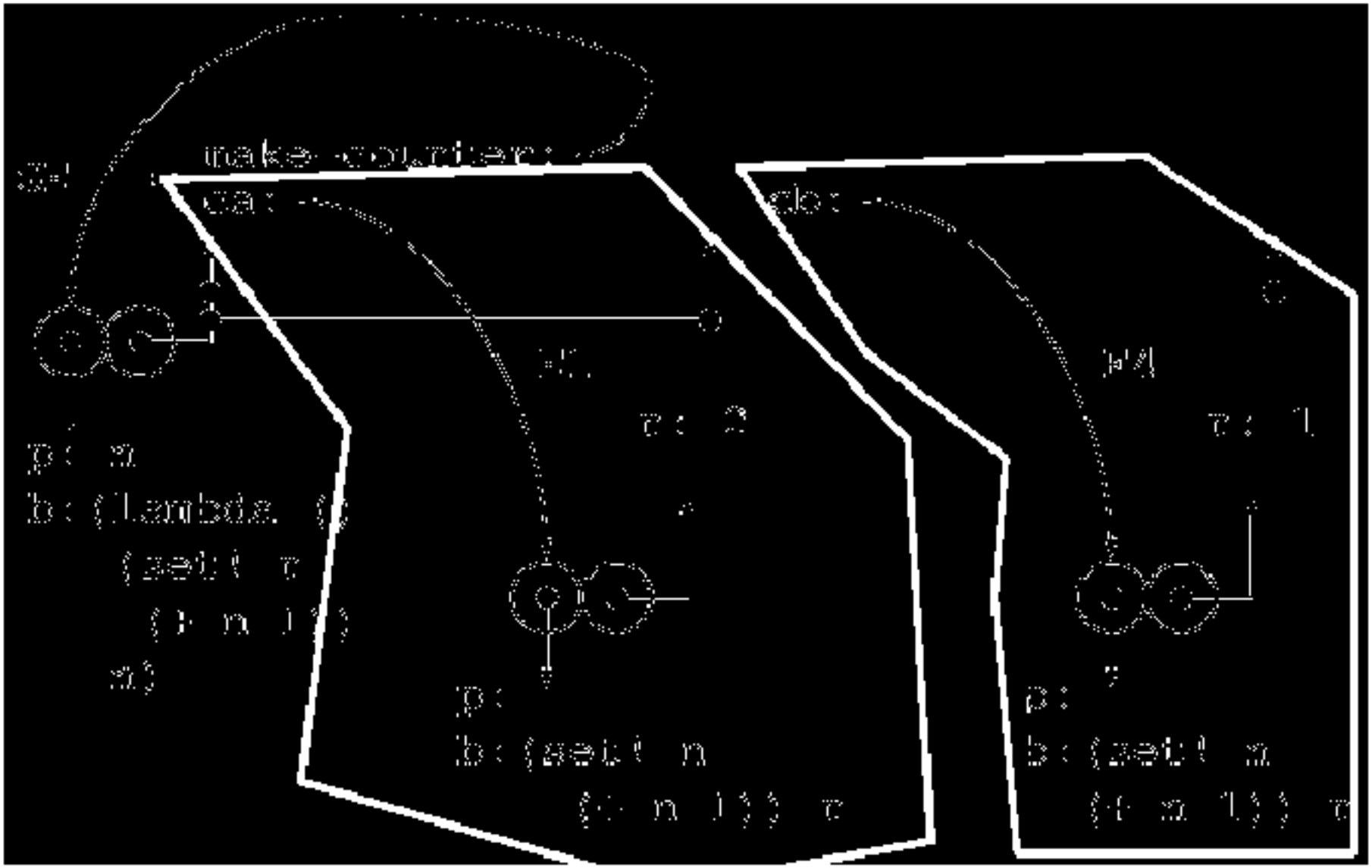
Счетчик. Продолжение



Счетчик. Продолжение



Счетчик. Продолжение



Таблица

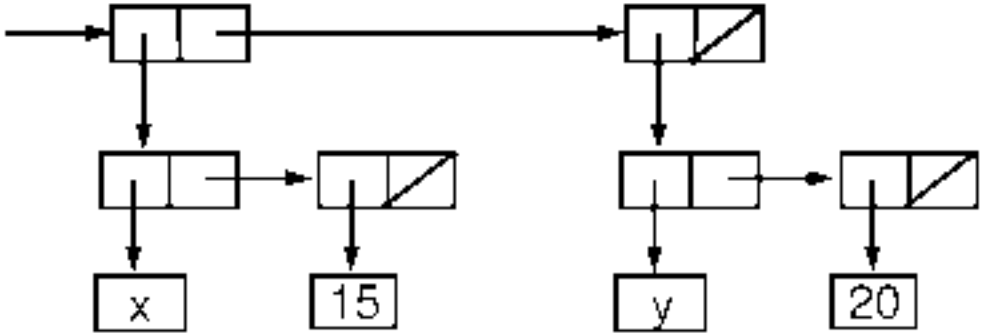
- конструктор (make-table)
- селектор (**lookup** <table> <key>)
- мутатор (**insert!** <table> <key> <value>)
- Список пар: (*table* (key1 . val1) (key2 . val2)…)

```
(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)

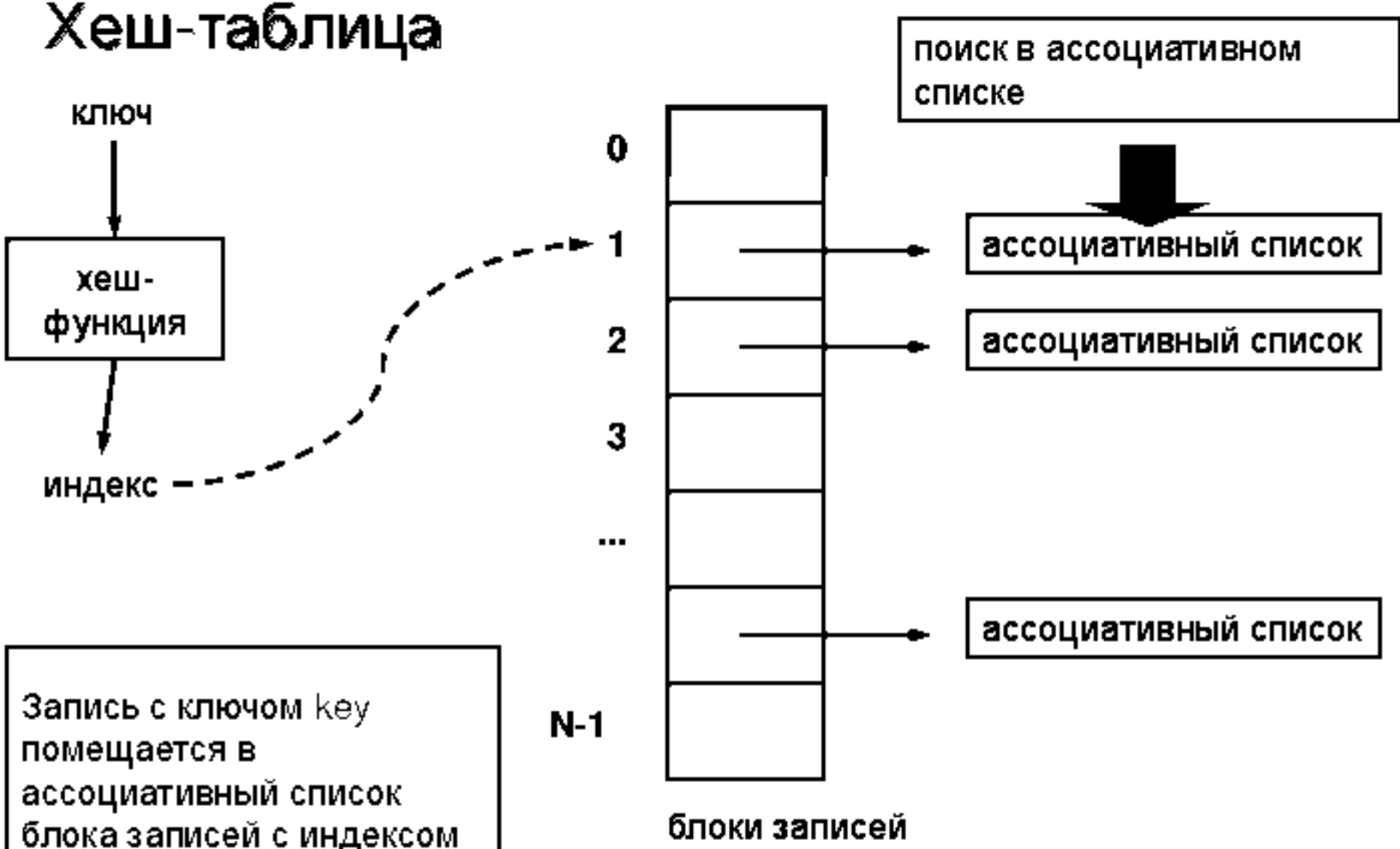
```
> (define beatles (vector 'john 'paul 'george 'pete))
> (vector-set! beatles 3 'ringo)
> beatles ==> '#(john paul george ringo)
> (vector-length beatles) ==> 4
```

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 7
Scheme и объектно-ориентированное программирование

Объектно-ориентированное программирование

- Абстракция данных, объединяющая состояние и функции
- Механизм передачи сообщений
- Объектные модели
  - диаграмма классов
  - диаграмма объектов
- Пример: «космические войны»

Абстракции

- Процедурные абстракции
- Абстракции данных

- Общее предназначение: выражать сложные понятия через более простые, скрывая детали.

- Вопросы:
  - Как лучше всего представить систему набором абстракций?
  - Как сделать простым расширение системы?
    - Добавлять новые типы данных?
    - Добавлять новые функции?

Подход «от данных»

- Структуры данных
  - Сборка сложных структур с помощью cons:
    point, line, 2dshape, 3dshape
  - Использование заглавного звена – тега, указывающего тип данных
    (define (make-point x y) (list 'point x y))
  - Реализация структуры данных требует набора функций: конструктора, селекторов, мутаторов, …

- Обобщённые операции

```
(define (scale x factor)
      (cond ((point? x)  (point-scale x factor))
            ((line? x)   (line-scale x factor))
            ((2dshape? x)(2dshape-scale x factor))
            ((3dshape? x)(3dshape-scale x factor))
            (else (display "unknown type"))))
```

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию

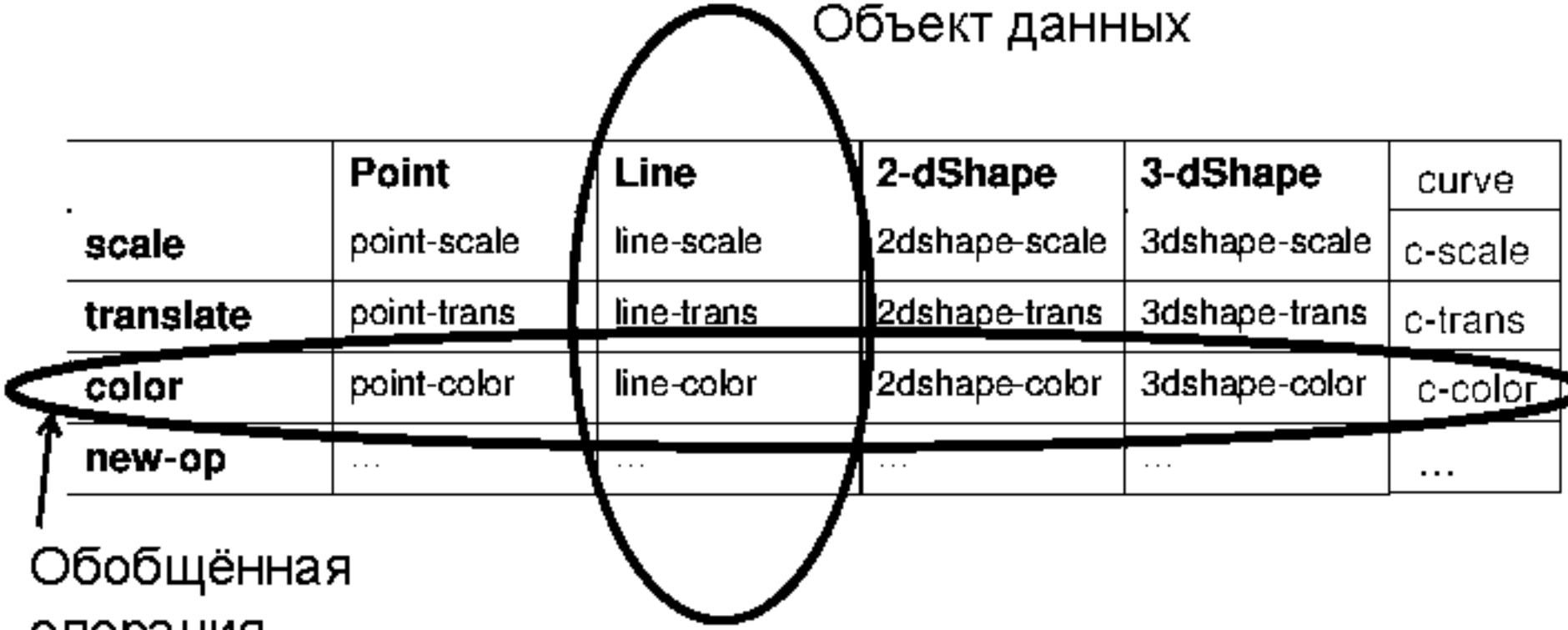
	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color
new-op	…	…	…	…

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию
- при добавлении нового типа данных
  - необходимо переписать каждую обобщённую операцию

	Point	**Line**	**2-dShape**	**3-dShape**	curve
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	c-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans	c-trans
color	point-color	line-color	2dshape-color	3dshape-color	c-color
new-op	…	…	…	…	…

Два взгляда на мир



	Point	**Line**	**2-dShape**	**3-dShape**	curve
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	c-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans	c-trans
color	point-color	line-color	2dshape-color	3dshape-color	c-color
new-op	…	…	…	…	…

Что такое объект данных?

- Это структура данных
  - соединённая с набором собственных операций
  - для которой есть общее понятие (line), и экземпляры (line17)
  - экземпляр хранит свои свойства (состояние)

- Мы изобрели ООП!

ООП в Scheme: Функции и состояние

- У функции есть
  - **параметры** и **тело** описанные в lambda-выражении
  - **окружение**, где хранятся связывания её имён

- Можно использовать функцию для хранения (и сокрытия) данных и предоставления доступа к ним.
- При вызове функции создаётся новое окружение.
- Нужно иметь доступ к этому окружению.
  - для доступа нужны операции (конструктор, селекторы, мутаторы, ...)
  - мутаторы изменяют состояние

Пример: мутируемые пары как объекты

```
(define (mcons x y)
  (lambda (msg)
    (cond ((eq? msg 'MCAR) x)
          ((eq? msg 'MCDR) y)
          ((eq? msg 'MPAIR?) #t)
          (else (display "wrong msg")))))

(define (mcar p)   (p 'MCAR))
(define (mcdr p)   (p 'MCDR))
(define (mpair? p)
   (and (procedure? p) (p 'MPAIR?)))
```

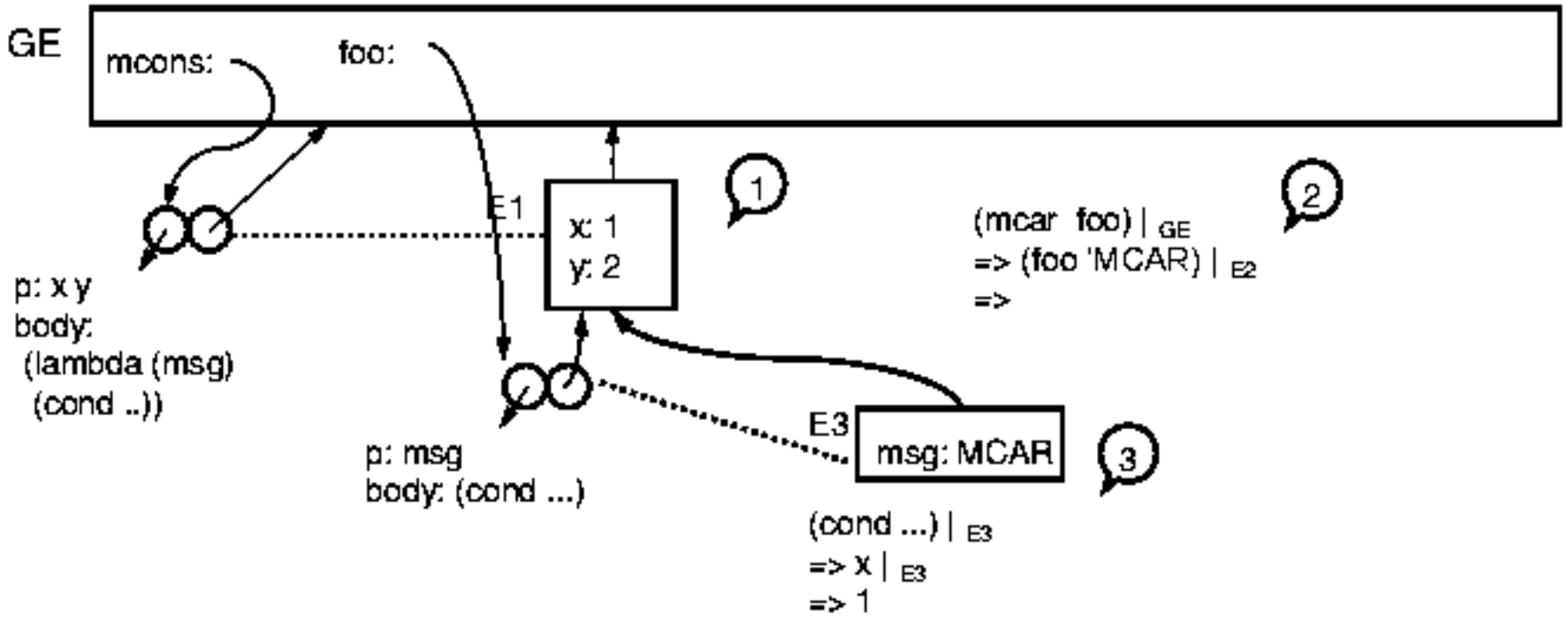
Пара-объект с точки зрения MBO

```
(define (mcons x y)
 (lambda (msg)
  (cond ((eq? msg 'MCAR) x)
      ((eq? msg 'MCDR) y)
      ((eq? msg 'MPAIR?) #t)
      (else ...)
```

(define foo (mcons 1 2)) ①

(mcar foo) ②

====> (foo 'MCAR) ③



Мутация пары как изменение состояния объекта

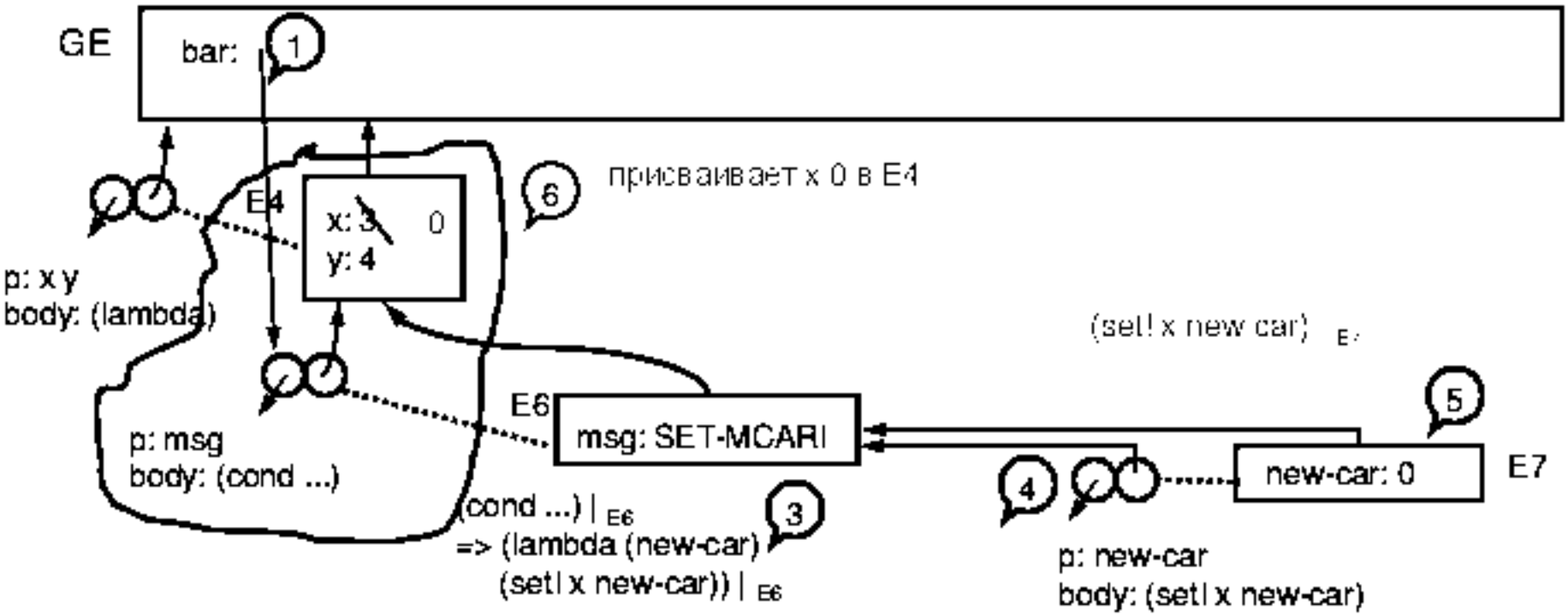
```
(define (mcons x y)
  (lambda (msg)
    (cond ((eq? msg 'MCAR) x)
          ((eq? msg 'MCDR) y)
          ((eq? msg 'MPAIR?) #t)
          ((eq? msg 'SET-MCAR!)
           (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-MCDR!)
           (lambda (new-cdr) (set! y new-cdr)))
          (else ...)
(define (set-mcar! p new-car)
  ((p 'SET-MCAR!) new-car))
(define (set-mcdr! p new-cdr)
  ((p 'SET-MCDR!) new-cdr))
```

Пример мутации в MBO

(define bar (mcons 3 4)) ①

(set-mcar! bar 0) ②

(set-mcar! bar 0) |GE
=> ((bar 'SET-MCAR!) 0) |E5



Стиль передачи сообщений

- дадим имена lambda и сделаем их приватными

```
(define (mcons x y)
  (define (change-car new-car) (set! x new-car))
  (define (change-cdr new-cdr) (set! y new-cdr))
  (lambda (msg . args)
    (cond ((eq? msg 'MCAR) x)
          ((eq? msg 'MCDR) y)
          ((eq? msg 'MPAIR?) #t)
          ((eq? msg 'SET-MCAR!) (change-car (first args)))
          ((eq? msg 'SET-MCDR!) (change-cdr (first args)))
          (else ...))))
```

- перепишем селектор и мутатор

```
(define (mcar p)
        (p 'MCAR))
(define (set-mcar! p val)
        (p 'SET-MCAR! val))
```

Стили программирования: функциональный и объектно-ориентированный

- Функциональное программирование:
  - Система организуется набором функций, обрабатывающих данные

  (do-something <data> <arg> ...)

  (do-another-thing <data>)
- ООП:
  - Система организуется как набор объектов, обменивающихся сообщениями

  (<object> 'do-something <arg>)

  (<object> 'do-another-thing)
  - Объект инкапсулирует данные и операции

Основные термины ООП

- **Класс**
  - описывает общую структуру и поведение однотипных экземпляров
  - в Scheme класс – это функция – операция-конструктор.

- **Экземпляр**:
  - Отдельный объект некоторого класса
  - в Scheme, объект – это функция-обработчик сообщений, созданная операцией-конструктором некоторого класса

Пример. «Космические войны»

- В игре участвуют
  - корабли
  - планеты
  - объекты других типов
- Примеры экземпляров классов
  - Millenium Falcon
  - Enterprise
  - Earth

Корабль

```
(define (make-ship position velocity num-torps)
   (define (move)
     (set! position (add-vect position ...)))
   (define (fire-torp)
     (cond ((> num-torps 0) ...)
           (else 'fail)))
   (lambda (msg)
     (cond ((eq? msg 'position) position)
           ((eq? msg 'velocity) velocity)
           ((eq? msg 'move) (move))
           ((eq? msg 'attak) (fire-torp))
           (else (display "wrong message to ship")))))
```

Класс корабль

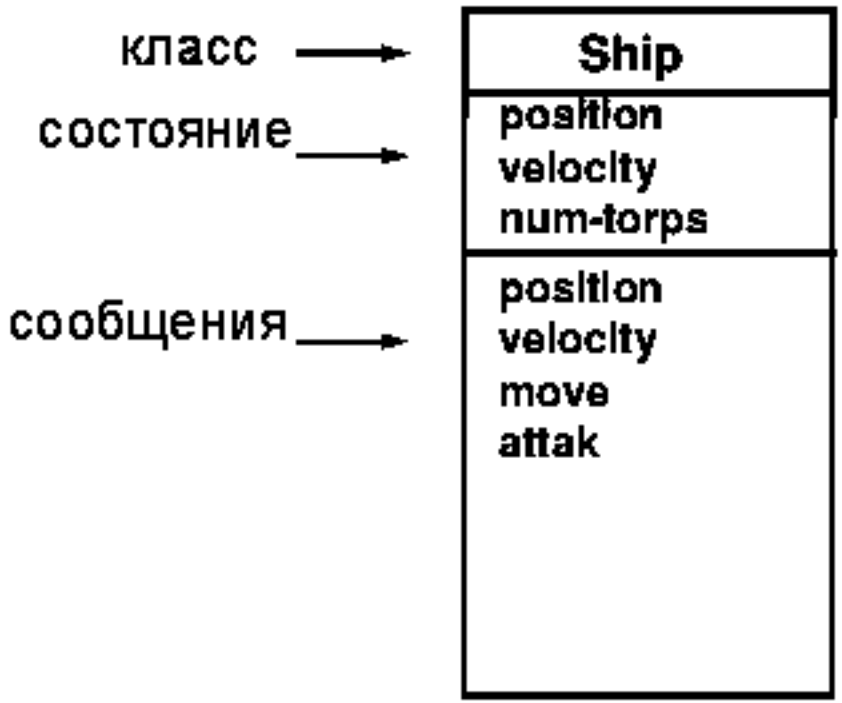


Диаграмма объектов

```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(define war-bird
  (make-ship (make-vect -10 10) (make-vect 10 0) 10))
```

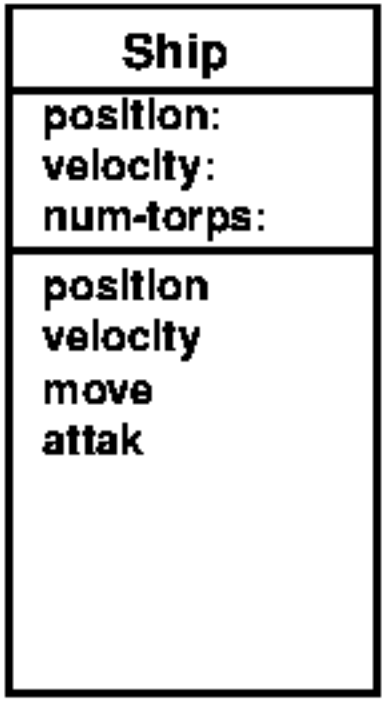
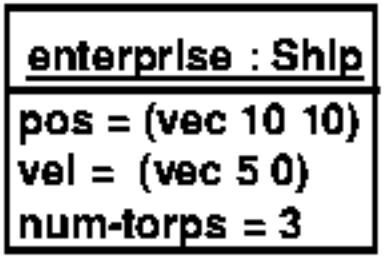
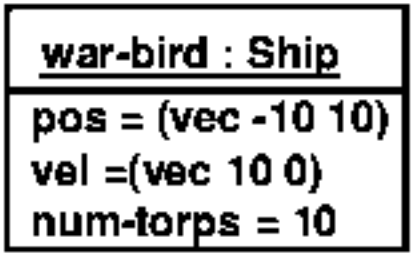


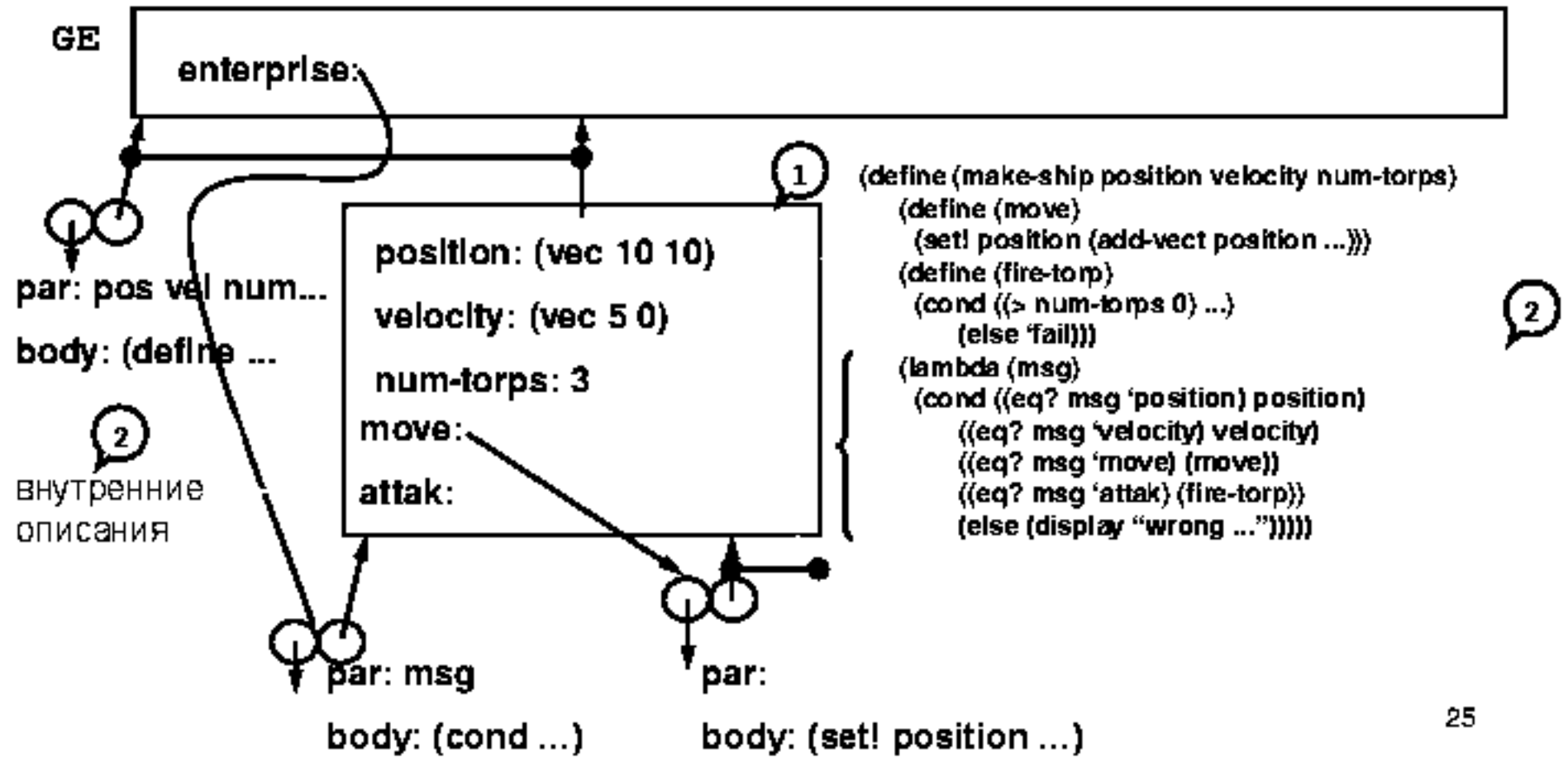
Диаграмма объектов





С точки зрения модели вычислений с окружениями

```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(enterprise 'move) ==>
(enterprise 'position) ==>  ?
```

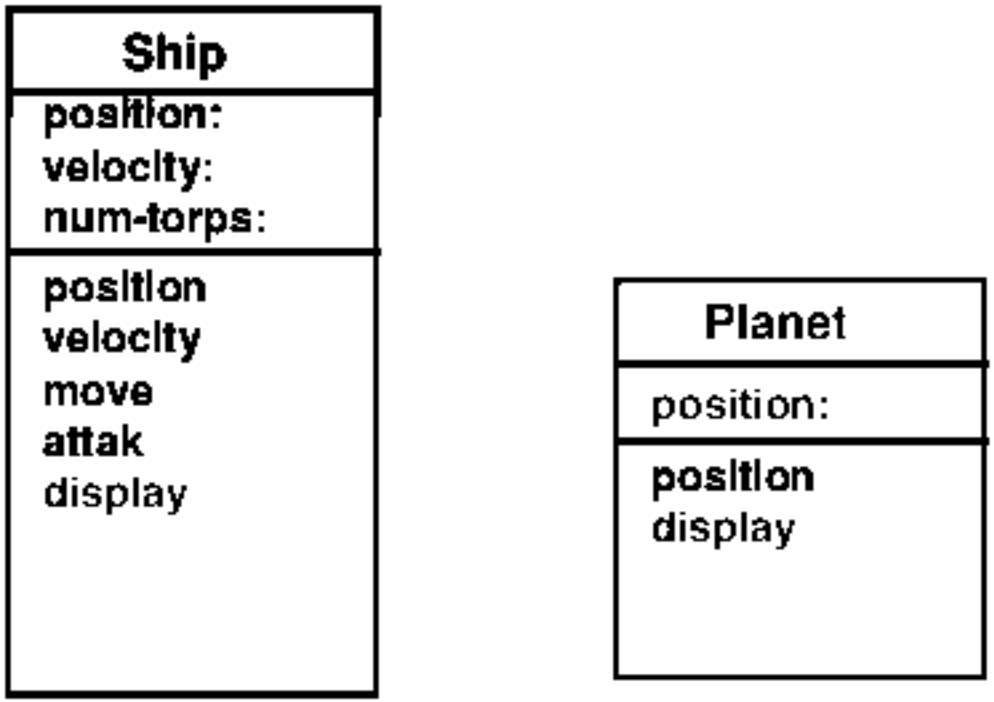


Продолжим наполнять мир «космических войн»

- Добавим планеты – класс Planet

- Добавим вывод
  - отображение объектов на экране
  - допустим, что вывод умеет делать функция draw
  - добавим сообщение ‘display, обработка которого заключается в вызове draw с нужными параметрами.

Обновленная диаграмма классов



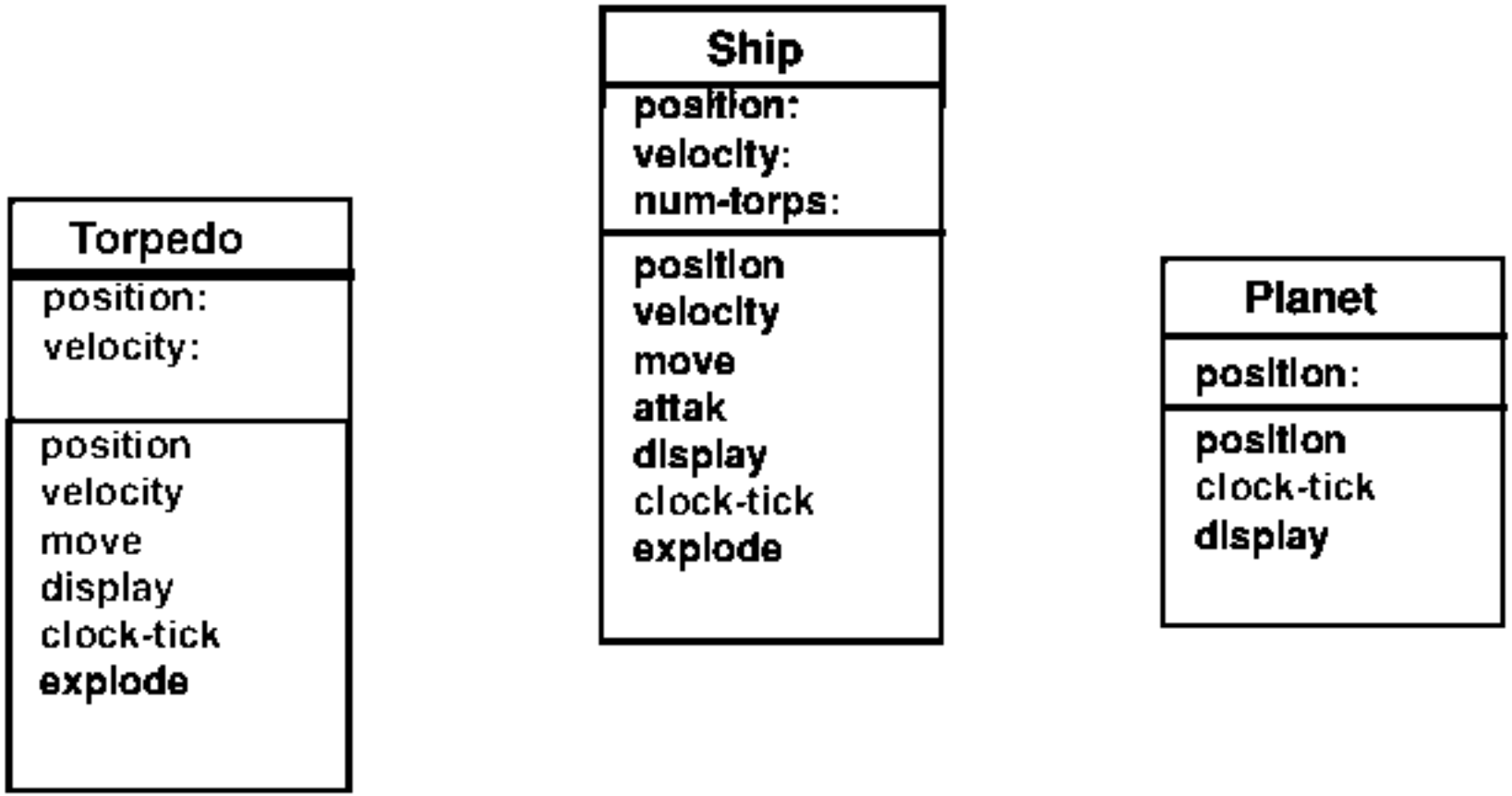
Класс Планета

```
(define (make-planet position)
  (lambda (msg)
    (cond
        ((eq? msg ‘position) position)
        ((eq? msg ‘display) (draw ...))
        (else (display “wrong message to planet”)))))

(define earth (make-planet (make-vect 0 0)))
(earth ‘position) ==> ...
```

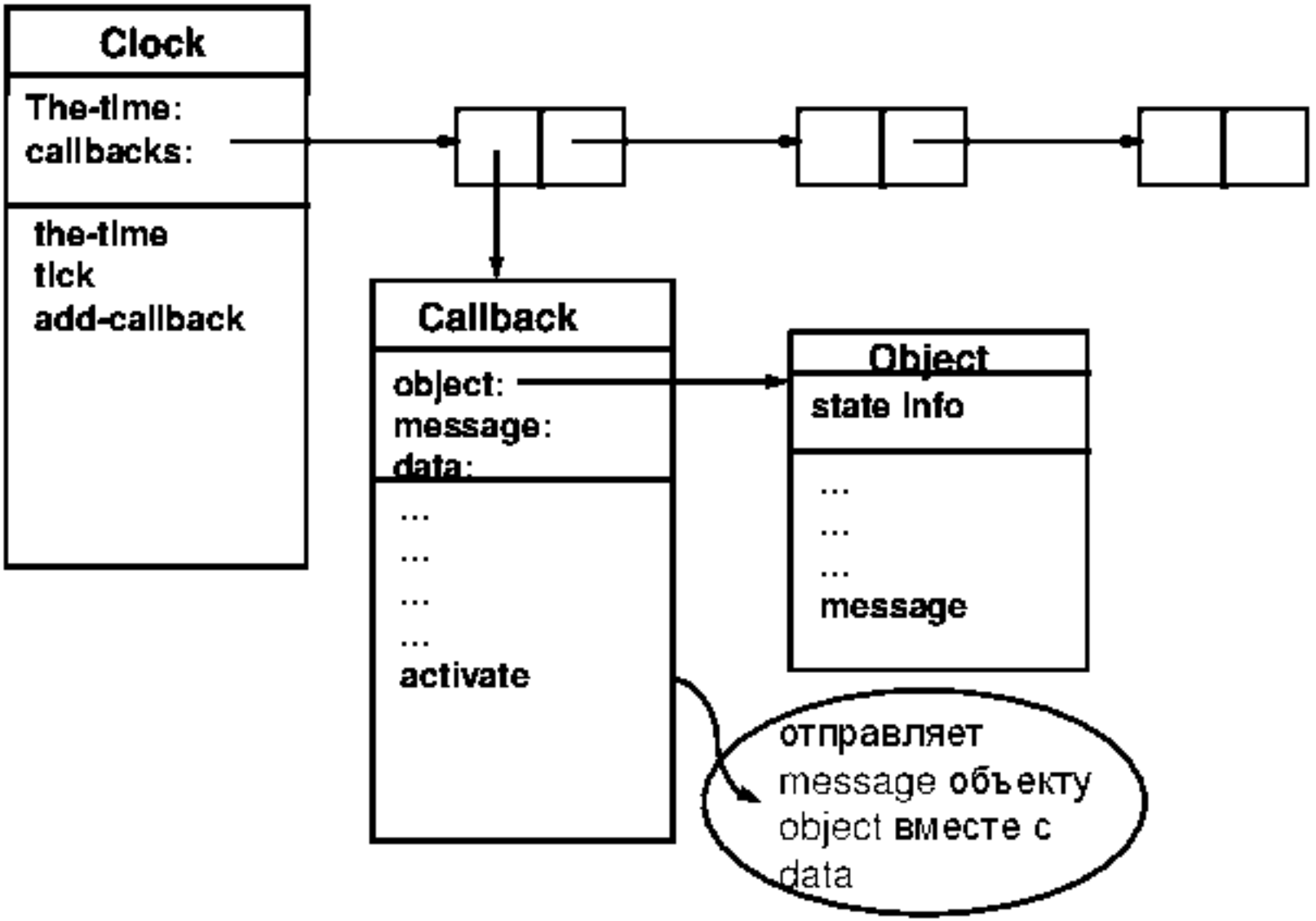
Запустим время

- Движение в космосе
  - добавим часы, отсчитывающие время
  - будем отслеживать положение движущихся объектов
  - часы будут посылать сообщение ‘clock-tick движущимся объектам, чтобы те обновили своё состояние
- Добавим класс Torpedo

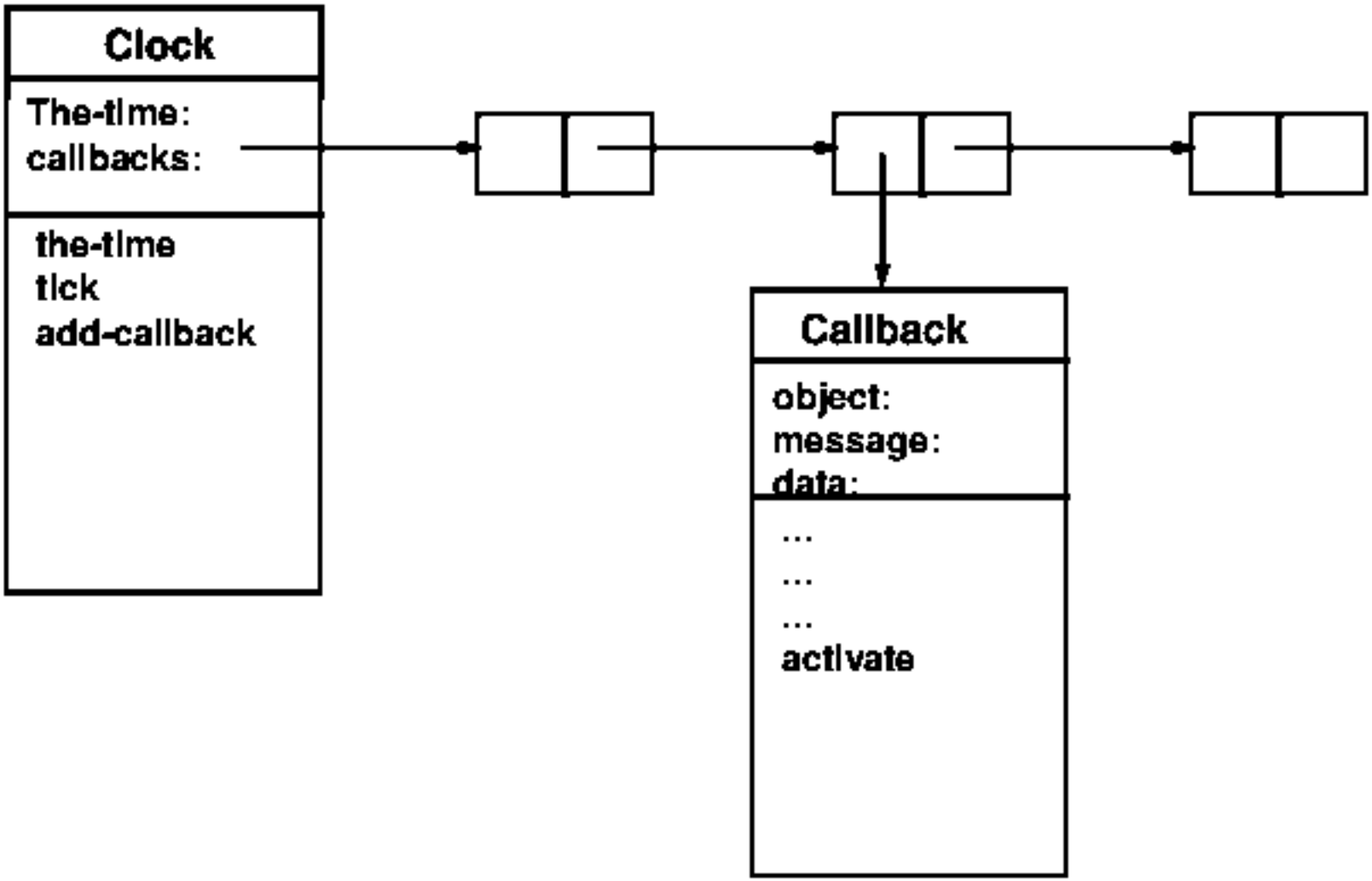
Изменения на диаграмме классов



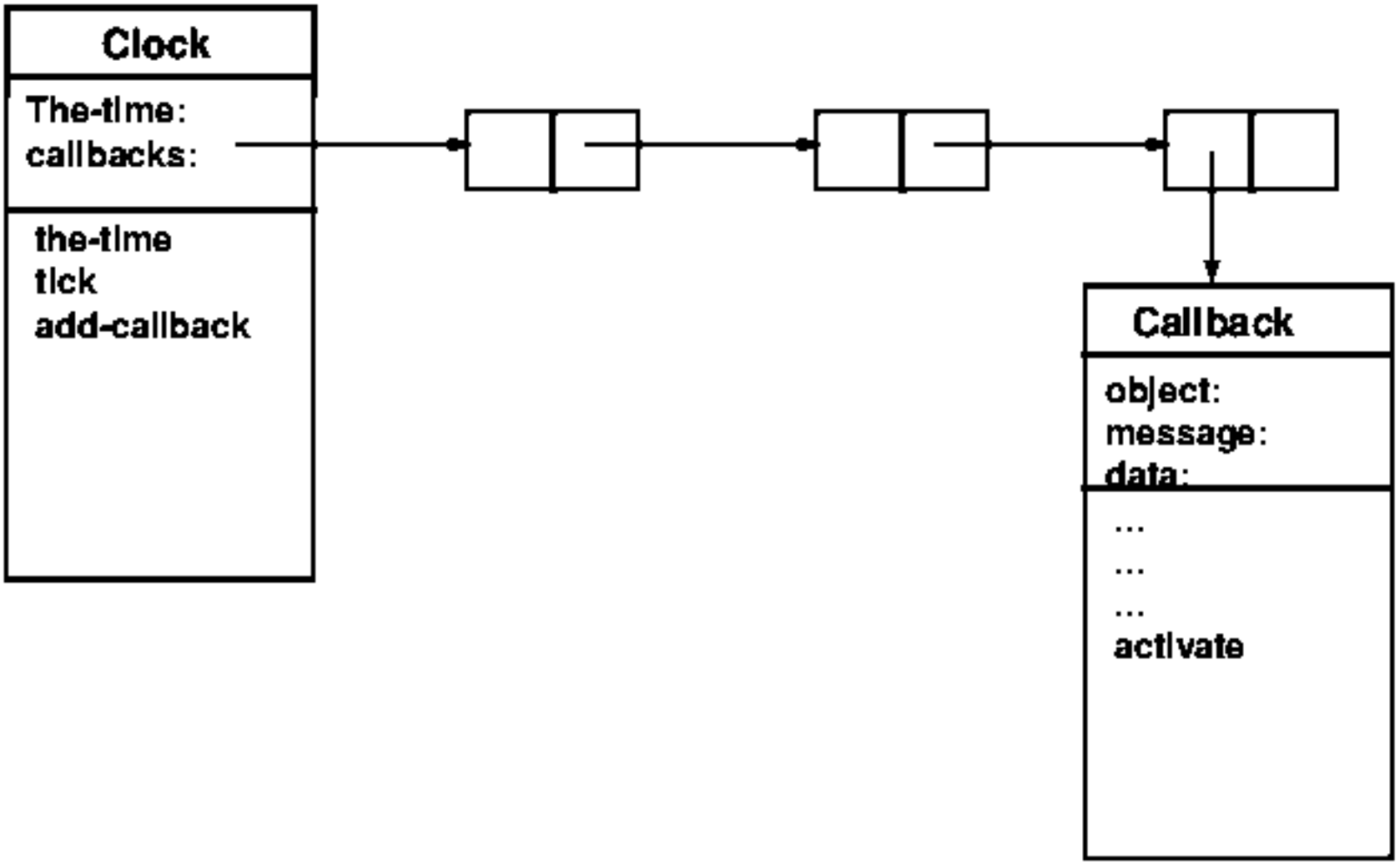
Часы управляют всем



Часы управляют всем



Часы управляют всем



Реализация часов

```
(define (make-clock . args)
  (let ((the-time 0)
        (callbacks '()))
    (lambda (msg)
      (cond
          ((eq? msg 'the-time) (lambda (self) the-time))
          ((eq? msg 'tick)  (lambda (self)
                      (map (lambda (x) (x 'activate)) callbacks)
                      (set! the-time (+ the-time 1))))
          ((eq? msg 'add-callback)
           (lambda (self cb)
            (set! callbacks (cons cb callbacks))
            'added)))
          (else (display "wrong message to clock"))))))
```

Реализация callback

```
(define (make-clock-callback object message . data)
  (lambda (msg)
    (cond
      ((eq? msg 'object)  (lambda (self) object))
      ((eq? msg 'message)  (lambda (self) message))
      ((eq? msg 'activate) (lambda (self)  (object message data)))
       (else (display "wrong message to clock-callbak")))))
```

Торпеда

```
(define (make-torpedo position velocity)
  (define (explode torp)
    (display “torpedo goes off!”)
    (remove-from-universe torp))
  (define (move)
    (set! position ...))
  (define (me msg . args)
    (cond
        ((eq? msg ‘position) position)
        ((eq? msg ‘velocity) velocity)
        ((eq? msg ‘move) (move))
        ((eq? msg ‘explode) (explode (car args)))
        ((eq? msg ‘display) (draw ...))
        (else (display “wrong message to torp”))))
  (clock ‘add-callback
     (make-clock-callback me   ‘move))
me)                          объект   сообщение
```

Промежуточные итоги

- Рассмотрели ОО-стиль программирования:
  - отличный от функционального
  - подходящий для симуляторов, сложных систем, ...
- Рассмотрели объектные модели
  - н/з от языка реализации

    класс – шаблон структуры и поведения объектов

    экземпляр – конкретный объект, созданный по шаблону

    диаграммы классов и диаграммы объектов

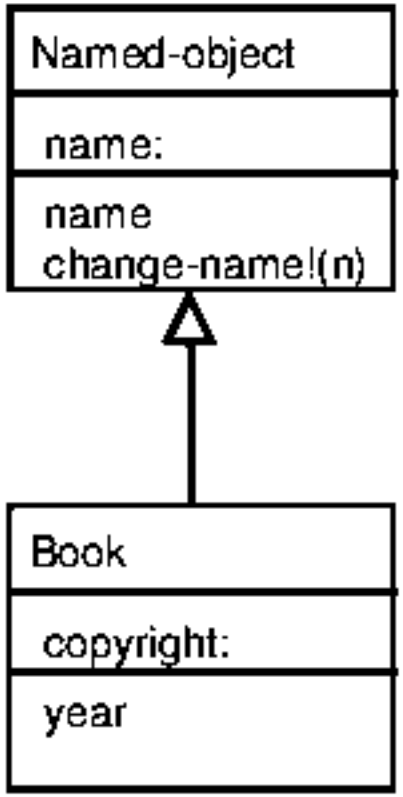
Диаграмма с обобщением (наследованием)

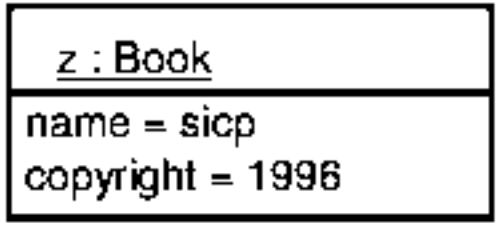
суперкласс

подкласс

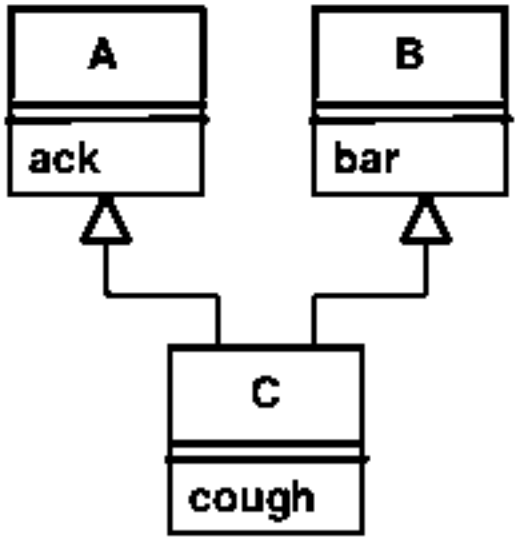
атрибут

операция





Множественное наследование

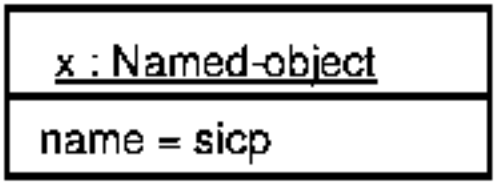


- Суперкласс и подкласс
  - A – суперкласс C
  - C – подкласс A и B
- Подкласс наследует атрибуты и операции своих суперклассов
  - У C три операции: ack, bar, cough

Класс именованных объектов

Диаграмма классов

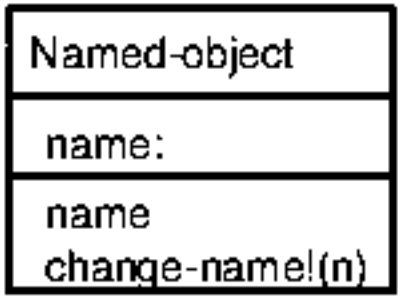
Диаграмма объектов

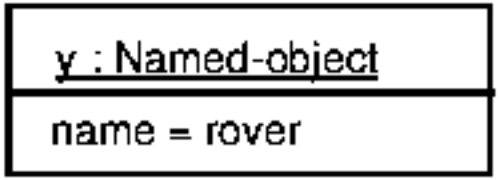


класс

атрибуты

операции





Учёт наследования

- Класс: описывается функцией, возвращающей конструктор
  - определяет общую структуру и поведение своих экземпляров
    - набор атрибутов
    - обработчик сообщений
    - определяет суперклассы, наследуемую структуру и поведение.
  - Корневой класс: Root
    - Корень иерархии наследования. Все классы – напрямую или косвенно его наследники.
  - Типизация:
    - Каждый класс должен реализовывать операцию type, возвращающую путь к Root в иерархии

Учёт наследования

- Экземпляр: порождается конструктором (например create-named-object)
  - все экземпляры различны в смысле eq?
  - новый формат отправки сообщений:

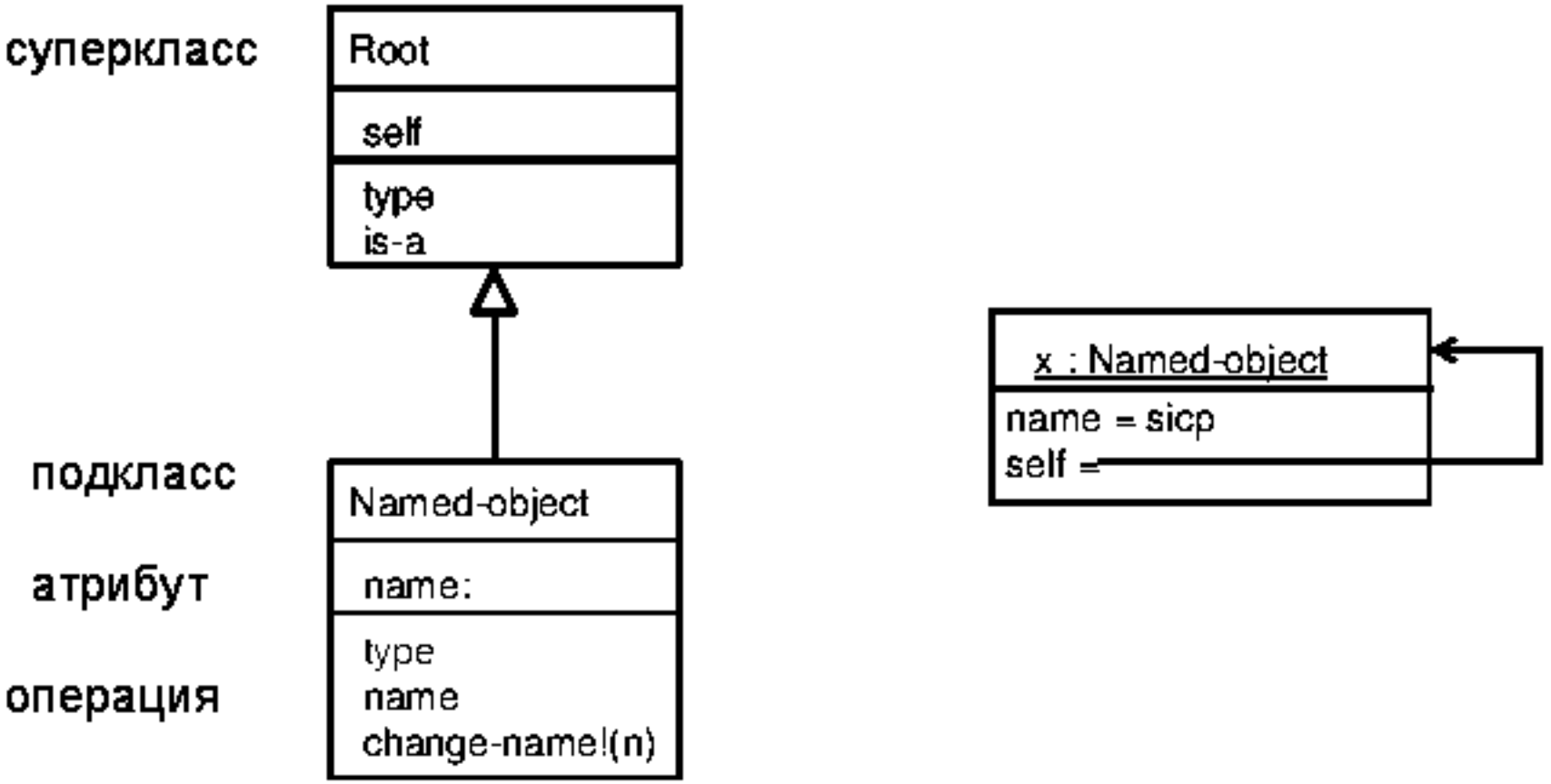
(ask <instance> '<message> <arg1> ... <argn>)

  - у всех экземпляров есть операции:

(ask <instance> 'type) ==> (<type> <supertype> ...)

(ask <instance> 'is-a <some-type>) ==> <boolean>

Обновлённая диаграмма



- **Named-object** наследует от **Root:**
  - атрибут self : у каждого объекта есть ссылка на себя
  - операцию **is-a**
  - переопределяет операцию **type**

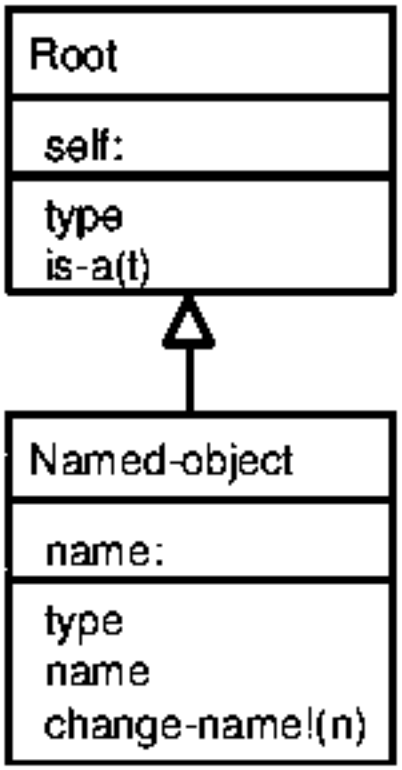
Обновлённая диаграмма

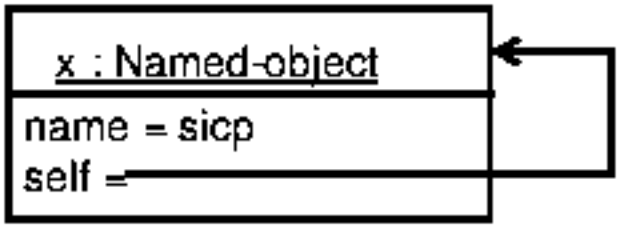
суперкласс

подкласс

атрибут

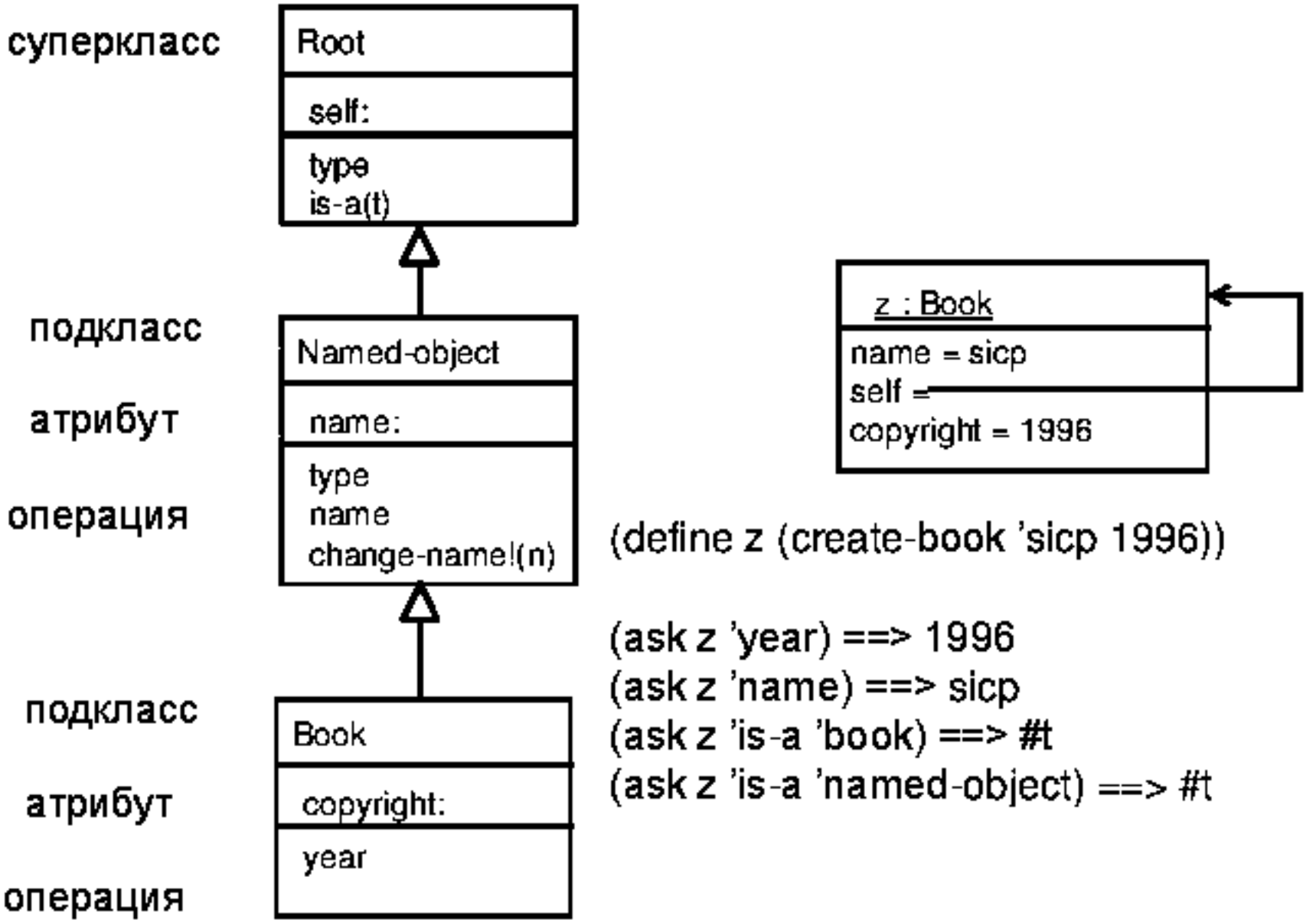
операция






```
(define x (create-named-object 'sicp))
(ask x 'name) => sicp
(ask x 'change-name! 'sicp-2nd-ed)
(ask x 'name) => sicp-2nd-ed)
(ask x 'type) => (named-object root)
(ask x 'is-a 'named-object) => #t
(ask x 'is-a 'clock) => #f
(ask x 'is-a 'root) => #t
```

Обновлённая диаграмма



Типовое описание класса

- Описание класса по имени <type> содержит
  - указание суперклассов
  - состояние (включающее self)
  - определяющий сообщения и
    - обязательно обрабатывающий type как указано ниже
    - имеющий ветвь (else (get-method …)) для сообщений суперкласса

```
(define (<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (<super1> self <args>)
        (<super2>-part (<super2> self <args>)
        <other superclasses>
        <other local state> )


        ((eq? msg 'type)
                                              )
        <другие сообщения и                >
              (get-method msg <super1>-part <super2>-part …))))))
```

Порождение экземпляров. Объекты Instance

- Для класса с именем <type> описываем функцию create-<type>
  - Она использует create-instance – функцию высшего порядка
    - порождающую объект Instance
    - добавляющую объекту Instance обработчик сообщений класса <type>
    - возвращающую ссылку на объект Instance

(define (create-<type> <arg1> <arg2> ... <argn>)
 (create-instance <type> <arg1> <arg2> ... <argn>))

- Объект создаётся вызовом create-<type>

(define <instance> (create-<type> <arg1> <arg2> ... <argn>))

Пример. Класс Book

конструктор для Book

```
(define (create-book name copyright)
  (create-instance book name copyright))
```

обработчик сообщений

атрибуты Book

```
(define (book self name copyright)
  (let ((named-object-part    (named-object  self  name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type) (lambda () (type-extend 'book 'named-object)))
        ((eq? msg 'year) (lambda () copyright))
        (else (get-method msg named-object-part))))))
```

суперкласс

создание обработчика суперкласса

новое сообщение

унаследованные методы

Ещё пример: класс Named-object

```
(define (create-named-object name)
  (create-instance named-object name))

(define (named-object self name)
 (let ((root-part (root self)))
  (lambda (msg)
   (cond
    ((eq? msg 'type)
        (lambda () (type-extend 'named-object root-part)))
    ((eq? msg 'name) (lambda () name))
    ((eq? msg 'change-name!) (lambda (newname) (set! name newname)))
    (else (get-method msg root-part))))))
```

Использование объектов Instance

- поиск метода: для <сообщения> в <instance>
-
- оба шага сразу: ask

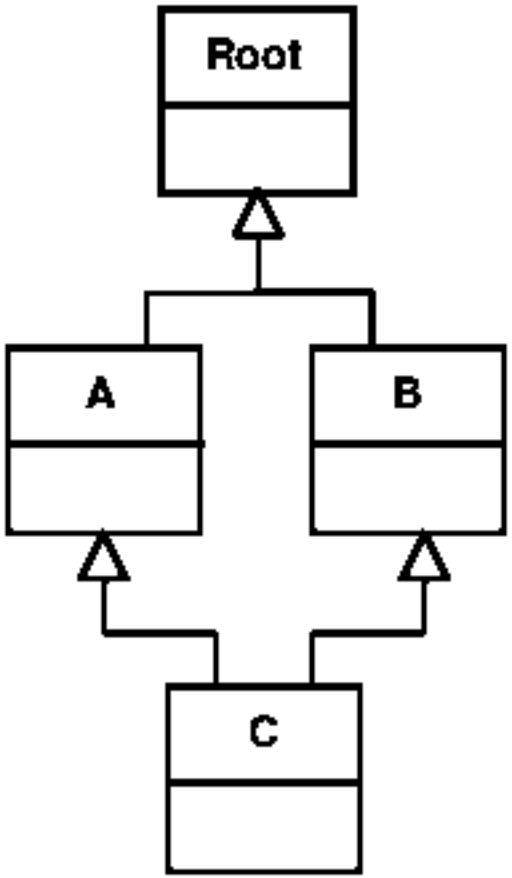
(define <instance> (create-<type> <arg1> <arg2> ... <argn>))

(define sm-method ('<сообщение> <instance>))

(ask <instance> '<сообщение> <m-arg1> ... <m-argm>)

Иерархия типов

- При наследовании у экземпляра несколько типов




```
(define a-instance (create-a))
(define c-instance (create-c))

(ask a-instance 'type) => (a root)
(ask c-instance 'type) => (c a b root)

(ask c-instance 'is-a 'c)=> #t
(ask c-instance 'is-a 'b)=> #t
(ask c-instance 'is-a 'a)=> #t
(ask c-instance 'is-a 'root)=> #t

(ask a-instance 'is-a 'c)=> #f
(ask a-instance 'is-a 'b)=> #f
(ask a-instance 'is-a 'a)=> #t
```

Точки зрения на ООП

- Точка зрения модели
  - диаграмма классов и диаграмма объектов
  - терминология: сообщения, операции, обобщение, суперкласс, подкласс, ...
- Точка зрения использования
  - Соглашения о том как писать ОО-код на Scheme:
    - описывать класс
      - указывать его суперклассы
    - порождать экземпляры
    - использовать экземпляры (вызывать операции)

Далее:

→ Точка зрения реализации

- Как реализованы экземпляры, классы, наследование

Вернёмся к примеру

суперкласс

подкласс

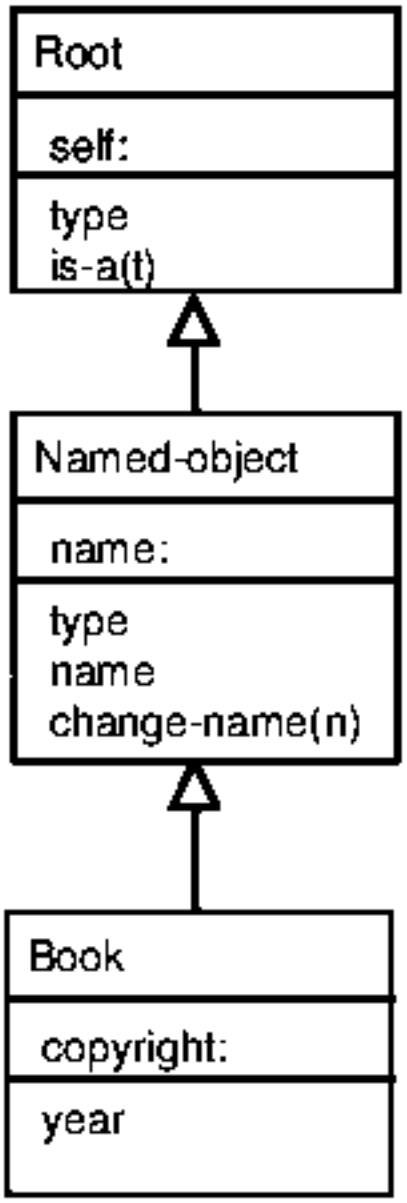
атрибут

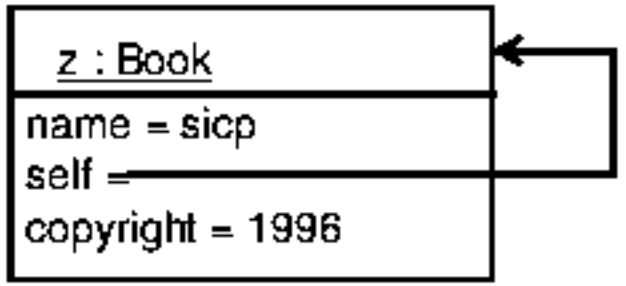
операция

подкласс

атрибут

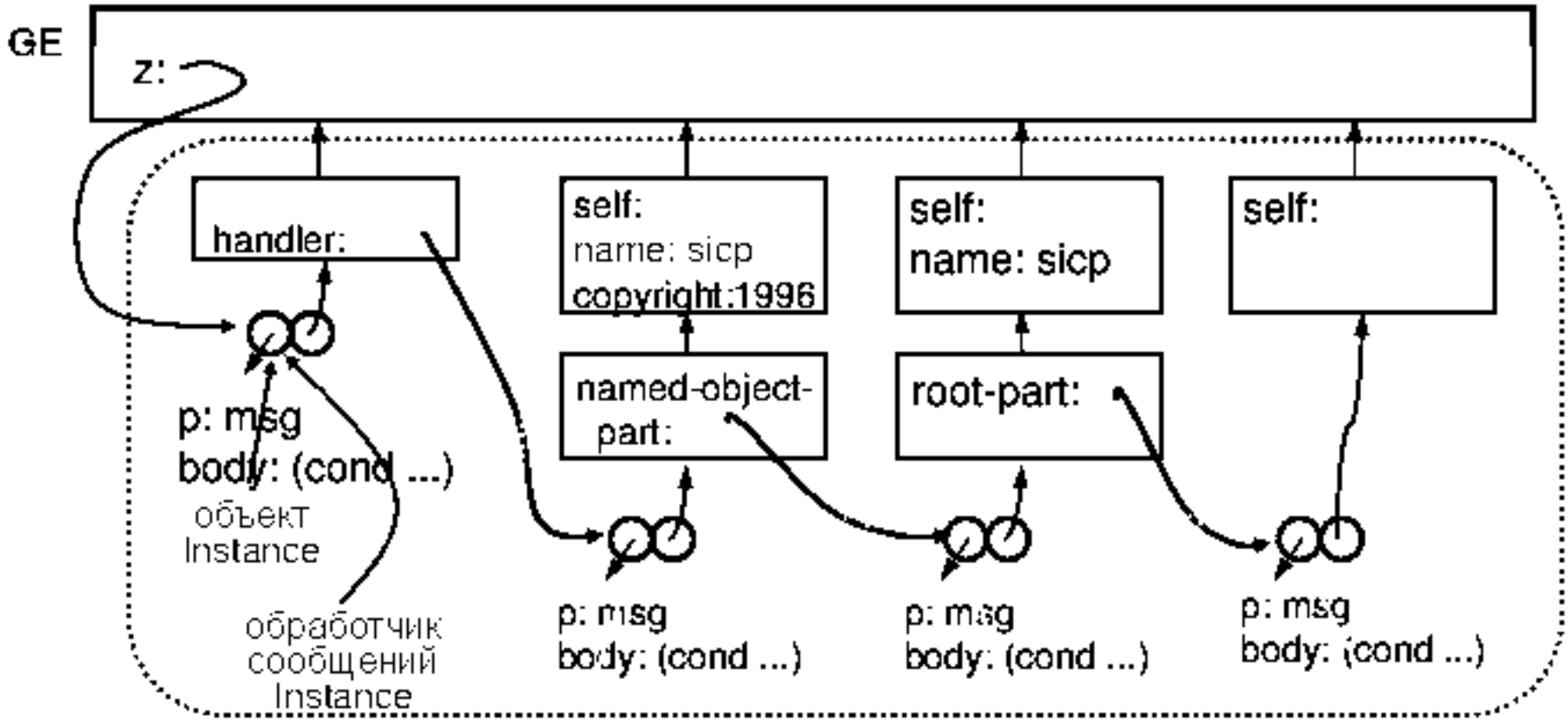
операция





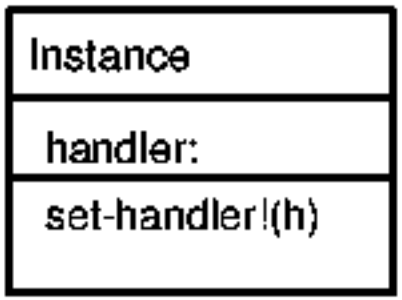
С точки зрения модели окружений

(define z (create-book 'sicp 1996))



Точка зрения реализации: Instance

```
(define (make-instance)
  (let ((handler #f))
    (lambda (msg)
      (cond
        ((eq? msg 'set-handler!)
         (lambda (handler-proc)
           (set! handler handler-proc)))
        (else (get-method msg handler))))))
```




```
(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (ask instance 'set-handler! handler)
    instance))
```

Точка зрения реализации: get-method и ask

- получить метод:

```
(define (get-method message object)
  (object message))
```

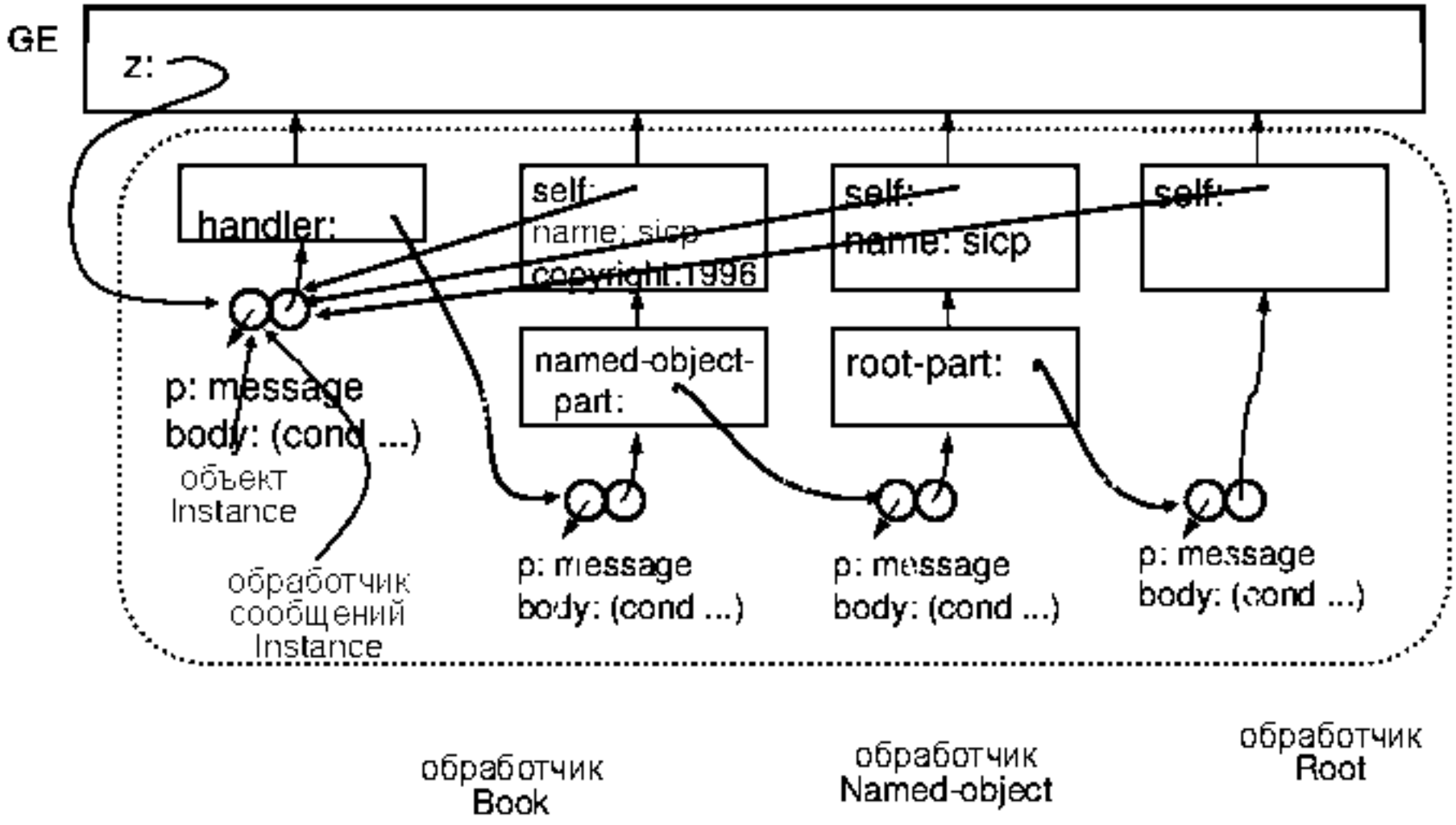
"ask" – комбинация поиска метода и его вызова

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (display “wrong method”))))
```

(apply op args) ==> (op arg1 arg2 … argn)

В модели окружений

(define z (create-book 'sicp 1996))



Точка зрения использования. Self

- В каждом классе есть атрибут self
  - self – это ссылка на весь экземпляр

- Зачем? Как и когда используется self ?
  - При реализации операции объект может послать сообщение своей части: например, внутри операции класса Book, можно

    (ask named-object-part 'change-name 'mit-sicp)
  - Иногда нужно послать сообщение экземпляру целиком: например

    (ask self 'year)
    - В ситуации, когда операция подкласса переопределяет операцию суперкласса нужна возможность указать, чью операцию мы хотим вызвать: объекта или унаследованной части объекта

- Рассмотрим на примере

Класс Person

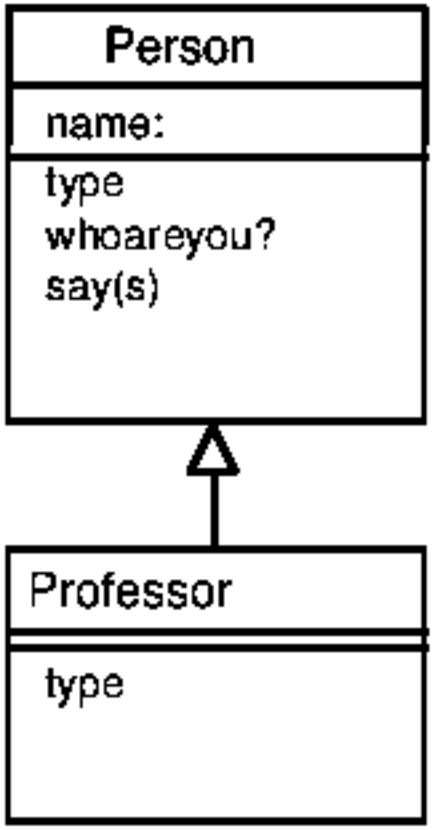
(define p1 (create-person 'joe))
(ask p1 'whoareyou?) ==> joe

(ask p1 'say '(the sky is blue)) ==> (the sky is blue)

Реализация Person

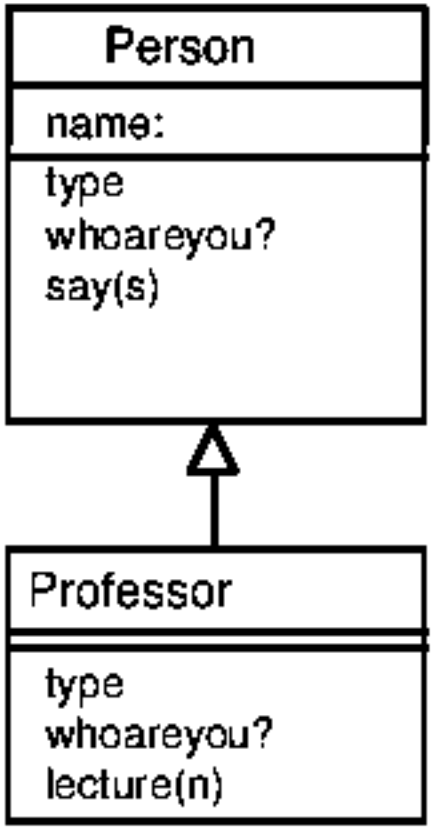

```
(define (create-person name)
  (create-instance person name))

(define (person self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
          ((eq? msg 'type)
                          (lambda () (type-extend 'person root-part)))
          ((eq? msg 'whoareyou?)    (lambda () name))
          ((eq? msg 'say)                (lambda (stuff) stuff))
          (else (get-method msg root-part))))))
```

Класс Professor




```
(define prof1 (create-professor 'fred))

(ask prof1 'say '(the sky is blue))
==> (the sky is blue)
```

Класс Professor



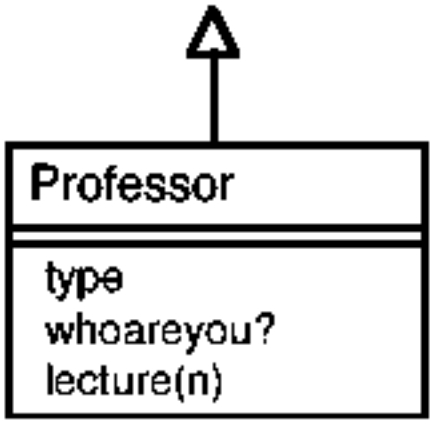
(define prof1 (create-professor 'fred))

(ask prof1 'whoareyou?)
==> (prof fred)
операция профессора 'whoareyou?
использует операцию персоны
'whoareyou?

(ask prof1 'lecture '(the sky is blue))
==> (therefore the sky is blue)

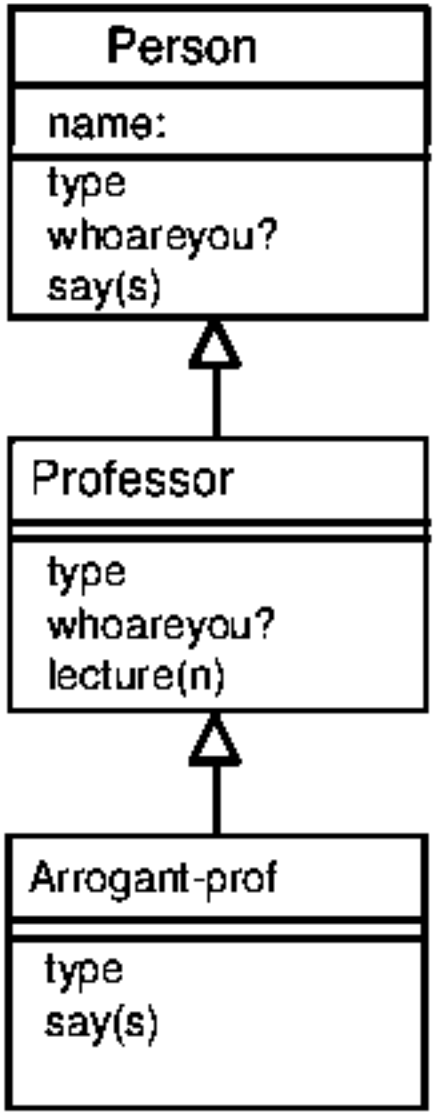
операция lecture использует операцию say

Реализация Professor



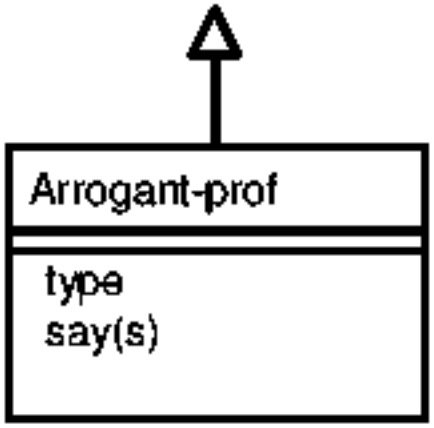

```
(define (create-professor name)
  (create-instance professor name))
(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type)
           (lambda () (type-extend 'professor person-part)))
        ((eq? msg 'whoareyou?)
           (lambda () (list 'prof (ask person-part 'whoareyou?))))
        ((eq? msg 'lecture)
           (lambda (notes)
             (cons 'therefore (ask person-part 'say notes))))
        (else (get-method msg person-part))))))
```

Класс Arrogant-prof



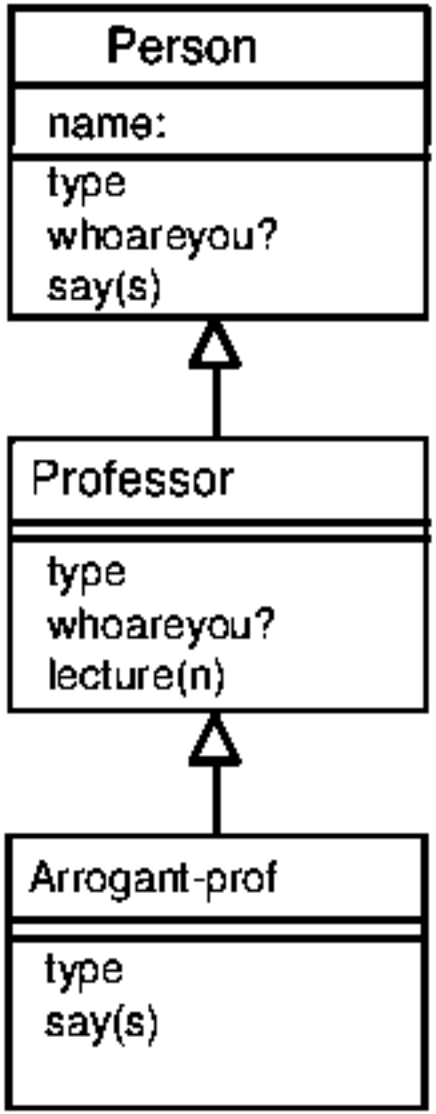

```
(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'whoareyou?)
==> (prof perfect)

(ask ap1 'say '(the sky is blue))
==> (the sky is blue obviously)
```

Реализация Arrogant-prof




```
(define (create-arrogant-prof name)
  (create-instance arrogant-prof name))

(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (msg)
      (cond
          ((eq? msg ‘type)
        (lambda () (type-extend 'arrogant-prof prof-part)))
          ((eq? msg ‘say) (lambda (stuff)
                    (append (ask prof-part 'say stuff)
                                (list 'obviously))))
        (else (get-method msg prof-part))))))
```

Чудаковатость Arrogant-prof

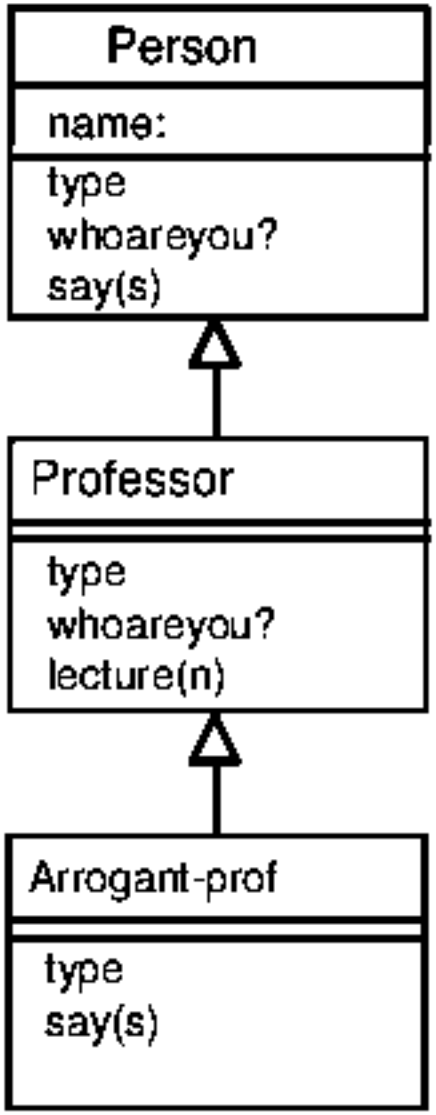


(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'lecture '(the sky is blue))
==> (therefore the sky is blue)

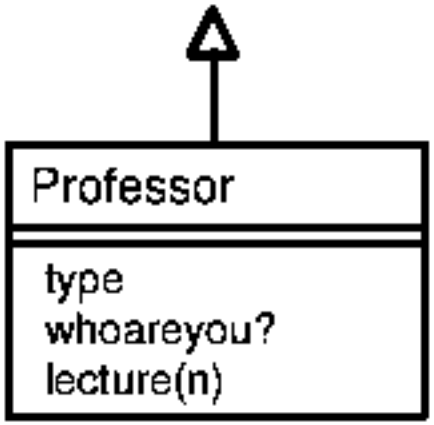
- Почему в конце фразы нет "obviously"?
  - lecture использует say класса Professor, а не say класса Arrogant-professor

Исправленный Arrogant-prof



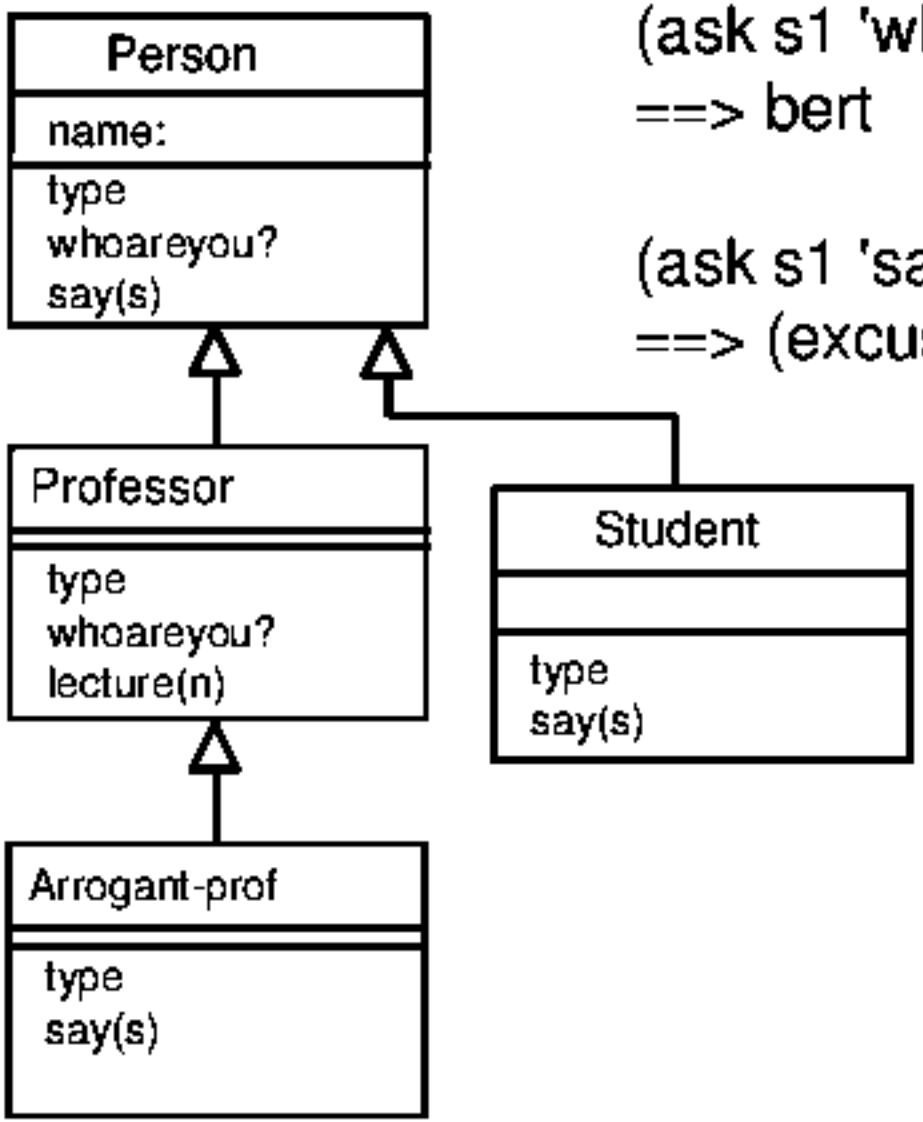

```
(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'lecture '(the sky is blue))
==> (therefore the sky is blue obviously)
```

Пересмотренная реализация Professor



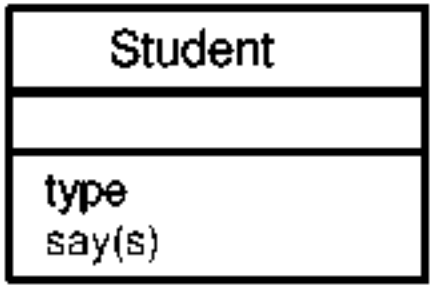

```
(define (create-professor name)
  (create-instance professor name))

(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
          ((eq? msg ‘type)
       (lambda () (type-extend 'professor person-part)))
          ((eq? msg ‘whoareyou?) (lambda () (list 'prof name)))
          ((eq? msg ‘lecture) (lambda (notes)
                    (cons 'therefore
                          (ask self 'say notes))))
        (else (get-method msg person-part))))))
```

Класс Student



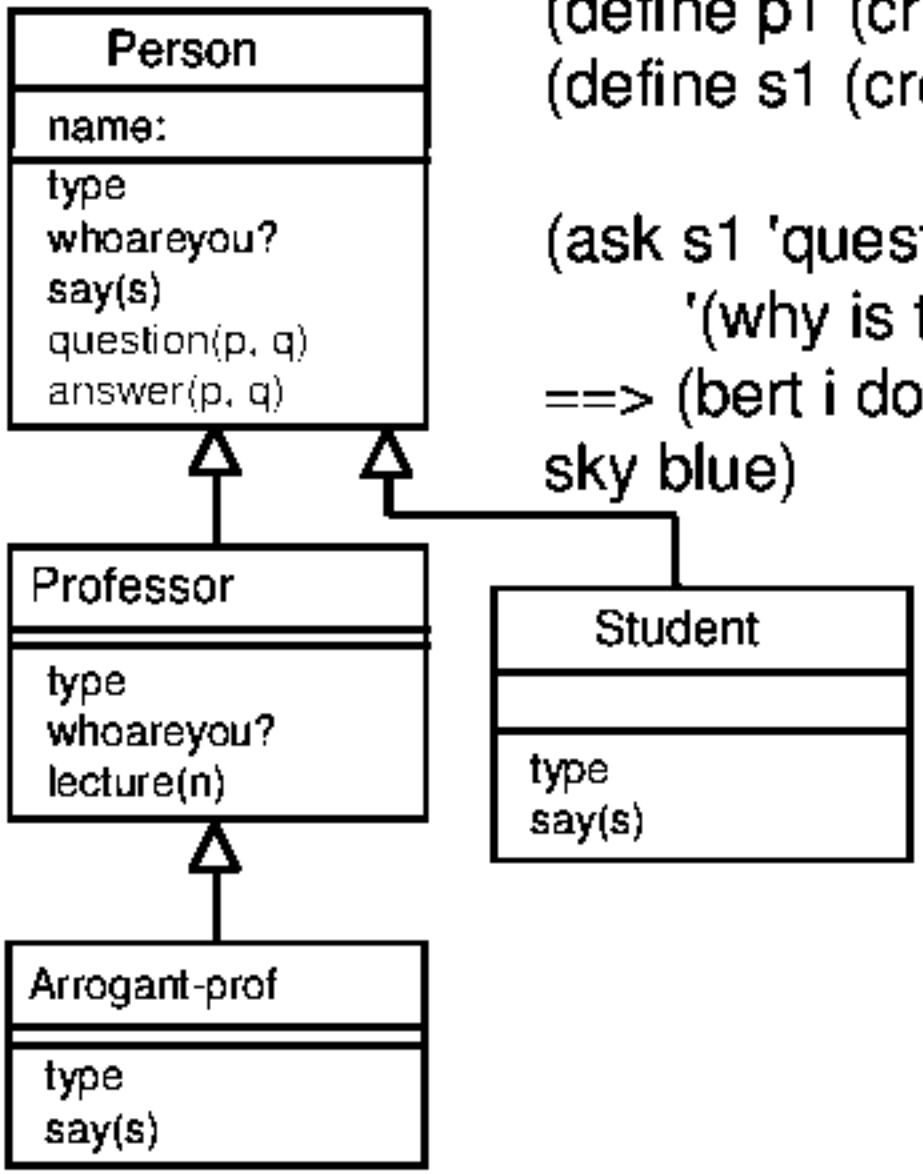

```
(define s1 (create-student 'bert))

(ask s1 'whoareyou?)
==> bert

(ask s1 'say '(i do not understand))
==> (excuse me but i do not understand)
```

Реализация Student




```
(define (create-student name)
  (create-instance student name))

(define (student self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
          ((eq? msg 'type)
        (lambda () (type-extend 'student person-part)))
           ((eq? msg 'say) (lambda (stuff)
                    (append '(excuse me but)
                            (ask person-part 'say stuff))))
          (else (get-method msg person-part))))))
```

Операции question и answer



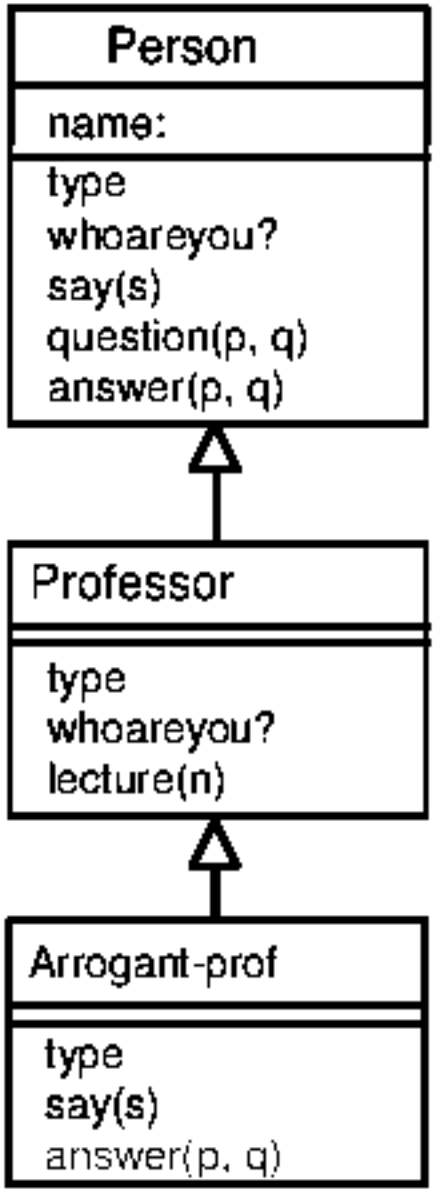

```
(define p1 (create-person 'joe))
(define s1 (create-student 'bert))

(ask s1 'question p1
        '(why is the sky blue))
==> (bert i do not know about why is the
sky blue)
```

Новая реализация Person

Person
name:
type whoareyou? say(s) question(p, q) answer(p, q)

```
(define (person self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
            ((eq? msg 'type)
            (lambda () (type-extend 'person root-part)))
            ((eq? msg 'whoareyou?)   (lambda () name))
            ((eq? msg 'say)          (lambda (stuff) stuff))
            ((eq? msg 'question)
            (lambda (of-whom query)
              (ask of-whom 'answer self query)))
            ((eq? msg 'answer)
            (lambda (whom query)
              (ask self 'say
                (cons (ask whom 'whoareyou?)
                  (append '(i do not know about)
                          query)))))
           (else (get-method msg root-part))))))
```

Arrogant-prof – специальный ответ




```
(define s1 (create-student 'bert))
(define prof1 (create-professor 'fred))
(define ap1 (create-arrogant-prof 'perfect))

(ask s1 'question ap1
        '(why is the sky blue))
==> (this should be obvious to you obviously)

(ask prof1 'question ap1
         '(why is the sky blue))
==> (but you wrote a paper about why
    is the sky blue obviously)
```

Новая реализация Arrogant-prof

```
(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type)
        (lambda () (type-extend 'arrogant-prof prof-part)))
        ((eq? msg 'say) (lambda (stuff)
                  (append (ask prof-part 'say stuff)
                          (list 'obviously))))
        ((eq? msg 'answer)
        (lambda (whom query)
          (cond ((ask whom 'is-a 'student)
                  (ask self 'say
                     '(this should be obvious to you)))
                 (
                  (ask self 'say
                     (append '(but you wrote a paper about)
                             query)))
                (else (ask prof-part 'answer whom query)))))
        (else (get-method msg prof-part))))))
```

Итоги

- Мы описываем иерархии классов
  - используя соглашение по структуре описания класса
    - для наследования структуры и поведения из суперкласса
- Управление поведением
  - "ask" к своей части
  - "ask" к self
- Дополнительное управление на основе типов объектов

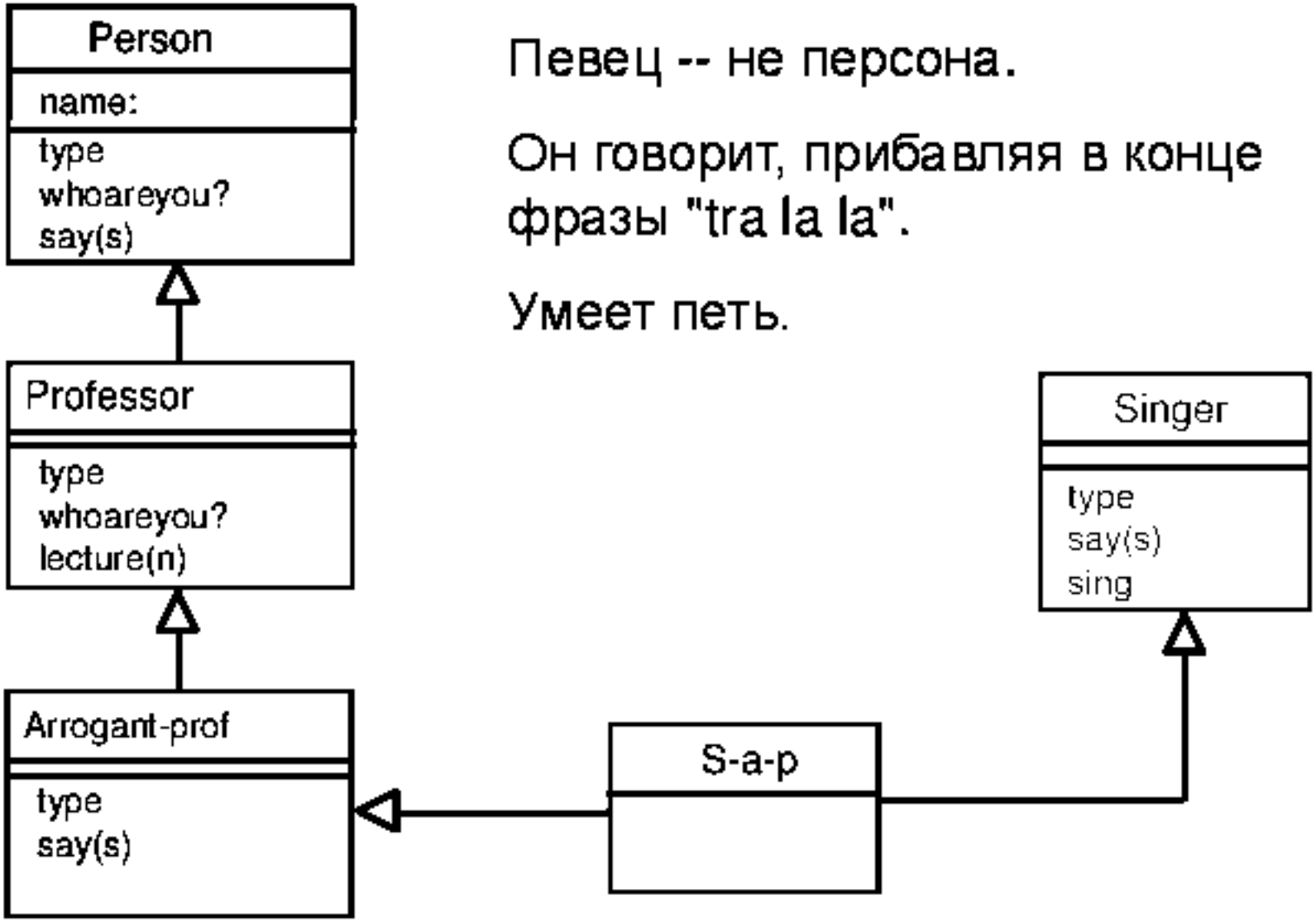
Дальше больше

- Просто объекты
  - соглашение о сообщениях и операциях
  - self

- Наследование (одиночное)
  - внутренние части от суперклассов
  - в операции можно обратиться к внутренней части
  - get-method для суперкласса находит нужный метод

- **Множественное наследование**

Singer и Singing-arrogant-prof



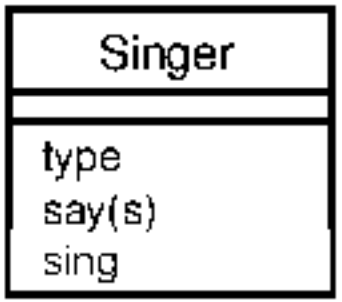
Певец -- не персона.

Он говорит, прибавляя в конце фразы "tra la la".

Умеет петь.

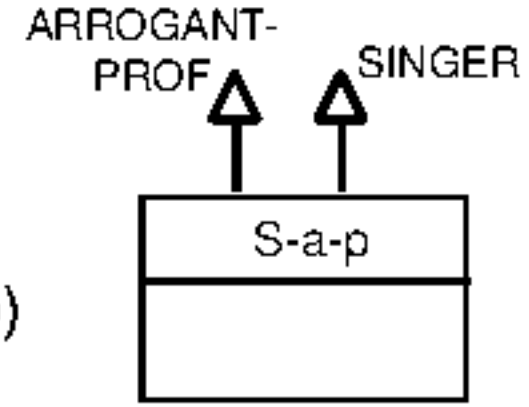
Реализация Singer

```
(define (create-singer)
  (create-instance singer))

(define (singer self)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
            ((eq? msg 'type)
             (lambda () (type-extend 'singer root-part)))
             ((eq? msg 'say)
             (lambda (stuff) (append stuff '(tra la la))))
             ((eq? msg 'sing)
             (lambda () (ask self 'say '(the hills are alive))))
            (else (get-method msg root-part))))))
```



Singer – базовый класс (у него один суперкласс -- Root)

Реализация Singing-arrogant-prof




```
(define (create-singing-arrogant-prof name)
  (create-instance singing-arrogant-prof name))

(define (singing-arrogant-prof self name)
  (let ((singer-part (singer self))
        (arr-prof-part (arrogant-prof self name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type)
         (lambda () (type-extend 'singing-arrogant-prof
                                 singer-part
                                 arr-prof-part)))
        (else (get-method msg singer-part arr-prof-part))))))
```

Пример с Singing-arrogant-professor

```
(define sap1 (create-singing-arrogant-prof  'zoe))
(ask sap1 'whoareyou?)
==> (prof zoe)

(ask sap1 'sing)
==> (the hills are alive tra la la)

(ask sap1 'say '(the sky is blue))
==> (the sky is blue tra la la)

(ask sap1 'lecture '(the sky is blue))
==> (therefore the sky is blue tra la la)
```

- операция say класса arrogant-prof не работает в sap1 (нет в конце фраз)
  - get-method находит say в классе singer, указанном первым
- если нам нужен точный контроль,
  - то следует реализовать say в классе Singing-arrogant-professor

Точка зрения реализации: Множественное наследование

- Как реализовать новый get-method?
  - просматривать экземпляры по порядку, пока не будет найден тот, который может обработать сообщение.

```
(define (get-method message object)
  (object message))
```

превращается в

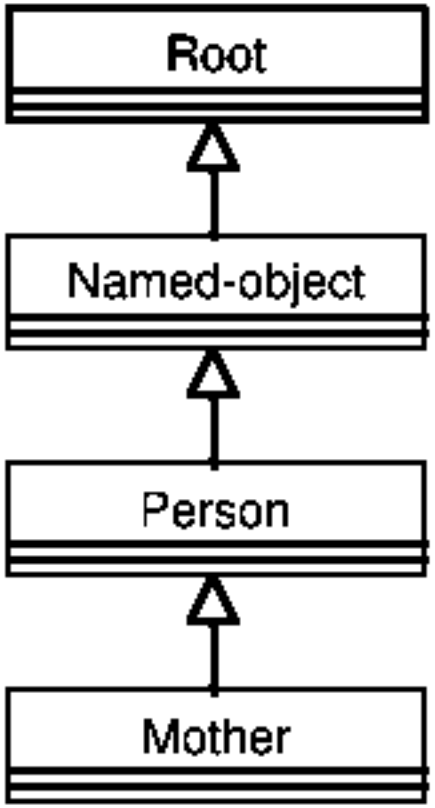
```
(define (get-method message . objects)
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method ((car objects) message)))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
  (try objects))
```

Ещё пример

- Цель примера: продемонстрировать разницу между
  - “is-a” (связью обобщения)
  - “has-a” (ассоциацией)

- Добавим родственные связи к person!

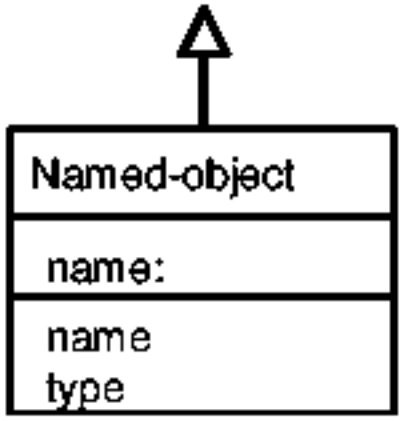
Родственные связи



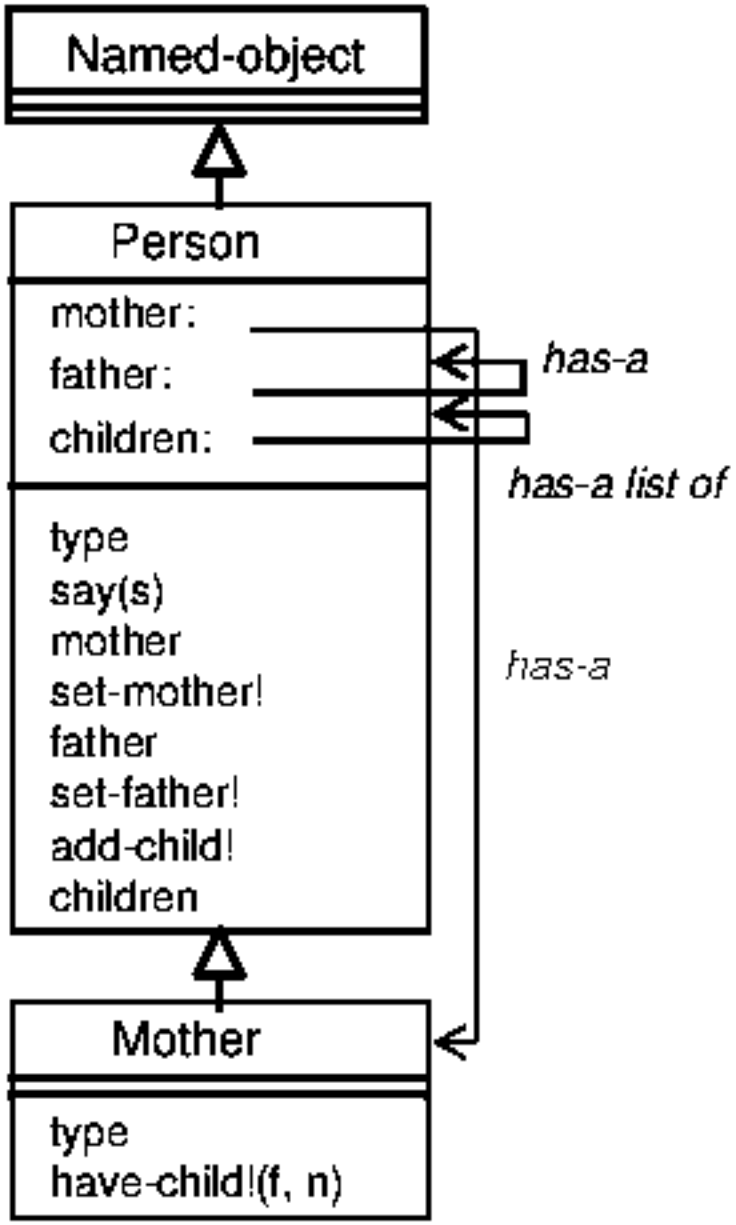
- Добавим класс **Mother** и рассмотрим, что получится с точки зрения
  - диаграммы классов
  - поведения
  - диаграммы объектов
  - описания классов
  - модели вычислений с окружениями

Описание Named-object

```
(define (create-named-object name)
  (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
        ((eq? msg 'type)
         (lambda () (type-extend 'named-object root-part)))
        ((eq? msg 'name) (lambda () name))
        (else (get-method msg root-part))))))

(define (names-of objects)
    (map (lambda (x) (ask x 'name)) objects))
```

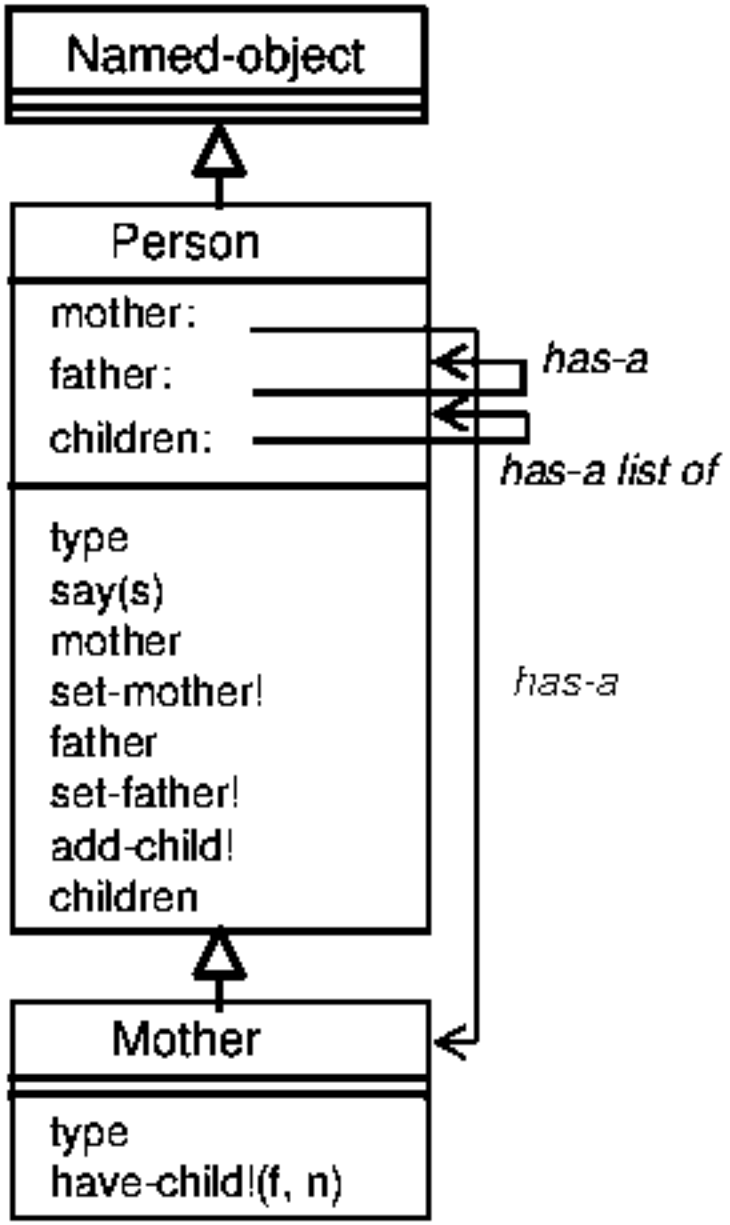


Класс Person



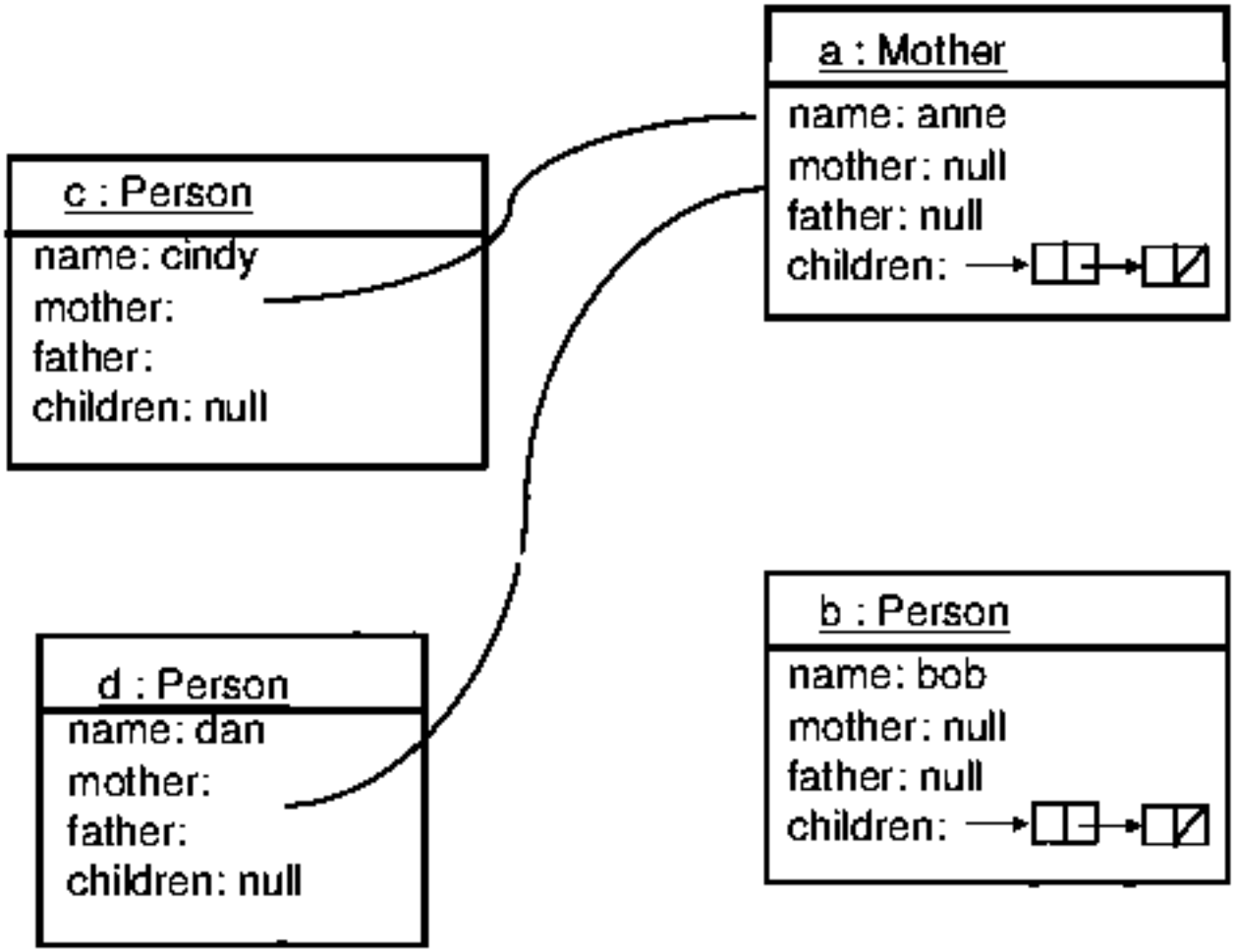
- Person подкласс Named-object
- состояние: теперь у персоны…
  - *есть* мать (экземпляр Mother)
  - *есть* отец (экземпляр Person)
  - *есть* список детей (экземпляров person)
- поведение: добавлены операции для управления состоянием
- Mother подкласс Person
  - добавлена операция have-child!(father, name)

Поведение



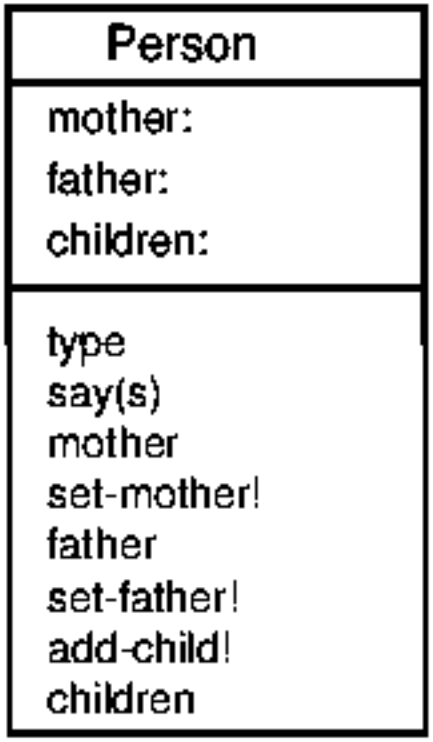

```
(define a (create-mother 'anne))
(define b (create-person 'bob))
(ask a 'name)  ==> anne
(ask b 'name) ==> bob
(ask a 'type) ==>
        (mother person named-object root)

(ask b 'type) ==>
        (person named-object root)

(define c (ask a 'have-child! b 'cindy))
(define d (ask a 'have-child! b 'dan))
(names-of (ask a 'children)) ==>
        (dan cindy)

(names-of (ask b 'children)) ==>
        (dan cindy)

(ask d 'name) ==> dan
(ask (ask d 'mother) 'name) ==> anne
```

Диаграмма объектов

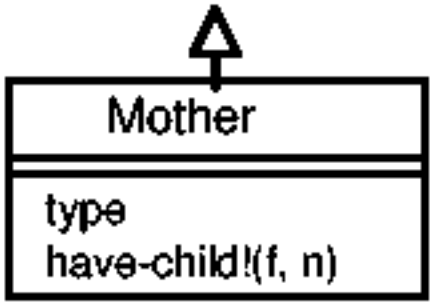


Описание класса Person

```
(define (create-person name)
  (create-instance person name))

(define (person self name)
  (let ((named-part (named-object self name))
        (mother null)
        (father null)
        (children null))
    (lambda (msg)
      (cond
        ((eq? msg 'type) (lambda () (type-extend 'person named-part)))
        ((eq? msg 'say) (lambda (stuff) (display stuff)))
        ((eq? msg 'mother) (lambda () mother))
        ((eq? msg 'father) (lambda () father))
        ((eq? msg 'children) (lambda () children))
        ((eq? msg 'set-mother!) (lambda (mom) (set! mother mom)))
        ((eq? msg 'set-father!) (lambda (dad) (set! father dad)))
        ((eq? msg 'add-child!)   (lambda (child)
                                   (set! children (cons child children))
                                   child))
        (else (get-method msg named-part))))))
```



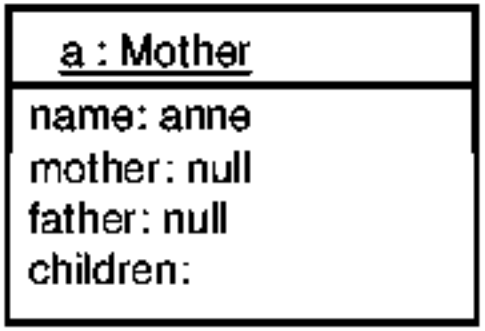
Описание класса Mother

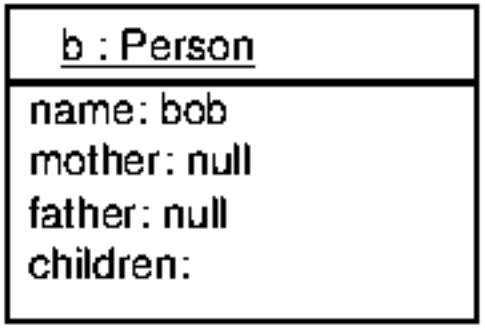



```
(define (create-mother name)
  (create-instance mother name))

(define (mother self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type) (lambda () (type-extend 'mother person-part)))
        ((eq? msg 'have-child!)
         (lambda (dad child-name)
           (let ((child (create-person child-name)))
             (ask child 'set-mother! self)
             (ask child 'set-father! dad)
             (ask self 'add-child! child)
             (ask dad 'add-child! child))))
        (else (get-method msg person-part))))))
```

Добавление объекта-ребёнка

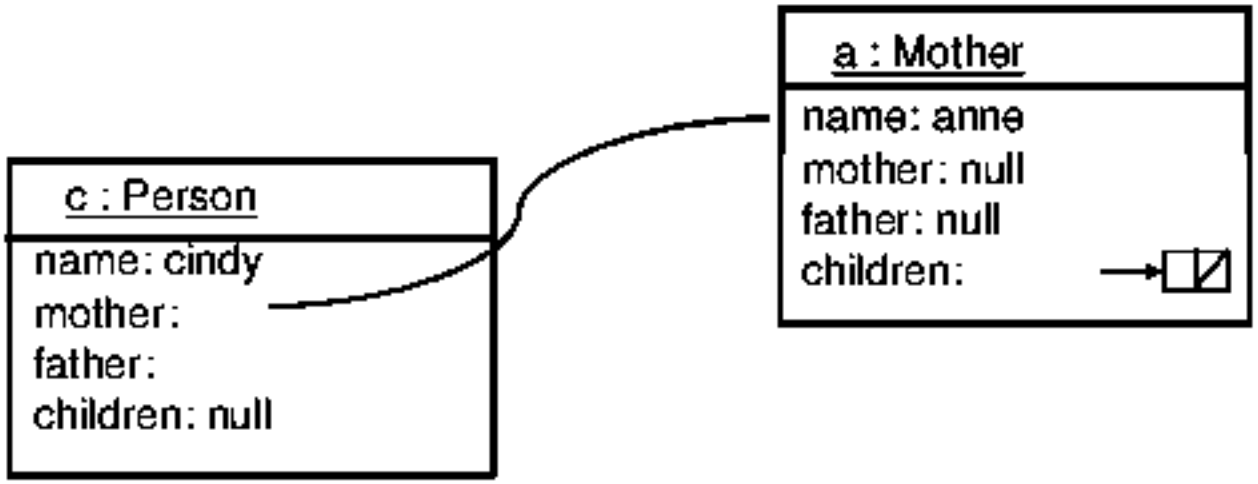
```
(define a (create-mother 'anne))
(define b (create-person 'bob))
```

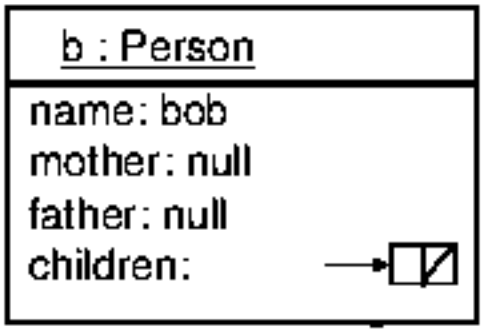





```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
    (ask child 'set-father! dad)
    (ask self 'add-child child)
    (ask dad 'add-child child)))
```

Добавление объекта-ребёнка



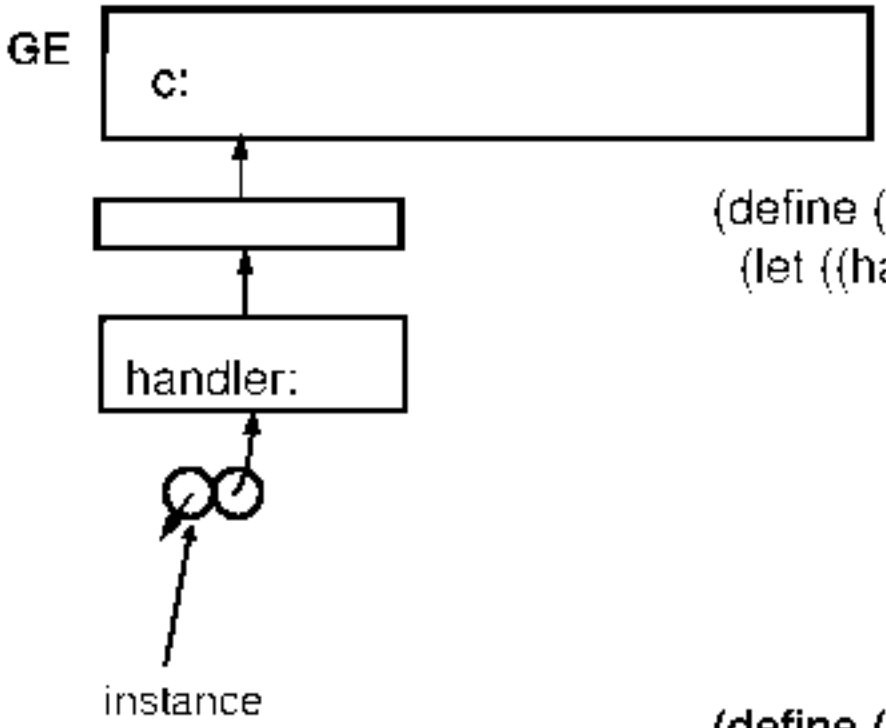



```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
```

))

В модели окружений

(create-person 'cindy) =>
(create-instance person 'cindy)



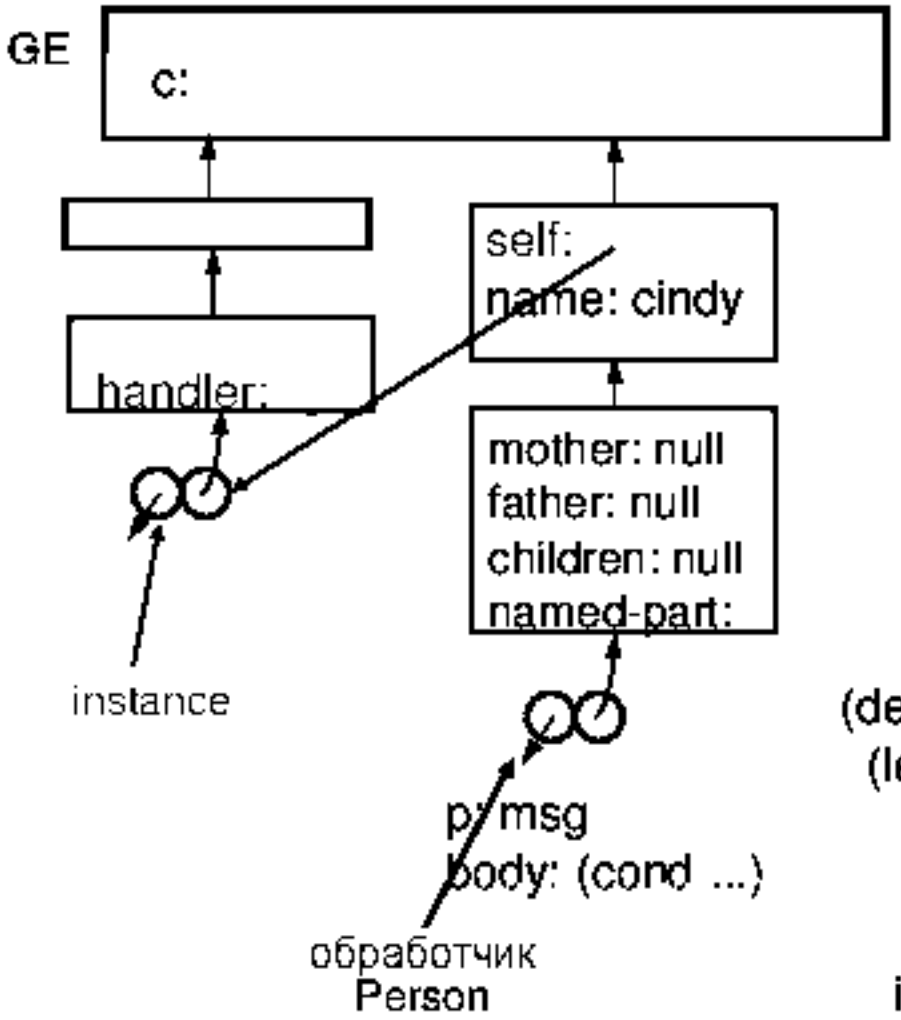

```
(define (make-instance)
  (let ((handler #f))
```

```
))
```

```
(define (create-instance maker . args)
  (let* ((instance (make-instance) )
         (handler (apply maker instance args) ))
    (ask instance 'set-handler! handler)
    instance))
```

В модели окружений

(create-person 'cindy) =>
(create-instance make-person 'cindy)



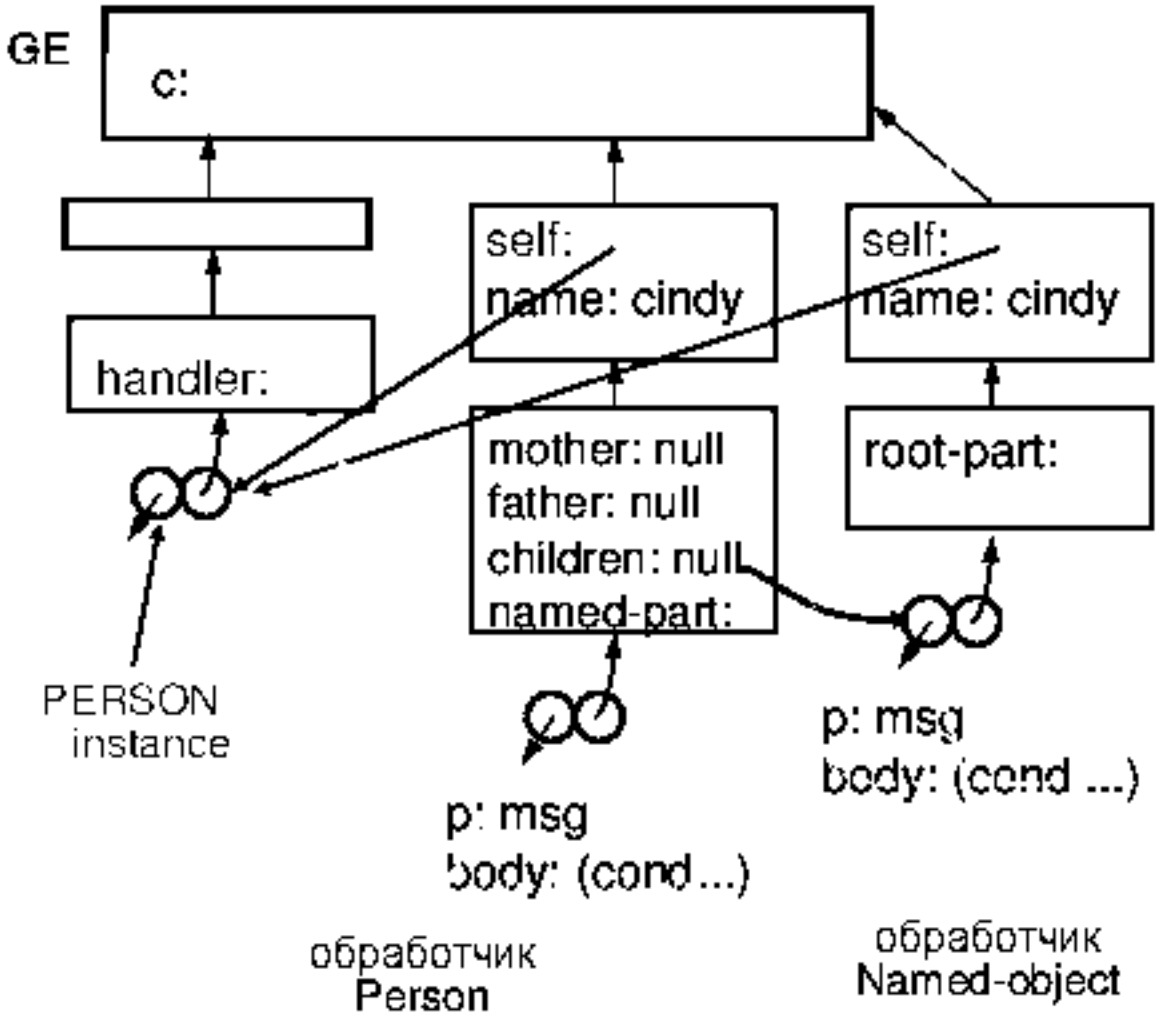

```
(define (person self name)
  (let ((named-part (make-named-object
self name))
        (mother nil)
        (father nil)
        (children nil))
    (lambda (message)
     (case message ...))))
```

```
(define (create-instance maker . args)
 (let* ((instance (make-instance) )
        (handler (apply maker
                        instance args) ))

                                  instance))
```

В модели окружений

(create-person ‘cindy) =>
(create-instance person ‘cindy)

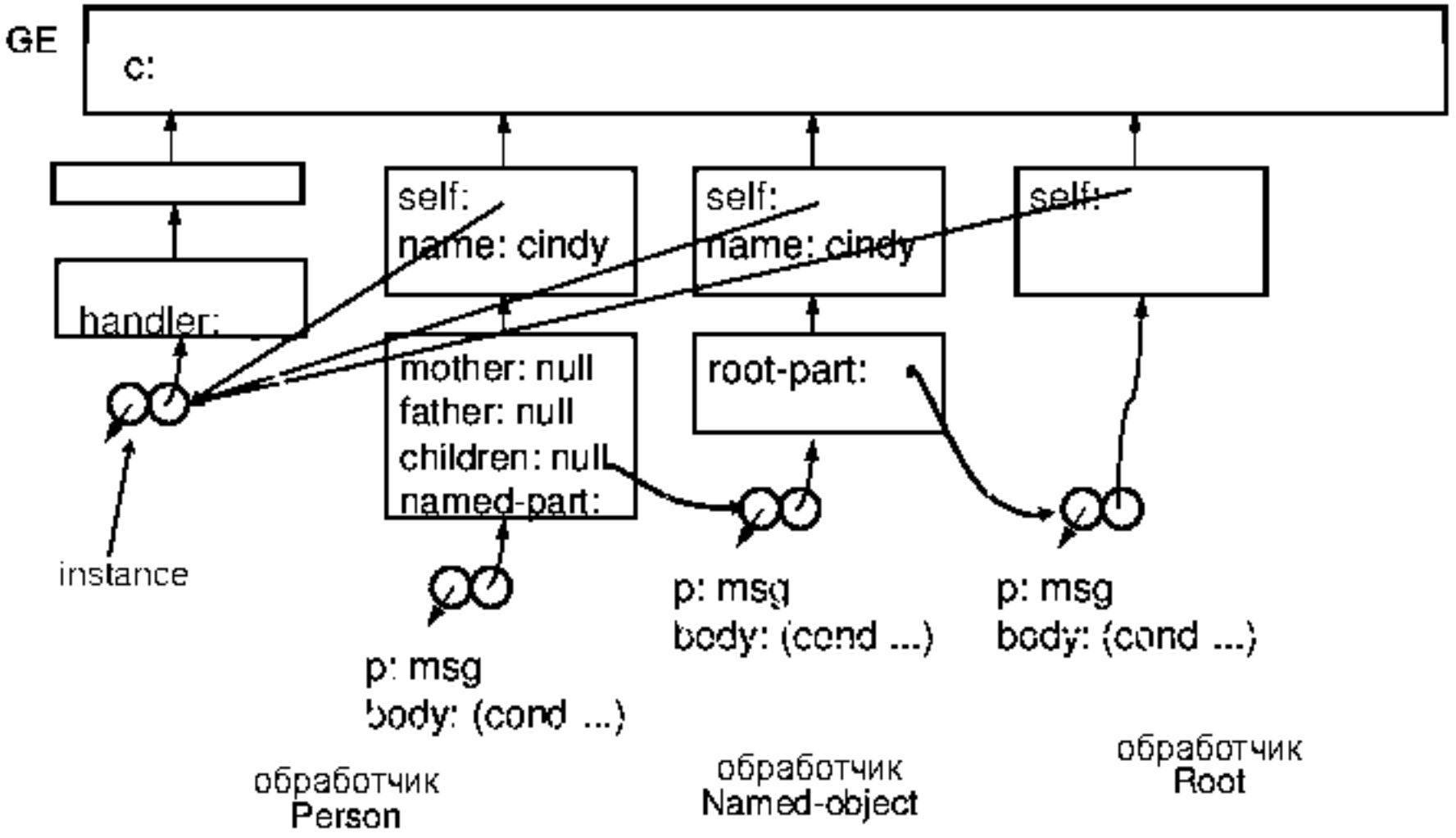
```
(define (named-object self name)
  (let ((root-part (root self)))
    (lambda (msg)
      ( cond...))))
```



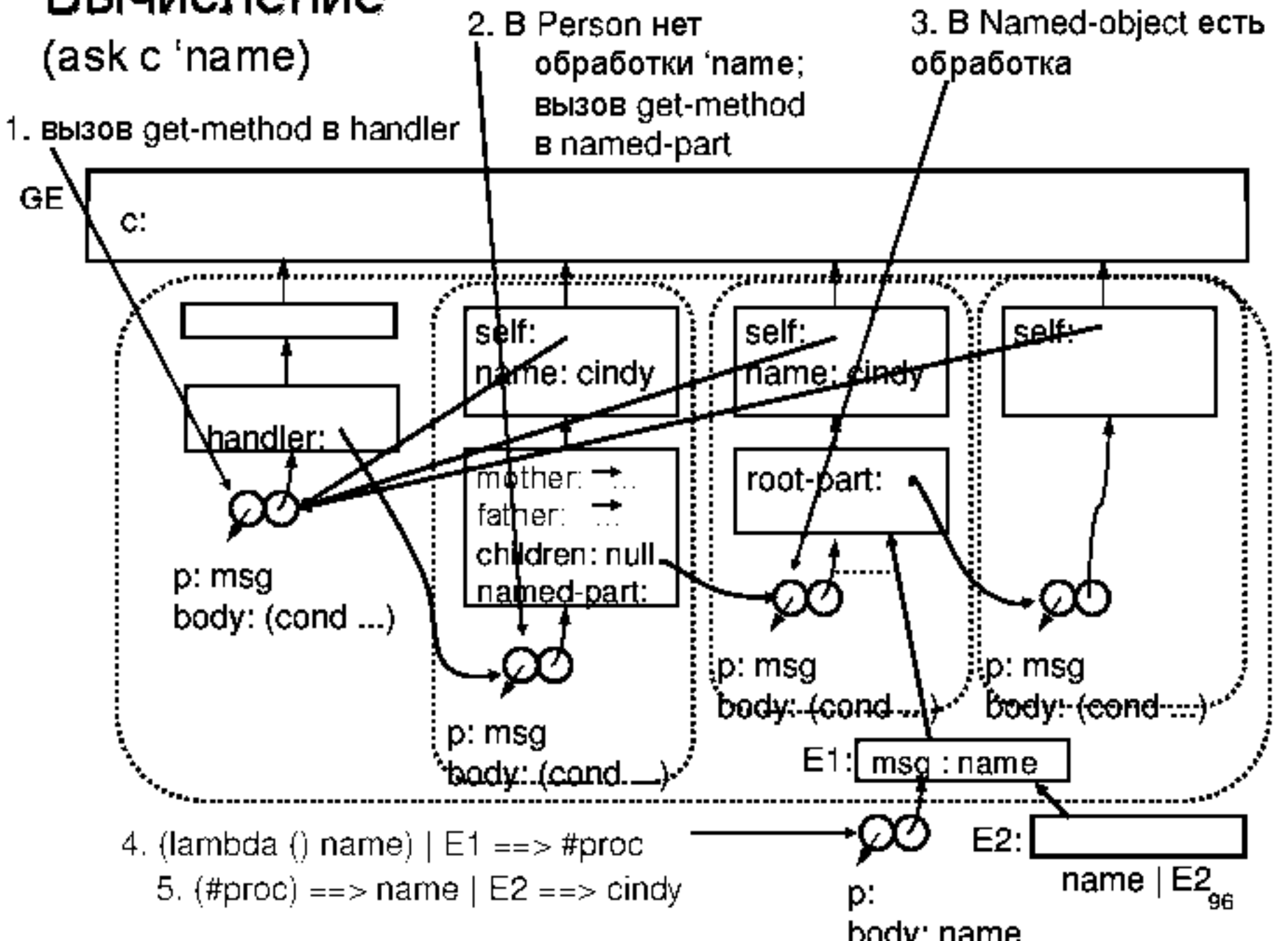
В модели окружений

(create-person ‘cindy) =>
(create-instance person ‘cindy)

```
(define (root self)
  (lambda (msg)
    ( cond...))))
```



Вычисление
(ask c 'name)



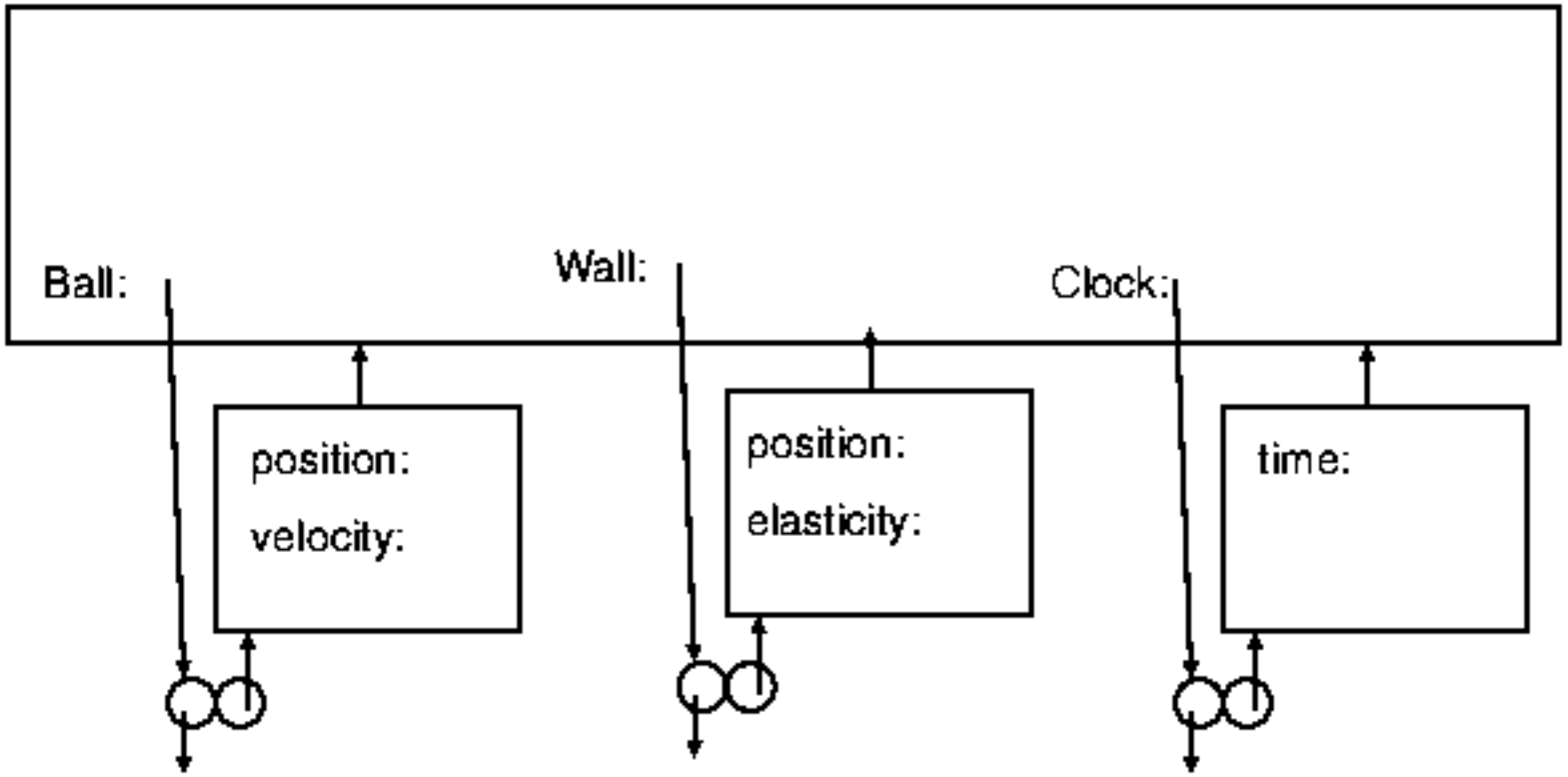
Итог лекции 7

- Классы: описывают общую структуру и поведение
- Экземпляры содержат:
  - «заглавное звено» (self)
  - цепочку обработчиков
- Иерархия классов
  - Наследование структур и поведения суперклассов
  - Множественное наследование: правила поиска операций
- ООП
  - **Модель:** диаграммы классов и объектов
  - **Использование:** способы описания классов и создания экземпляров
  - **Реализация:** отображение понятий ООП в Scheme

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

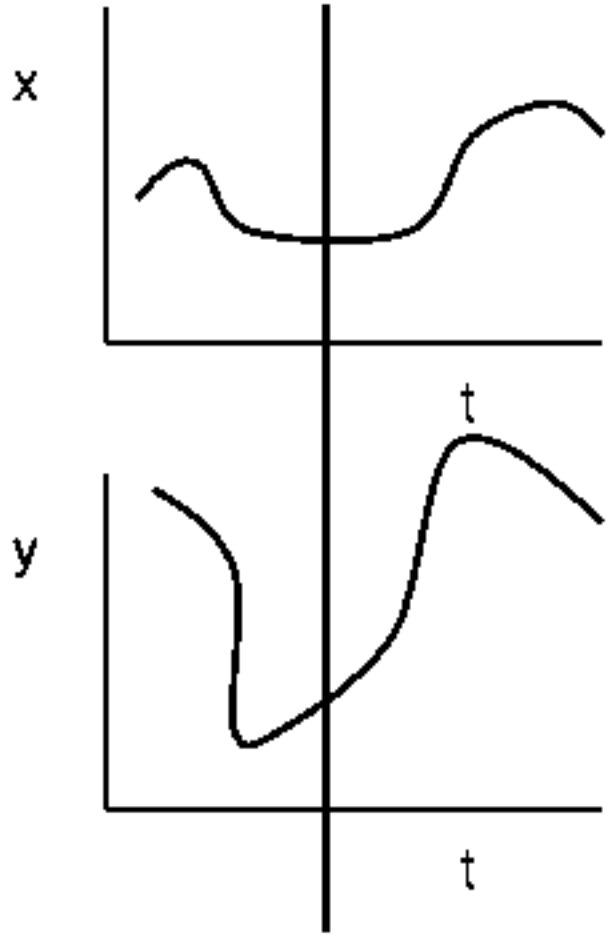
Clock:
1
2
3
4
5
…

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

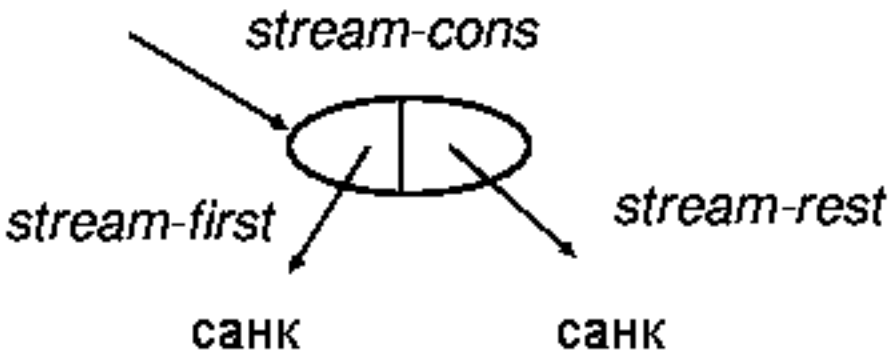
(stream-rest str) -- возвращает хвост потока

- 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)

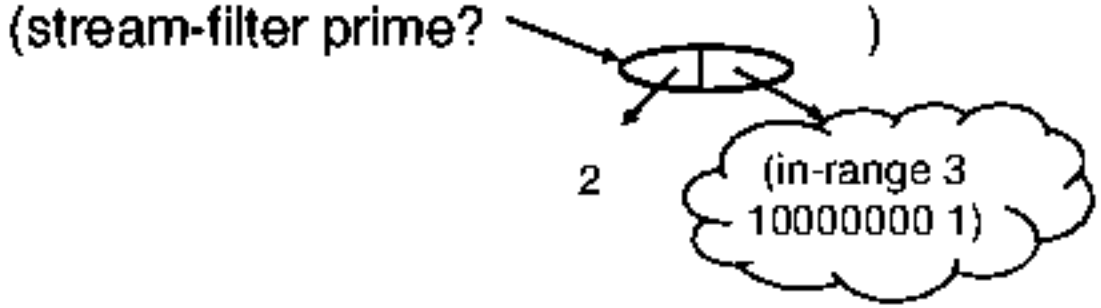
```
(stream-ref
  (stream-filter (lambda (x) (prime? x))
                 (in-range 2 100000000 1))
  100)
```

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))))
```

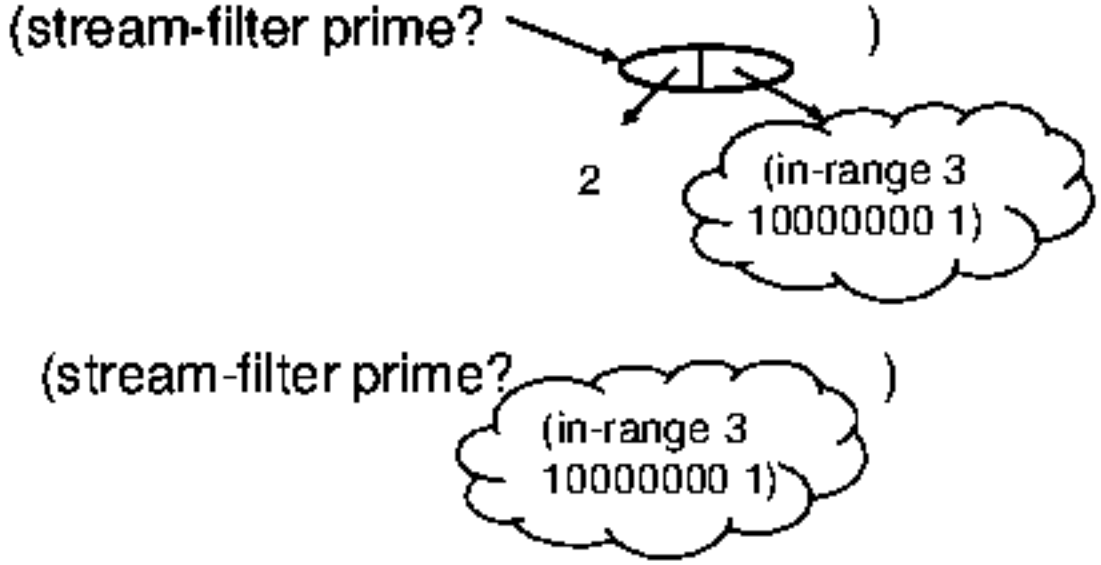
Вычисления в примере

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



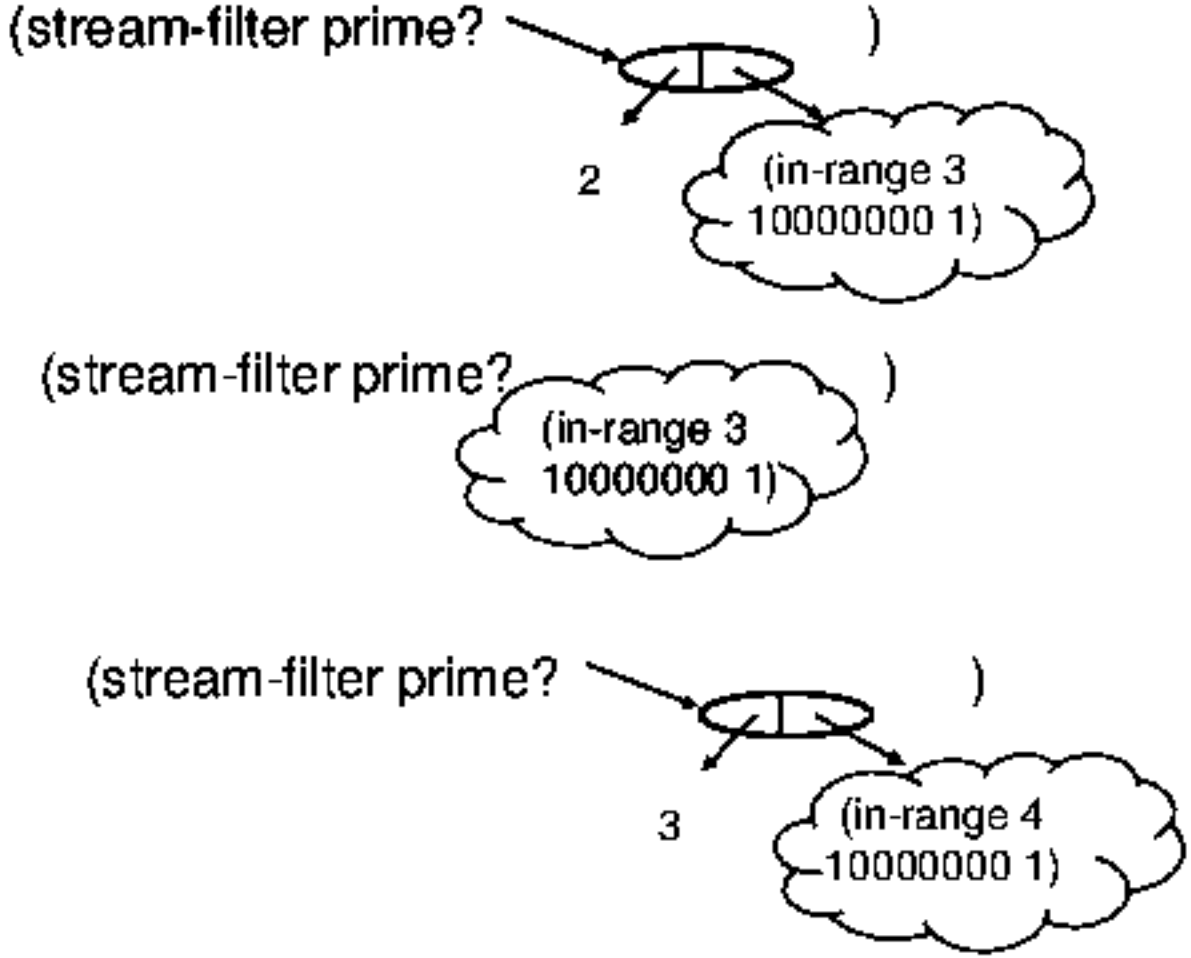
Вычисления в примере

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



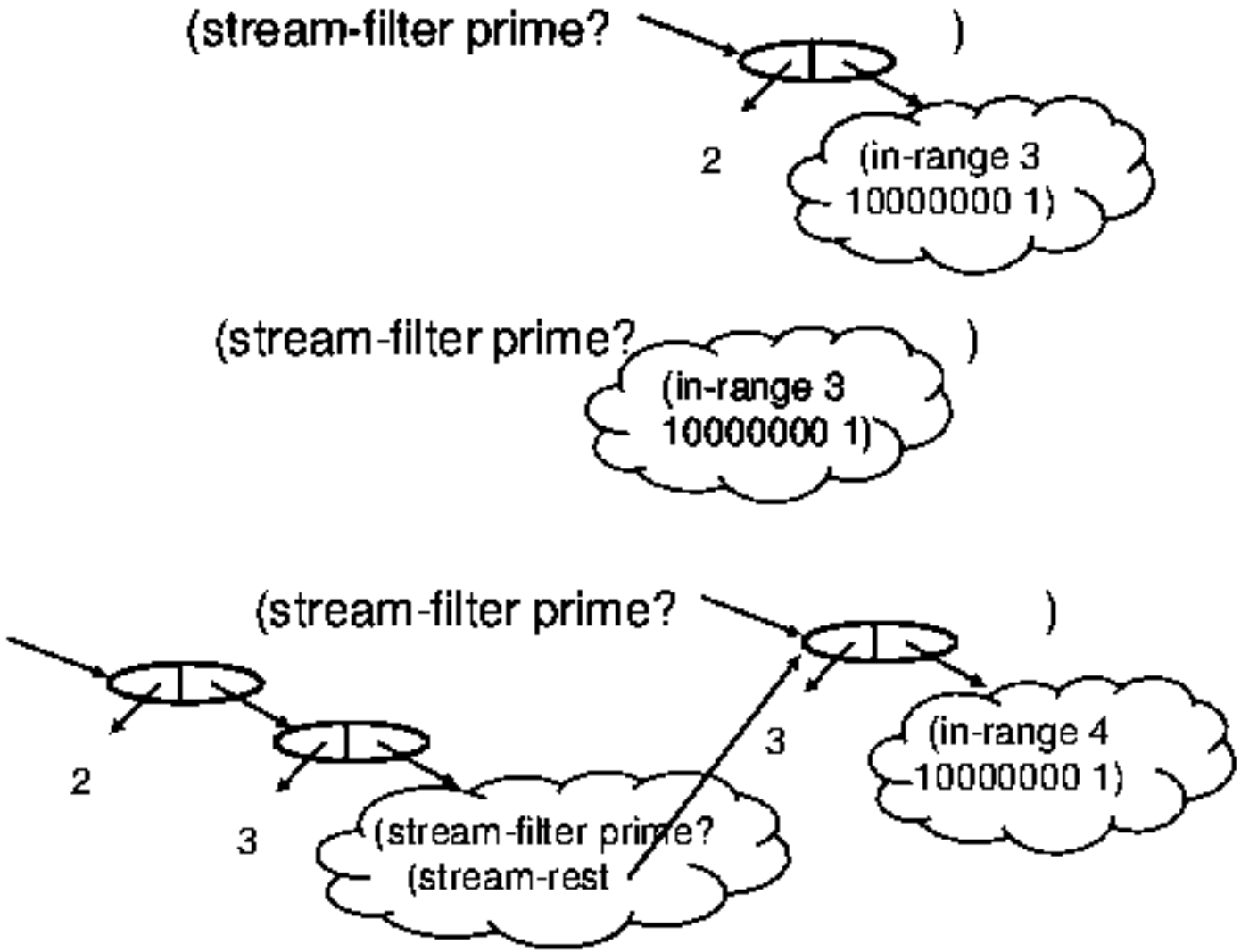
Вычисления в примере

(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

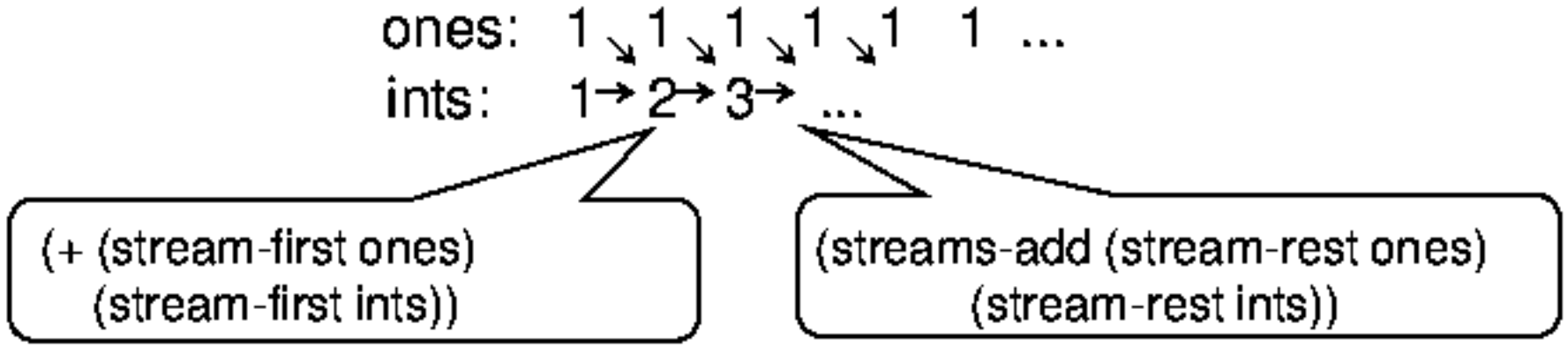
(stream-ref ones 1000) ==>1

(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



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:

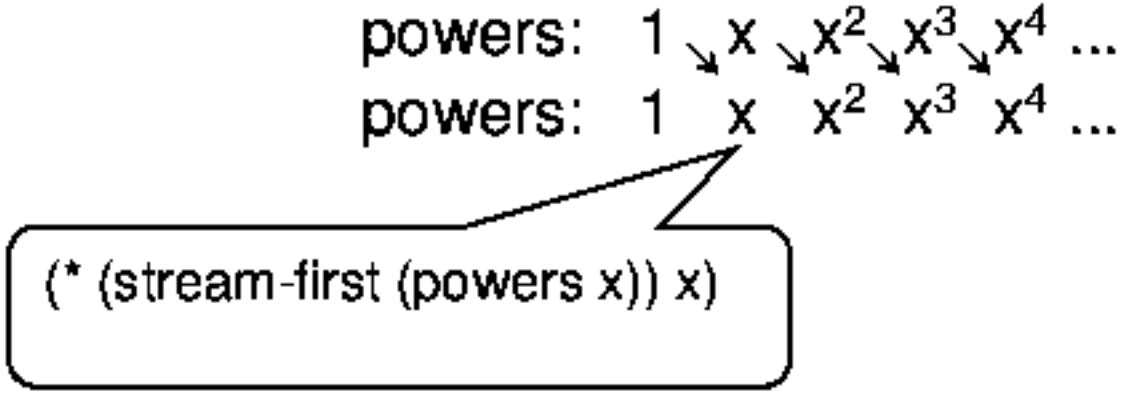
```
(define (fibgen a b) ; порождающая функция
  (stream-cons a (fib-gen (b (+ a b))))
(define fibs (fibgen 0 1))
```

Ещё одна функция для неявного порождения

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

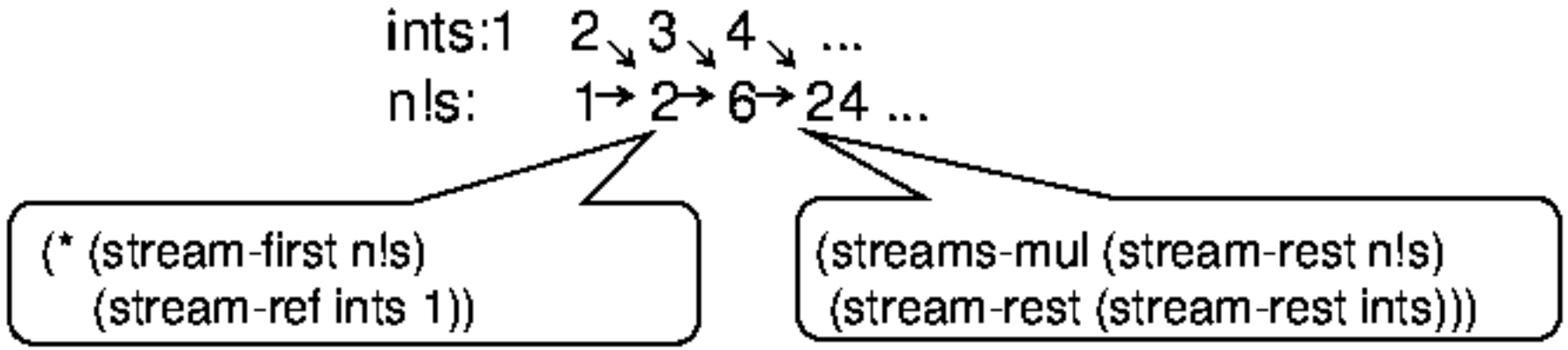
неявное описание потока степеней (1 x x^2 x^3 ...)

```
(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))))
```



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		26	27	28	29	30
31	32	33	34		36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54		56	57	58	59	60
61	62	63	64		66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84		86	87	88	89	90
91	92	93	94		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		26	27	28	29	30
31	32	33	34		36	37	38	39	40
41	42	43	44	45	46	47	48		50
51	52	53	54		56	57	58	59	60
61	62	63	64		66	67	68	69	70
71	72	73	74	75	76		78	79	80
81	82	83	84		86	87	88	89	90
	92	93	94		96	97	98	99	100

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

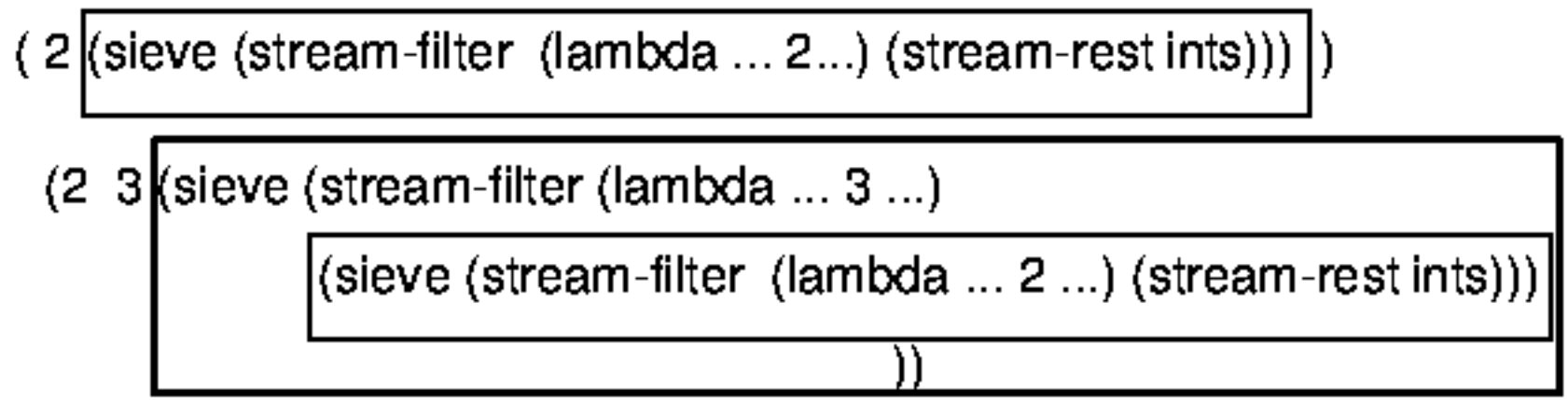
Поиск всех простых чисел

	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)))
```



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
         (make-rats (+ n 1)))))
(define rats (make-rats 1))
```

```
1/1
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))))
```

→ $g(0) = S_0$

→ $g(0) + x\ g'(0) = S_1 = S_0 + x\ g'(0)$

→ $g(0) + x\ g'(0) + x^2/2\ g''(0) = S_2 = S_1 + x^2/2\ g''(0)$

→ $g(0) + x\ g'(0) + x^2/2\ g''(0) + x^3/3!\ g'''(0) = S_3 = S_2 + x^3/3!\ g'''(0)$

Степенные ряды как потоки

```
(define (power-series g) ; даст функцию для ряда S_n
  (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 ...
```

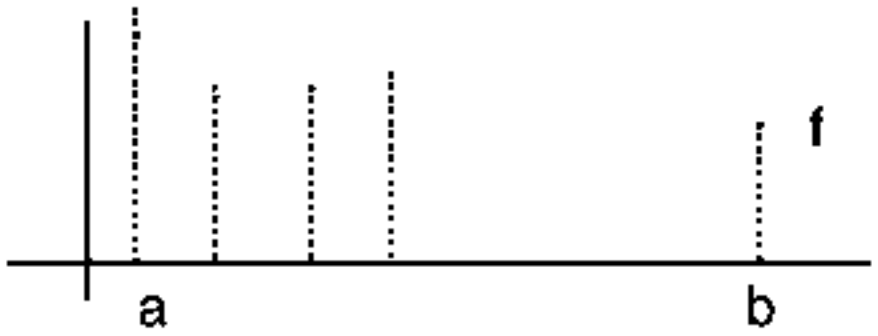
Потоковый поиск квадратного корня

- 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) ==> 1.41213562373689
```

- Вычисление описано как генерация приближений и их проверка.

Потоковое интегрирование



метод трапеций

```
(define (trapezoid f a b h) ; сумма метода трапеций ∫_a^b 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

$\pi = 4*\text{arctg}(1) = {}_0\int^1 4dx/(1+x^2)$

```
(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 с 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 с не конфликтует с с 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

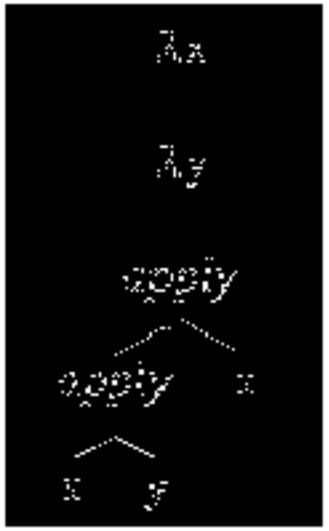
Аппликация левоассоциативная!

(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

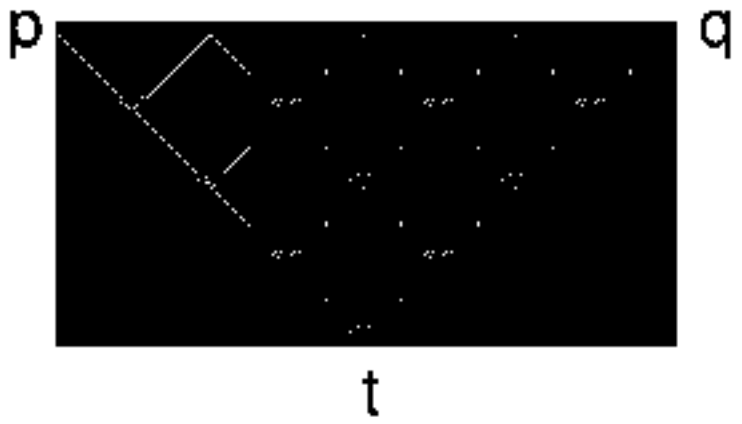
Теорема Черча-Россера

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 := $\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
- PLUS := $\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$

т. е. PLUS := $\lambda n.\ \lambda m.\ m$ SUCC n

- MULT := $\lambda m.\ \lambda f.\ m\ (n\ f)$

т. е. MULT := $\lambda m.\ \lambda n.\ m$ (PLUS n) 0

Пример

PLUS 2 3 == $(\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x))$

$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda t.\ \lambda x.\ t\ (t\ (t\ x)))$ --->

$(\lambda n.\ \lambda f.\ \lambda x.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ f\ (n\ f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$ --->

$(\lambda n.\ \lambda f.\ \lambda x.\ (\quad \lambda b.\quad (\quad b))\quad (n\ f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$--->

$(\lambda n.\ \lambda f.\ \lambda x.\ (\quad f\ (f\ \ ))\qquad)\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$ --->

$(\quad .\ \lambda f.\ \lambda x.\ f\ (f\qquad))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$ --->

$(\lambda f.\ \lambda x.\ f\ (f\ \ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))\qquad))$ -->

$(\lambda f.\ \lambda x.\ f\ (f\ \ (\lambda a.\ \lambda b.\ a(a\ (a\ b)))\qquad))$ =>

$(\lambda f.\ \lambda x.\ f\ (f\ \ f(f\ (f\ x))\ \ ))$ -- это 5

Логика в λ-исчислении

- TRUE := $\lambda x.\ \lambda y.\ x$
- FALSE := $\lambda x\ .\ \lambda y.\ y$
- AND := $\lambda p.\ \lambda q.\ p\ q\ p$
- OR := $\lambda p.\ \lambda q.\ p\ p\ q$
- NOT := $\lambda p.\ \lambda a.\ \lambda b.\ p\ b\ a$
- IFTHENELSE := $\lambda p.\ \lambda a.\ \lambda b.\ p\ a\ b$