ВВЕДЕНИЕ В ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

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

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. Объектно-ориентированное программирование в Scheme

Раздел 3:

- Тема 8. Основные сведения о λ-исчислении

Балльная система

максимум
≈ 140 баллов

критерий 2014:
50..70 «удовл»
71..110 «хор»
111..150 «отл»

обязательная сдача практических заданий!!!

Семинары/практикум

- Среда программирования Dr. Racket

  racket-lang.org
- Общее первое задание «Доктор»

  4 блока упражнений, с 1 по 3й обязательные
- Индивидуальное второе задание «Генетическое программирование»

  4 этапа выполнения с промежуточными сдачами

Web-страница: sp.cs.msu.ru/scheme

ВК-группа: vk.com/sp_scheme

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
- Чернов А. В. Учебное пособие по заданию «Генетическое программирование». – М. ВМК МГУ. 2006

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

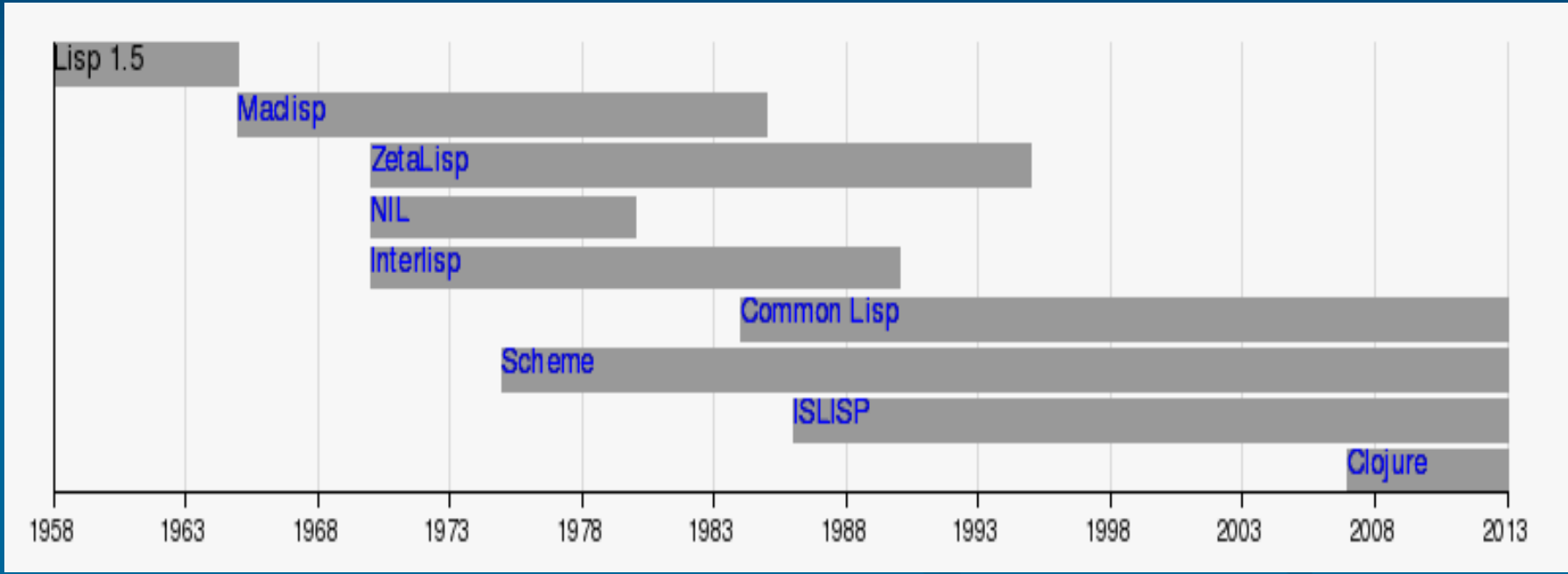
- Функциональные программы легко писать.
- Функциональные программы короче императивных.
- Функциональные программы легче понимать и анализировать.
- Модульность – естественное свойство функциональных программ.
- Функциональные языки удобны при решении задач ИИ.

Исторический экскурс

Джон Маккарти (1927-2011) MIT

В 1958 году создал язык LISP (*LI*St *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

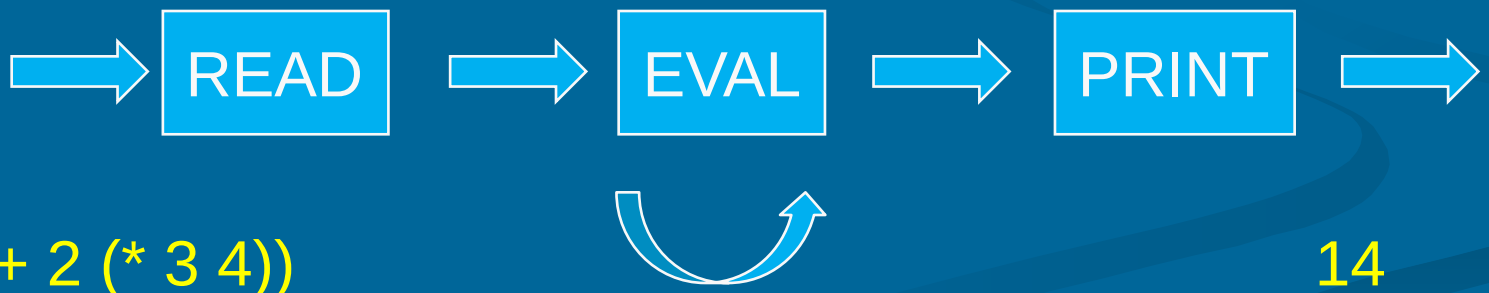
Отличия Scheme от обычных (императивных) языков программирования

- 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, ПОЛИЗ

  (+ 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



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 write или display

Знакомство со Scheme
Выражения

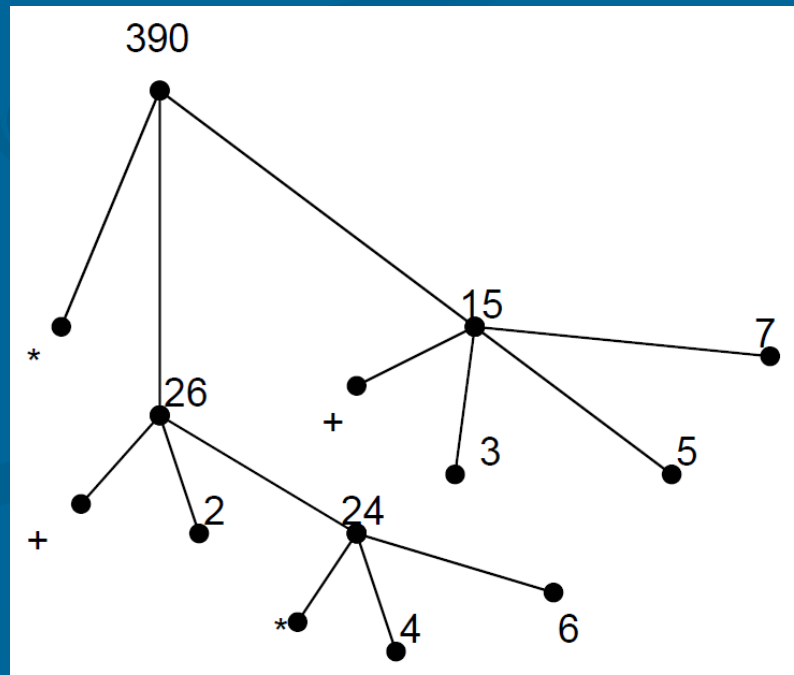
- Литералы
  - Литеры #\a #\A #\newline #\space
  - Числа -1 1/6 10.005 #o777 2-i
  - Булевы значения #t #f
  - Строки "Hello \"world#\\"
  - «Цитаты» (quote (+ 1 2)) или 'xyz
  - Пустой список '() или empty
- Имена xyz x->y
- Спецформы (define и т. п.)
- Вызовы функций

Выражения (продолжение)
Вызовы функций (или комбинации)

- Запись комбинации: $(c_1\ c_2\ \ldots c_n)$
- 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 (display "Input N:")
       (newline)
       (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

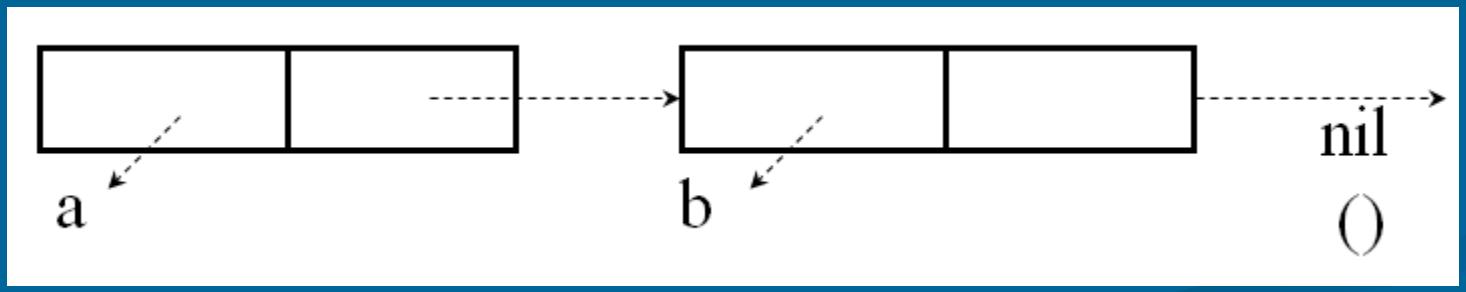
Знакомство со Scheme
Строки

- 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 empty () (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))

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.