ЛЕКЦИЯ 1
Основные сведения о языке Scheme

Аргументы в пользу функционального программирования

Считается, что:

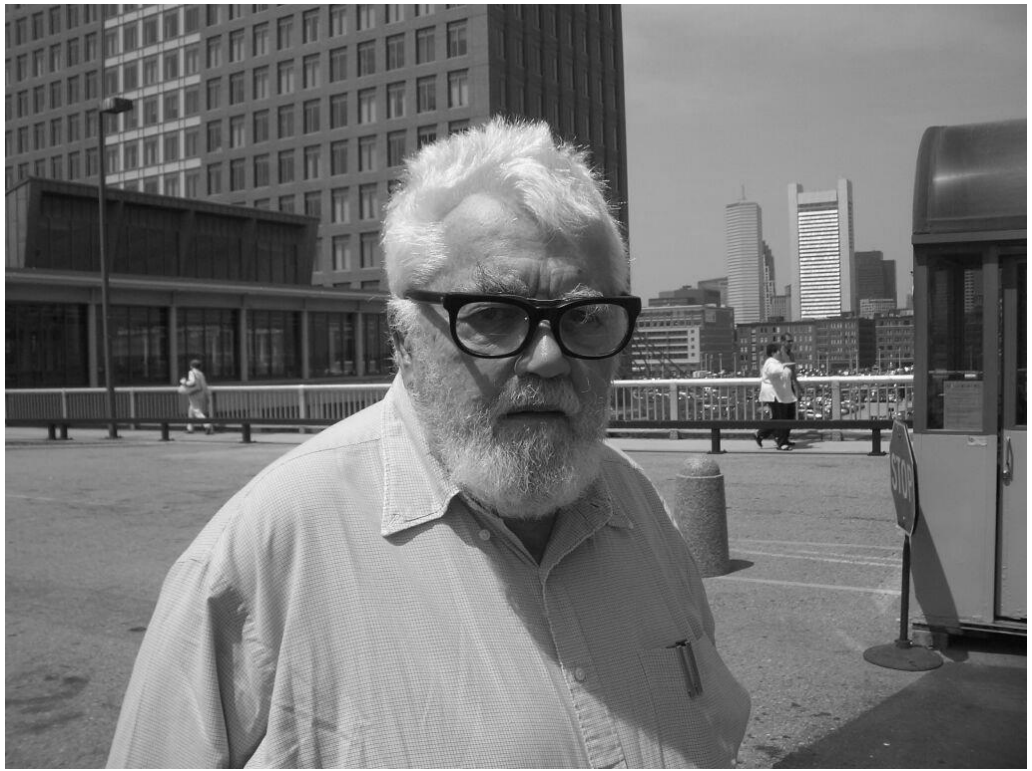
- Функциональные программы легко писать.
- Функциональные программы короче императивных.
- Функциональные программы легче понимать и анализировать.
- Модульность – естественное свойство функциональных программ.
- Функциональные языки удобны при решении задач ИИ.

В рамках курса не ставится задача присоединить Вас к лагерю сторонников (противников?) функционального программирования!

Исторический экскурс

Джон Маккарти

(1927-2011) MIT



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

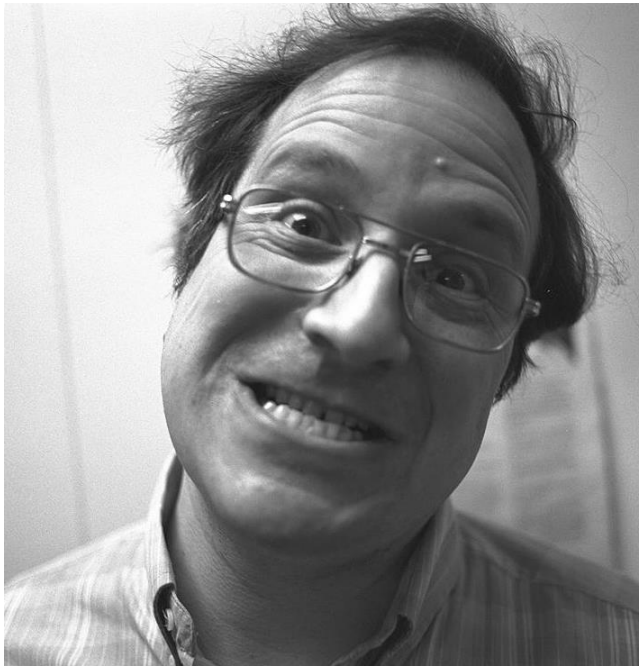
Диалекты языка Lisp

среди основных диалектов выделим

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

Язык Scheme

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



Джеральд Сассман

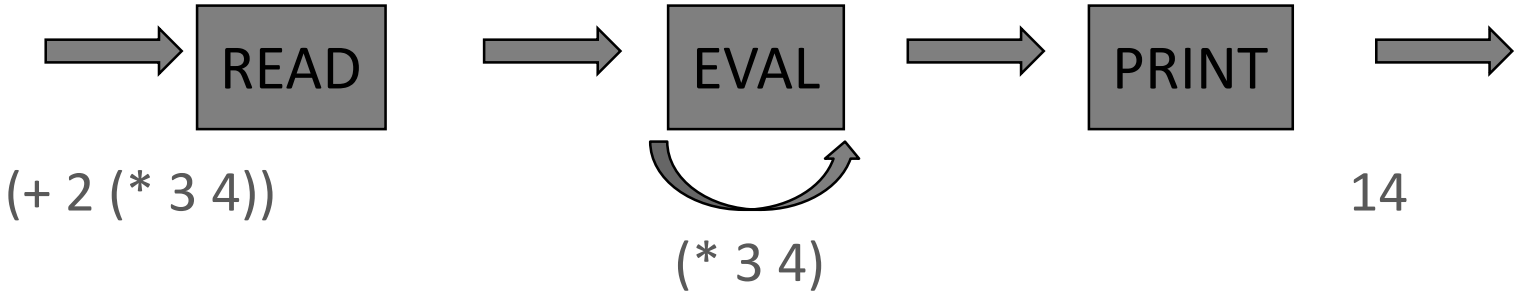
Гай Стил

Отличия Scheme от императивных языков программирования

- Основа -- 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 набором функций, вызывающих друг друга.

Отличия Scheme от 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 императивных языков программирования

- Автоматическое управление памятью.
- Управляющая структура программы -- рекурсия.
  - нет циклов
  - нет переменных
  - нет присваиваний *[в «чистом» Scheme, в «грязном» есть]*
- Динамическая типизация.

Особенности Scheme среди диалектов Lisp

- Минималистичный язык
- Длинная арифметика
- Ленивые вычисления

Знакомство со Scheme. Имена и окружения

- Идентификаторы x x->y name#
  не могут включать в себя разделители ( ) ; " ' ` | [ ] { } , или начинаться с #
- Связать имя и значение позволяет define
  > (define size 2)
  > (define dblsize (+ size size))
- define – специальная форма, *вычисляющая* значение 2-го аргумента и связывающая вычисленное значение с 1-м аргументом
- *Стрелочная диаграмма* size ⟶ 2 dblsize ⟶ 4
- Окружение – место, где хранятся связывания. Новое окружение создается из старого при добавлении связываний. Если добавляется новое связывание имени, то прежнее связывание этого имени *затеняется*.

Знакомство со Scheme. Внешнее представление

- Почти каждое 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
- Пустое значение: > (void) ->
  но > (list (void)) -> (#void)

Знакомство со Scheme. Выражения

- Литералы *(то, что является своим значением)*
  - Литеры: #\a #\A #\newline #\space
  - Числа: -1 1/6 10.005 #b101 #o777 #x3BB -2-3i -inf.0 +inf.0
  - Булевы значения: #t #f
  - Строки: "Hello \"world#\\"
  - Символы, «цитаты» *(куски кода)*: (quote (+ 1 2)) или 'xyz
  - Пустой список: () или null
  - Список, записанный с quote или ': (quote (+ 1 2 3)) или '(+ 1 2 3)
- Имена: xyz x->y
- Спецформы: define *и т. п.*
- Вызовы функций (*комбинации*): (+ 1 2)

Вызовы предопределённых функций

- Запись вызова (комбинация): (c_1 c_2 …c_n)
- Вычисление вызова:

a) найти значения всех c_i *(стандарт не определяет порядок вычисления c_i)*

b) применить предопределённую функцию, являющуюся значением c_1 к значениям остальных c_i

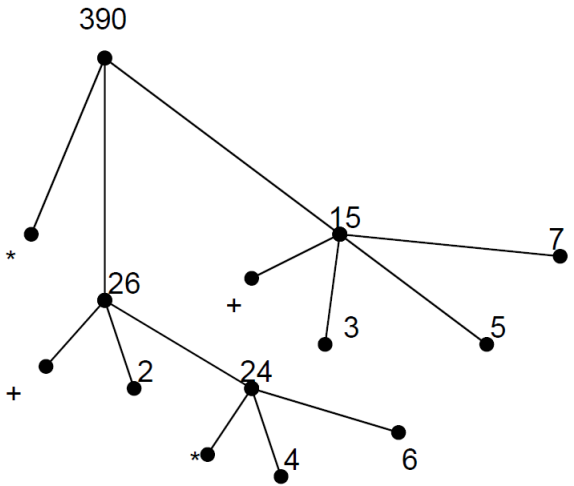
- Вычисление комбинаций рекурсивно.
- Комбинация в виде дерева

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

- Спецформы – не комбинации!

  > () -> error! (нет c_1!)

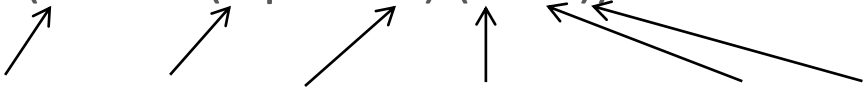
  > '() -> () (пустой список – литерал)



Знакомство со Scheme. «Свои» функции

- Определение своей (*Вашей*) 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)))
```

- Можно узнать, является ли значение функцией:

```
> (procedure? square)          -> #t
> (procedure? +)               -> #t
> (procedure? '+)              -> #f
```

«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)

- Другой способ вычисления -- *нормальный порядок* (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))) -->*

  } подстановка

  (+ (6 6) (* 10 10)) -->*
  (+ 36 100) -->
  -> 136

  } редукция

- При *норм. порядке* пока что-то не понадобится, оно не вычисляется.
- (+ 5 1) и (* 5 2) считали дважды
- Ранее считали в *аппликативном порядке* («вычисли все 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
> (procedure? (lambda (x) (+ x x)))       -> #t
```

- (define (<имя> <параметры>) <тело>)

  на самом деле сокращённо

  (define <имя> (lambda (<параметры>) <тело>))

- «Синтаксический сахар» – избыточные конструкции языка, введённые ради удобства тех, кто на нём пишет.

«Свои» функции (продолжение)

- Из-за двух версий define есть возможность «скармливать» Scheme сомнительные конструкции:

```
> (define devils_dozen 13)
> devils_dozen  ->  13
> (devils_dozen) -> error! ; (c1 – не функция!)
> (procedure? devils_dozen) -> #f ; (devils_dozen – не функция!)
> (define (devils_dozen2) 13)
; т. е. (define devils_dozen2 (lambda () 13))
> (devils_dozen2)  ->  13
> devils_dozen2 ->  #<procedure:devils_dozen2>
> (procedure? devils_dozen2) -> #t
```

Прагматика функции без параметров, возвращающей один и тот же результат, *сомнительна. Плохой стиль! Но по ходу курса мы найдем прагматику для таких функций.*

Спецформа cond

- «Разбор 0 или более 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 $p_i \neq$ #f.
- 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.

  > (cond) -> ; ничего > (cond (#f 1) (#f 2)) -> ; тоже ничего
- Пример: > (define (sign x) (cond ((< x 0) -1)
  ((= x 0) 0)
  (else 1)))

Логические спецформы and и or. Функция not

- (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 *не ложны*, то значение and = значению <e_n> (≠ #f).

  > (and) -> #t

  > (and 1) -> 1

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

  > (or) -> #f

  > (or #f #f) -> #f

  > (or #\f #f) -> #\f

- Можно использовать *функцию* not. > (not #f) -> #t

not от всего, что не #f, даст #f:

> (not 2) -> #f > (not #t) -> #f

«Продвинутые» варианты альтернатив в cond

- Любая альтернатива в cond может иметь вид:
  (<p_i> => <e_i>) ; <e_i> имеет значением функцию от одного элемента
  При срабатывании такой альтернативы к результату вычисления <p_i> применяется функция – значение <e_i> и результат возвращается как результат cond.
  Пример: (define (not#f? x) (cond ((not x) => not) (else #t)))
  > (not#f? -10) -> #t > (not#f? #f) -> #f
  > (not#f? #t) -> #t > (not#f? +) -> #t
- Любая альтернатива в cond может после <p_i> не содержать <e_i>.
  При срабатывании такой альтернативы, т. е., когда значение <p_i> не #f, тогда как результат возвращается само значение <p_i>.
  Пример: > (define (abs x) (cond ((< x 0) (- x)) (x)))

Спецформа if

- (if <предикат> ; сначала вычисляется предикат
  <следствие> ; если он ≠ #f, считаем следствие
  <альтернатива>) ; иначе, считаем альтернативу
- Пример: (define (abs x) (if (< x 0) (- x) x))
  > (abs -10) -> 10
- Это лишь пример. Есть стандартный abs, его не надо реализовывать самим.
- Вопрос: Можно ли if не делать спецформой, а реализовать функцией через cond так, чтобы он работал точно также как спецформа if?
  (define (my-if b t e) (cond …))

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))
- Но при любом порядке вызов (if (= 0 0) 0 (p)) *не даст зацикливание.*

```
> (if (= 0 0)  0  (p)) -> 0
```

Спецформа case

- 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.

1. Вычисляем <ключ>.
2. Сравниваем значение ключа с вариантами первой ветви. Если оно есть в <$vars_1$>, 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)
  (else 'unknown)) -> composite

Спецформа begin

(begin <exp_1>
<exp_2>
… <exp_n>)

- Вычисляет все подвыражения по порядку.
- Значением формы является значение последнего подвыражения <exp_n>.
- Помогает, если нужно сделать ввод/вывод.
- Пример: (begin (println "Input N:") (read))

Знакомство со Scheme. Числа

- Башня числовых типов:

  number

  complex 1+2i 1/2-3/4i 0-i -i +i ; внутри записи нет пробелов!

  real 0.001 3.14e-87 -3.14e80 -inf.0 +inf.0

  rational 1/3 -5/3 но не ~~-5/-3~~

  integer -1 #xff #b101 #o777

- Функции проверки типа number? real? … integer?
- Функции = < > <= >= принимают ≥2 аргументов (и + - * /)
- 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), где x > 0 и целое
- (random x y) – псевдослучайное целое число из [x, y), где y > x

Числа (продолжение)

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

```
(define (two-of-three x y z)
  (cond  ((or (<= x y z) (<= x z y)) (* y z))
         ((or (<= y x z) (<= y z x)) (* x z))
         (else (* x y))))
> (two-of-three 1 2 3)    -> 6
> (two-of-three 3 2 1)    -> 6
> (two-of-three 3 1 2)    -> 6
```

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

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)

- Изменим 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?
- Проверка на строку string?

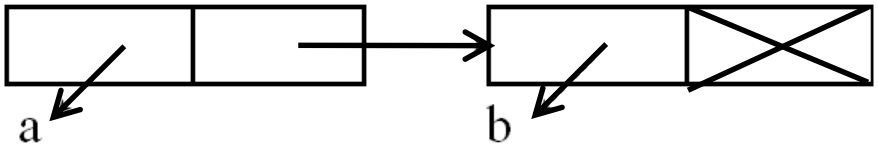
Знакомство со Scheme. Списки

Внешнее представление > '(a b c) -> (a b c) или '(a b c)

в виде точечной пары > '(a . (b . (c . ()))) -> (a b c)

- Конструирование списка из головы и хвоста-списка: 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)

Список из литералов в коде записываем с ', но список из вычисляемых значений получаем вызовом ф-ции list > (list '+ 1 (+ 1 2)) -> (+ 1 3)
> '(+ 1 (+ 1 2)) -> (+ 1 (+ 1 2))

- Проверка на список list?

> (list? '(a b)) -> #t > (list? '()) -> #t > (list? 'a) -> #f

- Длина списка length. *Считается за линейное время!*
- 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)

- Перевернуть reverse
- Проверить вхождение элемента member memv memq

> (member 2 '(1 2 3)) -> (2 3) > (member 4 '(1 2 3)) -> #f

Списки (продолжение)

Напишем свою версию list-ref

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

Своя версия length

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

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>) применяет первый аргумент-функцию ко второму, рассматривая 2ой как список аргументов вызова

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

Ещё о вычислениях

Функция (eval <выражение>) разquoteит цитаты

(далее указаны результаты работы с интерпретатором; если пытаться запускать код, то результат будет иным!)

```
> (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)))      -> ?
```

eval – мощный инструмент, который используют *при необходимости*

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.

Прагматика: не делаем «тяжелых» повторных вычислений.

Функции с переменным количеством параметров

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

```
; функция с ∀количеством параметров, возвращающая
; их значения в списке (аналог list).
(define (f . params) params)
> (f)             ->    ()
> (f 1 2 3 4 5) ->              (1 2 3 4 5)
; функция с 1≤ параметрами, добавляющая 1й к остальным
; и возвращающая список сумм
(define (first+ x . tail)
     (if (null? tail) null (cons (+ x (car tail)) (apply first+ (cons x (cdr tail))))))
> (first+ 10 1 2 3 4) ->     (11 12 13 14)
> (first+ 10)          ->        ()
```

lambda с переменным количеством параметров

- те же фокусы с явной lambda:

```
(define f (lambda params params))
> (f)              ->      ()
> (f 1 2 3 4 5)    ->      (1 2 3 4 5)

(define first+ (lambda (x . tail)
    (if (null? tail) null (cons (+ x (car tail)) (apply first+ (cons x (cdr tail)))))))
> (first+ 10 1 2 3 4)      ->      (11 12 13 14)
> (first+ 10)              ->      ()
```

- Пример lambda без параметров см. на слайде 25.

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 функции для них.
- Функции eval и apply.
- Спец. формы let, let*.