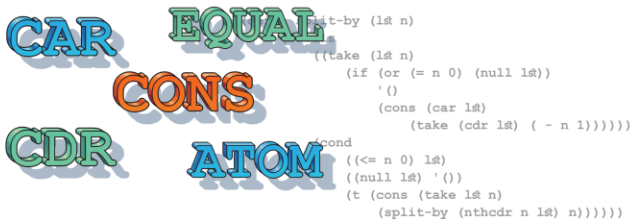
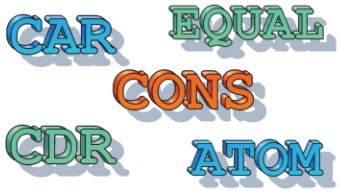


Функциональное программирование: базовый курс

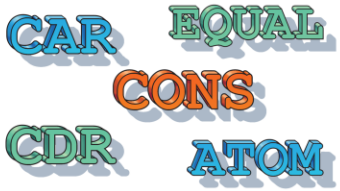
Лекция 4.

Функции и рекурсия.



Лекция 4
Функции и рекурсия

Определение функций и типы формальных аргументов

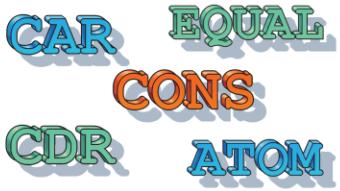


тип возвращаемого значения

список аргументов функции с указанием типов

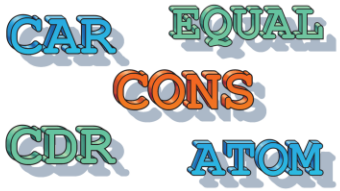
```
int sum_n(int arr[], size_t n) {
    int res = 0;
    for (int i = 0; i < n; i++)
        res += arr[i];
    return res;
}
```

возврат значения



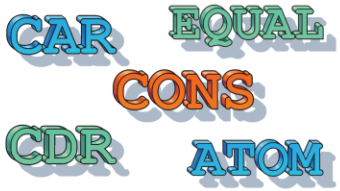
Функции в Лиспе

```
(defun sum-n (arr n)
    (let ((res 0))
        (dotimes (i n)
            (incf res (aref arr i)))
        res))
```



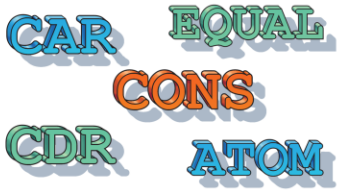

```
(defun sum-n (seq n)
    (let ((res 0))
        (dotimes (i n)
            (incf res (elt seq i)))
        res))
```

```
[1]> (sum-n #(10 20 30 40) 2)
30
[2]> (sum-n '(1 2 3 4) 2)
3
```



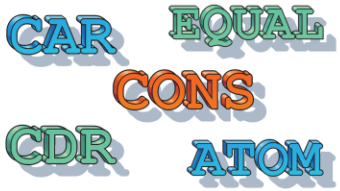
Определение функций в Лиспе

```
(defun имя-функции (список-аргументов)
    "Строка документации с описанием функции"

    ; тело функции
)
```



Строка документации

```
(defun sum-n (seq n)
    "Суммирует n первых элементов
        последовательности seq"
    (let ((res 0))
        (dotimes (i n)
            (incf res (elt seq i)))
        res))


[1]> (documentation 'sum-n 'function)
    "Суммирует n первых элементов
        последовательности seq"
```



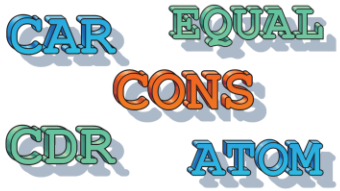
Формальные и фактические аргументы

формальные аргументы

```
[1]> (defun func (a b c)
                 (* a b c))
FUNC
```

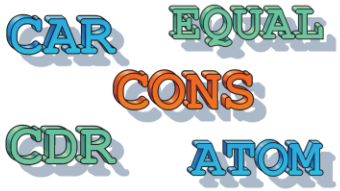
фактические аргументы

```
[2]> (func 1 6 7)
42
```



Обязательные аргументы

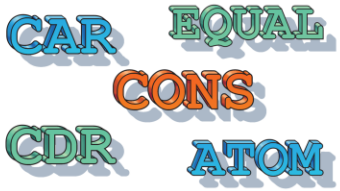
```
(defun func (a1 a2 a3)
    (list a1 a2 a3))

[1]> (func 1 2 3)
(1 2 3)
[2]> (func 1 2)
*** - EVAL/APPLY: Too few arguments
[3]> (func 1 2 3 4)
*** - EVAL/APPLY: too many arguments
```



Необязательные аргументы

- указываются в списке аргументов после ключевого слова &optional

```
(defun func (a1 a2 &optional o1 o2)
    (list a1 a2 o1 o2))

[1]> (func 1 2)
(1 2 NIL NIL)
[2]> (func 1 2 3)
(1 2 3 NIL)
[3]> (func 1 2 3 4)
(1 2 3 4)
```

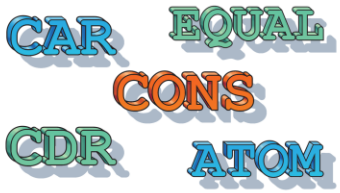


Инициализация необязательных аргументов

```
(defun func (a1 &optional (o1 42))
    (list a1 o1))
```

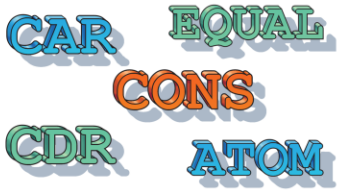
умалчиваемое значение
для аргумента o1

```
[1]> (func 1 2)
(1 2)
[2]> (func 1)
(1 42)
```



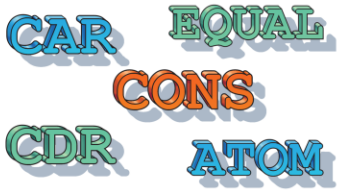
Проверка инициализации необязательных аргументов

```
(defun func (&optional (o1 42 o1-p))
    (list o1 o1-p))

[1]> (func)
(42 NIL)
[2]> (func 42)
(42 T)
[3]> (func 1)
(1 T)
```



Пример простой функции сложения чисел

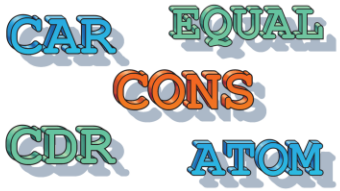
```
(defun my-plus (&optional
    (o1 0) (o2 0) (o3 0))
    (+ o1 o2 o3))

[1]> (my-plus)
0
[2]> (my-plus 1)
1
[3]> (my-plus 1 41)
42
[4]> call-arguments-limit
4096
```



Агрегирование необязательных аргументов

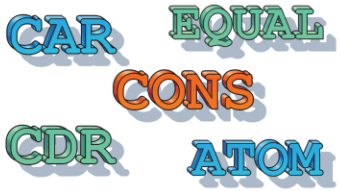
- агрегирование необязательных аргументов выполняется с помощью ключевого слова &rest

```
(defun func (a1 a2 &rest r)
    (list a1 a2 r))

[1]> (func 1 2)
(1 2 NIL)
[2]> (func 1 2 3 4 5 6)
(1 2 (3 4 5 6))
```



Пример простой функции сложения чисел

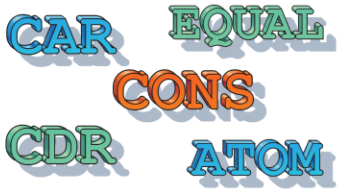
```
(defun my-plus (&rest r)
    (let ((res 0))
        (dolist (arg r)
            (incf res arg))
    res))

[1]> (my-plus)
0
[2]> (my-plus 1 2 3 4)
10
```



Ключевые аргументы

- указываются в списке аргументов после ключевого слова &key

```
(defun func (&key (k1 0) (k2 11))
    (list k1 k2))

[1]> (func)
(0 11)
[2]> (func :k2 42)
(0 42)
[3]> (func :k2 2 :k1 4)
(4 2)
```



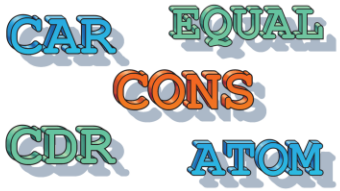
Ключевые аргументы

```
(defun func (&key
        ((:some-arg k1) (random 100) k1-p)
        ((:another-arg k2) (random 100) k2-p))
    (list k1 k2))
```

имя для указания фактического аргумента

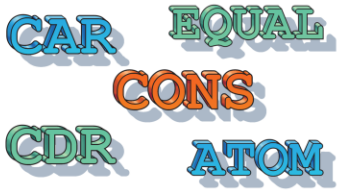
имя соответствующего формального аргумента внутри функции

```
[1]> (func)
(35 60)
[2]> (func :another-arg 42)
(84 42)
[3]> (func :another-arg 42 :some-arg 42)
(42 42)
```



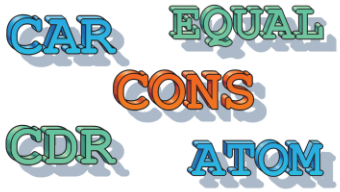
Ключевые аргументы

```
(defun func (&key k1 k2 &allow-other-keys)
    (list k1 k2))


[1]> (func)
(NIL NIL)
[2]> (func :k2 42)
(NIL 42)
[3]> (func :k2 42 :some-arg 42)
(NIL 42)
```



Пример использования &allow-other-keys

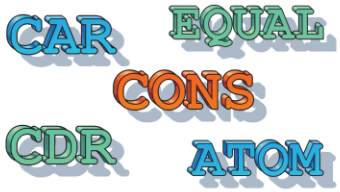
```
(defun make-strings-array
    (str dims
        &rest keyword-pairs
        &key (start 0) end &allow-other-keys)

  (apply #'make-array dims
         :initial-element
                   (subseq str start end)
         :allow-other-keys t
         keyword-pairs))
```



Комбинирование типов аргументов в списке аргументов

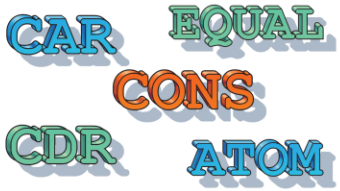
- порядок типов аргументов в списке:
  - обязательные аргументы
  - необязательные аргументы &optional
  - остальные аргументы &rest
  - ключевые аргументы &key

```
[1]> lambda-list-keywords
(&OPTIONAL &REST &KEY &ALLOW-OTHER-KEYS &AUX &BODY &WHOLE
&ENVIRONMENT)
```



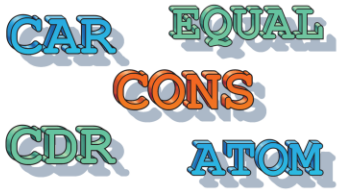
Нежелательное комбинирование необязательных и ключевых аргументов

```
(defun func (a1 &optional o1 o2 &key k1)
    (list a1 o1 o2 k1))

[1]> (func 1 2 3)          ; OK
(1 2 3 NIL)
[2]> (func 1 2 3 :k1 42)   ; OK
(1 2 3 42)
[3]> (func 1 :k1 42)       ; Ошибка
(1 :k1 42 NIL)
[4]> (func 1 2 :k1 42)     ; Ошибка
*** - FUNC: keyword arguments in (42) should occur pairwise
```



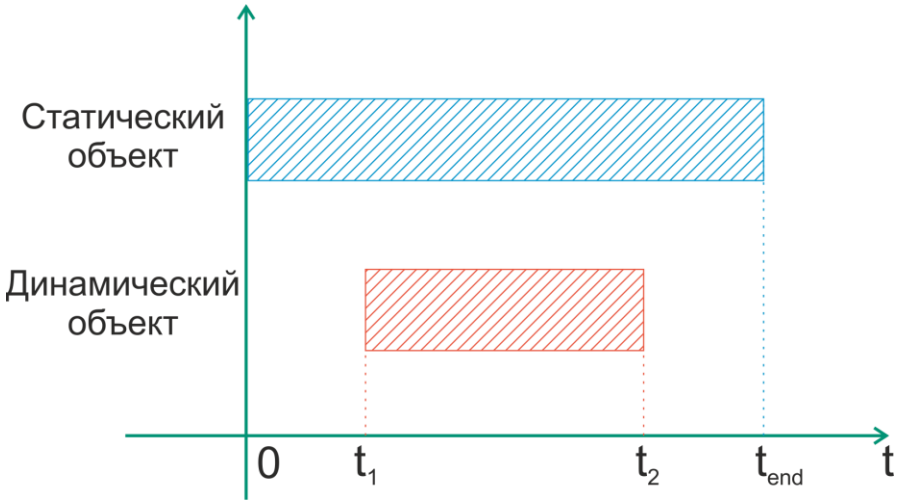
Лекция 4
Функции и рекурсия

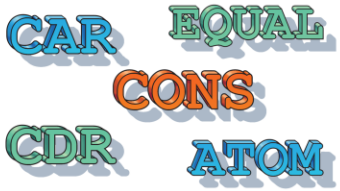
Локальные переменные и функции



Время жизни

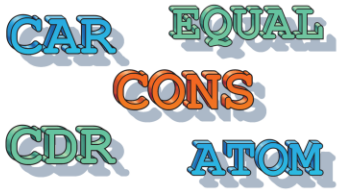
- время жизни (extent)
  - неограниченное (indefinite extent)
  - ограниченное (dynamic extent)





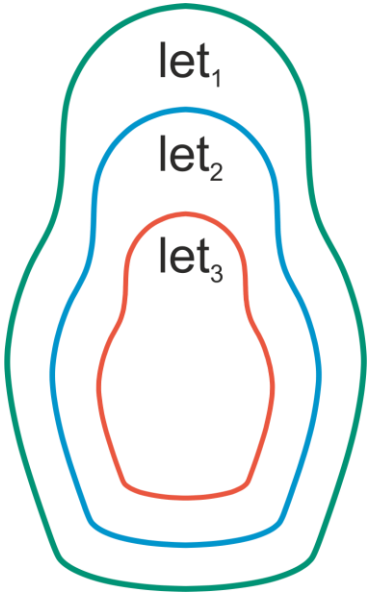
Управление памятью

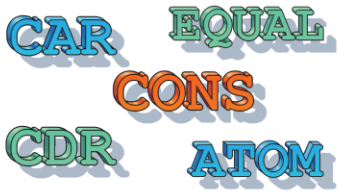
Языки с автоматическим управлением памятью	Языки с ручным управлением памятью
Java, C#, PHP, Perl, Ruby, Python, Go, Swift, JavaScript, Smalltalk, все функциональные языки: Lisp, Scheme, Clojure, ML, Ocaml, F#, Haskell, …	C, C++, Objective-C Pascal, Delphi, Oberon Fortran Ada



Область видимости

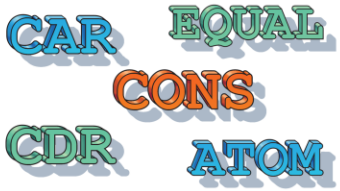
- область видимости (scope)
  - глобальная (global, indefinite)
  - локальная, или лексическая (local or lexical)





Область видимости и время жизни объектов в Лиспе

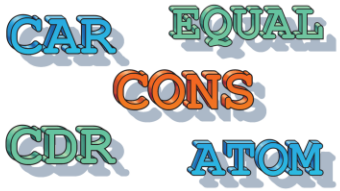
	неограниченное время жизни (indefinite extent)	ограниченное время жизни (dynamic extent)
глобальная область видимости (global scope)	глобальные константы и переменные (t, nil, pi и т.д.), в том числе определенные с помощью defconstant	переменные, определенные с помощью defvar/defparameter, а также объявленные как special
локальная область видимости (lexical scope)	переменные, связанные с помощью конструкций типа let/flet, а также переменные в замыканиях (closures)	точки выхода, установленные с помощью конструкций типа block/tagbody, а также ресурсы, выделенные в конструкциях типа with-…



Форма let

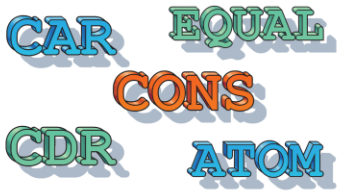
```
(let список_переменных список_форм)
```

```
[1]> (let (x y) (list x y))
(NIL NIL)
[2]> (let ((x 0) (y 1)) (list x y))
(0 1)
[3]> (let  ((x (random 100))
            (y (random 100)))
           (list x y))
(50 76)
```



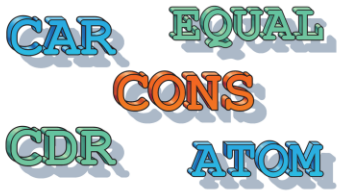
Вложенные формы let

```
[1]> (let   ((x 1) (y 2))
            (format t "x = ~d, y = ~d~%" x y)
            (let ((x 11) (y 22))
                 (format t "x = ~d, y = ~d~%" x y))
            (setf x 0)
            (format t "x = ~d, y = ~d~%" x y))

x = 1, y = 2
x = 11, y = 22
x = 0, y = 2
NIL
```



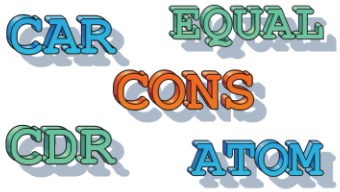
Пример использования let: решение квадратного уравнения

```
(defun square-eqn-roots(a b c)
    "Функция возвращает список
        корней квадратного уравнения
        с коэффициентами a b c"

    ; тело функции
)

[1]> (square-eqn-roots 1 0 -4)
(2 -2)
```



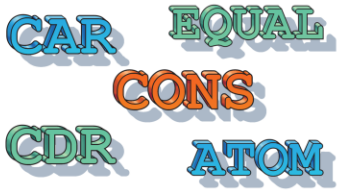
Пример использования let: решение квадратного уравнения

```
(defun square-eqn-roots (a b c)
    "Функция возвращает список
        корней квадратного уравнения
        с коэффициентами a b c"
    (let ((D (- (* b b) (* 4 a c)))
          (x1 (/ (+ (- b) (sqrt D)) (* 2 a)))
          (x2 (/ (- (- b) (sqrt D)) (* 2 a))))
        (list x1 x2)))

[1]> (square-eqn-roots 1 0 -4)
*** - LET: variable D has no value
```



Пример использования let: решение квадратного уравнения

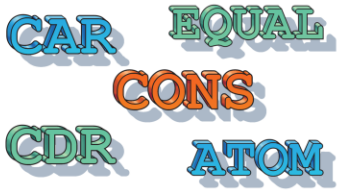
```
(defun square-eq-roots (a b c)
    "Функция возвращает список
        корней квадратного уравнения
        с коэффициентами a b c"
    (if (= a 0)
        (error "Не квадратное уравнение (a = 0)"))
    (let* ((D (- (* b b) (* 4 a c)))
           (x1 (/ (+ (- b) (sqrt D)) (* 2 a)))
           (x2 (/ (- (- b) (sqrt D)) (* 2 a))))
        (if (= x1 x2) (list x1) (list x1 x2))))

[1]> (square-eqn-roots 1 0 -4)
(2 -2)
[2]> (square-eqn-roots 3 -2 6)
(#C(1/3 1.3743685) #C(1/3 -1.3743685))
```



Дополнительные аргументы функции

- указываются в списке аргументов после ключевого слова &aux

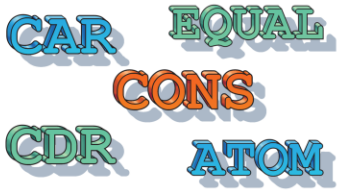
```
(defun sum-n-1 (seq n)
     (let ((res 0))
          (dotimes (i n)
               (incf res (elt seq i)))
          res))

(defun sum-n-2 (seq n &aux (res 0))
     (dotimes (i n)
          (incf res (elt seq i)))
     res)

[1]> (sum-n-2 '(1 2 3 4) 2)
3
[2]> (sum-n-2 '(1 2 3 4) 2 10)
*** - EVAL/APPLY: too many arguments given to SUM-N-2
```



Деконструкция списков

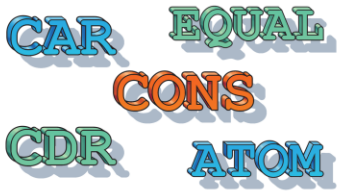
(**destructuring-bind** список_переменных
исходный_список список_форм)

```
[1]> (destructuring-bind ((x1 x2) (square-eqn-roots 1 0 -4)
                  (format t "x1 = ~a, x2 = ~a" x1 x2))
(2 -2)
NIL

[2]> (destructuring-bind ((x1 x2) (square-eqn-roots 1 0 4)
                  (format t "x1 = ~a, x2 = ~a" x1 x2))
x1 = #C(0 2), x2 = #C(0 -2)
NIL

[3]> (destructuring-bind ((x1 x2) (square-eqn-roots 1 0 0)
                  (format t "x1 = ~a, x2 = ~a" x1 x2))
*** - The object to be destructured should be a list with 2 elements, not (0).

[4]> (destructuring-bind ((x1 &optional x2) (square-eqn-roots 1 0 0)
                  (format t "x1 = ~a, x2 = ~a" x1 x2))
x1 = 0, x2 = NIL
NIL
```



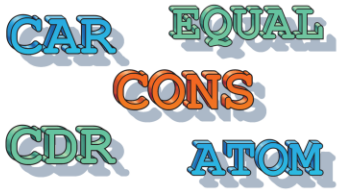
Деконструкция списков

```
[1]> (destructuring-bind (x1 ((x2) x3) x4)
           '(11 ((22) 33) 44)
           (list x1 x2 x3 x4))
(11 22 33 44)
```



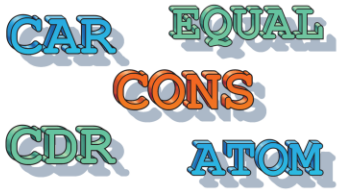
Создание локальных функций с помощью flet и labels

```
(defun square-eqn-roots-2 (a b c)
    (labels ((discr (a b c)
                    (- (* b b) (* 4 a c)))
             (root (a b c func)
                   (/ (funcall func (- b)
                               (sqrt (discr a b c)))
                      (* 2 a))))
        (list
            (root a b c #'+)
            (root a b c #'-))))

[1]> (square-eqn-roots-2 1 0 -4)
(-2 2)
```



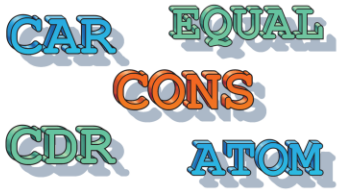
Лекция 4
Функции и рекурсия

Возвращаемые значения функции



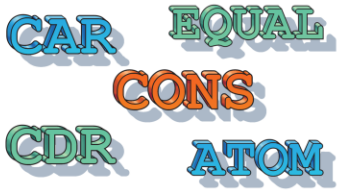
Возврат значения из функции

```
(block nowhere
    ...
    (return-from nowhere 42)
    ...)

(defun nowhere ()
    ...
    (return-from nowhere 42)
    ...)
```



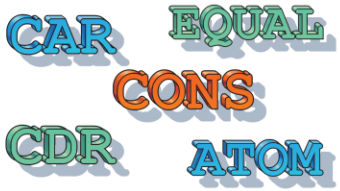
Возврат значения из функции

```
CL-USER(1): (macroexpand
                  '(defun nowhere ()
                      (return-from nowhere 42)
                      (print "we won't see this")))

(PROGN
(EVAL-WHEN (:COMPILE-TOPLEVEL) (SB-C:%COMPILER-DEFUN 'NOWHERE 'NIL T))
(SB-IMPL::%DEFUN 'NOWHERE
                 (SB-INT:NAMED-LAMBDA NOWHERE
                     NIL
                   (BLOCK NOWHERE
                     (RETURN-FROM NOWHERE 42)
                     (PRINT "we won't see this")))
                 (SB-C:SOURCE-LOCATION)))
T
```



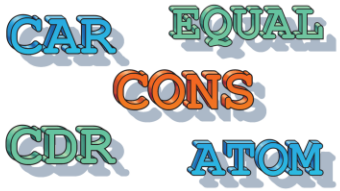
Неявные блоки без имени

```
[1]> (dolist (i '(1 2 3 0 4 5))
               (when (= i 0) (return))
               (print i))

1
2
3
NIL
```



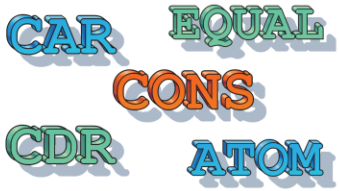
Неявные блоки без имени

```
CL-USER(1): (macroexpand
                        '(dolist (i '(1 2 3 0 4 5))
                                 (when (= i 0) (return))
                                 (print i)))

(BLOCK NIL
  (LET ((#:N-LIST627 '(1 2 3 0 4 5)))
    (TAGBODY
     #:START628
      (UNLESS (ENDP #:N-LIST627)
        (LET ((I (TRULY-THE (MEMBER 5 4 0 3 2 1)
                            (CAR #:N-LIST627))))
          (SETQ #:N-LIST627 (CDR #:N-LIST627))
          (TAGBODY (WHEN (= I 0) (RETURN)) (PRINT I)))
        (GO #:START628))))
  NIL)
T
```



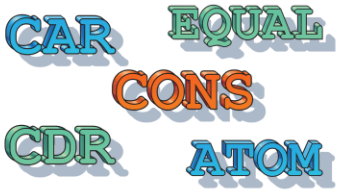
Множественные возвращаемые значения

```
[1]> (truncate 3.5)
3 ;
0.5
[2]> (* 2 (truncate 3.5))
6
[3]> (parse-integer
               "42 is the Answer"
               :junk-allowed t)
42 ;
2
[4]> (+ 2 (parse-integer "2"))
4
```



Поиск в списке свойств

```
[1]> (setf *plist* '(one 1 two 2 three 3 four 4))
(ONE 1 TWO 2 THREE 3 FOUR 4)

[2]> (get-properties *plist* '(two four))
TWO ;
2 ;
(TWO 2 THREE 3 FOUR 4)
```

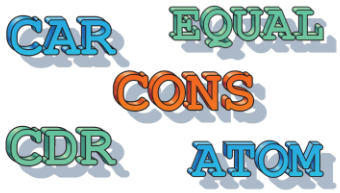


Доступ к множественным возвращаемым значениям

```
(multiple-value-bind
        список_переменных  форма_М
        прочие_формы)
```

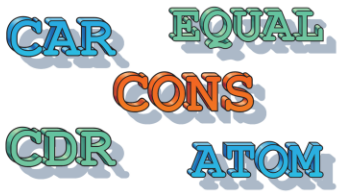
список переменных, каждая из которых связывается с одним из значений, которые возвращает форма_М

форма, которая возвращает несколько значений



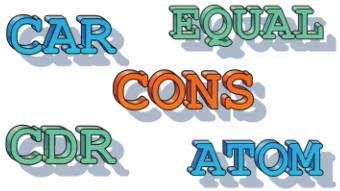
Доступ к множественным возвращаемым значениям

```
[1]> (multiple-value-bind
                (key val lst)
                    (get-properties
                            *plist* '(two four))

                (format t
                        "~a : ~a" key val))
TWO : 2
NIL
```



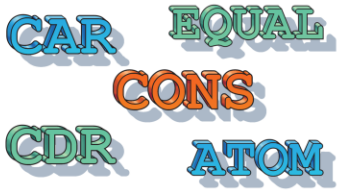
Возврат из функции нескольких значений

```
[1]> (values 1 2 3)
1 ;
2 ;
3
[2]> (values-list '(11 22 33))
11 ;
22 ;
33
```

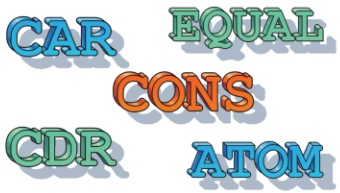


Возврат из функции нескольких значений

```
(defun count-odds (lst)
    (let ((count 0) (odds-list nil))
        (dolist (i lst)
            (when (oddp i)
                (push i odds-list)
                (incf count)))
        (values count (nreverse odds-list))))

[1]> (count-odds '(1 2 3 4))
2 ;
(1 3)
[2]> (cound-odds '(2 4 6))
0 ;
NIL
[3]> (* 10 (count-odds '(1 2 3 4)))
20
```

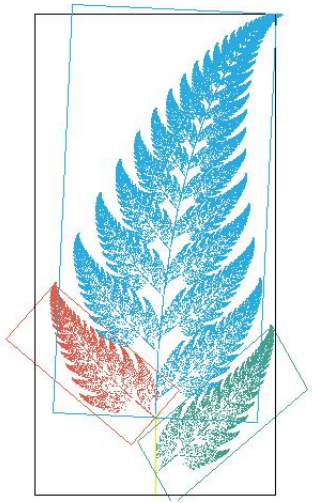


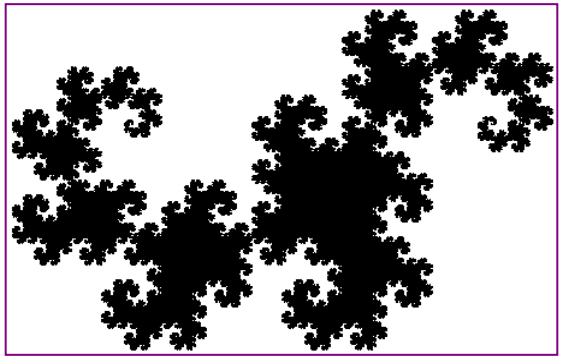
Итерация и рекурсия

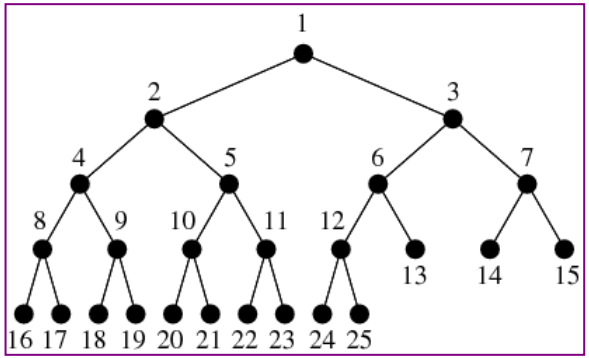


Что такое рекурсия

- **Рекурсия** — способ общего определения объекта или действия через сам этот объект или действие







Числа Фибоначчи

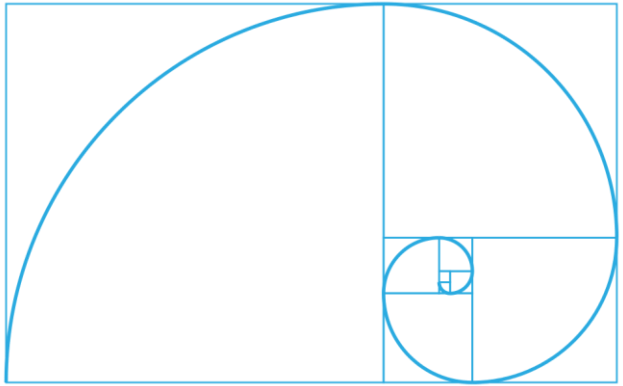
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

$$F_0 = 1$$

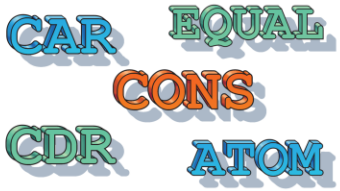
$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

рекуррентная формула
ряда Фибоначчи



спираль Фибоначчи

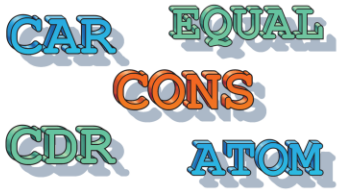


Рекурсивное вычисление чисел Фибоначчи

```
(defun fib (n)
    "n-ный член ряда Фибоначчи"
    (if (< n 2)
        1
        (+ (fib (- n 1)) (fib (- n 2)))))
```

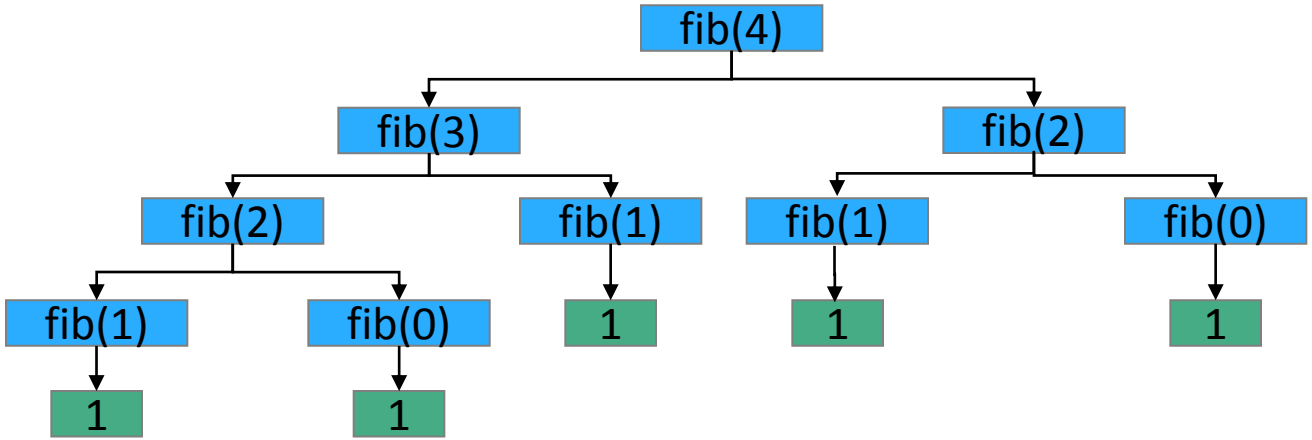
F_{n-1} F_{n-2}

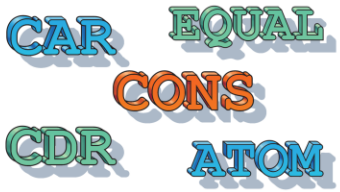
```
[1]> (fib 4)
5
[2]> (fib 9)
55
[3]> (fib 44)  ; может занять много времени
1134903170
```



Глубина рекурсии

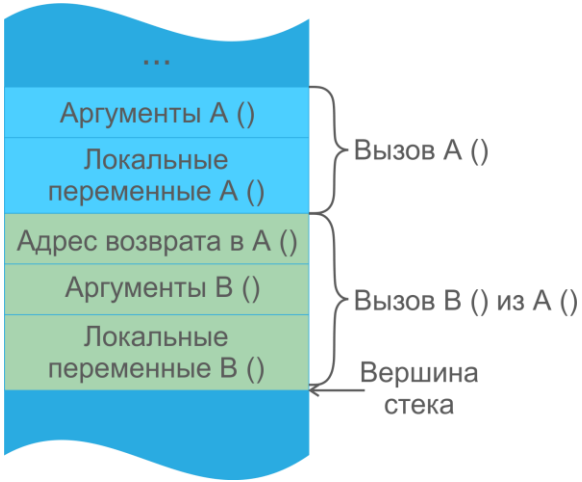
```
(defun fib (n)
    "n-ный член ряда Фибоначчи"
    (if (< n 2)
        1
        (+ (fib (- n 1)) (fib (- n 2)))))
```



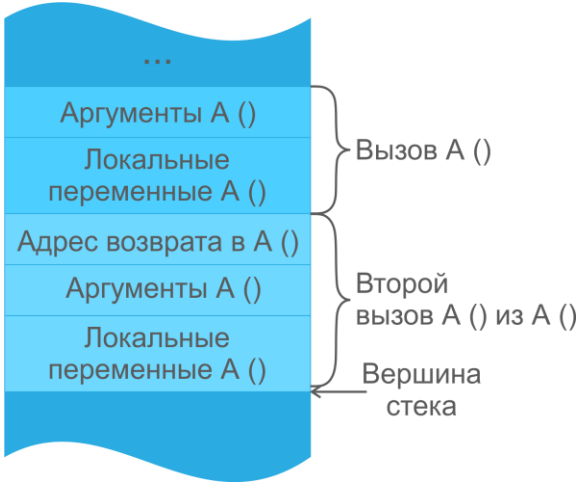


Использование стека при рекурсивных вызовах

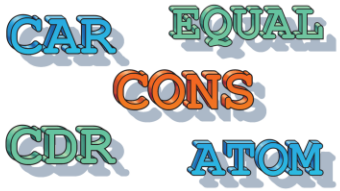
- Стек – специальная область памяти для хранения адресов возврата, регистров процессора и другой информации о состоянии потока



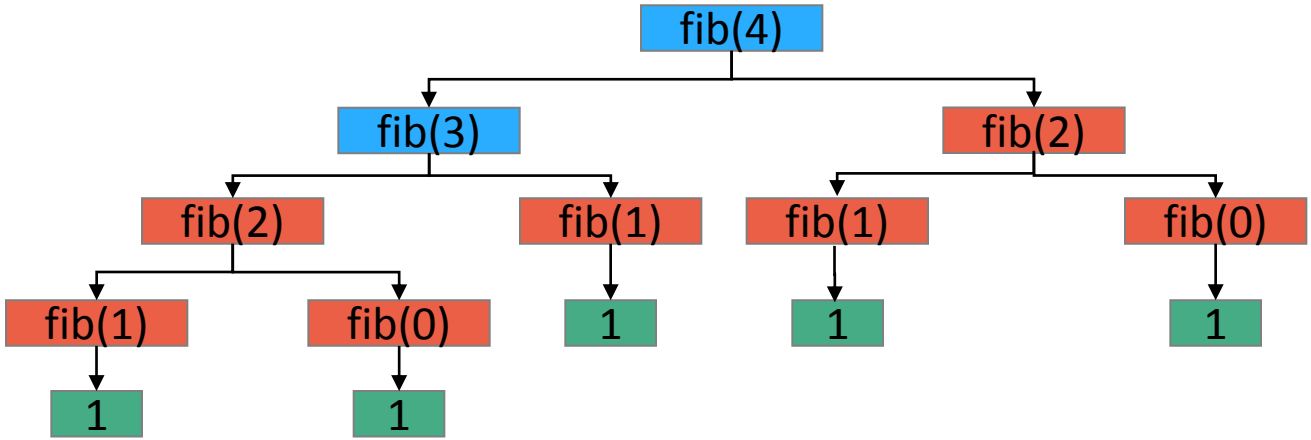
состояние стека при вызове функции B() из A()

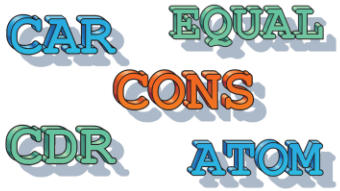


состояние стека при вызове функции A() из A()



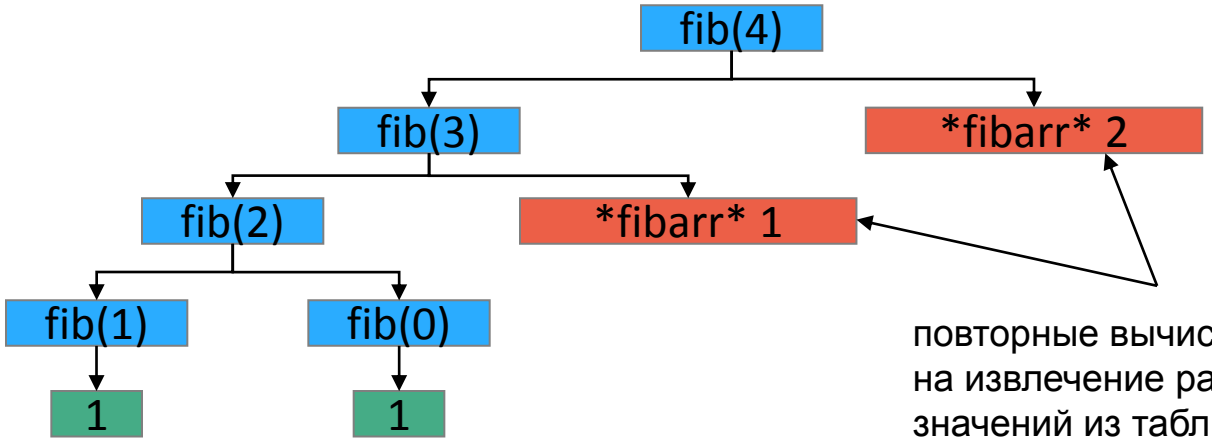
- Мемоизация (memoization) – один из способов оптимизации программ путем сохранения промежуточных результатов для исключения повторных вычислений



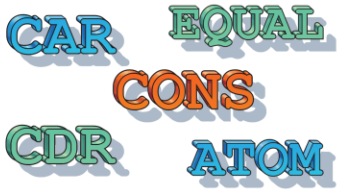


Мемоизация

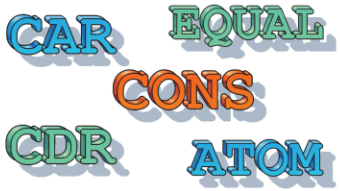
```
[1]> (defparameter *fibarr*
               (make-array 100 :initial-element nil))
*FIBARR*
[2]>  (setf   (aref *fibarr* 0) 1
                        (aref *fibarr* 1) 1)
1
```



повторные вычисления заменяются на извлечение ранее вычисленных значений из таблицы (массива)

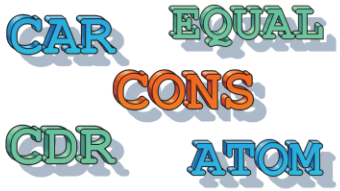



```
(defun fib-memo (n)
    "n-ный член ряда Фибоначчи"
    (if (null (aref *fibarr* n))
        (setf (aref *fibarr* n)
            (+ (fib-memo (- n 1))
                    (fib-memo (- n 2))))
        (aref *fibarr* n)))

[1]> (fib-memo 4)
5
[2]> (fib-memo 44) ; вычисляется быстро
1134903170
```



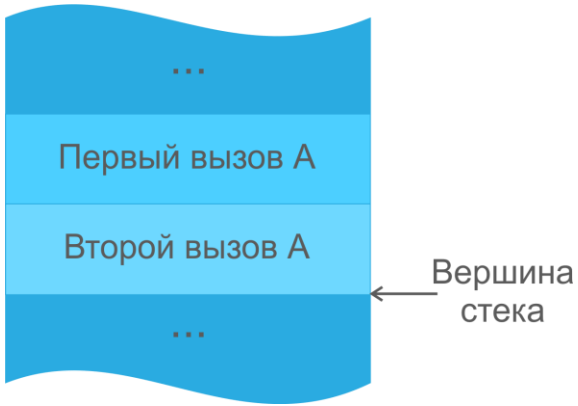
Замена рекурсии на итерацию

```
(defun fib-iter (n)
     "n-ный член ряда Фибоначчи"
     (let ((prev 1) (next 1))
        (dotimes (i (- n 1))
            (shiftf prev next (+ prev next)))
        next))

[1]> (fib-iter 4)
5
[2]> (fib-iter 44)
1134903170
```

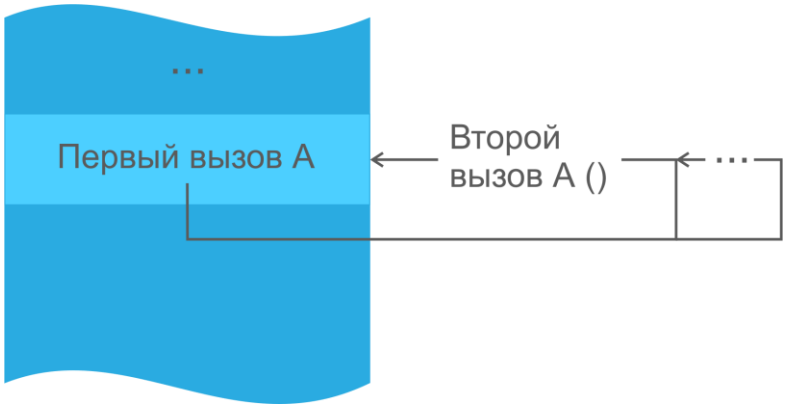


Хвостовая рекурсия

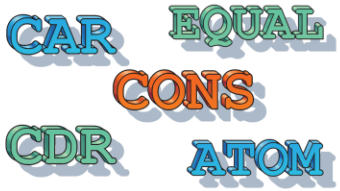
- Хвостовая рекурсия – рекурсия, при которой рекурсивный вызов является последним вычисляемым выражением в функции



состояние стека при обычных рекурсивных вызовах



состояние стека при оптимизации хвостовых рекурсивных вызовов

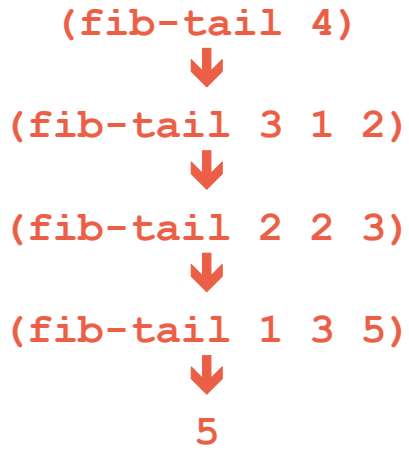


Замена рекурсии на хвостовую рекурсию

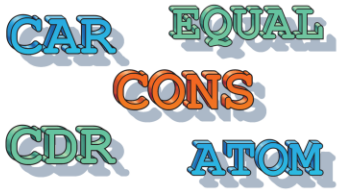
```
(defun fib-tail (n &optional (prev 1) (next 1))
    "n-ный член ряда Фибоначчи"
    (if (< n 2)
        next
        (fib-tail (- n 1) next (+ prev next))))
```

последнее выражение в теле функции – рекурсивный вызов

```
[1]> (fib-tail 4)
5
[2]> (fib-tail 44)
1134903170
```



последовательность рекурсивных вызовов функции fib-tail



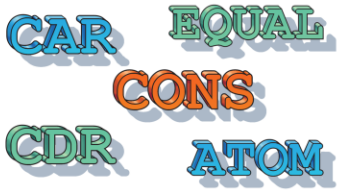
Сравнение производительности функций

CLISP:

```
[1]> (time (fib 35))
Real time: 15.582205 sec.
Run time: 15.357195 sec.
Space: 0 Bytes
14930352
```

SBCL:

```
CL-USER(1): (time (fib 35))
Evaluation took:
  0.279 seconds of real time
  0.278721 seconds of total run time (0.277618 user, 0.001103 system)
  100.00% CPU
  695,891,614 processor cycles
  0 bytes consed
```

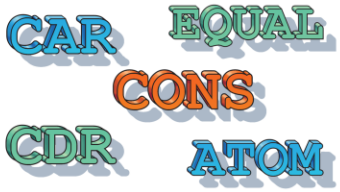


Сравнение производительности функций

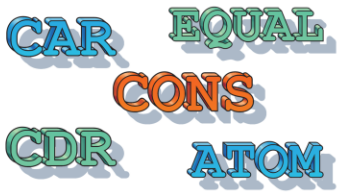
Количество процессорных тактов (из вывода time)

n	fib	fib-memo	fib-iter	fib-tail
10	7,394	3,607/1,537	2,497	2,865
20	581,942	5,844/1,695	3,230	3,565
30	94,214,737	14,020/2,035	3,592	3,425
40	8,508,226,223	18,686/1,219	4,995	3,955
60	–	23,380/1,505	5,965	4,675
80	–	29,465/1,473	6,797	6,629
99	–	32,910/1,679	11,300	14,915

```
Лисп: SBCL 1.3.1
  ЦП: Intel Core i7 2ГГц
```



Рекурсивная обработка данных

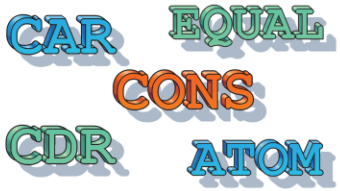


Задача: деление списка на равные части

- Требуется разбить исходный список на списки по n элементов в каждом:

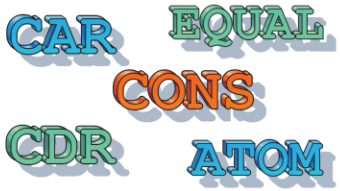
```
(1 2 3 4 5) -> ((1 2) (3 4) (5)), n = 2
```

```
[1]> (split-by '(1 2 3 4 5) 2)
((1 2) (3 4) (5))
```



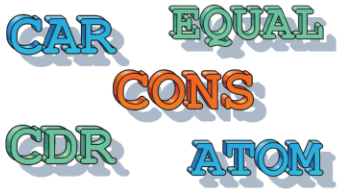
Некоторые полезные функции

```
[1]> (cdddr '(1 2 3 4 5))
(4 5)
[2]> (nthcdr 3 '(1 2 3 4 5))
(4 5)

[3]> (subseq '(1 2 3 4) 0 3)
(1 2 3)
[4]> (subseq '(1 2 3 4) 2 4)
(3 4)
```



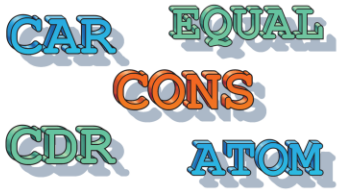
Первый вариант функции split-by

```
(defun split-by (lst n)
    (if (consp lst)
        (cons
            (subseq lst 0 n)
            (split-by (nthcdr n lst) n))
        '() ))

[1]> (split-by '(1 2 3 4) 2)
((1 2) (3 4))
```



Проблемы с первым вариантом функции split-by

```
[1]> (split-by '(1 2 3 4) 3)
 *** - SYSTEM::LIST-ACCESS: NIL is not a pair

[2]> (subseq '(1 2 3) 0 4)
 *** - SYSTEM::LIST-ACCESS: NIL is not a pair

[3]> (split-by '(1 2 3 4) 0)
*** - Program stack overflow. RESET
```

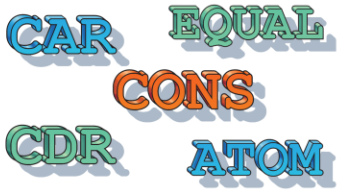


Макрос cond

```
(cond
    (условие1 формы1)
    ...
    (условиеN формыN)
    (t формы-по-умолчанию)
```

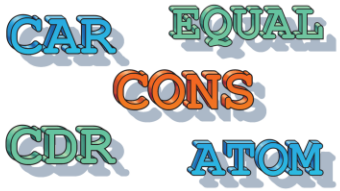
```
(if (> n 0)
    (print "n > 0")
    (print "n <= 0"))
```

```
(cond
    ((> n 0) (print "n > 0"))
    (t (print "n <= 0")))
```



Второй вариант функции split-by

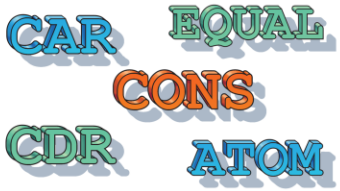
```
(defun split-by (lst n)
     (labels
          ((take (lst n)
               (if (or (= n 0) (null lst))
                    '()
                    (cons (car lst)
                         (take (cdr lst) (- n 1))))))
          (cond
               ((<= n 0) lst)
               ((null lst) '())
               (t (cons (take lst n)
                         (split-by (nthcdr n lst) n))))))

[1]> (split-by '(1 2 3 4) 3)
((1 2 3) (4))
[2]> (split-by '(1 2 3 4) 0)
(1 2 3 4)
```



Задача: сглаживание списка

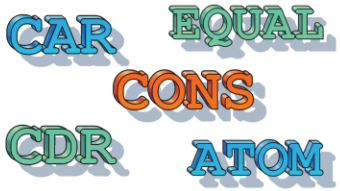
- Требуется устранить из исходного списка все вложенные списки, сохранив элементы, не являющиеся списками, и удалив значения nil и пустые списки:

```
[1]> (flatten '(() 1 (2 (3 ((4)) 5) (6)) (()) 7))
(1 2 3 4 5 6 7)
```



Использование функции append

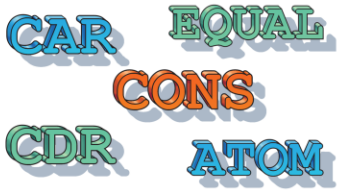
```
[1]> (append '() (1))
(1)
[2]> (append '(1) '())
(1)
[3]> (append '(1) '(2))
(1 2)
[4]> (append '(1 2) 3)
(1 2 . 3)
```



Функция flatten

```
(defun flatten (lst)
        (append '()
            (if (atom (car lst))
                (if (car lst) (list (car lst)))
                (flatten (car lst)))
            (if lst (flatten (cdr lst)))))
```

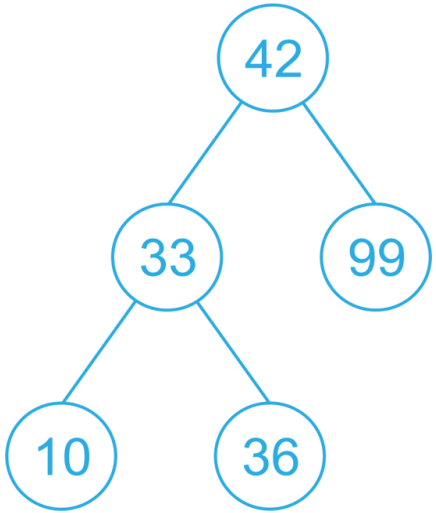
```
[1]> (flatten '(() 1 (2) ((3))))
(1 2 3)
```

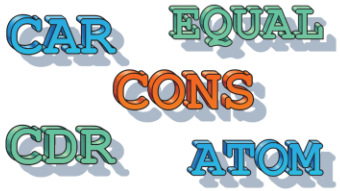


Двоичные деревья поиска

(ключ левое_поддерево правое_поддерево)

```
(42) или (42 nil nil)

(42 (33) (99))

(42 (33 (10) (36)) (99))
```





Задача: обход двоичного дерева

- прямой (preorder)

  `42 33 10 36 99`

- обратный (postorder)

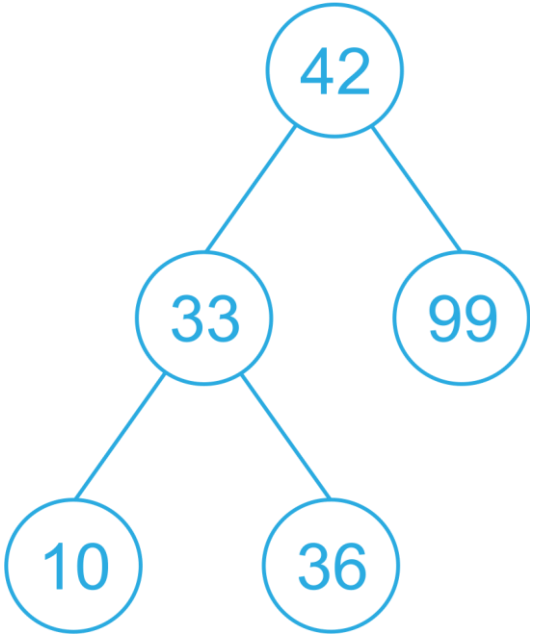
  `10 36 33 99 42`

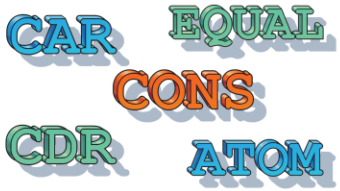
- симметричный (inorder)

  `10 33 36 42 99`

- по уровням (level-order)

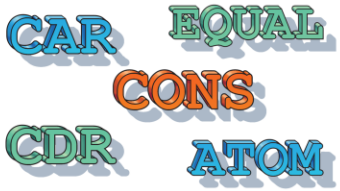
  `42 33 99 10 36`





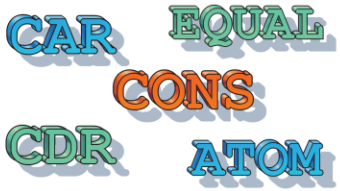
Прямой обход двоичного дерева

```
[1]> (defparameter *tree* '(42 (33 (10) (36)) (99)))
*TREE*

[2]> (bst-preorder *tree*)
42 33 10 36 99
NIL
```



Прямой обход двоичного дерева

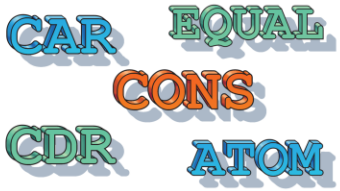
```
(defun left (tree) (second tree))
(defun right (tree) (third tree))


(defun bst-preorder (tree)
    (format t "~a " (car tree))
    (unless (null (left tree))
        (bst-preorder (left tree)))
    (unless (null (right tree))
        (bst-preorder (right tree))))


[1]> (bst-preorder *tree*)
42 33 10 36 99
NIL
```



Обход двоичного дерева по уровням

```
(defun bst-levelorder (tree)
     (labels ((do-bst-levelorder (nodes &aux lst)
          (dolist (node nodes)
               (unless (null node)
                    (format t "~a " (car node))
                    (if (left node)
                         (setf lst
                              (append lst (list (left node)))))
                    (if (right node)
                         (setf lst
                              (append lst (list (right node))))))))
          (if lst (do-bst-levelorder lst))))

          (do-bst-levelorder (list tree))))
```

```
[1]> (bst-levelorder *tree*)
42 33 99 10 36
NIL
```



Что мы узнали из этой лекции

- как передавать необязательные, ключевые и дополнительные аргументы функциям в Лиспе
- что такое время жизни и область видимости
- как использовать формы связывания и чем они отличаются (let, let*, flet, labels)
- как выполнить деконструкцию списка с помощью destructuring-bind
- как вернуть несколько значений из функции
- что такое рекурсия и почему она важна для функционального программирования
- что такое мемоизация
- что такое хвостовая рекурсия