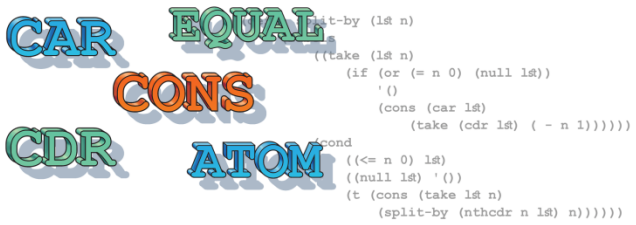
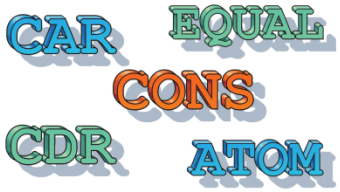


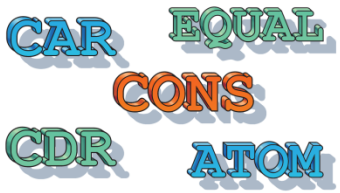
Функциональное программирование: базовый курс

Лекция 6.

Отображение и свертка последовательностей.

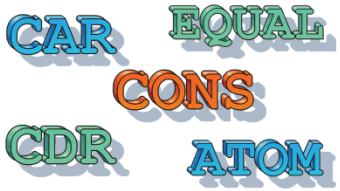


Отображение последовательностей



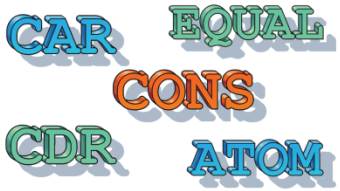
Обработка последовательностей

- отображение
- фильтрация
- поиск
- свертка



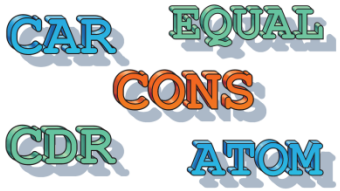
Отображение вместо цикла

```
[1]> (defparameter *nums* '(1 2 3 4 5))
*NUMS*

[2]> (let (acc)
              (dolist (i *nums*)
                     (push (* i i) acc))
              (nreverse acc))
(1 4 9 16 25)
```



Отображение вместо цикла

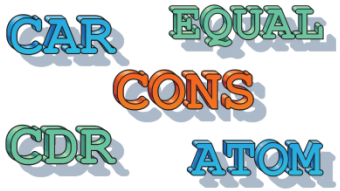
```
(defun simple-map (fn lst)
        (let (acc)
                (dolist (i lst)
                        (push (funcall fn i) acc))
                (nreverse acc)))

[1]> (simple-map #'abs '(-1 3.4 -7.1))
(1 3.4 7.1)

[2]> (simple-map #'(lambda (x) (* x x)) *nums*)
(1 4 9 16 25)

[3]> (simple-map #'(lambda (s) (string-upcase s))
        '("my" "other" "CaR" "is" "cDr"))
("MY" "OTHER" "CAR" "IS" "CDR")
```



Функция map

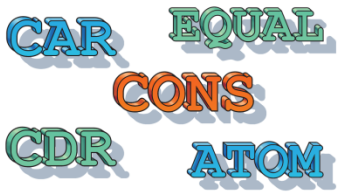
(**map** тип-результата функция последовательности)

```
[1]> (map 'list #'abs '(-1 3.4 -7.1))
(1 3.4 7.1)

[2]> (map 'vector #'(lambda (x) (* x x)) *nums*)
(1 4 9 16 25)

[3]> (map nil #'(lambda (x) (* x x)) *nums*)
NIL
```



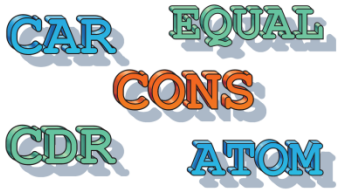
Функция map

```
[1]> (map nil #'(lambda (x) (print (* x x))) *nums*)
1
4
9
16
25
NIL
```



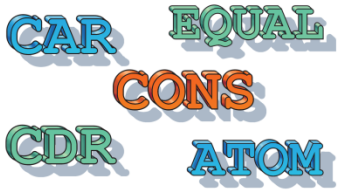
Обработка нескольких последовательностей с помощью map

```
[1]> (defparameter *chars* #(#\a #\b #\c))
*CHARS*

[2]> (map 'list #'(lambda (x y) (cons x y))
                *chars* *nums*)
((#\a . 1) (#\b . 2) (#\c . 3))
```



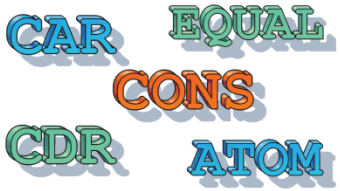
Функция mapcar

```
[1]> (mapcar #'(lambda (x y) (cons x y))
               *chars* *nums*)
*** - MAPCAR: A proper list must not end with #(#\a #\b #\c)

[2]> (setf *chars* '(#\a #\b #\c))
(#\a #\b #\c)

[3]> (mapcar #'(lambda (x y) (cons x y))
               *chars* *nums*)
((#\a . 1) (#\b . 2) (#\c . 3))
```



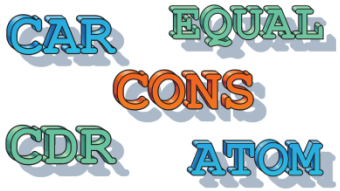
Функция mapc

```
[1]> (mapc #'(lambda (x y) (cons x y))
             *chars* *nums*)
(#\a #\b #\c)

[2]> (mapc #'(lambda (x y) (print (cons x y)))
             *chars* *nums*)
(#\a . 1)
(#\b . 2)
(#\c . 3)
(#\a #\b #\c)
```



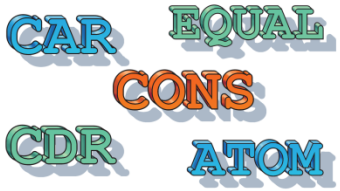
Функция maplist

```
[1]> (maplist
        #'(lambda (x) (print x)) *nums*)
(1 2 3 4 5)
(2 3 4 5)
(3 4 5)
(4 5)
(5)
((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))
```



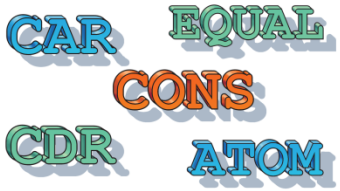
Функция maplist

```
[1]> (maplist
        #'(lambda (x) (print x) (apply #'+ x)) *nums*)
(1 2 3 4 5)
(2 3 4 5)
(3 4 5)
(4 5)
(5)
(15 14 12 9 5)
```



Функция mapl

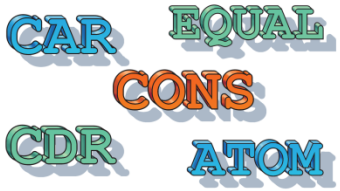
```
[1]> (mapl
       #'(lambda (x) (print x) (apply #'+ x)) *nums*)
(1 2 3 4 5)
(2 3 4 5)
(3 4 5)
(4 5)
(5)
(1 2 3 4 5)
```



Количество элементов в результирующей последовательности

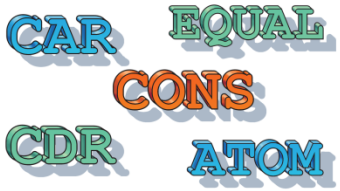
- Если количество элементов в результирующей последовательности должно отличаться от количества элементов в исходных последовательностях, то следует использовать функции mapcan и mapcon

```
(1 2 0 3 0 4 0 0 5)  ──────►  (1 2 3 4 5)

(1 2 3 4)
(a b c d)            ──────►  (1 a 2 b 3 c 4 d)
```



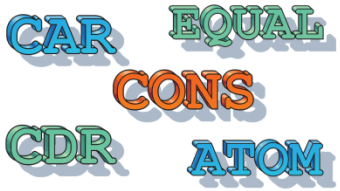
Функция mapcan

```
[1]> (mapcar
        #'(lambda (x y) (list x y)) '(1 2 3) '(4 5 6))
((1 4) (2 5) (3 6))

[2]> (mapcan
        #'(lambda (x y) (list x y)) '(1 2 3) '(4 5 6))
(1 4 2 5 3 6)
```



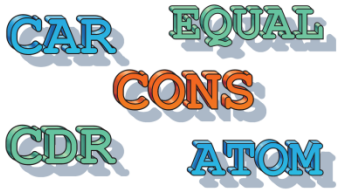
Функция mapcan

```
[1]> (mapcar
       #'(lambda (x) (if (evenp x) (list x) nil)
                 '(1 2 3 4 5 6))
(NIL (2) NIL (4) NIL (6))

[2]> (mapcan
       #'(lambda (x) (if (evenp x) (list x) nil)
                 '(1 2 3 4 5 6))
(2 4 6)
```



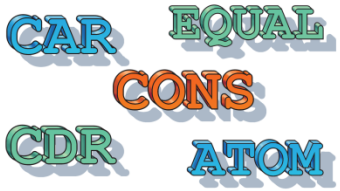
Функция mapcan

```
[1]> (mapcar
       #'(lambda (x) (if (evenp x) (list x) nil)
                 '(1 2 3 4 5 6))
(NIL (2) NIL (4) NIL (6))

[2]> (apply #'nconc (mapcar
       #'(lambda (x) (if (evenp x) (list x) nil)
                 '(1 2 3 4 5 6)))
(2 4 6)
```



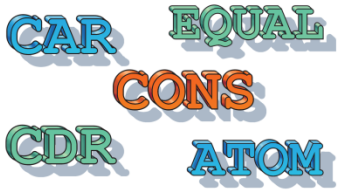
Сглаживание списка

```
(defun flatten-redux (lst)
  (cond ((null lst) nil)
        ((atom lst) (list lst))
        (t (mapcan #'flatten-redux lst))))

[1]> (flatten-redux '(() 1 (2) ((3 4))))
(1 2 3 4)
```

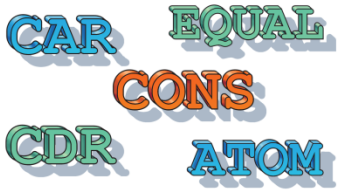


Функция map-into

```
[1]> (defparameter *a* '(0 0 0))
*A*

[2]> (map-into *a* #'+ '(1 2 3) '(20 40 50))
(21 42 53)
```

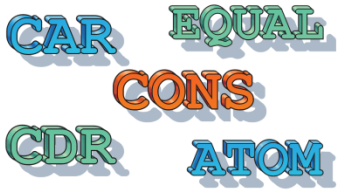


Фильтрация и свертка последовательностей



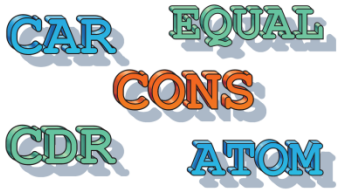
Фильтрация последовательностей

```
[1]> (mapcan #'(lambda (x)
                       (if (oddp x) (list x) nil))
             '(1 2 3 4 5))
(1 3 5)
```



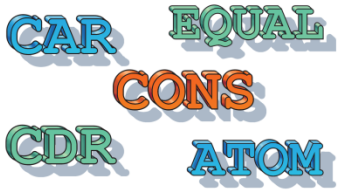
Функция remove

```
[1]> (remove 4 '(1 2 3 4 5 6 7) :test #'<)
(1 2 3 4)

[2]> (remove 4 '(1 2 3 4 5 6 7) :test #'< :count 2)
(1 2 3 4 7)

[3]> (remove "lisp"
             '("Python" "Lisp" "C++" "Ruby")
             :test-not #'string-equal)
("Lisp")
```



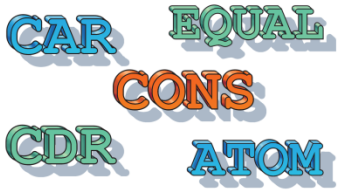
Функция remove

```
[1]> (mapcan #'(lambda (x)
                       (if (oddp x) (list x) nil))
             '(1 2 3 4 5))
(1 3 5)

[2]> (remove nil '(1 2 3 4 5)
             :test-not #'(lambda (x y) (oddp y)))
(1 3 5)
```



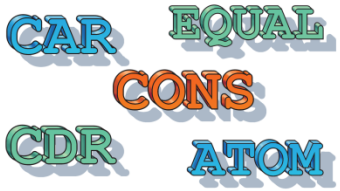
Функции remove-if и remove-if-not

```
[1]> (mapcan #'(lambda (x)
                       (if (oddp x) (list x) nil))
             '(1 2 3 4 5))
(1 3 5)

[2]> (remove-if #'(lambda (x) (not (oddp x)))
             '(1 2 3 4 5))
(1 3 5)

[3]> (remove-if-not #'oddp '(1 2 3 4 5))
(1 3 5)
```



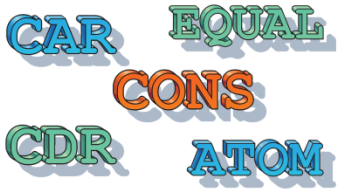
Функции remove-if и remove-if-not

```
[1]> (remove-if #'(lambda (x) (> x 4)
                  '(1 2 3 4 5 6 7))
(1 2 3 4)

[2]> (remove-if #'(lambda (x) (> x 4)
                  '(1 2 3 4 5 6 7) :count 2)
(1 2 3 4 7)

[3]> (remove-if-not #'(lambda (x) (> x 4)
                  '(1 2 3 4 5 6 7))
(5 6 7)
```



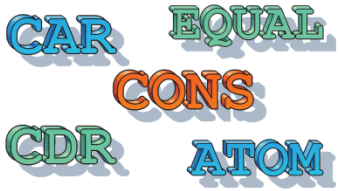
Семейство функций substitute

```
[1]> (substitute 42 24 '(24 0 24))
(42 0 42)

[2]> (substitute-if 42 #'oddp '(1 2 3 4 5))
(42 2 42 4 42)

[3]> (substitute-if-not 42 #'oddp '(1 2 3 4 5))
(1 42 3 42 5)
```

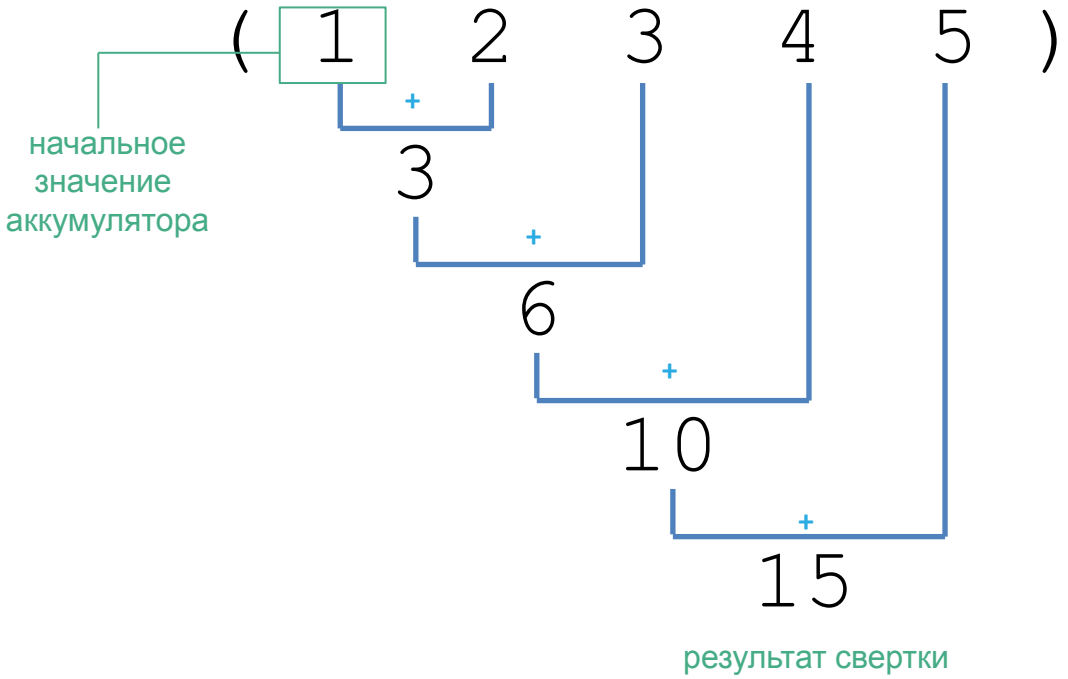


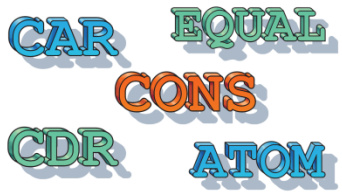
Функции поиска

Функция	Назначение
find, find-if, find-if-not	Поиск первого элемента, отвечающего заданному критерию
position, position-if, position-if-not	Поиск индекса первого элемента, отвечающего заданному критерию
member, member-if, member-if-not	Проверка наличия элемена, отвечающего заданному критерию
assoc, assoc-if, assoc-if-not	Поиск элемента по ключу в ассоциативном списке
rassoc, rassoc-if, rassoc-if-not	Поиск элемента по значению в ассоциативном списке
count, count-if, count-if-not	Подсчет количества элементов, отвечающих заданному критерию
search	Поиск одной последовательности внутри другой
mismatch	Поиск индекса первого несовпадающего элмента двух последовательностей



Свертка последовательностей





Свертка последовательностей

начальное значение аккумулятора

(1 2 3 4 5)

\- \- \- \-

3 6 10 15

левая свертка

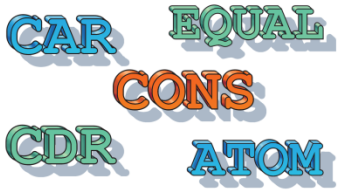
начальное значение аккумулятора

(1 2 3 4 5)

\- \- \- \-

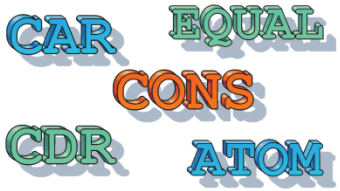
-1 4 -2 3

правая свертка



Функция reduce

```
[1]> (reduce #'+ '(1 2 3 4 5))
15

[2]> (reduce #'- '(1 2 3 4 5))
-13

[3]> (reduce #'- '(1 2 3 4 5) :from-end t)
3
```

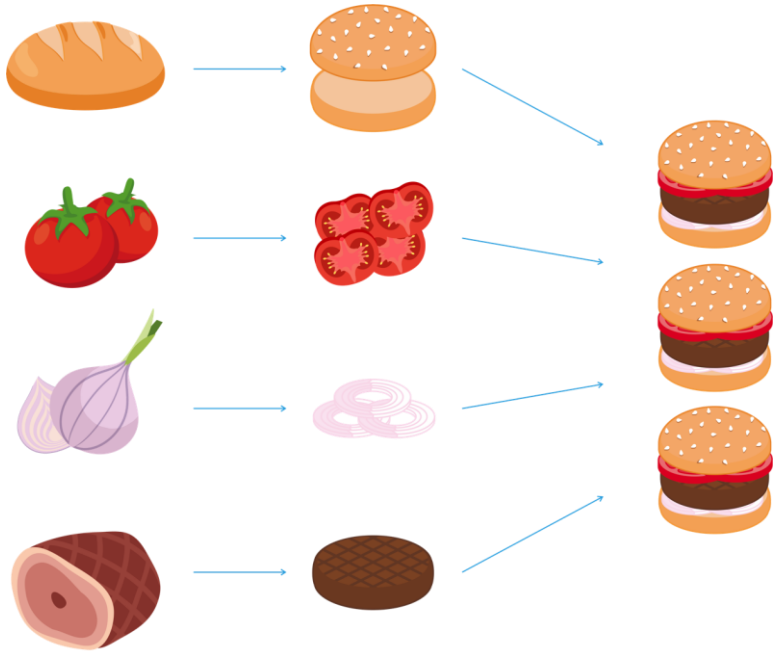


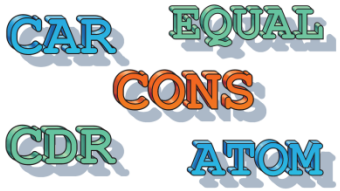
Реализация map с помощью reduce

```
(defun map-on-reduce (fn lst)
        (nreverse
                (reduce #'(lambda (acc x)
                                  (push (funcall fn x) acc))
                        lst
                        :initial-value nil)))

[1]> (map-on-reduce #'(lambda (x) (1+ x)) '(1 2 3 4))
(2 3 4 5)
```

CAR EQUAL
CONS
CDR ATOM

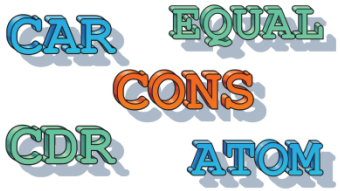
MapReduce





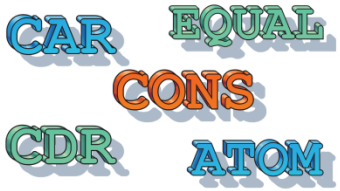
Задача: подсчет количества простых чисел в списке

```
(defparameter *max-random-num* 100)

(defun random-list (n)
       (let (test-list)
              (dotimes (i n)
                     (push
                            (1+ (random *max-random-num*))
                            test-list))
              test-list))

[1]> (random-list 10)
(82 77 29 76 34 33 6 3 78 8)
```



Задача: подсчет количества простых чисел в списке

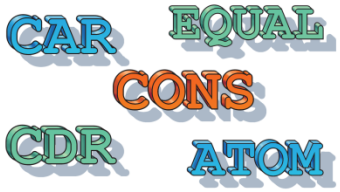
- Способы получения простых чисел
  - решето Эратосфена
  - решето Сундарама
  - решето Аткина
- Проверки простоты числа
  - перебор делителей
  - **тест Вильсона**
  - тест Миллера
  - тест Агравала-Каяла-Саксены
  - тесты для специальных чисел (тест Пепина, тест Люка, ...)



Тест Вильсона

Если $n - 1 = (n - 1)! \bmod n$, то n – простое.

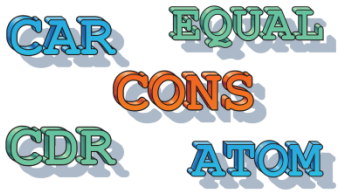
n	$(n - 1)! \bmod n$
2	1
3	2
4	2
5	4
6	0
7	6
…	…



Факториал

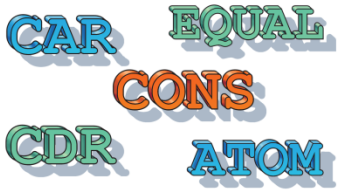
```
(defun factr (n &optional (res 1))
       (if (or (= n 0) (= n 1))
              res
              (factr (- n 1) (* res n))))
```

```
[1]> (factr 3)
6
[2]> (factr 300)
306057512216440636035370461297268629388588804173576999416776741259476533176716867465515291422477573499391478887017263688642639077590031542268429279069745598412254769302719546040080122157762521768542559653569035067 8...
```



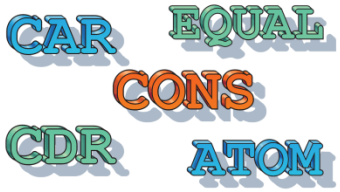
Проверка на простоту с помощью теста Вильсона

```
(defun primep (x)
       (= (- x 1) (mod (factr (- x 1) 1) x)))

[1]> (primep 3)
T

[2]> (primep 300)
NIL
```

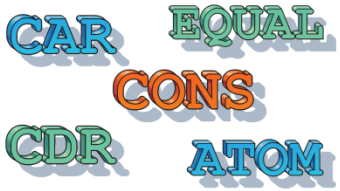


Задача: подсчет количества простых чисел в списке

```
[1]> (defparameter lst (random-list 10))
LST

[2]> lst
(92 39 77 70 3 16 59 77 9 44)

[3]> (reduce #'+
        (mapcar #'(lambda (x) (if x 1 0))
                (mapcar #'primep lst)))
2

[4]> (remove-if-not #'primep lst)
(3 59)
```



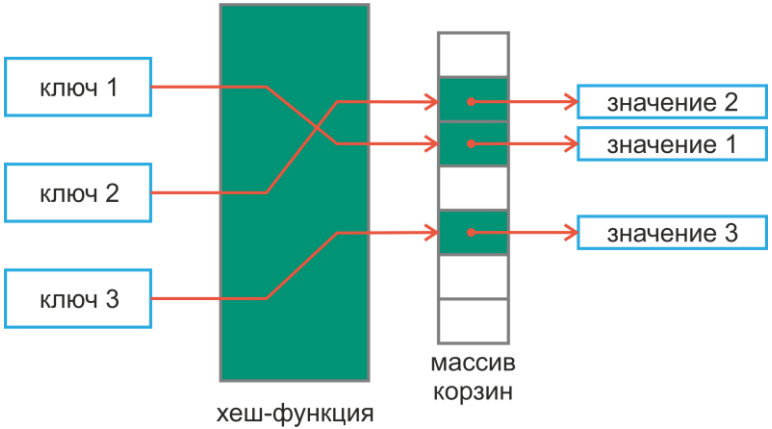
Работа с хеш-таблицами

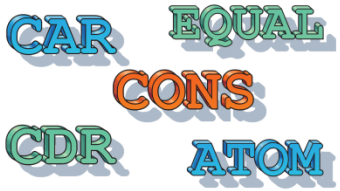


Хеш-таблицы

Хеш-таблица (hash table) – структура данных для хранения пар "ключ – значение"

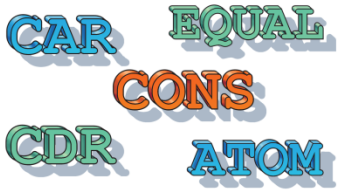
- поиск, добавление и удаление элементов происходит в среднем за время O(1)





Хеш-таблицы и ассоциативные списки

```
[1]> (defparameter *alist*
       '((one . 1) (two . 2) (four . 4)))

[2]> (assoc 'two *alist*)
(TWO . 2)

[3]> (assoc-if
              #'(lambda (x) (not (eq x 'two)))
       *alist*)
(ONE . 1)
```



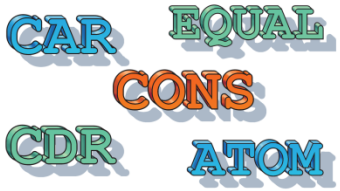
Создание хеш-таблицы

В CLISP:

```
[1]> (defparameter *ht* (make-hash-table))
*HT*
[2]> *HT*
#S(HASH-TABLE :TEST FASTHASH-EQL)
```

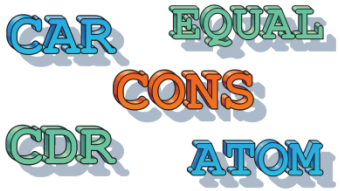
В SBCL:

```
CL-USER(1): (defparameter *ht* (make-hash-table))
*HT*
CL-USER(2): *HT*
#<HASH-TABLE :TEST EQL :COUNT 0 {1005DCE043}>
```



Создание хеш-таблицы

```
[1]> (defparameter *ht* (make-hash-table :test #'equal))
*HT*
[2]> *HT*
#S(HASH-TABLE :TEST FASTHASH-EQUAL)
[3]> (eql "test" "test")
NIL
[4]> (equal "test" "test")
T
[5]> (eqlual "test" "Test")
NIL
[6]> (equalp "test" "Test")
T
```

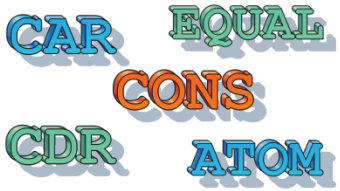


Создание хеш-таблицы

```
[1]> (defparameter *ht* (make-hash-table :test #'string=))
*** - MAKE-HASH-TABLE: Illegal :TEST argument
```

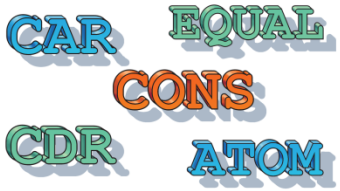
ТОЛЬКО

```
eq
eql
equal
equalp
```



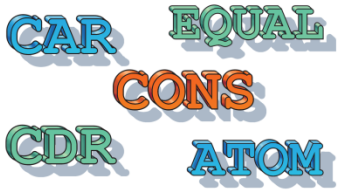
Поиск по ключу в хеш-таблицах

```
[1]> (defparameter *ht* (make-hash-table :test #'equal))
*HT*

[2]> (gethash "test" *ht*)
NIL ;
NIL

[3]> (gethash "test" *ht* 42)
42 ;
NIL
```



Добавление или обновление пары ключ-значение в хеш-таблице

```
[1]> (defparameter *ht* (make-hash-table :test #'equal))
*HT*

[2]> (setf (gethash "test" *ht*) 42)
42

[3]> (gethash "test" *ht*)
42 ;
T
```



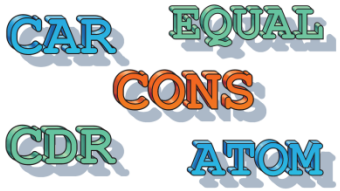
Добавление или обновление пары ключ-значение в хеш-таблице

```
[1]> (defparameter *ht* (make-hash-table :test #'equal))
*HT*

[2]> (setf (gethash "NIL but T" *ht*) nil)
NIL

[3]> (gethash "NIL but T" *ht*)
NIL ;
T
```

← такой ключ есть, и его значение – NIL

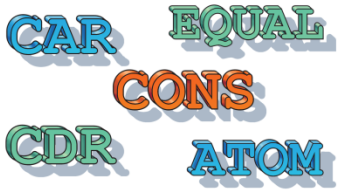
```
[4]> (gethash "just NIL" *ht*)
NIL ;
NIL
```

← такого ключа нет



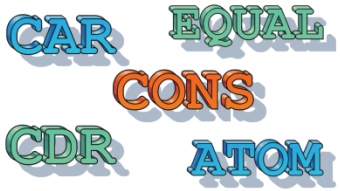
Добавление или обновление пары ключ-значение в хеш-таблице

```
[1]> (defparameter *ht* (make-hash-table :test #'equal))
*HT*

[2]> (setf (gethash "test" *ht*) 42)
NIL

[3]> (remhash "test" *ht*)
T

[4]> (gethash "test" *ht*)
NIL ;
NIL
```



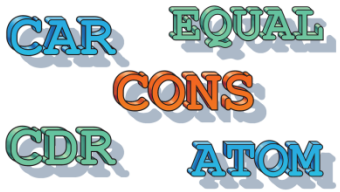
Инициализация хеш-таблицы

```
[1]> (setf *h* (hash-table-from-list
                        '(      "one"           1
                                "two"           2
                                "three"         3
                                "four"  4)
                        :test #'equal))

*H*
```



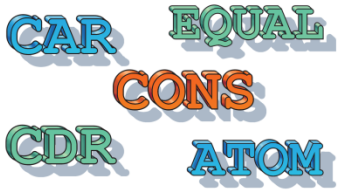
Функция hash-table-from-list

```
(defun hash-table-from-list
                (lst &rest keyword-pairs
                        &key (start 0) end &allow-other-keys)
        (let ((h (apply
                        #'make-hash-table
                        :allow-other-keys t keyword-pairs))
                (ls (subseq lst start end)))
                ; вспомогательная функция, которая
                ; добавляет в хэш h пары ключ-значение
                ; из списка ls
                (hash-table-add-list h ls)))
```



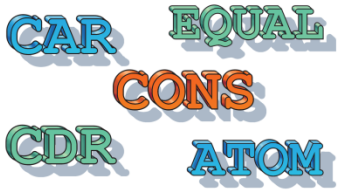
Функция hash-table-add-list

```
(defun hash-table-add-list (h lst &aux k v)
       (loop
              (if (null lst) (return))
              (setf k (pop lst))
              (if (null lst) (return))
              (setf v (pop lst))
              (setf (gethash k h) v))
       h)
```



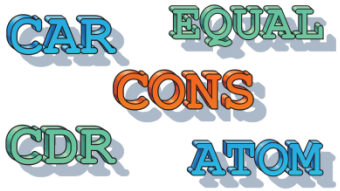
Функция set%

```
(defun set% (h key val &rest other-pairs)
        (let ((pairs (append (list key val) other-pairs)) k v)
                (hash-table-add-list h pairs)))

[1]> (defparameter *h* (hash-table-from-list
                '("oliver" "twist" "david" "copperfield")
                :test #'equal))
*H*

[2]> (set% *h* "david" "bowie" "oliver" "stone")
#S(HASH-TABLE :TEST FASTHASH-EQUAL
        ("david" . "bowie") ("oliver" . "stone"))
```



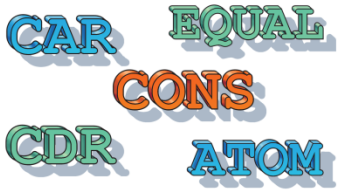
Функция %

```
(defun % (h key &rest other-keys)
        (let ((keys (append (list key) other-keys)))
                (if (> (length keys) 1)
                        (mapcar #'(lambda (x)
                                        (gethash x h)) keys)
                        (gethash (car keys) h))))

[1]> (% *h* "david" "oliver")
("bowie" "stone")
```



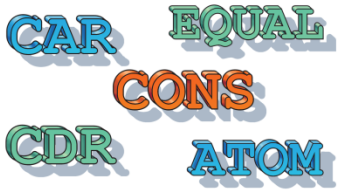
Обработка хеша в цикле

```
(defun hash-iterate (fn h)
        (with-hash-table-iterator (get-e h)
                (dotimes (i (hash-table-count h))
                        (multiple-value-call fn (get-e)))))

[1]> (hash-iterate #'(lambda (has-more k v)
                                (format t "~a ~a~%" k v)) *h*)

david bowie
oliver stone
NIL
```



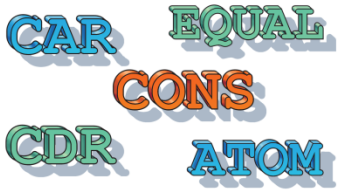
Отображение хеш-таблицы с помощью функции maphash

```
(defun hash-table-key-value-pairs (h)
        (let (pairs)
                (maphash #'(lambda (k v)
                                (push (list k v) pairs)) h)
                (nreverse pairs)))

[1]> (hash-table-key-value-pairs *h*)
(("david" "bowie") ("oliver" "stone"))
```



Отображение хеш-таблицы с помощью функции maphash

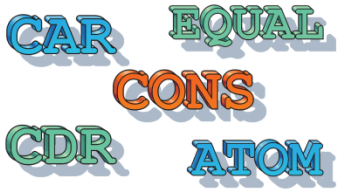
```
(defun real-maphash (fn h)
        ; возвращает новую хеш-таблицу,
        ; каждое значение в которой модифицировано
        ; функцией fn
        )
```



Лекция 6

Отображение и свертка последовательностей

Использование отображений и сверток на практике



Задача: анализ списка регистрационных номеров

Дан список номеров транспортных средств, нарушивших скоростной режим:

```
(defparameter *vrps* '(
        "0245ок;43" "с227на;69"
        "с304вв;70" "у333ух;71"
        "ек201;69" ...
        ))
```

Необходимо выяснить, водители-частники из каких регионов совершили наибольшее количество нарушений?

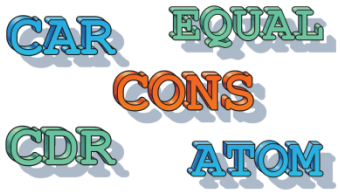
CAR EQUAL CONS CDR ATOM

Задача: анализ списка регистрационных номеров

"с227на;69"

регистрационные знаки
легковых такси
и транспортных средств
для перевозки более 8 человек

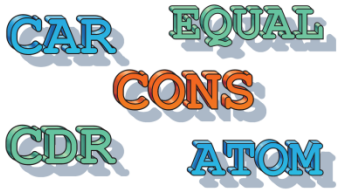
регистрационные знаки
транспортных средств
МВД России



Задача: анализ списка регистрационных номеров

План решения задачи:

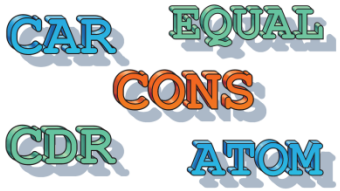
1. Оставить в списке только номера частных транспортных средств.
2. Преобразовать список номеров в список регионов.
3. Подсчитать количество нарушителей из каждого региона.
4. Расположить список регионов в порядке убывания количества нарушителей.



Задача: анализ списка регистрационных номеров

План решения задачи:

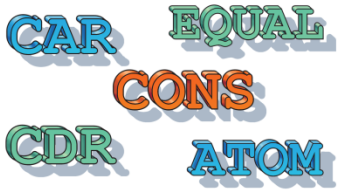
1. Оставить в списке только номера частных транспортных средств.
   - remove-if
2. Преобразовать список номеров в список регионов.
   - mapcar
3. Подсчитать количество нарушителей из каждого региона.
   - reduce
4. Расположить список регионов в порядке убывания количества нарушителей.
   - sort



Проверка типа регистрационного номерного знака

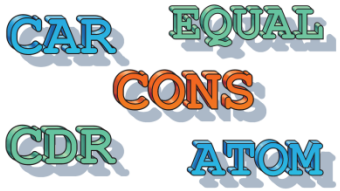
- в серии могут встречаться буквы кириллицы
  - А, В, Е, К, М, Н, О, Р, С, Т, У, Х


```
(defun vrp-letter-p (c)
       (and (find c "авекмнорстух"
                    :test #'char-equal) t))
```



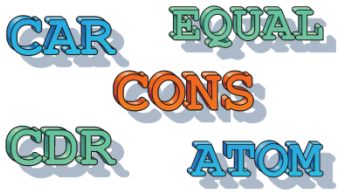
Проверка типа регистрационного номерного знака

```
(defun private-vrp-p (v)
       (and (>= (length v) 9)
            (char= #\; (elt v 6))
            (vrp-letter-p (elt v 0))
            (digit-char-p (elt v 1))
            (digit-char-p (elt v 2))
            (digit-char-p (elt v 3))
            (vrp-letter-p (elt v 4))
            (vrp-letter-p (elt v 5))))
```



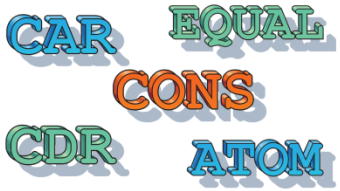
Задача: анализ списка регистрационных номеров

```
(defun get-region-list (lst)
       (remove-if
              (complement #'private-vrp-p)
              lst))
```



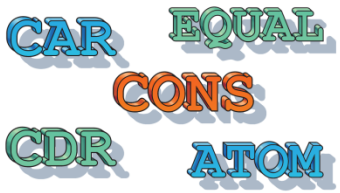
Задача: анализ списка регистрационных номеров

```
(defun region-from-vrp (v)
        (subseq v (1+ (position #\; v))))

(defun get-region-list (lst)
        (mapcar
                #'region-from-vrp
                (remove-if
                        (complement #'private-vrp-p)
                        lst)))
```



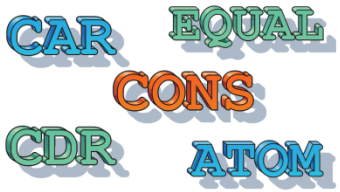
Свертка списка с кодами регионов

```
(defun get-region-list (lst)
  (reduce
    #'(lambda (h e) (hash++ h e) h)
    (mapcar
      #'region-from-vrp
      (remove-if
        (complement #'private-vrp-p)
        lst))
    :initial-value (make-hash-table :test #'equal)))
```



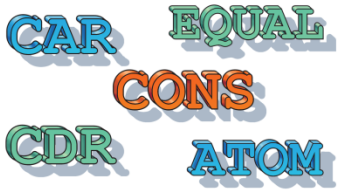
Инкремент значения в хеш-таблице

```
(defun hash++ (h key &optional (n 1))
       (let ((val (gethash key h)))
              (cond
                     ((or (null val) (not (numberp
val)))
                            (set% h key n))
                     (t (set% h key (+ val n))))
              (gethash key h)))
```



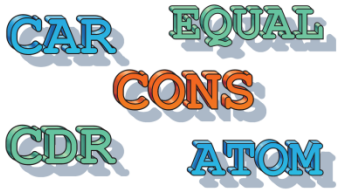
Получение списка пар из хеш-таблицы

```
(defun get-region-list (lst)
  (hash-table-key-value-pairs
    (reduce
      #'(lambda (h e) (hash++ h e) h)
      (mapcar
        #'region-from-vrp
        (remove-if
          (complement #'private-vrp-p)
          lst))
      :initial-value (make-hash-table :test #'equal))))
```



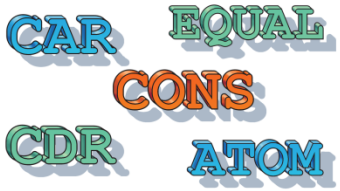
Функция sort

```
[1]> (defparameter lst '(20 4 6 8 0 3 1 7))
LST

[2]> (sort lst #'>)
(20 8 7 6 4 3 1 0)

[3]> (sort lst #'<)
(0 1 3 4 6 7 8 20)
```



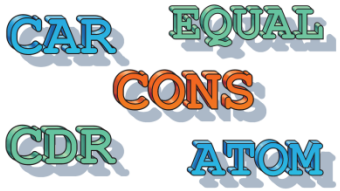
Функция sort

```
[1]> (defparameter lst
                '(("177" 14) ("78" 30) ("98" 10)))
LST

[2]> (sort lst
                #'(lambda (x y) (> (second x) (second y))))
(("78" 30) ("177" 14) ("98" 10))

[3]> (sort lst #'> :key #'second)
(("78" 30) ("177" 14) ("98" 10))
```



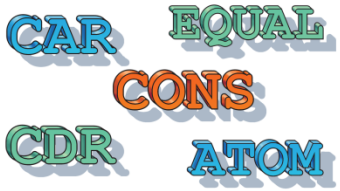
Получение отсортированного списка пар

```
(defun get-region-list (lst)
  (sort
    (hash-table-key-value-pairs
      (reduce
        #'(lambda (h e) (hash++ h e) h)
        (mapcar
          #'region-from-vrp
          (remove-if
            (complement #'private-vrp-p)
            lst))
        :initial-value (make-hash-table :test #'equal)))
    #'> :key #'second))
```



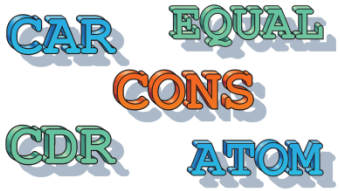
Задача: анализ списка регистрационных номеров

```
[1]> *vrps*
("ao365;78" "0245ок;43" "c227на;69" "a1234;78" "ек201;69"
"c304вв;70" "y333ух;71" "o001оо;78" "ху045;78" "a144нс;78"
"a144нс;78" "e4433;98" "002cd1;178" "a144нс;78" "e042кх;777"
"e043кх;777" "3340но;150" "e044кх;777")

[2]> (get-region-list *vrps*)
(("78" 4) ("777" 3) ("69" 1) ("70" 1) ("71" 1))
```



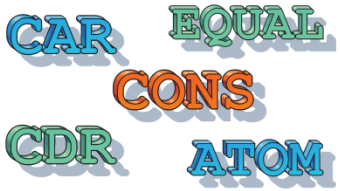
Задача: анализ списка регистрационных номеров

```
(defun get-region-list (lst)
  (sort
    (hash-table-key-value-pairs
      (reduce
        #'(lambda (h e) (hash++ h (region-from-vrp e)) h)
        (remove-duplicates
          (remove-if
            (complement #'private-vrp-p) lst)
          :test #'string-equal)
        :initial-value (make-hash-table :test #'equal)))
    #'> :key #'second))
```



Задача: анализ списка регистрационных номеров

```
[1]> *vrps*
("ao365;78" "0245ок;43" "c227на;69" "a1234;78" "ек201;69"
"c304вв;70" "y333ух;71" "o001оо;78" "ху045;78" "a144нс;78"
"a144нс;78" "e4433;98" "002cd1;178" "a144нс;78" "e042кх;777"
"e043кх;777" "3340но;150" "e044кх;777")

[2]> (get-region-list *vrps*)
(("777" 3) ("78" 2) ("69" 1) ("70" 1) ("71" 1))
```



Что мы узнали из этой лекции

- что такое отображение последовательностей
- какие функции используются для отображения: map, mapcar, mapc, maplist, mapcan, mapcon
- что такое свертка последовательностей
- чем отличается левая свертка от правой
- как работать с хеш-таблицами
- как применять отображения, фильтрации и свертки при решении практических задач