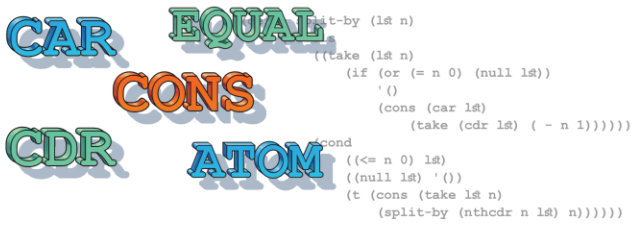
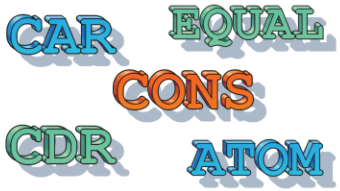


Функциональное программирование: базовый курс

Лекция 2.

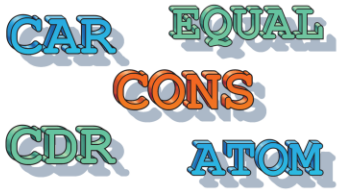
Основы языка Lisp.



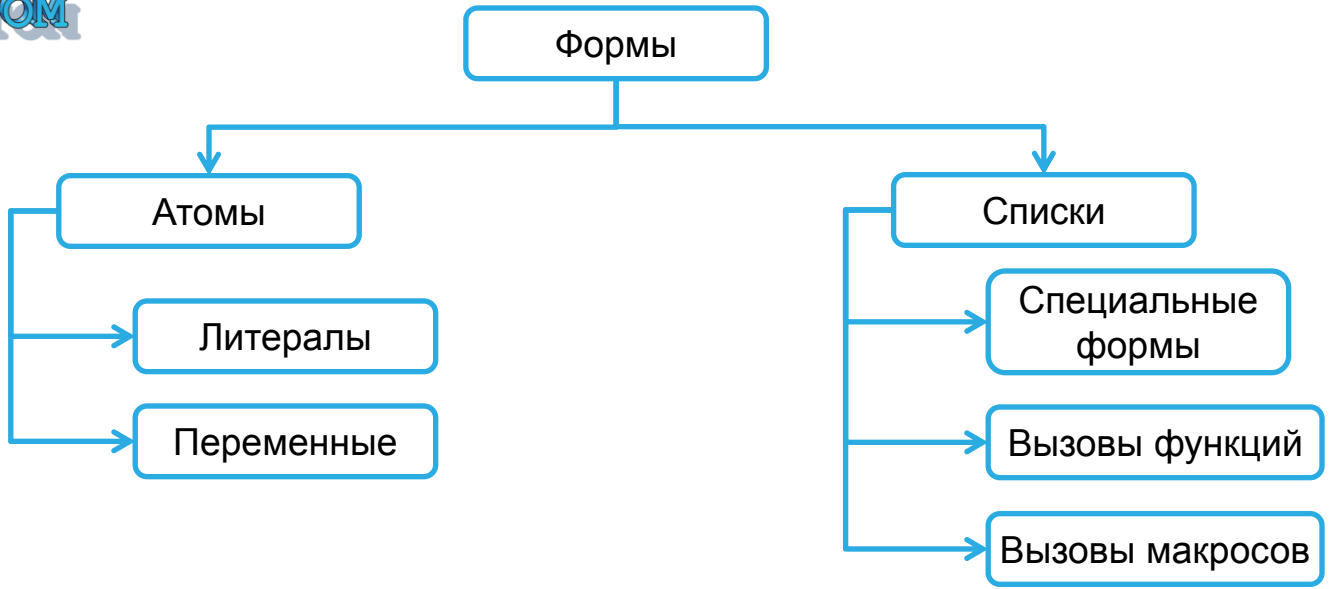
Лекция 2

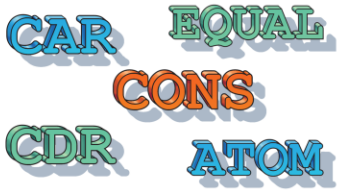
Основы языка Lisp

Формы и функции


CAR
EQUAL
CONS
CDR
ATOM

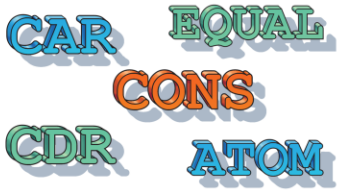
Формы


Формы
Атомы
Литералы
Переменные
Списки
Специальные формы
Вызовы функций
Вызовы макросов



Вызов функции

```
[1]> (random 100)
73
[2]> (+ 111 222)
333
[3]> (format t "~r" (+ 111 222))
three hundred and thirty-three
NIL
[4]> (machine-type)
"X86_64"
```

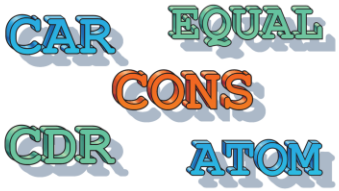


Определение функции

- Для того, чтобы создать новую функцию, используется форма **defun**:

```
(defun имя-функции (список-параметров)
; тело функции
)
```

```
(defun print-n-randoms (n max-limit)
    (dotimes (i n)
        (format t "~r~%" (random max-limit))))
```



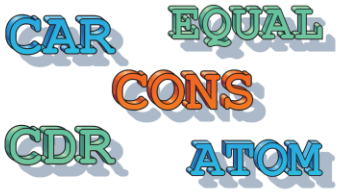
Оформление кода

- Правильно:

```
(defun print-n-randoms (n max-limit)
    (dotimes (i n)
        (format t "~r~%" (random max-limit))))
```

- Неправильно:

```
(defun print_n_randoms (n maxLimit)
    (dotimes (i n)
        (format t "~r~%" (random maxLimit)
        )
    )
)
```



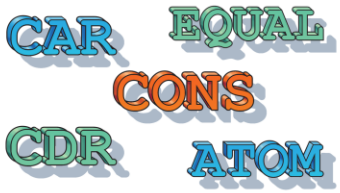
Комментарии

- Однострочные комментарии

```
;;;; комментарий в начале файла
;;; комментарий на верхнем уровне (перед функцией)
;; комментарий в теле функции
; комментарий в конце строки
```

- Многострочные комментарии

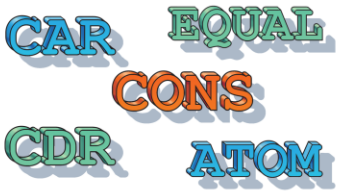
```
#|
     это
     комментарий
     на нескольких
     строках
|#
```



Вызов функции

```
[1]> (print-n-randoms 2 100)
thirty-nine
seventy
NIL
[2]> (print-n-randoms 4 1000)
two hundred and four
sixty-nine
one hundred and fifty-one
eight hundred and forty-five
NIL
[3]> (print-n-randoms 0 1000)
NIL
[4]> (print-n-randoms -1 1000)
NIL
```

```
[5]> (print-n-randoms 1 -1000)
*** - RANDOM: argument should be positive
and an integer or float, not -1000
The following restarts are available:
ABORT          :R1      Abort main loop
Break 1[6]> :r1
[7]> (print-n-randoms 1 10.5)
*** - SVREF: 0.08213514 is not a correct
index into
The following restarts are available:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort main loop
```



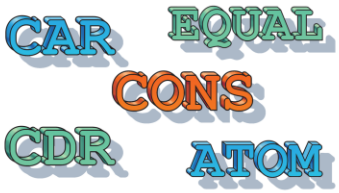
Как получить справку по функциям

- Справка в интерпретаторе:

```
[1]> (describe 'defun)
DEFUN is the symbol DEFUN, lies in #<PACKAGE COMMON-LISP>, is accessible in 8 packages
CLOS, COMMON-LISP, COMMON-LISP-USER, EXT, POSIX, REGEXP, SCREEN, SYSTEM,
names a macro, has 1 property SYSTEM::DOC.

[2]> (describe 'print-n-randoms)
PRINT-N-RANDOMS is the symbol PRINT-N-RANDOMS, lies in #<PACKAGE COMMON-LISP-USER>
...
Argument list: (N MAX-LIMIT)
```

- Справка в Интернете (Common Lisp HyperSpec):

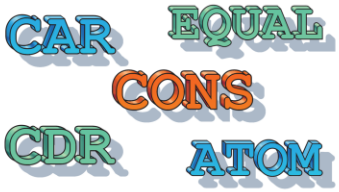
http://clhs.lisp.se/Front/StartPts.htm



Кратко о макросах

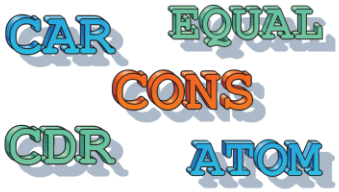
- Макросы позволяют манипулировать текстом программы и определять новые синтаксические конструкции

```
[1]> (macro-function 'format)
NIL
[2]> (macro-function 'defun)
#<COMPILED-FUNCTION DEFUN>
```



Специальные формы

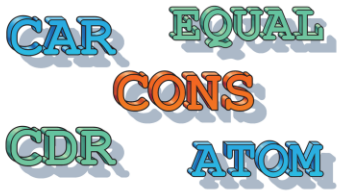
```
block
catch
eval-when
flet
function
go
if
labels
let
let*
load-time-value
locally
macrolet
multiple-value-call
multiple-value-prog1
progn
progv
quote
return-from
setq
symbol-macrolet
tagbody
the
throw
unwind-protect
```



Условный выбор – форма if

```
(if условие форма1 форма2)
```

```
(if (= 4 (* 2 2))
    (print "Don't panic")
    (print "Panic!"))

"Don't panic"
```



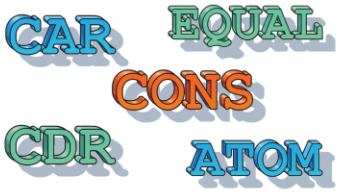
Форма if и условные операторы в других языках

- Си:

```
if (n > 100)
    too_large_a_number();
// ветка else отсутствует
```

- Лисп:

```
(if (> n 100)
    (too-large-a-number))
;; вторая форма отсутствует
```



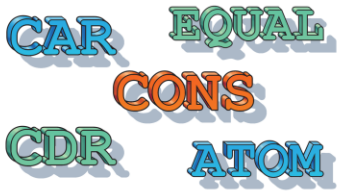
Форма if и условные операторы в других языках

- Си:

```
if (n > 100) {
    too_large_a_number();
    do_something_crazy();
}
```

- Лисп:

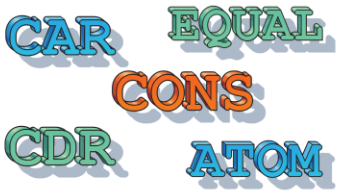
```
(if (> n 100)
    ((too-large-a-number)
     (do-something-crazy)))    ; ошибка!
```



Последовательное выполнение – форма progn

```
(progn форма1 форма2 … формаN)
```

```
(if (> n 100)
    (progn (too-large-a-number)
                (do-something-crazy)))   ; OK
```

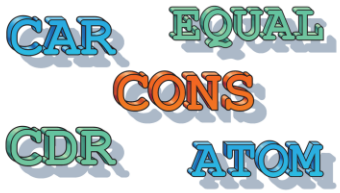


Условные формы when и unless

(**when** условие форма1 форма2 … формаN)

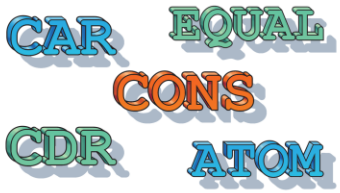
(**unless** условие форма1 форма2 … формаN)

```
(when (> n 10)
    (too-large-a-number)
    (do-something-crazy))
```



Условные формы when и unless

```
[1]> (macroexpand  '(when (> n 10)
            (too-large-a-number)
            (do-something-crazy)))
(IF (> N 2) (PROGN (too-large-a-number) (do-something-crazy))) ;
T

[2]> (macroexpand  '(unless (> n 10)
            (too-large-a-number)
            (do-something-crazy)))
(IF (NOT (> N 2)) (PROGN (too-large-a-number) (do-something-crazy))) ;
T
```



Форма if и условные операторы в других языках

В Си невозможно использовать оператор if как выражение:

```
my_func(m,
    if (n > 100) n else 100);
```

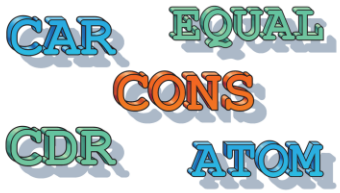
синтаксическая ошибка

Но можно использовать условную операцию:

```
my_func(m, n > 100 ? n : 100);
```

В Лиспе форму if можно передавать в другие формы:

```
(my-func m
    (if (> n 100) n 100))

(setf (if (> n 100)
    *some-global-var*
    *other-global-var*) 42)
```



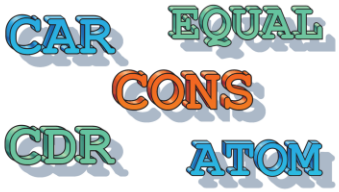
Переменные и символы

Для представления переменных в программах на Лиспе используется специальный тип данных – символ (symbol)

symbol ≠ character

```
A..Z  a..z  0..9  +  -  *  /  @
$  %  ^  &  _  =  <  >  ~  .
```

Правильные имена	Правильные имена, но так переменные лучше не называть	Неправильные имена
`i`	`<<<__-_-__>>>`	`1101`
`my-var`	`b^2-4*a*c`	`#www#`
`variable-with-a-long-name`	`192.168.1.1`	`wrong|name`
`*dynamic*`	`|?!(){}|`	`bad()idea`
	`сомнительная_переменная`	



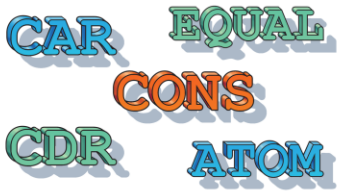
Имена символов

`lisp` `Lisp` `LISP` `LiSp`

один и тот же символ, регистр букв не имеет значения

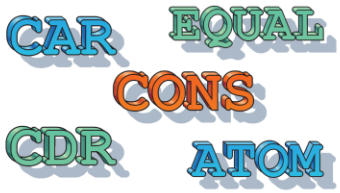
```
[1]> (defvar *answer* 42)
*ANSWER*
[2]> *AnSwEr*
42
[3]> (setf *Answer* 0)
0
[4]> *ANSWER*
0
```

earmuffs



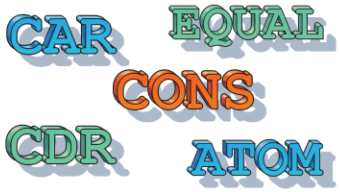
Присваивание с помощью setf

```
[1]> (setf x 42)
42
[2]> (type-of x)
(INTEGER 0 281474976710655)
[3]> (setf x "Dum spiro spero")
"Dum spiro spero"
[4]> (type-of x)
(SIMPLE-BASE-STRING 15)
[5]> (setf x pi)
3.141592653589793238L0
[6]> (type-of x)
LONG-FLOAT
[7]> (setf pi 3.14 e 2.71 g 9.81)
9.81
```



Литералы

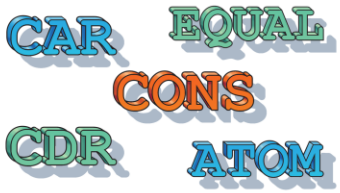
```
[1]> 42
42
[2]> 4.2e1
42.0
[3]> 1/3
1/3
[4]> #b1111
15
[5]> #\a
#\a
[6]> #\≠
#\NOT_EQUAL_TO
```



Лекция 2

Основы языка Lisp

Система типов Common Lisp



Система типов Common Lisp

Числа
(Numbers)

Символы
(Characters)

Функции
(Functions)

Списки
(Lists)

Массивы
(Arrays)

Хеш-таблицы
(Hash tables)

Символы
(Symbols)

Пакеты
(Packages)

Структуры
(Structures)

Потоки
(Streams)

Пути
(Pathnames)

Условия
(Conditions)

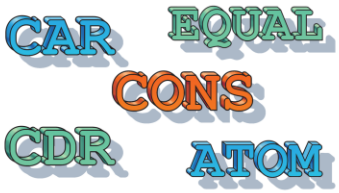
Классы
(Classes)

Методы
(Methods)

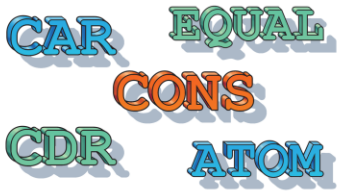
Обобщенные функции
(Generic functions)

Таблицы чтения
(Readtables)

Состояния ГПСЧ
(Random-states)



- Целые (integer)
- Рациональные (rational)
- Вещественные (floating-point)
- Комплексные (complex)

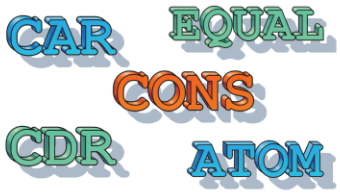


Целые числа

```
[1]> (* 12345678912345678 98765432198765432)
1219326313565005313472031691126352 69
```

#bRddd...d

```
[2]> #2R101010      ; 42 в двоичной системе
42
[3]> #16R2A         ; в шестнадцатеричной системе
42
[4]> #25R1H         ; в двацатипятиричной системе
42
```



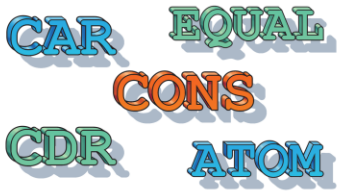
Числа в разных системах счисления

dec	bin	oct	hex
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

$$(1101\ 1110\ 1010\ 1101\ 1011\ 1110\ 1110\ 1111)_2$$

D E A D B E E F

$$(\mathrm{DEADBEEF})_{16}$$



Целые числа

```
[1]> #b10100101
165
```

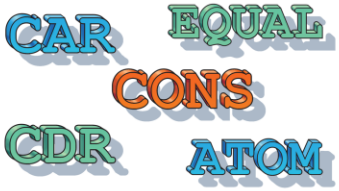
#b или #B = #2R

```
[2]> #o10
8
```

#o или #O = #8R

```
[3]> #x20
32
```

#x или #X = #16R

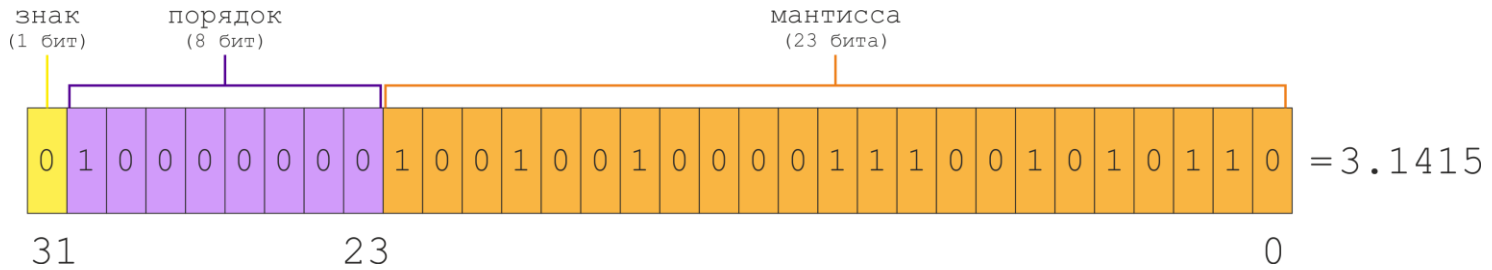


Дроби

```
[1]> (+ 3/5 1/7)
26/35

[2]> (- #xA/AA #xA/FF)
1/51
```

Вещественные числа

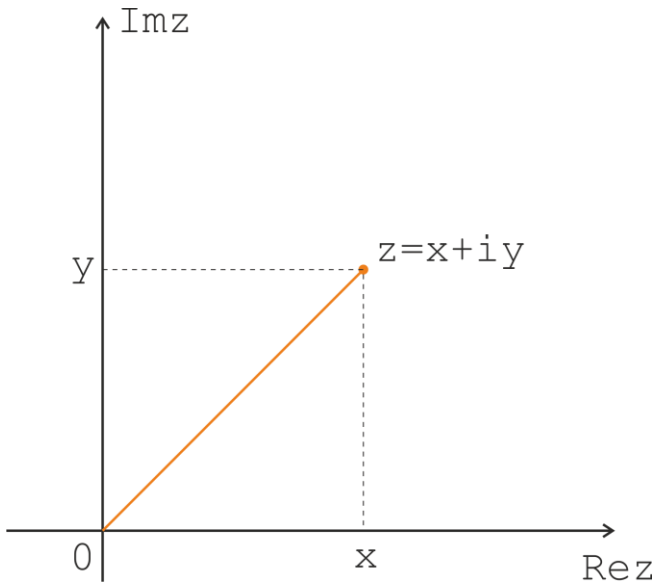
```
[1]> 3.14
3.14
[2]> pi
3.141592653589793238L0
[3]> 0.031415e2
3.1415
```

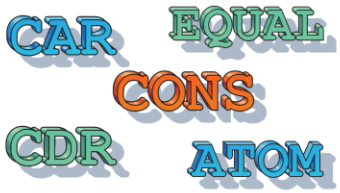


формат IEEE 754-2008, single precision

Комплексные числа

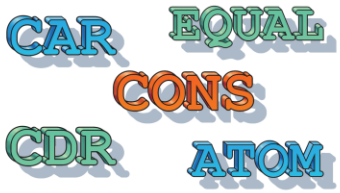
```
[1]> #C(100 -100)
#C(100 -100)
[2]> (* #C(5 -1) #C(2 -2))
#C(8 -12)
[3]> #c(3.14 0)
3.14
[4]> #c(3.14 0.0)
#C(3.14 0.0)
[5]> (expt #c(0 1) 2) ; i² = -1
-1
```





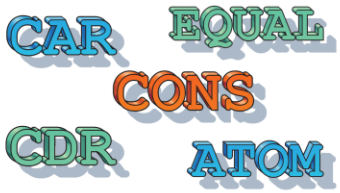
Символы (characters)

```
[1]> #\a
#\a
[2]> (format t "~c" #\a)
a
NIL
[3]> (format t "~c" #\NOT_EQUAL_TO)
≠
NIL
```



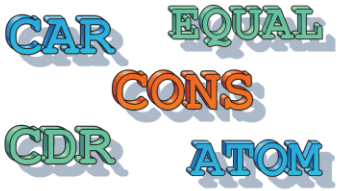
Полезные функции для работы с символами

```
[1]>  (char-upcase #\a)
#\A
[2]> (char-downcase #\A)
#\a
[3]> (char-invertcase \#A)
#\a
[4]> (char-code #\a)
97
[5]> (code-char 97)
#\a
[6]> (code-char (- 97 32))
#\A
```



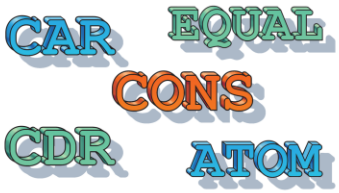
Полезные функции для работы с символами

```
[1]> (char-name #\A)
"LATIN_CAPITAL_LETTER_A"
[2]> (char-name #\ë)
"CYRILLIC_SMALL_LETTER_IO"
[3]> (name-char "CYRILLIC_SMALL_LETTER_IO")
#\CYRILLIC_SMALL_LETTER_IO
[4]> (format t "~c"
           (char-upcase
               (name-char "CYRILLIC_SMALL_LETTER_IO")))
Ë
NIL
[5]> (char-code \#Ë)
1025
```

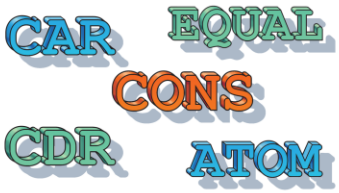


- LISP = LISt Processing

```
[1]> '(1 2 3)
(1 2 3)
[2]> (quote (1 2 3))
(1 2 3)
[3]> '(1 two "three" (() () () ()))
(1 TWO "three" (NIL NIL NIL NIL))
[4]> ()
NIL
```

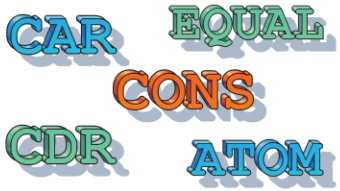


- Векторы (vectors)
- Многомерные массивы (multi-dimentional arrays)
- Строки (strings)
- Битовые вектора (bit-vectors)



Хеш-таблицы

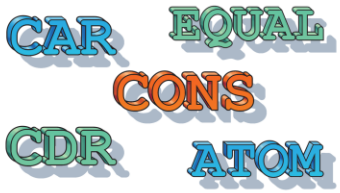
```
[1]> (setf h (make-hash-table))
#S(HASH-TABLE :TEST FASTHASH-EQL)
[2]> (setf (gethash 'name h) 'carl)
CARL
[3]> (gethash 'name h)
CARL;
T
```



Лекция 2

Основы языка Lisp

Предикаты и элементарные логические функции



Истина и ложь в Лиспе

Истинное значение – t

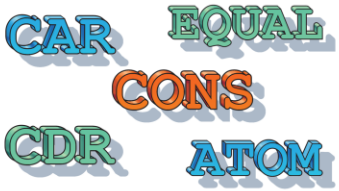
```
[1]> (> 20 10)
T
```

t = 't

Ложное значение – nil

```
[2]> (> 10 20)
NIL
```

nil = 'nil = () = '()



Истина и ложь в Лиспе

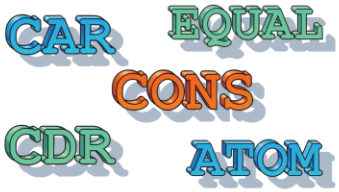
- Си:

```
if (0)
    printf("true");
else
    printf ("false");
```

выводит на экран `false`

- Лисп:

```
[1]> (if 0 'true 'false)
TRUE
```

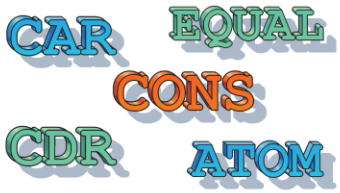


Логические функции

```
[1]> (not nil)
T
[2]> (and t nil)
NIL
[3]> (or t nil)
T
[4]> (xor t t)
NIL
```

x	NOT x
0	1
1	0

x	y	x AND y	x OR y	x XOR y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



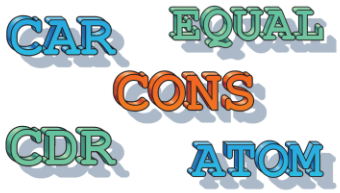
Оптимизация вычислений логических выражений (short-circuiting)

```
[1]> (defvar n 11)
N
[2]> (and (> n 10) (< n 20))
T
[3]> (or (> n 10) (< n 20))
T
```

истинное выражение, следующие формы можно не вычислять

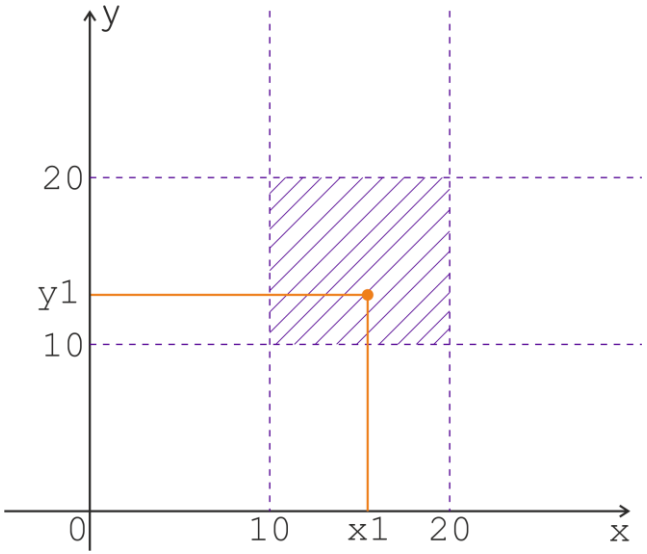
```
[4]> (setf n 9)
9
[5]> (and (> n 10) (< n 20))
NIL
```

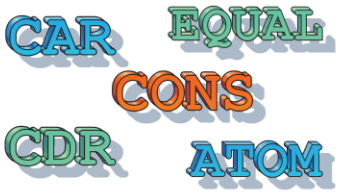
ложное выражение, следующие формы можно не вычислять



Количество операндов в логических функциях

```
[1]> (setf x1 15 y1 12)
12
[2]> (and
            (> x1 10)
            (< x1 20)
            (> y1 10)
            (< y1 20))
T
[3]> (and (> x1 10))
T
[4]> (and)
T
```

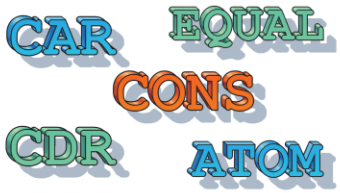




Предикаты

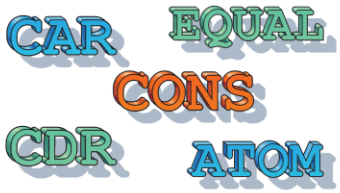
Предикат – функция, которая проверяет некоторое условие и возвращает логическое значение (nil или отличное от nil значение)

```
[1]> (zerop 10)
NIL
[2]> (zerop 0)
T
[3]> (oddp 7)
T
[4]> (string-lessp "abc" "abd")
2
```



Проверка билета на «счастливость»


```
(defun happy-ticket-p (num)
   (and (>= num 0) (<= num 999999)
      (= сумма_младших_трех_цифр
         сумма_старших_трех_цифр)))
```



Как получить цифру десятичного числа?

Пусть дано целое неотрицательное число d, состоящее из m десятичных цифр:

$$d_{m-1}...d_2d_1d_0$$

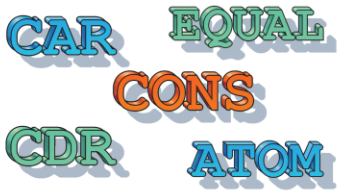
Необходимо по номеру n найти n-ную цифру числа.

$$1234$$

$$1234 \;/\; 100 = 1234 \;/\; 10^2 = 12.34$$

$$[12.34] = 12$$

$$12 \bmod 10 = 2$$



Некоторые полезные функции

- нахождение модуля числа

```
[1]> (abs -1234)
1234
```

- возведение в степень

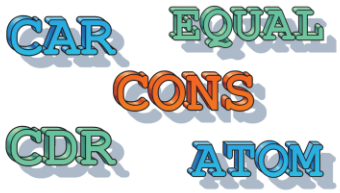
```
[2]> (expt 10 2)
100
```

- отбрасывание дробной части

```
[3]> (truncate 12.34)
12
0.34
```

- нахождение остатка от деления

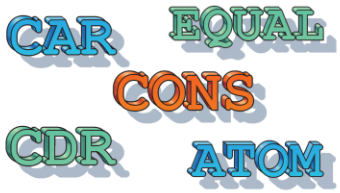
```
[4]> (rem 12 10)
2
```



Как получить цифру десятичного числа?

```
(defun nth-dec-digit (num n)
    (rem
        (truncate
            (/ (abs num) (expt 10 n)))
        10))
```

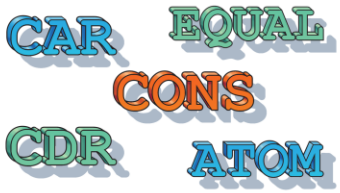
```
[1]> (nth-dec-digit 1234 1)
3
[2]> (nth-dec-digit 1234 0)
4
```



Проверка билета на «счастливость»

```
(defun happy-ticket-p (num)
     (and (>= num 0) (<= num 999999)
          (=   (+   (nth-dec-digit num 0)
                    (nth-dec-digit num 1)
                    (nth-dec-digit num 2))
               (+   (nth-dec-digit num 3)
                    (nth-dec-digit num 4)
                    (nth-dec-digit num 5)))))
```

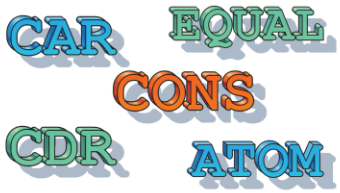
```
[1]> (happy-ticket-p 268736)
T
[2]> (happy-ticket-p 123456)
NIL
[3]> (happy-ticket-p 123)        ; 123 => 000123
NIL
```



Функции проверки на равенство: eq

- Для проверки на равенство символов (symbols) используется функция eq:

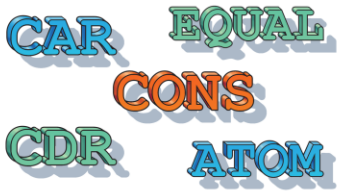
```
[1]> (eq t nil)
NIL
[2]> (eq nil ())
T
[3]> (eq 10 10)
T
[4]> (eq 10 10.0)
NIL
[5]> (eq #\a #\a)
T
[6]> (eq "string" "string")
NIL
```



Функции проверки на равенство: eql

- Для проверки на равенство чисел одного типа, символов (characters) и символов (symbols) можно использовать eql:

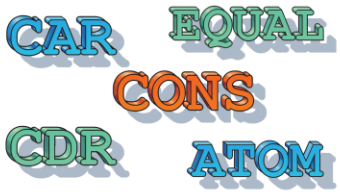
```
[1]> (eql nil 'nil)
T
[2]> (eql 10 10)
T
[3]> (eql 10 10.0)
NIL
[3]> (eql #\a #\a)
T
[4]> (eql "string" "string")
NIL
```



Функции проверки на равенство: equal

- Для проверки на равенство атомов, списков, строк и битовых векторов можно использовать equal:

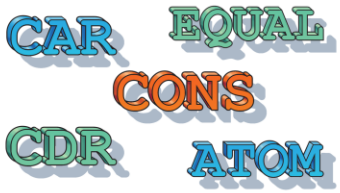
```
[1]> (equal '(1 2 3) '(1 2 3))
T
;; битовые вектора
[2]> (equal #4*1101 #4*1100)
T
[3]> (equal "string" "string")
T
```



Функции проверки на равенство: =

- Для проверки на равенство чисел используется функция =

```
[1]> (= 10 10.0)
T
[2]> (= 10 #C(10.0 0.0))
T
[3]> (= 10 100/10)
T
[4]> (= #xFF 255)
T
```



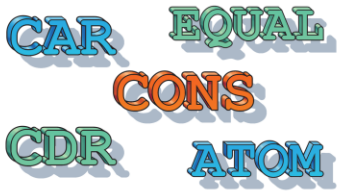
Функции проверки на равенство: char= и char-equal

- Для проверки на равенство символов (characters) без учета регистра используется функция char-equal

```
[1]> (char-equal #\a #\A)
T
[2]> (char-equal #\a #\b)
NIL
```

- Для проверки на равенство символов (characters) с учетом регистра используется функция char=

```
[3]> (char= #\a #\A)
NIL
[4]> (char= #\a #\a)
T
```



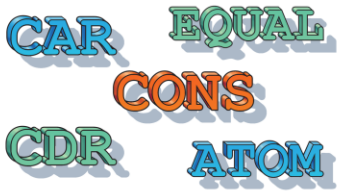
Функции проверки на равенство: string= и string-equal

- Для проверки на равенство строк без учета регистра используется функция string-equal

```
[1]> (string-equal "Lisp rocks!" "LiSp RoCkS!")
T
[2]> (string-equal "flip" "flop")
NIL
```

- Для проверки на равенство строк с учетом регистра используется функция string

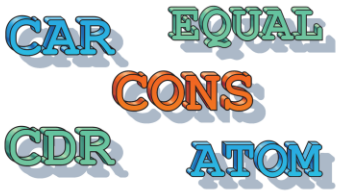
```
[3]> (string= "Lisp rocks!" "LiSp RoCkS!")
NIL
[4]> (string= "flop" "flop")
T
```



Функции проверки на равенство: equalp

- equalp – наиболее общая функция проверки на равенство, которая подходит и для простых типов, и для составных:

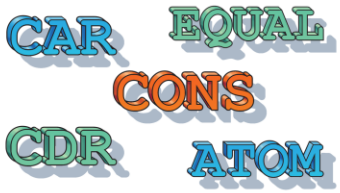
```
[1]> (equalp 10 10.0)
T
[2]> (equalp #\a #\A)
T
[3]> (equalp "lisp" "LISP")
T
[4]> (equalp '(1 2 3) '(1 2 3))
T
[5]> (equalp 10 "10")
NIL
```



Проверка на неравенство

```
[1]> (= 10 20)
NIL
[2]> (/= 10 20)
T
[3]> (not (= 10 20))
T
[4]> (eq 'aa 'bb)
NIL
[5]> (not (eq 'aa 'bb))
T
```

предикат	парный предикат
eq	–
eql	–
equal	–
=	/=
char=	char/=
char-equal	char-not-equal
string=	string/=
string-equal	string-not-equal
equalp	–



Выбор функции проверки на равенство

Нужно сравнивать **только** символы (symbols)? — да → **eq**

нет ↓

Нужно сравнивать **только** числа? — да → **=**

нет ↓

Нужно сравнивать **только** строки или символы (chars)? — нет → Нужно сравнивать **только** простые типы или списки? — да → **equal**

да ↓

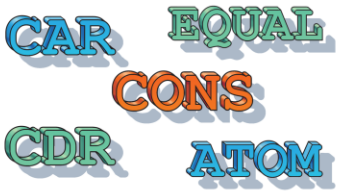
Нужно учитывать регистр? — да → **char=** или **string=**

нет ↓

Нужно сравнивать **только** простые типы или списки? — нет ↓ **equalp**

нет ↓

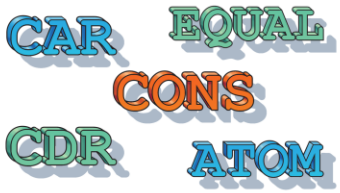
char-equal или **string-equal**



Стандартные предикаты: функции сравнения

```
[1]> (char-code #\c)
99
[2]> (char-code #\C)
67
[3]> (string>= "abc" "abC")
2
```

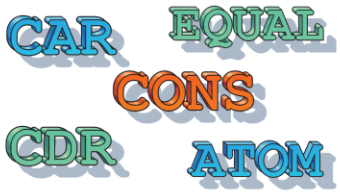
сравнение чисел	сравнение символов (chars)	сравнение строк
>	char> char-greaterp	string> string-greaterp
<	char< char-lessp	string< string-lessp
>=	char>= char-not-lessp	string>= string-not-lessp
<=	char<= char-not-greaterp	string<= string-not-greaterp



Стандартные предикаты: функции сравнения

```
[1]> (= 42 42 42)
T
[2]> (< 0 1 2 3)          ; 0 < 1 < 2 < 3
T
[3]> (< 1 2 3 0)          ; 1 < 2 < 3 ≮ 0
NIL
```

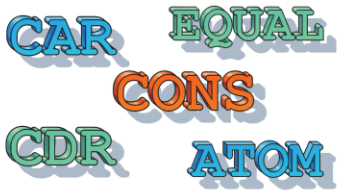
```
(defun happy-ticket-p (num)
     (and (>= num 0) (<= num 999999)
     ;; остальные условия
)

(defun happy-ticket-p (num)
     (and (<= 0 num 999999)
     ;; остальные условия
)
```



Стандартные предикаты: проверка на принадлежность к типу

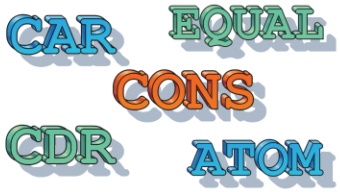
```
[1]> (setf x 42)
42
[2]> (typep x 'number)
T
[3]> (numberp x)
T
[4]> (null x)
NIL
[5]> (null nil)
T
[6]> (listp '(a b c))
T
```

предикат
null
symbolp
atom
consp
listp
numberp
integerp
rationalp
floatp
realp
complexp
stringp
functionp



Стандартные предикаты: прочие функции

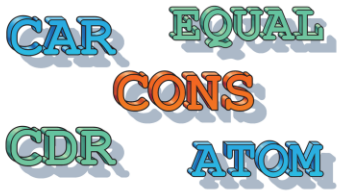
```
[1]> (oddp 7)
T
[2]> (evenp 7)
NIL
[3]> (upper-case-p #\A)
T
[4]> (yes-or-no-p "Lisp rocks?")
Lisp rocks? (yes or no) sure
Please type "yes" for yes or "no" for no.
Lisp rocks? (yes or no) yes
T
```



Лекция 2

Основы языка Lisp

Арифметические функции, поразрядные операции и манипуляции с байтами



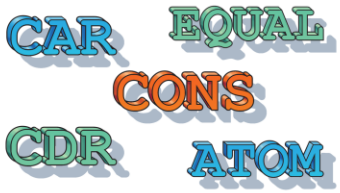
Стандартные арифметические функции

- сложение, вычитание, умножение, деление

```
[1]> (* 1.5 11/13 0.2e-1 #C(1 0))
0.025384616
```

- инкремент и декремент
  - 1+, 1–
  - incf, decf (похожи на операции ++ и – – из Си)

```
[2]> (1+ 42)              ;; не (1 + 42) !
43
[3]> (incf 42)            ;; ошибка!
*** - GET-SETF-EXPANSION: Argument 42 is not a SETF place.
[4]> (defvar x 42)
X
[5]> (incf x)             ;; так правильно
43
[6]> (incf x 10)
53
```



Прочие математические функции

- максимум и минимум

```
[1]> (max 0 -1.5 2.22)
2.22
[2]> (min 0 -1.5 2.22)
-1.5
```

- возведение в степень

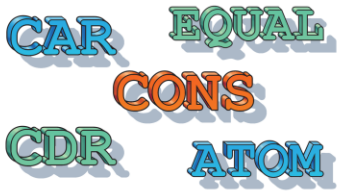
```
[3]> (expt 2 8)
256
[4]> (expt 8 1/3)
2.0
```

- извлечение квадратного корня

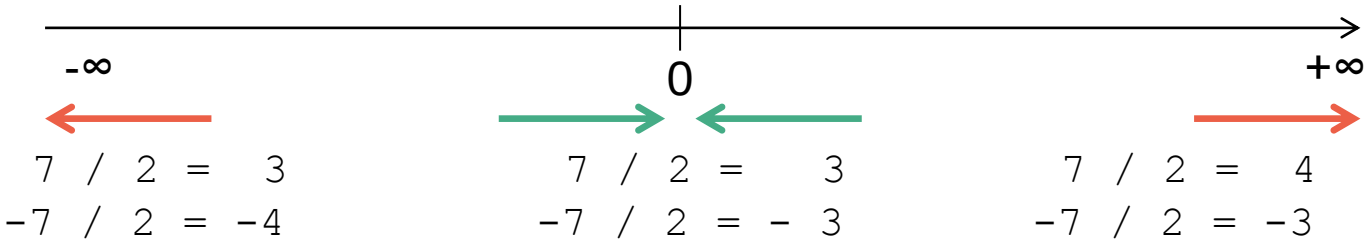
```
[5]> (sqrt 4)
2.0
```

Common Lisp HyperSpec™
The very definition of class.

http://clhs.lisp.se



Округление до целого



- округление в сторону -∞

```
[1]> (floor -7.5)
-8 ;
0.5
```

- округление в сторону +∞

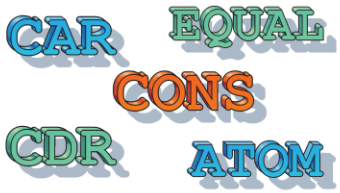
```
[2]> (ceiling -7.5)
-7 ;
-0.5
```

- округление в сторону 0

```
[3]> (truncate -7.5)
-7 ;
-0.5
```

- округление в сторону ближайшего целого

```
[4]> (round -7.5)
-8 ;
0.5
```



Остаток от деления

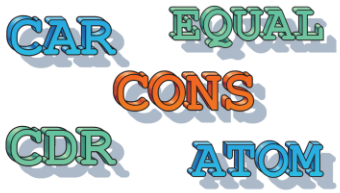
a mod b = a - floor(a/b)*b

a rem b = a - truncate(a/b)*b

```
[1]> (rem 5 3)
2
[2]> (mod 5 3)
2
[3]> (rem 5 -3)
2
[4]> (mod 5 -3)
-1
```

5 rem - 3 = 5 - (5/(-3))*(-3) = 5 - (**-1**)*(-3) = 2

5 mod -3 = 5 - (5/(-3))*(-3) = 5 - (**-2**)*(-3) = -1



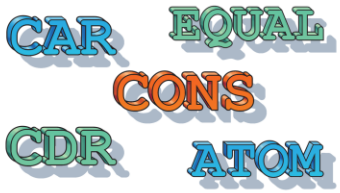
Размер целого числа в байтах

Функция integer-length позволяет узнать, сколько двоичных разрядов требуется для представления числа:

```
[1]> (integer-length 15)
4
[2]> (integer-length 16)
5
[3]> (integer-length #xFFF)
12
```

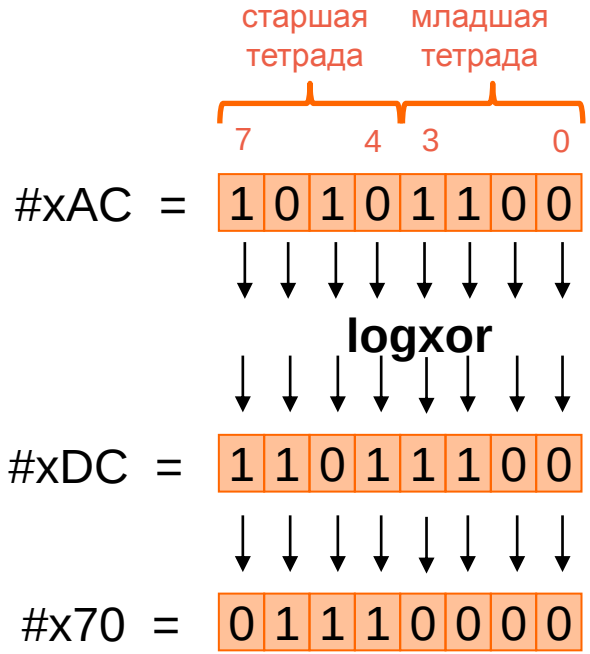
Зная, что в байте 8 двоичных разрядов, количество байтов можно найти так:

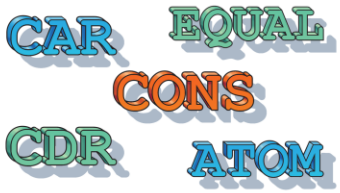
```
[4]> (ceiling (integer-length #xFFF) 8)
2;
-4
```



Побитовые операции

```
[1]> (lognot 1)
-2
[2]> (logand 7 8)
0
[3]> (logior #xAC #xDC)
252
[4]> (logxor #xAC #xDC)
112
[5]> (format t "#x~x"
           (logxor #xAC #xDC))
#x70
NIL
[6]> (logbitp 2 6)
T
```

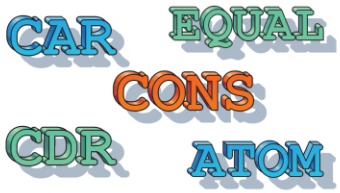




Арифметический сдвиг

(ash число количество_разрядов)

```
[1]> (ash 1 1)
2
[2]> (ash 1 7)
128
[3]> (ash 1 -1)
0
[4]> (ash -2 -1)
-1
```

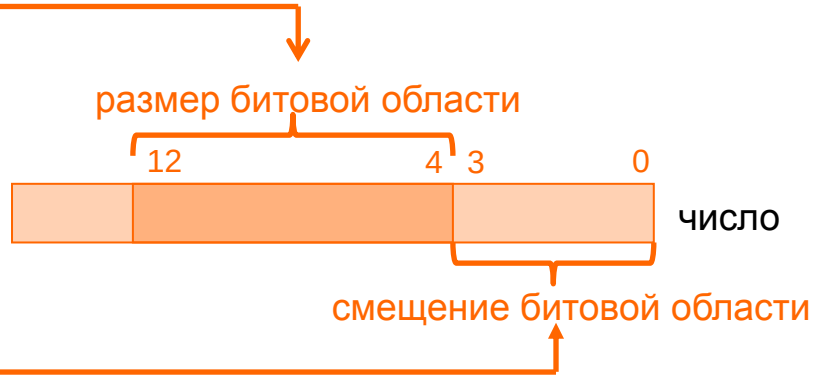


Работа с битами и байтами: функция byte

(byte размер позиция)

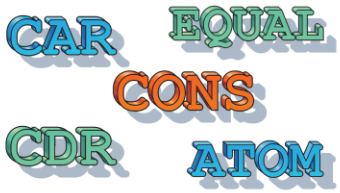
B sbcl:

```
* (byte 9 4)
(9 . 4)
```



B clisp:

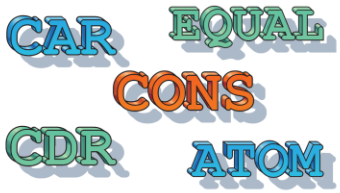
```
[1]> (byte 9 4)
#S(BYTE :SIZE 9 :POSITION 4)
```



Работа с битами и байтами: функция ldb

(ldb спецификация число)

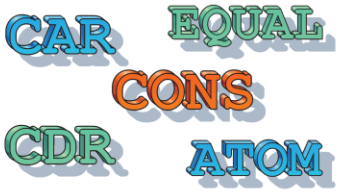
```
[1]> (ldb (byte 8 0) #xCAFEBABE)
190     ; #xBE
[2]> (ldb (byte 8 16) #xCAFEBABE)
254     ; #xFE
[3]> (ldb (byte 4 16) #xC0FFEE)
0
```



Работа с битами и байтами: функция dpb

(dpb значение спецификация число)

```
[1]> (defvar x #xD00D)
X
[2]> (dpb #xEA (byte 8 4) x)
57005
[3]> (format t "~x" x)
D00D
NIL
[4]> (setf x (dpb #xEA (byte 8 4) x))
57005
[5]> (format t "~x" x)
DEAD
NIL
```



Задача

Прочитать беззнаковое целое число из стандартного потока ввода и изменить порядок следования битов в байтах числа на обратный. Результат вывести в стандартный поток вывода.

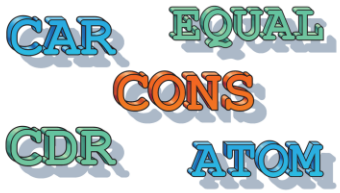
Исходное число: 345

345 = 00000001 01011001

меняем порядок битов каждом байте на обратный

10000000 10011010 = 32922

Результат: 32922



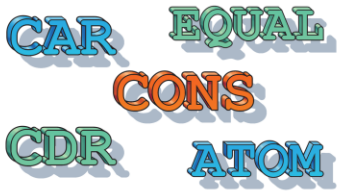
Чтение из стандартного потока ввода и запись в стандартный поток вывода

- чтение – функция read

```
[1]> (defvar num 0)
NUM
[2]> (setf num (read))
101
101
```

- запись – функции print или format

```
[3]> (print num)
101
101
[4]> (format t "~d" num)
101
NIL
```



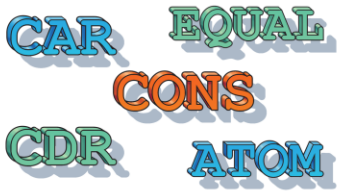
Циклы: простой цикл с помощью loop

(loop форма1 ... формаN)

```
[1]> (setf n 3)
3
[2]> (loop
        (unless (>= n 0) (return))
        (print n)
        (decf n))

3
2
1
0
NIL
```

значения переменной n, напечатанные функцией print

значение, возвращенное формой (return) из цикла



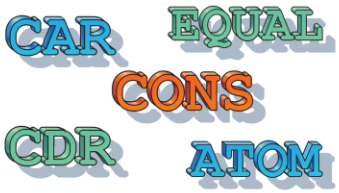
Циклы: простой цикл с помощью loop

```
[1]> (setf n 0)
0
[2]> (loop
        (when (> n 3) (return 111))
        (print n)
        (incf n))
```

```
3
2
1
0
111
```

значения переменной n, напечатанные функцией print

значение, возвращенное формой (return 111) из цикла



Циклы: цикл со счетчиком dotimes

(**dotimes** форма_инициализации форма1 … формаN)
(переменная формаC [формаR])

```
[1]> (dotimes (i 3) (print i))
0
1
2
NIL
```

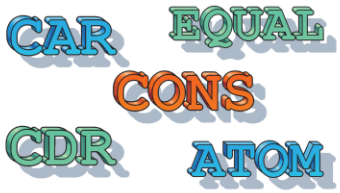
значения переменной i, напечатанные функцией print

значение, возвращенное из цикла (форма возврата отсутствует)

```
[2]> (dotimes (i 3 (* i 2)) (print i))
0
1
2
6
```

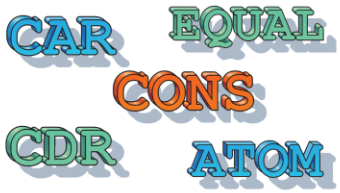
значения переменной i, напечатанные функцией print

значение, возвращенное из цикла формой (* i 2)



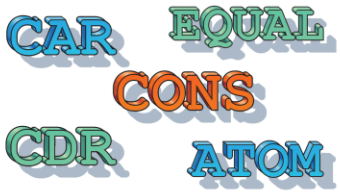
Решение задачи

```
(defvar x 0)

;; прочитать число из потока ввода
(setf x (read))

;; цикл по байтам числа
(dotimes (i (ceiling (integer-length x) 8))
    ; тут обрабатываем i-тый байт
    ; чтобы получить i-тый байт используем
    ; (ldb (byte 8 (* i 8)) x)
)

;; вывести результат
(print x)
```



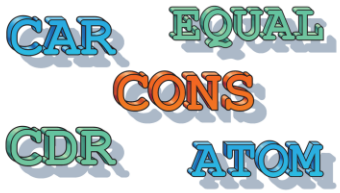
Решение задачи: функция change-byte

```
(defun change-byte (b)
    (dotimes (i 4 b)
        (when биты_не_равны
            (setf b измененный_байт))))
```



Решение задачи: проверка битов в байте

```
;; b – байт
;; i – номер бита в байте
(not (eq
    (logbitp i b)
    (logbitp (- 7 i) b)))
```

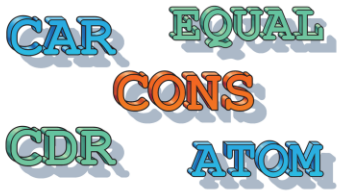


Решение задачи: инвертирование битов в байте

```
11010110      ——— исходный байт
logxor
10000001      ——— маска
--------
01010111      ——— результат
```

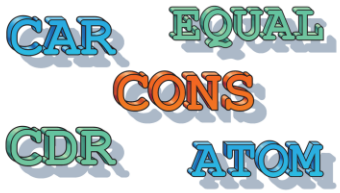
```
(logxor
   b ——— исходный байт
   (+ (ash 1 i) (ash 1 (- 7 i))))
```

маска



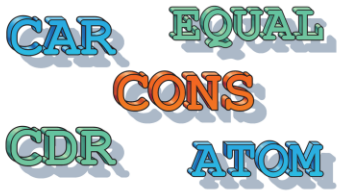
Решение задачи: функция change-byte

```
(defun change-byte (b)
    (dotimes (i 4 b)
        (when
            (not (eq
                (logbitp i b)
                (logbitp (- 7 i) b)))
            (setf b
                (logxor
                    (+ (ash 1 i) (ash 1 (- 7 i)))
                    b)))))
```



Решение задачи

```
(defvar x 0)

;; читаем число из потока ввода
(setf x (read))

;; цикл по байтам числа
(dotimes (i (ceiling (integer-length x) 8))
     (setf x
          (dpb
               (change-byte (ldb (byte 8 (* i 8)) x))
               (byte 8 (* i 8)) x))))

;; выводим результат
(print x)
```



Что мы узнали из этой лекции

- что такое формы и какие они бывают
- как записывать условия в Лиспе (if, when, unless)
- как объявлять переменные (defvar) и как изменять их значение (setf)
- ложное значение – пустой список (nil), все остальные значения считаются истинными
- проверки делаются с помощью предикатов, есть множество стандартных предикатов
- функции для работы с числами