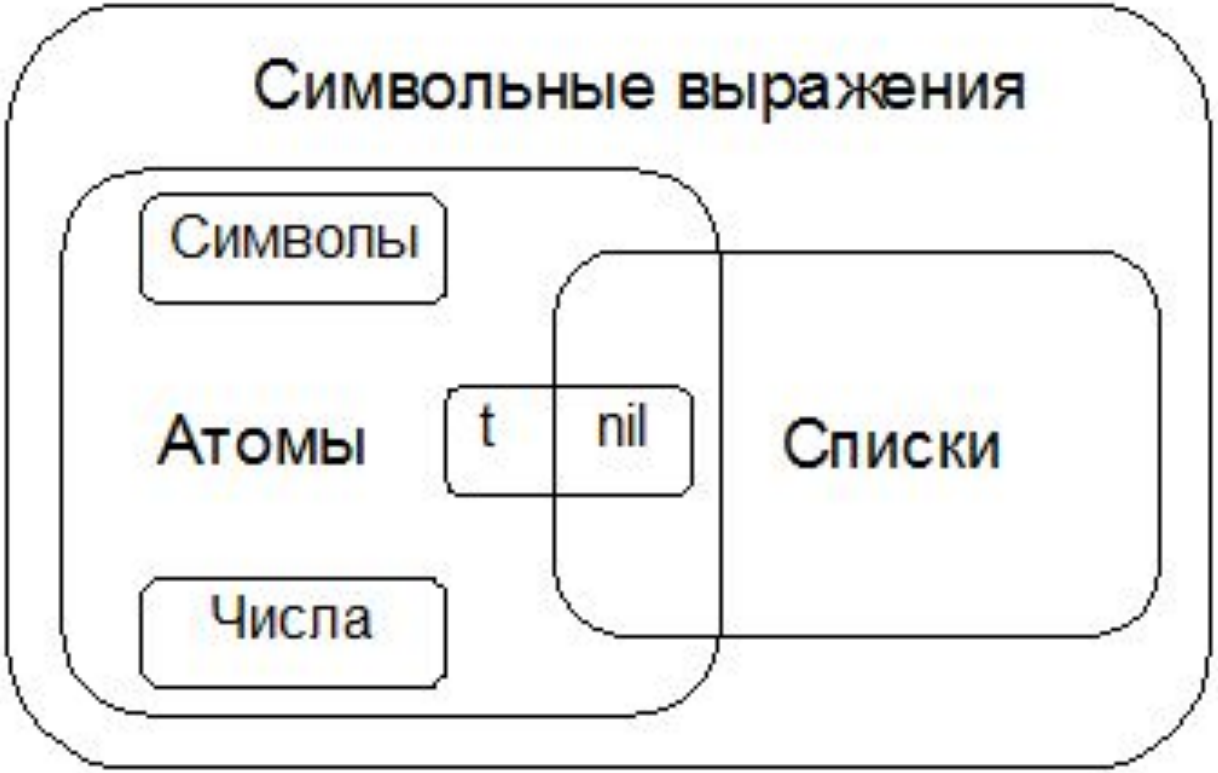
Лекция №2
Программирование на языке ЛИСП.
Символ. Определение функций

Символьные выражения

Все вышеперечисленные объекты (атомы и списки) называют **символьными выражениями.** Отношения между различными символьными выражениями можно представить следующим образом:



(S)-выражение – это либо атом, либо список; атомы являются простейшими S-выражениями.

Запись чисел

123 ==> 123 +123 ==> 123 -123 ==> -123 123. ==> 123 2/3 ==> 2/3 -2/3 ==> -2/3	4/6 ==> 2/3 6/3 ==> 2 #b10101 ==> 21 #b1010/1011 ==> 10/11 #o777 ==> 511 #xDADA ==> 56026 #36rABCDEFGHIJKLMNOPQRSTUVWXYZ ==> 83375038547304152410503771358112592678 35

Возможна запись числа с основанием, отличным от 10. Если число предваряется #B или #b, то число считывается как двоичное, в котором разрешены только цифры 0 и 1. Строки #O или #o используются для восьмеричных чисел (допустимые цифры – 0-7), а #X или #x используются для шестнадцатеричных (допустимые цифры 0-F или 0-f). Вы можете записывать числа с использованием других оснований (от 2 до 36) с указанием префикса #nR, где n определяет основание (всегда записывается в десятичном виде).
Common Lisp имеет четыре подтипа для чисел с плавающей точкой: short, single, double и long. Маркеры экспоненты в виде букв s, f, d, l (и их заглавные эквиваленты) обозначают использование short, single, double и long подтипов. Буква e показывает, что должно использоваться представление по умолчанию (первоначально подтип single).
комплексные числа имеют отличающийся синтаксис, а именно: префикс #C или #c:
#c(2 1) ==> #c(2 1)
#c(2/3 3/4) ==> #c(2/3 3/4)

Арифметические функции

- Арифметические функции могут быть использованы с целыми или действительными аргументами. Число аргументов для большинства арифметических функций может быть разным.
- **(+ x1 x2 ... xn)** возвращает x1 + x2 + x3 + ... + xn.
- **(- x1 x2 ... xn)** возвращает x1 - x2 - x3 - ... - xn.
- **(* y1 y2 ... yn)** возвращает y1 x y2 * y3 * ... * yn.
- **(/ x1 x2 ... xn)** возвращает x1/x2/... /xn.
- Специальные функции для прибавления и вычитания единицы: **(1+ x)** и **(1- x).**

Примеры арифметических выражений

(+ 1 2) ==> 3
(+ 1 2 3) ==> 6
(+ 10.0 3.0) ==> 13.0
(+ #c(1 2) #c(3 4)) ==> #c(4 6)
(- 5 4) ==> 1
(- 2) ==> -2
(- 10 3 5) ==> 2
(* 2 3) ==> 6
(* 2 3 4) ==> 24
(/ 10 5) ==> 2
(/ 10 5 2) ==> 1
(/ 2 3) ==> 2/3
(/ 4) ==> 1/4
(+ #c(1 2) 3) ==> #c(4 2)
(+ #c(1 2) 3/2) ==> #c(5/2 2)
(+ #c(1 1) #c(2 -1)) ==> 3
(+ 1 2.0) ==> 3.0
(/ 2 3.0) ==> 0.6666667

Математические функции

- Функция логарифм имеет следующий прототип (log arg) и (log arg base)

```
> (log 2.7)
0.9932518
```

- Вычисление тригонометрических функций:

```
> (sin 3.14)        > (atan 3.14)
0.00159265          1.26248
```

Логические операции

- **Сравнение с пустым списком** :

> (NULL T)

NIL

- **Отрицание** :

> (NOT NIL)

T

- **Логическое "И" (аргументов может быть 2 и более)**

> (AND T NIL)

NIL

- **Логическое "ИЛИ"**

> (OR T NIL)

T

Перед числами и символами T и NIL не нужно ставить апостроф

Арифметические операции сравнения

Поддерживаются стандартные операции, применимые к числовым вычислениям:

=, <, >, <=, >=

(>= 1 (- 3 2)) ==> T

(< 1 2) ==> T

(= 'a 'a) ==> error: bad argument type - A

(= nil '()) ==> error: bad argument type - NIL

(= 1 1.0 #c(1.0 0.0) #c(1 0)) ==> T

(/= 1 2 3) ==> T

(/= 1 2 3 1) ==> NIL

(<= 2 3 3 4) ==> T

(<= 2 3 4 3) ==> NIL

(max 10 11) ==> 11

(min -12 -10) ==> -12

(max -1 2 -3) ==> 2

ZEROP, MINUSP , PLUSP, EVENP, ODDP (число)

Знаки (Characters) - тип объекта

#\x обозначает знак x

#\Space обозначает знак «пробел»

Функции сравнения знаков

Numeric Analog	Case-Sensitive	Case-Insensitive
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL
<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

Строки – составной тип данных

Строка	Содержание	Комментарий
`"foobar"`	`foobar`	**Обычная строка.**
`"foo\"bar"`	`foo"bar`	**Обратный слэш маскирует кавычку.**
`"foo\\bar"`	`foo\bar`	**Первый обратный слэш маскирует второй.**
`"\"foobar\""`	`"foobar"`	**Обратные слэши маскируют кавычки.**
`"foo\bar"`	`foobar`	**Обратный слэш "маскирует" знак b**

Сравнение строк

Numeric Analog	Case-Sensitive	Case-Insensitive
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

Имя и значение символа

- Использование символов в качестве переменных
- Изначально символы в Лиспе не имеют значения. Значения имеют только константы.

```
> t
T
> 1.6
1.6
```

- Если попытаться вычислить символ, то система выдает ошибку.

- Значения символов хранятся в ячейках, закрепленных за каждым символом.
- Если в эту ячейку положить значение, то символ будет связан (bind) со значением. В процедурных языках говорят "будет присвоено значение".
- Для Лиспа есть отличие: Не оговаривается, что может хранится в ячейке: целое, атом, список, массив и т.д. В ячейке может хранится что угодно.
- С символом может быть связана не только ячейка со значением, а многие другие ячейки, число которых не ограничено.
- Для связывания символов используется три функции (псевдофункции):
  - **SET**
  - **SETQ**
  - **SETF**

Функция SET вычисляет оба своих аргумента и связывает первый аргумент:

(SET символ значение) ⇒ значение

SET ‘a ‘(b c d)) => (b c d)
a =>(b c d)
(**SET** (CAR a) (CDR (o f g)) => (f g)
a => (b c d)
(CAR a) => b
b => (f g)
(**Symbol-value** (CAR a)) => (f g)
(**BOUNDP** ‘c) ⇒ NIL
(**BOUNDP** ‘a) ⇒ T

***Функция SETQ – невычисляющее присваивание*:**

(SETQ символ значение) ⇒ значение

d => Error
(SETQ d ‘(l m n)) => (l m n)
d => (l m n)

Функции, обладающие побочным эффектом, называются псевдофункциями (set, setq, setf).

Функция SETF - обобщенная функция присваивания:

(SETF ячейка-памяти значение)

(**SETF** ячейка ‘(a b c)) => (a b c)

ячейка => (a b c)

(**SETF** x 1 y 2)

>x ==>1 >y==> 2

Таблица 6-1. Присваивание с помощью = в других языках программирования

Присваивание ...	**Java, C, C++**	**Perl**	**Python**
... переменной	x = 10;	$x = 10;	x = 10
... элементу массива	a[0] = 10;	$a[0] = 10;	a[0] = 10
... элементу хэш-таблицы	–	$hash{'key'} = 10;	hash['key'] = 10
... полю объекта	o.field = 10;	$o->{'field'} = 10;	o.field = 10

SETF работает сходным образом: первый "аргумент" **SETF** является "местом" для хранения значения, а второй предоставляет само значения. Как и с оператором = в этих языках, вы используете одинаковую форму и для выражения "места", и для получения значения.

Эквиваленты присваиваний для Lisp следующие:

AREF — функция доступа к массиву, (SETF (AREF mass 0) "aaa") => "aaa”

GETHASH осуществляет операцию поиска в хэш-таблице,

field может быть функцией, которая обращается к слоту под именем field определенного пользователем объекта

(setf x (+ x 1))
(setf x (- x 1))

Модифицирующие макросы:
(incf x) === (setf x (+ x 1))
(decf x) === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))

С символом можно связать именованные свойства:

(имя1 значение1 имя2 значение2 ... имяN значениеN)

Функция GET - возвращает значение свойства, связанного с символом:

(GET символ свойство)

(SETF (GET символ свойство) значение)

(SETF (GET ‘студент ’группа) ’КИО8-15) => КИО8-15
(GET ‘студент ’группа) => КИО8-15

(SYMBOL-PLIST символ)

(SYMBOL-PLIST ‘студент) => (имя Иван отчество Иванович фамилия Иванов)

Передача свойств другому символу:

(SETF (get 'st 'sv1) '1) => 1
(GET 'st 'sv1) => 1
(SETQ st1 'st) => st
(GET st1 'sv1) => 1

Использование функций EVAL, QUOTE

(EVAL (QUOTE (+ 2 3))) => 5

(SETQ x '(a b c)) => (a b c)
'x => x
x => (a b c)

(EVAL 'x) => (a b c)
(EVAL x) => error: unbound function – a

(SETQ x '(+ 2 3)) => (+ 2 3)
x => (+ 2 3)
(EVAL 'x) => (+ 2 3)
(EVAL x) => 5

Форматирование кода Lisp

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)

(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))

(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

Комментарии:

;;;; Четыре точки с запятой для комментария в начале файла

;;; Комментарий из трех точек с запятой обычно является параграфом комментариев,
;;; который предваряет большую секцию кода

;; Две точки с запятой показывают, что комментарий применен к последующему коду.
;; Заметьте, что этот комментарий имеет такой же отступ, как и последующий код
(some-function-call)

(another i) ; этот комментарий применим только к этой строке
(and-another) ; а этот для этой строки
(baz)))

Функции

Лямбда-выражение: (LAMBDA (*x1 x2 ... xn*) *fn*).
Пример: (LAMBDA (x y) (+ x y))

Лямбда-вызов: (лямбда-выражение a1 a2 ... an),
здесь ai - формы, задающие фактические параметры.
Пример: ((LAMBDA (x y) (+ x y)) 1 2) => 3

лямбда-преобразование:
1. выполняется связывание параметров. Вычисляются фактические параметры и с их значениями связываются соответствующие формальные параметры;
2. вычисляется тело лямбда-выражения и полученное значение возвращается в качестве значения лямбда-вызова;
3. формальным параметрам возвращаются связи, существовавшие перед лямбда-преобразованием.

Вложенные лямбда-вызовы:
((LAMBDA (x) ((LAMBDA (y) (LIST x y)) 'b)) 'a) => (a b)

Применение LAMBDA-выражений:
1. Передача функционального параметра при вызове функции (функционала)
2. Использование для создания замыканий (closures) – функций, которые привязаны к контексту, в котором они были созданы.

Определение функций

(DEFUN *имя лямбда-список тело*) => имя

Имя функции **frob-widget** лучше соответствует стилю Lisp, чем **frob_widget** или **frobWidget**

(DEFUN list1 (x y) (cons x (cons y nil))) => list1
(list1 ‘c ‘n) => (c n)

(FBOUNDP *'символ*) => T, Nil

Тело функции состоит из любого числа s-выражений:

```
(defun verbose-sum (x y)
   (format t "Summing ~d and ~d.~%" x y)
   (+ x y))
```

Содержит два выражения в теле функции

```
CL-USER 39 : 5 > verbose-sum 2 3
Summing 2 and 3.
5
```

Списки параметров функций

обязательные, необязательные , ключевые и остаточные

Необязательные параметры: значения объявленных при помощи **&OPTIONAL** параметров можно в вызове не указывать. В этом случае они связываются со значением NIL или со значением выражения по умолчанию, если таковое имеется.

Пример: здесь x –обязательный параметр, y – необязательный со значением по умолчанию.

```
> (defun fn (x &optional (y (+ x 2))) (list x y))
fn
> (fn 2) вычисляется значение по умолчанию
(2 4)
> (fn 2 6)    умолчание не используется
(2 6)

(defun foo (a b &optional (c 3 c-supplied-p)) (list a b c c-supplied-p))

(foo  1 2) ==> (1 2 3 NIL)
(foo 12 3) ==> (1 2 3 T)
(foo 1 2 4) ==> (1 2 4 T)
```

Переменное количество параметров: параметр, указанный после **&REST**, связывается со списком несвязанных параметров, указанных в вызове. Таким функциям можно передавать переменное количество параметров.

Любые аргументы, оставшиеся после связывания обязательных и необязательных параметров, будут собраны в список, который станет значением остаточного параметра &rest

Пример:

```
> (defun fn (x &optional y &rest z) (list x y z))
fn
> (fn 2 3)
(2 3 nil)
> (fn 1 2 3 4 5)
(1 2 (3 4 5))
>  (defun fn (x &optional y &rest z) (list x y (car z)))
fn
> (fn 1 2 3 4 5 )
(1 2 3)
```

Ключевые параметры: если формальные параметры обозначены в определении функции словом **&KEY**, то соответствующие им фактические параметры можно перечислять в вызове в произвольном порядке при помощи специальных ключей.

Ключом является **имя формального параметра, перед которым поставлено двоеточие**, напр. «:X». Соответствующий фактический параметр следует за ключом и отделяется от него пробелом.

Ключевые параметры являются необязательными.

Пр.

```
> (defun fn (&key x y (z 3)) (list x y z))
> (fn :z 4 :x 5 :y 6) ==> (5 6 4)
> (fn :y 6 :x 5) ==>  (5 6 3)

>(defun foo (&key a b c) (list a b c))
>(foo) ==> (NIL NIL NIL)
>(foo :a 1) ==> (1 NIL NIL)
>(foo :b 1) ==> (NIL 1 NIL)
>(foo :a 1 :c 3 :b 2) ==> (1 2 3)

>(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b))) (list a b c b-supplied-p))
  >foo :a 1) ==> (1 0 1 NIL)
  >(foo :b 1) ==> (0 1 1 T)
  >(foo :b 1 :c 4) ==> (0 1 4 T)
  >(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Символ имеет:
1. имя;
2. значение, назначенное функцией присваивания (setq);
3. описание вычислений (лямбда - выражение), назначенное определением функции DEFUN.
4. список свойств.

Пример. Пусть имена отца и матери некоторого лица хранятся как значения соответствующих свойств у символа, связанного с именем этого лица. Определим:
функцию (родители x), которая возвращает в качестве значения список родителей;
предикат (сестры-братья x1 x2), который истинен в том случае,
если x1 и x2 имеют общего родителя.

Решение.

```
(SETQ  родители  ‘(Оля Коля) ) => (Оля Коля)
родители=> (Оля Коля)
(SETF (GET 'Иван 'мама) 'Оля)  =>  Оля
(SETF (GET 'Иван 'папа) 'Коля)  => Коля
(SETF (GET 'Петр 'мама) 'Оля)  =>  Оля
(SETF (GET 'Петр 'папа) 'Коля)  =>  Коля
(SYMBOL-PLIST 'Иван)  =>  (папа Коля мама Оля)
(SYMBOL-PLIST 'Петр)  =>  (папа Коля мама Оля)
(DEFUN родители (x) (LIST (GET x 'мама) (GET x 'папа)))  =>  родители
(родители 'Сергей)  =>  (nil nil)
(родители 'Иван)  =>  (Оля Коля)
(DEFUN сестры-братья (x1 x2)
   (or  (EQ (GET x1 'мама)  (GET x2 'мама))
   (EQ (GET x1 'папа )(GET x2  'папа)) ) )  =>  сестры-братья
(сестры-братья ‘Иван ‘Петр)  =>  T
```