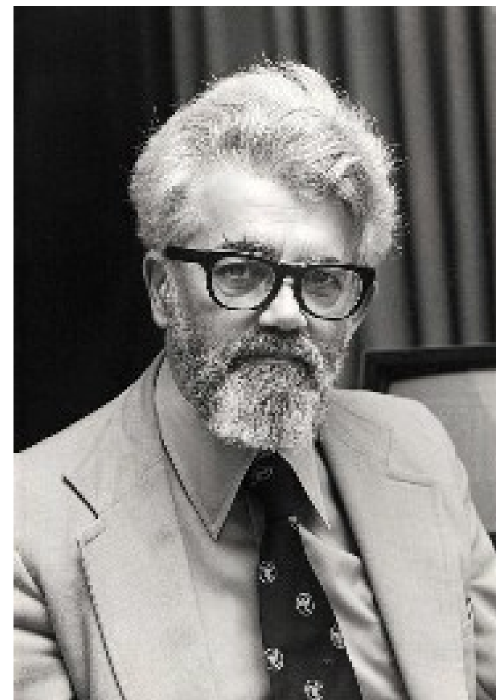
Интеллектуальные системы и технологии

Лекция 4.

Язык для работы со списками LISP

Автор - Джон Маккарти (John McCarthy) в 1957 году

Стэнфордский университет



Основные понятия

- Язык Lisp является языком функционального программирования
- В Lisp’е как программы, так и данные, представляются одинаково – в виде списков. Например, запись (+ 1 2) может толковаться, в зависимости от контекста, как список, состоящий из трех элементов (данные), или как вызов функции суммирования с двумя аргументами (программа).

Основные понятия (2)

- При написании программ на Lisp’е используются символы и создаваемые на основе символов символьные выражения. Символ в Lisp’е аналогичен переменной в традиционном языке программирования – это имя, состоящее из букв латиницы, цифр и некоторых специальных литер. Символ, как и переменная, может иметь какое-либо значение, то есть представлять какой-либо объект.
- Наряду с символами, в Lisp’е используются также:
  - числа, целые и вещественные;
  - специальные константы t и nil, обозначающие логические значения true (истина) и false (ложь);
  - списки.

Основные понятия (3)

- Символы, числа и специальные константы представляют простейшие объекты, на основе которых строятся все прочие объекты данных. Поэтому для них используется обобщающее название – атомы
- Все вышеперечисленные объекты (атомы и списки) называют символьными выражениями (S-выражение).

Списки

- Список – это основной тип данных в Lisp. Список заключается в круглые скобки, элементы списка разделяются пробелами. Пустой список обозначается парой скобок – (). Для обозначения пустого списка используется также специальная константа nil.
- Примеры списков:
  – пятиэлементный список
  (1 2 3 4 5)
  – четырехэлементный список
  (1 2 ((3) 4) 5)
  – одноэлементный список
  ((1 2 3 4 5))
- Первый элемент списка называется головой списка, все прочие элементы, кроме первого, представленные как список, называются хвостом списка.

Функции

- В Lisp’е для вызова функции принята единообразная префиксная форма записи, при которой как имя функции, так и ее аргументы записываются в виде элементов списка, причем имя функции – это всегда первый элемент списка.
- В Lisp’е передача параметров в функцию осуществляется по значению.
- Параметры используются для того, чтобы передать данные в функцию, а результат функции возвращается как ее значение, а не как значение параметра, передаваемого по ссылке.

Примеры

- Возвращаемое значение 8

(+ 3 5)

- Возвращаемое значение 21

(* (+ 1 2) (+ 3 4))

- Возвращаемое значение 0

(sin 3.14)

Списки (2)

- Список может рассматриваться и не как вызов функции, а как перечень равноправных элементов. Для блокирования вызова функции используется, в свою очередь, функция quote.
- Например, список

(+ 3 5)

- будет восприниматься как вызов функции суммирования с аргументами 3 и 5. Если же использовать данный список в качестве аргумента функции quote

(quote (+ 3 5))

- то список воспринимается именно как список. То есть применение функции quote блокирует вызов функции и ее имя воспринимается как обычный элемент списка. Или, иначе говоря, если список является аргументом функции quote, то первый элемент списка не считается именем функции, а все прочие элементы не считаются аргументами функции.
- Для функции quote существует сокращенная форма записи, вместо имени функции quote используется апостроф перед открывающейся скобкой списка. Например:

‘(+ 3 5)

Примеры (2)
> означает «вводите»

```
> (+ 3 5)
8
> (quote (+ 3 5))
(+ 3 5)
> ‘(+ 3 5)
(+ 3 5)
> (quote (quote (+ 3 5)))
(QUOTE (+ 3 5))
```

Функция присваивания

- Для выполнения операции присваивания используется функция set. Формат функции (set variable value). Причем, если не требуется вычисления аргументов, их нужно предварить апострофами. Например:

```
> (set ‘x 5)
5
> x
5
> (set ‘y (+6 12))
18
> y
18
> (set 'a 'b)
B
> (set ‘a 'c)
C
```

Функция присваивания (2)

- Вместо функции set можно использовать функцию setq, также выполняющую присваивание, но при ее использовании нет необходимости предварять первый аргумент апострофом. Для первого аргумента блокировка вычисления выполняется автоматически.
- Например:

```
> (setq x 5)
5
> x
5
> (setq y (+6 12))
18
> y
18
```

- Функцию, которая в качестве значения возвращает только константы nil или t, называют предикатом.
- Для определения собственной функции можно воспользоваться стандартной функцией defun (сокращение от DEfine FUNction). Эта стандартная функция позволяет создавать собственные функции, причем не запрещается переопределять стандартные функции, то есть в качестве имени собственной функции использовать имя стандартной функции. Функция defun выглядит следующим образом: (defun name (fp1 fp2 … fpN) (form1 form1 … formN)).
- Name – это имя новой функции, (fp1 fp2 … fpN) – список формальных параметров, а (form1 form1 … formN) – тело функции, то есть последовательность действий, выполняемых при вызове функции.

- Пример – функция для вычисления суммы квадратов:

```
(defun squaresum (x y) (+ (* x x) (* y y)))
```

Результат работы:

```
>(squaresum 3 4)
25
>(squaresum -2 -4)
20
```

Еще один пример: функция deftype, определяющая тип выражения (пустой список, атом или список).

```
(defun deftype(arg) (cond ((null arg) ‘emptylist) ((atom arg) ‘atom) (t ‘list)))
```

Результат работы:

```
>(deftype ())
EMPTYLIST
>(deftype 'abc)
ATOM
>(deftype '(a b c))
LIST
```

5 базовых функций

- (car list) – отделяет голову списка (первый элемент списка);
- (cdr list) – отделяет хвост списка (все элементы, кроме первого, представленные в виде списка);
- (cons head tail) – соединяет элемент и список в новый список, где присоединенный элемент становится головой нового списка;
- (equal object1 object2) – проверяет объекты на равенство;
- (atom object) – проверяет, является ли объект атомом.

Примеры применения базовых функций

```
> (car ‘(one two three))
ONE
> (car ‘(one))
ONE
> (car ‘())
NIL
> (cdr ‘(first second third))
(SECOND THIRD)
> (cdr ‘(first))
NIL
> (cdr ‘())
NIL
> (cons 1 ‘(2 3))
(1 2 3)
> (cons 1 nil)
(1)
> (cons nil nil)
(NIL)
```

```
> (equal ‘(1 2 3) ‘(1 2 3))
T
> (equal ‘(1 2 3) ‘(3 2 1))
NIL
> (equal ‘one ‘two)
NIL
> (atom ‘one)
T
> (atom 1)
T
> (atom (1))
NIL
```

Управляющие структуры (предложения)

- Для организации различных видов программ в Lisp’e служат разнообразные управляющие структуры, которые позволяют организовывать ветвление, циклы, последовательные вычисления:
  – предложение let – служит для одновременного присваивания значений нескольким символам;
  – предложения prog1, prog2, progn – используются для организации последовательных вычислений;
  – предложение cond – используется для организации ветвления, и, следовательно, очень важно для организации рекурсии;
  – предложение do – традиционный цикл

Предложение let

- (let ((var_1 value_1) (var_2 value_2) … (var_n value_n)) form_1 form_2 … form_m)
- Работает предложение let следующим образом: переменным var_1, var_2, … var_n присваиваются (параллельно!) значения value_1, value_2, … value_n, а затем вычисляются (последовательно!) значения форм form_1 form_2 … form_m. В качестве значения всего предложения let возвращается значение последней вычисленной формы form_m
- Пример:

```
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
```

Предложение let (2)

- После завершения выполнения предложения let переменные var_1, var_2, … var_n получают значения, которые они имели до использования в этом предложении, то есть предложение let выполняет локальные присваивания.

```
> (setq x ‘three)
THREE
> (setq y ‘four)
FOUR
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
> x
THREE
> y
FOUR
```

Предложение prog1, prog2, progn

(prog* form_1 form_2 … form_n)

- Все три предложения работают одинаково, последовательно вычисляются значения форм form_1, form_2, …, form_n.
- Различие между предложениями проявляется только в тех значениях, которые они возвращают: предложение prog1 возвратит значение первой формы form_1, предложение prog2 возвратит значение второй формы form_2, а предложение progn возвратит значение последней формы form_n. Во всем остальном эти предложения ничем не отличаются.

```
> (prog1 (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
2
> (prog2 (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
4
> (progn (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
16
```

Предложение cond

- Предложение cond предназначено для организации ветвления (это предложение является аналогом оператора выбора – переключателя switch в языке С). Формат предложения cond выглядит следующим образом:

```
(cond
(predicate1 form1)
(predicate2 form21 form22 … form2M)
(predicate3)
…
(predicateN formN)
)
```

Предложение cond (2)

- При выполнении предложения cond последовательно вычисляются значения предикатов, обозначенных как predicate. Если предикат возвращает значение t, тогда вычисляется значение вычислимой формы form и полученное значение возвращается в качестве значения всего предложения cond. Другими словами, идет последовательный перебор предикатов до тех пор, пока не встретиться предикат, который будет истинен.
- Для некоторого предиката может отсутствовать вычислимая форма. В таком случае предложение cond возвратит значение самого предиката. Для некоторого предиката вычислимых форм может быть несколько. В таком случае формы будут вычислены последовательно, и значение последней формы будет возвращено как значение всего предложения cond.

Пример с cond

Определение функции:

```
(defun snumberp (num)
(cond
((numberp num)
(cond
((< num 0) ‘neg_number)
((= num 0) ‘zero)
((> num 0) ‘pos_number)
)
)
(t ‘not_number)
)
)
```

Результат работы программы:

```
> (snumberp 1)
POS_NUMBER
> (snumberp -1)
NEG_NUMBER
>(snumberp 0)
ZERO
> (snumberp 'a)
NOT_NUMBER
```

Предложение do

Аналог цикла for в языке С.

```
(do
((var_1 value_1) (var_2 value_2) … (var_n
  value_n))
(condition form_yes_1 form_yes_2 …
  form_yes_m)
form_no_1 form_no_2 … form_yes_k
)
```

Выполнение предложения do

- Предложение do работает следующим образом: первоначально переменным var_1, var_2, …, var_n присваиваются значения value_1, value_2, …, value_n (параллельно, как в предложении let).
- Затем проверяется условие выхода из цикла condition.
  - Если условие выполняется, последовательно вычисляются формы form_yes_1, form_yes_2, …, form_yes_m, и значение последней вычисленной формы form_yes_m возвращается в качестве значения всего предложения do.
  - Если же условие condition не выполняется, последовательно вычисляются формы form_no_1, form_no_2, …, form_yes_k, и вновь выполняется переход в проверке условия выхода из цикла condition.

Пример с do

```
> (defun power (x n)
(do
;присваивание начального значения переменной result
((result 1))
;условие выхода их цикла
((= n 0) result)
;повторяющиеся действия
(setq result (* result x)) (setq n (- n 1))))
POWER
> (power 2 3)
8
```

Простая рекурсия

- Функция является рекурсивной, если в ее определении содержится вызов этой же функции. Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.
- Например, задача нахождения значения факториала n! сводится к нахождению значения факториала (n-1)! и умножения найденного значения на n.
- Пример: нахождение значения факториала n!.

```
> (defun factorial (n)
(cond
;факториал 0! равен 1
((= n 0) 1)
;факториал n! равен (n-1)!*n
(t (* (factorial (- n 1)) n))))
FACTORIAL
>(factorial 3)
6
```

Запуск вычисления факториала с трассировкой

```
> (trace factorial)
(FACTORIAL)
> (factorial 3)
Entering: FACTORIAL, Argument list: (3)
Entering: FACTORIAL, Argument list: (2)
Entering: FACTORIAL, Argument list: (1)
Entering: FACTORIAL, Argument list: (0)
Exiting: FACTORIAL, Value: 1
Exiting: FACTORIAL, Value: 1
Exiting: FACTORIAL, Value: 2
Exiting: FACTORIAL, Value: 6
6
```

Отключение трассировки:

```
> (untrace factorial)
NIL
```

Пример «копирование списка»

```
> (defun copy_list (list)
(cond
;копией пустого списка является пустой список
((null list) nil)
;копией непустого списка является список, полученный из головы и
    копии
;хвоста исходного списка
(t (cons (car list) (copy_list (cdr list))))))
COPY_LIST

>(copy_list '(1 2 3))
(1 2 3)
>(copy_list ())
NIL
```

Пример «проверка принадлежности элемента списку»

```
> (defun member (el list)
(cond
;список просмотрен до конца, элемент не найден
((null list) nil)
;очередная голова списка равна искомому элементу, элемент
    найден
((equal el (car list)) t)
;если элемент не найден, продолжить его поиск в хвосте списка
(t (member el (cdr list)))))
MEMBER

> (member 2 '(1 2 3))
T
> (member 22 '(1 2 3))
NIL
```

Пример «реверс списка»

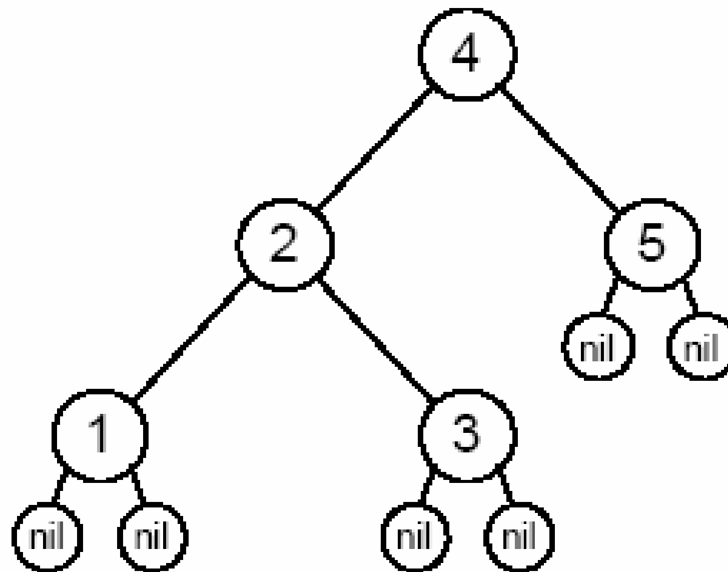
```
> (defun reverse (list)
(cond
;реверс пустого списка дает пустой список
((null list) nil)
;соединить реверсированный хвост списка и голову
   списка
(t (append (reverse (cdr list)) (cons (car list) nil)))))
REVERSE
> (reverse '(one two three))
(THREE TWO ONE)
> (reverse ())
NIL
```

Представление бинарного дерева

(4 (2 (1 nil nil) (3 nil nil)) (5 nil nil))

Или

(((nil 1 nil) 2 (nil 3 nil)) 4 (nil 5 nil))



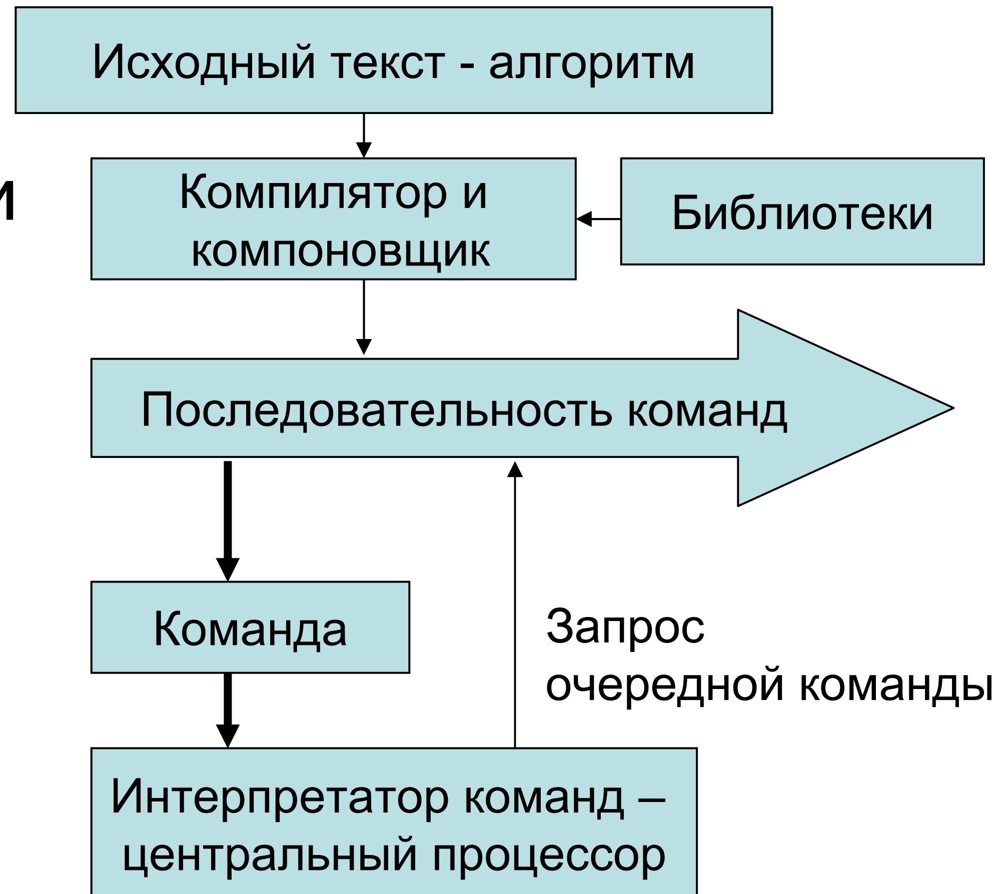
Пример: обход дерева и подсчет числа узлов дерева

```
> (defun node_counter (tree)
(cond
;число узлов пустого дерева равно 0
((null tree) 0)
;число узлов непустого дерева складывается из: одного корня,
;числа узлов левого поддерева и числа узлов правого поддерева
(t (+ 1 (node_counter (second tree)) (node_counter (third tree))))))
NODE_COUNTER
> (node_counter '(4 (2 (1 nil nil) (3 nil nil)) (5 nil nil)))
5
```

- First, second и third – стандартные функции для получения 1-го, 2-го и 3-го элементов списка, соответственно

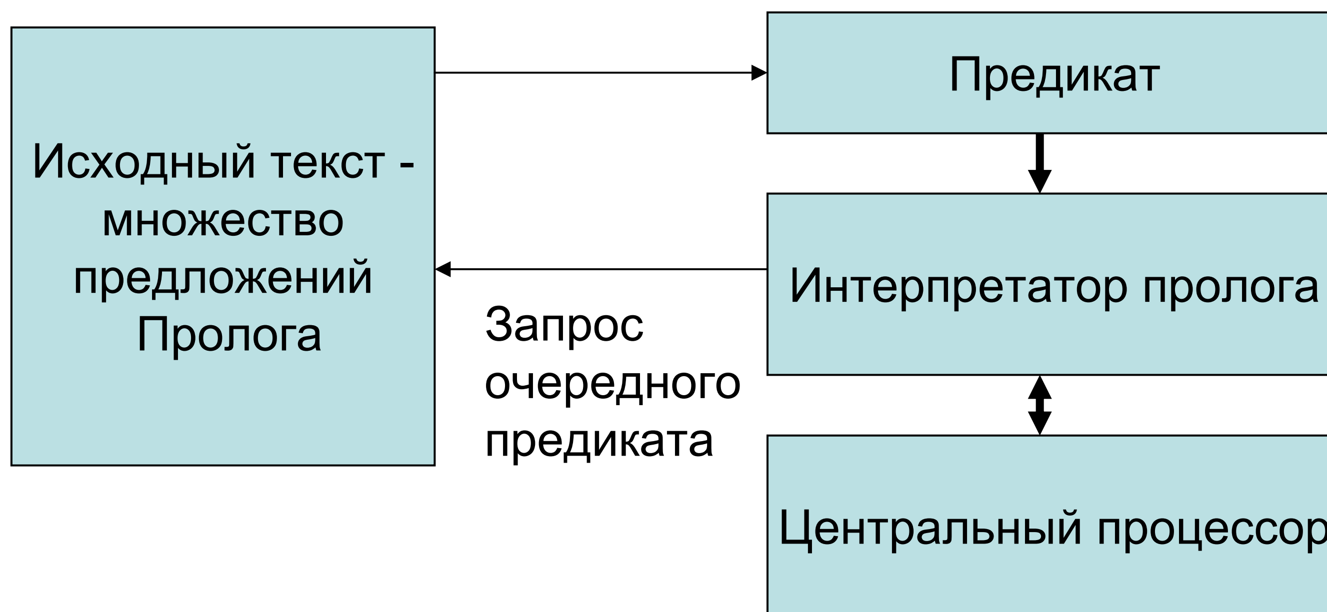
Сравнение языков

- Алгоритмические и объектно-ориентированные
  - Pascal,
  - C
  - C++
  - C#
  - Java



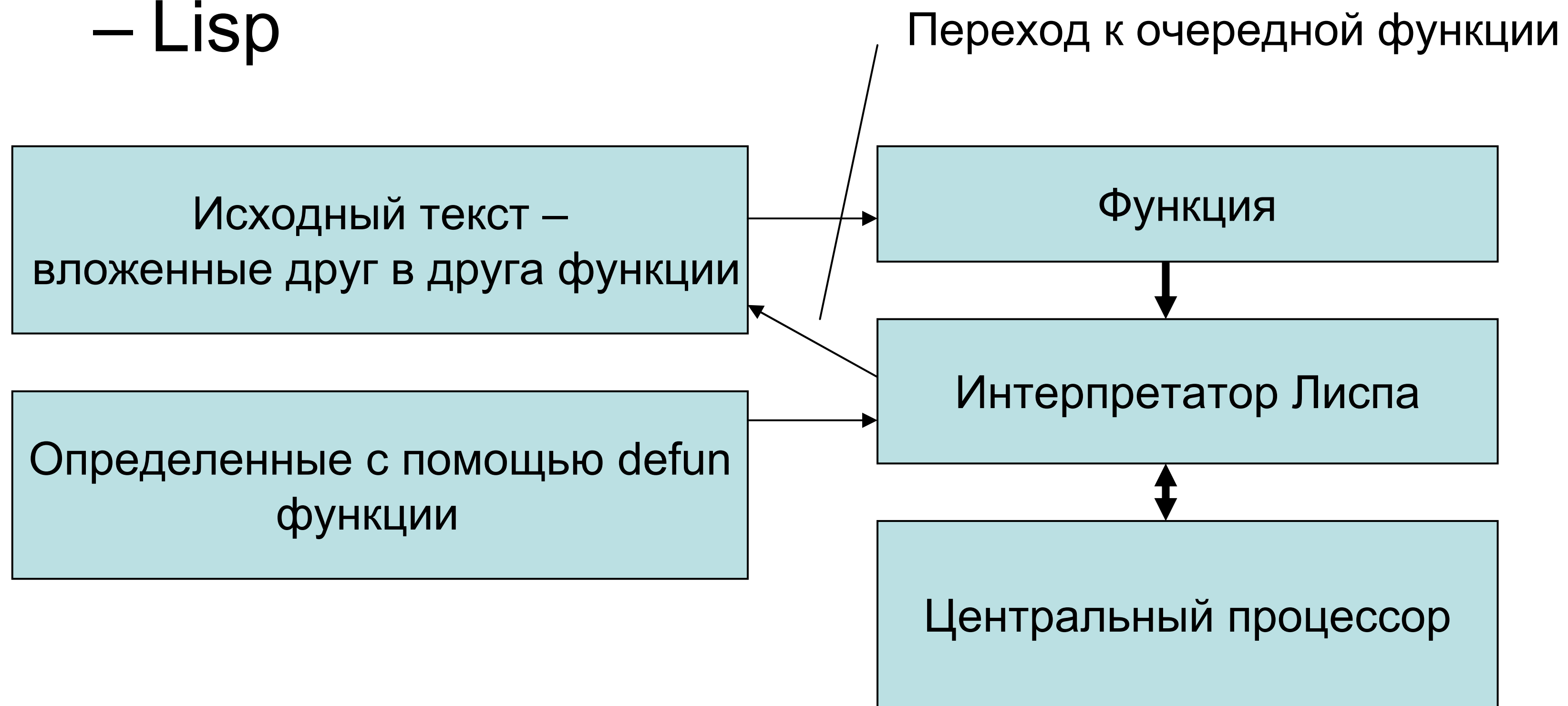
Сравнение языков (2)

- Логические
  - Prolog



Сравнение языков (3)

- Функциональные
  - Lisp



Литература

- Новицкая Ю.В. Основы логического и функционального программирования. - НГТУ, 2004.
- Хювёнен Э., Сеппянен Й. Мир Лиспа. - М.: Мир, 1990.
- Хендерсон П. Функциональное программирование: применение и реализация. - М.:Мир, 1983.
- Маурер У. Введение в программирование на языке ЛИСП. – М.: Мир, 1978.