Функциональное и логическое программирование

Лекция №1. Теоретические основы функционального программирования

Лекция №2. Базовые функции языка Лисп.

Лисп это древний язык программирования, разработанный Джоном Маккарти в 1961 году. Название языка происходит из «lisp processing» (обработка списков).
Существует альтернативная расшифровка названия LISP: Lots of Irritating Superfluous Parentheses («Много раздражающих лишних скобок») — намёк на особенности синтаксиса языка.

Особенности языка Лисп

1. Символьная обработка
Символьная обработка позволяет эффективно работать с такими структурами, как предложения естественного языка, значения слов и предложений, нечеткие понятия и т. д. И на их основе принимать решения, проводить рассуждения и осуществлять другие, свойственные человеку способы обращения с данными.

2. Одинаковая форма данных и программы
И то и другое представляется списочной структурой, имеющей одинаковую форму. Таким образом, программы могут обрабатывать и преобразовывать другие программы и даже самих себя. Например, можно введенное и сформированное в результате вычислений выражение данных проинтерпретировать в качестве программы и непосредственно выполнить (так называемое программирование, управляемое данными).

3. Хранение данных, не зависящее от места
Списки, представляющие программы и данные, состоят из списочных ячеек, расположение и порядок которых в памяти не существенны.

4. Автоматическое и динамическое управление памятью
Программисту не надо заботиться об учете памяти.

5. Лисп — безтиповой язык программирования
В лиспе имена символов, переменных, списков, функций и других объектов не закреплены предварительно за какими-нибудь типами данных.

Основные структуры данных (символы, числа, списки)

1. Символы
Буквы, цифры, знаки
+ - * / @ $ % ^ & _ > < .(точка) ? ! { } [ ] :

2. Числа
целые числа 56, вещественные 67.89, вещественные в научной нотации 9.1E-31, рациональные числа 5/9.

3. Логические значения
T (истина) NIL (ложь)
любой объект отличный от NIL имеет логическое значение истина.

4. Константы и переменные
Константы это числа и логические значения T и NIL. Остальные символы

5. Атомы
Атомы это символы и числа, плюс логические значения.

6. Списки
Список в Лиспе — упорядоченная последовательность, элементами которой являются атомы и списки (подсписки). Списки заключаются в круглые скобки, элементы списка разделяются пробелами.
Пустой список — список в котором нет ни одного элемента, обозначается () или NIL. NIL может быть элементом списками.

Символьное выражение	примечание
NIL	То же самое, что и ()
(NIL)	Список, состоящий из атома NIL
(())	То же самое, что и (NIL)
((()))	То же самое, что и ((NIL))
(NIL ())	Список из двух пустых списков или двух NIL

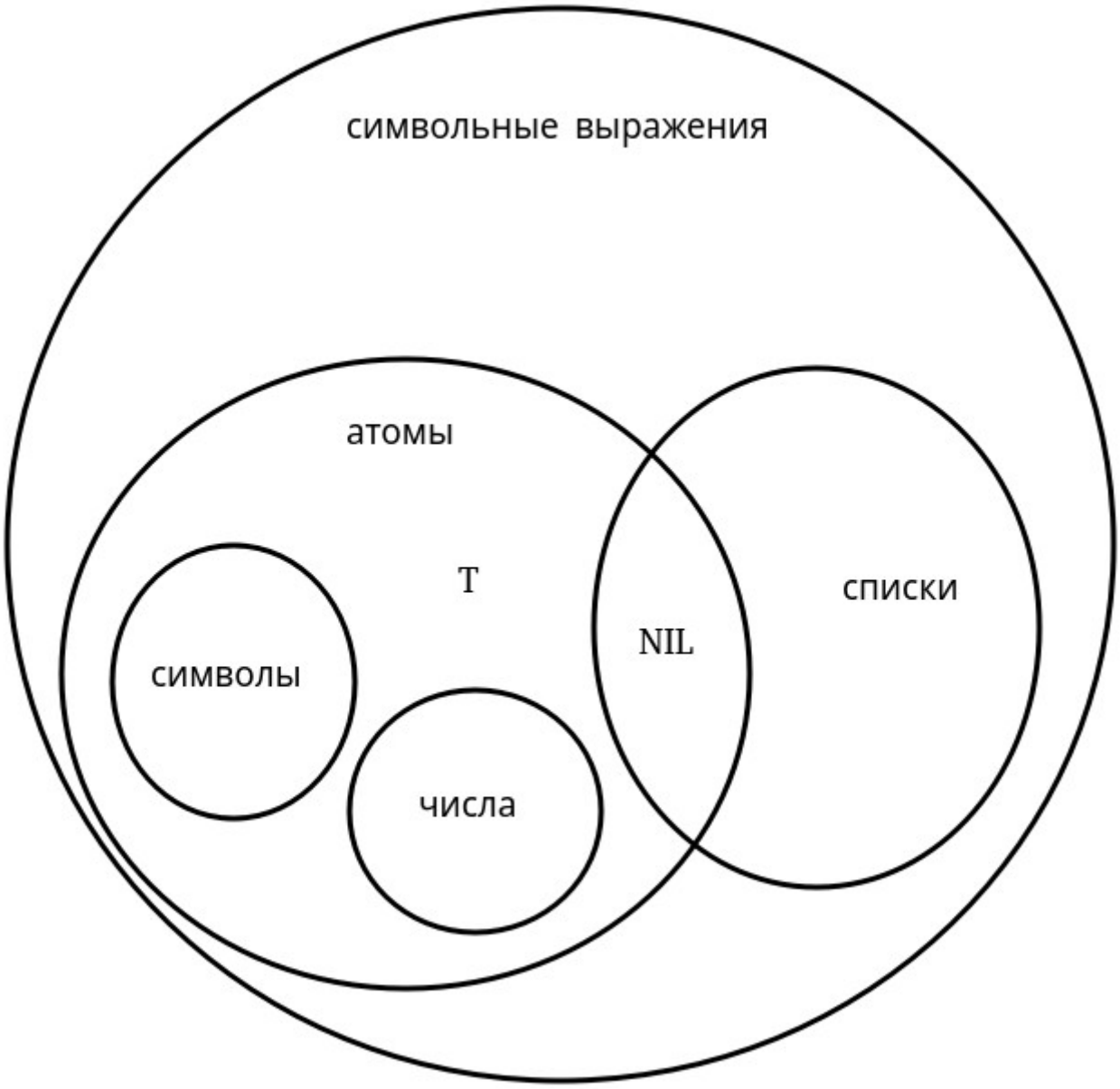

Рис.1 — Символьные выражения.

Список как средство представления знаний, например, информацию о семье можно представить списком с тремя элементами: муж, жена, дети.

```
((husband Tom 45)
 (wife Ann 44)
 (children (Pat woman 22)
           (Bob man 18)))
```

Понятие функции

Мы будем использовать и определять только те фунции Лиспап, которые являются «чистыми», т. е. Не имеют побочных эффектов и значение функции зависит только от

значений её параметров. Т.е. будем учиться «чистому» функциональному программированию.
В отличии от математики и арифметических выражениях, в Лиспе используется единообразная префиксная нотация (табл.2, 3).

Табл. 2 — Примеры записей функций в математике и в Лиспе

Префиксная нотация в математике	**Единообразная префиксная нотация в Лиспе**
f(x)	(f x)
g(x,y)	(g x y)
h(x,g(y z))	(h x (g y z))

Табл.3 — Примеры записей выражений в математике и в Лиспе

Инфиксная нотация в математике	**Единообразная префиксная нотация в Лиспе**
x + y	(+ x y)
x / y	(/ x y)
x * (y + z)	(* x (+ y z))

Блокировка вычислений выражений

В некоторых случаях не надо вычислять значение выражения, а нужно само выражение. Нас, например, может не интересовать значение функционального вызова (+ 2 3), равное 5, а мы хотим обрабатывать форму (+ 2 3) как список. В таких случаях используют специальную функцию QUOTE с одним аргументом. В качестве краткого обозначения для функции QUOTE используют апостроф перед аргументом (табл.4).

Табл.4 — Примеры вызова и результат

Вызов	**Результат**
(quote (+ 2 3))	(+ 2 3)
`(+ 2 `(* 3 5))	(+ 2 (quote (* 3 5)))

Базовые функции

Большинство Лисп-программ можно записать, испоьлзуя только пять простейших функций над символическими выражениями и одну специальную форму (условное выражение):
1. Полиморфный предикат равенства (equal X Y), где X и Y любые символьные выражения, функция возвращает T или NIL.

2. Предикат, проверяющий, является ли символьное выражение X атомом (atom X).
3. Функция (first X) извлекающая первый элемент списка X.
4. Функция (rest X) извлекающая «хвост» списка X.
5. Функция (cons X Y) возвращает список, состоящий из первого элемента X и «хвоста» Y.

Условие (if X Y Z), если X истина, то возвращает Y, иначе Z.

Определение: список это или пустой элемент или последовательность из двух частей: первого элемента и списка.

- - -

Определение: предикат это функция, которая проверяет некоторое условие и возвращает значение логического типа (истина или ложь, T или NIL).

Формат предиката	**действие**
atom <expr>	Проверка, является ли параметр атомом
eq <symb1> <symb2>	Проверка тождественности символов
= <num1> <num2> . . .	Проверка равенства чисел
eql <val1> <val2>	Сравнивает числа одинакового типа, или символы
equal <val1> <val2>	То же что и eql, кроме того, проверка списков
if <expr1> <expr2> <expr3>	условие
null <expr>	Проверка, является ли выражение null
listp <expr>	Проверяет, является ли параметр списком
numberp <expr>	Проверяет, является ли числом
zerop <num>	Выдает инстину, если значение 0 (ноль)
not <expr>	Инвертирует логическое значение выражения
and <expr1> <expr2> . . .	Выдает истину, если все параметры истина
or <expr1> <expr2> . . .	Выдает истину, если хотябы один из параметров истина

Формирование списка из элементов
list <item1> <item2> <item3> . . .

Формат арифметических функций	**действие**
+ <num1> <num2> . . .	сложение
- <num1> <num2> . . .	вычитание
***** <num1> <num2> . . .	умножение
/ <num1> <num2> . . .	деление
sqrt <num>	вычисление квадратного корня
mod <num1> <num2>	остаток от деления
truncate <num>	округление, отбрасывает дробную часть
1+ <num>	Прибавление единицы
1- <num>	вычитание единицы

Определение функций

Программирование на Лиспе сводится к определению необходимых функций, а выполнение программы — к вызову функции с нужными аргументами (фактическими параметрами).
Определение функции:
- дать имя функции
- определить формальные параметры функции
- определить, что должна делать функция (тело функции)
Формат определения функции

```
(defun <имя> (<параметры>)
 <тело функции>
)
```

компонент	синтаксически	смысл
<имя>	атом	Имя функции с помощью которого выполняется вызов функции
<параметры>	список атомов	Список формальных параметров («локальные переменные»), которые имеют значение только внутри функции, инициализируются при вызове, после чего значение поменять нельзя
<тело функции>	символьное выражение	Тело функции, вычисляемое символьное выражение (форма)

Пример: написать функцию вставки символа + между двумя параметрами и формирования списка из этих элементов.

```
(defun insert_plus (s1 s2)
 (list s1 `+ s2)
)
```

Пример вызова:

```
(print (insert_plus 'a 'b))
```

Результат вызова:

```
(A + B)
```

Определение
Форма — символьное выражение, значение которого может быть вычислено интерпретатором:
- константы
- символы, которые используются в качестве переменных
- определения функций и вызовы функций
- специальные формы (quote, if и т. п.)

Пример объявления функции внутри функции

```
(defun insert_plus (s1 s2 s3)
 (defun insert_mul(x1 x2)
```

```
   (list x1 `* x2)
 )
 (insert_mul
  (list s1 `+ s2)
  s3
 )
)
(print (insert_plus 'a 'b 'c))
(print (insert_mul '5 '8))
```
Результат вызова:
```
((A + B) * C)
(5 * 8)
```

т. е. нет смысла объявлять внутри.

Проверка работы функции с несколькими вызовами:
```
(defun test(x1 x2)
  (print "test")
  (+ x1 x2)
  (- x1 x2)
)
(print (test 15 8))
```
результат вызова:
```
"test"
7
```
т. е. работают все вызовы, но результат определяется по последнему вызову.

Некоторые специальные формы

Имя и значение символа.
Символы в Лиспе можно использовать как переменные. В этом случае он могут обозначать некоторые выражения. У символов изначально нет никакого значения как у констант. Попытка вывести значение символа приводит к ошибке.
Пример:
```
(print b)
```
Результат:
```
Error(s), warning(s):
*** - EVAL: variable B has no value
```

При помощи формы **`set`** символу можно присвоить значение.
Формат:
```
(set <expr1> <expr2>)
```
Действие: вычисляется выражение <expr1> если результат является символом, то ему присваивается значение полученное из выражения <expr2>, иначе ошибка.
Пример:
```
(set 'c (* 2 3))
(print c)
```
Результат:
```
6
```

Пример:
```
(defun n2(lst)
  (car (cdr lst)) ;(cadr lst)
)
(set (n2 '(a b c d e)) (+ 2 3))
(print b)
```
Результат:
```
5
```

Форма **setq** тоже самое что и **set**, но 1-й параметр не вычисляется (вычисление блокируется), о чём напоминает буква q в названии функции.
Пример:
```
(setq a `b)
(print a)
```
Результат:
```
B
```

Формы **set** и **setq** имеют побочный эффект — создание связи между символом и значением.
В Лиспе все действия возвращают некоторые значения, в том числе и **set** и **setq** (функции, основное предназначение которых, заключается в осуществлении побочного эффекта). Формы с побочным эффектом в Лиспе можно рассматривать как псевдофункции, так как они всегда возвращают некоторое значение.
Псевдофункции **set** и **setq** представляют оператор присваивания в Лиспе, использование этих псевдофункций нарушает принцип прозрачности по ссылкам. Поскольку мы изучаем чистое функциональное программирование, то формы **set** и **setq** если и будут использоваться, то только для одноразового присваивания значений символам, используемым в качестве глобальных переменных.
Использование псевдофункций **set** и **setq** внутри тела функции несовместимо с чистым функциональным программированием.

Пример использования псевдофункций внутри функций

Объявление функции	**Соответствие парадигме чисто функционального программирования**

`(defun inc1(n)`
`  (setq n (+ n 2))`
`)`
	Не соответствует

`(defun inc2(n)`
`  (+ n 2)`
`)`
	Соответствует

Пример нарушения прозрачности по ссылкам
```
(setq flag T)
(defun f(n)
    (if (setq flag (not flag)) n (* 2 n))
)
```

Вызов	результат
(print (f 3)) (print (f 3)) (print (+ (f 1) (f 2))) (print (+ (f 2) (f 1)))	6 3 4 5

Интерпретатор Лиспа **eval**. Вызов этой функции может снять эффект блокировки вычисления или позволяет найти значение выражения (выражение, которое записано как список).

Вызов	результат
(print (eval '(+ 11 12)))	23

Ветвление cond

Формат:

```
(cond
 (p1 a1)
 (p2 a2)
 . . .
 (pN aN)
)
```

вариант формата:

```
(cond
 (p1 a1)
 . . .
 (pi)
 . . .
 (pN aN)
)
```

Предикатами p*i* и результирующими выражениями a*i* могут быть произвольные формы. Значение предложения cond определяется следующим образом:

- Выражения p*i* , являющиеся предикатами, вычисляются последовательно слева на право (сверху вниз) до тех пор, пока не встретится выражение, значением которого не является NIL, т. е. логическое значение которого является истина.
- Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения cond
- Если истинного предиката нет, то значением cond является NIL

Рекомендуется в качестве последнего предиката использовать T (константа ИСТИНА)

Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения cond при истинности предиката выдается само значение предиката.

Пример:

```
(defun f(x)
    (cond
        ((< x 3) "x<3")
        ((> x 10) "x>10")
        (x)
    )
)
```

Вызов	результат
(print (f -1)) (print (f 0)) (print (f 2)) (print (f 4)) (print (f 9)) (print (f 11)) (print (f 20))	"x<3" "x<3" "x<3" 4 9 "x>10" "x>10"

let — создание локальной связи (используется для объявления локальных переменных внутри функций)

формат

```
(let ((var1 value1) (var2 value2) … )
 (expr)
)
```

Действие: сначала переменным var1, var2, … присваивается значения соответственно value1, value2, … . Затем вычисляется значение expr которое возвратит функция let.

Примечание:

1. После завершения работы функции let, переменные var1, var2 разрушаются
2. Присвоение значений переменным var1, var2, … выполняется одновременно.

flet и labels — объявление локальных функций (функций внутри функций).

Форма labels используется в случае, когда хотя бы одна из локальных функций является рекурсивной.

Форматы:

```
(flet ((fname1 params1 body1)
         (fname2 params2 body2)
         …
       )
      expr
)
```

Аналогично для labels

fname1, fname2, … - имена локальных функций

params1, params2, ... — списки формальных параметров в скобках

body1, body2, ... — тела локальных функций

expr — выражение, содержащее вызов локальной функции fname1, fname2, ...

```
(defun mega_f(x y)
 (flet ((f1 (x y) (+ x y))
        (f2 (x y) (* x y)))
        (/ (f2 x y) (f1 x y))
 )
)
(write (mega_f 3.0 2.0))
```

Результат:

```
1.2
```

Пример:

$$f(x)=\begin{pmatrix} max(x,4)^2+1, & x\leqslant 1 \\ 2\cdot max(x,4)^{2,} & x>1 \end{pmatrix}$$

Наверно, первое, что приходит в голову, следующая реализация:

```
(defun f1(x)
 (if (<= x 1) (+ 1 (* (max x 4) (max x 4))
      (* 2 (max x 4) (max x 4))
 )
)
```

здесь «подфункция» $max(x,4)^2$ вычисляется 1 раза, в обоих случаях, но её запись пришлось сделать два раза, вычисление «подфункции» $max(x,4)$ выполняется 2 раза в обоих случаях

```
(defun f2(x)
 (+ (* (max x 4) (max x 4))
  ( if (<= x 1) 1 (* (max x 4) (max x 4)) )
)
)
```

здесь «подфункция» $max(x,4)^2$ вычисляется 2 раза

```
(defun g(x) (*  (max x 4) (max x 4)) )
(defun f3(x) (+ (g x) (if (<= x 1) 1 (g x)) )
```

здесь «подфункция» $g(x)=max(x,4)^2$ объявлена 1 раз, но вычисляется 2 раза

```
(defun f4(x)
    (if (<= x 1)
             (+ 1 (g x))
             (* 2 (g x))
    )
)
```

аналогично

```
(defun f5(x)
 (let ( (a (* (max x 4) (max x 4)) ) )
      (+ a (if (<= x 1) 1 a))
 )
)
```

Введена локальная переменная **a** здесь «подфункция» $max(x,4)^2$ вычисляется 1 раза, вычисление «подфункции» $max(x,4)$ выполняется 2 раза

```
(defun f6(x)
 (flet ( (a (y) (* (max x 4) (max x 4)) ) )
      (if (<= x 1)
              (+ 1 (a x))
              (* 2 (a x))
      )
 )
)
```

аналогично

```
(defun g1 (x)
 (let ((a (max x 4)))
```

```
 (* a a)
)
(defun f7(x)
 (if (<= x 1)
      (+ 1 (g1 x))
      (* 2 (g1 x))
 )
)
```
«подфункции» $max(x,4)$ выполняется 1 раз

```
(defun f8(x)
 (flet ((g1 (y)
           (let ((a (max y 4)))
             (* a a)
           )
         ))
   (if (<= x 1)
        (+ 1 (g1 x))
        (* 2 (g1 x))
   )
 )
)
```
аналогично

2. Виды рекурсий

прямая, косвенная, линейная
прямая рекурсия — когда функция вызывает саму себя.
Косвенная — когда имеется последовательность вызовов нескольких функций f1, f2, …, fk друг друга: f1 вызывает f2, f2 вызывает f3, … , fk вызывает f1.
Линейная — когда в функции присутствует только один прямой рекурсивный вызов.

```
(even-ones n)
(odd-ones n)

(defun even-ones (n)
 (cond
  ((zerop n) t)
  ( (zerop (mod n 2)) ( even-ones (truncate (/ n 2)) ) )
  ( t ( odd-ones (truncate (/ n 2)) ) )
 )
)


(defun odd-ones(n)
 (cond
  ((zerop n) nil)
  ((zerop (mod n 2)) ( odd-ones (truncate (/ n 2))))
  ( t ( even-ones (truncate (/ n 2)) ) )
 )

)
```
хвостовая

```
(defun fib1(n)
 (cond
  ((= n 1) 1)
  ((= n 2) 1)
  ( t ( + (fib1 (- n 1)) (fib1 (- n 2)) ) )
 )
)

(defun fib2 (n) (f 1 1 1 n))
(defun f (k a b n)
 (if (= k n)
      a
     (f (+ 1 k) b (+ a b) n)
 )
)
```

удаленная

```
ak(0,m) = m+1
ak(n,0)=ak(n-1,1), n>0
ak(n,m)=ak(n-1,ak(n,m-1)), n>0, m>0

(defun ak(n m)
  (cond
    ( (zerop n) (+ 1 m) )
    ( (zerop m) (ak (- n 1) 1) )
    (t (ak (- n 1) (ak n (- m 1)) ) )
  )
)
```