Язык Лисп

Язык Лисп – один из старейших языков программирования и первый *функциональный язык*, получивший широкое распространение. Ядро языка было создано в 60-х годах прошлого века известным ученым Дж. Маккарти для решения задач обработки символьной информации .
Основная структура данных языка Лисп – список, отсюда и название языка: *Lisp – List Processing*.

Атомы и списки, функции

Обрабатываемые в языке Лисп данные можно разделить на скалярные (простые) и структурные (составные). К скалярным данным относятся ***атомы*** (атом – неделимое целое), а к структурным – ***списки***, или списочные структуры. В свою очередь атомы подразделяются на:

-- *символьные*, т.е. идентификаторы (например: `BETA` , `A` , `SYM_1`);

-- *числовые*, т.е. целые и вещественные числа (`-25`, `375.08` и др.).

Атомы и списки, функции

Среди множества атомов базового Лиспа ***константами*** являются:
-- все числовые атомы;
-- символьные атомы **T** и **NIL**, обозначающие соответственно логические значения *истина* и *ложь*; а атом **NIL** также обозначает *пустой список*

Атомы и списки, функции

Ключевым понятием языка Лисп является ***список*** – рекурсивная структура, которая может быть описана следующими БНФ (металингвистическими формулами Бэкуса-Наура):

список ::= (*последовательность_элементов*) |
пустой список

последовательность_элементов ::= *элемент* |
элемент ␣ *последовательность_элементов*

элемент ::= *атом* | *список*

пустой список ::= () | NIL

Атомы и списки, функции

Примеры списков языка Лисп:

(BETA 56 (9)) ()

((C ()) NIL 81 (EE B C)) ((((T))))

Атомы и списки, функции

Глубиной списка считается максимальное количество вложенных пар скобок. Так, на верхнем уровне списка (BETA 56 (9)) находятся три элемента: символьный атом BETA, число 56 и вложенный список (9), а глубина этого списка равна 2. Единственной константой составного типа является пустой список (), который может быть записан и как атом NIL, обе эти записи эквивалентны.

Атомы и списки, функции

Лисп-программа представляет собой последовательность форм, а *форма*, или *вычислимое выражение* – это атом или список, который можно вычислить и получить значение.

Вычисление программы реализует ***лисп-интерпретатор***, который считывает очередную входящую в программу форму, вычисляет её и выводит полученный результат (атом или список).

Атомы и списки, функции

Вычислимые атомы и списки:

для чисел: 72 => 72

для символьных атомов: t => T

() => NIL

NIL => NIL

Символьный атом вычислим только тогда, когда он представляет собой имя формального параметра (аргумента) некоторой функции, и этот параметр имеет значение, например: x => (B C D).

Атомы и списки, функции

Список можно вычислить, если он представляет собой обращение к функции, или *функциональный вызов*: $(f\ e_1\ e_2 \ldots e_n)$,
где f – символьный атом, обозначающий имя вызываемой функции, а $e_1, e_2, \ldots, e_n$ – аргументы этой функции, $n \geq 0$.
n - число аргументов функции.
В случае n=0 имеем вызов функции без аргументов: (f). Обычно $e_1, e_2, \ldots, e_n$ являются вычислимыми выражениями и вычисляются последовательно слева направо.

Атомы и списки, функции

Функции Лиспа обычно делятся на:
-- *встроенные*, или стандартные, которые могут применяться без определения;
-- *определяемые* пользователем в его программе.

Другое важное в Лиспе деление функций на классы учитывает количество и вычислимость аргументов функции. Различают:

обычные функции (их большинство);

особые, или специальные функции.

Атомы и списки, функции

Обычная функция имеет строго фиксированное число аргументов, и при вычислении её значения интерпретатор сначала вычисляет значения её аргументов слева направо, а уже затем функция применяется к вычисленным значениям.

У *особой* функции нарушается хотя бы одно из двух указанных требований, т.е. у неё может быть произвольное количество аргументов и/или некоторые её аргументы могут не вычисляться.

Базовый набор функций

Функция **car** от одного аргумента возвращает первый элемент списка, являющегося значением её аргумента.

Функция **cdr** возвращает *хвост* списка, являющегося значением её единственного аргумента (хвостом, или остатком списка является список без своего первого элемента). *Contents of Adress Register* и *Contents of Decrement Register*.

Функция **cons** от двух аргументов (cons e_1 e_2) строит новый список, первым элементом которого является значение первого аргумента e_1, а хвостом – значение второго аргумента e_2. По сути, эта функция включает заданный элемент (значение выражения e_1) в начало списка, являющегося значением e_2.

Базовый набор функций

Пусть переменная x имеет значение (GAMMA (15)), а переменная y имеет значение (TETA):

x => (GAMMA (15)) y => (TETA)

Тогда

(car x) => GAMMA

(cdr x) => ((15))

(cdr y) => NIL

(car (cdr x)) => (15)

(car(car(cdr x))) => 15

(cons x y) => ((GAMMA (15)) TETA)

(cons y ()) => ((TETA))

Базовый набор функций

Функция **atom** вырабатывает значение T, если значением её единственного аргумента является атом (числовой или символьный). В противном случае возвращается NIL.

Функция-предикат **eq** (сокращение от англ. *equal*) проверяет совпадение двух своих аргументов-атомов, вырабатывая значение T, когда:
1) значением одного из аргументов является атом, и одновременно
2) значения аргументов равны (идентичны).
В ином случае значением функции eq является NIL. Будем считать, что когда оба аргумента функции eq – списки, её результат неопределён. Для чисел есть отдельная ф-я.

Базовый набор функций

x => (GAMMA (15))

(atom NIL) => T
(atom T) => T
(atom x) => NIL
(atom ()) => T
(atom (cdr (cdr x))) => T
(eq (atom 5) T) => T
(eq (car x)(cdr x)) => NIL

Базовый набор функций

Особая функция **quote**, которая в качестве своего значения выдаёт сам аргумент, не вычисляя его:
(quote e) => e
(cons 5 (quote(A T))) => (5 A T)

(quote (ATOM B)) => (ATOM B)
'(ATOM B) => (ATOM B) -- другое обозначение quote
(cons (quote(A (B))) NIL) => ((A (B)))
(cons '(A (B)) NIL) => ((A (B)))
(atom '(NIL)) => NIL

Базовый набор функций

Функция базового набора **eval** выполняет двойное вычисление своего аргумента. Эта функция является обычной, и первое вычисление аргумента выполняет так же, как и любая обычная функция. Полученное при этом выражение вычисляется ещё раз.

(eval (quote (atom b))) => T
(eval (quote (quote quote))) => QUOTE
(eval '(car '(a b c))) => A
(eval (cons (quote car) (quote (x)))) => GAMMA
(car x)

Базовый набор функций

Функция базового набора **cond** (сокращение от англ. *condition* – условие) служит средством разветвления вычислений. В строго функциональном программировании вызов этой функции, как правило, имеет вид

(cond (p_1 e_1) (p_2 e_2) … (p_n e_n)) , $n \geq 1$.

Обращение к функции cond называется ***условным выражением***, выражения (p_i e_i) – ***ветвями*** условного выражения, а выражения-формы p_i – ***условиями*** ветвей. Функция cond является особой, поскольку в условном выражении может быть произвольное количество ветвей, и не все формы e_i вычисляются в общем случае.

Базовый набор функций

```
(cond ((eq 'A T) 'are_equal)
    (T 'not_equal))                    =>    NOT_EQUAL

(cond ((eq 'A1 'A2) 'are_equal)
    ('not_equal))                      =>    NOT_EQUAL
```

Здесь выражение e_2 опущено, его роль играет p_2

Базовый набор функций

Пример. Сложение по модулю 2:

```
(cond (x (cond (y NIL)
                    (T)
               )
          (y)
)
```

Эта форму можно упростить так:

```
(cond (x (eq y NIL)) (y))
```

```
(cond (NIL (eq NIL NIL)) (NIL))  =>  NIL.
```

Базовый набор функций

Пример. AND и OR
(cond (x y))
(cond (x) (y))

Определение функций

лямбда-выражение

лямбда-выражение ::= (lambda *лямбда-список* *тело_функции*)

Лямбда-список представляет собой лисповский список из символьных атомов, рассматриваемых как имена *формальных параметров* функции. В частном случае этот список может быть пустым, что соответствует функции без аргументов.

Телом функции является некоторое лисповское выражение (форма), в которое в общем случае входят заданные в лямбда-списке формальные параметры. Тело функции служит для вычисления её значения.

Определение функций

Приведём пример лямбда-выражения с двумя параметрами x и y:

(lambda (x y) (cond (x) (y)))

и лямбда-выражения с одним параметром x:

(lambda (x) (cond (x) (y)))

Отличие этих выражений в том, что в последнем случае переменная y не внесена в число параметров функции.

Определение функций

Пример безымянной функции, вычисляющей дизъюнкцию двух своих аргументов (при расширенном понимании истинного значения):
(lambda (x y) (cond (x) (y)))

Лямбда-вызов является формой следующей структуры:
((lambda (x_1 x_2 … x_k) e) p_1 p_2 … p_k)
где $k \geq 0$,
p_1, p_2 … p_k – произвольные выражения-формы (фактические параметры),
x_1, x_2 … x_k – символьные атомы (формальные параметры).

Определение функций

Пример

((lambda(x y)(cond (x) (y))) 'A 3) => A

Определение функций

((lambda (x_1 x_2 … x_k) e) p_1 p_2 … p_k)
этапы вычисления этой формы при $k \geq 1$:
Вычисление аргументов: последовательно вычисляются фактические параметры p_1, p_2, … p_k лямбда-вызова, пусть их значения – v_1, v_2, …, v_k .
Связывание формальных параметров: формальные параметры x_1, x_2, … x_k попарно связываются (на время вычисления лямбда-вызова) соответственно со значениями v_1, v_2, …, v_k фактических параметров лямбда-вызова:
$x_1=v_1$, $x_2=v_2$, …, $x_k=v_k$.

Определение функций

Вычисление тела: вычисляется форма e (тело функции), причём всюду, где необходимо вычислить x_i, в качестве его значения берется v_i. Вычисленное таким образом значение формы e служит итоговым значением лямбда-вызова.

Определение функций

Вычисление тела: вычисляется форма e (тело функции), причём всюду, где необходимо вычислить x_i, в качестве его значения берется v_i. Вычисленное таким образом значение формы e служит итоговым значением лямбда-вызова.

В частном случае k=0 вычисление лямбда-вызова без аргументов: ((lambda () e)) сводится к вычислению формы e и выдаче полученного результата в качестве значения этого лямбда-вызова.

Определение функций

Для неоднократного применения функции (а также для построения рекурсивной функции) требуется средство её именования – для этого служит особая встроенная функция defun, обращение к которой обычно имеет вид:
(defun *имя_функции лямбда-список*
тело_функции)

В качестве имени функции выступает символьный атом. Значением этой формы является имя определяемой функции:
(defun f (x_1 x_2 … x_k) e) => F, $k \geq 0$

Определение функций

```
(defun List2 (x y) (cons x (cons y ()))) => LIST2

(List2 '(A) 'B) => ((A) B)
(List2 'Y  'X) => (Y X)
(List2 T '(C (E))) => (T (C (E)))


(defun InsNew(x)
    (cons(car x)(cons 'NEW (cdr x)))) => INSNEW

(InsNew '(F (T E ())C)) => (F NEW (T E NIL) C)
```

Определение функций

Лисповский список является функциональным вызовом только в двух случаях:
Первый случай – обращение к функции по имени: (f e_1 … e_k), k≥0; этот случай включает обращение к функции let.
Второй случай – лямбда-вызов, или обращение к безымянной функции: (λ e_1 … e_k), где λ – лямбда-выражение, k≥0.
Подчеркнём, что первый элемент списка-вызова функции не вычисляется лисп-интерпретатором, он представляет собой либо явно заданное имя функции, либо явно заданное определяющее выражение функции.

Встроенные функции

Функции обработки списков:
28 обычных функций от одного аргумента caar, cadr, cdar, cddr, caaar, caadr, …, cdddar, cddddr, действие которых эквивалентно определённой суперпозиции функций car и cdr

(caddr '(P (C E) B ())) => B ,
(caadr '(P (C E) B ())) => C .

Встроенные функции

Функция-конструктор list, составляющая список из значений своих аргументов. Эта функция относится к особым, поскольку у неё может быть произвольное число аргументов, но при этом все аргументы вычисляются.

(list '(NUM К) ()'C) => ((NUM К) NIL C)
(list '(B) 'A) => ((B) A)
(list '(A) '(B)) => ((A)(B)) ,
но (cons '(A) '(B)) => ((A) B).

Встроенные функции

Арифметические функции

(+ 12 -67 34) => -21
(* 1 2 3 4) => 24
(- -15 -3) => -12
(- -56) => 56
(/ -12 3) => -4

Встроенные функции

Арифметические предикаты

(= 1/4 0.25) => T
(/= 1/4 (- 0.3 0.05)) => NIL
(< 7 (+ 12 3)) => T
(<= 7 (+ -12 3)) => NIL

(evenp 25) => NIL
(evenp (+ -15 -3)) => T

Встроенные функции

```
(defun eql (x y)(cond ((numberp x)
                       (cond ((numberp y)(= x y))
                             (T NIL)))
                      (T (eq x y))))
```

Встроенные функции

Предикаты типа

```
(defun null (x) (eq x NIL))

(null '(M))  =>  NIL
(null (cdr '(M)))  =>  T
```

Функция-предикат listp выдает значение T, если значением её аргумента является список, и NIL в противном случае.

```
(listp 'A )  => NIL
(listp (car '((A)) )) => T
(listp (caar '((A)))) => NIL        (listp (cdr '((a)))) => T
```

Встроенные функции

Предикаты типа

(numberp (car '(12 A S))) => T
(numberp (cadr '(12 A S))) => NIL
(symbolp (car '(12 A S))) => NIL
(symbolp (cadr '(12 A S))) => T
(symbolp '(12 A S)) => NIL
(numberp '(12 A S)) => NIL
(symbolp ()) => T

Встроенные функции

Логические функции

(defun not (x) (eq x NIL))

(not NIL) => T
(not T) => NIL

(not '(B ())) => NIL

(and e_1 e_2 … e_n), n≥0.
(or e_1 e_2 … e_n), n≥0

Встроенные функции

При n=0 значения функций: (and)=>T, (or)=>NIL.

Примеры :

(and (atom T)(= 1 23)(eq 'A 'B)) => NIL

(and (< 12 56) (atom 'Z) '(A S)) => (A S)

(or (eq 'A 'B) (atom 'Y) ()) => T

(or (eq 'A 'B) 'Y '(T R)) => Y

(or (atom '(Y V)) () (eq 'A 'B)) => NIL

Рекурсивные функции

Напишем функцию, вычисляющую длину списка

(Length '(A (5 6) D)) => 3

Рекурсивные функции

Напишем функцию, вычисляющую длину списка

(Length '(A (5 6) D)) => 3

```
(defun Length (L)
      (cond ((null L) 0)
```

Рекурсивные функции

Напишем функцию, вычисляющую длину списка

(Length '(A (5 6) D)) => 3

```
(defun Length (L)
      (cond ((null L) 0)
            (T (+ 1 (Length (cdr L))) )))
```

Рекурсивные функции

В качестве следующей задачи запрограммируем функцию-предикат Member от двух аргументов A и L: (Member A L)

Рекурсивные функции

В качестве следующей задачи запрограммируем функцию-предикат Member от двух аргументов A и L: (Member A L)

```
(defun Member (A L)
  (cond ((null L) NIL)
        ((eq A (car L)) T)
        (T (Member A (cdr L)) )))
```

Рекурсивные функции

Ещё одна задача – построение рекурсивной функции Append от двух аргументов-списков L1 и L2. Функция соединяет (сливает) элементы верхнего уровня обоих списков в один список, например:

(Append '(Q R T) '(K M)) => (Q R T K M)

(Append '(B (A)) '((C C) () D))

=> (B (A) (C C) NIL D)

Рекурсивные функции

```
(defun Append(L1 L2)
  (cond ((null L1) L2)
        (T (cons (car L1)(Append (cdr L1) L2)))))
```

Рекурсивные функции

```
(defun Append(L1 L2)
  (cond ((null L1) L2)
        (T (cons (car L1)(Append (cdr L1) L2)))))
```

Задание

Написать функцию RevAppend, меняющую порядок элементов в сцепленных списках на противоположный

Лисп-программа

Типичная лисп-программа включает:
определения новых функций на базе встроенных функций и других функций, определённых в этой программе;
вызовы этих новых функций для конкретных значений их аргументов.

(read) => атом или спсок -- функция ввода
(prin1 e) -- вычисляет e и печатает результат

Рекурсивное программирование

(Reverse '(A (B D) C)) => (C (B D) A)

```
(defun Reverse (L)
      (cond ((null L) NIL)
            (T (append (Reverse (cdr L))
                       (cons (car L) NIL) )) ))
```

Рекурсия может быть косвенной

Рекурсивное программирование

(Reverse '(A (B D) C)) => (C (B D) A)

```
(defun Reverse (L)
      (cond ((null L) NIL)
            (T (append (Reverse (cdr L))
                       (cons (car L) NIL) )) ))
```

Рекурсия может быть косвенной

Рекурсивное программирование

```
( (defun MemberS (A L)
 (cond((atom L)(eql A L)) ; дошли до атома
          (T(or(MemberS A(car L));поиск в левом
поддереве
               (MemberS A(cdr L));поиск в правом
поддереве
)) ))
```

Ищет элемент A в списке L. Параллельная рекурсия