ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

Лекция 2. Описание функций в Лиспе. Рекурсивные функции.

Лямбда-исчисление как основа определения функций.

Определение функций и их вычисление в Лиспе основано на лямбда-исчислении Черча. В предложенном Черчем исчислении функция записывается в следующем виде :

lambda(x1, x2, … ,xn).fn

В Лиспе исчисление Черча заимствовано для определение вычислений и описания параметров функций :

(lambda (*x1 x2 … xn*) *fn*)

Символ lambda означает, что мы имеем дело с определением производимых функцией действий. Символы *xi* являются формальными параметрами определения, которые именуют аргументы в описывающем вычисления теле функции *fn*. Телом функции является произвольная форма, значение которой может вычислить интерпретатор Лиспа.

Формальность **параметров означает, что их можно заменить на любые другие символы, и это не отразится на определяемых функцией действиях.**

Неименованные функции Лиспа.

Определение 1. **Лямбда-выражение – это определение вычислений и параметров функций в чистом виде без фактических параметров :**

(lambda (<список формальных параметров>) <тело функции>).

Для применения описанной таким образом функции к некоторым аргументам *a1, … ,an* необходимо в вызове функции поставить лямбда-выражение на место имени функции :

(< лямбда-выражение > *a1 a2 … an*).

Такую форму вызова называют лямбда-вызовом.

Если вычислять значения аргументов не нужно, то такую функцию нужно определять с помощью nlambda – выражения (от англ. No-spread lambda) :

(nlambda (<список формальных параметров>) <тело функции>)

nlambda – функции применяются при разработке новых синтаксических форм для расширения языка, а также при реализации интерпретаторов проблемно-ориентированных языков с лиспоподобной структурой.

Порядок вычисления лямбда-вызовов.

1. **Вычисляются значения фактических параметров.**
2. **Выполняется связывание параметров : первый формальный параметр связывается с вычисленным значением первого фактического, второй формальный параметр связывается с вычисленным значением второго фактического и т.д.**
3. **Выполняется тело лямбда-выражения. Полученное значение возвращается в качестве значения всего лямбда-вызова.**
4. **Разрывается связь между формальными и фактическими параметрами.**

Примечание. LAMBDA и NLAMBDA – вызовы могут быть вложенными.

Примеры неименованных функций GCLisp’a.

((lambda (x y) ((atom x) y) ((atom y) x) (cons x y)) (car ‘((a b c) d))(cadr ‘(1 2 3 4)))	((nlambda (x y) ((atom x) y) ((atom y) x) (cons x y)) (car ‘((a b c) d))(cadr ‘(1 2 3 4)))

X→(a b c) Y→2 Результат : (a b c)	X →(car ‘((a b c) d)) Y →(cadr ‘(1 2 3 4)) Результат : (car ‘((a b c) d)) cadr ‘(1 2 3 4))

Вложенные лямбда-вызовы.

((lambda (x) ((lambda (y) (cons x y)) (cdr '(a b c)))) (car '((d e f))))	((nlambda (x) ((lambda (y) (cons x y)) (cdr '(a b c)))) (car '((d e f))))
X→(d e f) Y→(b c) Результат : ((d e f) b c)	X→ (car '((d e f))) Y → (cdr '(a b c)) Результат : ((car '((d e f))) cdr '(a b c))

Описание неименованных функций в newLISP-tk.

В общих чертах синтаксис описания неименованных функций в newLISP-tk сходен с описанием аналогичных конструкций в GCLisp'e. Основное отличие касается описания функций в целом и заключается в обязательности объявления управляющих структур if и cond в случае наличия разветвлений в структуре преобразования (вычисления).

<u>Пример.</u> Описать функцию, которая возвращает в качестве результата y в случае атомарного x и список '(x y) - в противном случае .

Реализация на GCLisp’e :

```
((lambda (x y)
   ((atom (car x))
   y) (cons x y))
 (car '((f) g h)) '(r t y))
```

Реализация на newLISP-tk :

```
((lambda (x y)
  (if (atom? (first x))
  y) (cons x y))
 (first '((f) g h)) '(r t y))
```

Именованные функции.

Лямбда-выражение есть не имеющий имени механизм, соответствующий используемому в алгоритмических языках определению функции. Для многократного вызова одной и той же функции, но с различными фактическими параметрами функцию необходимо именовать подобно именованию данных посредством функции SET. Дать имя и определить новую функцию в GCLisp'е можно с помощью функции DEFUN :

(defun <*имя*> <*лямбда-выражение*>)

Вызов именованной функции :

(<*имя*> <*список фактических параметров*>)

Пример описания и вызова именованной функции в GCLisp'е :

$$f(x,y)=\begin{cases} y, \text{ если } x - \text{атом} \\ x, \text{ если } y - \text{атом} \\ (x\ y) - \text{иначе} \end{cases}$$

```
(defun f1 (list1 list2)
 ((lambda (x y)
  ((atom x) y)
  ((atom y) x)
  (cons x y))
  (car list1)(cadr list2)))
```

Вызов функции :

(f1 '(1 2) '(3 4))

Результат : 4

Современная сокращенная нотация записи именованных функций.

Функция DEFUN в GCLisp'е соединяет символ имени функции с лямбда-выражением, после чего символ начинает представлять (именовать) определенные этим лямбда-выражением вычисления.

В современной нотации для удобства исключены внешние скобки лямбда-выражения и сам символ LAMBDA :

(defun <*имя*> (<*список формальных параметров*>) <*тело функции*>)

Пример (сокращенная запись для f1)

```
: (defun f2 (list1 list2)
 ((atom (car list1))(cadr list2))
 ((atom (cadr list2)) (car list1))
 (cons (car list1) (cadr list2)))
```

Описание именованных функций в newLISP-tk.

В общих чертах синтаксис именованной функции имеет следующий вид :

(define (<имя функции> <список формальных параметров>)

<тело функции>)

Пример записи функций f1 и f2 в newLISP-tk :

```
(define (f1_new list1 list2)
  ((lambda (x y)
     (cond ((atom?
        x) y) ((atom?
        y) x)
        (true (cons x y))
     )
   )
   (first list1)(first (rest list2))
  )
)
```

```
(define (f2_new list1
   list2) (cond
      ((atom? (first list1)) (first (rest list2)) )
      ((atom? (first (rest list2))) (first list1))
      (true (cons (first list1) (first (rest list2))
))
   )
)
```

Функции и ветвление.

Для разветвления вычислений в Лиспе служит функция COND. Синтаксическая форма COND позволяет управлять вычислениями на основе определяемых предикатами условий.

Структура условной формы в GCLisp'e :

(cond (*p1 a1*)

(*p2 a2*)

. . .

(*pN aN*)

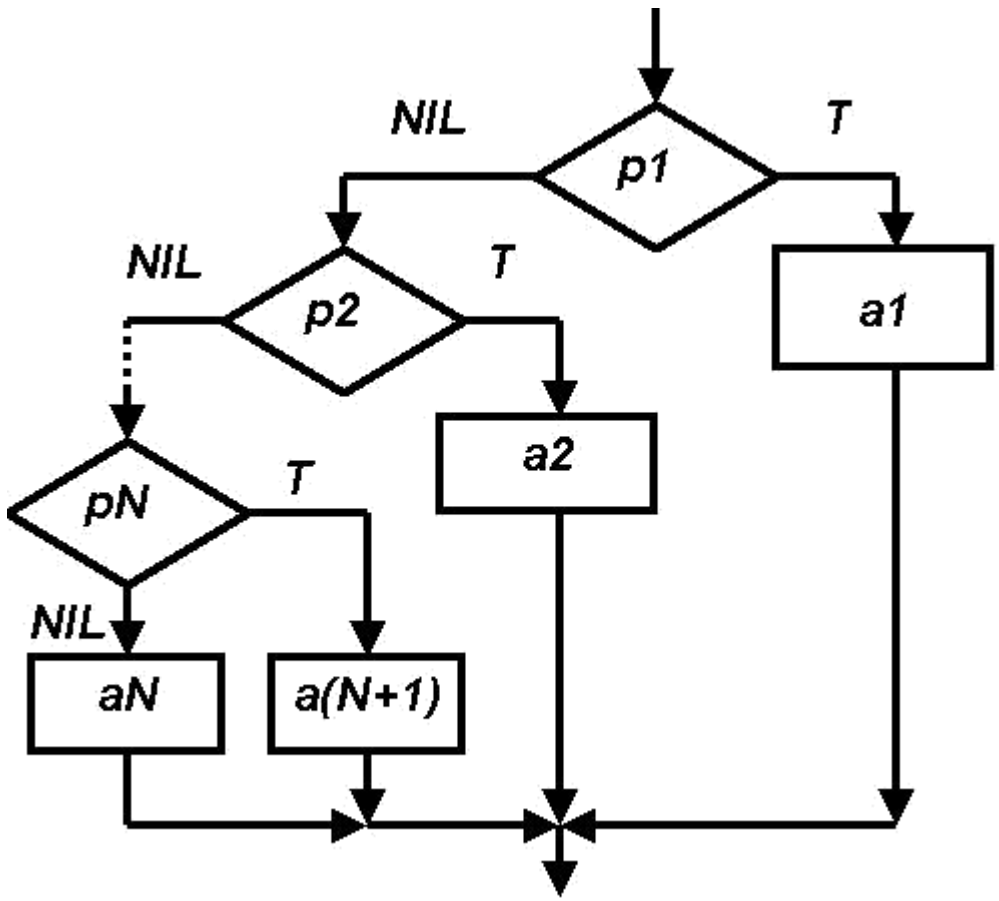
a(N+1)

)

***Примечание.**

Предикатами *pi* и результирующими выражениями *ai* могут быть произвольные формы.

Алгоритмический аналог :



Порядок вычисления условной формы.

1. **Предикативные выражения *pi* последовательно вычисляются сверху вниз до тех пор, пока результатом вычисления одного из *pi* не будет *T*.**
2. **Вычисляется результирующее выражение ai и полученное значение возвращается в качестве значения всего COND-предложения.**
3. **Если среди значений *pi* нет истинных и в теле условия отсутствует результирующее выражение, соответствующее ветви “иначе”, то значением COND будет NIL.**

Пример описания на GCLisp’е функции, содержащей ветвление.

Опишем на GCLisp’е функцию f2 из предыдущего примера, но с применением ветвления .

$$f(x,y)=\begin{cases} y, \text{если } x - \text{атом} \\ x, \text{если } y - \text{атом} \\ (x\ y) - \text{иначе} \end{cases}$$

В нашем примере : X⇔(car list1), Y⇔(cadr list2)

Вызов функции : (f3 ‘(1 2) ‘(3 4))

Результат : 4

Описание на Лиспе : (defun f3 (list1 list2) (cond

((atom (car list1)) (cadr list2)) ((atom (cadr list2)) (car list1)) (cons (car list1)(cadr list2))

)

)

Понятие рекурсии.

Определение. **Функция называется рекурсивной, если в ее теле содержится вызов самой этой функции.**

Принято различать рекурсию по значению и рекурсию по аргументам.

1. случае *рекурсии по значению* рекурсивный вызов определяет результат функции. Пример – принадлежность объекта списку.

2. случае *рекурсии по аргументам* в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции. Пример – нахождение суммы элементов списка.

Рекурсия называется *простой*, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.

Основные правила построения рекурсивных функций.

- Определить количество и вид аргументов.
- Определить характер результата.
- Задать как минимум одно условие окончания рекурсии.
- Определить формирование результата функции.
- Описать формирование новых значений аргументов для рекурсивного вызова.

Пример 1 : суммирование элементов списка (muLISP).

Количество аргументов : 1 – список.

Результат : число.

Условие окончания рекурсии – пустой список

: ((null list) 0)

Формирование результата функции :

(+ (car list) сумма хвоста)

Сумма хвоста есть результат рекурсивного вызова проектируемой функции с хвостом списка в качестве аргумента.

Описание функции :

```
(defun sum (list)
   ((null list) 0)
   (+ (car list)(sum (cdr list))))
```

Результат (sum ‘(9 7 5 6 4)) есть 31.

Реализация функции суммирования элементов списка в newLISP-tk.

```
(define (sum_new lst)
 (cond
   ((null? lst) 0)
   (true (+ (first lst) (sum_new (rest lst))
         )
   )
 )
)
```

Пример 2 : принадлежность списку (muLISP).

Количество аргументов : 2 – произвольное s-выражение (obj) и список (lst).

Результат : значение T (истина), либо NIL

(ложь). Условий окончания рекурсии два :

((null lst) nil) – пустой список, искомый объект не

найден, ((equal obj (car lst)) T) – объект найден.

В случае несовпадения искомого объекта и головы списка результирующее значение вычисляется путем вызова проектируемой функции с хвостом списка в качестве второго аргумента. Полное описание функции :

```
(defun member (obj lst)
((null lst) nil)
((equal obj (car lst)) T)
(member obj (cdr lst)))
```

Результат (member 7 ‘(1 2 3 4))

есть nil.

Результат (member 3 ‘(1 2 3 4))

есть T.

Реализация функции принадлежности элемента списку в newLISP-tk.

```
(define (my_member obj
 lst) (cond
   ((null? lst) nil)
   ((= obj (first lst)) true)
   (true (my_member obj (rest lst)) )
 )
)
```

Пример 3 : объединение списков (muLISP).

Вариант 1.

Количество аргументов : 2 – списки lst1 и

lst2. Результат : список.

Условий окончания рекурсии три :

((and (null lst1)(null lst2)) nil) - оба списка

пустые, ((null lst1) lst2) – первый список пустой,

((null lst2) lst1) – второй список пустой.

Результат функции строится путем присоединения головы первого списка в качестве головы к результату вызова проектируемой функции с хвостом первого списка в качестве первого аргумента и вторым списком в качестве второго аргумента.

Описание первого варианта функции объединения списков.

(defun append (lst1 lst2)

((and (null lst1)(null lst2))

nil) ((null lst1) lst2)

((null lst2) lst1)

(cons (car lst1)(append (cdr lst1) lst2)))

Результат (append ‘(1 2 3 4) ‘(5 6 7 8))

есть (1 2 3 4 5 6 7 8)

Рассмотрим варианты более компактной записи условия окончания рекурсии.

Объединение списков : второй вариант.

Количество аргументов : 2 – списки lst1 и

lst2. Результат : список.

Условий окончания рекурсии два : ((null

lst1) lst2) – первый список пустой, ((null

lst2) lst1) – второй список пустой.

Результат функции формируется аналогично варианту

1. Полное описание функции :

```
(defun append (lst1
 lst2) ((null lst1) lst2)
 ((null lst2) lst1)
 (cons (car lst1)(append (cdr lst1) lst2)))
```

Объединение списков : третий вариант.

Количество аргументов : 2 – списки lst1 и

lst2. Результат : список.

Условие окончания рекурсии : ((null lst1) lst2).

Результат функции формируется аналогично варианту 1.

Полное описание функции :

```
(defun append (lst1 lst2)
((null lst1) lst2)
 (cons (car lst1)(append (cdr lst1) lst2)))
```

Реализация функции объединения списков (третий вариант) в newLISP-tk.

```
(define (my_append lst1 lst2) (cond
   ((null? lst1) lst2)
   (true (cons (first lst1) (my_append (rest lst1) lst2) )
   )
 )
)
```