

Функции высших порядков.

Лекция 6.

Специальности : 230105, 010501

Различие между данными и функциями.

Функции, рассмотренные нами ранее, относятся к функциям первого порядка, поскольку их аргументы и значения относятся по своему типу к данным, то есть трактуются как данные.

Следует отметить, что данные и программы в Лиспе представляются одинаково, а различие между понятиями “данные” и “функция” определяется не на основе их структуры, а в зависимости от их использования.

Если аргумент используется в функции лишь как объект, участвующий в вычислениях, то мы имеем дело с обыкновенным аргументом, представляющим данные. Если же он используется как средство, определяющее вычисления, например, выступая в роли лямбда-выражения, то мы имеем дело с функцией.

Пример (muLISP) :

`(car '(lambda (x)(list x)))` **➔** `LAMBDA`

`((lambda (x)(list x)) car)` **➔** `(car)`

Понятие функционала.

Определение 1. Аргумент, значением которого является функция, называют в функциональном программировании функциональным аргументом.

В роли функционального аргумента может выступать :

- имя функции, с которым связано описание
- Лямбда-выражение `'(lambda (<список формальных параметров> <тело лямбда-выражения>)`
- Всякий лисповский объект, значением которого является функция.

Пример для muLISP (не реализуется в newLISP-tk) :

результатом вызова `(list 'lambda '(x)(list 'list 'x))` будет лямбда-выражение : `(lambda (x) (list x))`

В newLISP-tk данный пример может быть реализован с помощью макроопределения : `(define-macro (my-lambda x) (list x))`, которое транслируется в выражение : `(lambda-macro (x) (list x))`.

Другой вариант : `(fn (x)(list x))`, результатом вызова будет `(lambda (x) (list x))`

Определение 2. Функционалом называется функция, аргумент которой может быть интерпретирован как функция.

Виды функционалов.

Аргументом функции может быть функция, однако, функция может быть и результатом. Такие функции называют функциями с функциональным значением.

Определение 3. Функционалом с функциональным значением называется функционал, вызов которого возвращает в качестве результата новую функцию. Причем в построении этой функции могут использоваться функции, получаемые функционалом в качестве аргументов.

Определение 4. Аппликативным или применяющим функционалом называется функция, которая позволяет применять функциональный аргумент к его параметрам.

В Лиспе имеется 3 применяющих функционала : `apply`, `funcall` и `eval`, из которых `funcall` не реализован в `newLISP-tk`.

`APPLY` применяет функцию к списку аргументов.

`FUNCALL` вызывает функцию с аргументами.

Применяющие функционалы.

APPLY есть функция двух аргументов, из которых первый представляет собой функцию, которая применяется к элементам списка – второго аргумента : (apply <функция> <список>).

Пример : (apply '+ '(2 3)) дает в качестве результата 5.

FUNCALL по своему действию аналогичен **APPLY**, но аргументы для вызываемой функции он принимает не списком, а по отдельности : (funcall <функция> <арг1> ... <аргN>).

Примеры : (funcall '+ 2 3) дает в качестве результата 5.

(funcall '+ '(2 3)) дает в качестве результата (2 3).

Различие между **APPLY** и **FUNCALL** состоит в обязательности списочного представления аргументов у **APPLY**. **FUNCALL** аналогичен по действию **APPLY**, но аргументы для вызываемой функции принимаются не списком, а по отдельности.

Следует отметить, что функциональным аргументом может быть только “настоящая” функция. Специальные формы, такие как **QUOTE**, **SETQ** и макросы для этих целей не подходят.

Примеры использования применяющих функционалов.

Задача 1. Написать функционал, выполняющий действие над каждым элементом списка и объединяющий результаты в список.

; Описание функционала в muLISP:

```
(defun mapping (fun lst)
  ((null lst) nil)
  (cons (funcall fun (car lst))
        (mapping fun (cdr lst))))
```

; Функция увеличения элемента на 1:

```
(defun p1 (obj)
  (+ 1 obj))
```

; Остаток от деления на 2:

```
(defun m2 (obj)
  (mod obj 2))
```

Примеры вызовов:

(mapping p1 '(2 3 4)) дает в качестве результата '(3 4 5)

(mapping m2 '(2 3 4)) возвращает '(0 1 0)

Задача 1 : вариант для newLISP-tk.

; Описание функционала в newLISP-tk :

```
(define (new_mapping fun lst)
  (cond
    ((null? lst) '())
    (true (cons (apply fun (list (first lst)))
                (new_mapping fun (rest lst))))
  ))
```

; Функция увеличения элемента на 1 :

```
(define (p1 obj)
  (+ 1 obj)
)
```

; Остаток от деления на 2 :

```
(define (m2 obj)
  (mod obj 2))
```

Примеры использования применяющих функционалов.

Задача 2. Написать функционал, который проверяет выполнение некоторого условия (SYMBOLP/symbol?, INTEGERP/integer?, MINUSP, ZEROP/zero?) для каждого элемента списка.

; Описание функционала в muLISP :

```
(defun every (fun lst)
  ((null lst) t)
  ((funcall fun (car lst))(every fun (cdr lst)))
  nil)
```

Пример вызова :

```
(every integerp '(1 2))
```

возвращает T в качестве результата.

; Описание функционала в newLISP-tk :

```
(define (every fun lst)
  (cond
    ((null? lst) true)
    ((apply fun (list (first lst)))(every fun (rest lst)))
    (true nil)))
```

Пример вызова :

```
(every integer? '(1 2))
```

возвращает true в качестве результата.

Примеры использования применяющих функционалов.

Задача 3. Написать функционал, который возвращает Т, если найдется хотя бы один элемент списка, для которого предикативная функция fun дает Т.

; Описание функционала в muLISP :

```
(defun certain (fun lst)
  ((null lst) nil)
  ((funcall fun (car lst)) t)
  (certain fun (cdr lst)))
```

Пример вызова :

(certain integerp '(1 2 e r)) возвращает Т в качестве результата.

; Описание функционала в newLISP-tk :

```
(define (certain fun lst)
  (cond
    ((null? lst) nil)
    ((apply fun (list (first lst))) true)
    (true (certain fun (rest lst)))))
```

Пример вызова :

(certain integer? '(1 2 e r))
возвращает true в качестве результата.

Редукция как функция высшего порядка.

Редукция позволяет производить действия рекурсивно с элементами списка `lst` с условием окончания рекурсии `init`. `fun` – функция, которая вызывается для работы с элементами списка `lst`.

```
; Описание функционала в muLISP :  
(defun reduce (fun lst init)  
  ((null lst) init)  
  (funcall fun (car lst)(reduce fun (cdr lst) init)))
```

```
; Описание функционала в newLISP-tk  
(define (reduce fun lst init)  
  (cond  
    ((null? lst) init)  
    (true (apply fun (list (first lst)  
                          (reduce fun (rest lst) init)  
                          ))))
```

Примеры вызовов :

`(reduce '+ '(1 2 3 4) 0)` дает в качестве результата 10

`(reduce '* '(1 2 3 4) 1)` возвращает 24

Функция REDUCE встроена в **muLISP**.

Описание редукции с помощью локального определения.

muLISP :

```
(defun reduce1 (fun lst init)
  ((null lst) init)
  ((lambda (z)
    (funcall fun (car lst) z))
   (reduce1 fun (cdr lst) init))
)
```

newLISP-tk :

```
(define (reduce1 fun lst init)
  (cond
    ((null? lst) init)
    (true ((lambda (z)
             (apply fun (list (first lst) z)))
           (reduce1 fun (rest lst) init))))))
```

При этом результат рекурсивного вызова функции `reduce1` для хвоста списка `lst` становится фактическим параметром лямбда-вызова.

Применение редукции.

Задача (из предыдущей лекции). Есть список. Сформировать список, содержащий два элемента : сумма и произведение элементов списка.

; Решение (muLISP) :

```
(defun my_reduce (fun lst init)
  (cond
    ((null lst) init)
    (t (apply fun (list (car lst)(my_reduce fun (cdr lst) init))))
  ))
```

; Функция накопления результата

```
(defun acc (obj lst)
  (list (+ obj (car lst))
        (* obj (nth 1 lst))))
)
```

```
(my_reduce 'acc '(1 2 3 4) (list '0 '1))
```

Данный вызов дает в качестве результата список (10 24)

“Сумма-произведение”: реализация с применением функции редукции (продолжение).

; Решение (newLISP-tk) :

```
(define (reduce fun lst init)
  (cond
    ((null? lst) init)
    (true (apply fun (list (first lst)(reduce fun (rest lst) init))))
  ))
```

; Функция накопления результата

```
(define (acc obj lst)
  (list (+ obj (first lst))
        (* obj (nth 1 lst)))
)
```

```
(reduce 'acc '(1 2 3 4) (list '0 '1))
```

Данный вызов дает в качестве результата список (10 24)

Отображающие функционалы.

Определение 5. Отображающие функционалы Лиспа или MAP-функционалы есть функции, отображающие некоторым образом список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью.

Имена MAP - функций начинаются на MAP, их вызов имеет вид :

(MAPx fn I1 I2 ... IN). Здесь I1 ... IN - списки, а fn - функция от N аргументов.

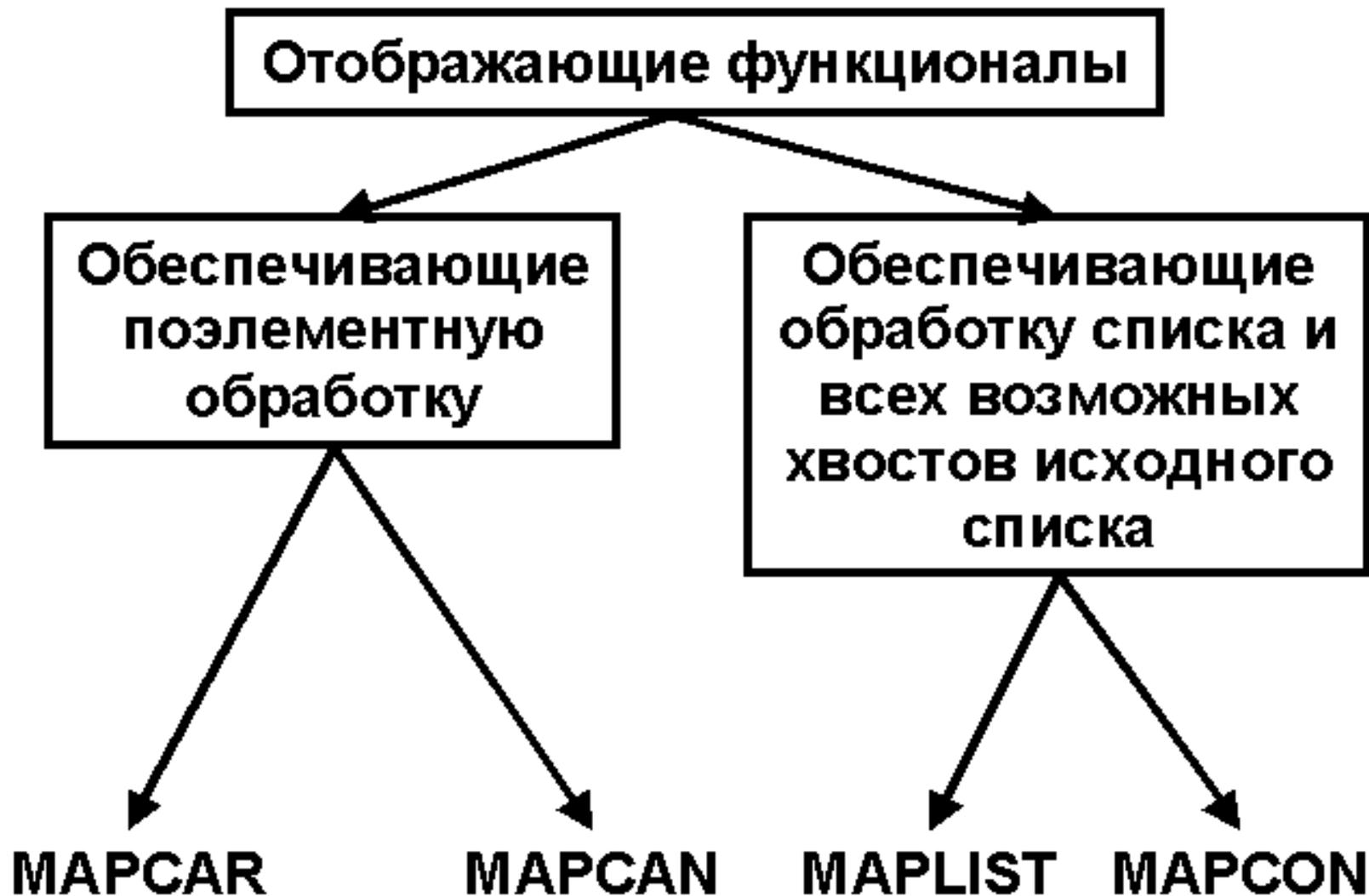
Как правило, MAP - функция применяется к одному аргументу-списку, то есть fn - функция от одного аргумента :

(MAPx fn список)

В newLISP-tk определен один отображающий функционал map. Он отображает аргументы-списки в новый список применением к одинаково расположенным элементам этих списков функции, представленной первым аргументом.

Пример : (map + '(1 2 3) '(50 60 70)) возвращает '(51 62 73).

Виды отображающих функционалов.



Функционалы поэлементной обработки.

1). Функционал MAPCAR обеспечивает реализацию функционального аргумента над всеми элементами списка и объединяет результаты в список.

Пример : (mapcar 'list '(a b c)) дает '((a)(b)(c)).

2). Функционал MAPCAN аналогичен MAPCAR, отличие состоит в объединении списков-результатов с использованием структуроразрушающей псевдофункции NCONC.

Пример : (mapcan 'list '(a b c)) дает '(a b c)

Функционалы списочной обработки. Псевдофункционалы.

1). Функционал **MAPLIST** обеспечивает реализацию функционального аргумента над списком и всеми его хвостовыми частями.

Пример : `(maplist 'list '(a b c))` дает `'(((a b c))((b c))((c)))`.

2). Функционал **MAPCON** аналогичен **MAPLIST**, отличие состоит в использовании структуроразрушающей псевдофункции **NCONC**.

Пример : `(mapcon 'list '(a b c))` дает `'((a b c)(b c)(c))`.

Псевдофункционалы **MAPC** и **MAPL** используются для получения побочного эффекта. Аналогичны по действию **MAPCAN** и **MAPCON** и отличаются тем, что не объединяют и не собирают результаты, а теряют их.

Пример.

`(mapc 'list '(a b c))`

`(mapl 'list '(a b c))`

В обоих случаях будет возвращен список `'(a b c)`

Использование отображающих функционалов.

Задача. Преобразовать локальное определение LET в локальное определение LAMBDA. Решение :

```
(setq letlist '((form1 fact1)(form2 fact2)(form3 fact3)))
```

Вызов (mapcar 'car letlist) дает список формальных параметров : (form1 form2 form3).

Вызов (mapcar 'cadr letlist) дает список фактических параметров : (fact1 fact2 fact3).

Результирующее локальное лямбда-определение получается следующим образом :

```
(list 'lambda (mapcar 'car letlist) 'body)
```

Как результат получаем : (lambda (form1 form2 form3) body)

Получение лямбда-вызова на основе описанного преобразования локального определения LET в локальное определение LAMBDA происходит следующим образом :

```
(cons (list 'lambda (mapcar 'car letlist) 'body) (mapcar 'cadr letlist))
```

В результате получаем :

```
((lambda (form1 form2 form3) body) fact1 fact2 fact3)
```

Вариант описания редукции на основе преобразования LET-LAMBDA

**; Запись локального определения LET с помощью
; локального определения LAMBDA**

```
(defun let (letlist body)
  (cons (list 'lambda (mapcar 'car letlist) body)
        (mapcar 'cadr letlist))
)
```

**; Запись reduce с применением
; локального определения LET.**

```
(defun reduce2 (fun lst init)
  ((null lst) init)
  (eval (let (setq z (reduce2 fun (cdr lst) init))
          (funcall fun (car lst) z))
  )
)
```

Автофункции.

Класс автофункций образуют функции, использующие или копирующие себя. Данный класс есть результат объединения использования рекурсии и функционалов.

Различают автоаппликативные (получающие сами себя в качестве аргументов) и авторепликативные (возвращающие сами себя) функции.

; muLISP-пример

; автоаппликативного варианта факториала

```
(defun fact (n)  
  (factorial 'factorial n))
```

```
(defun factorial (f n)  
  ((zerop n) 1)  
  (* n (funcall f f (- n 1))))
```

Автофункции (продолжение).

; Автоаппликативный вариант факториала
; в newLISP-tk

```
(define (fact_autofun n)  
  (factorial 'factorial n))
```

```
(define (factorial f n)  
  (cond  
    ((zero? n) 1)  
    (true (* n (apply f (list f (- n 1)))))))
```

Возможными применениями автоаппликативных функций могут быть задачи, сохраняющие неизменными определенные свойства : применимость, репродуцируемость, способность к самоизменениям – приспособляемости, согласованности и обучаемости. Основная проблема – учет инвариантных свойств вычислений.