ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

Лекция 3. Методы разработки функциональных программ. Функции высших порядков.

Основные и вспомогательные функции.

Использование вспомогательных функций и применение накапливающих параметров являются основными методами разработки функциональных программ. Понятия основной и вспомогательной функции являются относительными : одна и та же функция может использовать для вычисления своего значения другие функции из числа описанных в программе, но в то же время и ее могут вызывать как вспомогательную.

Различают нисходящее и восходящее проектирование функциональных программ. Основным отличием нисходящего проектирования является решение задачи с использованием разработанных ранее функций в роли вспомогательных.

Пример нисходящего проектирования для задачи преобразования списка в множество.

Основные отличия списков и

множеств. Список :

- **Упорядоченная последовательность : (1 2 3)≠(3 2 1);**
- **Один и тот же элемент может встречаться дважды.**

Множества :

- **Не упорядочены;**
- **Каждый элемент встречается ровно один раз.**

Аргумент : список.

Результат : множество.

Условие выхода из рекурсии : ((null lst) nil)

Вариант 1 : преобразование списка в множество.

Генерация результата :

Включить голову списка в множество, полученное из хвоста, из которого удалены все вхождения головы.

Предположим, что мы имеем функцию удаления всех вхождений объекта в список :

GCLisp :

```
(defun delete (obj list)
 ((null list) nil)
 ((equal obj (car list))(delete obj (cdr list)))
 (cons (car list)(delete obj (cdr list))))
```

newLISP-tk :

```
(define (delete_from_list obj lst)
 (cond
    ((null? lst) '())
    ((= obj (first lst)) (delete_from_list obj (rest lst) ) )
    (true (cons (first lst) (delete_from_list obj (rest lst))))))
```

Будем использовать вышеописанную функцию для удаления из исходного списка вхождений повторяющихся элементов :

GCLisp :

```
(defun list_set1 (list)
 ((null list) nil)
 (cons (car list)
       (list_set1 (delete (car list)
                          (cdr list)))))
```

newLISP-tk :

```
(define (list_set1 lst)
 (cond
    ((null? lst) '())
    (true (cons (first lst)
          (list_set1 (delete_from_list (first lst) (rest lst)))))))
```

Вариант 2 : преобразование списка в множество.

Генерация результата :

Если голова списка содержится в хвосте, то возвратить в качестве результата множество, полученное из хвоста списка. Иначе включить голову в множество, полученное из хвоста. Для определения принадлежности объекта списку будем использовать разработанную нами ранее функцию принадлежности элемента списку.

GCLisp :

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

newLISP-tk :

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

```
(defun list_set2 (list)
 ((null list) nil)
 ((member (car list)(cdr list))
          (list_set2 (cdr list)))
 (cons (car list)(list_set2 (cdr list))))
```

```
(define (list_set2 lst)
 (cond
  ((null? lst) '())
  ((my_member (first lst)(rest lst))(list_set2 (rest lst)))
  (true (cons (first lst)(list_set2 (rest lst))))))
```

Пример восходящего проектирования : объединение множеств.

Аргументы : 2 множества lst1 и lst2 в списочном представлении.

Результат : множество.

Генерация результата :

Включить голову списка lst1 в множество-результат объединения хвоста lst1 и lst2. Для этого необходимо построить функцию, включающую заданный объект в список, если он там отсутствует.

GCLisp :

```
(defun unit (lst1 lst2)
((null lst1) lst2)
(put (car lst1)(unit (cdr lst1) lst2)))
```

newLISP-tk :

```
(define (put obj lst)
 (cond
   ((my_member obj lst) lst)
   (true (cons obj lst))))

(define (unit lst1 lst2)
 (cond
   ((null? lst1) lst2)
   (true (put (first lst1)(unit (rest lst1) lst2)))))
```

Путем использования функции put получаем еще один вариант функции преобразования списка в множество.

GCLisp :

```
(defun list_set3 (list)
 ((null list) nil)
 (put (car list) (list_set3 (cdr list))))
```

newLISP-tk :

```
(define (list_set3 lst)
 (cond
  ((null? lst) '())
  (true (put (first lst) (list_set3 (rest lst))))))
```

Преобразование списка в множество при наличии элементов-списков (GCLisp).

```
; Функция сравнения множеств
(defun compare_sets (set1 set2)
 ((and (null set1)(null set2)) T)
 ((member (car set1) set2)
  (compare_sets (delete (car set1)(cdr set1))
                (delete (car set1) set2)))
 ((member_of_set (list_set4 (car set1)) (list_set4 set2))
  (compare_sets (list_set4 (del_set (car set1)(cdr set1)))
                (list_set4 (del_set (car set1) set2)))))
; Расширенная функция принадлежности
; множества другому множеству
(defun member_of_set (set1 set2)
 ((and (null set2)(not (null set1))) nil)
 ((and (not (atom (car set2)))
       (compare_sets set1 (car set2))) T)
 (member_of_set set1 (cdr set2)))
; Функция удаления элемента-множества
(defun del_set (set1 set2)
 ((null set2) nil)
 ((compare_sets set1 (car set2))
  (del_set set1 (cdr set2)))
 (cons (car set2)(del_set set1 (cdr set2))))

(defun list_set4 (list)
 ((null list) nil)
; Очередной элемент списка является атомом
 ((atom (car list))
  (cons (car list)(list_set4 (delete (car list)(cdr list)))))
; Очередной элемент списка является списком
 (cons (list_set4 (car list))
       (list_set4 (del_set (car list)(cdr list)))))
```

Преобразование списка в множество при наличии элементов-списков (newLISP-tk).

```
; Функция сравнения множеств
(define (compare_sets set1 set2)
 (cond
  ((and (null? set1)(null? set2)) true)
  ((my_member (first set1) set2)
          (compare_sets (delete_from_list (first set1)(rest set1))
                  (delete_from_list (first set1) set2)))
  ((member_of_set (list_set4 (first set1)) (list_set4 set2))
          (compare_sets (list_set4 (del_set (first set1)(rest set1)))
                  (list_set4 (del_set (first set1) set2))))))

; Расширенная функция принадлежности множества другому множеству
(define (member_of_set set1 set2)
 (cond
  ((and (null? set2)
      (not (null? set1))) nil)
  ((and (not (atom? (first set2)))
      (compare_sets set1 (first set2))) true)
  (true (member_of_set set1 (rest set2)))))

; Функция удаления элемента-множества
(define (del_set set1 set2)
 (cond
  ((null? set2) '())
  ((compare_sets set1 (first set2))
   (del_set set1 (rest set2)))
  (true (cons (first set2)(del_set set1 (rest set2))))))

; Головная функция
(define (list_set4 lst)
 (cond
  ((null? lst) '())
; Очередной элемент списка является атомом
  ((atom? (first lst))
    (cons (first lst)
       (list_set4 (delete_from_list (first lst)(rest lst)))))
; Очередной элемент списка является списком
  (true (cons (list_set4 (first lst))
          (list_set4 (del_set (first lst)(rest lst)))))))
```

Использование накапливающих параметров.

Задача : написать функцию реверсирования

списка. Аргумент : список lst.

Результат : тот же список, переписанный в обратном

порядке. Условие окончания рекурсии : пустой список.

Генерация результата (GCLisp) : (append (reverse (cdr lst))(cons (car lst) nil))

```
(defun reverse (lst)
 ((null lst) nil)
 (append (reverse (cdr lst))(cons (car lst) nil)))
```

Рассмотрим возможность снижения вычислительной сложности задачи путем уменьшения числа вызовов простейших базовых функций. Функция выполняется тем эффективнее, чем меньше число вызовов cons она предполагает. Исследуем зависимость числа N вызовов функции cons функцией reverse от числа n элементов реверсируемого списка.

Оценка вычислительной сложности для задачи реверсирования списка.

Пусть дан список ‘(1 2 3 4 5). Вызываем (reverse ‘(1 2 3 4 5))
Шаг 1. 1-й аргумент append : ‘(2 3 4 5). Результат вызова cons : ‘(1) – 2-й аргумент append.
Шаг 2. 1-й аргумент append : ‘(3 4 5). Результат вызова cons : ‘(2) – 2-й аргумент append.
Шаг 3. 1-й аргумент append : ‘(4 5). Результат вызова cons : ‘(3) – 2-й аргумент append.
Шаг 4. 1-й аргумент append : ‘(5). Результат вызова cons : ‘(4) – 2-й аргумент append.
Шаг 5. 1-й аргумент append : ‘(). Результат вызова cons : ‘(5) – 2-й аргумент append. Достигнуто условие окончания рекурсии (пустой список).
Итого на этапе развертывания рекурсии получилось n, то есть 5 вызовов cons. Рассмотрим теперь свертывание рекурсии.

Вычислительная сложность для реверсирования списка (продолжение).

Посредством функции append конструируем список-результат из извлекаемых из стека результатов рекурсивных вызовов :

(append nil ‘(5)) → ‘(5) - 0 вызовов cons.

(append '(5) ‘(4)) → ‘(5 4) - 1 вызов cons.

(append '(5 4) ‘(3)) → ‘(5 4 3) - 2 вызова cons.

(append '(5 4 3) ‘(2)) → ‘(5 4 3 2) - 3 вызова cons.

(append '(5 4 3 2) ‘(1)) → ‘(5 4 3 2 1) - 4 вызова cons.

Число вызовов cons на этапе свертывания рекурсии соответствует сумме n-1 первых членов арифметической прогрессии :

$$\frac{(1+(n-1))*(n-1)}{2} = \frac{n*(n-1)}{2}.$$

Общее число вызовов cons при работе функции reverse составляет $n + \frac{n*(n-1)}{2} = \frac{2*n+n^2-n}{2} = \frac{n^2+n}{2} = \frac{n*(n+1)}{2}$,

т.е. $N = \frac{n*(n+1)}{2}$.

Применение вспомогательной функции с накапливающим параметром для реверсирования списка.

Рассмотрим GCLisp-вариант функции реверсирования , который предполагает *N=n* вызовов cons. Данный вариант функции реверсирования использует вспомогательную функцию с накапливающим параметром, который передается в качестве результата в основную функцию по завершению рекурсии.

```
(defun reverse (lst)
 (rev lst nil))

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

Накапливающий параметр

Функция rev работает следующим образом :

lst1	lst2
(1 2 3 4)	()
(2 3 4)	(1)
(3 4)	(2 1)
(4)	(3 2 1)
()	(4 3 2 1)

Возвращается в качестве результата

Реализация функции реверсирования списка в newLISP-tk.

Без использования накапливающего параметра :

```
(define (my_reverse lst)
 (cond
   ((null? lst) '())
   (true (append (my_reverse (rest lst)) (cons (first lst) '()))))
 )
)
```

С использованием накапливающего параметра :

```
(define (my_reverse lst)
 (rev lst '())
)

(define (rev lst init)
 (cond
   ((null? lst) init)
   (true (rev (rest lst) (cons (first lst) init)))
 )
)
```

Другой пример (GCLisp) : нахождение суммы и произведения элементов списка.

Задача. Есть список. Сформировать список, содержащий два элемента : сумма и произведение элементов списка.

```
; Суммирование элементов списка
(defun sum (lst)
 ((null lst) 0)
 (+ (car lst)(sum (cdr lst))))
; Произведение элементов списка
(defun mult (lst)
 ((null lst) 1)
 (* (car lst)(mult (cdr lst))))
; Формирование списка из суммы и произведения
; элементов исходного списка
; Первый вариант решения – с применением sum и mult
; в качестве вспомогательных функций.
(defun s_m1 (lst)
 (list (sum lst)(mult lst)))
```

Сумма и произведение элементов списка (продолжение).

```
; Вариант функции формирования списка
; "сумма и произведение"
; с применением накапливающих параметров
(defun s_m2 (lst)
  (sm lst 0 1))
; Накопление результата
(defun sm (lst s p)
 ((null lst)(list s p))
 (sm (cdr lst)(+ (car lst) s)(* (car lst) p)))
; Еще один вариант решения задачи.
(defun s_m3 (lst)
 ((null lst)(list 0 1))
 (list (+ (car lst)(car (s_m3 (cdr lst))))
       (* (car lst)(cadr (s_m3 (cdr lst))))))
```

Сумма и произведение элементов списка (newLISP-tk).

```
; Вариант функции формирования списка
; "сумма и произведение"
; с применением накапливающих параметров
; Реализация в newLISP-tk
(define (s_m2 lst)
  (sm lst 0 1))
; Накопление результата
(define (sm lst s p)
 (cond
   ((null? lst)(list s p))
   (true (sm (rest lst)(+ (first lst) s)(* (first lst) p)))))
; newLISP-вариант реализации функции s_m3.
(define (s_m3 lst)
 (cond
 ((null? lst)(list 0 1))
 (true (list (+ (first lst)(first (s_m3 (rest lst))))
             (* (first lst)(nth 1 (s_m3 (rest lst))))))))
```

Локальные определения.

Локальные определения относятся к управляющим структурам Лиспа и обеспечивают :
1). Сокращение количества рекурсивных вызовов функций;
2). Делают программу более удобочитаемой.
Существует две конструкции локальных определений в Лиспе : LET и LAMBDA.
Функция LET создает локальную связь и является синтаксическим видоизменением LAMBDA-вызова, в котором формальные и фактические параметры помещены совместно в начале формы :

```
(let ((формальный параметр 1 фактический параметр 1)
        . . .
      (формальный параметр 1 фактический параметр 1))
     <тело функции> )
```

В muLISPe LET является библиотечной функцией, ее можно использовать, вызвав COMMON.LSP через RDS.

Описание “нахождения суммы и произведения” с применением LET и LAMBDA.

GCLisp :

```
(defun s_m4 (lst)
 ((null lst)(binomial 0 1))
  (let
      ((n (car lst))
       (z (s_m4 (cdr lst)))
       )
      (binomial (+ n (car z))(* n (cadr z)))
  )
)
(defun s_m5 (lst)
 ((null lst)(binomial 0 1))
 ((lambda (n z)
     (binomial (+ n (car z))(* n (cadr z))))
  (car lst)
  (s_m5 (cdr lst))
 )
)
```

newLISP-tk :

```
(define (s_m4 lst)
 (cond
  ((null? lst)(list 0 1))
  (true
   (let
      ((n (first lst))
       (z (s_m4 (rest lst)))
       )
      (list (+ n (first z))(* n (nth 1 z)))
   ))))

(define (s_m5 lst)
 (cond
  ((null? lst)(list 0 1))
  (true
    ((lambda (n z)
       (list (+ n (first z))(* n (nth 1 z))))
     (first lst)(s_m5 (rest lst))
    )
  )))
```

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 elal, 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 T, если найдется хотя бы один элемент списка, для которого предикативная функция fun дает T.

```
; 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 T 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 l1 l2 ... lN). Здесь l1 ... lN - списки, а fn - функция от N аргументов.

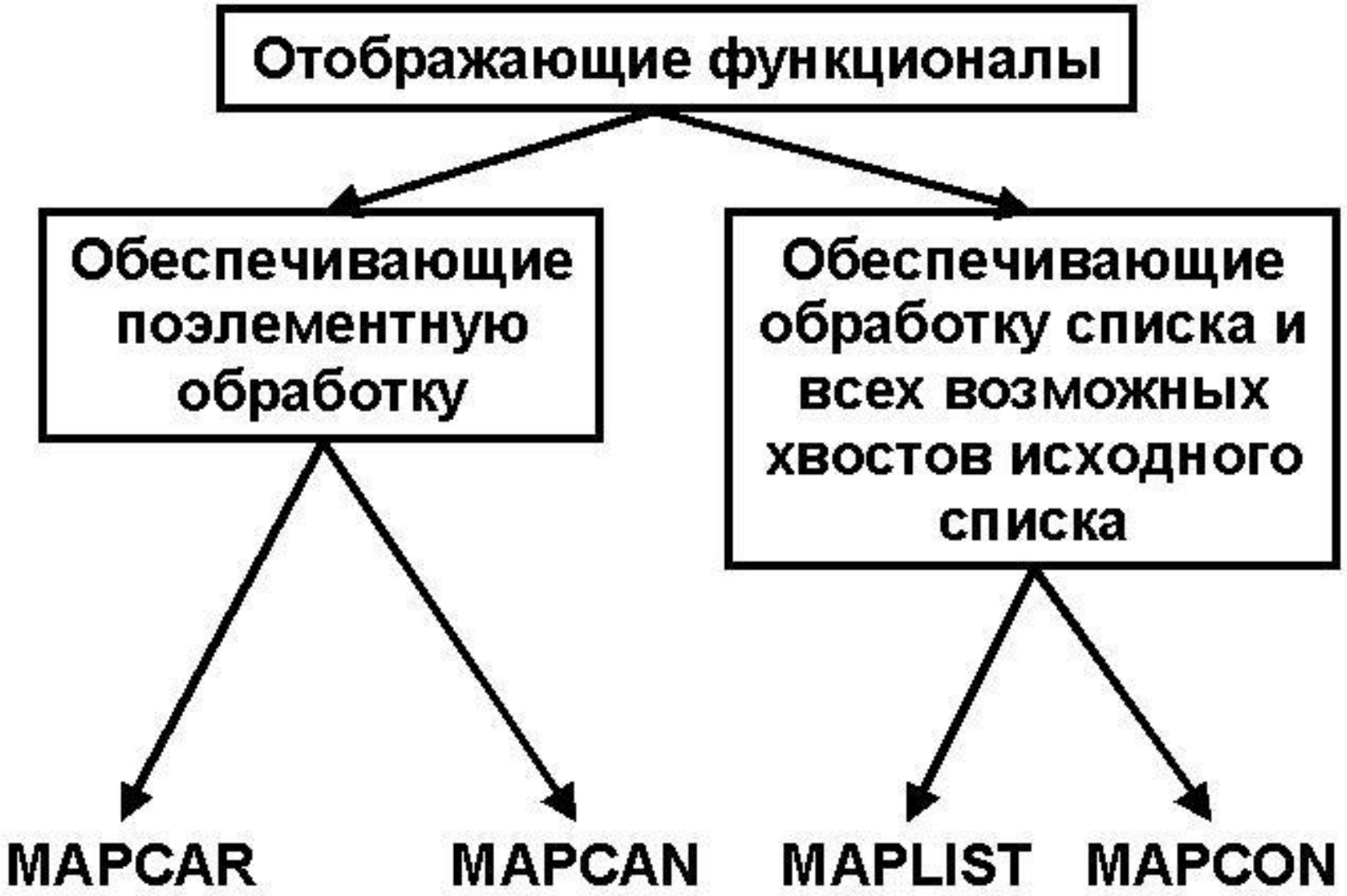
Как правило, MAP - функция применяется к одному аргументу-списку, то есть fn - функция от одного аргумента :

(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.