

Методы разработки функциональных программ.

Лекция 5.

Специальности : 230105, 010501

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, но только тогда, когда при декомпозиции исходной задачи получающиеся подзадачи решаются с помощью этих функций.

Пример нисходящего проектирования для задачи преобразования списка в множество.

Основные отличия списков и множеств.

Список :

- Упорядоченная последовательность : $(1\ 2\ 3) \neq (3\ 2\ 1)$;
- 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 : $((\text{null lst}) \text{nil})$

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 :

muLISP :

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

newLISP (реализовано в 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 :

muLISP :

```
(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 вернуть 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.

muLISP :

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

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

newLISP-tk :

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

```
(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.

muLISP :

```
(defun put (obj list)
  ((member obj list) list)
  (cons obj list))

(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.

muLISP :

```
(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 (muLISP).

```
; 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))))))
```

; 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))))))
```

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

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 (muLISP) : `(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 muLISP-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))
```

Накапливающий
параметр

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

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-тк.

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 (muLISP) : 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 muLISPе LET является библиотечной функцией, ее можно использовать, вызвав COMMON.LSP через RDS.

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

muLISP :

```
(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)))
     )))
)
```