Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 10

По дисциплине «Функциональное и логическое программирование»

Студент: Тимонин А. С.

Группа ИУ7-62б

Преподаватель Толпинская Н. Б.

Москва.
2020 г.

Практическая часть

Задание 7.

Пусть list-of-list список, состоящий из списков. Написать функцию, которая вычисляет сумму длин всех элементов list-of-list, т.е. например для аргумента ((1 2) (3 4)) -> 4.

```
(defun getLengthHelper (list-of-list sum) ;перестает
считать оставшиеся элементы после входа в список в главном
списке
    (cond
        ((null (car list-of-list)) sum)

        ((atom (car list-of-list)) (getLengthHelper (cdr
list-of-list) (+ 1 sum)))
        ( (listp (car list-of-list)) (getLengthHelper (car
list-of-list) sum) )
    )
)

; list-of-list - список, sum - результирующая сумма
; (getLengthHelper (list 1 2 (list 2 3)) 0) -> 4

(defun getLength (list-of-list)
    (getLengthHelper list-of-list 0)
)

;list-of-list - список
; (getLength (list 1 2 (list 2 3))) -> 4



(defun getLength1 (list-of-lists)
    (reduce #'+
        (mapcar (lambda (lst)
                    (if (listp lst)
                        (getLength1 lst)
                        1
                    )
                ) list-of-lists
        )
    )
)
;list-of-list - список
```

```
; (getLength (list (list 1 2) (list 2 4 7))) -> 5
```

Задание 8.

Написать рекурсивную версию (с именем reg-add) вычисления суммы чисел заданного списка. Например: (reg-add (2 4 6)) -> 12

```
(defun getSumHelper (lst sum)
    (cond
        ( (null (car lst)) sum )
        ( t (getSumHelper (cdr lst) (+ (car lst) sum)) )
    )
)
; lst – список
; (getSumHelper (list 2 4 6) 0) -> 12

(defun reg-add (lst)
    (getSumHelper lst 0)
)
; lst – список с числами
; (red-add (list 2 4 6)) -> 12
```

Задание 9.

Написать рекурсивную версию с именем recnth функции nth.

```
(defun recnth(n lst)
    (cond
        ( (or
            (< n 0)
            (> n (length lst))
            (null (car lst))
          )  Nil )
        ( (= n 0) (car lst) )
        ( t (recnth (- n 1) (cdr lst)) )
    )
)
; n – порядок искомого элемента, lst – список
; (myNth 3 (list 1 2 3 6 4 1)) -> 6
; (myNth 8 (list 1 2 3 6 4 1)) -> NIL
```

Задание 10.

Написать рекурсивную функцию alloddr, которая возвращает t когда все элементы списка нечетные.

```
(defun allOddp (lst)
   (cond
        ((null (car lst)) t )
        ( (not (oddp (car lst))) Nil )
        ( t  (allOddp (cdr lst)) )
    )
)
; lst – список
; (allOddp (list 1 2 3 4)) -> Nil
; (allOddp (list 1 3)) -> T
; (allOddp ()) -> T
```

Задание 11.

Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка - аргументы.

```
(defun lastEl (lst)
    (if (null (cdr lst))
        (car lst)
        (lastEl (cdr lst))
    )
)
; lst – список
; (lastEl (list 1 2 3 4 5)) -> 5
; (lastEl ()) -> NIL
```

Задание 12.

Написать рекурсивную функцию, относящуюся к дополняемой рекурсии с одним тестом завершения, которая вычисляет сумму всех чисел от 0 до n-ого аргумента функции.

Вариант:

1) от n-аргумента функции до последнего >= 0,

2) от n-аргумента функции до m-аргумента с шагом d.

Вариант 1.

```
(defun goToN(lst n)
    (cond
        ( (or (< n 0) (> n (length lst))
                (null (car lst))) (list 0) )
        ( (= n 0) lst )
        ( t (goToN (cdr lst) (- n 1)))
    )
)
; lst - список
; n - нужный порядковый номер в списке
; goToN - проходит список до N элемента и возвращает все
оставшиеся элементы списка
; (goToN (list 1 2 3 4 5) 2) -> (3 4 5)



(defun getNsum1Helper (lst)
    (if (null (car lst))
        0
        (+ (car lst) (getNsum1Helper (cdr lst)))
    )
)
; lst - список
; getNsum1Helper - вычисляет сумму всех элементов списка
lst
; (getNsum1Helper (list 1 2 3)) -> 6



(defun getNsum1 (lst n)
    (getNsum1Helper (goToN lst n))
)
; lst - список, n - нужный порядковый номер в списке
; getNsum1 - функция объединяющая предыдущие 2 функции,
чтобы найти сумму элементов от Nого до последнего
```

Вариант 2.

```
(defun goToN(lst n)
    (cond
        ( (or (< n 0) (> n (length lst))
                (null (car lst))) (list 0) )
        ( (= n 0) lst )
        ( t (goToN (cdr lst) (- n 1)))
    )
)
; lst - список
; n - нужный порядковый номер в списке
```

```
; goToN - проходит список до N элемента и возвращает все
оставшиеся элементы списка
; (goToN (list 1 2 3 4 5) 2) -> (3 4 5)



(defun getNsum2Helper (lst n tt d result)
    (cond
        ( (or (null (car lst)) (>= n tt)) result)
        (t (getNsum2Helper (goToN lst d) (+ n d) tt d (+
result (car lst))))
    )
)
; lst - список
; n, tt - начальная и конечная границы
; d - шаг
; result - результирующая переменная
;(getNsum2Helper (list 1 2 3 4 5) 0 2 1 0) -> 3
; getNsum2Helper - подсчитывает сумму элементов списка с
0ой позиции по tt с шагом d и записывает результат в result



(defun getNsum2 (lst n tt d)
    (getNsum2Helper (goToN lst n) n tt d 0)
)
; lst - список
; n, tt - начальная и конечная границы
; d - шаг
; (getNsum2 (list 1 2 3 4 5 6 7 8 9 10) 0 6 2) -> 9
; getNsum2 - функция-оболочка
```

Задание 13.

Написать рекурсивную функцию, которая возвращает последнее нечетное число из числового списка, возможно создавая некоторые вспомогательные функции.

```
(defun lastOddpNumHelper (lst val)
    (cond
        ( (null (car lst)) val)
        ( (oddp (car lst))
                (lastOddpNumHelper (cdr lst) (car lst)) )
        ( t (lastOddpNumHelper (cdr lst) val) )
    )
```

```
)
; lst – список, val – результирующее значение
; lastOddpNumHelper – вычисляет последний нечетный элемент
; (lastOddpNumHelper (list 1 2 3 4 5 6 7 8) Nil) -> 7


(defun lastOddpNum (lst)
    (lastOddpNumHelper lst Nil)
)
; lst – список
; (lastOddpNum (list 1 2 3 4 5 6 7 8)) -> 7
```

Задание 14.

Используя cons-дополняемую рекурсию с одним тестом завершения, написать функцию которая получает как аргумент список чисел, а возвращает список квадратов этих чисел в том же порядке.

```
(defun getPowerNumsHelper (lst)
    (
        cons (* (car lst) (car lst))
            (if (> (length (cdr lst)) 0)
                (getPowerNums (cdr lst))
                nil
            )
    )
)

; lst – список
; (getPowerNumsHelper (list 1 2 3 4)) -> (1 4 9 16)
; getPowerNumsHelper – функция возведения в квадрат списка,
без проверки на пустоту списка


(defun getPowerNums (lst)
    (if (null lst)
        Nil
        (getPowerNumsHelper lst)
    )
)
; lst – список
; (getPowerNums (list 1 2 3 4)) -> (1 4 9 16)
```

```
; getPowerNums – проверка на пустоту списка, в случае
отрицания запускается функция getPowerNumsHelper
```

Задание 15.

Написать функцию с именем select-odd, которая из заданного списка выбирает все нечетные числа.

(Вариант 1: select-even,

Вариант 2: вычисляет сумму всех нечетных чисел(sum-all-odd) или сумму всех четных чисел (sum-all-even) из заданного списка.)

Вариант 1.

```
(defun getNextOddp (lst)
    (cond
        ( (null (car lst)) Nil )
        ( (oddp (car lst)) (getListOdd lst) )
        ( t (getNextOddp (cdr lst)) )
    )
)
; lst – список
; getNextOddp – доходит в списке до первого нечетного
элемента
; (getNextOddp (list 2 3 4)) -> (3)

(defun getListOdd (lst)
    (
        cons (car lst)
            (if (> (length (cdr lst)) 0)
                (getNextOddp (cdr lst))
                nil
            )
    )
)
; lst – список
; (getListOdd (list 2 3 4)) -> (2 3)
; getListOdd – добавляет в список все нечетные, не проверяя
первого полученного на вход

(defun select-odd (lst)
    (getListOdd (getNextOddp lst))
)
; select-odd – функция-обертка, которая запускает
getListOdd, getNextOddp
```

```
; (select-odd (list 1 2 3 4 5 6)) -> (1 3 5)
```

Вариант 2.

```
(defun getNextOddp (lst sum)
    (cond
        ( (or (null lst) (null (car lst))) nil )
        ( (oddp (car lst)) lst)
        ( t (getNextOddp (cdr lst) sum) )
    )
)
; lst - список, sum - результирующая сумма
; getNextOddp - доходит до первого нечетного элемента в
списке
; (getNextOddp (list 2 3 4) 0) -> (3 4)

(defun getListOdd (lst sum)
    (cond
        ( (or (null lst) (null (car lst))) sum )
        ( (oddp (car lst))
                (getListOdd (cdr lst) (+ sum (car lst))) )
        ( t (getListOdd (getNextOddp (cdr lst) sum) sum) )
    )
)
; lst - список, sum - результирующая сумма
; getListOdd - вычисляет сумму нечетных элементов
; (getListOdd (list 1 2 3 4 5) 0) -> 9

(defun sum-all-even (lst)
    (getListOdd lst 0)
)
; lst - список
; (sum-all-even (list 1 2 3 4 5)) -> 9
; sum-all-even - функция-оболочка, вычисляющая сумму всех
нечетных элементов в списке
```

Теоретическая часть

Способы организации повторных вычислений в Lisp.

1. Использование функционалов;
2. Использование рекурсии.

Что такое рекурсия?

Рекурсия – это ссылка на определяемый объект во время его определения.

Рекурсия в Lisp - естественный принцип обработки списков.

Классификация рекурсивных функций в Lisp.

1. Простая функция - один рекурсивный вызов в теле;
2. Рекурсия первого порядка - рекурсивный вызов встречается несколько раз;
3. Взаимная рекурсия - используется несколько функций, рекурсивно вызывающих друг друга.

В силу возможной сложности и разнообразия постановок задач, возможны комбинации и усложнения приведенных групп функций.

Функция называется косвенно рекурсивной в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов определяемой (первой) функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна.

Если в теле функции явно используется вызов этой же функции, то имеет место прямая рекурсия.

Рекурсия называется однократной, если функция вызывает саму себя один раз. Если функция вызывает саму себя два раза, то рекурсия называется двукратной и т.д.

Различные способы организации рекурсивных функций и порядок их реализации.

Существуют типы рекурсивных функций:

1. хвостовая;
2. дополняемая;
3. множественная; взаимная рекурсия;
4. рекурсия более высокого порядка.

Способы повышения эффективности реализации рекурсии.

В целях повышения эффективности рекурсивных функций рекомендуется
формировать результат не на выходе из рекурсии, а на входе в рекурсию, все действия выполняя до ухода на следующий шаг рекурсии. Это и есть хвостовая рекурсия.

Для превращения не хвостовой рекурсии в хвостовую и в целях формирования результата (результирующего списка) на входе в рекурсию рекомендуется использовать дополнительные (рабочие) параметры. При этом становится необходимым создать функцию-оболочку (как в задании 4) для реализации очевидного обращения к функции.