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»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9

По курсу: "Функциональное и Логическое программирование"

Тема Использование функционалов и рекурсии.

Группа ИУ7-63Б

Студент Сукочева А.

Преподаватель Толпинская Н.Б.

Преподаватель Строганов Ю. В.

Москва — 2021 г.

Практическая часть

Задание 1. Написать функцию, которая выбирает из заданного списка только те числа, которые расположены между двумя заданными границами.

```
(defun check-border (x a b)
    (and (>= x a) (<= x b)) )

(defun select-between (lst a b)
    (cond ((null lst) ())
    ((symbolp (car lst)) (cons (car lst) (select-between (cdr lst) a b)))
    ((listp (car lst)) (cons (select-between (car lst) a b) (select-between
       (cdr lst) a b)))
    ((check-border (car lst) a b) (cons (car lst) (select-between (cdr lst) a
       b)))
    (T (select-between (cdr lst) a b))) )
```

Пример использования:

```
(select-between '(1 2 (a b 3 4) T c 4 6 11 5) 2 7) ;; (2 (A B 3 4) T C 4 6 5)
```

Задание 2. Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов. (Напомним, что $A \times B$ — это множество всевозможных пар $(a\ b)$, где $a \in A$, $b \in B$)

```
(defun decart-elem (lst elem)
        (cond ((null lst) ())
        (T (cons (list elem (car lst)) (decart-elem (cdr lst) elem)))) )
```

Пример использования:

```
(decart-elem '(a b c) 'd) ;; => ((D A) (D B) (D C))
```

```
(defun decart (lst1 lst2)
        (cond ((null lst1) nil)
        (T (append (decart-elem lst2 (car lst1)) (decart (cdr lst1) lst2)))) )
```

Пример использования:

```
;; ((A D) (A E) (A F) (B D) (B E) (B F) (C D) (C E) (C F))
(decart '(a b c) '(d e f))
```

Задание 3. Почему так реализовано `reduce`, в чем причина?

```
        (reduce #'+ ()) ; 0
        (reduce #'* ()) ; 1
```

$L = (e_1 e_2 ... e_n)$
(reduce F L) $\equiv$ (F (... (F (F e_1 initial-value) e_2)) ... e_n)

```
(reduce '+ '(1 2 3 4)) ;; 10
(reduce '* '(1 2 3 4)) ;; 24
```

Причина состоит в том, что если начальное значение при умножении будет равно нулю, то и результат будет равен нулю

```
(reduce '* '(1 2 3 4) :initial-value 0) ;; 0
```

В случае сложения результат будет на единицу больше, что является некорректным результатом.

```
(reduce '+ '(1 2 3 4) :initial-value 1) ;; 11
```

Задание 4. 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 list-of-list-rec (lst len)
        (cond ((null lst) len)
        (T (list-of-list-rec (cdr lst) (+ len (length (car lst)) )))))

(defun list-of-list (lst)
        (list-of-list-rec lst 0))
```

Пример использования:

```
(list-of-list '((1 2 3) (4 5))) ;; => 5
```

Вариант для структурированного списка:

Примечание: +1 потому что нужно ещё учесть сам список (который дальше будет просматриваться).

```
(defun list-of-list-rec (lst len)
    (cond ((null lst) len)
        ((listp (car lst))
        (+ 1 (list-of-list-rec (car lst) 0) (list-of-list-rec (cdr lst) len)))
        (T (list-of-list-rec (cdr lst) (+ 1 len))) ))

(defun list-of-list (lst)
    (- (list-of-list-rec lst 0) (length lst)))
```

Пример использования:

```
(list-of-list '((1 2) (3 4) (5 6 (7 8) 9))) ;; => 10
```

Задание 5. Написать функцию, возвращающую квадрат смешанного стуктурированного списка. В результирующем списке только числа.

```
(defun square-lst (lst)
        (cond ((null lst) Nil)
        ((symbolp (car lst)) (square-lst (cdr lst)))
        ((listp (car lst)) (append (square-lst (car lst)) (square-lst (cdr
           lst))))
        (T (cons (* (car lst) (car lst)) (square-lst (cdr lst)))) ))
```

Пример использования:

```
(square-lst '((1 2 a) 'b 3 T 4))     ;; => (1 4 9 16)
(square-lst '(((((1 2 3)))))         ;; => (1 4 9)
```

Теоретическая часть

Классификация рекурсивных функций

Рекурсия – ссылка на описываемый объект во время его описания.

1. Простая - рекурсивный вызов встречается один раз.
2. Второго порядка - несколько рекурсивных вызовов.
3. Взаимная - несколько функций, которые могут друг друга взаимно вызывать.
4. Хвостовая рекурсия - результат строится на входе в рекурсию. На выходе ничего считать не нужно.

   Для преобразования не хвостовой рекурсии в хвостовую можно использовать дополнительные параметры, в которые при каждом вызове рекурсии будет формироваться и записываться результат. В этом случае необходимо использовать функцию-оболочку для запуска рекурсивной функции с начальными значениями дополнительных параметров.
5. Дополняемая рекурсия - использует дополнительную функцию. (результат рекурсии используется, как аргумент некоторой другой функции, которая называется дополняемой функцией)