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

к лабораторной работе №10

По курсу: «Функциональное и логическое программирование»

Тема: «Вложенные рекурсия и функционалы».

Студент: Якуба Д.В.
Группа: ИУ7-63Б
Преподаватели: Толпинская Н. Б.,
Строганов Ю. В.

Москва, 2021 г.

Практическая часть

Задание 1. Написать рекурсивную версию (с именем rec-add) вычисления суммы чисел заданного списка. Например: (rec-add (2 4 6)) -> 12.

Решение:

```
; без работы со структурированными смешанными списками
(defun rec-add (lst)
    (cond
        ((null lst) 0)
        (t (+ (car lst) (rec-add (cdr lst))))))

; с обработкой смешных структурированных списков
(defun rec-add (lst)
    (cond
        ((null lst) 0)
        ((symbolp (car lst)) (rec-add (cdr lst)))
        ((listp (car lst)) (+ (rec-add (car lst)) (rec-add (cdr lst))))
        ((numberp (car lst)) (+ (car lst) (rec-add (cdr lst))))))
```

```
(rec-add '(1 2 3)) -> 6;
(rec-add '(1 2 (7 8) 3)) -> 21;
(rec-add '(1 2 (7 oh 8) i 3 ((4 just wanna ((sleep)))))) -> 25;
```

Задание 2. Написать рекурсивную версию с именем rec-nth функции nth.

Решение:

```
(defun rec-nth (index lst)
    (cond
        ((or (null lst) (< index 0)) nil)
        ((zerop index) (car lst))
        (t (rec-nth (- index 1) (cdr lst)))))
```

```
(rec-nth 3 '(0 1 2 3)) -> 3;
(rec-nth 3 '(0 1 2 (3 3))) -> (3 3);
(rec-nth 3 '((3 3))) -> nil;
(rec-nth -3 '((3 3))) -> nil;
```

Задание 3. Написать рекурсивную функцию alloddr, которая возвращает t, когда все элементы списка нечетные.

Решение:

```
; описанные функции возвращают T при факте передачи в них пустых списков.
; в случае, если потребуется, чтобы при первичной передаче пустого списка
; возвращался nil - достаточно добавить обёрточную функцию с первым условием co
nd

; без работы с структурированными смешанными списками
(defun allodr (lst)
    (cond
```

```
        ((null lst) t)
        (t (and (oddp (car lst)) (allodr-inner (cdr lst))))))

; для работы с структурированными смешанными списками
(defun allodr (lst)
    (cond
        ((null lst) t)
        ((listp (car lst)) (allodr (car lst)))
        ((symbolp (car lst)) (allodr (cdr lst)))
        (t (and (oddp (car lst)) (allodr (cdr lst))))))
```

```
(allodr '(1 2 (3 4 (5 6)))) -> NIL;
(allodr '(1 1 (3 3 (5 5)))) -> T;
(allodr '(1 1 (3 (7) ((((((9)))))) 3 (5 5)))) -> T;
```

Задание 4. Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка-аргумента.

Решение:

```
(defun my-last (lst)
    (cond
        ((null (cdr lst)) lst)
        (t (my-last (cdr lst)))))
```

```
(my-last '(1 2 3 4)) -> (4);
(my-last '(1 2 3 (2 3))) -> ((2 3));
(my-last '(1 2 3 ())) -> (NIL);
```

Задание 5. 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.

Решение:

```
; от 0 до n-го
(defun sum-to-n (lst to-n)
    (cond
        ((or (null lst) (< to-n 0)) 0)
        (t (+ (car lst) (sum-to-n (cdr lst) (- to-n 1))))))

; от n-го до первого < 0
(defun sum-to-lz (lst)
    (cond
        ((or (null lst) (< (car lst) 0)) 0)
        (t (+ (car lst) (sum-to-lz (cdr lst))))))
```

```
(defun sum-from-n-to-lz (lst from-n)
    (cond
        ((null lst) 0)
        ((> from-n 0) (sum-from-n-to-lz (cdr lst) (- from-n 1)))
        (t (sum-to-lz lst))))

; от n-го до m с шагом h
(defun make-step (lst step-h)
    (cond
        ((> step-h 0) (make-step (cdr lst) (- step-h 1)))
        (t lst)))

(defun sum-to-m (lst to step-h)
    (cond
        ((or (null lst) (< to 0)) 0)
        (t (+ (car lst) (sum-to-m (make-step lst step-h) (- to step-h) step-
h)))))

(defun go-to-n (lst from to step-h)
    (cond
        ((null lst) 0)
        ((> from 0) (go-to-n (cdr lst) (- from 1) to step-h))
        (t (sum-to-m lst to step-h))))

(defun sum-from-to-step (lst from-n to-m step-h)
    (cond
        ((or (null lst) (= step-h 0)) 0)
        ((< from-n to-m) (go-to-n lst from-n (- to-m from-n) (abs step-h)))
        (t (go-to-n lst to-m (- from-n to-m) (abs step-h)))))
```

Задание 6. Написать рекурсивную функцию, которая возвращает последнее нечетное число из числового списка, возможно создавая некоторые вспомогательные функции.

Решение:

```
(defun ret-last-odd-inner (lst num)
    (cond
        ((null lst) num)
        (t (ret-last-odd-inner (cdr lst) (cond
                                            ((oddp (car lst)) (car lst))
                                            (t num))))))

(defun ret-last-odd (lst)
    (ret-last-odd-inner lst nil))
```

```
(ret-last-odd '(1 2 3 4 5)) -> 5;
(ret-last-odd '(1 2 3 4 2)) -> 3;
(ret-last-odd '(2 2 2 4 2)) -> nil;
```

Задание 7. 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.

Решение:

```
(defun get-sqr-list (lst)
    (cond
        ((null lst) nil)
        (t (cons (* (car lst) (car lst)) (get-sqr-list (cdr lst))))))
```

```
(get-sqr-list '(1 2 3 4)) -> (1 4 9 16);
(get-sqr-list '(1 -2 -3 4)) -> (1 4 9 16);
(get-sqr-list '(8 3 7 2)) -> (64 9 49 4);
```

Задание 8. 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.)

```
(defun select-odd (lst)
    (cond
        ((null lst) nil)
        ((oddp (car lst)) (cons (car lst) (select-odd (cdr lst))))
        (t (select-odd (cdr lst)))))

(defun sum-all-odd (lst)
    (cond
        ((null lst) 0)
        ((evenp (car lst)) (sum-all-odd (cdr lst)))
        (t (+ (car lst) (sum-all-odd (cdr lst))))))

(defun select-even (lst)
    (cond
        ((null lst) nil)
        ((evenp (car lst)) (cons (car lst) (select-even (cdr lst))))
        (t (select-even (cdr lst)))))

(defun sum-all-even (lst)
    (cond
        ((null lst) 0)
        ((oddp (car lst)) (sum-all-even (cdr lst)))
        (t (+ (car lst) (sum-all-even (cdr lst))))))
```

```
(select-even '(1 2 3 4 5 6)) -> (2 4 6);
(select-even '(-3 -2 -1 0 1 2 3 4 5 6)) -> (-2 0 2 4 6);
(select-odd '(1 2 3 4 5 6)) -> (1 3 5);
(select-odd '(-3 -2 -1 0 1 2 3 4 5 6)) -> (-3 -1 1 3 5);
(sum-all-odd '(1 2 3 4 5 6)) -> 9;
(sum-all-odd '(-3 -2 -1 0 1 2 3 4 5 6)) -> 5;
(sum-all-even '(1 2 3 4 5 6)) -> 12;
```

```
(sum-all-even '(-3 -2 -1 0 1 2 3 4 5 6)) -> 10;
```

Задание 9. Создать и обработать смешанный структурированный список с информацией: ФИО, зарплата, возраст, категория (квалификация). Изменить зарплату, в зависимости от заданного условия, и подсчитать суммарную зарплату. Использовать композиции функций.

Решение:

```
'(
    ((fio . Jasurbek_Mihailov) (salary . 3000) (age . 30) (pos . Manager))
    ((fio . Torwalds_Linus) (salary . 3) (age . 25) (pos . Director))
    ((fio . Karl_Marx) (salary . 300) (age . 55) (pos . Manager))
    ((fio . Dmitry_Yakuba) (salary . 500) (age . 20) (pos . Suicider)))

(defun get-val-by-key (key row)
    (cond
        ((null row) nil)
        ((equal (caar row) key) (cdar row))
        (t (get-val-by-key key (cdr row)))))

(defun by-key-no-cdr (key row)
    (cond
        ((null row) nil)
        ((equal (caar row) key) (car row))
        (t (by-key-no-cdr key (cdr row)))))

(defun change-salary-with-cond (table col-name predicate salaryFun)
    (cond
        ((null table) 'done)
        ((funcall predicate (get-val-by-key col-name (car table)))
            (and
                (rplacd (by-key-no-
cdr 'salary (car table)) (funcall salaryFun (get-val-by-
key 'salary (car table))))
                (change-salary-with-cond (cdr table) col-
name predicate salaryFun)))
        (t (change-salary-with-cond (cdr table) col-
name predicate salaryFun))))

(change-salary-with-cond table 'pos #'(lambda (got-pos) (equal 'Manager got-
pos)) #'(lambda (sal) (setf sal (/ sal 2))))

(defun get-sum-salary (table)
    (cond
        ((null table) 0)
        (t (+ (get-val-by-key 'salary (car table)) (get-sum-
salary (cdr table))))))
```

```
(change-salary-with-cond table 'pos #'(lambda (got-pos) (equal 'Manager got-
pos)) #'(lambda (sal) (setf sal (/ sal 2)))) -> Урезание зарплаты в два раза для
JASURBEK_MIGAILOV и KARL_MARX.
```

Теоретическая часть

1. Классификация рекурсивных функций.

1) Простая рекурсия. Вызов является единственным.

2) Рекурсия второго порядка. Присутствует несколько рекурсивных вызовов.

3) Взаимная рекурсия. Несколько рекурсивных функций, которые могут друг друга вызывать.

4) Хвостовая рекурсия. При очередном рекурсивном вызове функции все действия до входа выполнены, а при выходе ничего более делать не потребуется.

5) Дополняемая рекурсия. Используется для обработки car и cdr указателей. Результат рекурсии используется в качестве аргумента другой функции:

```
(defun func(x)
      (cond (end_test end-value)
            (t (add_function add_value (func changed_x))))
```

Частные случаи: cons-дополняемая рекурсия, дополняемая функция встречается после прерывания рекурсии.