



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 №6  
по дисциплине «Функциональное и логическое  
программирование»

Тема Использование функционалов

Студент Зайцева А. А.

Группа ИУ7-62Б

Оценка (баллы) \_\_\_\_\_

Преподаватели Толпинская Н.Б., Строганов Ю. В.

Москва — 2022 г.

# Практические задания

Используя функционалы:

1. Напишите функцию, которая уменьшает на 10 все числа из списка-аргумента этой функции.

1. Если все элементы списка – числа.

```
1 ; используя функционал mapcar
2 (defun all_minus_10 (lst)
3   (mapcar #'(lambda (x) (- x 10)) lst)
4 )
5
6 ; используя функционал mapcan
7 (defun all_minus_10 (lst)
8   (mapcan #'(lambda (x) (cons (- x 10) Nil)) lst)
9 )
10
11 (all_minus_10 '(0 10 -10 5.5 2/3)) => (-10 0 -20 -4.5 -28/3)
12 (all_minus_10 Nil) => NIL
```

2. Элементы списка – любые объекты.

а) Работа только по верхнему уровню.

```
1 ; используя функционал mapcar.
2 (defun all_minus_10 (lst)
3   (mapcar
4     #'(lambda (x) (cond ((numberp x) (- x 10)) (x)))
5     lst
6   )
7 )
8
9 ; рекурсивно
10 ; а) с помощью рекурсии, собирающей результат на выходе
11 (defun all_minus_10 (lst)
12   (cond
13     ((null lst) Nil)
14     ((numberp (car lst)) (cons (- (car lst) 10) (all_minus_10 (cdr lst))))
15     (t (cons (car lst) (all_minus_10 (cdr lst)))))
16 )
17 )
18
19
20
```

```

21 ; б) через "хвостовую рекурсию"
22 (defun all_minus_10_inner (lst result)
23   (cond
24     ((null lst) (reverse result))
25     ((numberp (car lst)) (all_minus_10_inner (cdr lst) (cons (- (car
26       lst) 10) result)))
27     (t (all_minus_10_inner (cdr lst) (cons (car lst) result))))
28   )
29 )
30 (defun all_minus_10 (lst)
31   (all_minus_10_inner lst ()))
32 )
33
34
35 (all_minus_10 '(0 a "abc" (1 k) 2/3)) => (-10 A "abc" (1 K) -28/3)

```

б) Работа по всем уровням структурированного списка.

```

1 ; вспомогательная функция, так как тарсар работает только со списками
2 ; (не с точечными парами).
3 (defun all_minus_10_cons (cns)
4   (cond
5     ((and (numberp (car cns)) (numberp (cdr cns))) (cons (- (car cns)
6       10) (- (cdr cns) 10)))
7     ((numberp (car cns)) (cons (- (car cns) 10) (cdr cns)))
8     ((and (numberp (cdr cns)) (atom (car cns))) (cons (car cns) (- (cdr
9       cns) 10)))
10    ((and (numberp (cdr cns)) (consp (car cns))) (cons (
11      all_minus_10_cons (car cns)) (- (cdr cns) 10)))
12    ((consp (car cns)) (cons (all_minus_10_cons (car cns)) (cdr cns)))
13    (t (cons (car cns) (cdr cns))))
14  )
15 ; используя функционал тарсар.
16 ; Для определения того, является ли x точечной парой, используется проверка (atom (cdr x))
17 ; вместо (consp x) из-за следующих результатов:
18 ; (listp '(k . 10))=(consp '(k . 10)) => T
19 ; (consp '(1 (K . 10)))=(listp '(1 (K . 10))) => T
20 (defun all_minus_10 (lst)
21   (mapcar
22     #'(lambda (x)
23       (cond
24         ((numberp x) (- x 10))
25         ((atom x) x)
26         ((atom (cdr x)) (all_minus_10_cons x))
27         (t (all_minus_10 x))
28       ))
29     lst)
30 )

```

```

28
29 ; рекурсивно
30 ; а) с помощью рекурсии, собирающей результат на выходе
31 (defun all_minus_10 (lst)
32   (cond
33     ((null lst) Nil)
34     ((numberp (car lst)) (cons (- (car lst) 10) (all_minus_10 (cdr lst)
35     )))
36     ((atom (car lst)) (cons (car lst) (all_minus_10 (cdr lst))))
37     ((atom (cdr (car lst))) (cons (all_minus_10_cons (car lst)) (
38     all_minus_10 (cdr lst)))))
39   (t (cons (all_minus_10 (car lst)) (all_minus_10 (cdr lst)))))
40 )
41 ; б) через "хвостовую рекурсию"
42 (defun all_minus_10_inner (lst result)
43   (cond
44     ((null lst) (reverse result))
45     ((numberp (car lst)) (all_minus_10_inner (cdr lst) (cons (- (car
46     lst) 10) result))))
47     ((atom (car lst)) (all_minus_10_inner (cdr lst) (cons (car lst)
48     result))))
49     ((atom (cdr (car lst))) (all_minus_10_inner (cdr lst) (cons (
50     all_minus_10_cons (car lst)) result))))
51     (t (all_minus_10_inner (cdr lst) (cons (all_minus_10_inner (car lst)
52     ) ()) result))))
53 )
54 (defun all_minus_10 (lst)
55   (all_minus_10_inner lst ()))
56 )
57 (all_minus_10 '(0 a "abc" (1 (k . 10)) 2/3 ((1 . f) . 10))) =>
58 (-10 A "abc" (-9 (K . 0)) -28/3 ((-9 . F) . 0))

```

## 2. Напишите функцию, которая умножает на заданное число-аргумент все числа из заданного списка-аргумента, когда:

1. Все элементы списка – числа.

```
1 ; используя функционал mapcar
2 (defun mult_all (lst num)
3   (mapcar #'(lambda (x) (* x num)) lst)
4 )
5 (mult_all '(0 10 -10 5.5 2/3) 2) => (0 20 -20 11.0 4/3)
```

2. Элементы списка – любые объекты.

```
1 ; используя функционал mapcar.
2 (defun mult_all (lst num)
3   (mapcar
4     #'(lambda (x) (cond ((numberp x) (* x num)) (x)))
5     lst
6   )
7 )
8
9 ; рекурсивно
10 ; а) с помощью рекурсии, собирающей результат на выходе
11 (defun mult_all (lst num)
12   (cond
13     ((null lst) Nil)
14     ((numberp (car lst)) (cons (* (car lst) num) (mult_all (cdr lst)
15       num)))
16     (t (cons (car lst) (mult_all (cdr lst) num))))
17 )
18
19 ; б) через "хвостовую рекурсию"
20 (defun mult_all_inner (lst result num)
21   (cond
22     ((null lst) (reverse result))
23     ((numberp (car lst)) (mult_all_inner (cdr lst) (cons (* (car lst)
24       num) result) num))
25     (t (mult_all_inner (cdr lst) (cons (car lst) result) num)))
26 )
27
28 (defun mult_all (lst num)
29   (mult_all_inner lst () num)
30 )
31
32
33 (mult_all '(0 a "abc" (1 k) 2/3) 2) => (0 A "abc" (1 K) 4/3)
```

### 3. Написать функцию, которая по своему списку-аргументу `lst` определяет является ли он палиндромом (то есть равны ли `lst` и `(reverse lst)`)

1. Проверка на равенство исходного списка и инвертированного исходного списка.

```
1 (defun is_palindrome (lst)
2   (equalp lst (reverse lst)))
3 )
```

2. Проверка на равенство инвертированной первой половины исходного списка и второй половины исходного списка (если список нечетной длины, то центральный элемент не попадает ни в первый, ни во второй список) (этот вариант и было предложено реализовать).

`(nthcdr n lst)` выполняет для списка `lst` операцию `cdr` `n` раз, и возвращает результат. `(floor n)` усекает значения по нижней границе. `(ceiling n)` усекает значения по верхней границе.

```
1 (defun first_n_rev (n lst res)
2   (cond
3     ((or (null lst) (= n 0)) res)
4     (t (first_n_rev (- n 1) (cdr lst) (cons (car lst) res))))
5   )
6 )
7
8 (defun is_palindrome (lst)
9   (let ((half_len (/ (length lst) 2)))
10    (equalp
11      (first_n_rev (floor half_len) lst ())
12      (nthcdr (ceiling half_len) lst)
13    )
14  )
15 )
```

3. Рекурсивно: сравнить первый и последний элемент исходного списка, первый и последний элемент исходного списка без первого и последнего элемента, и так далее (если длина списка нечетная, то центральный элемент ни с чем не сравнивается).

```
1 ; в данной задаче допустимо возвращать инвертированный список
2 (defun list_without_last (lst res)
3   (cond
4     ((null (cdr lst)) res)
5     (t (list_without_last (cdr lst) (cons (car lst) res))))
6   )
7 )
8
9 (defun is_palindrome (lst)
10  (cond
11    ((null (cdr lst)) t)
12    ((eql (car lst) (car (last lst))) ;т.к. (last '(1 2))=>(2)
13      (is_palindrome (list_without_last (cdr lst) ())))
14  )
15 )
```

Все варианты функций проверялись на следующих тестах:

```
1 (is_palindrome Nil) => T
2 (is_palindrome '(1)) => T
3 (is_palindrome '(1 2 3)) => NIL
4 (is_palindrome '(1 2 1)) => T
5 (is_palindrome '(1 2 3 1)) => NIL
6 (is_palindrome '(1 2 2 1)) => T
```

#### 4. Написать предикат `set-equal`, который возвращает `t`, если два его множества-аргумента содержат одни и те же элементы, порядок которых не имеет значения

Все элементы первого множества последовательно удаляются из обоих множеств. Если исходные множества эквиваленты, то в конце получим два пустых множества.

```
1 (defun set-equal (set1 set2)
2   (cond
3     ((null set1) (null set2)) ;3 тест
4     ((null set2) Nil) ;4 тест
5     (t (set-equal (cdr set1) (remove (car set1) set2))))
6   )
7 )
8
9 (set-equal '(1 2 3) '(1 2 3)) => T
10 (set-equal '() '()) => T
11 (set-equal '(1 2) '(1 2 3)) => NIL
12 (set-equal '(1 2) '(1)) => NIL
```

#### 5. Написать функцию которая получает как аргумент список чисел, а возвращает список квадратов этих чисел в том же порядке.

##### 1. Используя функционал `mapcar`

```
1 (defun squares (lst)
2   (mapcar #'(lambda (x) (* x x)) lst)
3 )
4
5 (squares '(0 10 -10 5.5 2/3)) => (0 100 100 30.25 4/9)
```

##### 2. Рекурсивно

```
1 ; а) с помощью рекурсии, собирающей результат на выходе
2 (defun squares (lst)
3   (cond
4     ((null lst) Nil)
5     (t (let ((x (car lst))) (cons (* x x) (squares (cdr lst))))))
6   )
7 )
8
9
10
11
```



```

12 ; б) через "хвостовую рекурсию"
13 (defun squares_inner (lst result)
14   (cond
15     ((null lst) (reverse result))
16     (t (squares_inner (cdr lst) (let ((x (car lst))) (cons (* x x)
17                                                                result))))
17   )
18 )
19
20 (defun squares (lst)
21   (squares_inner lst ()))
22 )

```

**6. Напишите функцию, select-between, которая из списка-аргумента, содержащего только числа, выбирает только те, которые расположены между двумя указанными границами-аргументами и возвращает их в виде списка (упорядоченного по возрастанию списка чисел (+ 2 балла)).**

(Границы не включительно)

1. Используя функционал remove-if-not, сортировка по неубыванию после выделения элементов, которые расположены между двумя указанными границами-аргументами.

Сортировка по неубыванию:

```

1 ; блочная (карманная, корзинная) сортировка
2 (defun my_sort (lst)
3   (cond ((null lst) Nil)
4     (t (nconc
5         (my_sort (remove-if-not (lambda (x) (> (car lst) x)) (cdr lst)))
6         (remove-if-not (lambda (x) (= (car lst) x)) lst)
7         (my_sort (remove-if-not (lambda (x) (< (car lst) x)) (cdr lst))))
8     )
9   )
10 )
11
12 ; сортировка выбором
13
14 ; в эту функцию не должен попадать пустой список
15 ; (лямбда-функция принимает строго 2 аргумента)
16 (defun list_max (lst)
17   (reduce
18     #'(lambda (a b) (cond ((> a b) a) (b)))
19     lst)
20 )

```

```

21
22 (defun my_sort_inner (lst res)
23   (cond
24     ((null lst) res)
25     (t (let*
26          (
27            (cur_max (list_max lst))
28            (lst_rest (remove cur_max lst :count 1))
29          )
30       (my_sort_inner lst_rest (cons cur_max res))
31     )
32   )
33 )
34 )
35
36 (defun my_sort (lst)
37   (my_sort_inner lst ())
38 )

```

### Функция

```

1 (defun select-between-inner (lst a b)
2   (remove-if-not #'(lambda (x) (< a x b)) lst)
3 )
4
5 (defun select-between (lst a b)
6   (cond
7     ((< a b) (select-between-inner lst a b))
8     (t (select-between-inner lst b a))
9   )
10 )
11 (defun select-between-sorted (lst a b)
12   (my_sort (select-between lst a b))
13 )

```

2. Отсортировать список по невозрастанию. Оставить часть списка, начиная с первого элемента, меньшего верхней границы. Инвертировать список (получаем отсортированный по неубыванию). Оставить часть списка, начиная с первого элемента, большего нижней границы.

Блочная сортировка по невозрастанию:

```

1 (defun my_sort (lst)
2   (cond ((null lst) Nil)
3     (t (nconc
4         (my_sort (remove-if-not (lambda (x) (< (car lst) x)) (cdr lst)))
5         (remove-if-not (lambda (x) (= (car lst) x)) lst)
6         (my_sort (remove-if-not (lambda (x) (> (car lst) x)) (cdr lst))))
7   ))
8 )

```

## Функция

```
1 (defun leave_upper (lst b)
2   (member (find-if #'(lambda (x) (< x b)) lst) lst)
3 )
4
5 (defun leave_lower (lst a)
6   (member (find-if #'(lambda (x) (> x a)) lst) lst)
7 )
8
9 (defun select-between-sorted-inner (lst a b)
10  (leave_lower
11    (reverse
12      (leave_upper
13        (my_sort lst)
14        b)
15      )
16    a)
17 )
18
19 (defun select-between-sorted (lst a b)
20  (cond
21    ((< a b) (select-between-sorted-inner lst a b))
22    (t (select-between-sorted-inner lst b a))
23  )
24 )
25
26 (select-between-sorted '(1 3 2 4 5 7 6) 1 7) => (2 3 4 5 6)
27 (select-between-sorted '(1 3 2 4 5 7 6) 6 2) => (3 4 5)
```

(Рекурсивно – см. 7 лабораторная работа, номер 5)

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

1. С помощью функционалов `mapcar` и `mapcan`

```
1 (defun decart (lstx lsty)
2   (mapcan
3     #'(lambda (x) (mapcar
4       #'(lambda (y) (cons x (cons y Nil)))
5       lsty
6     ))
7     lstx)
8 )
9 (decart '(a b) '(1 2)) => ((a 1) (a 2) (b 1) (b 2))
```

## 2. Рекурсивно

```
1 ; по принципу move-to с лекции
2 ; составляет список из каждого элемента lst и cons-ячейки cns
3 (defun decart_row (lst cns res)
4   (cond
5     ((null lst) res)
6     (t (decart_row (cdr lst) cns (cons (cons (car lst) cns) res))))
7   )
8 )
9
10 ; по принципу move-to с лекции
11 ; (nconc вместо cons, lsty неизменен, lstx уменьшается в размере)
12 (defun decart_inner (lstx lsty res)
13   (cond
14     ((null lstx) res)
15     (t (decart_inner
16         (cdr lstx)
17         lsty
18         (nconc
19           (decart_row lsty (cons (car lstx) Nil) ())
20           res
21         )
22       )
23     )
24   )
25 )
26
27 (defun decart (lstx lsty)
28   (decart_inner lstx lsty ())
29 )
30
31 (decart '(a b) '(1 2)) => ((2 B) (1 B) (2 A) (1 A))
```

## 8. Почему так реализовано reduce, в чем причина?

Функционал reduce выполняет следующее преобразование исходного списка  $L \Rightarrow (e1\ e2\ \dots\ en)$  с использованием начального значения  $A$  и бинарной операции-функции  $F$ :  $(\text{reduce } F\ L\ A) = (F(\dots(F(F\ A\ e1)\ e2))\dots en)$ . Можно сказать, что значение  $A$  логически добавляется в начало списка  $L$ .

- Если список содержит ровно один элемент и начальное значение не задано, то этот элемент возвращается, а функция не вызывается.
- Если список пуст и задано начальное значение, то возвращается начальное значение, а функция не вызывается.
- Если список пуст и начальное значение не задано, то функция вызывается без аргументов, и reduce возвращает то, что вернет функция. Это единственный случай, когда функция вызывается с другим количеством аргументов, кроме двух.

В требованиях сказано, что функция должна принимать в качестве аргументов два элемента последовательности или результаты объединения этих элементов. Функция также должна быть способна не принимать никаких аргументов.

```
1 (reduce #' + ()) => 0
2 ; в задании указано, что (reduce #' + 0) => 0, но
3 (reduce #' + 0) => Value of 0 in (REDUCE #' + 0) is 0, not a SEQUENCE.
4 ; возможно, имелось в виду следующее:
5 (reduce #' + () :initial-value 0) => 0
6 ; или
7 (reduce #' + '(0)) => 0
```

$(\text{reduce } \#'+ () : \text{initial-value } 0)$ : список пуст, начальное значение указано, следовательно, функция  $+$  не вызывается, и reduce возвращает начальное значение.

$(\text{reduce } \#'+ '(0))$ : список состоит из одного элемента, начальное значение не указано, следовательно, функция  $+$  не вызывается, и reduce возвращает единственный элемент списка.

$(\text{reduce } \#'+ ())$ : список пуст, начальное значение не указано, следовательно, функция  $+$  вызывается без аргументов, и reduce возвращает то, что вернет функция  $+$ .  $(+) \Rightarrow 0$ , поэтому и  $(\text{reduce } \#'+ ()) \Rightarrow 0$ .

Примечание:  $(\text{reduce } \#'^* ()) \Rightarrow$  по той же причине, но для функций  $-$  и  $/$  такой вызов осуществить невозможно, так как они не могут принимать ноль аргументов.

9. 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

```
1 ; с использованием функционалов mapcar и reduce
2 (defun sum_of_legths (ll)
3   (reduce #'+ (mapcar #'length ll))
4 )
5
6 ; рекурсивно
7 (defun sum_of_legths_inner (ll sum)
8   (cond
9     ((null ll) sum)
10    (t (sum_of_legths_inner (cdr ll) (+ sum (length (car ll))))))
11 )
12 )
13
14 (defun sum_of_legths (ll)
15   (sum_of_legths_inner ll 0)
16 )
17
18 (sum_of_legths '()) => 0
19 (sum_of_legths '(())) => 0
20 (sum_of_legths '((1 2))) => 2
21 (sum_of_legths '((1 2) (3 4))) => 4
22 (sum_of_legths '((1 2) (3) (4 5))) => 5
```