



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
22
23
24
25
26 ; б) через "хвостовую рекурсию"
27
28 (defun all_minus_10_inner (lst result)
29   (cond
30     ((null lst) (reverse result))
31     ((numberp (car lst)) (all_minus_10_inner (cdr lst) (cons (- (car
32       lst) 10) result)))
33     (t (all_minus_10_inner (cdr lst) (cons (car lst) result))))
34   )
35 )
36 (defun all_minus_10 (lst)
37   (all_minus_10_inner lst ()))
38 )
39
40
41 (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     ((numberp (cdr cns)) (cons (car cns) (- (cdr cns) 10)))
9     (t (cons (car cns) (cdr cns))))
10  )
11
12 ; используя функционал тарсар.
13 ; Для определения того, является ли x точечной парой, используется проверка (atom (cdr x))
14 ; вместо (consp x) из-за следующих результатов:
15 ; (listp '(k . 10))=(consp '(k . 10)) => T
16 ; (consp '(1 (K . 10)))=(listp '(1 (K . 10))) => T
17 (defun all_minus_10 (lst)
18   (mapcar
19     #'(lambda (x)
20       (cond
21         ((numberp x) (- x 10))
22         ((atom x) x)
23         ((atom (cdr x)) (all_minus_10_cons x))
24         ((listp x) (all_minus_10 x))

```

```

24
25     ))
26     lst)
27 )
28
29
30 ; рекурсивно
31 ; а) с помощью рекурсии, собирающей результат на выходе
32 (defun all_minus_10 (lst)
33   (cond
34     ((null lst) Nil)
35     ((numberp (car lst)) (cons (- (car lst) 10) (all_minus_10 (cdr lst)
36     )))
37     ((atom (car lst)) (cons (car lst) (all_minus_10 (cdr lst))))
38     ((atom (cdr (car lst))) (cons (all_minus_10_cons (car lst)) (
39     all_minus_10 (cdr lst))))
40     (t (cons (all_minus_10 (car lst)) (all_minus_10 (cdr lst)))))
41   )
42 )
43
44 ; б) через "хвостовую рекурсию"
45 (defun all_minus_10_inner (lst result)
46   (cond
47     ((null lst) (reverse result))
48     ((numberp (car lst)) (all_minus_10_inner (cdr lst) (cons (- (car
49     lst) 10) result)))
50     ((atom (car lst)) (all_minus_10_inner (cdr lst) (cons (car lst)
51     result)))
52     ((atom (cdr (car lst))) (all_minus_10_inner (cdr lst) (cons (
53     all_minus_10_cons (car lst)) result)))
54     (t (all_minus_10_inner (cdr lst) (cons (all_minus_10_inner (car lst
55     ) ()) result)))
56   )
57 )
58
59 (defun all_minus_10 (lst)
60   (all_minus_10_inner lst ()))
61 )
62
63 (all_minus_10 '(0 a "abc" (1 (k . 10)) 2/3)) => (-10 A "abc" (-9 (K .
64   0)) -28/3)

```

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) num)))
15     (t (cons (car lst) (mult_all (cdr lst) num))))
16 )
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) num) result) num))
24     (t (mult_all_inner (cdr lst) (cons (car lst) result) num)))
25 )
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 первой половины исходного списка и инвертированной второй половины исходного списка (если список нечетной длины, то центральный элемент не попадает ни в первый, ни во второй список).

`(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 (n lst)
2   (cond
3     ((or (null lst) (= n 0)) Nil)
4     (t (cons
5         (car lst)
6         (first_n (- n 1) (cdr lst))
7       )
8   )
9 )
10 )
11
12 (defun is_palindrome (lst)
13   (let ((half_len (/ (length lst) 2)))
14     (equalp
15       (first_n (floor half_len) lst)
16       (reverse (nthcdr (ceiling half_len) lst)))
17   )
18 )
19 )
```

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 (defun list_without_last (lst)
2   (cond
3     ((null (cdr lst)) Nil)
4     (t (cons
5         (car lst)
6         (list_without_last (cdr lst))
7       )
8   )
9 )
10 )
11
12 (defun is_palindrome (lst)
13   (cond
14     ((null (cdr lst)) t)
15     ((eql (car lst) (car (last lst))) ;т.к. (last '(1 2))=>(2)
16      (is_palindrome (list_without_last (cdr lst))))
17   )
18 )
```

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 )

```

TODO и марсан6. 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)).

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     )
8   lstx
9 )
10 )
11
12 (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

```

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

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

`(reduce #' + ())`: список пуст, начальное значение не указано, следовательно, функция `+` вызывается без аргументов, и `reduce` возвращает то, что вернет функция `+`. `(+) => 0`, поэтому и `(reduce #' + ()) => 0`.

Примечание: `(reduce #' * ()) =>` 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
```

Литература