



КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Тема Использование функционалов и рекурсии

Студент Пересторонин П.Г.

Группа ИУ7-63Б

Оценка

Преподаватель Толпинская Н. Б.

Москва — 2021 г.

Оглавление

1	Задания	2
1.1	Написать функцию, которая выбирает из заданного списка только те числа, которые больше 1 и меньше 10 (Вариант: между 2 заданными границами)	2
1.1.1	С помощью функционалов	2
1.1.2	Рекурсия	2
1.2	Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов	3
1.2.1	С помощью функционалов	3
1.2.2	Рекурсия	3
1.3	Почему так реализовано <code>reduce</code> , в чем причина?	3
1.4	Пусть <code>list-of-lists</code> — список, состоящий из списков. Написать функцию, которая вычисляет сумму длин всех элементов <code>list-of-lists</code> , то есть, например, для аргумента <code>((1 2) (3 4))</code> <code>-> 4</code>	4
1.4.1	С помощью функционалов	4
1.4.2	Рекурсия	4
1.5	Используя рекурсию, написать функцию, которая по исходному списку строит список квадратов чисел смешанного структурированного списка	5
2	Ответы на вопросы к лабораторной работе	6
2.1	Классификация рекурсивных функций	6

1 Задания

1.1 Написать функцию, которая выбирает из заданного списка только те числа, которые больше 1 и меньше 10 (Вариант: между 2 заданными границами)

Реализация для случая, когда между 2 заданными границами

1.1.1 С помощью функционалов

```
1 (defun list-between-only (from to lst)
2   (mapcan #'(lambda (el)
3               (if (and (< el to) (> el from)) (list el)))
4   lst))
```

1.1.2 Рекурсия

```
1 (defun list-btwn-only-internal (from to lst acc)
2   (cond ((null lst) acc)
3         (T (let ((head (car lst)))
4               (cond ((and (< head to) (> head from))
5                     (list-btwn-only-internal from to (cdr lst) (cons head acc)))
6                 (T (list-btwn-only-internal from to (cdr lst) acc))))))
7 (defun list-btwn-only-rec (from to lst)
8   (reverse (list-btwn-only-internal from to lst nil)))
```

1.2 Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов

1.2.1 С помощью функционалов

```
1 (defun list-mul (lst1 lst2)
2   (mapcan #'(lambda (outer-el)
3             (mapcar #'(lambda (inner-el)
4                         (cons outer-el inner-el))
5                           lst2))
6     lst1))
```

1.2.2 Рекурсия

```
1 (defun list-mul-internal (cur-lst1 src-lst1 lst2 acc)
2   (cond ((null lst2) acc)
3         ((null cur-lst1) (list-mul-internal src-lst1 src-lst1 (cdr lst2) acc))
4         (T (list-mul-internal (cdr cur-lst1) src-lst1 lst2 (cons (cons (car cur-lst1)
5                                                                           (car lst2)) acc))))))
5 (defun list-mul-rec (lst1 lst2)
6   (and lst1 lst2 (reverse (list-mul-internal lst1 lst1 lst2 nil))))
```

1.3 Почему так реализовано reduce, в чем причина?

`(reduce '+ ()) -> 0`

Поведение в данном примере обусловлено работой функции `+`. Эта функция — функционал, который при 0 количестве аргументов возвращает значение 0. Если подать на вход `reduce` функцию, которая не может обработать 0 аргументов (например, математическая функция `cons`), то вызов `reduce` с пустым списком в качестве второго аргумента вернет ошибку

(invalid number of arguments: 0). При этом, если подано более одного аргумента, то `reduce` выполняет следующие действия:

1. сохраняет первый элемент списка в область памяти (для определенности назовем ее `acc`);
2. для всех остальных элементов списка выполняет переданную в качестве первого аргумента функцию, подавая на вход 2 аргумента (`acc` и очередной элемент списка) и сохраняя результат в `acc`.

Пример упрощенной реализации `reduce` (в данной реализации опущены проверки аргументов):

```
1 (defun my-reduce-internal (func lst acc)
2   (cond ((null lst) acc)
3         (t (my-reduce-internal func (cdr lst) (funcall func acc (car lst))))))
4 (defun my-reduce (func lst)
5   (cond ((null lst) (funcall func))
6         (t (my-reduce-internal func (cdr lst) (car lst)))))
```

1.4 Пусть `list-of-lists` — список, состоящий из списков. Написать функцию, которая вычисляет сумму длин всех элементов `list-of-lists`, то есть, например, для аргумента `((1 2) (3 4))` $\rightarrow 4$

1.4.1 С помощью функционалов

```
1 (defun sum-lens (list-of-lists)
2   (reduce #'(lambda (acc lst) (+ acc (length lst)))
3         (cons 0 list-of-lists)))
```

1.4.2 Рекурсия

```
1 (defun sum-lens-rec-internal (lol acc)
2   (cond ((null lol) acc)
3         (T (sum-lens-rec-internal (cdr lol) (+ acc (length (car lol)))))))
4 (defun sum-lens-rec (list-of-lists)
5   (and (listp list-of-lists) (sum-lens-rec-internal list-of-lists 0)))
```

1.5 Используя рекурсию, написать функцию, которая по исходному списку стоит список квадратов чисел смешанного структурированного списка

```
1 (defun square (x) (* x x))
2 (defun square-list-internal (lst acc)
3   (cond ((null lst) acc)
4         (T (square-list-internal (cdr lst) (cons (square (car lst)) acc)))))
5 (defun square-list (lst)
6   (reverse (square-list-internal lst nil)))
```

2 Ответы на вопросы к лабораторной работе

2.1 Классификация рекурсивных функций

Рекурсия — ссылка на описываемый объект во время его описания.

Классификация рекурсивных функций:

- Простая (рекурсивный вызов — единственный);
- Второго порядка (несколько рекурсивных вызовов);
- Взаимная рекурсия (используются несколько рекурсивных функций, которые могут друг друга вызывать).
- Хвостовая рекурсия (при очередном вызове рекурсивной функции все действия до входа выполнены, а при выходе ничего более делать не приходится);
- Дополняемая рекурсия (результат рекурсии используется, как аргумент некоторой другой функции (которую называют *дополняемой функцией*); частный случай — **cons**-дополняемая рекурсия).