|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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»\_\_\_\_\_\_\_\_\_\_\_

**Лабораторная работа № 9**

|  |  |
| --- | --- |
| **Дисциплина** Функциональное и логическое программирование  **Тема** \_Рекурсивные функции\_  **Студент** \_Ильясов И. М.\_  **Группа** \_ИУ7-63Б\_  **Оценка (баллы)** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Преподаватель** \_Толпинская Н. Б.\_ |  |

Москва, 2020 г.

**Задание 1.** Написать предикат set-equal, который возвращает t, если два его множества-аргумента содержат одни и те же элементы, порядок которых не имеет значения.

На вход функции подаются два списка-аргумента. Проверяется, является ли каждый список подмножеством другого.

subsetp – предикат, который возвращает T в случае если каждый элемент списка lst1 встречается в списке lst2. Иначе возвращается Nil.

set-difference – возвращает список элементов списка lst1, которых нет в списке lst2.

(defun set-equal (x y)

(and (subsetp x y) (subsetp y x))

)

ИЛИ

(defun set-equal (a b)

(if (> (length a) (length b))

(if (NULL (set-difference a b)) T Nil)

(if (NULL (set-difference b a)) T Nil)

)

)

**Задание 2.** Напишите необходимые функции, которые обрабатывают таблицу из точечных пар: (страна. столица), и возвращают по стране - столицу, а по столице – страну.

Рекурсия:

На вход подается список точечных пар и страна/столица. При найденном значении возвращается соответствующее значение столицы/страны, иначе Nil.

(defun find\_country\_cons (cons\_cc city)

(cond

((null cons\_cc) Nil)

((equal (cdr (car cons\_cc)) city) (car (car cons\_cc)))

( T (find\_country\_cons (cdr cons\_cc) city))

)

)

(defun find\_city\_cons (cons\_cc country)

(cond

((null cons\_cc) Nil)

((equal (car (car cons\_cc)) country) (cdr (car cons\_cc)))

(T (find\_city\_cons (cdr cons\_cc) country))

)

)

Функционалы:

На вход подается список точечных пар и столица/страна. При найденном значении возвращается соответствующее значение столицы/страны, иначе Nil.

(defun cons\_city (cons\_cc country)

(reduce #'(lambda (a b) (or a b))

(mapcar #'(lambda (el)

(and (equal (car el) country) (cdr el))

) cons\_cc

)

)

)

(defun cons\_country (cons\_cc city)

(reduce #'(lambda (a b) (or a b))

(mapcar #'(lambda (el)

(and (equal (cdr el) city) (car el))

) cons\_cc

)

)

)

**Задание 3.** Напишите функцию, которая умножает на заданное число-аргумент все числа

из заданного списка-аргумента, когда

a) все элементы списка – числа,

6) элементы списка – любые объекты.

На вход функции подается список элементов и число, на которое каждый из элементов нужно умножить.

Функционалы:

С использованием функционала mapcar и лямбда-функции производится умножение каждого элемента списка lst на число k.

(defun multiplication (lst k)

(mapcar #'(lambda (x) (\* x k)) lst)

)

С использованием функционала mapcar и лямбда-функции производится умножение каждого элемента списка lst на число k (при этом сначала производится проверка того, что элемент списка является числом – numberp).

(defun multiplication (lst k)

(mapcar #'(lambda (x) (if (numberp x) (\* x k) x)) lst)

)

Рекурсия:

В этой реализации функция рекурсивно применяется для оставшегося хвоста списка. Пока список не будет равен Nil, создается список из обновленных значений.

(defun multiplication (lst k)

(if lst

(cons (\* (car lst) k) (multiplication (cdr lst) k))

)

)

В этой реализации функция рекурсивно применяется для оставшегося хвоста списка. Пока список не будет равен Nil, создается список из обновленных значений (при этом проводится проверка, является ли элемент числом).

(defun multiplication (lst k)

(if lst

(cons

(if (numberp (car lst)) (\* (car lst) k) (car lst))

(multiplication (cdr lst) k)

)

)

)

**Задание 4.** Напишите функцию, которая уменьшает на 10 все числа из списка аргумента этой функции.

Функционалы:

С использованием функционала mapcar и лямбда-функции производится вычитание из каждого элемента, который является числом, десяти.

(defun minus\_10 (lst)

(mapcar #'(lambda (elem)

(cond

((numberp elem) (- elem 10))

(T elem))

) lst

)

)

Рекурсия:

(defun minus\_10 (lst)

(if lst

(

(lambda (elem result)

(cons

(if (numberp elem) (- elem 10) elem) result

)

)

(car lst)

(minus\_10 (cdr lst))

)

)

)

**Задание 5.** Написать функцию, которая возвращает первый аргумент списка-аргумента, который сам является непустым списком.

На вход функции подается список. Возвращается первый аргумент списка-аргумента, который не является пустым списком.

Функционалы:

(defun first-list (lst)

(cond

((null lst) nil)

((listp (first lst)) (first lst))

(T (func (rest lst)))

)

)

Рекурсия:

(defun first-list (lst)

(cond

((null lst) nil)

((atom (car lst)) (first-list (cdr lst)))

(T (car lst)))

)

**Задание 6.** Написать функцию, которая выбирая из заданного списка только те числа, которые больше 1 и меньше 10. (Вариант: между двумя заданными границами).

На вход функции принимаются список и границы диапазона.

Функционалы:

(defun between (lst a b)

(remove nil

(mapcar #'(lambda (elem)

(if (numberp elem)

(if (< elem b)

(if (> elem a) elem))

)

) lst

)

)

)

Рекурсия:

(defun between (lst a b)

(if lst ((lambda (elem result) (if (numberp elem) (if (< a elem b) (cons elem result) result))) (car lst) (between (cdr lst) a b)))

)

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

На вход подаются два списка. Вычисляется декартово произведение двух списков-аргументов.

Функционалы:

С использованием функционала mapcar, mapcan и лямбда-функции производится вычисление декартова произведения списков-аргументов. В лямбда-функции создается список, пробегаясь для каждого элемента A по всем элементам B

(defun func (A B)

(mapcan #'(lambda (A)

(mapcar #'(lambda (B)

(list A B)

) B

)

) A

)

)

**Задание 8.** Почему так реализовано reduce, в чем причина?

(reduce #'+ ()) -> 0  
(reduce #'\* ()) -> 1  
  
Сначала функция проверяет список-аргумент. Если он пуст, возвращается значение

функции при отсутствии аргументов. Также reduce использует аргумент :initial-value. Этот аргумент определяет значение, к которому будет применена функция при обработке первого элемента списка-аргумента. Если список-аргумент пуст, то будет возвращено значение initial-value. Для суммы значение по умолчанию :initial-value равно 0, для умножения – 1.

**Ответы на вопросы**

* Способы организации повторных вычислений в Lisp.

Повторные вычисления можно организовать при помощи функционалов и рекурсии.

Существует 2 типа функционалов:

1. Применяющие (apply, funcall)
2. Отображающие (mapcar, maplist, reduce)

* Различные способы использования функционалов.
* mapcar – функция func применяется к головам первым элементам списков, затем ко вторым и т.д., и результаты применения собираются в результирующий список.
* maplist – func применяется к “хвостам” списков, начиная с полного списка.
* mapcan, mapcon – аналогичны mapcar и maplist, используется память исходных данных, не работают с копиями.
* (reduce #’func lst). reduce – функция func применяется каскадным образом (к первым двум, затем к результату и следующему и так далее).
* Что такое рекурсия? Способы организации рекурсивных функций.

Рекурсия — это ссылка на определяемый объект во время его определения.

Рекурсивная функция вызывает саму себя. Вопрос организации рекурсии — это вопрос эффективного способа организации рекурсии. В Lisp существует классификация рекурсивных функций:

1. простая рекурсия - один рекурсивный вызов в теле
2. рекурсия первого порядка - рекурсивный вызов встречается несколько раз

взаимная рекурсия - используется несколько функций, рекурсивно вызывающих друг друга.

В силу возможной сложности и разнообразия постановок задач, возможны комбинации и усложнения приведенных групп функций. Существуют типы рекурсивных функций: хвостовая, дополняемая, множественная, взаимная рекурсия и рекурсия более высокого порядка.

* Способы повышения эффективности реализации рекурсии.

Использование хвостовой рекурсии. Если условий выхода несколько, то надо думать о порядке их следования. Некачественный выход из рекурсии может привести к переполнению памяти из-за "лишних" рекурсивных вызовов.

Преобразование не хвостовой рекурсии в хвостовую, возможно путем использования дополнительных параметров. В этом случае необходимо использовать функцию-оболочку для запуска рекурсивной функции с начальными значениями дополнительных параметров.