|  |  |
| --- | --- |
| 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**

|  |  |
| --- | --- |
| **Тема** Lisp  **Студент** Белоусова Ю.С.  **Группа** ИУ7-61Б  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель** Толпинская Н.Б. |  |

Москва.

2020 г.

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

a, b – входные множества, x1,y1,x2,y2, a1 – служебные переменные, el – отдельный элемент. При реализации функционалами, внешний mapcar перебирает элементы первого множества, внутренний – второго. Внутренний reduce сводит результаты внутреннего mapcar к T, если нашлось совпадение текущего элемента первого списка с хотя бы одним элементом второго, иначе к NIL. Получается список, хранящий для каждого элемента первого множества информацию о его вхождении во второе. Внешний reduce сводит результаты внешнего mapcar к T, если для всех элементов первого списка нашлось совпадение во втором, иначе к NIL.

Для рекурсивной реализации были созданы функции contains (определяет, содержит ли данный список данный элемент) и rec\_includep, определяющая, входит ли полностью первое множество во второе.

В обеих реализациях остается проверить, что a включено в b и b включено в a.

; Функционалы

(defun set-equal-func (a b)

(defun func\_includedp (a b)

(reduce (lambda (x1 y1) (and x1 y1))

(mapcar (lambda (a1) (reduce (lambda (x2 y2) (or x2 y2))

(mapcar (lambda (el) (equalp a1 el)) b)))

a

)

)

)

(and (func\_includedp a b) (func\_includedp b a))

)

; Рекурсия

(defun set-equal-rec(a b)

(defun rec\_includep(a b)

(defun contains(lst el)

(cond

((null lst) Nil)

((equal el (car lst)) T)

(t (contains (cdr lst) el))

)

)

(cond

((not (contains b (car a))) nil)

((null (cdr a)) T)

(t (rec\_includep (cdr a) b))

)

)

(and (rec\_includep a b) (rec\_includep b a))

)

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

При реализации функционалами, mapcar перебирает все элементы данного списка, если голова очередного элемента совпадает с заданным значением, возвращается его хвост. Если совпадает хвост, возвращается голова. Полученный список сокращается с помощью reduce (до первого непустого элемента)

При рекурсивной реализацией аналогичная проверка выполняется для очередного элемента, если ни голова, ни хвост не совпали, возвращается результат рекурсивного вызова для исходного списка без головы (cdr).

(defun make\_pairlist (a b)

(mapcar #'cons a b))

(set 'countries '(Russia The\_USA The\_UK Belarus))

(set 'capitals '(Moscow Washington London Minsk))

; (set 'pairs (make\_pairlist countries capitals))

; Функционалы

(defun find\_pair\_func(val lst)

(reduce (lambda (a b)

(or a b))

(mapcar (lambda (pair)

(cond

((equal (car pair) val) (cdr pair))

((equal (cdr pair) val) (car pair)))

)

lst

)

)

)

; Рекурсия

(defun find\_pair\_rec(val lst)

(let ((pair (car lst)))

(cond

((not pair) nil)

((equal (car pair) val) (cdr pair))

((equal (cdr pair) val) (car pair))

(T (find\_pair\_func val (cdr lst)))

)

)

)

1. 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**.

Вариант а):

; Функционалы

(defun mul\_nums\_func (lst num)

(mapcar (lambda (el)

(\* el num))

lst

)

)

; Рекурсия

(defun mul\_nums\_rec (lst num)

(defun muler (mul lst res)

(cond

((null lst) res)

(t (muler mul (cdr lst) (append res (cons (\* mul (car lst)) nil)))))

)

(muler num lst nil)

)

Вариант б): добавляется проверка. Если очередной элемент – число, умножение выполняется, иначе оно пропускается

; Функционалы

(defun mul\_nums\_func\_b (lst num)

(remove nil (mapcar (lambda (el)

(cond ((numberp el)(\* el num))))

lst

))

)

; Рекурсия

(defun mul\_nums\_rec\_b (lst num)

(defun muler (mul lst res)

(cond

((null lst) res)

(t (muler mul (cdr lst) (append res (cond

((numberp (car lst)) (cons (\* mul (car lst)) nil))

(t nil)

)))

)

)

)

(muler num lst nil)

)

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

; Функционалы

(defun reducer\_func (lst)

(mapcar (lambda (el) (- el 10)) lst)

)

; Рекурсия

(defun reducer\_rec (lst)

(defun inner\_reducer (lst res)

(cond

((not lst) res)

(t (inner\_reducer (cdr lst) (append res (cons (- (car lst) 10) nil))))

)

)

(inner\_reducer lst nil)

)

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

При реализации функционалами, каждый элемент проходит проверку, является ли он списком и при этом не NIL. Благодаря свойствам and, возвращается либо nil, либо сам элемент (если проверка пройдена), затем, с помощью reduce + or, список сокращается до первого непустого элемента.

На всякий случай, рекурсивная реализация выполнена в двух вариантах: с помощью cond и с внешним макросом or. В обеих рекурсивных реализациях очередной элемент подвергается аналогичной проверке и, если проходит ее, возвращается в качестве результата, иначе возвращается результат рекурсивного вызова для оставшейся части списка.

; Функционалы

(defun first\_list\_func (lst)

(reduce (lambda (a b) (or a b))

(mapcar (lambda (x) (and (listp x) x)) lst)

)

)

; Рекурсия

(defun first\_list\_rec (lst)

(let ((el (car lst))) (cond

((and (listp el) el) el)

(t (first\_list\_rec (cdr lst)))

))

)

(defun first\_list\_rec+ (lst)

(let ((el (car lst))) (or

(and (listp el) el)

(first\_list\_rec (cdr lst)) ; or является макросом, раскрываясь, )) ; он выполнит сначала and,

) ; а потом рекурсивный вызов, то есть, рекурсия все еще хвостовая

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

mapcar возвращает исходный список, в котором все непрошедшие проверку элементы заменены на nil. Эти элементы удаляются с помощью remove

; Функционалы

(defun selector\_func (lst a b)

(remove nil (mapcar (lambda (el) (and (numberp el) (> el a) (< el b) el)) lst))

)

; Рекурсия

(defun selector\_rec (lst a b)

(defun inner\_selector (lst a b res)

(cond

((not lst) res)

(t (let ((el (car lst)))

(inner\_selector (cdr lst) a b (append res (and (numberp el) (> el a) (< el b) (cons el nil))))))

)

)

(inner\_selector lst a b nil)

)

1. Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов.

; Функционалы

(defun decart\_func (x y)

(mapcan (lambda (x)

(mapcar (lambda (y)

(list x y))

y)

)

x)

)

При рекурсивной реализации el\_by\_list формирует список списков, описывающий пары очередного элемента первого списка со всеми элементами второго. Выполнив ее для всех элементов первого списка и объединив результаты, получим искомое декартово произведение.

; Рекурсия

(defun decart\_rec (x y)

(defun inner\_decart (x y res)

(defun el\_by\_list (el lst res)

(cond

((not lst) res)

(t (el\_by\_list el (cdr lst) (append res (list (list el (car lst))))))

)

)

(cond

((not x) res)

(t (inner\_decart (cdr x) y (append res (el\_by\_list (car x) y nil))))

)

)

(inner\_decart x y nil)

)

1. Почему так реализовано reduce, в чем причина?  
   (reduce #’+()) -> 0   
   (reduce #'\* ()) -> 1

Для поддержки списков, в которых меньше двух элементов, функция reduce использует initial-value, (чтобы получить сумму (5), нужно сложить 5 и 0, чтобы получить произведение (5), 5 нужно умножить на 1), в случае, если в списке элементов нет, возвращается initial-value.

**Вывод**

Правильно организованная рекурсивная реализация является более оптимальной, чем реализация на функционалах. Во-первых, хвостовая рекурсия легко оптимизируема и большинство систем выполняет эту оптимизацию автоматически, а во-вторых, функционалы не обладают досрочным выходом и перебирают список до конца, даже если результат уже известен, в то время. как в рекурсии, при нахождении результата. можно сразу его вернуть.

**Ответы на вопросы**

* Способы организации повторных вычислений в Lisp,

Для организации многократных вычислений в Lisp могут быть использованы функционалы — функции, которые особым образом обрабатывают свои аргументы, а также рекурсию. Многие вычисления в функциональных языках программирования естественно выражать в виде рекурсивных функций, т. к. в Lisp используются рекурсивно определенные структуры. Кроме них, в lisp можно организовать повторные вычисления с помощью циклов, но они также сводятся к рекурсии.

* Различные способы использования функционалов,

Функционалы используются для вычисления одной функции с разными входными значениями. Например, для применения какой-то операции ко всему списку. Некоторые функционалы используют повторные вычисления, что позволяет организовывать накие вычисления на этих функционалах (например, mapcar)

* Что такое рекурсия? Способы организации рекурсивных функций,

Рекурсия — это ссылка на определяемый объект во время его определения. Существуют типы рекурсивных функций: хвостовая, дополняемая, множественная, взаимная рекурсия и рекурсия более высокого порядка.

* Способы повышения эффективности реализации рекурсии.

Для повышения эффективности реализации рекурсии нужно использовать именно хвостовую рекурсию, так как она легко оптимизируема и большинство систем способны производить такую оптимизацию автоматически.