|  |  |
| --- | --- |
| 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 № 10**

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

Москва.

2020 г.

1. 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

Начальная сумма равна 0.

Если поступивший список пуст, возвращаем накопленную сумму, иначе, если элемент – атом, его длина равна 1, возвращаем значение рекурсивного вызова от остатка списка и увеличенной на 1 суммы. Иначе возвращаем результат рекурсивного вызова от остатка списка и увеличенной на длину списка-элемента суммы.

(defun sublists\_sum (lst)

(defun tailed\_sum (lst sum)

(cond

((not lst) sum)

((atom (car lst)) (tailed\_sum (cdr lst) (+ sum 1)))

(T (tailed\_sum (cdr lst) (+ sum (length (car lst)))))

)

)

(tailed\_sum lst 0)

)

1. Написать рекурсивную версию (с именем reg-add) вычисления суммы чисел заданного списка.

Начальная сумма равна 0.

Если поступивший список пуст, возвращаем накопленную сумму, иначе возвращаем результат рекурсивного вызова от остатка списка и суммы, увеличенной на значение элемента

(defun reg-add (lst)

(defun inner\_adder (lst sum)

(cond

((not lst) sum)

(t (inner\_adder (cdr lst) (+ sum (car lst))))

)

)

(inner\_adder lst 0)

)

1. Написать рекурсивную версию с именем recnth функции nth.

n выступает в роли счетчика. Если n равен 0, мы достигли искомого элемента, возвращаем его, если n > 0, возвращаем результат рекурсивного вызова от n-1 и остатка списка. Если n < 0, возвращаем nil (или можно вернуть 'error)

(defun recnth (n lst)

(cond

((= n 0) (car lst))

((> n 0) (recnth (- n 1) (cdr lst)))

(t nil)

)

)

1. Написать рекурсивную функцию alloddr, которая возвращает t когда все элементы списка нечетные.

Если список пуст, нечетных элементов не было встречено, возвращаем T. Иначе, если очередной элемент четный, возвращаем nil, иначе возвращаем результат рекурсивного вызова для остатка списка.

(defun alloddr (lst)

(cond

((not lst) t)

((evenp (car lst)) nil)

(t (alloddr (cdr lst)))

)

)

1. Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка – аргумента

Если хвост списка пуст, данный элемент является последним, возвращаем его, иначе возвращаем результат рекурсивного вызова для остатка списка.

(defun reclast(lst)

(cond

((not (cdr lst)) (car lst))

(t (reclast (cdr lst)))

)

)

1. Написать рекурсивную функцию, относящуюся к дополняемой рекурсии с одним тестом завершения, которая вычисляет сумму всех чисел от 0 до n-ого аргумента функции. Вариант: 1) от п-аргумента функции до последнего >= 0, 2) от п-аргумента функции до т-аргумента с шагом d.

Если список пуст или n - неположительное число, то слагаемые отсутствуют и сумма 0. Иначе возвращаем сумму текущего элемента и результата рекурсивного вызова для n-1 элемента оставшегося списка.

(defun sum\_0\_n (lst n)

(cond

((or (not lst) (<= n 0)) 0)

(t (+ (car lst) (sum\_0\_n (cdr lst) (- n 1))))

)

)

Если список пуст, результат – 0. Если очередной элемент отрицателен и n = 0, то мы достигли первого отрицательного элемента после n, результат тоже 0. Иначе, если n = 0, текущий элемент положителен и лежит после n-го, искомая сумма равна сумме этого элемента с результатом рекурсивного вызова от остатка списка. если n > 0, заданный диапазон еще не достигнут, пропускаем элемент и возвращаем результат рекурсивного вызова от остатка списка и уменьшенного на 1 n.

(defun sum\_n\_pos (lst n)

(cond

((or (not lst) (and (= n 0) (< (car lst) 0))) 0)

((= n 0) (+ (car lst) (sum\_n\_pos (cdr lst) n)))

((> n 0) (sum\_n\_pos (cdr lst) (- n 1)))

(t 'error)

)

)

Если список пуст, его сумма – 0. Если b < 0, текущий элемент лежит за границей диапазона, текущий элемент и все последующие не учитываются, результат – 0. Иначе, если a = 0, текущий элемент нужно добавить к общей сумме, поэтому возвращается его сумма с результатом рекурсивного вызова при смещенном на шаг списке. Для этого правая граница, уменьшается на шаг, нужное количество элементов пропускается с помощью функции nthcdr.

Если a > 0, нужно пропустить первые a элементов, вычтя из b a и пропустив a элементов с помощью nthcdr.

Если a < 0, возвращаем ошибку.

(defun sum\_a\_b\_step (lst a b step)

(cond

((or (not lst) (< b 0)) 0)

((= a 0) (+ (car lst) (sum\_a\_b\_step (nthcdr step lst) 0 (- b step) step)))

((> a 0) (sum\_a\_b\_step (nthcdr a lst) 0 (- b a) step))

((< a 0) 'error)

)

)

1. Написать рекурсивную функцию, которая возвращает последнее нечетное число из числового списка, возможно создавая некоторые вспомогательные функции

Последнее значение запоминаем в num. Изначально оно равно nil. Если список пуст, после num не было встречено нечетных элементов, поэтому num является результатом. Если очередной элемент нечетный, возвращаем результат рекурсивного вызова для остатка списка с num равным текущему элементу. Иначе возвращаем результат рекурсивного вызова для остатка списка с тем же num, так как число четное и оно не меняет num.

(defun last\_odd (lst)

(defun inner\_last\_odd (lst num)

(cond

((not lst) num)

((oddp (car lst)) (inner\_last\_odd (cdr lst) (car lst)))

(t (inner\_last\_odd (cdr lst) num))

)

)

(inner\_last\_odd lst nil)

)

1. Используя cons-дополняемую рекурсию с одним тестом завершения, написать функцию которая получает как аргумент список чисел, а возвращает список квадратов этих чисел в том же порядке.

Если список пуст, возвращаем nil, иначе возвращаем список из квадрата текущего элемента и результата рекурсивного вызова для остатка списка.

(defun square\_all(lst)

(cond

((not lst) nil)

(t (cons (\* (car lst) (car lst)) (square\_all (cdr lst))))

)

)

1. Написать функцию с именем select-odd, которая из заданного списка выбирает все нечетные числа. (Вариант 1: select-even, вариант 2: вычисляет сумму всех нечетных чисел(sum-all-odd) или сумму всех четных чисел (sum-all-even) из заданного списка. )

Если список пуст, возвращаем накопленный в res список. иначе, если текущий элемент нечетный, возвращаем результат рекурсивного вызова для остатка списка и res с добавленным текущим элементом, иначе пропускаем текущий элемент и возвращаем результат рекурсивного вызова для остатка списка с тем же res. Выбор четных элементов работает аналогично. При вычислении суммы, вместо добавления элемента в res, в res записывается сумма этого элемента и res, в остальном все аналогично.

(defun select-odd (lst)

(defun inner\_select-odd (lst res)

(cond

((not lst) res)

((oddp (car lst)) (inner\_select-odd (cdr lst) (append res (cons (car lst) nil))))

(t (inner\_select-odd (cdr lst) res))

)

)

(inner\_select-odd lst nil)

)

(defun select-even (lst)

(defun inner\_select-even (lst res)

(cond

((not lst) res)

((evenp (car lst)) (inner\_select-even (cdr lst) (append res (cons (car lst) nil))))

(t (inner\_select-even (cdr lst) res))

)

)

(inner\_select-even lst nil)

)

(defun sum-all-odd (lst)

(defun inner\_sum-odd (lst res)

(cond

((not lst) res)

((oddp (car lst)) (inner\_sum-odd (cdr lst) (+ res (car lst))))

(t (inner\_sum-odd (cdr lst) res))

)

)

(inner\_sum-odd lst 0)

)

(defun sum-all-even (lst)

(defun inner\_sum-even (lst res)

(cond

((not lst) res)

((evenp (car lst)) (inner\_sum-even (cdr lst) (+ res (car lst))))

(t (inner\_sum-even (cdr lst) res))

)

)

(inner\_sum-even lst 0)

)

**Ответы на вопросы**

1. 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.

1. Что такое рекурсия? Классификация рекурсивных функций в Lisp,

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

1. Различные способы организации рекурсивных функций и порядок их реализации

При организации рекурсии можно использовать как функции с именем, так и локально определенные с помощью лямбда выражений функции (см лекции). Кроме этого, при использовании рекурсии для решения сложных программ можно использовать функционалы внутри рекурсивной функции или использовать рекурсивную функцию внутри функционала.

1. 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.