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

*По предмету: «Функциональное и логическое программирование»*

Преподаватель: Толпинская Н.Б.,

Студент: Мирзоян С.А.,

Группа: ИУ7-65Б

Москва, 2020 г.

В языке Lisp возможна организация повторных вычислений. Ее можно организовать с помощью циклов (do) и рекурсий

Макрос do – основной итерационный оператор в Common Lisp.

(do ((variable1 value1 updated-value1)

(variable2 value2 updated-value2)

(variable3 value3 updated-value3)

...)

(test return-value)

(s-expressions)

)

Пример использования:

(do ((x 0 (+ 2 x))

(y 20 (- y 2))

((= x y) (- x y))

(format t “~% x = ~d y - ~d” x y)

Результат:

x = 0 y = 20

x = 2 y = 18

x = 4 y = 16

x = 6 y = 14

x = 8 y = 12

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

Классификация рекурсивных функций:

1. простая рекурсия - один рекурсивный вызов в теле
2. рекурсия первого порядка - рекурсивный вызов встречается несколько раз
3. взаимная рекурсия - используется несколько функций, рекурсивно вызывающих друг друга.

Типы рекурсивных функций:

* 1. хвостовая,
  2. дополняемая,
  3. множественная,
  4. взаимная рекурсия,
  5. рекурсия более высокого порядка.

При организации рекурсии можно использовать:

* + 1. Функции с именем
    2. Локально определенные с помощью лямбда выражений функции
    3. Функционалы
    4. Рекурсивную функцию внутри функционала

Важным моментом в организации рекурсии является выход из нее. Из рекурсии всегда должен быть выход. Некачественный выход из рекурсии может привести к переполнению памяти из-за "лишних" рекурсивных вызовов. Кроме того возможна потеря аргумента - кажется что функция возвращает результат и он используется, но на деле результат теряется и ответ неверен.

Способы повышения эффективности:

* + - 1. **Хвостовая рекурсия.** В целях повышения эффективности рекурсивных функций рекомендуется формировать результат не на выходе из рекурсии, а на входе в рекурсию, все действия выполняя до ухода на следующий шаг рекурсии. Это и есть хвостовая рекурсия.
      2. **Дополняемая рекурсия** — при обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова , а вне его  
         В качестве дополнительной функции в дополняемой рекурсии очень часто используется cons , которая формирует результат.
      3. **Выделяют группу функций множественной рекурсии**. На одной ветке происходит сразу несколько рекурсивных вызовов. Количество условий выхода также может зависеть от задачи.

Для превращения не хвостовой рекурсии в хвостовую и в целях формирования результата (результирующего списка) на входе в рекурсию, рекомендуется использовать дополнительные (рабочие) параметры. При этом становится необходимым создать функцию — оболочку для реализации очевидного обращения к функции:

(defun into\_one\_level (lst)

( into\_one lst () )

)

(defun into\_one (lst rst)

(cond ( (null lst) rst )

( (atom lst) (cons lst rst) )

( t (into\_one (car lst)

(into\_one (cdr lst) rst)

)

)

)

)

**Задачи**

**7. 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.**

(defun sum\_lengths (list-of-lists)

(reduce #'+

(mapcar (lambda (x)

(if (listp x) (sum\_lengths x) 1)

)

list-of-lists

)

)

)

**8. Написать рекурсивную версию (с именем rec-add) вычисления суммы чисел заданного списка. Например: (rec-add (2 4 6)) -> 12**

(defun rec\_add\_inner (lst sum)

(let ( (head (car lst))

(tail (cdr lst))

)

(cond ((null lst) sum)

((listp head) (rec\_add\_inner tail (rec\_add\_inner head sum)) )

((numberp head) (rec\_add\_inner tail (+ sum head)) )

(t (rec\_add\_inner tail sum))

)

)

)

(defun rec\_add (lst)

(if (eq lst nil)

nil

(rec\_add\_inner lst 0)

)

)

**9. Написать рекурсивную версию с именем rec-nth функции nth.**

(defun rec\_nth\_inner (elem curr target)

(cond ((= curr target) (car elem))

((eq elem nil) nil)

(t (rec\_nth\_inner (cdr elem) (+ curr 1) target))

)

)

(defun rec\_nth (num lst)

(rec\_nth\_inner lst 0 num)

)

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

(defun alloddr (lst);

(let ((head (car lst))

(tail (cdr lst))

)

(cond ((null lst) t)

((listp head)

(and (alloddr head) (alloddr tail))

)

((not (numberp head)) nil)

((evenp head) nil)

(t (alloddr tail))

)

)

)

**11. Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка-аргумента.**

(defun mylast (curr)

(if (eq (cdr curr) nil)

(car curr)

(mylast (cdr curr))

)

)

**12. Написать рекурсивную функцию, относящуюся к дополняемой рекурсии с одним тестом завершения, которая вычисляет сумму всех чисел от 0 до n-аргумента функции.**

**Вариант: 1) от n-аргумента функции до последнего >= 0,**

**2) от n-аргумента функции до m-аргумента c шагом d.**

(defun get\_n\_sum (curr n)

(if (or (eq curr nil) (= n 0))

0

(+ (car curr) (get\_n\_sum (cdr curr) (- n 1)))

)

)

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

(defun get\_last\_odd\_inner (curr value)

(cond ((eq curr nil) value)

((oddp (car curr)) (get\_last\_odd\_inner (cdr curr) (car curr)))

(t (get\_last\_odd\_inner (cdr curr) value))

)

)

(defun get\_last\_odd (lst)

(get\_last\_odd\_inner lst nil)

)

**14. 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.**

(defun square\_all (lst);

(mapcar #'(lambda (x)

(cond ((numberp x) (\* x x))

((listp x) (square\_all x))

(t x)

)

)

lst

)

)

**15. Написать функцию с именем select-odd, которая из заданного списка выбирает все нечетные числа.**

**Вариант: select-even;**

**Вариант: вычисляет сумму только всех нечетных чисел (sum-all-odd) или сумму всех четных чисел (sum-all-odd) из заданного списка.)**

(defun select\_odd\_inner (lst result)

(mapcar #'(lambda (x)

(cond ((listp x) (select\_odd\_inner x result))

((and (numberp x) (oddp x))

(nconc result (cons x nil))

)

)

)

lst

)

(cdr result)

)

(defun select\_odd (lst);

(select\_odd\_inner lst (cons nil nil))

)