|  |  |
| --- | --- |
| 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**\_\_**

|  |  |
| --- | --- |
| **Дисциплина** Функциональное и логическое программирование  **Студент** Зейналов З. Г.  **Группа** ИУ7-61Б  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель** Толпинская Н. Б. |  |

Москва.

2020 г.

1. **Ход работы**

1. **Пусть list-of-list список, состоящий из списков. Написать функцию, которая вычисляет сумму длин всех элементов list-of-list.**

(defun sum\_lengths (list-of-lists)

 (reduce #'+

   (mapcar (lambda (x)

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

     list-of-lists

   )

 )

)

В #reduce по умолчанию на операцию + initial\_value равно 0, поэтому не возникает необходимости использовать дополнительные переменные. С помощью mapcar проходимся по элементам списка и анализируем их, если это список, то рекурсивно проходимся по нему, если число – возвращаем 1.

Пример работы:

\* (sum\_lengths '((1 23 1 1)))

4

\* (sum\_lengths '((1 23 (1 2) 1)))

5

\* (sum\_lengths '((1 23 (1 2 ) 1)))

5

\* (sum\_lengths '((1 23 (1 2 5) 1)))

6

\* (sum\_lengths '((1 23 (1 (1 2 3) 5) 1)))

8

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

(defun reg-add-recur (lst sum)

  (cond

  ((null lst) sum)

  ((listp (car lst)) (reg-add-recur (car lst) (reg-add-recur (cdr lst) sum)))

  (t (reg-add-recur (cdr lst) (+ sum (car lst))))

  )

)

Рекурсивная функция принимает список lst и текущее значение суммы sum. Если список пуст, то возвращается значение суммы. Если элементом списка является список, то вызывается рекурсивно функция для него и для хвоста списка. Рекурсивный вызов функции выполняется с изменённым текущим значением суммы.

(defun reg-add (lst)

  (reg-add-recur lst 0)

)

Вызов рекурсивной функции со списком lst и начальным значением суммы равным 0.

Пример работы:

\* (reg-add '(1 5 (3 1) () 9))

8

\* (reg-add '(1 3 (5 2) ((1 2) (1 2)) 1))

14

\* (reg-add '())

0

1. **Написать рекурсивную версию с именем recnth функции nth.**

(defun recnth\_rec (n lst)

  (if (eq n 0) (car lst) (recnth\_rec (- n 1) (cdr lst))

  )

)

Рекурсивная функция принимает номер элемента n из списка lst. Возвращает элемент с этим номером (нумерация с нуля). Если n равен нулю, то возвращаем элемент из головы списка, иначе уменьшаем n на единицу и рекурсивно вызываем функцию для хвоста списка.

Пример работы:

\* (recnth\_rec 10 '(1 2 3 4))

NIL

\* (recnth\_rec 2 '(1 2 3 4))

3

\* (recnth\_rec 4 '(1 2 3 4))

NIL

\* (recnth\_rec 0 '(1 2 3 4))

1

4. **Написать рекурсивную функцию alloddr, которая возвращает t когда все**

**элементы списка нечетные.**

(defun alloddr\_rec (lst state)

  (cond

    ((null lst) state)

    ((evenp (car lst)) Nil)

    (t (alloddr\_rec (cdr lst) (setq state T)))

  )

)

Рекурсивно проверяется элемент головы списка на чётность, если он чётный, то возвращается nil, иначе (элемент нечётный) вызывается рекурсивная функция для хвоста списка со значением state равным T (означает, что пока все встреченные элементы нечётны).

(defun alloddr (lst)

  (alloddr\_rec lst Nil)

)

Вызов рекурсивной функции alloddr\_rec со списком lst и начальным результатом Nil.

Пример работы:

\* (alloddr '())

NIL

\* (alloddr '(4 4))

NIL

\* (alloddr '(2 1))

NIL

\* (alloddr '(1 1))

T

5. **Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка – аргументы.**

(defun last\_elem (lst)

  (cond

  ((null (cdr lst)) (car lst))

  (t (last\_elem (cdr lst)))

  )

)

Функция принимает список, если хвост списка является Nil, значит текущий элемент последний, и он возвращается. В ином случае рекурсивно вызывается данная функция для хвоста списка.

Пример работы:

\* (last\_elem '(1 2 3))

3

\* (last\_elem '())

NIL

\* (last\_elem '(1 2 (3)))

(3)

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

(defun sum\_all\_rec (lst n sum)

  (cond

  ((or (null lst) (eq n 0)) sum)

  (t (sum\_all\_rec (cdr lst) (- n 1) (+ sum (car lst))))

  )

)

Пока список не пуст или n не равно 0, рекурсивно вызывается функция со значением n - 1 , изменённым значением суммы для хвоста списка.

(defun sum\_all (lst n)

  (sum\_all\_rec lst n 0)

)

Вызов рекурсивной функции со списком lst, номером элемента n и начальным значением суммы 0.

Пример работы:

\* (sum\_all '(1 2 3 4 5) 0)

0

\* (sum\_all '(1 2 3 4 5) 2)

3

\* (sum\_all '(1 2 3 4 5) 10)

15

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

(defun lastOdd (lst value)

    (cond

        ((null lst) value)

        ((oddp (car lst)) (lastOdd (cdr lst) (car lst)))

        (t (lastOdd (cdr lst) value))

    )

)

Пока список не станет пустым, производится проверка текущего элемента на нечётность, если он нечётный, то он записывается в value. Изначально значения value nil. Иначе рекурсивно вызывается данная функция для хвоста списка.

(defun modd (lst)

    (lastOdd lst nil)

)

Вызов рекурсивной функции с обращённым списком.

Пример работы:

\* (mOdd '(1 2 3 4 5))

5

\* (mOdd '(1 2 3 4 5 7))

7

\* (mOdd '(1 2 3 4 5 7 8 8 58))

7

\* (mOdd '(1))

1

\* (mOdd '())

NIL

\*

8. **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)

    (cond

    ((null lst) nil)

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

    )

)

Пока список lst не станет пустым, добавляем в список квадрат текущего элемента и рекурсивно вызываем функцию для хвоста списка.

\* (square\_all '(1 2 3))

(1 4 9)

\* (square\_all '(-2 3 3))

(4 9 9)

\* (square\_all '())

NIL

\* (square\_all '(1))

(1)

9. **Написать функцию с именем select-odd, которая из заданного**

**списка выбирает все нечетные числа.**

(defun select-odd-recur (lst)

  (cond

  ((null lst) nil)

  (t (cons (if (oddp (car lst)) (car lst) Nil) (select-odd-recur (cdr lst))))

  )

)

Пока список не станет пустым, проверяется каждый элемент списка. Если он нечётный, то добавляется в новый список, иначе в новый список добавляется Nil.

(defun select-odd (lst)

  (remove nil (select\_all\_odd lst))

)

Вызывается рекурсивная функция, затем из списка, полученного в результате убираются элементы Nil.

Пример работы:

\* (select-odd '(1 2 3))

(1 3)

\* (select-odd '(3))

(3)

\* (select-odd '())

NIL

\* (select-odd '(222 2 2))

NIL

1. **Контрольные вопросы**
2. Способы организации повторных вычислений в Lisp.

Для организации повторных вычислений в Lisp могут быть использованы функционалы – функции высокого порядка, которые получают в качестве аргумента функциональный объект. Также для организации повторных вычислений может быть использована рекурсия. Рекурсия – это ссылка на определяемый объект во время его определения.

1. Различные способы использования функционалов.

Функционалы используются в Lisp для организации повторных вычислений. В Lisp используются применяющие и отображающие функционалы, функционалы, являющиеся предикатами, функционалы, использующие предикаты в качестве функционального объекта.

1. Что такое рекурсия? Способы организации рекурсивных функций.

Рекурсия – это ссылка на определяемый объект во время его определения. Виды рекурсии: хвостовая, дополняемая, множественная, взаимная и рекурсии высокого порядка.

1. Способы повышения эффективности реализации рекурсии.

Один из методов повышения эффективности рекурсии является организация хвостовой рекурсии. Для этого может потребоваться использовать дополнительные параметры. Такая рекурсия может быть путём формальной и гарантированно корректной перестройки кода заменена на итерацию. Такая оптимизация реализована во многих оптимизирующих компиляторах, а в трансляторах Scheme, одного из диалектов Lisp, такая оптимизация является обязательной.