**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** Рекурсивные функции

**Студент** Сушина А.Д.

**Группа** ИУ7-61б

**Оценка (баллы)** _____

**Преподаватель** Толпинская Н.Б.

Москва.
2020 г

**Цель работы:** приобрести навыки организации рекурсии в Lisp

**Задачи работы:** изучить способы организации хвостовой, дополняемой, множественной, взаимной рекурсии и рекурсии более высокого порядка в Lisp.

## Ход работы

**Практическое задание:№7, №8, №9, №10, №11, №12, №13, №14, №15 из лабораторной 6.**

**Задания:**

**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_of_len_help(lst res)
   ( cond ((null lst) res)
        ((listp (car lst)) (sum_of_len_help (cdr lst) (+ res (length (car lst)))))
        (t (sum_of_len_help (cdr lst) (+ res 1)))
   )
)
(defun sum_of_len(lst)
   (sum_of_len_help lst 0)
)
```

Функция sum_of_len_help принимает два аргумента: список и доп аргумент для результата. Подсчитывает сумму длин всех элементов списка. Если список пустой, возвращает результат. Если список голова списка — список, то прибавляет ее длину к результату и выполняет рекурсивный вызов функции для хвоста списка. Иначе добавляет единицу к результату(встречен атом, его длина — единица) и выполняет рекурсивный вызов функции для хвоста списка.

Функция sum_of_len принимает на вход список. Подсчитывает сумму длин всех элементов списка. Выполняет вызов функции sum_of_len_help с начальным параметром для результата — 0.

Второй вариант реализации. Подсчитывает количество атомов на всех уровнях.

```
(defun sum_of_len_help(lst res)
   ( cond ((null lst) res)
        ((listp (car lst)) (sum_of_len_help (cdr lst) (sum_of_len_help (car lst) res)))
        (t (sum_of_len_help (cdr lst) (+ res 1)))
   )
)
```

```
(defun sum_of_len(lst)
   (sum_of_len_help lst 0)
)
```

> (sum_of_len `((1 2) (3 4 (5 6)) ) ) → 6

Функция sum_of_len_help принимает два аргумента: список и доп аргумент для результата. Если список пустой, возвращает результат. Если список голова списка — список, то выполняет рекурсивный вызов функции с агрументами: хвост функции и результат выполнения рекурсивного вызова функции для головы списка и текущего результата. Иначе добавляет единицу к результату(встречен атом, его длина — единица) и выполняет рекурсивный вызов функции для хвоста списка.

Функция sum_of_len принимает на вход список. Подсчитывает сумму длин всех элементов списка. Выполняет вызов функции sum_of_len_help с начальным параметром для результата — 0.

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

**Например: (reg-add (2 4 6)) → 12**

```
(defun reg-add-help(lst res)
   ( cond ((null lst) res)
         ((numberp (car lst)) (reg-add-help (cdr lst) (+ res (car lst))))
         ((listp (car lst)) (reg-add-help (cdr lst)  (reg-add-help (car lst) res)))
         (t (reg-add-help (cdr lst) res))
   )
)
(defun reg-add(lst) ( reg-add-help lst 0))
```

> (reg-add (2 4 6 (5 5))) → 22

Функция reg-add-help принимает на вход два аргумента — список и доп параметр для результата. Если список пустой, то возвращает результат. Если голова списка — число, выполняет рекурсивный вызов функции reg-add-help для параметров хвост списка и сумма текущего результата и головы списка. Если голова списка — список, то выполняет рекурсивный вызов функции для головы списка, а затем выполняет рекурсивный вызов функции для хвоста списка и результата предыдущего вызова. Иначе выполняет рекурсивный вызов функции для хвоста списка с тем же результатом.

Функция reg-add принимает на вход список. Возвращает сумму всех элементов. Выполняет вызов функции с начальным значением результата 0.

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

```
(defun recnth(num lst)
   ( cond ((null lst) nil)
         ((equal num 0) (car lst))
         (t (recnth (- num 1) (cdr lst)))
   )
)
```

> (recnth 0 `(1 2 3 4)) → 1

> (recnth 3 `(1 2 3 4)) → 4

> (recnth 2 `(1 2 (3) 4)) →(3)

Функция recnth принимает на вход два аргумента: номер и список. Если список пустой, возвращает nil. Если номер равен 0, то возвращает текущую голову списка. Иначе выполняет рекурсивный вызов функции с двумя аргументами: уменьшенным на единицу значением номера и хвостом списка.

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

```
(defun alloddr_help(lst rst)
   ( cond ((null rst) nil)
         ((null lst)rst)
         ((listp (car lst)) (alloddr_help (cdr lst) (alloddr_help (car lst) rst)))
         ((not(and (numberp (car lst)) (oddp (car lst)))) nil)
         (t (alloddr_help (cdr lst) rst))
   )
)
(defun alloddr(lst) (alloddr_help lst t))
```

> (alloddr `(1 3 5)) → T

> (alloddr `(1 3 ( 5 7) 5)) → T

> (alloddr `(1 3 4 5)) → Nil

> (alloddr `(1 3 (4 5) 5)) → Nil

> (alloddr `(1 3 a b c 5)) → Nil

Функция alloddr_help принимает на вход два аргумента — список и доп параметр для результата. Если результат пустой, возвращает nil. Это обеспечивает выход из рекурсии, если найдено четное число в каком-либо подсписке. Если список пустой, то возвращает результат (все элементы нечетные, не было выхода с nil). Если голова списка список выполняется

рекурсивный вызов функции alloddr_help для двух аргументов: хвоста списка и результата рекурсивного вызова функции alloddr_help для головы списка и текущего результата. Если голова не число или четное число, возвращается nil. Иначе выполняется рекурсивный вызов функции для хвоста списка и текущего результата.

Функция alloddr принимает на вход список. Возвращает t, когда все элементы списка нечетные. Выполняет вызов функции с начальным значением результата t.

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

```
(defun my_last(lst)
   ( cond ((null (cdr lst)) (car lst))
         (t (my_last (cdr lst)))
   )
)
```

> (my_last `(1 2 3 4)) → 4

Функция my_last принимает на вход список. Возвращает последний элемент списка. Если текущей элемент последний, возвращает его. Иначе выполняет рекурсивный вызов самой себя для хвоста списка.

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

```
(defun get_n_sum(lst n)
   ( cond
         ((or (equal n -1) (null lst) ) 0)
         (t (+ (car lst) (get_n_sum (cdr lst) (- n 1) )))
   )
)
```

>(get_n_sum `(1 2 1 3 1 4) 3)→ (**1 2 1 3** 1 4) → 7

Функция get_n_sum принимает на вход два аргумента: список и номер. Вычисляет сумму всех элементов до н-ого. Если номер равен -1 или список закончился, вовзвращает 0. Иначе прибавляет текущий элемент к результату рекурсивного вызова функции для хвоста списка и номера, уменьшенного на 1.

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

```
(defun from_n_to_end(lst n)
   ( cond  ((or (null lst) (and (equal n 0) (< (car lst) 0)) ) 0)
```

```
        ((equal n 0) (+ (car lst) (from_n_to_end (cdr lst) n)) )
        (t (from_n_to_end (cdr lst) (- n 1)))
    )
)
```

>(from_n_to_end `(1 2 1 3 1 4)  3) → (1 2 1 **3 1 4**)→ 8

>(from_n_to_end `(1 2 1 3 -1 4)  3) → (1 2 1 **3** -1 4) → 3

Функция get_n_to_end принимает на вход два аргумента: список и номер. Вычисляет сумму всех элементов от н-ого до последнего больше нуля. Если список пустой или встречен отрицательный элемент после н-ого, возвращает 0. Если номер равен нулю, то прибавляет голову списка к результату рекурсивного вызова функции для хвоста списка и номера. Иначе выполняет рекурсивный вызов функции для хвоста списка и номера, уменьшенного на единицу.

**2) от n-аргумента функции до т-аргумента с шагом d.**

```
(defun from_n_to_t(lst n m step)
   ( cond ((or (null lst) (< m 0)) 0)
        ((equal n 0) (+ (car lst) (from_n_to_t (nthcdr step lst) n (- m step) step)) )
        (t (from_n_to_t (cdr lst) (- n 1) (- m 1) step))
   )
)
```

> (from_n_to_t `(1 2 1 3 1 4)  2 5 2) → (1 2 **1** 3 **1** 4) → 1+1 = 2

Функция get_n_to_t принимает на вход 4 аргумента: список, начало и конец интервала и шаг. Вычисляет сумму всех элементов от н-ого до конца. Если список пустой или достигнут конец интервала, возвращает 0. Если номер равен нулю, то прибавляет голову списка к результату рекурсивного вызова функции для аргументов: списочной ячейки находящейся через шаг от текущей, текущего начального номера(0), уменьшенного на шаг конца интервала и шага. Иначе выполняет рекурсивный вызов функции для хвоста списка, начала и конца интервала, уменьшенных на единицу и шага.

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

```
(defun lastodd_help(lst res)
   ( cond ((null lst) res)
        ((oddp (car lst)) (lastodd_help (cdr lst) (car lst)))
        (t (lastodd_help (cdr lst) res))
   )
```

)

(defun lastodd(lst) (lastodd_help lst nil))

> (lastodd `( 1 2 3 4 5)) → 5

Функция lastodd_help принимает на вход два аргумента — список и доп параметр для результата. Если список пустой, то возвращает результат. Если голова списка — нечетный элемент, выполняет рекурсивный вызов для хвоста списка, в качестве результата голову списка. Иначе выполняет рекурсивный вызов функции для хвоста списка и текущего результата.

Функция lastodd принимает на вход список. Возвращает последний нечетный элемент. Выполняет вызов функции lastodd_help с начальным значением результата -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 squares_all(lst)
   ( cond ((null lst) nil)
         (t (cons (* (car lst) (car lst)) (squares_all (cdr lst))))
   )
)
```

> ( squares_all `( 1 2 3 4)) → (1 4 9 16)

Функция squares_all принимает на вход список. Если список пустой, возвращает nil. Иначе выполняет объединение головы списка в квадрате с рекурсивным вызовом самой себя для хвоста списка.

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

```
(defun select-odd-h(lst res)
   ( cond ((null lst)  (cdr res))
         ((and (numberp (car lst))
               (oddp (car lst)))
          (select-odd-h (cdr lst) (append res (list (car lst))))
         )
         ((listp (car lst))
          (select-odd-h (cdr lst)
                        (append res (select-odd-h (car lst) '(nil)))
          )
```

```
        )
        (t (select-odd-h (cdr lst) res))
    )
)
(defun select-odd(lst) (select-odd-h lst '(nil)))
```

> (select-odd `( 1 2 (3 4) 5 ((6 (7)) 8 9))) →( **1** 2 (**3** 4) **5** ((6 (**7**)) 8 **9**)) → (1 3 5 7 9)

Функция select-odd-h принимает на вход два аргумента — список и доп параметр для результата. Если список пустой, то возвращает хвост результата(т.к. первый элемент nil). Если голова списка нечетное число, то выполняет рекурсивный вызов самой себя для хвоста списка, где в качестве результата передается объединение текущего результат и головы списка(в конец результата добавляется голова текущего списка). Если голова списка тоже список, то выполняет рекурсивный вызов самой себя для хвоста списка, где в качестве результата передается объединение текущего результата и результата рекурсивного вызова функции для головы списка с начальным значением результата (nil). Иначе выполняет рекурсивный вызов самой себя для хвоста списка, в качестве результата передается текущий результат.

Функция select-odd принимает на вход список. Возвращает список из всех нечетных элементов. Выполняет вызов select-odd-h с начальным значением результата (nil)

**Вариант 1: select-even,**

```
(defun select-even-h(lst res)
    ( cond ((null lst)  (cdr res))
        ((and (numberp (car lst))
            (evenp (car lst)))
         (select-even-h (cdr lst) (append res (list (car lst))))
        )
        ((listp (car lst))
         (select-even-h (cdr lst)
                (append res (select-even-h (car lst) '(nil)))
         )
        )
        (t (select-even-h (cdr lst) res))
    )
)
(defun select-even(lst) (select-even-h lst '(nil)))
```

> (select-even `( 1 2 (3 4) 5 ((6 (7)) 8 9))) → ( 1 **2** (3 **4**) 5 ((**6** (7)) **8** 9)) → (2 4 6 8)

Функции select-even и select-even-h работают точно также, как и две предыдущие, только для четных чисел.

**вариант 2: вычисляет сумму всех нечетных чисел(sum-all-odd) или сумму всех четных чисел (sum-all-even) из заданного списка.**

```
(defun sum-all-even-h(lst res)
   ( cond ((null lst)  res)
         ((and (numberp (car lst))
              (evenp (car lst)))
          (sum-all-even-h (cdr lst) (+ res (car lst)))
         )
         ((listp (car lst))
          (sum-all-even-h (cdr lst)
                    (sum-all-even-h (car lst) res)
          )
         )
         (t (sum-all-even-h (cdr lst) res))
   )
)
(defun sum-all-even(lst) (sum-all-even-h lst 0))
```

> (sum-all-even `( 1 2 (3 4) 5 ((6 (7)) 8 9))) → ( 1 2 (3 4) 5 ((6 (7)) 8 9)) → (2 4 6 8) → 20

Функции sum-all-even и sum-all-even-h работают точно также, как и две предыдущие, однако вместо создания списка с четными элементами, производится сложение. Начальное значение результата — 0. Если найдено четное число, выполняется рекурсивный вызов функции для хвоста, где в качестве результата передается сумма результата и головы списка.

**Теоретические вопросы:**

- **Способы организации повторных вычислений в Lisp,**

  Для организации многократных вычислений в Lisp могут быть использованы функционалы — функции, которые особым образом обрабатывают свои аргументы. Также для организации многократных вычислений в Lisp может быть использована рекурсия. Рекурсия — это ссылка на определяемый объект во время его определения.

- **Что такое рекурсия? Классификация рекурсивных функций в Lisp,**

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

  В LISP существует классификация рекурсивных функций:
  - простая рекурсия - один рекурсивный вызов в теле
  - рекурсия первого порядка - рекурсивный вызов встречается несколько раз

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

Виды рекурсии:

- Хвостовая. Результат формируется  не на выходе из рекурсии, а на входе в рекурсию, все действия выполняются до ухода на следующий шаг рекурсии.
- Дополняемая. При обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова , а вне его.
- Множественная. На одной ветке происходит сразу несколько рекурсивных вызовов. Количество условий выхода также может зависеть от задачи.
- Взаимная.
- рекурсии высокого порядка.

- **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. Кроме этого, при организации рекурсии можно использовать функционалы или использовать рекурсивную функцию внутри функционала. При изучении рекурсии рекомендуется организовывать и отлаживать реализацию отдельных подзадач исходной задачи, обращая внимание на эффективность реализации и качество работы, а потом, при необходимости, встраивать эти функции в более крупные, возможно в виде  ламбда-выражений.

- **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.