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 №5 по курсу "Функциональное и логическое программирование"

Тема Использование управляющих структур, работа со списками

Студент Ковалец К. Э.

Группа ИУ7-63Б

Преподаватели Толпинская Н. Б., Строганов Ю. В.

Москва — 2022 г.

1 Практические задания

1.1 Мои функции

Листинг 1.1 – my-reverse

```
(defun my-reverse-rec (lst rev-lst)
    (cond
        ((null lst)
            rev-lst)
        (T
            (my-reverse-rec (cdr lst) (cons (car lst) rev-lst)))
    )
)

(defun my-reverse (lst)
    (my-reverse-rec lst nil))

;; (MY-REVERSE '(1 2 3 4 5)) -> (5 4 3 2 1)
```

Листинг 1.2 – my-length

```
(defun my-length-rec (set len)
    (cond
        ((null set)
            len)
        (T
            (my-length-rec (cdr set) (+ len 1)))
    )
)

(defun my-length (set)
    (my-length-rec set 0))

;; (MY-LENGTH '(1 2 3 4)) -> 4
```

Листинг 1.3 – my-last

```
(defun my-last (lst)
    (cond
        ((null (cdr lst))
            (list (car lst)))
        (T
            (my-last (cdr lst)))
    )
)

;; (MY-LAST '(1 2 4 5)) -> (5)
```

Листинг 1.4 – my-append

```
(defun my-append (lst1 lst2)
    (cond
        ((null lst1)
            lst2)
        (T
            (cons (car lst1) (my-append (cdr lst1) lst2)))
    )
)

;; (MY-APPEND '(1 2 3) '(4 5 6)) -> (1 2 3 4 5 6)
```

Листинг 1.5 – my-sort

```
(defun find-min-elem-rec (lst cur_min)
    (cond
        ((null lst)
            cur_min)
        (T
            (find-min-elem-rec (cdr lst)
                (cond
                    ((< (car lst) cur_min)
                        (setf cur_min (car lst)))
                    (T
                        cur_min))))
    )
)

(defun find-min-elem (lst)
    (find-min-elem-rec lst (car lst)))

(defun insert (lst elem elem_instead)
    (cond
        ((null lst)
            elem_instead)
        ((eql (car lst) elem_instead)
            (setf (car lst) elem))
        (T
            (insert (cdr lst) elem elem_instead))
    )
)

(defun my-sort-rec (res_lst lst)
    (let* ((min_elem (find-min-elem lst)))
        (cond
            ((null lst)
                res_lst)
            (T
                (insert lst (car lst) min_elem)
                (my-sort-rec
                    (my-append res_lst (list min_elem))
                    (cdr lst)))
        )
    )
)

(defun my-sort (lst)
    (my-sort-rec NIL lst))

;; (MY-SORT '(4 7 2 2 8 1 3)) -> (1 2 2 3 4 7 8)
```

1.2 Задание 1

Написать функцию, которая по своему списку-аргументу lst определяет является ли он палиндромом (то есть равны ли lst и $(reverse\ lst)$).

Листинг 1.6 – Решение задания 1

```
(defun compare (lst1 lst2)
    (cond
        ((null lst1)
            T)
        ((eql (car lst1) (car lst2))
            (compare (cdr lst1) (cdr lst2)))
    )
)

(defun palindrome (lst)
    (compare lst (my-reverse lst)))

;; (palindrome '(1 2 3 2 1)) -> T
;; (palindrome '(1 2 3 2)) -> NIL
```

1.3 Задание 2

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

Листинг 1.7 – Решение задания 2

```
(defun elem-in-set (elem set)
    (cond
        ((null set)
            NIL)
        ((eql elem (car set))
            T)
        (T
            (elem-in-set elem (cdr set)))
    )
)

(defun check-set-equal (set1 set2)
    (cond
        ((null set1)
            T)
        ((elem-in-set (car set1) set2)
            (check-set-equal (cdr set1) set2))
    )
)


(defun set-equal (set1 set2)
    (if (eql (my-length set1) (my-length set2))
        (check-set-equal set1 set2)
    )
)

;;(SET-EQUAL '(1 2 3 4 5) '(4 2 5 1 3)) -> T
;;(SET-EQUAL '(1 2 3 4 5) '(4 2 5 1 7)) -> NIL
```

1.4 Задание 3

Напишите свои необходимые функции, которые обрабатывают таблицу из 4-х точечных пар: (страна . столица), и возвращают по стране – столицу, а по столице – страну .

Листинг 1.8 – Решение задания 3

```
(defun find-capital (country table)
    (cond
        ((null table)
            Nil)
        ((eql country (caar table))
            (cdar table))
        (T
            (find-capital country (cdr table)))
    )
)

(defun find-country (capital table)
    (cond
        ((null table)
            Nil)
        ((eql capital (cdar table))
            (caar table))
        (T
            (find-country capital (cdr table)))
    )
)

;; (find-capital
;;     'Russia
;;     '((USA . Washington)
;;       (Russia . Moscow)
;;       (Germany . Berlin)))
;; -> MOSCOW

;; (find-country
;;     'Moscow
;;     '((USA . Washington)
;;       (Russia . Moscow)
;;       (Germany . Berlin)))
;; -> RUSSIA
```

1.5 Задание4

Напишите функцию `swap-first-last`, которая переставляет в списке-аргументе первый и последний элементы.

Листинг 1.9 – Решение задания 4

```
(defun swap-first-last-rec (first_elem rev_lst lst)
    (cond
        ((null (cdr lst))
            (my-append (cons (car lst) (my-reverse rev_lst)) (list
               first_elem)))
        (T
            (swap-first-last-rec first_elem (cons (car lst) rev_lst) (cdr
               lst)))
    )
)

(defun swap-first-last (lst)
    (swap-first-last-rec (car lst) NIL (cdr lst)))

;; (SWAP-FIRST-LAST '(1 2 3 4 5)) -> (5 2 3 4 1)
```

1.6 Задание 5

Напишите функцию `swap-two-ellement`, которая переставляет в списке-аргументе два указанных своими порядковыми номерами элемента в этом списке.

Листинг 1.10 – Решение задания 5

```
(defun get-elem (lst i)
    (cond
        ((null lst)
            NIL)
        ((eql i 0)
            (car lst))
        (T
            (get-elem (cdr lst) (- i 1)))
    )
)

```

```
(defun set-elem (lst elem i)
    (cond
        ((null lst)
            NIL)
        ((eql i 0)
            (setf (car lst) elem))
        (T
            (set-elem (cdr lst) elem (- i 1)))
    )
)

(defun swap-two-ellement (lst i j)
    (let* ((tmp_elem (get-elem lst i)))
        (set-elem lst (get-elem lst j) i)
        (set-elem lst tmp_elem j)
        lst
    )
)

;;(SWAP-TWO-ELLEMENT '(1 2 3 4 5) 1 4) -> (1 5 3 4 2)
```

1.7 Задание 6

Напишите две функции, `swap-to-left` и `swap-to-right`, которые производят одну круговую перестановку в списке-аргументе влево и вправо, соответственно.

Листинг 1.11 – Решение задания 6

```
(defun swap-to-left (lst)
    (my-append (cdr lst) (list (car lst))))

;; (SWAP-TO-LEFT '(1 2 3 4)) -> (2 3 4 1)

(defun swap-to-right (lst)
    (my-append (my-last lst) (my-reverse (cdr (my-reverse lst)))))

;; (SWAP-TO-RIGHT '(1 2 3 4)) -> (4 1 2 3)
```

1.8 Задание 7

Напишите функцию, которая добавляет к множеству двухэлементных списков новый двухэлементный список, если его там нет.

Листинг 1.12 – Решение задания 7

```
(defun check-set-equal (set1 set2)
    (cond
        ((null set1)
            T)
        ((eql (car set1) (car set2))
            (check-set-equal (cdr set1) (cdr set2)))
        (T
            NIL)
    )
)

(defun set-equal (set1 set2)
    (if (eql (my-length set1) (my-length set2))
        (check-set-equal set1 set2)
    )
)

(defun is-sublist (lst sublist)
    (cond
        ((null lst)
            NIL)
        ((set-equal (car lst) sublist)
            T)
        (T
            (is-sublist (cdr lst) sublist))
    )
)

(defun add-list (lst new_lst)
    (if (is-sublist lst new_lst)
        lst
        (my-append lst (list new_lst))
    )
)

;; (ADD-LIST '((1 2) (3 4) (1 1)) '(5 5)) -> ((1 2) (3 4) (1 1) (5 5))
;; (ADD-LIST '((1 2) (3 4) (1 1)) '(1 1)) -> ((1 2) (3 4) (1 1))
```

1.9 Задание 8

Напишите функцию, которая умножает на заданное число-аргумент первый числовой элемент списка из заданного 3-х элементного списка-аргумента, когда a) все элементы списка – числа, b) элементы списка – любые объекты.

Листинг 1.13 – Решение задания 8

```
;; a) все элементы списка -- числа
(defun mult-first-num-arg (lst num)
    (setf (car lst) (* (car lst) num))
    lst
)

;; (MULT-FIRST-NUM-ARG '(1 2 3 4) 5) -> (5 2 3 4)

;; b) элементы списка -- любые объекты
(defun mult-first-num-arg (lst num)
    (cond
        ((null lst)
            NIL)
        ((numberp (car lst))
            (setf (car lst) (* (car lst) num)))
        (T
            (mult-first-num-arg (cdr lst) num))
    )
    lst
)

;; (MULT-FIRST-NUM-ARG '(a (1 2) 3 4) 5) -> (A (1 2) 15 4)
;; (MULT-FIRST-NUM-ARG '(a (1 2)) 5) -> (A (1 2))
```

1.10 Задание 9

Напишите функцию, `select-between`, которая из списка-аргумента из 5 чисел выбирает только те, которые расположены между двумя указанными границами-аргументами и возвращает их в виде списка (упорядоченного по возрастанию чисел списка (+ 2 балла)).

Листинг 1.14 – Решение задания 9

```
(defun find-right-numbers (res_lst lst b1 b2)
    (let* ((cur_elem (car lst)))
        (cond
            ((null lst)
                res_lst)
            ((and (numberp cur_elem)
                    (> cur_elem b1)
                    (< cur_elem b2))
                (find-right-numbers
                    (my-append res_lst (list cur_elem))
                    (cdr lst) b1 b2))
            (T
                (find-right-numbers res_lst (cdr lst) b1 b2))
        )
    )
)

(defun select-between (lst b1 b2)
    (let* ((res_lst NIL))
        (my-sort (find-right-numbers res_lst lst b1 b2))
    )
)

;; (SELECT-BETWEEN '(1 5 4 2 3) 2 5) -> (3 4)
;; (SELECT-BETWEEN '(1 5 4 2 3) 0 6) -> (1 2 3 4 5)
;; (SELECT-BETWEEN '(1 5 4 2 3) 7 9) -> NIL
```

2 Ответы на теоретические вопросы к лабораторной работе

2.1 Структурноразрушающие и не разрушающие структуру списка функции

2.1.1 Не разрушающие структуру функции

Данные функции не меняют сам объект-аргумент, а создают копию. Если после использования функции сохраняется возможность работать с исходным списком, то значит функция не разрушает структуру.

- **append** – объединяет списки. Можно передать больше двух аргументов. Создает копию для всех аргументов, кроме последнего;
- **reverse** – возвращает копию исходного списка, элементы которого переставлены в обратном порядке. В целях эффективности работает только на верхнем уровне;
- **remove** – данная функция удаляет элемент по значению. По умолчанию используется *eql* для сравнения на равенство, но можно передать другую функцию через ключевой параметр *test*. Модифицирует список, но работает с копией, поэтому не разрушает структуру;
- **rplace** – переставляет *car*-указатель на второй элемент-аргумент (S-выражение);
- **rplacd** – переставляет *cdr*-указатель на второй элемент-аргумент (S-выражение);
- **subst** – заменяет все элементы списка, которые равны второму переданному элементу-аргументу на первый элемент аргумент. По умолчанию для сравнения используется функция *eql*.

2.1.2 Структуроразрушающие функции

Данные функции меняют сам объект-аргумент, невозможно вернуться к исходному списку. Чаще всего такие функции начинаются с префикса n. Если после использования функции теряется возможность работы с тем списком, который был изначально, то значит его структура разрушилась.

- **nconc** – работает аналогично функции *append*, только копирует не свои аргументы, а разрушает структуру;
- **nreverse** – работает аналогично функции *reverse*, только не создает копии списка;
- **nsubst** – работает аналогично функции *subst*, только не создает копии списка.

2.2 Отличие в работе функций cons, list, append, nconc и их результаты

- **cons** – имеет фиксированное количество аргументов (два). В случае, когда аргументами являются атомы создает точечную пару. В случает, когда первый аргумент атом а второй список, атом становится головой, а второй аргумент (список) становится хвостом;
- **list** – не имеет фиксированное количество аргументов. Создает список, у которого голова – это первый аргумент, хвост – все остальные аргументы. Результат всегда список;
- **append** – принимает на вход произвольное количество аргументов и для всех аргументов, кроме последнего создает копию, ссылая при этом последний элемент каждого списка-аргумента на первый элемент следующего по порядку списка-аргумента (так как модифицируются все списки-аргументы, кроме последнего, копирование для последнего не делается в целях эффективности);

- **nconc** – работает как функция *append*, но строит такой результат следующим образом: для каждого непустого аргумента-списка, nconc устанавливает в *cdr* его последней *cons*-ячейки ссылку на первую *cons*-ячейку следующего непустого аргумента-списка. После этого она возвращает первый список, который теперь является головой результата. Главное отличие от функции *append* заключается в том, что функция *nconc* разрушает структуру списка.