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»

Отчет по лабораторной работе №4
по дисциплине
"Функциональное и логическое программирование"

Тема Использование управляющих структур, работа со списками

Студент Малышев И. А.

Группа ИУ7-61Б

Оценка (баллы) _____

Преподаватель: Толпинская Н. Б.

Москва — 2022 г.

Теоретические вопросы

1. Синтаксическая форма и хранение программы в памяти

В Lisp формы представления программы и обрабатываемых ею данных одинаковы – они представлены в виде S-выражений. Программы могут обрабатывать и преобразовывать другие программы или сами себя. В памяти программа представляется в виде бинарных узлов, так как она состоит из S-выражений.

2. Трактовка элементов списка

Если отсутствует блокировка вычислений, то первый элемент списка трактуется как имя функции, а остальные элементы – как аргументы функции. На рисунке 1 представлена схема работы функции **eval**.

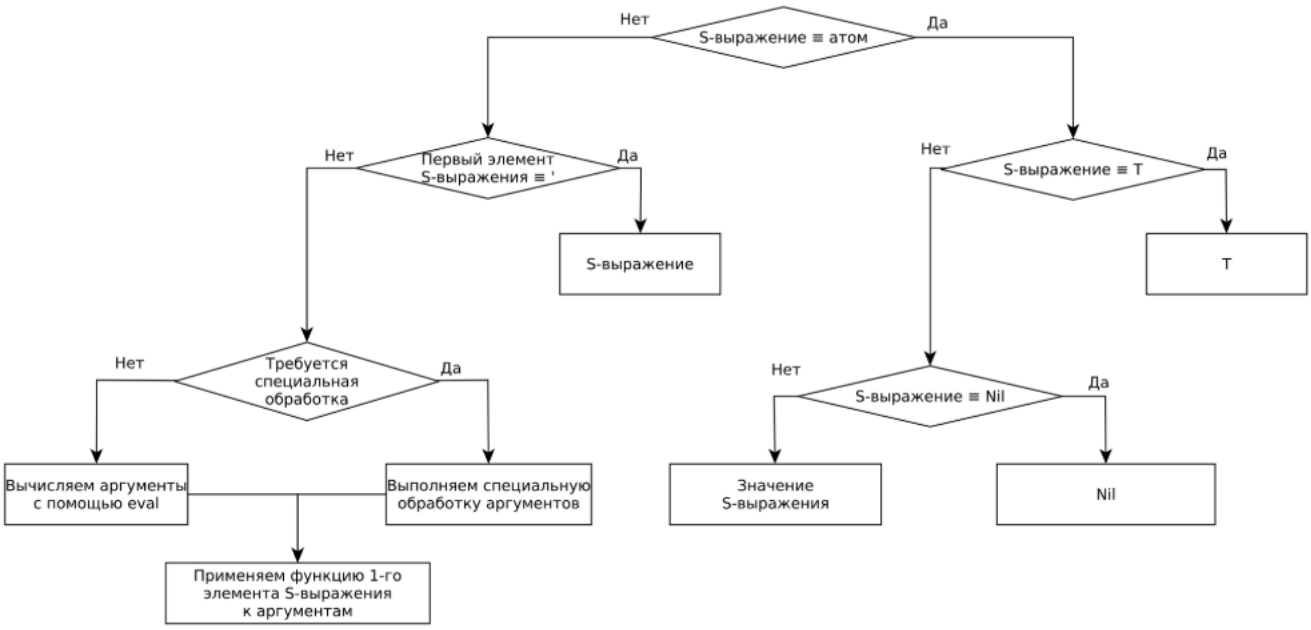

Рис. 1: Схема работы функции **eval**.

3. Порядок реализации программы

Работа программы циклична: сначала программа ожидает ввода S-выражения, затем передает полученное S-выражение интерпретатору – функции eval, а в конце, после отработки функции eval, выводит последний полученный результат.

4. Способы определения функции

Определение функций пользователя в Lisp-е возможно двумя способами:

- с использованием Лямбда-нотаци (функции без имени): (lambda (<аргументы>) (<тело>));
- с использованием макро определения DEFUN: (defun <имя> (<аргументы>) (<тело>)).

Практические задания

1. Чем принципиально отличаются функции cons, list, append?

Пусть:

```
(setf lest1 '(a b))
(setf lst2 '(c d)).
```

Каковы результаты вычисления следующих выражений?

(cons lstl lst2) -> ((A B) C D)
(list lst1 lst2) -> ((A B) (C D))
(append lst1 lst2) -> (A B C D)

2. Каковы результаты вычисления следующих выражений, и почему?

(reverse ()) -> Nil
(last ()) -> Nil
(reverse '(a)) -> (A)
(last '(a)) -> (A)
(reverse '((a b c))) -> ((A B C))
(last '((a b c))) -> ((A B C))

3. Написать, по крайней мере, два варианта функции, которая возвращает последний элемент своего списка-аргумента.

Листинг 1: Решение задания №3

```
(defun last-elem (lst)
  (if (cdr lst)
    (last-elem (cdr lst))
    (car lst)
  )
)

(defun last-elem (lst) (car (reverse lst)))
```

4. Написать, по крайней мере, два варианта функции, которая возвращает свой список-аргумент без последнего элемента.

Листинг 2: Решение задания №4

```
(defun no-last-elem (lst) (reverse (cdr (reverse lst))))



(defun no-last-elem-internal (lst acc)
  (if (cdr lst)
    (no-last-elem-internal (cdr lst) (append acc (cons (car lst) Nil)))
    acc
  )
)

(defun no-last-elem (lst) (no-last-elem-internal lst ()))
```

5. Написать простой вариант игры в кости, в котором бросаются две правильные кости. Если сумма выпавших очков равна 7 или 11 – выигрыш, если выпало (1,1) или (6,6) — игрок право снова бросить кости, во всех остальных случаях ход переходит ко второму игроку, но запоминается сумма выпавших очков. Если второй игрок не выигрывает абсолютно, то выигрывает тот игрок, у которого больше очков. Результат игры и значения выпавших костей выводить на экран с помощью функции print.

Листинг 3: Решение задания №5

```

(defun random-cube-value ()
  (list (random 7) (random 7)))

(defun dices-sum (pair)
  (+ (car pair) (car (cdr pair))))

(defun check-absolute-win (sum)
  (or (= sum 7) (= sum 11)))

(defun check-rerun (pair)
  (let* ((fst (car pair))
    (snd (car (cdr pair))))
  (or (= fst snd 1) (= fst snd 6))
  )
)

(defun dices ()
  (let* ((fst-dices-pair (random-cube-value))
    (fst-dices-sum (dices-sum fst-dices-pair)))
      (format T "Player one dices: ~a.~%" fst-dices-pair)
      (cond ((check-absolute-win fst-dices-sum)
```

```
23          (format T "Player one win!~%"))
24        ((check-rerun fst-dices-pair)
25          (format T "Rerun!~%") (dices))
26        (T (let* ((snd-dices-pair (random-cube-value))
27              (snd-dices-sum (dices-sum snd-dices-pair)))
28                (format T "Player two dices: ~a.~%" snd-dices-pair)
29                (cond ((check-absolute-win snd-dices-sum) (format T
                     "Player two win!~%"))
30                ((> fst-dices-sum snd-dices-sum) (format T "Player one
                     win!~%"))
31                (T (format T "Player two win!~%")))))))
32 )
```