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 №4 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 Задание 1

Чем принципиально отличаются функции `cons, list, append`?

Пусть `(setf lst1 '(a b)) (setf lst2 '(c d))`. Каковы результаты вычисления следующих выражений?

Листинг 1.1 – Решение задания 1

```
(cons lst1 lst2)   ;; ((A B) C D)
(list lst1 lst2)   ;; ((A B) (C D))
(append lst1 lst2) ;; (A B C D)
```

1.2 Задание 2

Каковы результаты вычисления следующих выражений, и почему?

Листинг 1.2 – Решение задания 2

```
(reverse ())          ;; NIL
(last ())             ;; NIL
(reverse '(a))        ;; (A)
(last '(a))           ;; (A)
(reverse '((a b c))) ;; ((A B C))
(last '((a b c)))     ;; ((A B C))
```

1.3 Задание 3

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

Листинг 1.3 – Решение задания 3

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

;; (LAST-ELEM '(1 2 3)) -> 3

(defun last-elem-2 (lst)
    (if (eq (cdr lst) nil)
        (car lst)
        (last-elem-2 (cdr lst))))

;; (LAST-ELEM-2 '(1 2 3)) -> 3

(defun last-elem-3 (lst)
    (car (last lst)))

;; (LAST-ELEM-3 '(1 2 3)) -> 3
```

1.4 Задание 4

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

Листинг 1.4 – Решение задания 4

```
(defun without-last-elem (lst)
    (reverse (cdr (reverse lst))))

;; (WITHOUT-LAST-ELEM '(1 2 3)) -> (1 2)

(defun without-last-elem-2 (lst)
    (if (cdr lst)
        (cons (car lst) (without-last-elem-2 (cdr lst)))
        ()))

;; (WITHOUT-LAST-ELEM-2 '(1 2 3)) -> (1 2)
```

1.5 Задание 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 получает 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 1.5 – Решение задания 5

```
(defvar name-first  'first_player)
(defvar name-second 'second_player)
(defvar dice-first)
(defvar dice-second)
(defvar tmp-dice)

(defun roll-one-dice ()
    (+ (random 6) 1))

(defun roll-two-dice ()
    (list (roll-one-dice) (roll-one-dice)))

(defun sum (dice)
    (+ (car dice) (cadr dice)))

(defun print-res (name dice)
    (print `(Win ,name)))

(defun is-win (dice)
    (cond ((= (sum dice) 7 ))
            ((= (sum dice) 11))
    )
)

(defun repeat-roll (dice)
    (cond ((= (car dice) (cadr dice) 6))
            ((= (car dice) (cadr dice) 1))
    )
)

(defun players-move (name)
    (setf tmp-dice (roll-two-dice)
```

```
    )
    (print `(,name ,tmp-dice sum = ,(sum tmp-dice))
    )
    (cond
        ((is-win tmp-dice)
            (list tmp-dice 1))
        ((repeat-roll tmp-dice)
            (players-move name))
        (T (list tmp-dice 0))
    )
)

(defun continue-game (dice-first)
    (setf dice-second (players-move name-second)
    )
    (cond
        ((= (cadr dice-second) 1)
            (print-res name-second dice-second))
        ((> (sum (car dice-first)) (sum (car dice-second)))
            (print-res name-first dice-first))
        ((< (sum (car dice-first)) (sum (car dice-second)))
            (print-res name-second dice-second))
        (T (print '(Draw)))
    )
)

(defun play ()
    (setf dice-first  (players-move name-first)
    )
    (cond
        ((= (cadr dice-first) 1)
            (print-res name-first dice-first))
        (T (continue-game dice-first))
    )
    (terpri) (terpri)
)

;; (FIRST_PLAYER (1 3) SUM = 4)
;; (SECOND_PLAYER (1 1) SUM = 2)
;; (SECOND_PLAYER (3 3) SUM = 6)
;; (WIN SECOND_PLAYER)

;; (FIRST_PLAYER (1 6) SUM = 7)
;; (WIN FIRST_PLAYER)

;; (FIRST_PLAYER (4 6) SUM = 10)
;; (SECOND_PLAYER (6 4) SUM = 10)
;; (DRAW)
```

2 Ответы на теоретические вопросы к лабораторной работе

2.1 Синтаксическая форма и хранение программы в памяти

В Lisp формы представления программы и обрабатываемых ею данных одинаковы и представляются в виде S-выражений. Программы могут обрабатывать и преобразовывать другие программы и сами себя. В процессе трансляции можно введенное и сформированное в результате вычислений выражение данных интерпретировать в качестве программы и непосредственно выполнить.

Так как программа представляет собой S-выражение, в памяти она представлена либо как атом, либо как точечная пара.

2.2 Трактовка элементов списка

Первый аргумент списка трактуется как имя функции, остальные – как аргументы этой функции.

2.3 Порядок реализации программы

Программа в языке Lisp представляется S-выражением, которое передается интерпретатору – функции eval, которая выводит последний, полученный после обработки S-выражения результат.

Работа функции eval представлена на рисунке 2.1.

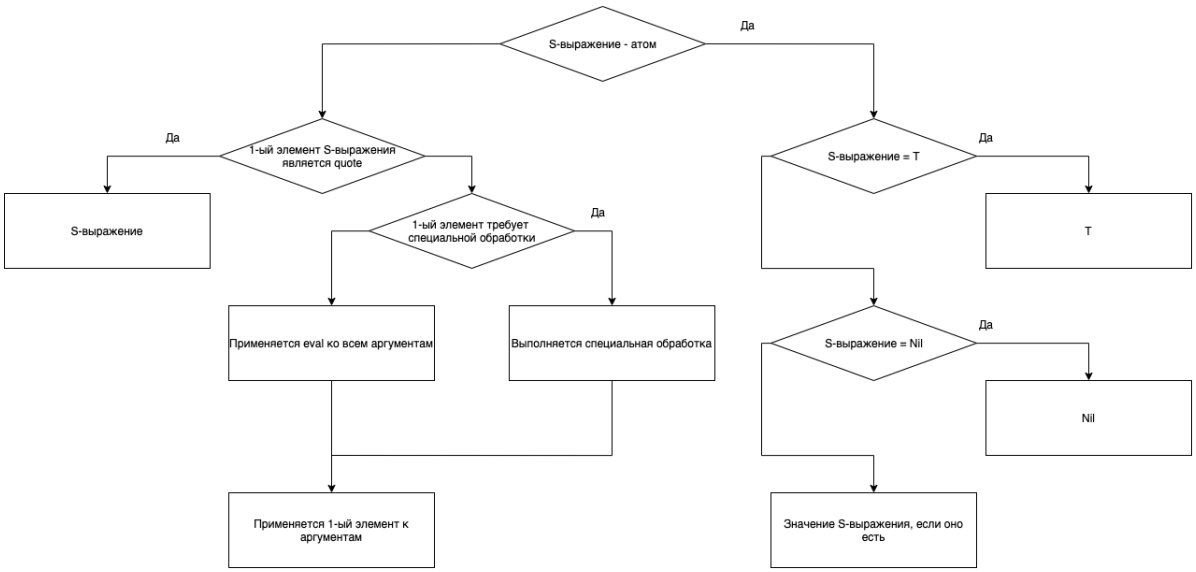

Рисунок 2.1 – Схема работы функции eval

2.4 Способы определения функции

Функцией называется правило, по которому каждому значению одного или нескольких аргументов ставится в соответствие конкретное значение результата.

- В Lisp можно определить функцию без имени с помощью λ**-выражений**. Lambda-определение безымянной функции:

  `(lambda <lambda-список> <форма>)`

  Lambda-вызов функции:

  `(<lambda-выражение> <формальные параметры>)`

- Также в Lisp можно определить функцию с именем с помощью **defun**. В таких функциях defun связывает символьный атом с Lambda-определением:

  `(defun f <lambda-выражение>)`

  Упрощенное определение:

  `(defun f(arg1, ..., argN) <формы>)`