Лабораторная работа №4 (Талышева ИУ7-65Б)

1. Теоретические вопросы:

1) Синтаксическая форма и хранение программы в памяти.

В языке Lisp программы и данные представлены в виде S-выражений. S-выражения могут быть атомами или точечными парами. Точечная пара имеет форму `(A . B)`, где `A` и `B` могут быть атомами или другими точечными парами. Списки являются частным случаем S-выражений и могут быть пустыми (например, `()` или `Nil`) или непустыми (например, `(A B C D)`). Непустые списки состоят из головы (S-выражение) и хвоста (также списка).

Синтаксически любая структура (точечная пара или список) заключается в круглые скобки. Например:

- **(A . B)** – точечная пара;
- **(A)** – список из одного элемента;
- **(A B C D)** – непустой список, который эквивалентен `(A . (B . (C . (D . Nil))))`;
- **()** или **Nil** – пустой список.

Элементы списка также могут быть списками, например: `(A (B C) (D (E)))`.

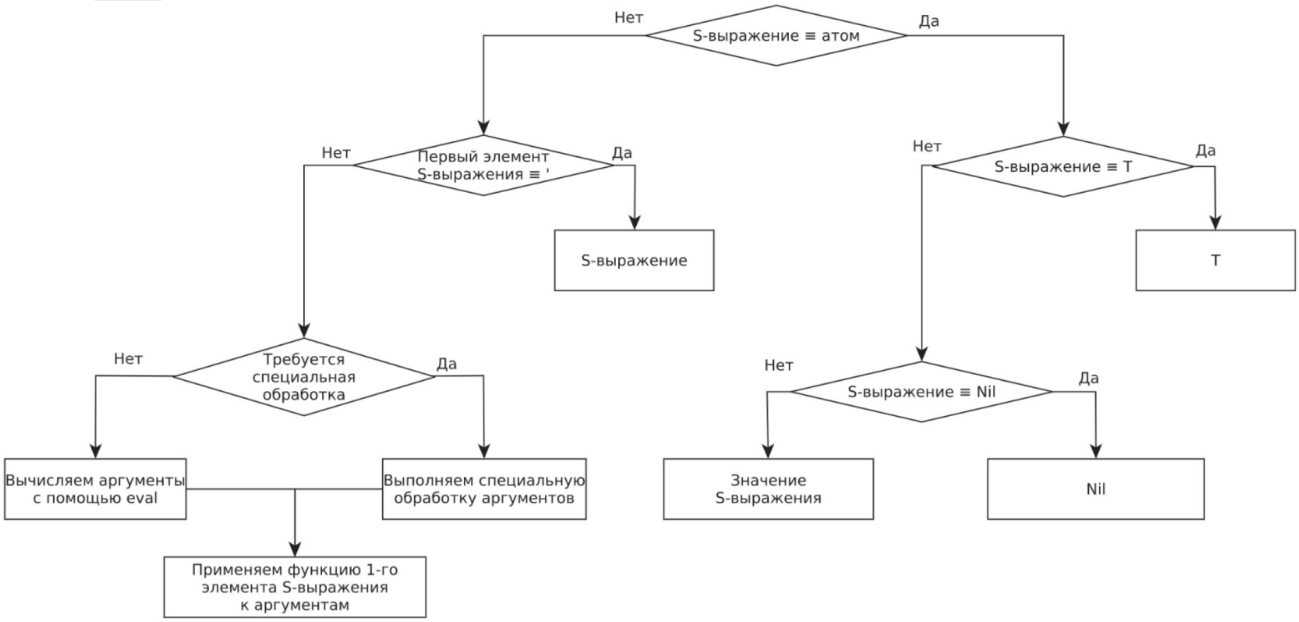
В памяти любая непустая структура Lisp представляется списковой ячейкой, которая хранит два указателя: на голову (первый элемент) и хвост (все остальные элементы). Программы в Lisp могут обрабатывать и преобразовывать другие программы или сами себя, так как они состоят из S-выражений и хранятся в виде бинарных узлов. Программа - это список, первый элемент которого имя функции, а остальное - её параметры.

2) Трактовка элементов списка.

Система считает, что первый элемент списка - имя функциии, а остальное её аргументы. Перед вызовом функции все её аргументы вычисляются. Чтобы список не рассматривался как имя функции, есть функция quote (сокращённо апостраф), которая возвращает свой аргумент без вычисления.

3) Порядок реализации программы.

Программа работает циклически: сначала она ожидает ввода S-выражения, затем передает его интерпретатору — функции `eval`. После выполнения `eval` программа выводит последний полученный результат. Алгоритм работы функции `eval` представлен на диаграмме:



4) Способы определения функции.

1) именованная функция

Синтаксис:

(defun имя_функции (список_аргументов) тело_функции)

Пример:

(defun add (a b) (+ a b))

(add 1 3)

-> 4

2) лямбда-выражение

Синтаксис:

(lambda (список_аргументов) тело_функции)

Пример:

((lambda (a b) (+ a b)) 1 3)

-> 4

5) Работа со списками.

Работа со списками является одной из ключевых особенностей языка Lisp, так как списки — это основная структура данных в Lisp. Список в памяти представляется как бинарная ячейка с указателями на голову и хвост списка. Функции в Lisp классифицируются следующим образом:

Конструкторы:

- `cons` — создает новую точечную пару, расставляя указатели на переданные 2 аргумента.
- `list` — создает список из элементов.
- `append` — объединяет списки.

Селекторы:

- `car` — возвращает первый элемент списка.
- `cdr` — возвращает список без первого элемента.

Предикаты:

- `null` — проверяет, пуст ли список.
- `atom` — проверяет, является ли элемент атомом.
- `listp` — проверяет, является ли элемент списком.

Функции сравнения:

- `eq` — сравнивает два объекта на идентичность.
- `equal` — сравнивает два объекта на равенство (в том числе списки).

2. Практические задания (Common Lisp):

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

Пусть

(setf lst1 '(a b c))

(setf lst2 '(d e))

Каковы результаты вычисления следующих выражений?

In [ ]:
```
(cons lst1 lst2)           ->        ((A B C) D E)
(list lst1 lst2)           ->        ((A B C) (D E))
(append lst1 lst2)         ->        (A B C D E)
```

2) Каковы результаты вычисления следующих выражений, и почему?

In [ ]:
```
(reverse '(a b c))             ->        (C B A)
(reverse ())                   ->        NIL
(reverse '(a b (c (d))))       ->        ((C (D)) B A)
(reverse '((a b c)))           ->        ((A B C))
(reverse '(a))                 ->        (A)
```

```
(last '(a b c))                 ->          (C)
(last '(a b (c)))               ->          ((C))
(last '(a))                     ->          (A)
(last '())                      ->          NIL
(last '((a b c)))               ->          ((A B C))
```

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

In [ ]:
```
(defun return-last-1 (lst) (car (last lst)))

(defun return-last-2 (lst) (car (reverse lst)))
```

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

In [ ]:
```
(defun return-without-last-1 (lst)
    (reverse (cdr (reverse lst))))

(defun return-without-last-2 (lst)
    (cond ((null (cdr lst)) Nil)
          (t (cons (car lst) (return-without-last-2 (cdr lst))))))
```

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

In [ ]:
```
(defun swap-first-last (lst)
  (let ((a (car lst))
        (c (car (last lst))))
     (cons c (reverse (cons a (cdr (reverse (cdr lst))))))))
```

6) Написать простой вариант игры в кости, в котором бросаются две игральные кости. Если сумма выпавших очков равна 7 или 11, игрок выиграл. Если выпало (1,1) или (6,6), игрок имеет 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.

In [ ]:
```
(setf *random-state* (make-random-state t)) ;; Setting up random numbers

;; Rolls two dice for a given player and returns the results
(defun throw-dices (player)
  (let ((d1 (+ 1 (random 6)))
        (d2 (+ 1 (random 6))))
     (print (list "Player" player "rolled" d1 d2))
     (list d1 d2 (+ d1 d2))))

;; Checks if a reroll is needed (if both dice are 1s or 6s)
(defun should-reroll? (lst-res-throw)
  (let ((d1 (car lst-res-throw))
        (d2 (cadr lst-res-throw)))
     (or (and (= d1 1) (= d2 1))
         (and (= d1 6) (= d2 6)))))

;; Rolls dice and rerolls if necessary
(defun roll-dices-with-reroll (player)
    (let ((result (throw-dices player)))
       (if (should-reroll? result) (throw-dices player) result)))

;; Checks for an instant win (sum is 7 or 11)
(defun check-instant-win (sum)
  (if (or (= sum 7) (= sum 11)) 'win sum))

;; Handles a player's turn: rolls dice and checks for an instant win
(defun play-player (player)
  (let ((result (roll-dices-with-reroll player)))
     (check-instant-win (caddr result))))

;; Determines the winner based on dice sums
(defun determine-winner (sum1 sum2)
  (cond ((> sum1 sum2) (print "Player 1 wins!"))
        ((< sum1 sum2) (print "Player 2 wins!"))
        (t (print "It's a draw!"))))

;; Runs the game: both players take turns, and the winner is determined
(defun play ()
  (let ((p1 (play-player 1)))
    (if (eql p1 'win)
        (print "Player 1 wins!")
```

```
        (let ((p2 (play-player 2)))
          (if (eql p2 'win)
              (print "Player 2 wins!")
              (determine-winner p1 p2))))))
```

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

In [ ]:
```
(defun del-first-last (lst)
  (reverse (cdr (reverse (cdr lst)))))

(defun is_palindrome (lst)
  (cond ((null lst) t)
        ((null (cdr lst)) t)
        (t (let ((a (car lst))
                 (c (car (last lst))))
             (cond ((equal a c)
                    (is_palindrome (del-first-last lst)))
                   (t Nil))))))
```

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

In [ ]:
```
(defun country-capital (table country)
  (cond ((null table) Nil)
        ((eql (caar table) country) (cdar table))
        (t (country-capital (cdr table) country))))

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

9) Напишите функцию, которая умножает на заданное число-аргумент первый числовой элемент списка из заданного 3-х элементного списка-аргумента, когда:

а) все элементы списка — числа:

In [ ]:
```
(defun multy-lst-num (lst num)
    (if (and (numberp (car lst))
             (numberp (cadr lst))
             (numberp (caddr lst)))
          (* (car lst) num)))
```

б) элементы списка — любые объекты.

In [ ]:
```
(defun multy-lst-num (lst num)
    (cond ((null lst) Nil)
          ((numberp (car lst)) (* (car lst) num))
          (t (multy-lst-num (cdr lst) num))))
```