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 курсу "Функциональное и логическое программирование"

Тема Использование управляющих структур, работа со списками

Студент Гурова Н.А.

Группа ИУ7-64Б

Оценка (баллы) _____

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

Москва — 2023 г.

1 Теоретические вопросы

1.1 Синтаксическая форма и хранение программы в памяти

Программа на Lisp представляет собой вызов функции на верхнем уровне. Все операции над данными оформляются и записываются как функции, которые имеют значение, даже если их основное предназначение — осуществление некоторого побочного эффекта. Программа является ничем иным, как набором запрограммированных функций.

Синтаксически программа оформляется в виде S-выражения (обычно – списка – частного случая точечной пары), которое очень часто может быть структурированным. Наличие скобок является признаком структуры.

Определения:

- S-выражение ::= <атом> | <точечная пара>
- Атомы:
  - символы (идентификаторы) — синтаксически — набор литер (букв и цифр), начинающихся с буквы;
  - специальные символы — T, Nil (используются для обозначения логических констант);
  - самоопределимые атомы — натуральные числа, дробные числа, вещественные числа, строки – последовательность символов, заключенных в двойные апострофы (например, "abc");
- Точечная пара:

```
1    Точечная пара ::= (<атом> . <атом>) |
2                      (<атом> . <точечная пара>) |
3                      (<точечная пара> . <атом>) |
4                      (<точечная пара> . <точечная пара>)
```

- Список:

```
1    Список ::= <пустой список> | <непустой список>, где
2
3    <пусой сисок> ::= ( ) | Nil,
4    <непустой список>::= (<первый элемент> . <хвост>),
5    <первый элемент> ::= <S–выражение>,
6    <хвост> ::= <список>
```

Атомы представляются в памяти пятью указателями (name, value, function, property, package), а любая непустая структура — списковой ячейкой (бинарным узлом), хранящей два указателя: на голову (первый элемент) и хвост — все остальное.

1.2 Трактовка элементов списка

По определению списка, приведенному выше: если список непустой, то он представляет из себя точечную пару из <первого элемента> и <хвоста>, где <первый элемент> — это <S-выражение>, а <хвост> — это <список>.

Список можно вычислить, если он представляет собой обращение к функции, или функциональный вызов: (f e1 e2 ... en), где f — символьный атом, имя вызываемой функции; e1, e2, ..., en — аргументы этой функции; n - число аргументов функции.

В случае n = 0 имеем вызов функции без аргументов: (f). Обычно e1, e2, ..., en являются вычислимыми выражениями и вычисляются последовательно слева направо.

Таким образом, если в процессе работы лисп-интерпретатора требуется вычислить некоторый список, то первым элементом этого списка должно быть имя функции. Если это не так, лисп-интерпретатор сообщает об ошибке и прерывает вычисление текущего выражения программы.

1.3 Порядок реализации программы

Обычно лисп-программа включает определения новых функций на базе встроенных функций и других функций, определённых в этой программе, а также вызовы этих новых функций для конкретных значений их аргументов.

Программа на Lisp представляет собой вызов функции на верхнем уровне и синтаксически оформляется в виде S-выражения. Вычисление программы реализует лисп-интерпретатор, который считывает очередную входящую в программу форму, вычисляет её (анализирует функцией eval) и выводит полученный результат (S-выражение).

Eval выполняет двойное вычисление своего аргумента. Эта функция является обычной, и первое вычисление аргумента выполняет так же, как и любая обычная функция. Полученное при этом выражение вычисляется ещё раз. Такое двойное вычисление может понадобиться либо для снятия блокировки вычислений (установленной функцией quote), либо же для вычисления сформированного в ходе первого вычисления нового функционального вызова.

1.4 Способы задания функций

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

Функционал (функция высшего порядка) — функция, аргументом или результатом которой является другая функция.

Определение функций пользователя в Лиспе возможно двумя способами.

1.4.1 Базисный способ определения функции

Предполагает использование λ-выражения. Так создаются функции без имени.

Способ задания функции: **λ-выражение: (lambda λ-список форма)**, где λ-список — формальные параметры функции, форма — тело функции.

Вызов такой функции осуществляется следующим образом: **(λ-выражение формы)**, где формы — это фактические параметры.

Вычисление функций без имени может быть также выполнено с помощью функционала **apply**: **(apply λ-выражение формы)**. Функционал apply является обычной функцией с двумя вычисляемыми аргументами, обращение к ней имеет вид

```
1    apply F L          ;   F — функциональный аргумент
2                       ;   L — список фактических параметров
```

Значение функционала — результат применения F к этим фактическим параметрам.

Вычисление функций без имени может быть также выполнено с помощью функционала **funcall**: **(funcall λ-выражение формы)**. Функционал funcall — особая функция с вычисляемыми аргументами, обращение к ней

```
1    funcall  F e1 .. en          ; n >= 0
```

Действие аналогично apply, отличие в том, что аргументы передаются не в виде списка, а по отдельности.

1.4.2 Использование макро-определения defun

Задание функции: **(defun имя-функции λ-выражение)** или в облегченной форме **(defun имя-функции (x_1, x_2, .. x_k) форма)**.

Вызов именованной функции: **(имя-функции последовательность-форм)**.

Для вызова можно также воспользоваться функционалами funcall и apply.

```
1    (foo 1 2 3)
2    (funcall  #'foo 1 2 3)
3    (apply #'foo (1 2 3))
```

1.5 Работа функций COND, IF, AND/OR

1.5.1 COND

Общий вид условного выражения:
(COND $(p_1\ e_{11}$ .. $e_{1m_1})$ $(p_2\ e_{21}$ .. $e_{2m_2})$ .. $(p_n\ e_{nn}$ .. $e_{nm_n}))$, $m_i \geq 0$, $n \geq 1$
Вычисление условного выражения выполняется по следующим правилам:

1. последовательно вычисляются условия $p_1..p_n$ ветвей выражения до тех пор, пока не встретится выражение p_i, отличное от NIL;

2. последовательно вычисляются выражения-формы $e_{i1}..e_{im_i}$ соответствующей ветви, и значение последнего выражения e_{im_i} возвращается в качестве значения функции cond;

3. если все условия p_i имеют значения NIL, то значением условного выражения становится NIL.

Пример:

```
(
cond
   ((< X 5) (print "a") X)
   ((= X 9) (print "b") X)
   (T       (print "c") X)
)

; Возвращает всегда X, при этом выведена будет одна из трех строк
```

1.5.2 IF

Макрофункция (IF C E1 E2), вычисляет значение выражения E1, если значение выражения C отлично от NIL, в ином случае она вычисляет E2.

```
(if c e1 e2) == (list 'cond (list c e1) (list T e2))

(if (< 3 5) (print "a") (print "b"))
   ==
(list 'cond (list (< 3 5) (print "a")) (list T (print "b")))
```

1.5.3 AND/OR

Вызов функции and, реализующей конъюнкцию, имеет вид (AND e_1 .. e_n).

При выполнении этого функционального обращения последовательно слева направо вычисляются аргументы e_i, до тех пор, пока не встретится значение, равное NIL. В этом случае выполнение функции прерывается, возвращается NIL. Если все e_i отличны от NIL, то результатом будет e_n.

Вызов функции or, реализующей конъюнкцию, имеет вид (OR e_1 .. e_n).

При выполнении этого функционального обращения последовательно слева направо вычисляются аргументы e_i, до тех пор, пока не встретится значение,

отличное от NIL. В этом случае выполнение функции прерывается, возвращается e_i. Если все e_i равны NIL, то результатом будет NIL.

Таким образом, значения функций and и or не обязательно равно T и NIL, а может быть произвольным атомом или списочным выражением.

1.5.4 APPEND

Функция append объединяет два списка.

Идея работы функции состоит в том, что рекурсивно откладываются вызовы функции CONS с элементами списка X до тех пор, пока он не исчерпается, после чего в качестве результата возвращается указатель на список Y и отложенные вызовы, завершая свою работу, формируют результат.

```
1     (
2         defun
3         my_append
4         (x y)
5         (
6         cond ((null x) y)
7               (t (cons (car x)
8                        (my_append (cdr x) y)
9                  )
10              )
11        )
12    )
13
14    (my_append '(1 2) '(3 4))        ; (1 2 3 4)
15    (append '(1 2) '(3 4))           ; (1 2 3 4)
```

1.5.5 REVERSE

Функция reverse изменяет порядок элементов в списке (на верхнем уровне) на обратный. Идея определения функции состоит в следующем: берем первый элемент списка, делаем из него одноэлементный список и объединяем его функцией append с перевернутым хвостом. Хвост списка сначала обращается рекурсивным вызовом.

```
(
    defun
    my_reverse
    (l)
    (
    cond ((null l) nil)
         (t (append (my_reverse (cdr l))
                    (cons (car l) nil)
            )
         )
    )
)

 (defvar a '(1 2 3))
(my_reverse a)          ; (3 2 1)
(reverse a)             ; (3 2 1)
```

1.5.6 LAST

С помощью этой функции можно выделить последний элемент списка. Как такую функцию можно определить:

```
(
   defun
   my_last
   (l)
   (cond ((null l) nil)
         ((null (cdr l)) (car l))
         (t (my_last (cdr l)))
   )
)

(defvar a '(1 2 3))
(my_last a)                    ; (3)
(last a)                       ; (3)
```

2 Практические задания

2.1 Чем принципиально отличаются функции cons, list, append?

Отличия:

- cons является базисной, list и append – нет;
- list и append принимают произвольное количество аргументов (причем аргументами append могут быть только списки), cons – фиксированное (два);
- cons создает точечную пару или список (в зависимости от второго аргумента), list и append – список;
- cons и list создают новые списковые ячейки (все), а append имеет общие списковые ячейки с последним списком.
- list и append определяются с помощью cons.

```
1  (setf lst1 '(a b c))
2  (setf lst2 '(d e))
3
4  (cons lst1 lst2)            ; ((a b c) d e)
5  (list lst1 lst2)            ; ((a b c) (d e))
6  (append lst1 lst2)          ; (a b c d e)
```

2.2 Каковы результаты вычисления следующих выражений, и почему?

Функция reverse переворачивает свой список-аргумент, т.е. меняет порядок ; его элементов верхнего уровня на противоположный.

```
1  (reverse '(a b c))              ; (c b a)
2  (reverse '(a b (c (d))))        ; ((c (d)) b a)
3  (reverse '(a))                  ; (a)
4  (reverse ())                    ; ()
5  (reverse '((a b c)))            ; (a b c)
```

Функция last возвращает последнюю cons-ячейку в списке.

```
(last '(a b c))                    ; (c)
(last '(a))                        ; (a)
(last '((a b c)))                  ; ((a b c))
(last '(a b (c)))                  ; ((c))
(last ())                          ; ()
```

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

```
(
   defun
   my_last1
   (l)
   (cond ((null l) nil)
         ((null (cdr l)) (car l))
         (t (my_last (cdr l))))
   )
)

(my_last1 a)                  ; (3)
```

```
(
   defun
   my_last2
   (l)
   (cons (car (reverse l) nil))
)

(my_last2 a)                  ; (3)
```

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

```
(
    defun
    without_last1
    (l)
    (
    if
        (null (cdr l))
        nil
        (cons (car l) (without_last1 (cdr l)))
    )
)

(WITHOUT_LAST1 a)           ; (1 2)
```

```
(
    defun
    without_last2
    (l)
    (reverse (cdr (reverse l)))
)

(WITHOUT_LAST2 a)           ; (1 2)
```

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

```
(
    defun
    swap-first-last
    (a)
    (
        append
        (cons(car (reverse (cdr a))) nil)
        (reverse (cdr (reverse (cdr a))))
        (cons (car a) nil)
    )
```

```
11      )
12
13      (SWAP-FIRST-LAST '(1 2 3 4 5))          ; (5 2 3 4 1)
14      (SWAP-FIRST-LAST '((1 2) 3 4 (4) 5))    ; (5 3 4 (4) (1 2))
```

2.6 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 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

```
1     (
2         defun
3         roll_dice
4         ()
5         (+ (random 6) 1)
6     )
7
8     (
9         defun
10        check_continue_game
11        (result)
12        (not (or (= result 7) (= result 11)))
13    )
14
15    (
16        defun
17        make_a_move
18        (player_i)
```

```
19          (
20              let (
21                  (dice1 (roll_dice))
22                  (dice2 (roll_dice))
23              )
24          (
25              if (
26                  and
27                      (print (list 'Игрок player_i 'бросает 'кости))
28                      (= dice1 dice2)
29                      (or (= dice1 1) (= dice1 6))
30                  )
31              (
32                  and
33                  (print (list 'Выпало  dice1 dice2  'Повторный 'бросок))
34                  (make_a_move player_i)
35              )
36              (
37                  and
38                  (print (list 'Выпало dice1 dice2))
39                  (+ dice1 dice2)
40              )
41          )
42          )
43      )
44      (
45          defun
46          compare_results
47          (res1 res2)
48          (
49              if
50              (check_continue_game res2)
51            (
52              and
53                (print (list 'Сравнение 'по 'очкам))
54                (print (list 'Игрок 1 'набрал res1))
55                (print (list 'Игрок 2 'набрал res2))
56                (cond
```

```
57                  ((< res1 res2) (and (print (list 'Игрок 2 'выиграл 'по
                        'очкам)) 2))
58                  ((> res1 res2) (and (print (list 'Игрок 1 'выиграл 'по
                        'очкам)) 1))
59                  ((and (print '(Ничья)) 0))
60                )
61              )
62              (
63                and
64                  (print (list 'Игрок 2 'набрал res2 'очков 'и 'выиграл
                        'абсолютно)) 2
65                )
66            )
67        )
68
69        (
70            defun
71            play_game
72            ()
73            (
74                let (
75                    (res1 (make_a_move 1))
76                )
77                (if (check_continue_game res1)
78                    (compare_results res1 (make_a_move 2))
79                    (and (print (list 'Игрок 1 'набрал res1 'очков 'и
                        'выиграл 'абсолютно)) 1)
80                )
81            )
82        )
83
84
85        ; (ИГРОК 1 БРОСАЕТ КОСТИ)
86        ; (ВЫПАЛО 1 5)
87        ; (ИГРОК 2 БРОСАЕТ КОСТИ)
88        ; (ВЫПАЛО 6 1)
89        ; (ИГРОК 2 НАБРАЛ 7 ОЧКОВ И ВЫИГРАЛ АБСОЛЮТНО)
```

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

```
(
    defun
    is_palindrom
    (l)
    (equal l (reverse l))
)

(IS_PALINDROM '(1 2 3 4 5))     ; NIL
(IS_PALINDROM '(1 2 3 2 1))     ; T
(IS_PALINDROM '((1 2) 3 (2 1))) ; NIL
(IS_PALINDROM '((1 2) 3 (1 2))) ; T
```

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

```
(
defvar country_table
'(
   (Россия . Москва)
   (Англия . Лондон)
   (Италия . Рим)
   (Испания . Мадрид)
)
)

(
    defun
    get_country_by_capital
    (capital table)
    (
```

```
16              cond
17                  ((null table) "Не найдено")
18                  ((equal (cdr (car table)) capital) (car (car table)))
19                  (t (get_country_by_capital capital (cdr table)))
20          )
21      )
22
23      (
24          defun
25          get_capital_by_country
26          (country table)
27          (
28              cond
29                  ((null table) "Не найдено")
30                  ((equal (car (car table)) country) (cdr (car table)))
31                  (t (get_capital_by_country country (cdr table)))
32          )
33      )
34
35      (
36          defun
37          get_by_value
38          (value table)
39          (
40              if
41              (equal "Не найдено" (get_country_by_capital value table))
42              (get_capital_by_country value table)
43              (get_country_by_capital value table)
44          )
45      )
46
47      (print (get_country_by_capital 'ФФФФ country_table))
48      (print (get_country_by_capital 'МОСКВА country_table))
49      (print (get_country_by_capital 'ЛОНДОН country_table))
50      (print (get_country_by_capital 'МАДРИД country_table))
51
52      (print (get_capital_by_country 'ФФФФ country_table))
53      (print (get_capital_by_country 'РОССИЯ country_table))
54      (print (get_capital_by_country 'АНГЛИЯ country_table))
```

```
55      (print (get_capital_by_country 'ИСПАНИЯ country_table))
56
57      (print (get_by_value 'FFF country_table))
58      (print (get_by_value 'Рим country_table))
59      (print (get_by_value 'Италия country_table))
```

2.9 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.

```
1       (
2           defun
3           simple_option
4           (x l)
5           (
6               cons (* (car l) x) (cdr l)
7           )
8       )
9
10      (print (simple_option 5 '(1 2 3 4 5)))                    ; (5 2 3 4
            5)
11
12
13      (
14          defun
15          difficult_option
16          (x l)
17          (
18              cond
19              ((null l) "Невозможно умножить (нет числовых значений)")
20              ((numberp (car l)) (cons (* (car l) x) (cdr l)))
21              (t (cons (car l) (difficult_option x (cdr l))))
22          )
23      )
24
```

```
25    (difficult_option 5 '(1 2 3 4 5))                ; (5 2 3 4 5)
26    (difficult_option 5 '((1) 2 3 4 5))              ; ((1) 10 3 4 5)
27    (difficult_option 5 '("1" (2 3 4) 5))            ; ("1" (2 3 4) 25)
28    (difficult_option 5 '("1" (2 3 4) (5)))          ; "Невозможно умножи
        ть (нет числовых значений)"
```