МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студентка гр. 7382 _____ Дерябина П.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Задание.

Вариант 21. Арифметическое выражение (т. е. допустимы операции сложения, вычитания, деления и умножения), которое надо вычислить, представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в постфиксной форме (<аргументы><операция>). Аргументов может быть 1, 2 и более. Например, (a(b(c-)*)+). На входе также дополнительно даётся список значений переменных ((x_1 c_1) (x_2 c_2) … ($x_к$ $c_к$)), где x_i – переменная, а c_i – её значение (константа).

Пояснение к заданию.

Нужно написать программу, принимающую на вход постфиксное арифметическое выражение, аргументы выражения и их значения. Пример интерпретации выражений: (abc/)=a/b/c; (a/)=a; (ab+)=a+b; (a+)=a; (abc-)=ab-c; (a-)=a; (a*b)=a*b; (a*)=a. Для вычисления выражения необходимо написать функции, создающие иерархический список, а также функции, вычисляющие значение выражения, которое хранится в данном иерархическом списке.

Описание алгоритма.

1. Описание алгоритма создания иерархического списка.

Для реализации алгоритма рассматриваем строку, содержащую арифметическое постфиксное выражение, с конца. Последний символ строки — закрывающая скобочка, перед которой стоит знак операции («+», «-», «*» или «/»), а перед знаком стоят аргументы или очередные закрывающие или открывающие скобочки.

Последовательно для каждого элемента создаем узел иерархического списка (содержащий операцию или аргумент; флаг, показывающий атом этот элемент или нет; а также указатели на следующий элемент и на

подсписок), причем если данный элемент не является атомом, то кроме узла следующего элемента создаем также узел подсписка (см. рис 1) и рекурсивно переходим на этот подсписок, чтобы заполнить его. После выхода из подсписка продолжается заполнение предыдущего уровня рекурсии.

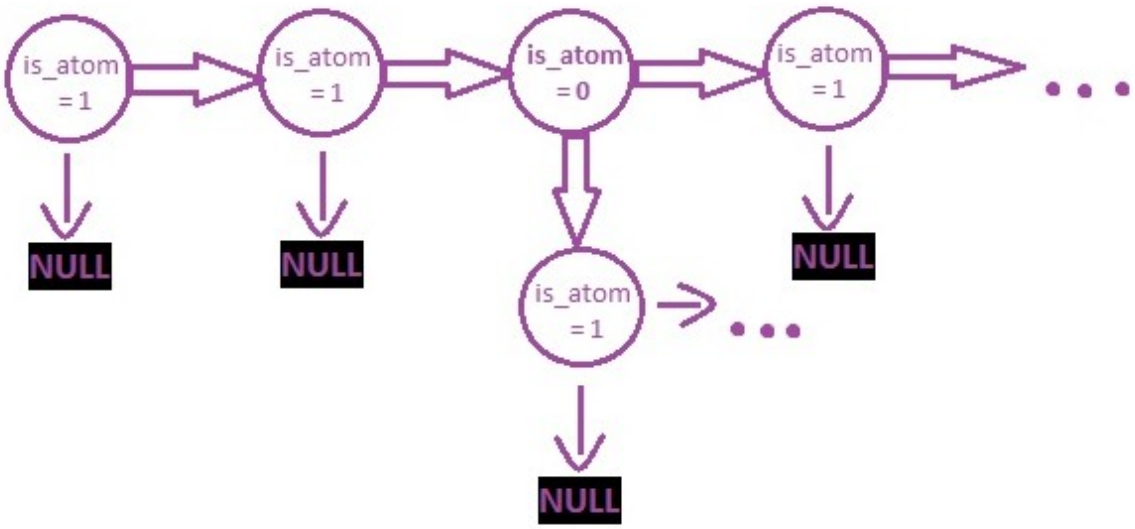

Рисунок 1 — заполнение списка

2. Описания алгоритма вычисления выражения

Первым элементом каждого подсписка является операция, которой надо подвергнуть каждый элемент данного подсписка. Таким образом перемещаемся по списку, пока не достигнем конца, применяя операцию текущего подсписка к его элементам, причем если встретился элемент, не являющийся атомом, то рекурсивно вызываем для него алгоритм, который возвращает его значение и переходит к следующему элементу (см. рис. 2).

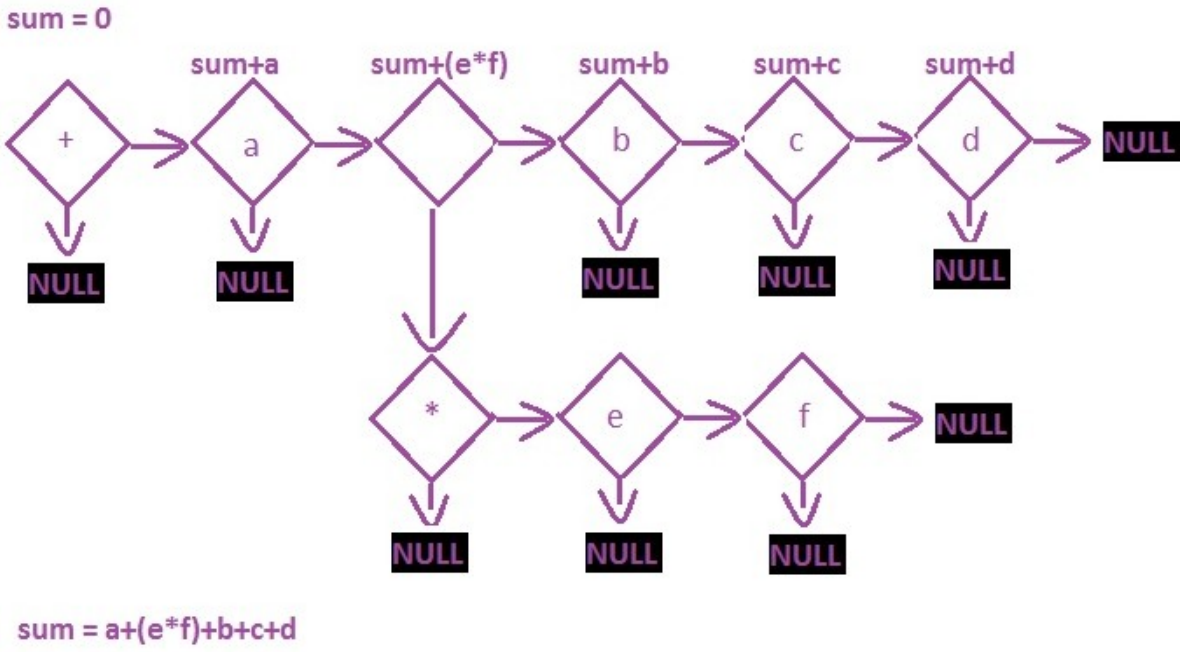

Рисунок 2 — вычисление списка

Описание функций и структур данных.

1. Структура данных Args.

Исходный код структуры:

```
typedef struct  Args{
 double value;
char var;
}Args;
```

Данная структура данных предназначена для хранения аргументов (char var) и их значений (double value).

2. Структура данных Node

Исходный код структуры:

```
typedef struct Node{
union{
char operation;
```

```
char var;
};
int is_atom;
struct Node *next;
struct  Node *sublist;
} Node;
```

Данная структура данных предназначена для узлов иерархического списка. Объединение, состоящее из символьных переменных operation и var обеспечивает экономию памяти и хранит либо операцию подсписка, либо аргумент. Переменна is_atom равна 1, если элемент является атомом и 0 — иначе. Указатель next предназначен для хранения адреса следующего элемента подсписка, указатель sublist хранит адрес следующего подсписка, если данный элемент не атом.

3. Функция createNode

Исходный код представлен в приложении А.

Данная функция создает узел списка, узел формируется из параметров, которые принимает функция: oper_or_var — переменная хранит операцию или аргумент, is_atom — определяет, является ли элемент атомом. Указатели next и sublist указывают на NULL.

4. Функция createList

Исходный код представлен в приложении А.

Данная функция создает иерархический список и принимает строку, содержащую арифметическое выражения и вспомогательную переменную для выводы вызовов рекурсии. Функция перебирает символы строки с конца, определяет является ли текущий элемент атомом и создает для него либо узел, куда записывает данные этого атома, либо создает для него подсписок, для которого происходит рекурсивный вызов функции, который заполняет значениями и его. После выхода из рекурсивного вызова продолжается заполнение предыдущего подсписка.

5. Функция to_calculate

Исходный код представлен в приложении А.

Функция вычисляет значение арифметического выражения и принимает 4 аргумента: массив args, хранящий аргументы и их значения; указатель на первый элемент списка list; размер массива size; и вспомогательную переменную level для выводы рекурсивных вызовов.

6. Функция перебирает элементы списка, применяя к ним операцию их подсписка, если встречается элемент, не являющийся атомом, то происходит рекурсивный вызов функции, который также возвращает значение этого подсписка, применяет к нему соответствующую операцию и переходит к следующему элементу.

7. Функция is_expression_correct

Исходный код представлен в приложении А.

Принимает строку, содержащую арифметическое выражение, проверяет на корректность: равное кол-во открывающих и закрывающий скобочек, наличие операции для каждого подсписка, наличие аргументов для списка, отсутствие пробелов в выражении.

8. Функция is_arg_correct

Исходный код представлен в приложении А.

Принимает на вход строку, содержащую один аргумент и его значение в формате <аргумент значение>. Проверяет на корректность, а именно: наличие пробела, аргумент должен быть буквой, а значение действительным числом.

9. Функция get_value

Исходный код представлен в приложении А.

Принимает 4 параметра: массив args с аргументами и их значениями, длину size массива args, аргумент var, чье значение нужно найти, переменную level для вывода вызовов рекурсии.

С помощью цикла функция находит соответствие между аргументом var и его значением в массиве args.

10. Функция Spaces

Исходный код представлен в приложении А.

Вспомогательная функция, принимающая на вход переменную level, хранящую уровень рекурсии. Функция выводит отступы для корректного отображения рекурсивных вызовов.

11. Функция get_vars

Исходный код представлен в приложении А.

Функция принимает строку, содержащую арифметическое выражение в качестве параметра и возвращает кол-во уникальных аргументов в выражении.

14. Функция delete_list()

Исходный код функции представлен в приложении А.

Функция принимает указатель на первый и второй элементы списка. С помощью цикла идет прохождение по всем элементам всех подсписков списка, и на каждой итерации удаляется первый элемент, затем первым становится второй, а вторым следующий за вторым. Цикл работает, пока нулевой элемент не станет указывать на NULL

13. Головная функция main

Исходный код представлен в приложении Б.

Функция считывает арифметическое выражение, вызывает для его функцию проверки на корректность, вычисляет кол-во уникальных аргументов выражения и создает для них массив. Далее происходит считывание аргументов с проверкой на корректность. В конце вызываются функции создания иерархического списка и вычисления его значения, результат записывается в файл.

Тестирование

Для более наглядной демонстрации работы программы был создан

ряд тестов и bash-скрипт, последовательно выводящий содержимое очередного теста и результат работы программы для этого теста. Код bashскрипта представлен в приложении Г, результат работы скрипта — в приложении В.

Рассмотрим тест 1:

Входные данные:

N

(abc+)

a 1

b 2

c 3

Сначала выводится приветственный текст и краткое пояснение к программе. Далее выводится блок «Forming list», где отображены вызовы рекурсии с ее глубиной, причем сами вызовы выведены подчеркнутым текстов для удобства восприятия. Между выводом вызовов рекурсии выводятся процесс вызова функции createNode с отображением того, какой узел создался.

Следующий блок «Calculating» также выводит рекурсивные вызовы функции to_calculate и также как между ними вычисляется значение каждого подсписка.

В конце выводится окончательный результат - значение арифметического выражения.

В табл. 1 представлены входные и выходные данные всех тестов

Таблица 1 — входные и выходные данные

Входные данные	Выходные данные
(abc+) a 1 b 2 c 3	Your expression is equally 6

(abc-) a 6 b 5 c 2.5	Your expression is equally -1.5
(abc/) a 10 b 2 c 10	Your expression is equally 0.5
(abc*) a 1 b 2 c 10	Your expression is equally 20
(ab-) a 5 b 1	Your expression is equally 4
(ba-) a 5 b 1	Your expression is equally -4
((abc*)(abc-)(abc/)+) a 5 b 2 c 1	Your expression is equally 14.5
((a+)(b/)(c*)(d-)+) a 0.006 b 0.06 c 0.6 d 6	Your expression is equally 6.666
(ab(c(d-)*)e+) a 55 b -15 c -1 d 10 e 13.13	Your expression is equally 43.13
((a+)b(bb/)-) a 100 b 2	Your expression is equally 97
(a(a(a+)+)) a 27	Operation before the bracket ')' is missed!
(abcd/) a 10 b 20 ccccccccccc d 30	Space between argument and its value!
(a(b(c(d-)+)/)*) a 10 b 2 c 3	You didn't enter value for var 'd', so lets suppose d=1 Your expression is equally 5

X 360	

Выводы.

В ходе работы была закреплена тема рекурсии, изучены иерархические списки, а также работа с ними.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
//#define SHOW_RECURSION

// Function to swap values at two pointers */
void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* This function takes three parameters:
    1. String
    2. Starting index of the substring
    3. Ending index of the substring.
    4. Counting var for array results
    5. Array where permutations will be in  */
void permute(char *str, int left, int right, int *num,
char **result)
{

#ifdef SHOWRECURSION // block for demonstrating
recursion's work
    if(left == 0)
    {
        printf("Work of recursion:\n");
    }
    char space[] = "   ";
```

```
        for(int i = 0; i<left; i++)
        {
            printf("%s", space);
        }
        printf("permute: level %d\n", left+1);
    #endif

        if (left == right)
        {
            strcpy(result[*num], str);
            (*num)++;
        }
        else
        {
            for (int i = left; i <= right; i++)
            {
                    swap((str+left), (str+i)); //symbol with
index i is fixed
                  permute(str, left+1, right, num, result);
//call function for substring
                swap((str+left), (str+i));
            }
        }
    }

    // calculation of factorial
    int fact(int n)
    {
        int res = 1;
        for(int i = n; i!=1; i--)
        {
            res*=i;
        }
        return res;
```

```
    }

    // to clean file output.txt
    void clean_file()
    {
        FILE *f = fopen("./output.txt", "w");
        fclose(f);
    }

    // returns 1 if input data is correct
    int is_correct(char *str)
    {
        for(int i = 0; i<strlen(str); i++)
        {
            if(str[i] == '0')
            {
                    printf("An acceptable symbol: [%c]\n",
str[i]);
                return 0;
            }
            if(isdigit(str[i]) == 0 && isalpha(str[i]) == 0)
            {
                    printf("An acceptable symbol: [%c]\n",
str[i]);
                return 0;
            }
        }
        return 1;
    }

    int main()
    {
        char str[30];
        char **result;
```

```
        int num = 0;
        int len;
        int fact_len;
         printf("Hello! This is permutation maker. Enter your
sequance: ");
        fgets(str, 28, stdin);
        printf("\n");
        if(str[strlen(str)-1] == '\n')
        {
            str[strlen(str)-1] = '\0';
        }
        if(!is_correct(str))
        {
            return 0;
        }

        len = strlen(str);
        fact_len = fact(len);
        result = calloc(fact_len, sizeof(char*));
        for(int i = 0; i<fact_len; i++)
        {
            result[i] = calloc(len+1, sizeof(char));
        }

        permute(str, 0, len-1, &num, result);

        printf("Your sequance: %s\n", str);
        clean_file();
        FILE *f = fopen("./output.txt", "a");
        for(int i = 0; i<fact_len; i++)
        {
            fprintf(f, "%s\n", result[i]);
            printf("%d: %s\n", i+1, result[i]);
        }
```

```
    fclose(f);

    for(int i = 0; i<fact_len; i++)
    {
        free(result[i]);
    }
    free(result);
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

КОД СКРИПТА

```
#!/bin/bash
gcc ./Source/lab1.c -o lab1.exe
echo -e 'Test1:'
cat ./Tests/test1.txt
echo -e 'Result:'
./lab1.exe < ./Tests/test1.txt
echo -e '\nTest2:'
cat ./Tests/test2.txt
echo -e 'Result:'
./lab1.exe < ./Tests/test2.txt
echo -e '\nTest3:'
cat ./Tests/test3.txt
echo -e 'Result:'
./lab1.exe < ./Tests/test3.txt
echo -e '\nTest4:'
cat ./Tests/test4.txt
echo -e 'Result:'
./lab1.exe < ./Tests/test4.txt
echo -e '\nTest5:'
cat ./Tests/test5.txt
echo -e '\nResult:'
./lab1.exe < ./Tests/test5.txt
echo -e '\nTest6:'
cat ./Tests/test6.txt
echo -e '\nResult:'
./lab1.exe < ./Tests/test6.txt
```

РЕЗУЛЬТАТ РАБОТЫ СКРИПТА

```
Test1:
123
Result:
```

```
Hello! This is permutation maker. Enter your sequance:
Your sequance: 123
1: 123
2: 132
3: 213
4: 231
5: 321
6: 312

Test2:
1
Result:
Hello! This is permutation maker. Enter your sequance:
Your sequance: 1
1: 1

Test3:
ABC
Result:
Hello! This is permutation maker. Enter your sequance:
Your sequance: ABC
1: ABC
2: ACB
3: BAC
4: BCA
5: CBA
6: CAB

Test4:
12.
Result:
Hello! This is permutation maker. Enter your sequance:
Your sequance: 12.
1: 12.
```

```
2: 1.2
3: 21.
4: 2.1
5: .21
6: .12

Test5:
?/_

Result:
Hello! This is permutation maker. Enter your sequance:
Your sequance: ?/_
1: ?/_
2: ?_/
3: /?_
4: /_?
5: _/?
6: _?/

Test6:
 a1

Result:
Hello! This is permutation maker. Enter your sequance:
Your sequance:  a1
1:  a1
2:  1a
3: a 1
4: a1
5: 1a
6: 1 a
```