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 задаётся 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-)=a-b-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.

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).

Описание структур данных и функций.

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 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.

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.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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 bash-скрипта представлен в приложении Г, результат работы скрипта — в приложении В.

Рассмотрим тест 1:

Входные данные:

(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 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
Test6: (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	Your expression is equally 43.13

c -1 d 10 e 13.13	
((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 X 360	You didn't enter value for var 'd', so lets suppose d=1 Your expression is equally 5

Выводы.

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

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ФАЙЛА API.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "API.h"

#define LINE "\x1b[4m"
#define END "\x1b[0m"


int get_vars(char *str){

    char arr[strlen(str)/2];
    int j = 0;
    for(int i = 0; i<strlen(str); i++)
    {
     if(isalpha(str[i]) && !strchr(arr, str[i]))
     {
         arr[j] = str[i];
         j++;
     }
    }

return j;

}


Node*  createNode(char  oper_or_var,  int  is_atom,  int
level){
    Spaces(level);
    printf("createNode: [is_atom = '%d', ", is_atom);
```

```
        Node* tmp = malloc(sizeof(Node));
        tmp->is_atom = is_atom;

        if(isalpha(oper_or_var))
        {
         tmp->var = oper_or_var;
         printf("var = '%c']\n", oper_or_var);
        }
        else
        {
         tmp->operation = oper_or_var;
         printf("operation = '%c']\n", oper_or_var);
        }
        tmp->next = NULL;
        tmp->sublist = NULL;

    return tmp;
    }

    Node* createList(char *str, int level){
        Spaces(level);
          printf("%screateList: level %d%s\n", LINE, level,
END);

        char substr[100];
        strcpy(substr, str);
        Node* tmp;
        if(str[strlen(str)-3] != ')')
        {
         tmp = createNode(str[strlen(str)-2], 1, level);
        }
        else
        {
```

```
          tmp = createNode(str[strlen(str)-2], 0, level);
         }
         Node *head = tmp;
         for(int i = strlen(str)-3; i >= 0; i--)
         {
          if(str[i] == '(')
             {
                 return head;
             }

          if(str[i] == ')')
          {
              if(tmp->is_atom == 1)
              {
               tmp->next   =   createNode(tmp->operation,   0,
level);
               substr[i+1] = '\0';
                       tmp->next->sublist  =  createList(substr,
level+1);
               Spaces(level);
               printf("%screateList:   level   %d%s\n",   LINE,
level, END);
                     tmp=tmp->next;
              }
              else
              {
               if(tmp->sublist != NULL)
               {
                     tmp->next  =  createNode(tmp->operation,  0,
level);
                  substr[i+1] = '\0';
                                          tmp->next->sublist  =
createList(substr, level+1);
                  Spaces(level);
```

```
                    printf("%screateList: level %d%s\n", LINE,
level, END);
                         tmp=tmp->next;
               }
               else
               {
                   substr[i+1] = '\0';
                                                tmp->sublist  =
createList(substr,level+1);
                   Spaces(level);
                    printf("%screateList: level %d%s\n", LINE,
level, END);
               }
              }
              int start = 1; int end = 0; i--;
                 while(start != end)
              {
               if(str[i] == ')')
                   start++;
               if(str[i] == '(')
                   end++;
                  if(start != end)
               {
                   i--;
               }
              }
          }
         if(str[i]!='(' && str[i]!=')'){
          tmp->next = createNode(str[i], 1, level);
          tmp=tmp->next;
          }
         }
     }
```

```
     double to_calculate(Args args[], Node *list, int size,
int level){
         int is_operation = 1;
         double sum = 0;
         double multi = 1;
         double div = 1;
         double sub = 0;
         char current_operation = list->operation;
         Spaces(level);
          printf("%sto_calculate: level %d, current_operation:
'%c'%s\n", LINE, level, current_operation, END);
         while(list != NULL){
             if(current_operation == '+'){
                 if(is_operation == 1)
                 {
                  if(list->is_atom == 0)
                  {
                       sum += to_calculate(args, list->sublist,
size, level+1);
                   Spaces(level);
                          printf("%sto_calculate:   level   %d,
current_operation:  '%c'%s\n", LINE, level, current_operation,
END);
                  }
                  is_operation = 0;
                 }
                 else
              {
                  if(list->is_atom == 0)
                  {
```

```
                    sum += to_calculate(args, list->sublist,
size, level+1);
                Spaces(level);
              printf("%sto_calculate:        level        %d,
current_operation: '%c'%s\n", LINE, level, current_operation,
END);
                }
              else
             {
                      sum += get_value(args, list->var, size,
level);
             }
          }
          list = list->next;
            }
            if(current_operation == '*'){
                if(is_operation == 1)
                {
                    if(list->is_atom == 0)
                    {
                            multi *= to_calculate(args, list-
>sublist, size, level+1);
                        Spaces(level);
                           printf("%sto_calculate: level %d,
current_operation: '%c'%s\n", LINE, level, current_operation,
END);
                    }
                 is_operation = 0;
                }
             else
          {
             if(list->is_atom == 0)
             {
```

```
                            multi *= to_calculate(args, list->sublist, size, level+1);
                Spaces(level);
                printf("%sto_calculate:        level        %d, current_operation: '%c'%s\n", LINE, level, current_operation, END);
                  }
                  else
                  {
                       multi *= get_value(args, list->var, size, level);
                  }
           }
           list = list->next;
          }
          if(current_operation == '/')
          {
           if(is_operation == 1)
           {
               if(list->is_atom == 0)
               {
                      if(list->next == NULL)
                      {
                                div *= to_calculate(args, list->sublist, size, level+1);
                      }
                      else
                      {
                              div *= 1/to_calculate(args, list->sublist, size, level+1);
                      }
                Spaces(level);
```

```
                printf("%sto_calculate:        level        %d,
current_operation:  '%c'%s\n", LINE, level, current_operation,
END);
              }
              is_operation = 0;
                 }
              else
              {
                  if(list->is_atom == 0)
                  {
                      if(list->next == NULL)
                         {
                               div *= to_calculate(args, list-
>sublist, size, level+1);
                         }
                         else
                         {
                             div *= 1/to_calculate(args, list-
>sublist, size, level+1);
                         }
                   Spaces(level);
                          printf("%sto_calculate:   level   %d,
current_operation:  '%c'%s\n", LINE, level, current_operation,
END);
                  }
                  else
                  {
                         if(list->next == NULL)
                         {
                             div *= get_value(args, list->var,
size, level);
                         }
                         else
                         {
```

```
                                div *= 1/get_value(args, list-
>var, size, level);
                            }
                        }
                }
                list = list->next;
               }
          if(current_operation == '-')
          {
                 if(is_operation == 1)
                 {
                     if(list->is_atom == 0)
                     {
                        if(list->next != NULL)
                   {
                               sub  -=  to_calculate(args,  list-
>sublist, size, level+1);
                   }
                   else
                   {
                               sub  +=  to_calculate(args,  list-
>sublist, size, level+1);
                   }
                Spaces(level);
                printf("%sto_calculate:        level        %d,
current_operation:  '%c'%s\n",  LINE,  level,  current_operation,
END);
                     }
                     is_operation = 0;
                }
              else
              {
                     if(list->is_atom == 0)
                     {
```

```
                    if(list->next != NULL)
                    {
                     sub  -=  to_calculate(args,  list->sublist,
size, level+1);
                    }
                    else
                    {
                     sub  +=  to_calculate(args,  list->sublist,
size, level+1);
                    }
                    Spaces(level);
                           printf("%sto_calculate:   level   %d,
current_operation:  '%c'%s\n",  LINE,  level,  current_operation,
END);
                      }
                      else
                      {
                    if(list->next != NULL)
                    {
                     sub  -=  get_value(args,  list->var,  size,
level);
                    }
                    else
                    {
                     sub  +=  get_value(args,  list->var,  size,
level);
                    }
                      }
                 }
                 list = list->next;
                }
          }
          Spaces(level);
          if(current_operation == '/')
```

```
    {
     printf("Result of this level: %lg\n", div);
     return div;
    }
    if(current_operation == '+')
    {
     printf("Result of this level: %lg\n", sum);
     return sum;
    }
    if(current_operation == '*')
    {
     printf("Result of this level: %lg\n", multi);
     return multi;
    }
    if(current_operation == '-')
    {
     printf("Result of this level: %lg\n", sub);
     return sub;
    }
}


void Spaces(int level){
    for(int i = 0; i < level; i++)
    {
        printf("   ");
    }
return;
}

int is_expression_correct(char *str){
    int opening = 0;
    int closing = 0;
    int args = 0;
```

```
        if(str[0] != '(' || str[strlen(str)-1] != ')')
        {
         printf("Brackets are not enough!\n");
         return 0;
        }

        for(int i = 0; i < strlen(str); i++)
        {
         if(isalpha(str[i]))
         {
             args += 1;
         }
         if(str[i] == '(')
         {
             opening += 1;
         }
         if(str[i] == ')')
         {
             closing += 1;
         }
         if(str[i] == ' ')
         {
             printf("Unacceptable symbol: ' '!\n");
             return 0;
         }
         if(str[i] == ')' && !(str[i-1] == '/' || str[i-1] ==
'*' || str[i-1] == '+' || str[i-1] == '-'))
         {
               printf("Operation before the bracket ')' is
missed!\n");
             return 0;
         }
```

```
          if(str[i] != '(' && str[i] != ')' && str[i+1] != ')'
&& !isalpha(str[i]))
          {
                 printf("Arguments must be a letter (a, b,
c,..)!\n");
              return 0;
          }
         }
         if(opening != closing)
         {
          printf("Number of opening and closing brackets must
be same!\n");
          return 0;
         }
         if(args == 0)
         {
          printf("Where are arguments??\n");
          return 0;
         }

     return 1;
     }

     double get_value(Args args[], char var, int size, int
level){

         for(int i = 0; i < size; i++)
         {
          if(args[i].var == var)
          {
              Spaces(level);
               printf("Argument='%c', value=%g\n", args[i].var,
args[i].value);
              return args[i].value;
```

```
         }
        }

         printf("You didn't enter value for var '%c', so lets
suppose %c=1 :)\n", var, var);
        return 1;
    }

    int is_arg_correct(char* str){

        int minos;
        if(str[strlen(str)-1] == '\n')
        {
         str[strlen(str)-1] = '\0';
        }
        if(!isalpha(str[0]))
        {
         printf("Arguments must be a letter!\n");
         return 0;
        }
        if(str[1] != ' ')
        {
         printf("Space between argument and its value!\n");
         return 0;
        }
        if(str[2] == '-')
        {
         minos = 3;
        }
        else
        {
         minos = 2;
        }
        for(int i = minos; i<strlen(str); i++)
```

```
    {
     if(str[i] == '.' && i == minos)
     {
         printf("Values must be a number!\n");
         return 0;
     }
     if(str[i] == '.')
     {
         continue;
     }
     if(!isdigit(str[i]))
     {
         printf("Values must be a number!\n");
         return 0;
     }
    }
return 1;
}
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ГОЛОВНОЙ ПРОГРАММЫ

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "API.h"

int main(){

    double result;
    char expression[100];
    char args_input[15];
    char c;
     printf("Hello! This is some kind of a calculator.\nI can add(+), subtract(-), divide(/) and multiply(*) numbers.\nNote: (a-)=a,  (a+)=a,  (a/)=a,  (a*)=a\nEnter  postfix  expression (something like ""(ab(cd*)+)"" ): ");
    fgets(expression, 98, stdin);
    if(expression[strlen(expression)-1] == '\n')
     expression[strlen(expression)-1] = '\0';

    printf("Your expression: %s\n", expression);

    if(!is_expression_correct(expression))
    {
     return 0;
    }

     printf("Enter arguments and their values, each with a new line (example:a 10): \n");
    int size = get_vars(expression);
    Args args[size];
```

```
    for(int i = 0; i<size; i++)
    {
     fgets(args_input,13,stdin);
     if(!is_arg_correct(args_input))
     {
         return 0;
     }
     args[i].var = args_input[0];
     args[i].value = atof(args_input+2);
    }


    printf("\nForming list...\n\n");
    Node* list = createList(expression, 1);

    printf("\nCalculating...\n\n");
    result = to_calculate(args, list, size, 1);

    printf("\nYour expression is equally %g\n", result);

    FILE *f = fopen("./output.txt", "w+");
    fprintf(f, "Expression: %s\n", expression);
    for(int i = 0; i<size; i++)
    {
     fprintf(f, "%c = %g\n", args[i].var, args[i].value);
    }
    fprintf(f, "Expression is equally %g\n", result);
    fclose(f);
return 0;
}
```

ПРИЛОЖЕНИЕ В
РЕЗУЛЬТАТ ТЕСТИРОВАНИЯ

```
Test1:
(abc+)
a 1
b 2
c 3
Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (abc+)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '+']
   createNode: [is_atom = '1', var = 'c']
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '+'
   Argument='c', value=3
   Argument='b', value=2
   Argument='a', value=1
   Result of this level: 6
```

```
Your expression is equally 6

Test2:
(abc-)
a 6
b 5
c 2.5
Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (abc-)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '-']
   createNode: [is_atom = '1', var = 'c']
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '-'
   Argument='c', value=2.5
   Argument='b', value=5
   Argument='a', value=6
   Result of this level: -1.5

Your expression is equally -1.5
```

```
Test3:
(abc/)
a 10
b 2
c 10
Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (abc/)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '/']
   createNode: [is_atom = '1', var = 'c']
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '/'
   Argument='c', value=10
   Argument='b', value=2
   Argument='a', value=10
   Result of this level: 0.5
```

```
Your expression is equally 0.5

Test4:
(abc*)
a 1
b 2
c 10
Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (abc*)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '*']
   createNode: [is_atom = '1', var = 'c']
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '*'
   Argument='c', value=10
   Argument='b', value=2
   Argument='a', value=1
   Result of this level: 20

Your expression is equally 20
```

```
Test5:
(ab-)
a 5
b 1

Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (ab-)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '-']
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '-'
   Argument='b', value=1
   Argument='a', value=5
   Result of this level: 4

Your expression is equally 4

Test6:
(ba-)
```

```
a 5
b 1

Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (ba-)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '1', operation = '-']
   createNode: [is_atom = '1', var = 'a']
   createNode: [is_atom = '1', var = 'b']

Calculating...

   to_calculate: level 1, current_operation: '-'
   Argument='a', value=5
   Argument='b', value=1
   Result of this level: -4

Your expression is equally -4
Test7:
((abc*)(abc-)(abc/)+)
a 5
b 2
c 1
Result:
```

```
     Hello! This is some kind of a calculator.
     I  can  add(+),  subtract(-),  divide(/)  and  multiply(*)
numbers.
     Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
     Enter  postfix  expression  (something  like  (ab(cd*)+)  ):
Your expression: ((abc*)(abc-)(abc/)+)
     Enter  arguments  and  their  values,  each  with  a  new  line
(example:a 10):

     Forming list...

        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '/']
           createNode: [is_atom = '1', var = 'c']
           createNode: [is_atom = '1', var = 'b']
           createNode: [is_atom = '1', var = 'a']
        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '-']
           createNode: [is_atom = '1', var = 'c']
           createNode: [is_atom = '1', var = 'b']
           createNode: [is_atom = '1', var = 'a']
        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '*']
           createNode: [is_atom = '1', var = 'c']
           createNode: [is_atom = '1', var = 'b']
           createNode: [is_atom = '1', var = 'a']
        createList: level 1
```

```
Calculating...

   to_calculate: level 1, current_operation: '+'
      to_calculate: level 2, current_operation: '/'
      Argument='c', value=1
      Argument='b', value=2
      Argument='a', value=5
      Result of this level: 2.5
   to_calculate: level 1, current_operation: '+'
      to_calculate: level 2, current_operation: '-'
      Argument='c', value=1
      Argument='b', value=2
      Argument='a', value=5
      Result of this level: 2
   to_calculate: level 1, current_operation: '+'
      to_calculate: level 2, current_operation: '*'
      Argument='c', value=1
      Argument='b', value=2
      Argument='a', value=5
      Result of this level: 10
   to_calculate: level 1, current_operation: '+'
   Result of this level: 14.5

Your expression is equally 14.5

Test8:
((a+)(b/)(c*)(d-)+)
a 0.006
b 0.06
c 0.6
d 6
Result:
Hello! This is some kind of a calculator.
```

```
     I  can  add(+),  subtract(-),  divide(/)  and  multiply(*)
numbers.
     Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
     Enter  postfix  expression  (something  like  (ab(cd*)+)  ):
Your expression: ((a+)(b/)(c*)(d-)+)
     Enter  arguments  and  their  values,  each  with  a  new  line
(example:a 10):

     Forming list...

        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '-']
           createNode: [is_atom = '1', var = 'd']
        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '*']
           createNode: [is_atom = '1', var = 'c']
        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '/']
           createNode: [is_atom = '1', var = 'b']
        createList: level 1
        createNode: [is_atom = '0', operation = '+']
           createList: level 2
           createNode: [is_atom = '1', operation = '+']
           createNode: [is_atom = '1', var = 'a']
        createList: level 1

     Calculating...
```

```
      to_calculate: level 1, current_operation: '+'
         to_calculate: level 2, current_operation: '-'
         Argument='d', value=6
         Result of this level: 6
      to_calculate: level 1, current_operation: '+'
         to_calculate: level 2, current_operation: '*'
         Argument='c', value=0.6
         Result of this level: 0.6
      to_calculate: level 1, current_operation: '+'
         to_calculate: level 2, current_operation: '/'
         Argument='b', value=0.06
         Result of this level: 0.06
      to_calculate: level 1, current_operation: '+'
         to_calculate: level 2, current_operation: '+'
         Argument='a', value=0.006
         Result of this level: 0.006
      to_calculate: level 1, current_operation: '+'
      Result of this level: 6.666

   Your expression is equally 6.666

   Test9:
   (ab(c(d-)*)e+)
   a 55
   b -15
   c -1
   d 10
   e 13.13
   Result:
   Hello! This is some kind of a calculator.
   I  can  add(+),  subtract(-),  divide(/)  and  multiply(*)
numbers.
   Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
```

```
     Enter postfix expression (something like (ab(cd*)+) ):
Your expression: (ab(c(d-)*)e+)
     Enter arguments and their values, each with a new line
(example:a 10):

     Forming list...

        createList: level 1
        createNode: [is_atom = '1', operation = '+']
        createNode: [is_atom = '1', var = 'e']
        createNode: [is_atom = '0', var = 'e']
           createList: level 2
           createNode: [is_atom = '0', operation = '*']
              createList: level 3
              createNode: [is_atom = '1', operation = '-']
              createNode: [is_atom = '1', var = 'd']
           createList: level 2
           createNode: [is_atom = '1', var = 'c']
        createList: level 1
        createNode: [is_atom = '1', var = 'b']
        createNode: [is_atom = '1', var = 'a']

     Calculating...

        to_calculate: level 1, current_operation: '+'
        Argument='e', value=13.13
           to_calculate: level 2, current_operation: '*'
              to_calculate: level 3, current_operation: '-'
              Argument='d', value=10
              Result of this level: 10
           to_calculate: level 2, current_operation: '*'
           Argument='c', value=-1
           Result of this level: -10
        to_calculate: level 1, current_operation: '+'
```

```
   Argument='b', value=-15
   Argument='a', value=55
   Result of this level: 43.13

Your expression is equally 43.13

Test10:
((a+)b(bb/)-)
a 100
b 2
Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ):
Your expression: ((a+)b(bb/)-)
Enter arguments and their values, each with a new line (example:a 10):

Forming list...

   createList: level 1
   createNode: [is_atom = '0', operation = '-']
      createList: level 2
      createNode: [is_atom = '1', operation = '/']
      createNode: [is_atom = '1', var = 'b']
      createNode: [is_atom = '1', var = 'b']
   createList: level 1
   createNode: [is_atom = '1', var = 'b']
   createNode: [is_atom = '0', var = 'b']
      createList: level 2
      createNode: [is_atom = '1', operation = '+']
      createNode: [is_atom = '1', var = 'a']
```

```
   createList: level 1

Calculating...

   to_calculate: level 1, current_operation: '-'
      to_calculate: level 2, current_operation: '/'
      Argument='b', value=2
      Argument='b', value=2
      Result of this level: 1
   to_calculate: level 1, current_operation: '-'
   Argument='b', value=2
      to_calculate: level 2, current_operation: '+'
      Argument='a', value=100
      Result of this level: 100
   to_calculate: level 1, current_operation: '-'
   Result of this level: 97

Your expression is equally 97

Test11:
(a(a(a+)+))
a 27

Result:
Hello! This is some kind of a calculator.
I can add(+), subtract(-), divide(/) and multiply(*) numbers.
Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a
Enter postfix expression (something like (ab(cd*)+) ): Your expression: (a(a(a+)+))
Operation before the bracket ')' is missed!

Test12:
(abcd/)
```

a 10

b 20

ccccccccccc

d 30

Result:

Hello! This is some kind of a calculator.

I can add(+), subtract(-), divide(/) and multiply(*) numbers.

Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a

Enter postfix expression (something like (ab(cd*)+)): Your expression: (abcd/)

Enter arguments and their values, each with a new line (example:a 10):

Space between argument and its value!

Test13:

(a(b(c(d-)+)/)*)

a 10

b 2

c 3

X 360

Result:

Hello! This is some kind of a calculator.

I can add(+), subtract(-), divide(/) and multiply(*) numbers.

Note: (a-)=a, (a+)=a, (a/)=a, (a*)=a

Enter postfix expression (something like (ab(cd*)+)): Your expression: (a(b(c(d-)+)/)*)

Enter arguments and their values, each with a new line (example:a 10):

```
Forming list...

   createList: level 1
   createNode: [is_atom = '0', operation = '*']
      createList: level 2
      createNode: [is_atom = '0', operation = '/']
         createList: level 3
         createNode: [is_atom = '0', operation = '+']
            createList: level 4
            createNode: [is_atom = '1', operation = '-']
            createNode: [is_atom = '1', var = 'd']
         createList: level 3
         createNode: [is_atom = '1', var = 'c']
      createList: level 2
      createNode: [is_atom = '1', var = 'b']
   createList: level 1
   createNode: [is_atom = '1', var = 'a']

Calculating...

   to_calculate: level 1, current_operation: '*'
      to_calculate: level 2, current_operation: '/'
         to_calculate: level 3, current_operation: '+'
            to_calculate: level 4, current_operation: '-'
You didn't enter value for var 'd', so lets suppose
d=1 :)
            Result of this level: 1
         to_calculate: level 3, current_operation: '+'
         Argument='c', value=3
         Result of this level: 4
      to_calculate: level 2, current_operation: '/'
      Argument='b', value=2
      Result of this level: 0.5
   to_calculate: level 1, current_operation: '*'
```

```
   Argument='a', value=10
   Result of this level: 5

Your expression is equally 5
```

ПРИЛОЖЕНИЕ Г

КОД BASH-СКРИПТА

```
#!/bin/bash
make
echo -e 'Test1:'
cat ./Tests/test1.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test1.txt
echo -e '\nTest2:'
cat ./Tests/test2.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test2.txt
echo -e '\nTest3:'
cat ./Tests/test3.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test3.txt
echo -e '\nTest4:'
cat ./Tests/test4.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test4.txt
echo -e '\nTest5:'
cat ./Tests/test5.txt
echo -e '\nResult:'
./lab2.exe < ./Tests/test5.txt
echo -e '\nTest6:'
cat ./Tests/test6.txt
echo -e '\nResult:'
./lab2.exe < ./Tests/test6.txt
echo -e 'Test7:'
cat ./Tests/test7.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test7.txt
```

```
echo -e '\nTest8:'
cat ./Tests/test8.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test8.txt
echo -e '\nTest9:'
cat ./Tests/test9.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test9.txt
echo -e '\nTest10:'
cat ./Tests/test10.txt
echo -e 'Result:'
./lab2.exe < ./Tests/test10.txt
echo -e '\nTest11:'
cat ./Tests/test11.txt
echo -e '\nResult:'
./lab2.exe < ./Tests/test11.txt
echo -e '\nTest12:'
cat ./Tests/test12.txt
echo -e '\nResult:'
./lab2.exe < ./Tests/test12.txt
echo -e '\nTest13:'
cat ./Tests/test13.txt
echo -e '\nResult:'
./lab2.exe < ./Tests/test13.txt
make -f Makefile clean
```