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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсия

Студентка гр. 9303 _____ Зарезина Е.А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Ознакомление с основными понятиями и приёмами рекурсивного программирования, освоение навыков программирования рекурсивных процедур и функций на языке программирования C++.

Основные теоретические положения.

Функция является рекурсивной, если оператор в теле функции вызывает функцию, содержащую данный оператор. Иногда называемая круговым определением, рекурсия является процессом определения чего-либо с использованием самой себя.

Простым примером является функция factr(), вычисляющая факториал целого числа. Факториал числа N является произведением чисел от 1 до N. Например, факториал 3 равен 1*2*3 или 6. Как factr(), так и его итеративный эквивалент показаны ниже:

```
/* Вычисление факториала числа */

int factr(int n) /* рекурсивно */
{
int answer;
if(n==1) return(1);
answer = factr(n-1)*n;
return(answer);
}
/* Вычисление факториала числа */
int fact (int n)    /* нерекурсивно */
{
int t, answer;
answer == 1;
for(t=1; t<=n; t++)
answer=answer*(t);
```

```
return(answer);
}
```

Нерекурсивная версия fact() использует цикл, начиная с 1 и заканчивая указанным числом, последовательно перемножая каждое число на ранее полученное произведение.

Рекурсивная функция factr() вызывается с аргументом 1, возвращает 1. В противном случае она возвращает произведение factr(n- 1) * n. Для вычисления этого значения factr() вызывается с n-1. Это происходит, пока n не станет равно 1.

При вычислении факториала числа 2, первый вызов factr() приводит ко второму вызову с аргументом 1. Данный вызов возвращает 1, после чего результат умножается на 2 (исходное значение n). Ответ, таким образом, будет 2. Можно попробовать вставить printf() в factr() для демонстрации уровней и промежуточных ответов каждого вызова.

Когда функция вызывает сама себя, в стеке выделяется место для новых локальных переменных и параметров. Код функции работает с данными переменными. Рекурсивный вызов не создает новую копию функции. Новыми являются только аргументы. Поскольку каждая рекурсивно вызванная функция завершает работу, то старые локальные переменные и параметры удаляются из стека и выполнение продолжается с точки, в которой было обращение внутри этой же функции. Рекурсивные функции вкладываются одна в другую как элементы подзорной трубы.

Рекурсивные версии большинства подпрограмм могут выполняться немного медленнее, чем их итеративные эквиваленты, поскольку к необходимым действиям добавляются вызовы функций. Но в большинстве случаев это не имеет значения. Много рекурсивных вызовов в функции может привести к переполнению стека. Поскольку местом для хранения параметров и локальных переменных функции является стек и каждый новый вызов создает новую копию переменных, пространство стека может исчерпаться. Если это произойдет, то возникнет ошибка - переполнение стека.

При написании рекурсивных функций следует иметь оператор if, чтобы заставить функцию вернуться без рекурсивного вызова. Если это не сделать, то, однажды вызвав функцию, выйти из нее будет невозможно. Это наиболее типичная ошибка, связанная с написанием рекурсивных функций. Надо использовать при разработке функции printf() и getchar(), чтобы можно было узнать, что происходит, и прекратить выполнение в случае обнаружения ошибки.

Постановка задачи.

Вариант 9

Разработать программу, которая по заданному *простому_логическому* выражению, не содержащему вхождений простых идентификаторов, вычисляет значение этого выражения.

простое_логическое::=TRUE | FALSE |

NOT простое_логическое |

(простое_логическое знак_операции простое_логическое)

знак-операции::= AND | OR

Реализация.

Функция main:

В функции *main* ведется диалог с пользователем. Пользователю предлагается ввести строку через консоль или ввести имя файла, в котором находится строка, которую он хочет проверить с помощью синтаксического анализатора.

В функции *simpleExpr* производится проверка, является ли полученное выражение логическим.

В функции *resSimpleExpr* вычисляется значение логического выражения.

Тестирование.

1. **Ввод:**

   1

   (TRUE OR FALSE)

   Вывод:

   Input line: (TRUE OR FALSE)

   Checking for first simple expression

   Found first simple expression

   Checking for OR or AND

   Found OR or AND

   Checking for second simple expression

   Found second simple expression

   Checking for last bracket

   Found last bracket

   It's right expression!

   Result: TRUE

2. **Ввод:**

   2

   test1.txt

   Вывод:

   InputLine from file: ((NOT FALSE AND TRUE) OR (TRUE OR TRUE))

   Input line: ((NOT FALSE AND TRUE) OR (TRUE OR TRUE))

   Checking for first simple expression

   Checking for first simple expression

   Checking for simple expression

   Found a simple expression

   Found first simple expression

   Checking for OR or AND

   Found OR or AND

Checking for second simple expression

Found second simple expression

Checking for last bracket

Found last bracket

Found first simple expression

Checking for OR or AND

Found OR or AND

Checking for second simple expression

Checking for first simple expression

Found first simple expression

Checking for OR or AND

Found OR or AND

Checking for second simple expression

Found second simple expression

Checking for last bracket

Found last bracket

Found second simple expression

Checking for last bracket

Found last bracket

It's right expression!

Result: TRUE

Вывод.

В ходе выполнения лабораторной работы были изучены основные приёмы и понятия рекурсивного программирования. Была разработана программа, являющая синтаксическим анализатором для логического выражения.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>

using namespace std;

enum Recursion{
	NEXT_LEVEL,
	STOP_REC,
};
enum BoolExpression{
	BOOL_TRUE,
	BOOL_FALSE,
	BOOL_NOT,
	BOOL_AND,
	BOOL_OR,
	OTHER,
};
void createTabs (int recLevel){
	for(int i = 0; i<recLevel; i++){
		cout<<"\t";
	}
}
```

```
BoolExpression resSimplExpr(stringstream& myExpr, string atom){
        if(atom == "TRUE"){
                return BOOL_TRUE;
        } else if(atom == "FALSE"){
                return BOOL_FALSE;
        } else if(atom == "NOT"){
                myExpr>>atom;
                BoolExpression callRec = resSimplExpr(myExpr,atom);
                if(callRec == BoolExpression::BOOL_TRUE){
                        return BOOL_FALSE;
                } else if(callRec == BoolExpression::BOOL_FALSE){
                        return BOOL_TRUE;
                }
        } else if(atom == "("){
                BoolExpression flag;
                myExpr>>atom;
                BoolExpression bool1 = resSimplExpr(myExpr,atom);
                myExpr>>atom;
                if(atom == "OR"){
                        flag = BoolExpression::BOOL_OR;
                } else if (atom == "AND") {
                        flag = BoolExpression::BOOL_AND;
                }
                myExpr>>atom;
                BoolExpression bool2 = resSimplExpr(myExpr,atom);
                myExpr>>atom;
                if(flag == BoolExpression::BOOL_OR){
                        if((bool1 == BoolExpression::BOOL_FALSE )&&(bool2
== BoolExpression::BOOL_FALSE)){
                                return BoolExpression::BOOL_FALSE;
```

```
                } else {
                        return BoolExpression::BOOL_TRUE;
                }
            } else if (flag == BoolExpression::BOOL_AND){
                if((bool1 == BoolExpression::BOOL_TRUE )&&(bool2 ==
BoolExpression::BOOL_TRUE)){
                        return BoolExpression::BOOL_TRUE;
                } else {
                        return BoolExpression::BOOL_FALSE;
                }
            }
        }
         return BoolExpression::OTHER;
    }


    Recursion simplExpr(stringstream& myExpr, string atom, int recLevel){
        recLevel++;
        createTabs(recLevel);
        if((atom == "TRUE")||(atom == "FALSE")){
            return NEXT_LEVEL;
        } else if(atom == "NOT"){
            myExpr>>atom;
            createTabs(recLevel);
            cout<<"Checking for simple expression"<<endl;
            Recursion callRec = simplExpr(myExpr,atom, recLevel);
            if(callRec == Recursion::STOP_REC){
                cout<<"Something went wrong!"<<endl;
                return STOP_REC;
            } else {
```

```
                createTabs(recLevel);
                cout<<"Found a simple expression"<<endl;
                return NEXT_LEVEL;
            }

        } else if(atom == "("){
            cout<<"Checking for first simple expression"<<endl;
            myExpr>>atom;
            Recursion callRec = simplExpr(myExpr,atom,recLevel);
            if(callRec == Recursion::NEXT_LEVEL){
                createTabs(recLevel);
                cout<<"Found first simple expression"<<endl;
                createTabs(recLevel);
                cout<<"Checking for OR or AND"<<endl;
                myExpr>>atom;
                if((atom == "OR")||(atom == "AND")){
                    createTabs(recLevel);
                    cout<<"Found OR or AND"<<endl;
                    createTabs(recLevel);
                    cout<<"Checking for second simple expression"<<endl;
                    myExpr>>atom;
                    Recursion callRec = simplExpr(myExpr,atom,recLevel);
                    if(callRec == Recursion::NEXT_LEVEL){
                        createTabs(recLevel);
                        cout<<"Found second simple expression"<<endl;
                        createTabs(recLevel);
                        cout<<"Checking for last bracket"<<endl;
```

```
                        myExpr>>atom;
                        if(atom==")"){
                                createTabs(recLevel);
                                cout<<"Found last bracket"<<endl;
                                return NEXT_LEVEL;
                        } else {
                                createTabs(recLevel);
                                cerr<<"NO last bracket!\n";
                                exit(1);
                        }
                } else {
                        createTabs(recLevel);
                        cerr<<"NO second simple expression!\n";
                        exit(1);
                }
        } else {
                createTabs(recLevel);
                cerr<<"NO OR or AND!\n";
                exit(1);
        }
    } else {
        createTabs(recLevel);
        cerr<<"NO first simple expression!\n";
        exit(1);
    }
} else {
    cerr<<"Wrong expression!";
    exit(1);
}
}
```

```
int main(){
    int recLevel = -1;
    stringstream myExpr;
    stringstream myExpr2;
    string atom;
    string inputLine;
    string inputLineCopy;
    cout<<"Select the text input method:\n\t1 - console\n\t2 - text from file\
nYour choice: "<<endl;
    char way;
    string c;
    cin>>way;
    cout<<"your choice: "<<way<<endl;
    getline(cin,c);
    if(way == '1'){
        cout<<"Enter an expression: "<<endl;
        getline(cin,inputLine);
    } else if(way == '2'){
        string fileName;
        cout<<"Enter a file name: "<<endl;
        getline(cin, fileName);
        ifstream fin (fileName);
        if(fin){
            getline(fin, inputLine);
            cout<<"InputLine from file: "<<inputLine<<endl;
        } else {
            cerr<<"File couldn't be open!\n";
            exit(1);
        }
```

```
        }
        inputLineCopy = inputLine.substr();
        cout<<"Input line: "<<inputLine<<endl;
        myExpr<<inputLine;
        myExpr>>atom;
        Recursion callRec = simplExpr(myExpr,atom,recLevel);
        if(callRec == Recursion::NEXT_LEVEL){
                cout<<"It's right expression!"<<endl;
                myExpr2<<inputLineCopy;
                myExpr2>>atom;
                BoolExpression boolRec = resSimplExpr(myExpr2, atom);
                if(boolRec == BoolExpression::BOOL_TRUE){
                        cout<<"Result: TRUE"<<endl;
                } else if (boolRec == BoolExpression::BOOL_FALSE){
                        cout<<"Result: FALSE"<<endl;
                }
        } else {
                cout<<"It's not an expression!"<<endl;
        }

        return 0 ;
}
```