Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Тверской государственный университет»

Факультет прикладной математики и кибернетики

Направление: 01.03.02 Прикладная математика и информатика

ОТЧЕТ ПО УЧЕБНО-ВЫЧИСЛИТЕЛЬНОЙ ПРАКТИКЕ

Выполнила:

студентка 2 курса

24 группы

Савинова Валентина

Константиновна

Тверь 2016

Оглавление

Грамматика для языка **3**

Транслирующая грамматика для языка **5**

Описание работы программы для генерации кода **7**

Описание работы интерпретатора **9**

Пример **11**

Исходный код программы **13**

Грамматика для языка

1. $< S >\rightarrow< P >< M > END$
2. $< P >\rightarrow< L >< P >$
3. $< P >\rightarrow e$
4. $< L >\rightarrow< M >< O >$
5. $< M >\rightarrow const$
6. $< O >\rightarrow$ LET VAR $=< E >$
7. $< O >\rightarrow GOTO < M >$
8. $< O >\rightarrow IF < T >< M >$
9. $< O >\rightarrow$ FOR VAR $=< E > TO < E > STEP < E >< P >< M > NEXT$
10. $< O >\rightarrow GOSUB < M >$
11. $< O >\rightarrow RETURN$
12. $< O >\rightarrow$ INPUT VAR
13. $< O >\rightarrow PRINT < E >$
14. $< T >\rightarrow< E >< T' >$
15. $< T' >\rightarrow=< E >$
16. $< T' >\rightarrow<< E >$

17. $< T' > \rightarrow >< E >$

18. $< E > \rightarrow < K >< I >< E' >$

19. $< E' > \rightarrow + < K >< I >< E' >$

20. $< E' > \rightarrow - < K >< I >< E' >$

21. $< E' > \rightarrow e$

22. $< I > \rightarrow e$

23. $< I > \rightarrow * < K >< I >$

24. $< I > \rightarrow / < K >< I >$

25. $< K > \rightarrow (< K >< I >< E' >)$

26. $< K > \rightarrow const$

27. $< K > \rightarrow VAR$

Транслирующая грамматика для языка

1. $< S > \rightarrow < P >< M > END$
2. $< P > \rightarrow < L >< P >$
3. $< P > \rightarrow e$
4. $< L > \rightarrow < M >< O >$
5. $< M > \rightarrow const[lprev]$
6. $< M1 > \rightarrow const$
7. $< O > \rightarrow$ LET VAR$[MARK_VAR] =< E > [> VAR]$
8. $< O > \rightarrow GOTO < M1 > [jmp\ lprev]$
9. $< O > \rightarrow IF < T >< M1 > [ij\ lprev]$
10. $< O > \rightarrow$

    FOR VAR $[MARK\_VAR]$ $[MARK\_VAR]$
    $[MARK\_VAR]$ $[MARK\_VAR] =< E >\ [> VAR]\ TO\ < E >\ [> tmp1]$
    $STEP\ < E >\ [> tmp2]\ [l0]\ [VAR]\ [tmp1]\ [<]\ [ij\ l1]\ [jmp\ l2]\ [l1]$
    $< P >\ [tmp2]\ [VAR]\ [+]\ [> VAR]\ [jmp\ l0]\ < M >\ NEXT\ [l2]$

11. $< O > \rightarrow GOSUB[sub] < M1 > [jmp\ lprev]$
12. $< O > \rightarrow RETURN[ret]$

13. $< O > \rightarrow INPUT\ VAR\ [MARK\_VAR]\ [?]\ [> VAR]$

14. $< O > \rightarrow PRINT < E > [!]$

15. $< T > \rightarrow < E >< T' >$

16. $< T' > \rightarrow = < E > [=]$

17. $< T' > \rightarrow < < E > [<]$

18. $< T' > \rightarrow > < E > [>]$

19. $< E > \rightarrow < K >< I >< E' >$

20. $< E' > \rightarrow + < K >< I > [+] < E' >$

21. $< E' > \rightarrow - < K >< I > [-] < E' >$

22. $< E' > \rightarrow e$

23. $< I > \rightarrow e$

24. $< I > \rightarrow * < K > [*] < I >$

25. $< I > \rightarrow / < K > [/] < I >$

26. $< K > \rightarrow (< K >< I >< E' >)$

27. $< K > \rightarrow const\ [c]$

28. $< K > \rightarrow VAR\ [MARK\_VAR]\ [VAR]$

Описание работы программы для синтаксического анализа и генерации кода

Перед проведением синтаксического анализа упорядочиваем строки программы по номеру метки по возрастанию. Программа моделирует работу автомата с магазинной памятью. В начальный момент времени в магазине автомата находится начальный нетерминал S, а считывающая головка обозревает первый символ входа, далее автомат работает по правилам:

1. Если на вершине стека находится терминальный символ, то проверяем какой символ обозревает считывающая головка. Если символ на стеке совпадает со входным символом, то удаляем символ со стека и сдвигаемся на входной ленте. Иначе генерируем ошибку.

2. Если на стеке находится символ «команда» (символы в грамматике стоящие в квадратных скобках), то убираем символ со стека и пишем его на выходной ленте. При этом на входной ленте не сдвигаемся.

3. Если на стеке лежит нетерминал, то по двум символам входной ленты выбираем правило из грамматики. Удаляем нетерминал со стека и кладем на него символы правой части выбранного правила в обратном порядке. На входной ленте не сдвигаемся.

4. Автомат принимает входное слово тогда и только тогда, когда в ходе ра-

боты не генерируются ошибки и после прочтения слова стек оказывается пустым.

При условии, что автомат принимает входное слово, результатом работы автомата является последовательность команд промежуточного вида: в ней присутствуют команды-метки, переходы по меткам (условные и безусловные) и имена переменных. Наша дальнейшая задача заключается в том, чтобы заменить имена переменных на номера ячеек памяти, удалить метки, а переходы по именованным меткам заменить переходами по номерам команд.

Первую задачу решаем следующим образом. Заводим счетчик номеров ячеек, пока список команд содержит имена переменных выполняем следующее. Находим первое по порядку и заменяем его на номер ячейки, согласно значению счетчика. Затем, ищем эту же переменную в списке команд и заменяем каждое вхождение этой переменной на номер соответствующей ячейки (отдельно обрабатываем случаи считывания ячейки и записи в нее). После увеличиваем значение счетчика на единицу и возвращаемся к поиску имен переменных.

Обработка имен меток происходит следующим образом. Пока список команд содержит необработанные именованные метки, выполняем следующее. Находим номер команды перед которой стоит метка (подсчитываем команды не являющиеся метками стоящие перед ней), затем ищем переходы по той метке и заменяем их переходами по номеру команды (как условные, так и безусловные), после чего помечаем метку как обработанную и возвращаемся к поиску меток. Когда необработанных меток не останется, удаляем все метки. Далее походим по списку команд и ищем переходы по именованным меткам. Если такие переходы имеются, то заменяем их на соответствующие переходы к номеру команды равному числу команд. Таким образом организуется выход и программы.

Описание работы интерпретатора

Интерпретатор представляет собой «машину» с двумя стеками (для вычислений и возвратов) и вектором ячеек памяти. В каждый момент времени также хранится номер текущей команды. Опишем команды, подаваемые на вход интерпретатору. Всякий раз, говоря о стеке, будем иметь в виду стек для вычислений, если иное не указано явно.

1. [!] – взять число со стека и распечатать его.
2. [?] – запросить с консоли число и положить его на стек.
3. $[Xi]$ – положить число из i-ой ячейки памяти на стек.
4. $[> Xi]$ – положить число с вершины стека в i-ую ячейки памяти.
5. $[c]$ –положить на стек константу c.
6. $[+]$ – взять со стека 2 числа, положить на стек их сумму.
7. $[-]$ – взять со стека 2 числа, положить на стек их разность.
8. $[*]$ – взять со стека 2 числа, положить на стек их произведение.
9. $[/]$ – взять со стека 2 числа, положить на стек их частное.
10. $[=]$ – взять со стека 2 числа, положить на стек 1 тогда и только тогда, когда они равны и 0 в противном случае.
11. $[<]$ – взять со стека 2 числа, положить на стек 1 тогда и только тогда, когда первое число меньше второго и 0 в противном случае.

12. $[>]$ – взять со стека 2 числа, положить на стек 1 тогда и только тогда, когда первое число больше второго и 0 в противном случае.

13. $[ij\ k]$ – взять со стека число перейти к команде с номером k, если оно не равно 0.

14. $[jmp\ k]$ – перейти к команде с номером k.

15. $[sub]$ – положить на стек возвратов номер текущей команды $+1$.

16. $[ret]$ – взять со стека возвратов номер команды и перейти к команде с номером большим на единицу.

Работа интерпретатора происходит следующим образом. На каждом шаге берется команда с текущим номером и выполняется согласно указанному выше. Во всех случаях кроме 13,14 и 16 увеличиваем номер текущей команды на единицу. Работа завершается, когда номер текущей команды становится больше количества команд.

Пример

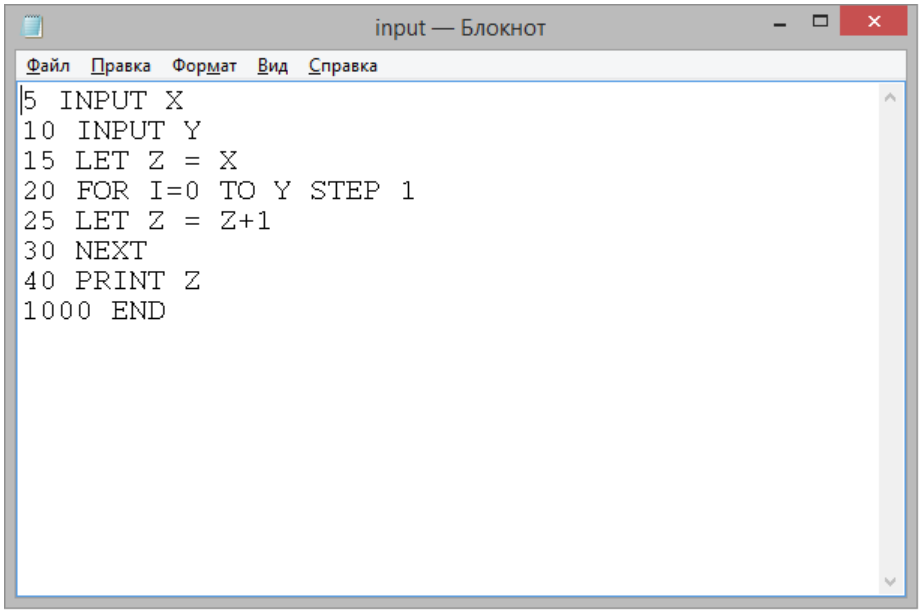

Рис. 1: Исходный код программы на MINI-BASIC

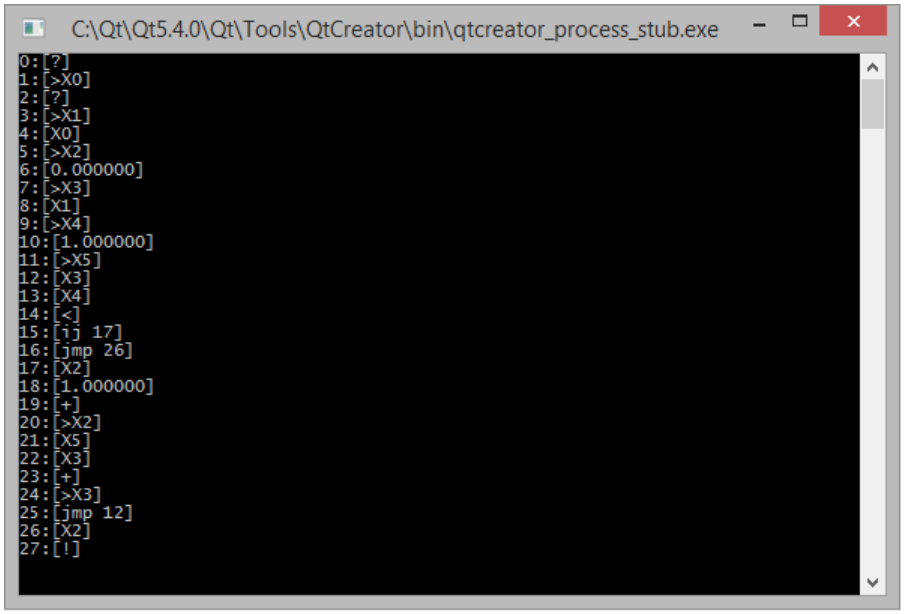

Рис. 2: Сгенерированный код по данной программе

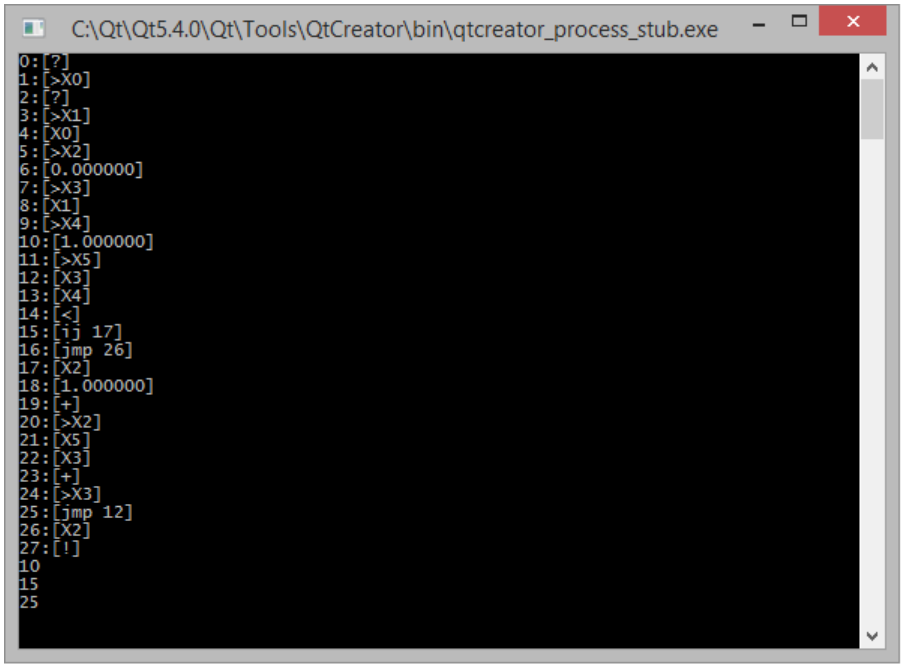

Рис. 3: Результат работы программы

Исходный код программы

```
#ifndef FAUTOMATA_H
#define FAUTOMATA_H
#include <vector>
#include <lexeme.h>
#include <iostream>
using namespace std;
class FAutomata
{
    vector<Lexeme> result;
    Lexeme currentLexeme;
    istream& in;
    bool (FAutomata::*current)();
    bool q0();
    bool q1();
    bool q2();
    bool q3();
    bool q4();
    bool q5();
    bool q6();
    bool q7();

public:
    FAutomata(istream& in);
    ~FAutomata();
    void start();
    vector<Lexeme> getResult(){return result;}
};

#endif // FAUTOMATA_H
```

```
#include "fautomata.h"
#include <math.h>

FAutomata::FAutomata(istream& i):in(i), current(&q0)
{
}

FAutomata::~FAutomata()
{

}
vector<string> KWord;

bool isSpecial(char c)
{
    string templ = "+-*/^()<>=";
    string target;
    target+= c;
    return templ.find_first_of(target)!=-1;
}

void FAutomata::start()
{
    KWord.clear();
```

```
    KWord.push_back("LET");
    KWord.push_back("TO");
    KWord.push_back("GOTO");
    KWord.push_back("FOR");
    KWord.push_back("STEP");
    KWord.push_back("IF");
    KWord.push_back("GOSUB");
    KWord.push_back("NEXT");
    KWord.push_back("RETURN");
    KWord.push_back("END");
    KWord.push_back("PRINT");
    KWord.push_back("INPUT");
    bool cnt = true;
    while(cnt)
    {
        cnt = (this->*current)();
    }


}

bool FAutomata::q0()
{
    char c = in.get();

    if (c == '\n')
    {
        result.push_back(Lexeme(EOL));
    }
    if(isspace(c))
    {
        return true;
    }
    if (c == '+')
    {
        result.push_back(Lexeme(OP_PLUS));
        return true;
    }
    if (c == '-')
    {
        result.push_back(Lexeme(OP_MINUS));
        return true;
    }
    if (c == '*')
    {
        result.push_back(Lexeme(OP_MUL));
        return true;
    }
    if (c == '/')
    {
        result.push_back(Lexeme(OP_DIV));
        return true;
    }
    if (c == '^')
    {
        result.push_back(Lexeme(OP_EXP));
        return true;
    }
    if (c == '(')
    {
        result.push_back(Lexeme(L_BRACKET));
        return true;
    }
    if (c == ')')
    {
        result.push_back(Lexeme(R_BRACKET));
        return true;
    }
    if (c == '<')
```

```
    {
        result.push_back(Lexeme(L_T));
        return true;
    }
    if (c == '>')
    {
        result.push_back(Lexeme(G_T));
        return true;
    }
    if (c == '=')
    {
        result.push_back(Lexeme(E_Q));
        return true;
    }
    if (isdigit(c))
    {
        currentLexeme.setType(NUMBER);
        currentLexeme.setValue(c-'0');
        current = &q1;
        return true;
    }
    if (isalpha(c))
    {
        c = toupper(c);
        string temp = "GLTFSINREP";
        string s;
        s+=c;
        if (-1==temp.find_first_of(s))
        {
            currentLexeme.setType(VAR);
            currentLexeme.setName(s);
            current = &q4;
            return true;
        }
        else
        {
            currentLexeme.setName(s);
            current = &q5;
            return true;
        }
    }
    if (c == '.')
    {
        currentLexeme.setType(NUMBER);
        currentLexeme.setValue(0);
        current = &q2;
        return true;
    }
    if(c == EOF)
    {
        return false;
    }
    throw "Unexpected input";


}
bool FAutomata::q1()
{
    char c = in.get();
    float temp = currentLexeme.getValue();
    if (isdigit(c))
    {
        currentLexeme.setValue(temp*10+c-'0');
        return true;
    }
    if (c == '.')
    {
        current = &q2;
        return true;
```

```
    }
    c=toupper(c);
    if (c == 'E')
    {
        current = &q3;
        return true;
    }
    if(c == EOF)
    {
        if(currentLexeme.getType()!=UNDEFINED)
        {
            result.push_back(currentLexeme);
        }
        return false;
    }
    else
    {
        result.push_back(currentLexeme);
        in.putback(c);
        current = &q0;
        currentLexeme = Lexeme();
        return true;
    }

}
bool FAutomata::q2()
{
    char c = in.get();
    float temp = currentLexeme.getValue();
    static int pos = 0;
    if (isdigit(c))
    {
        pos++;
        float m = c-'0';
        for(int i=0; i<pos;++i)
        {
            m/=10;
        }
        currentLexeme.setValue(temp+m);
        return true;
    }
    c=toupper(c);
    if (c == 'E')
    {
        current = &q3;
        pos = 0;
        return true;
    }
    if(c == EOF)
    {
        if(currentLexeme.getType()!=UNDEFINED)
        {
            result.push_back(currentLexeme);
        }
        return false;
    }
    else
    {
        pos = 0;
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        in.putback(c);
        current = &q0;
        return true;
    }


}
bool FAutomata::q3()
```

```
{
    char c = in.get();
    static int pw = 0;
    static int sgn = 1;
    if (c == '-')
    {
        sgn*=-1;
        return true;
    }
    if (isdigit(c))
    {
        pw=pw*10+c-'0';
        return true;
    }
    if(c ==EOF)
    {
        if(currentLexeme.getType()!=UNDEFINED)
        {
            pw=pw*sgn;
            float temp = currentLexeme.getValue();
            temp = temp * pow(10,pw);
            currentLexeme.setValue(temp);
            result.push_back(currentLexeme);
        }
        return false;
    }
    else
    {
        pw=pw*sgn;
        float temp = currentLexeme.getValue();
        temp = temp * pow(10,pw);
        pw = 0;
        sgn = 1;
        currentLexeme.setValue(temp);
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        in.putback(c);
        current = &q0;
        return true;
    }
}
bool FAutomata::q4()
{
    char c = in.get();
    if (isdigit(c))
    {
        currentLexeme.setName(currentLexeme.getName()+c);
        return true;
    }
    if (c == '\n')
    {
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        result.push_back(Lexeme(EOL));
        return true;
    }
    if (isspace(c))
    {
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        return true;
    }
    if (isSpecial(c))
    {
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
```

```
        in.putback(c);
        return true;
    }
    if (c==EOF)
    {
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        return false;
    }
    throw "Unexpected input";
}

bool FAutomata::q5()
{
    char c = in.get();
    if (isdigit(c))
    {
        currentLexeme.setName(currentLexeme.getName()+c);
        currentLexeme.setType(VAR);
        current = &q4;
        return true;
    }
    if (isSpecial(c))
    {
        currentLexeme.setType(VAR);
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        in.putback(c);
        return true;
    }
    if (c == '\n')
    {
        currentLexeme.setType(VAR);
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        result.push_back(Lexeme(EOL));
        return true;
    }
    if(isspace(c))
    {
        currentLexeme.setType(VAR);
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        current = &q0;
        return true;
    }
    if (c==EOF)
    {
        currentLexeme.setType(VAR);
        result.push_back(currentLexeme);
        currentLexeme = Lexeme();
        return false;
    }
    if (isalpha(c))
    {
        c = toupper(c);
        string temp = currentLexeme.getName()+c;
        currentLexeme.setName(temp);
        current = &q6;
        return true;
    }
    throw "Unexpected input";
}

bool FAutomata:: q6()
{
```

```
    char c = in.get();
    c = toupper(c);
    string temp = currentLexeme.getName()+c;
    if (isalpha(c))
    {
        currentLexeme.setName(temp);
        return true;
    }
    else
    {
        for(int i=0; i<KWord.size();++i)
        {
            if(KWord[i] == currentLexeme.getName())
            {
                currentLexeme.setType(LexemeType(LET+i));
                result.push_back(currentLexeme);
                currentLexeme = Lexeme();
                current = &q0;
                return true;
            }
        }
        if (currentLexeme.getName()=="REM")
        {
            currentLexeme = Lexeme();
            current = &q7;
            return true;
        }
    }
    throw "Unexpected input";
}
bool FAutomata::q7()
{
    char c =in.get();
    if (c==EOF)
    {
        return false;
    }
    if (c=='\n')
    {
        result.push_back(Lexeme(EOL));
        current = &q0;
    }
    return true;
}
```

```
#ifndef LEXEME_H
#define LEXEME_H
#include <string>

using namespace std;
enum LexemeType
{
    OP_PLUS,
    OP_MINUS,
    OP_MUL,
    OP_DIV,
    OP_EXP,
    L_BRACKET,
    R_BRACKET,
    E_Q,
    L_T,
    G_T,
    NUMBER,
    LET,
    TO,
    GOTO,
    FOR,
    STEP,
    IF,
```

```
    GOSUB,
    NEXT,
    RETURN,
    END,
    PRINT,
    INPUT,
    VAR,
    EOL,
    LABEL,
    LABEL_USE,
    UNDEFINED
};


class Lexeme
{
    LexemeType type;
    float value;
    string name;
public:
    Lexeme();
    Lexeme(LexemeType t);
    Lexeme(float value);
    ~Lexeme();
    LexemeType getType() const;
    void setType(LexemeType);
    float getValue() const;
    void setValue(float);
    string getName();
    void setName(string);
    bool operator == (const Lexeme& other) const;

};

#endif // LEXEME_H
```

```
#include "lexeme.h"

Lexeme::Lexeme(LexemeType t):type(t),name("")
{
}
Lexeme::Lexeme(float v):type(NUMBER),value(v),name("")
{
}
Lexeme::Lexeme():type(UNDEFINED),name("")
{
}
Lexeme::~Lexeme()
{

}
LexemeType Lexeme::getType() const
{
    return type;
}
float Lexeme::getValue() const
{
    return value;
}
void Lexeme::setType(LexemeType t)
{
    type = t;
}
void Lexeme::setValue(float v)
{
    value = v;
}
bool Lexeme:: operator == (const Lexeme& other) const
{
```

```
    return (this->type == other.type);
}
string Lexeme::getName()
{
    return this->name;
}

void Lexeme::setName(string s)
{
    this->name = s;
}
```

```
#ifndef LEXEMENAMESSINGLETON_H
#define LEXEMENAMESSINGLETON_H
#include <vector>
#include<iostream>
using namespace std;

class LexemeNamesSingleton
{
private :
    static LexemeNamesSingleton* instance;
    vector<string> lexemeTypeNames;
    LexemeNamesSingleton();
public:
    static LexemeNamesSingleton* getInstance()
    {
        if(!instance)
            instance = new LexemeNamesSingleton();
        return instance;
    }
    vector<string> getLexemeNames()
    {
        return lexemeTypeNames;
    }
};

#endif // LEXEMENAMESSINGLETON_H
```

```
#include "lexemenamessingleton.h"

LexemeNamesSingleton* LexemeNamesSingleton::instance = NULL;

LexemeNamesSingleton::LexemeNamesSingleton()
{
    lexemeTypeNames.push_back("OP_PLUS");
    lexemeTypeNames.push_back("OP_MINUS");
    lexemeTypeNames.push_back("OP_MUL");
    lexemeTypeNames.push_back("OP_DIV");
    lexemeTypeNames.push_back("OP_EXP");
    lexemeTypeNames.push_back("L_BRACKET");
    lexemeTypeNames.push_back("R_BRACKET");
    lexemeTypeNames.push_back("E_Q");
    lexemeTypeNames.push_back("L_T");
    lexemeTypeNames.push_back("G_T");
    lexemeTypeNames.push_back("NUMBER");
    lexemeTypeNames.push_back("LET");
    lexemeTypeNames.push_back("TO");
    lexemeTypeNames.push_back("GOTO");
    lexemeTypeNames.push_back("FOR");
    lexemeTypeNames.push_back("STEP");
    lexemeTypeNames.push_back("IF");
    lexemeTypeNames.push_back("GOSUB");
    lexemeTypeNames.push_back("NEXT");
    lexemeTypeNames.push_back("RETURN");
    lexemeTypeNames.push_back("END");
    lexemeTypeNames.push_back("PRINT");
    lexemeTypeNames.push_back("INPUT");
```

```
    lexemeTypeNames.push_back("VAR");
    lexemeTypeNames.push_back("EOL");
    lexemeTypeNames.push_back("LABEL");
    lexemeTypeNames.push_back("LABEL_USE");
    lexemeTypeNames.push_back("UNDEFINED");
}
```

```
#ifndef SAUTOMATA_H
#define SAUTOMATA_H
#include <iostream>
#include<lexeme.h>
#include <stack>
#include <vector>
#include "lexemenamessingleton.h"
using namespace std;
struct Symbol
{
    string name;
    Lexeme lexeme;
    Symbol(string k)
    {
        name = k;
        lexeme = Lexeme(UNDEFINED);
    }
    Symbol (Lexeme l)
    {
        name = l.getName();
        lexeme = l;
    }

    Symbol (string k, LexemeType l)
    {
        name = k;
        lexeme = Lexeme(l);
    }

    Symbol (LexemeType type)
    {
        name = "UNDEFINED";
        lexeme = Lexeme(type);
    }
    bool isTerminal () const
    {
        return lexeme.getType()!=UNDEFINED;
    }
    bool operator == (const Symbol& other) const
    {
        if(other.isTerminal()&&this->isTerminal())
        {
            return this->
    lexeme.getType()==other.lexeme.getType();
        }
        else
        {
            if (!other.isTerminal()&&!this->isTerminal())
            {
                return this->name==other.name;
            }
            else
                if (other.isSpecial()&&this->isSpecial())
                {
                    return this->name==other.name;
                }
                return false;
        }

    }
    bool isSpecial () const
    {
```

```
        if(this->name.size()==0)
            return false;
        return this->name.at(0) == '[';
    }

};

class sautomata
{
    stack<Symbol> st;
    int currentIndex;
public:
    sautomata();
    vector<Symbol> Start(vector<Lexeme>);
};

#endif // SAUTOMATA_H
```

```
#include "sautomata.h"
sautomata::sautomata()
{
    currentIndex = 0;
}
vector<Symbol> sautomata::Start(vector<Lexeme> input)
{
    vector<Symbol> res;
    stack<string> vars;
    int tempVarCounter = 0;
    int tempLabelCounter = 0;
    st.push(Symbol("S"));
    while (true)
    {
        if (st.size()==0)
        {
            if (currentIndex==input.size())
                return res;
            else
            {
                vector<string> lexemeNames =
    LexemeNamesSingleton::getInstance()->
          getLexemeNames();
                throw "Expected end of input but found "
    +lexemeNames[input[currentIndex].getType()];
            }
        }
        if (currentIndex>=input.size())
        {
            throw "Unexpected end of input";
        }
        Symbol S = st.top();
        if (S.isSpecial())
        {
            if(S.name == "[MARK_VAR]")
            {
                if(currentIndex>0)
                {
                    Lexeme prev = input[currentIndex-1];
                    if(prev.getType() == VAR)
                    {
                        vars.push(prev.getName());
                        st.pop();
                        continue;
                    }
                    else
                    {
                        throw "Expected variable, but found "
      +LexemeNamesSingleton::getInstance()->
      getLexemeNames()[prev.getType()];
                    }
```

```
            }
            else
            {
                throw "Expected variable, but found none";
            }
        }
        else if(S.name == "[VAR]")
        {
            S.name = "["+vars.top()+"]";
            res.push_back(S);
            st.pop();
            vars.pop();
            continue;
        }
        else if(S.name == "[>VAR]")
        {
            S.name = "[>"+vars.top()+"]";
            res.push_back(S);
            st.pop();
            vars.pop();
            continue;
        }
        else if(S.name == "[!prev]"
||S.name == "[ij !prev]"||S.name=="[jmp !prev]")
        {
            if(currentIndex>0)
            {
                Lexeme prev = input[currentIndex-1];
                if(prev.getType() == NUMBER)
                {
                    string prefix ="[";
                    if(S.name == "[ij !prev]")
                    {
                        prefix = "[ij ";
                    }
                    else if(S.name == "[jmp !prev]")
                    {
                        prefix = "[jmp ";
                    }
                    S.name = prefix+"!_"+
  std::to_string((int)prev.getValue())+"]";
                    res.push_back(S);
                    st.pop();
                    continue;
                }
                else
                {
                    throw "Expected lable, but found "
  +LexemeNamesSingleton::getInstance()->
  getLexemeNames()[prev.getType()];
                }
            }
            else
            {
                throw "Expected lable, but found none";
            }
        }
        else if (S.name == "[c]")
        {
            if(currentIndex>0)
            {
                Lexeme prev = input[currentIndex-1];
                if(prev.getType() == NUMBER)
                {
                    S.name = "["+
  std::to_string(prev.getValue())+"]";
                    res.push_back(S);
                    st.pop();
                    continue;
```

```
                }
                else
                {
                    throw "Expected constant, but found "
  +LexemeNamesSingleton::getInstance()->
  getLexemeNames()[prev.getType()];
                }
            }
            else
            {
                throw "Expected constant, but found none";
            }

        }
        else
        {
            res.push_back(S);
            st.pop();
            continue;
        }
    }
    if (S.isTerminal())
    {
        if (Symbol(input[currentIndex])==S)
        {
            st.pop();
            currentIndex++;
        }
        else
        {
            vector<string> lexemeNames =
LexemeNamesSingleton::getInstance()->
      getLexemeNames();
            throw string("Expected ") +
lexemeNames[S.lexeme.getType()]+
" but found "+
lexemeNames[input[currentIndex].getType()];
        }
    }
    else
    {
        st.pop();
        if (S==Symbol("S"))
        {
            st.push(Symbol(Lexeme(END)));
            st.push(Symbol("M"));
            st.push(Symbol("P"));
        }
        else if(S==Symbol("P"))
        {
            if(currentIndex+1<input.size()
              &&input[currentIndex+1].getType()!=END
              &&input[currentIndex+1].getType()!=NEXT)
            {
                st.push(Symbol("P"));
                st.push(Symbol("L"));
            }
        }
        else if(S==Symbol("L"))
        {
            st.push(Symbol("O"));
            st.push(Symbol("M"));
        }
        else if(S==Symbol("M"))
        {
            st.push(Symbol("[!prev]",LABEL));
            st.push(Symbol(Lexeme(NUMBER)));
        }
        else if(S==Symbol("M1"))
```

```
        {
            st.push(Symbol(Lexeme(NUMBER)));
        }
        else if(S==Symbol("O"))
        {
            LexemeType x = input[currentIndex].getType();
            switch (x) {
            case LET:
                st.push(Symbol("[>VAR]",VAR));
                st.push(Symbol("E"));
                st.push(Symbol(Lexeme(E_Q)));
                st.push(Symbol("[MARK_VAR]"));
                st.push(VAR);
                st.push(LET);
                break;
            case GOTO:
                st.push(Symbol("[jmp !prev]",
      LABEL_USE));
                st.push(Symbol("M1"));
                st.push(GOTO);
                break;
            case IF:
                st.push(Symbol("[ij !prev]",
      LABEL_USE));
                st.push(Symbol("M1"));
                st.push(Symbol("T"));
                st.push(IF);
                break;
            case FOR:
            st.push(Symbol("[!_t"+
to_string(tempLabelCounter+2)+"]",
      LABEL));
            st.push(NEXT);
            st.push(Symbol("M"));
            st.push(Symbol("[jmp !_t"+
to_string(tempLabelCounter)+"]",
      LABEL_USE));
            st.push(Symbol("[>VAR]",VAR));
            st.push(Symbol("[+]"));
            st.push(Symbol("[VAR]",VAR));
            st.push(Symbol("[tmp"+
to_string(tempVarCounter+1)+"]",VAR));
            st.push(Symbol("P"));
            st.push(Symbol("[!_t"+
to_string(tempLabelCounter+1)+"]",
       LABEL));
            st.push(Symbol("[jmp !_t"+
to_string(tempLabelCounter+2)+"]",
      LABEL_USE));
            st.push(Symbol("[ij !_t"+
to_string(tempLabelCounter+1)+"]",
      LABEL_USE));
            st.push(Symbol("[<]"));
            st.push(Symbol("[tmp"+
to_string(tempVarCounter)+"]",VAR));
            st.push(Symbol("[VAR]",VAR));
            st.push(Symbol("[!_t"+
to_string(tempLabelCounter)+"]",
      LABEL));
            st.push(Symbol("[>tmp"+
to_string(tempVarCounter+1)+"]",VAR));
            st.push(Symbol("E"));
            st.push(STEP);
            st.push(Symbol("[>tmp"+
to_string(tempVarCounter)+"]",VAR));
            st.push(Symbol("E"));
            st.push(TO);
            st.push(Symbol("[>VAR]",VAR));
            st.push(Symbol("E"));
```

```
            st.push(E_Q);
            st.push(Symbol("[MARK_VAR]"));
            st.push(Symbol("[MARK_VAR]"));
            st.push(Symbol("[MARK_VAR]"));
            st.push(Symbol("[MARK_VAR]"));
            st.push(VAR);
            st.push(FOR);
            tempVarCounter+=2;
            tempLabelCounter+=3;
              break;
            case GOSUB:
                st.push(Symbol("[jmp !prev]",LABEL_USE));
                st.push(Symbol("M1"));
                st.push(Symbol("[sub]"));
                st.push(GOSUB);
                break;
            case RETURN:
                st.push(Symbol("[ret]"));
                st.push(RETURN);
                break;
            case INPUT:
                st.push(Symbol("[>VAR]",VAR));
                st.push(Symbol("[?]"));
                st.push(Symbol("[MARK_VAR]"));
                st.push(VAR);
                st.push(INPUT);
                break;
            case PRINT:
                st.push(Symbol("[!]"));
                st.push(Symbol("E"));
                st.push(PRINT);
                break;
            default:
                throw "Unexpected start of operation";
            }
        }
        else if(S==Symbol("T"))
        {
            st.push(Symbol("T1"));
            st.push(Symbol("E"));
        }
        else if(S==Symbol("T1"))
        {
            LexemeType x = input[currentIndex].getType();
            switch (x) {
            case E_Q:
                st.push(Symbol("[=]"));
                st.push(Symbol("E"));
                st.push(E_Q);
                break;
            case L_T:
                st.push(Symbol("[<]"));
                st.push(Symbol("E"));
                st.push(L_T);
                break;
            case G_T:
                st.push(Symbol("[>]"));
                st.push(Symbol("E"));
                st.push(G_T);
                break;
            default:
                throw "Unexpected test symbol";
            }
        }
        else if(S==Symbol("E"))
        {
            st.push(Symbol("E*"));
            st.push(Symbol("|"));
            st.push(Symbol("K"));
```

```
            }
            else if(S==Symbol("E*"))
            {
                LexemeType x = input[currentIndex].getType();
                if( x == OP_PLUS)
                {
                    st.push(Symbol("E*"));
                    st.push(Symbol("[+]"));
                    st.push(Symbol("|"));
                    st.push(Symbol("K"));
                    st.push(OP_PLUS);
                }
                else if(x == OP_MINUS)
                {
                    st.push(Symbol("E*"));
                    st.push(Symbol("[-]"));
                    st.push(Symbol("|"));
                    st.push(Symbol("K"));
                    st.push(OP_MINUS);
                }
            }
            else if(S==Symbol("|"))
            {
                LexemeType x = input[currentIndex].getType();
                if( x == OP_MUL)
                {
                    st.push(Symbol("|"));
                    st.push(Symbol("[*]"));
                    st.push(Symbol("K"));
                    st.push(OP_MUL);
                }
                else if(x == OP_DIV)
                {
                    st.push(Symbol("|"));
                    st.push(Symbol("[/]"));
                    st.push(Symbol("K"));
                    st.push(OP_DIV);
                }
            }
            else if(S==Symbol("K"))
            {
                LexemeType x = input[currentIndex].getType();
                if( x == L_BRACKET)
                {
                    st.push(R_BRACKET);
                    st.push(Symbol("E*"));
                    st.push(Symbol("|"));
                    st.push(Symbol("K"));
                    st.push(L_BRACKET);
                }
                else if(x == NUMBER)
                {
                    st.push(Symbol("[c]"));
                    st.push(NUMBER);

                }
                else if(x == VAR)
                {
                    st.push(Symbol("[VAR]",VAR));
                    st.push(Symbol("[MARK_VAR]"));
                    st.push(VAR);
                }
                else throw "Unexpected start of expression";
            }
        }
    }
}
```

```
#include <iostream>
```

```
#include <fstream>
#include <fautomata.h>
#include <sautomata.h>
#include <algorithm>
#include<stackmachine.h>

using namespace std;

bool operator < (const vector<Lexeme>& v, const vector<Lexeme>& v1)
{
    if (v.size()==0||v1.size()==0)
        throw "Unexpected input";
    if ((*v.begin()).getType() != NUMBER ||(*v1.begin()).getType() != NUMBER)
        throw "Unexpected input";
    return (*v.begin()).getValue()<(*v1.begin()).getValue();
}
string getVarName (string name)
{
    name = name.substr(1, name.length()-2);
    if(name.at(0) == '>')
    {
        name = name.substr(1,name.length()-1);
    }
    return name;
}
string getLabel (string label)
{
    if (label.at(1)=='l')
    {
        return label.substr(3,label.length()-4);
    }
    if (label.at(1) == 'j')
    {
        return label.substr(7, label.length()-8);
    }
    if (label.at(1) == 'i')
    {
        return label.substr(6, label.length()-7);
    }
    return label;
}

int main()
{


    ifstream in("c:/input.txt",std::ifstream::in);
    FAutomata a(in);
    try
    {
        a.start();
        vector<Lexeme> res = a.getResult();
        vector<Lexeme> copy;
        for(vector<Lexeme>::iterator i = res.begin();i!=res.end();++i)

        {
            if ((*i).getType()==OP_MINUS)
            {
                if ((i == res.begin())||
                        ((*(i-1)).getType()!=R_BRACKET && (*(i-1)).getType()!= NUMBER))
                {
                    if((i+1) == res.end())
                    {
                        copy.push_back(*i);
                    }
                    else if ((*(i+1)).getType()== NUMBER)
                    {
                        (*(i+1)).setValue(-(*(i+1)).getValue());
                    }
```

```
            else
                copy.push_back(*i);
        }
        else
            copy.push_back(*i);
    }
    else
        copy.push_back(*i);
}
vector<vector<Lexeme>> lines;
vector<Lexeme> line;
for (vector<Lexeme>::iterator i = copy.begin();i!=copy.end();++i)
{
    if ((*i).getType()==EOL)
    {
        lines.push_back(line);
        line = vector<Lexeme>();
    }
    else
    {
        line.push_back((*i));
    }
}
if (line.size()!=0)
{
    lines.push_back(line);
}
sort(lines.begin(),lines.end());
res = vector<Lexeme>();
for (vector<vector<Lexeme>>::iterator i = lines.begin();i!=lines.end();i++)
{
    vector<Lexeme> v = (*i);
    for (vector<Lexeme>::iterator j = v.begin(); j!=v.end(); j++)
    {
        res.push_back((*j));
    }
}
sautomata sa;
vector<Symbol> out = sa.Start(res);
int currentIndex = 0;
bool exit = false;
while (!exit)
{
    exit = true;
    for (vector<Symbol>::iterator i = out.begin(); i!=out.end();++i)
    {
        if ((*i).lexeme.getType()==VAR)
        {
            exit = false;
            string name = (*i).name;
            if (name.at(1)=='>')
            {
                (*i).name = "[>X"+to_string(currentIndex)+"]";
            }
            else
            {
                (*i).name = "[X"+ to_string(currentIndex)+"]";
            }
            (*i).lexeme.setType(UNDEFINED);
            name = getVarName(name);
            for (vector<Symbol>::iterator j = i+1; j!= out.end();j++)
            {
                if ((*j).lexeme.getType()==VAR)
                {
                    string temp = (*j).name;
                    temp = getVarName(temp);
                    if (temp == name)
                    {
                        if ((*j).name.at(1)=='>')
```

```
                        {
                            (*j).name = "[>X"+to_string(currentIndex)+"]";
                        }
                        else
                        {
                            (*j).name = "[X"+ to_string(currentIndex)+"]";
                        }
                        (*j).lexeme.setType(UNDEFINED);
                    }
                }
            }
            currentIndex = currentIndex+1;
            break;
        }
    }
}
exit = false;
while (!exit)
{
    exit = true;
    int counter = 0;
    for (vector<Symbol>::iterator i = out.begin(); i!=out.end();++i)
    {
        if ((*i).lexeme.getType()==LABEL)
        {
            exit = false;
            string label = (*i).name;
            label = getLabel(label);
            for (vector<Symbol>::iterator j = out.begin(); j!= out.end();j++)
            {
                if ((*j).lexeme.getType()== LABEL_USE)
                {
                    string temp = (*j).name;
                    temp = getLabel(temp);
                    if(temp == label)
                    {
                        if((*j).name[1] == 'j')
                        {
                            (*j).name = "[jmp "+to_string(counter)+"]";
                            (*j).lexeme.setType(UNDEFINED);
                        }
                        else if((*j).name[1] == 'i')
                        {
                            (*j).name = "[ij "+to_string(counter)+"]";
                            (*j).lexeme.setType(UNDEFINED);
                        }
                        else
                        {
                            throw "Unexpected command: "+(*j).name;
                        }
                    }
                }
            }
        }
        else
        {
            counter++;
        }
    }
    break;
}
vector<Symbol>::iterator i = out.begin();
while(i!=out.end())
{
    if((*i).lexeme.getType() == LABEL)
    {
        out.erase(i);
        i = out.begin();
    }
```

```
            else
            {
                i++;
            }
        }
        for (vector<Symbol>::iterator i = out.begin(); i!=out.end();++i)
        {
            if((*i).lexeme.getType()==LABEL_USE)
            {
                if((*i).name[1] == 'j')
                {
                    (*i).name = "[jmp "+to_string(out.size())+"]";
                    (*i).lexeme.setType(UNDEFINED);
                }
                else if((*i).name[1] == 'i')
                {
                    (*i).name = "[ij "+to_string(out.size())+"]";
                    (*i).lexeme.setType(UNDEFINED);
                }
                else
                {
                    throw "Unexpected command: "+(*i).name;
                }
            }
        }
        int num = 0;
        for(vector<Symbol>::iterator i = out.begin();i!=out.end();++i)
        {

            cout<<num<<":"<<(*i).name<<endl;
            num++;
        }
        StackMachine st;
        st.start(out);

    }
    catch (const char*e)
    {
        cout<<"Error:"<<e<<endl;
    }
    catch(std::string str)
    {
        cout<<str<<endl;
    }

    return 0;
}
```

```
#ifndef STACKMACHINE_H
#define STACKMACHINE_H
#include<stack>
#include<vector>
#include <iostream>
#include<string>
#include<math.h>
#include<sautomata.h>
using namespace std;

class StackMachine
{
    stack<double> operationStack;
    stack<long> callStack;
    vector<double> memory;
    long currentPosition;
    void addCells();
    double getStackValue();
public:
    StackMachine();
    void start(vector<Symbol> commands);
```

```
};

#endif // STACKMACHINE_H
```

```
#include "stackmachine.h"
#include <algorithm>

StackMachine::StackMachine():currentPosition(0)
{

}

bool eq(double x1, double x2)
{
    return fabs(x1-x2)<pow(10,-6);
}


void StackMachine::start(vector<Symbol> commands)
{
    while(currentPosition<commands.size())
    {
        string command = commands[currentPosition].name;
        command = command.substr(1,command.length()-2);
        if(command.empty())
            throw "Illegal empty command";
        if(command.length()>=3&&command[0]=='>'&&command[1]=='X')//[>xi]
        {
            string indexString = command.substr(2,command.length()-2);
            int index = stoi(indexString);
            while(index>=memory.size())
            {
                addCells();
            }
            memory[index] = getStackValue();
            currentPosition++;
            continue;
        }
        if(command.length()>=2&&command[0] == 'X')
        {
            string indexString = command.substr(1,command.length()-1);
            int index = stoi(indexString);
            while(index>=memory.size())
            {
                addCells();
            }
            operationStack.push(memory[index]);
            currentPosition++;
            continue;
        }
        if(command=="!")
        {
            cout<<getStackValue()<<endl;
            currentPosition++;
            continue;
        }
        if(command=="?")
        {
            double x;
            cin>>x;
            operationStack.push(x);
            currentPosition++;
            continue;
        }
        if(command=="+")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2+x1);
```

```
            currentPosition++;
            continue;
        }
        if(command=="-")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2-x1);
            currentPosition++;
            continue;
        }
        if(command=="*")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2*x1);
            currentPosition++;
            continue;
        }
        if(command=="/")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2/x1);
            currentPosition++;
            continue;
        }
        if(command=="=")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(eq(x2,x1)?1:0);
            currentPosition++;
            continue;
        }
        if(command=="<")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2<x1?1:0);
            currentPosition++;
            continue;
        }
        if(command==">")
        {
            double x1 = getStackValue();
            double x2 = getStackValue();
            operationStack.push(x2>x1?1:0);
            currentPosition++;
            continue;
        }
        if(command.length()>=5&&command[0]=='j'&&command[1]=='m'&&command[2]=='p')
        {
            string indexString = command.substr(4,command.length()-4);
            int index = stoi(indexString);
            currentPosition = index;
            continue;
        }
        if(command.length()>=4&&command[0]=='i'&&command[1]=='j')
        {
            string indexString = command.substr(3,command.length()-3);
            int index = stoi(indexString);
            if(getStackValue())
            {
                currentPosition = index;
            }
            else
            {
                currentPosition++;
```

```
            }
            continue;
        }
        if(command=="sub")
        {
            callStack.push(currentPosition+1);
            currentPosition++;
            continue;
        }
        if(command=="ret")
        {
            if(callStack.empty())
            {
                throw "Call stack is empty";
            }
            currentPosition = callStack.top()+1;
            callStack.pop();
            continue;
        }
        {
            double constant = stod(command);
            operationStack.push(constant);
            currentPosition++;
            continue;
        }


    }
}

double StackMachine::getStackValue()
{
    if(operationStack.empty())
        throw "Operation stack is empty";
    double x = operationStack.top();
    operationStack.pop();
    return x;
}

void StackMachine::addCells()
{
    for(int i=0;i<100;++i)
    {
        memory.push_back(0);
    }
}
```