МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра автоматизированных систем управления

Изображение выглядит как текст, устройство

Автоматически созданное описание

ОТЧЁТ

по КУРСОВОЙ РАБОТЕ

**«***Константы с плавающей точкой языка С++***»**

по дисциплине: **«***Теория формальных языков и компиляторов***»**

Выполнил:Проверил:

Студент гр. *АВТ-912*, *АВТФ* *д.т.н., профессор*

*Насибуллин Р.Р.*

*Шорников Юрий Владимирович*

«\_\_\_» \_\_\_\_\_\_ 20\_\_г.«\_\_\_» \_\_\_\_\_\_ 20\_\_ г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск 2022

**РЕФЕРАТ**

Отчет 25 с., 6 рис., 4 источн..

ЯЗЫКОВОЙ ПРОЦЕССОР, КОМПИЛЯТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, АВТОМАТНАЯ ГРАММАТИКА, ГРАФ АВТОМАТНОЙ ГРАММАТИКИ, ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК

Цель работы – выполнить программную реализацию алгоритма синтаксического анализатора объявления переменных с плавающей точкой языка программирования С++.

В результате проектирования был написан синтаксический анализатор (парсер) для объявления переменных с плавающей точкой языка программирования С++.

**ОГЛАВЛЕНИЕ**

[**ПОСТАНОВКА ЗАДАЧИ** 4](#_Toc104244304)

[**ГРАММАТИКА ЯЗЫКА** 5](#_Toc104244305)

[**КЛАССИФИКАЦИЯ ГРАММАТИКИ** 6](#_Toc104244306)

[**МЕТОД АНАЛИЗА** 7](#_Toc104244307)

[**ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК** 8](#_Toc104244308)

[**ТЕСТИРОВАНИЕ** 9](#_Toc104244309)

[**ЗАКЛЮЧЕНИЕ** 11](#_Toc104244310)

[**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ** 12](#_Toc104244311)

[**ЛИСТИНГ АЛГОРИТМА СИНТАКСИЧЕСКОГО АНАЛИЗА** 13](#_Toc104244312)

# **ПОСТАНОВКА ЗАДАЧИ**

Цель работы – Константы с плавающей точкой языка С++.

Порядок выполнения работы:

1. Разбор порождающей грамматики
2. Определить тип грамматики по классификации Хомского
3. Выбор метода анализа
4. Реализация и тестирование

При описании грамматики необходимо определить ее по классификации Хомского как автоматную или контекстно-свободную. Классификация грамматики влияет на выбор и реализацию метода синтаксического анализа: для автоматной – конечный автомат, для контекстно-свободной – метод рекурсивного спуска.

Для программной реализации был использован язык C# и .NET Framework. Данный выбор обусловлен тем, что .NET Framework позволяет создавать приложения для операционной системы Windows, одной из самых распространённых операционных систем. Для запуска приложения требуется операционная система Windows 7, 8, 10 с установленным .NET Framework актуальной версии.

# **ГРАММАТИКА ЯЗЫКА**

G[Z] = {VT, VN, Z, P}

VT = { double, float, 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,g,h,I,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,·, ;}

VN = {<ТИП>, <ПЕРЕМЕННАЯ>, <ЗНАЧЕНИЕ>}.

P = {

1. Z→<ТИП><переменная>’=’<ЗначЕНИЕ>’;’

2) <ТИП>→ (double | float)

3) <ПЕРЕМЕННАЯ>→Б{Б}

4) <ЗНАЧЕНИЕ>→ Ц{Ц}’.’ Ц{Ц}

5) Б → ‘a’ | ‘b’ | ‘c’... |’z’ | ‘A’ | ‘B’ | ‘C’... |’Z’

6) Ц → 0 |1 |… |9

}

# **КЛАССИФИКАЦИЯ ГРАММАТИКИ**

Грамматика G[Z] по классификации Хомского относится к классу автоматных грамматик, так как все правила вывода относятся к виду:

A→ aB | a | ε.

В левой части допускаются только нетерминальные символы, а в правой части могут присутствовать как символы с терминального словаря, так и символы с нетерминального.

# **МЕТОД АНАЛИЗА**

Исходя из того, что грамматика является автоматной, анализ будет осуществляться при помощи графа конечного автомата.

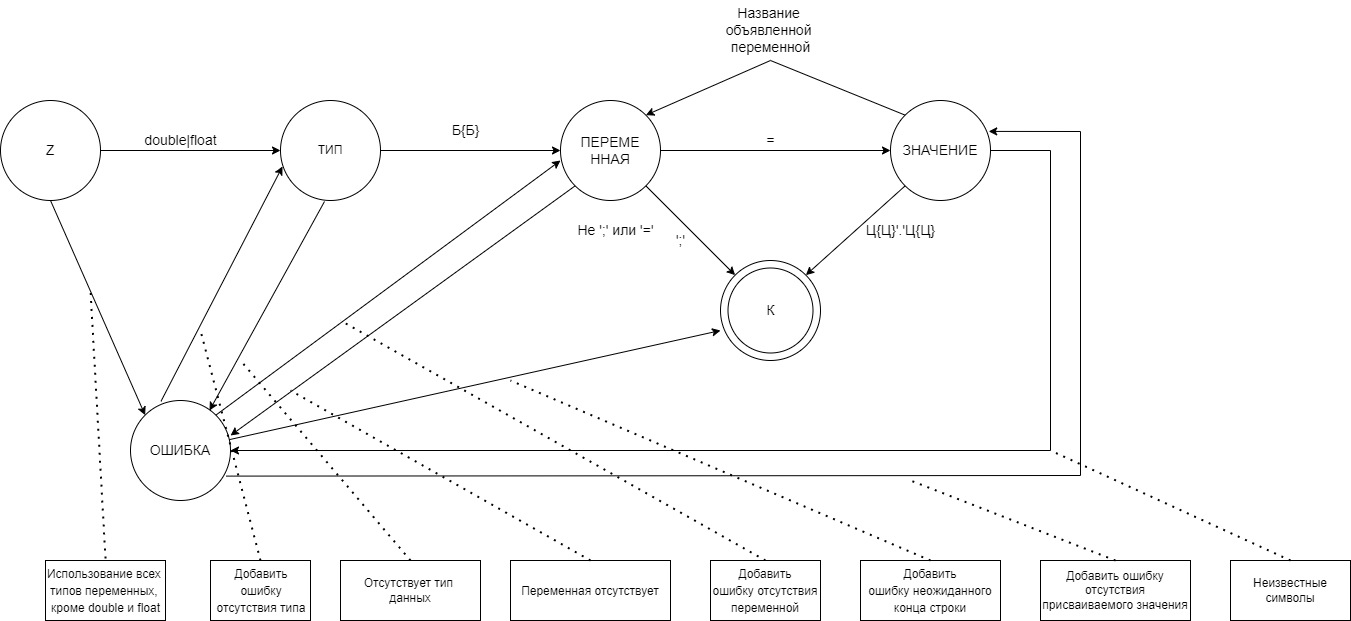


Рис. 1. Граф конечного автомата.

# **ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК**

Согласно заданию на курсовую работу, необходимо реализовать нейтрализацию синтаксических ошибок, используя метод Айронса.

Суть метода Айронса заключается в следующем:

При обнаружении ошибки (во входной цепочке в процессе разбора встречается символ, который не соответствует ни одному из ожидаемых символов), входная цепочка символов выглядит следующим образом: Tt, где T – следующий символ во входном потоке (ошибочный символ), t – оставшаяся во входном потоке цепочка символов после T. Алгоритм нейтрализации состоит из следующих шагов:

1. Определяются недостроенные кусты дерева разбора;

2. Формируется множество L – множество остаточных символов недостроенных кустов дерева разбора;

3. Из входной цепочки удаляется следующий символ до тех пор, пока цепочка не примет вид Tt, такой, что U => T, где U ∈ L, то есть до тех пор, пока следующий в цепочке символ T не сможет быть выведен из какого-нибудь из остаточных символов недостроенных кустов.

4. Определяется, какой из недостроенных кустов стал причиной появления символа U в множестве L (иначе говоря, частью какого из недостроенных кустов является символ U).

Таким образом, определяется, к какому кусту в дереве разбора можно «привязать» оставшуюся входную цепочку символов после удаления из текста ошибочного фрагмента.

Для автоматной грамматики предлагается свести алгоритм нейтрализации к последовательному удалению следующего символа во входной цепочке до тех пор, пока следующий символ не окажется одним из допустимых в данный момент разбора.

# **ТЕСТИРОВАНИЕ**

На рисунках 2 – 5 представлены тестовые примеры запуска разработанного синтаксического анализатора объявления переменных с плавающей точкой языка программирования С++

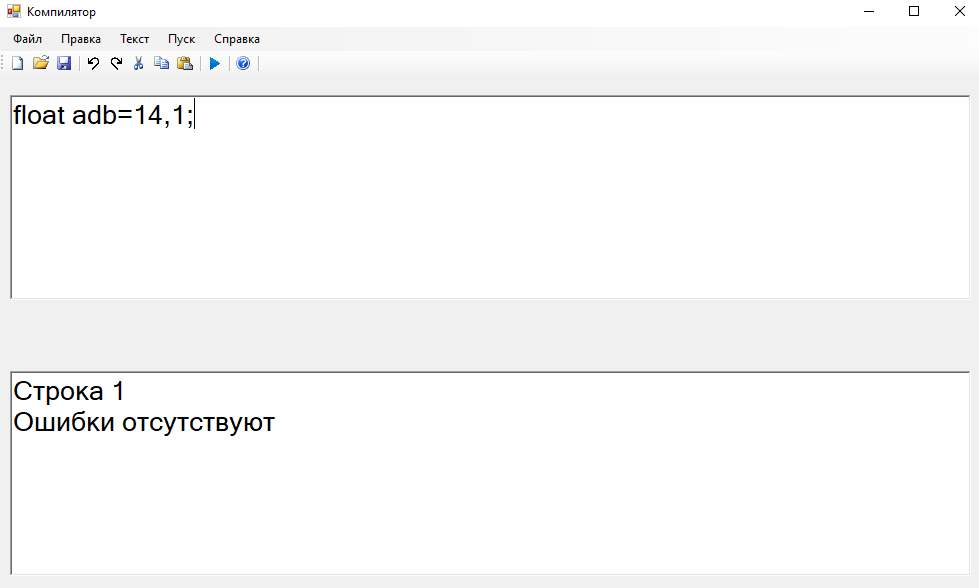


Рис 2. Пример ввода без ошибок.

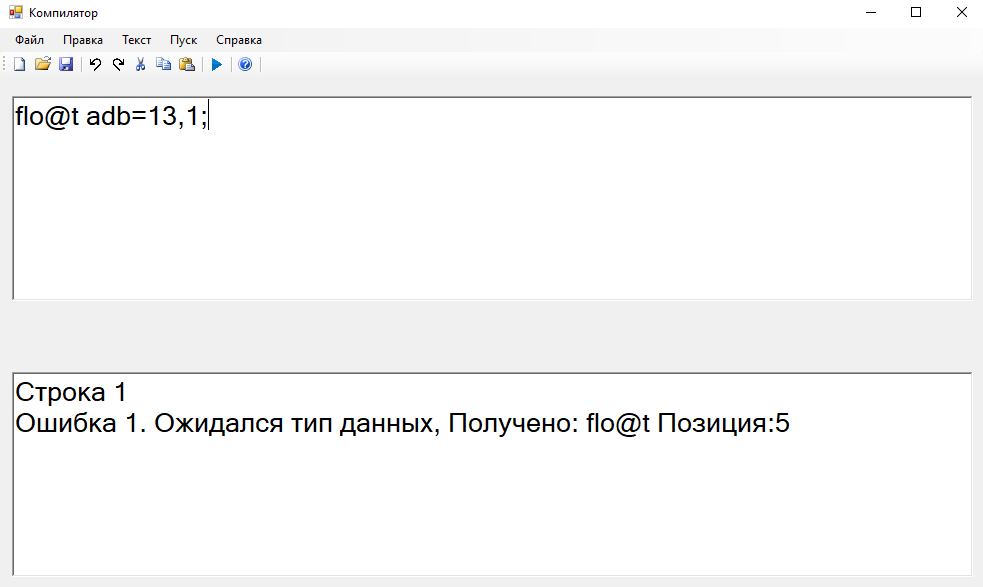


Рис. 3. Пример неверного типа данных

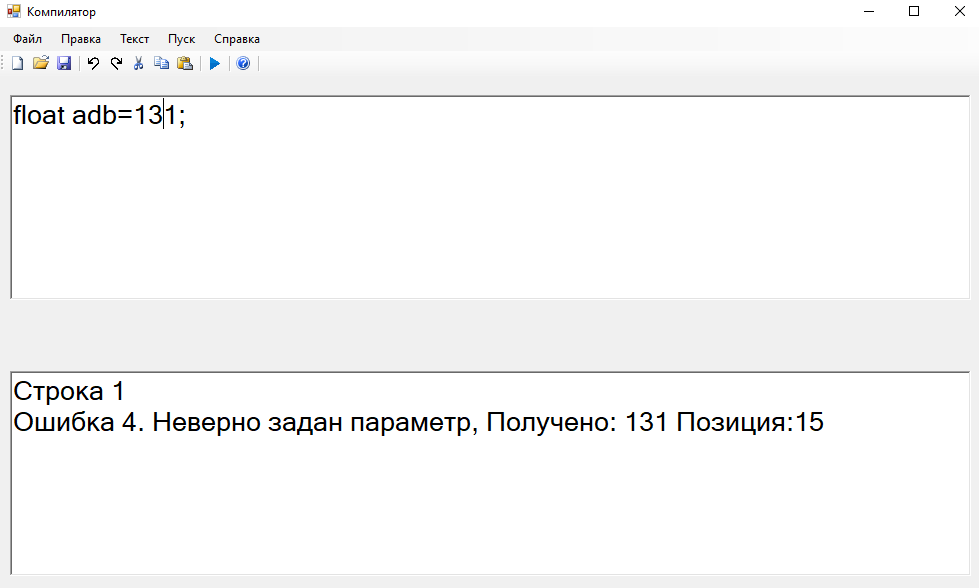


Рис. 4. Пример неверного ввода присваиваемого значения.

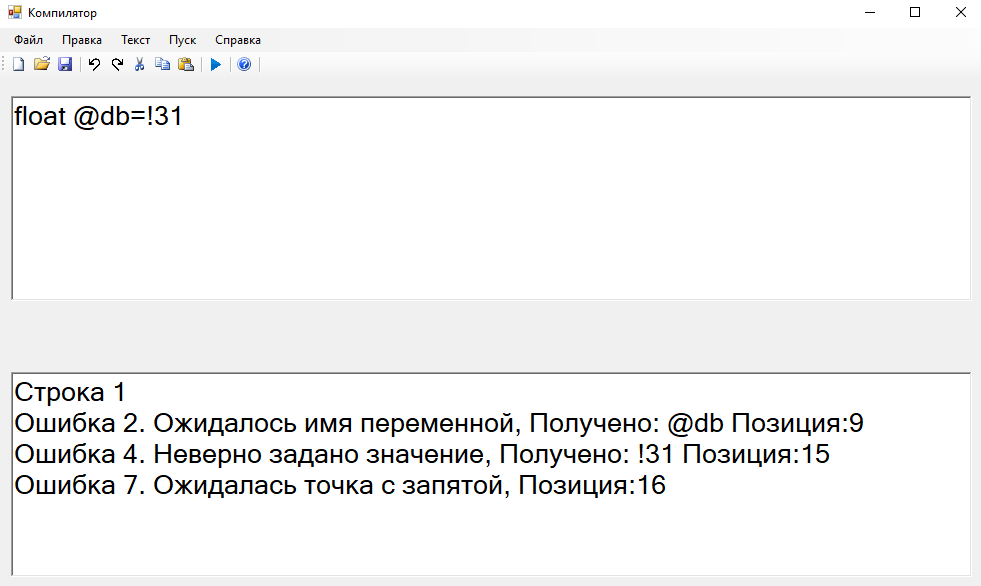


Рис. 5. Пример неверного ввода имени переменной, неверного значения и отсутствия точки с запятой.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения курсовой работы был реализован синтаксический анализатор (парсер) для объявления переменных с плавающей точкой языка С++. Была определена грамматика  G[Z] в нотации Хомского. Согласно классификации Хомского, грамматика G[Z] является автоматной. Для разработанной грамматики G[Z] был реализован граф конечного автомата.

# **СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Шорников, Ю. В. Теория и практика языковых процессов: Учебное пособие. – Новосибирск: Изд-во НГТУ, 2004.
2. Microsoft Developer Network [Электронный ресурс]. Microsoft.com (Дата обращения: 05.05.2022) URL: <http://msdn.microsoft.com/ru-RU/library>
3. Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии, инструменты: пер. с англ., 2-е изд. – М.: Вильямс, 2008.
4. Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. СПб.: Наука и техника, 2006.-320 с.

# **ЛИСТИНГ АЛГОРИТМА СИНТАКСИЧЕСКОГО АНАЛИЗА**

**Form1.cs**

private void StartToolStripButton\_Click(object sender, EventArgs e)

{

ResultsWindow.Text = "";

string[] str = CodeWindow.Text.Split(new char[] { '\n' });

for (int j = 0; j < str.Length; j++)

{

if (str[j].Contains('='))

str[j] = str[j].Replace("=", " = ");

if (str[j].Contains(';'))

str[j] = str[j].Replace(";", " ; ");

ResultsWindow.AppendText("Строка " + (j + 1) + "\r\n");

if (str.Length > 0)

{

string temp\_str = String.Empty;

var basicscaner = new Scanner(str[j]);

var symbol = basicscaner.Recursive\_Syntaxis();

int i = 0;

while (i < symbol.Length)

{

if (symbol[i] == '|')

{

ResultsWindow.AppendText(temp\_str);

temp\_str = String.Empty;

}

else

{

temp\_str += symbol[i];

}

i++;

}

}

}

}

**Scanner.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Lab1

{

public enum SymbolType

{

DataType,

Name,

Equally,

Argument,

Сomma,

Semicolon,

Eof,

Error

}

public struct Symbol

{

public SymbolType Type;

public string Value;

public int Index;

public Symbol(SymbolType type = SymbolType.Eof, string value = " ", int index = 0)

{

Type = type;

Value = value;

Index = index;

}

}

public class Scanner

{

private int \_index = 0;

private string \_inputString;

public int StringNum = 1;

private int switcIndex = 0;

private string result;

private int temp\_index\_start;

private int temp\_index\_last;

public int mistakes\_count = 0;

private bool last\_symbol = false;

private bool not\_false\_symbol = true;

public bool name\_have = false;

public Scanner(string inputString)

{

\_inputString = inputString;

\_index = 0;

}

public void Reset(int temp)

{

\_index = temp;

}

public void Set\_start\_point(int temp\_index)

{

if (temp\_index > 0)

temp\_index\_start = temp\_index - 1;

else

temp\_index\_start = 0;

}

public void Set\_last\_point(int number = 0)

{

temp\_index\_last = \_index - number;

}

public Symbol GetSymbol()

{

var value = String.Empty;

int temp\_index;

while (\_index < \_inputString.Length && \_inputString[\_index] == ' ')

{

\_index++;

}

temp\_index = \_index;

while (!Eof)

{

value += \_inputString[\_index];

\_index++;

if (Eof || value == ";")

{

break;

}

if (\_inputString[\_index] == '\n')

{

StringNum++;

}

if (\_inputString[\_index] == ' ')

{

Set\_start\_point(temp\_index);

break;

}

}

return CheckSymbolType(value);

}

public Symbol CheckSymbolType(string value)

{

if (GetCommaSymbol(value).Type != SymbolType.Error)

{

return GetCommaSymbol(value);

}

if (GetDataTypeSymbol(value).Type != SymbolType.Error)

{

return GetDataTypeSymbol(value);

}

if (GetSemicolonSymbol(value).Type != SymbolType.Error)

{

return GetSemicolonSymbol(value);

}

if (GetArgumentSymbol(value).Type != SymbolType.Error)

{

return GetArgumentSymbol(value);

}

if (GetNameSymbol(value).Type != SymbolType.Error && switcIndex >= 1)

{

return GetNameSymbol(value);

}

if (GetEquallySymbol(value).Type != SymbolType.Error)

{

return GetEquallySymbol(value);

}

return new Symbol(SymbolType.Error, value);

}

public string Recursive\_Syntaxis(bool exit = true)

{

string[] st = \_inputString.Split(' ');

int count = 0;

foreach (string \_s in st)

{

count++;

}

count--;

Symbol symbol = GetSymbol();

SymbolType prev\_type = 0;

SymbolType type;

string message = "";

switch (switcIndex)

{

case 0:

type = SymbolType.DataType;

message = "Ошибка 1. Ожидался тип данных, Получено: {0} Позиция:{1}";

if (not\_false\_symbol)

Set\_last\_point();

break;

case 1:

type = SymbolType.Name;

message = "Ошибка 2. Ожидалось имя переменной, Получено: {0} Позиция:{1}";

if (not\_false\_symbol)

Set\_last\_point();

break;

case 2:

type = SymbolType.Equally;

message = "Ошибка 3. Ожидался знак =, Получено: {0} Позиция:{1}";

if (not\_false\_symbol)

Set\_last\_point();

break;

case 3:

type = SymbolType.Argument;

message = "Ошибка 4. Неверно задано значение, Получено: {0} Позиция:{1}";

if (not\_false\_symbol)

Set\_last\_point();

break;

case 4:

type = SymbolType.Semicolon;

message = String.Format(" Ошибка 7. Ожидалась точка с запятой, Позиция:{0}", temp\_index\_last + 1);

if (not\_false\_symbol)

Set\_last\_point();

break;

default:

type = SymbolType.Eof;

if (mistakes\_count == 0)

result += String.Format("Ошибки отсутствуют") + "|";

break;

}

if (type == SymbolType.Eof)

return "";

if (symbol.Type == type)

{

switcIndex++;

not\_false\_symbol = true;

Set\_last\_point();

Recursive\_Syntaxis();

}

else

{

switcIndex++;

if (exit)

{

result += String.Format(message, symbol.Value, temp\_index\_last) + "|";

last\_symbol = false;

not\_false\_symbol = false;

mistakes\_count++;

Set\_last\_point();

return Recursive\_Syntaxis(false);

}

if (!exit)

{

if (!last\_symbol)

switcIndex--;

else switcIndex = switcIndex - 2;

}

not\_false\_symbol = false;

if (type == SymbolType.Eof)

{

return result;

}

Reset(temp\_index\_last);

Recursive\_Syntaxis();

}

return result;

}

private bool Eof

{

get

{

return \_index == \_inputString.Length;

}

}

private Symbol GetEquallySymbol(string value)

{

if (value == "=")

{

return new Symbol(SymbolType.Equally, value, \_index);

}

else

{

return GetErrorSymbol(\_index, value);

}

}

private Symbol GetNameSymbol(string value)

{

int i = 0;

bool wrong\_symbol = false;

while (i < value.Length)

{

if (!IsLetter(value[i]) && !IsDigit(value[i]) || IsDigit(value[0]))

{

wrong\_symbol = true;

}

i++;

}

if ((value != String.Empty && !wrong\_symbol) && (value != "double" || value != "float"))

{

return new Symbol(SymbolType.Name, value, \_index);

}

else

{

return GetErrorSymbol(\_index, value);

}

}

private Symbol GetSemicolonSymbol(string value)

{

if (value == ";")

{

return new Symbol(SymbolType.Semicolon, value, \_index);

}

else

{

return GetErrorSymbol(\_index, value);

}

}

private Symbol GetDataTypeSymbol(string value)

{

if (value == "double" || value == "float")

{

return new Symbol(SymbolType.DataType, value, \_index);

}

else

{

return GetErrorSymbol(\_index, value);

}

}

private Symbol GetCommaSymbol(string value)

{

if (value == ",")

{

return new Symbol(SymbolType.Сomma, value, \_index);

}

else

{

return GetErrorSymbol(\_index, value);

}

}

private Symbol GetErrorSymbol(int index, string value)

{

return new Symbol(SymbolType.Error, value, \_index);

}

private Symbol GetArgumentSymbol(string value)

{

int i = 0;

int comma = 0;

bool not\_digit = false;

while (i < value.Length)

{

if (IsDigit(value[i]))

{

i++;

}

else if (value[i] == ',')

{

comma++;

i++;

}

else

{

not\_digit = true;

break;

}

}

if (not\_digit || value == String.Empty || comma != 1)

{

return GetErrorSymbol(\_index, value);

}

else

{

return new Symbol(SymbolType.Argument, value, \_index);

}

}

private bool IsDigit(char c)

{

return c >= '0' && c <= '9';

}

private bool IsLetter(char c)

{

return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') || (c >= 'а' && c <= 'я') || (c >= 'А' && c <= 'Я');

}

}

}