НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**КУРСОВА РОБОТА**

з дисципліни «СИСТЕМНЕ ПРОГРАМУВАННЯ-2».

на тему: «Розробка компілятора підмножини команд мови С, що містять арифметичні дії і дужкові форми»

Студента 3 курсу групи ІО-91

спеціальності

123 «Комп’ютерна  інженерія»

Слюсаренка Андрія Сергійовича

Керівник      доцент Павлов В.Г.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_

Кількість балів \_\_\_\_\_\_\_\_\_\_

Оцінка:  ECTS   \_\_\_\_\_\_\_\_\_\_\_

Київ 2021 рік

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) інформатики та обчислювальної техніки

( повна назва )

Кафедра обчислювальної техніки

( повна назва )

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 123 «Комп’ютерна інженерія»

(шифр і назва)

***З А В Д А Н Н Я***

*НА КУРСОВУ РОБОТУ СТУДЕНТУ*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Слюсаренка Андрія Сергійовича\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(прізвище, ім’я, по батькові)*

1. Тема роботи «Розробка компілятора підмножини команд мови C (або Python), що містять арифметичні дії і дужкові форми»

керівник роботи Павлов В.Г. к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 22 грудня 2021р.

3. Вхідні дані до роботи:

- вхідна мова: C ;

- цільова мова: Assembler x86;

- типи конструкцій: арифметичні дії, розгалуження, функції.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- розробити лексичний аналізатор мови C ;

- розробити синтаксичний аналізатор мови C ;

- розробити генератор коду мови Assembler x86.

5. Перелік графічного матеріалу;

- алгоритм лексичного аналізу;

- алгоритм синтаксичного аналізу;

- дерева розбору.

6. Дата видачі завдання 15 жовтня 2021 р

**Зміст**

[3. Вступ 4](#_Toc91095888)

[4. Огляд методів побудови системних програм 5](#_Toc91095889)

[5. Блок-схема роботи програми 6](#_Toc91095890)

[6. Лістинг програми компілятору 8](#_Toc91095891)

[7. Результати перевірки працездатності програми 41](#_Toc91095892)

[7.1 Приклади виконання з помилками 41](#_Toc91095893)

[7.2 Приклади виконання без помилок: 43](#_Toc91095894)

[8. Висновок 45](#_Toc91095895)

[9. Список використаної літератури 46](#_Toc91095896)

3. Вступ

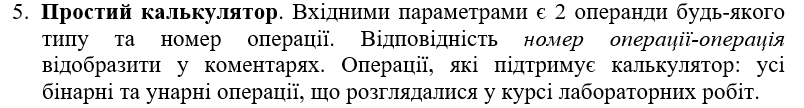
У даній роботі необхідно створити програму, що буде реалізувати заданий алгоритм роботи простого калькулятора на мові програмування відповідно варіанту, тобто С. Далі розробити компілятор, який виконує переклад вхідного коду програми на мові С, обробляє його та здійснює генерацію асемблерного коду MASM32. Для побудови компілятора було обрано мову С#. Так, як це мова зі статичною типізацією, то процес розробки і налагодження програми значно спрощується, прискорюючи пошук помилок та збільшуючи читабельність коду.

Виконання розробленої програми можна явно розділити на кілька етапів компіляції: лексичний аналіз – поділ вихідного коду на токени, парсинг – перевірка правильності синтаксису мови С використовуючи побудову дерева розбору, генерація коду - генерується вихідний асемблерний код на основі дерева розбору із попереднього етапу.

*Варіант:*

****

*Алгоритм за варіантом:*



4. Огляд методів побудови системних програм

При виконання програми-компілятора можна явно виділити 3 етапи компіляції:

* Лексичний аналіз (Front end)
* Синтаксичний аналіз (Middle end)
* Генерація коду (Back end)

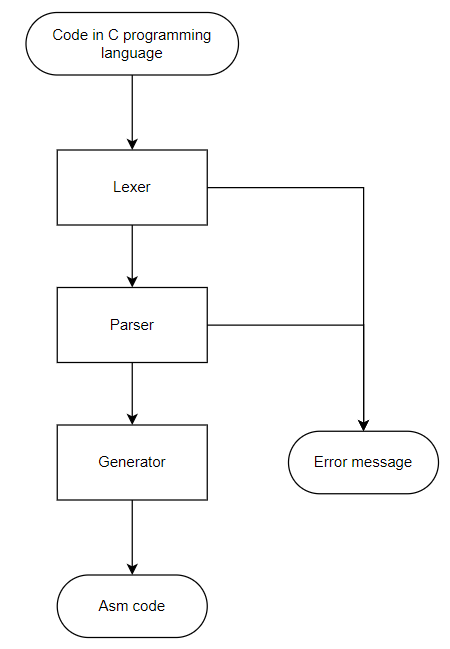
Лексичний аналіз - частина процесу компіляції при якому, відбувається дослідження витягненого коду програми на мові С з файлу у стрічку, шляхом розкладу його на складові частини, так звані токени, згідно правил, які визначає мова реалізації вхідної програми. Внаслідок виконання лексерy одержимо вихідні дані у вигляді списку токенів.

Синтаксичний аналіз - перевіряє валідність запису граматичних конструкцій мови, а саме проводиться перевірка пpавильності поєднання токенів один з одним у вхідному наборі даних згідно до правил самої мови. У разі успішного виконання вхідний список токенів має бути трансформований у структуру (*AST/parse tree*), так званого дерева розбору, інакше буде виведено повідомлення про помилку, у якій буде вказано, де це сталось і чому.

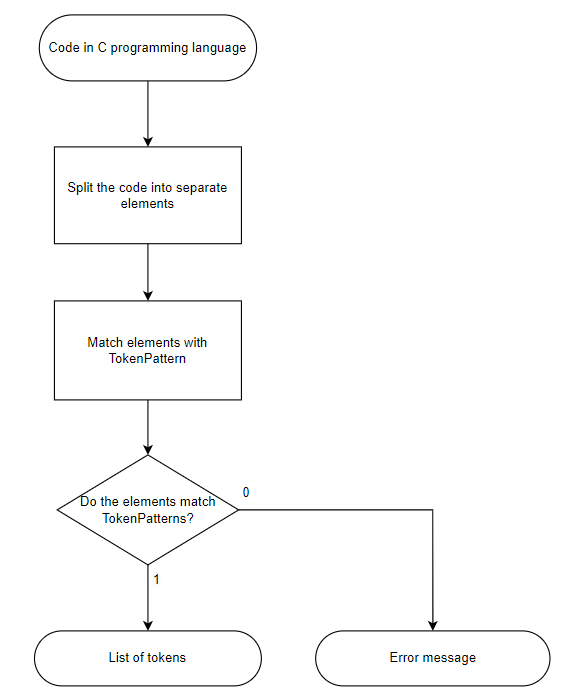
Генерація коду - генерується вихідний асемблерний код на основі дерева розбору із попереднього етапу.

Саме такою була типова архітектура компілятора колись. Це було пов’язано з обмеженим об’ємом пам’яті, якої в свою чергу було недостатньо для збереження всієї програми-компілятора. На сьогоднішній день такої проблеми вже немає, тому при розробці можна застосовувати інші методи, які не будуть мати зайві процедури, що значно прискорить процес компіляції.

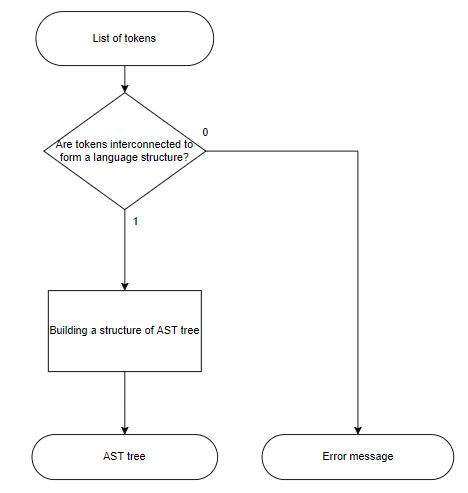
5. Блок-схема роботи програми



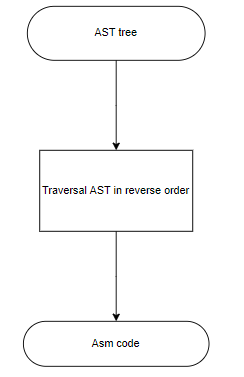
**Блок-схема роботи Лексера** (Лексичний аналіз)



**Блок-схема роботи Парсера** (Синтаксичний аналіз)



**Блок-схема роботи Генератора коду**



6. Лістинг програми компілятору

**Program.cs**

using System.Collections.Generic;  
using System.IO;  
  
namespace KR  
{  
 public class Program  
 {  
 private static void *Main*(string[] args)  
 {  
 const string *filenameIncome* = "KP-19-C#-IO-91-Sliusarenko.c";  
 const string *filenameOutcome* = "KP-19-C#-IO-91-Sliusarenko.asm";  
  
 string codeFromFile =  
 File.*ReadAllText*(  
 *$*"{Directory.*GetParent*(Directory.*GetCurrentDirectory*())}\\{*filenameIncome*}");  
  
 Lexer lexer = new Lexer(codeFromFile);  
 Dictionary<string, List<object>> tokens = lexer.ExtractTokens();  
  
 List<object> tokenVals = tokens["tokenVals"];  
 List<object> tokensExpressedInEnums = tokens["tokensExpressedInEnums"];  
  
 Parser parser = new Parser(tokensExpressedInEnums, tokenVals);  
  
 var tree = parser.Parse();  
  
 Generator generator = new Generator(tree);  
 File.*WriteAllText*(*$*"{Directory.*GetParent*(Directory.*GetCurrentDirectory*())}{*filenameOutcome*}",  
 generator.Generate());  
 }  
 }  
}

**Lexer.cs**

using System.Collections.Generic;  
using System.Text;  
  
namespace KR  
**{** public class Lexer  
 {  
 private readonly string \_textWithIncomingC;  
 private List<string> \_separatedTextOfIncomingC;  
 private List<dynamic> \_tokensExpressedInEnums;  
 private List<object> \_tokenVals;  
  
 private List<string> \_patternsSymb = new()  
 {  
 TokenPattern.*ADDITION*.Field, TokenPattern.*SEMICOLON*.Field, TokenPattern.*OR\_BRACKET*.Field,  
 TokenPattern.*CR\_BRACKET*.Field,  
 TokenPattern.*OС\_BRACE*.Field, TokenPattern.*CС\_BRACE*.Field, TokenPattern.*EQUAL*.Field,  
 TokenPattern.*SUBTRACTION*.Field,  
 TokenPattern.*COLON*.Field, TokenPattern.*QUEST\_MARK*.Field, TokenPattern.*COMMA*.Field, TokenPattern.*LT*.Field,  
 TokenPattern.*GT*.Field,  
 TokenPattern.*MODULO*.Field, TokenPattern.*BITWISE\_AND*.Field, TokenPattern.*BITWISE\_OR*.Field,  
 TokenPattern.*BITWISE\_XOR*.Field,  
 TokenPattern.*MULTIPLY*.Field, TokenPattern.*DIVISION*.Field  
 };  
  
 public Lexer(string textWithIncomingC)  
 {  
 \_textWithIncomingC = textWithIncomingC;  
 }  
  
 public Dictionary<string, List<object>> ExtractTokens()  
 {  
 FileSplitter();  
 TokenProcessor processor = new TokenProcessor();  
 \_tokensExpressedInEnums = new List<dynamic>();  
 \_tokenVals = new List<object>();  
  
  
 foreach (string item in \_separatedTextOfIncomingC)  
 {  
 if (!TokenPattern.*INVALID*.Equals(processor.MatchItemWithTokenPattern(item)))  
 {  
 \_tokensExpressedInEnums.Add(processor.MatchItemWithTokenPattern(item));  
 \_tokenVals.Add(item);  
 }  
 else if (TokenPattern.*INVALID*.Equals(processor.MatchItemWithTokenPattern(item)))  
 {  
 var amount = 0;  
 int index;  
 for (index = 0; index < item.Length; index++)  
 {  
 if (!TokenPattern.*INVALID*.Equals(processor.MatchItemWithTokenPattern(item[index].ToString())))  
 {  
 \_tokensExpressedInEnums.Add(processor.MatchItemWithTokenPattern(item[index].ToString()));  
 \_tokenVals.Add(item[index].ToString());  
 amount++;  
 }  
  
 if (!TokenPattern.*INVALID*.Equals(processor.MatchItemWithTokenPattern(item.Substring(index))))  
 {  
 \_tokensExpressedInEnums.Add(processor.MatchItemWithTokenPattern(item.Substring(index)));  
 \_tokenVals.Add(item.Substring(index));  
 amount += item.Substring(index).Length;  
 }  
 }  
  
 if (amount != item.Length) throw new UnsuspectedTokenException(\_tokenVals, index);  
 }  
 }  
  
  
 for (int i = 0; i < \_tokensExpressedInEnums.Count - 1; i++)  
 {  
 if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*VARIABLE*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*OR\_BRACKET*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*FUNCTION*;  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*EQUAL*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*EQUAL*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*EQUAL\_L*;  
 \_tokenVals[i] = TokenPattern.*EQUAL\_L*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*NOT*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*EQUAL*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*NOT\_EQUAL*;  
 \_tokenVals[i] = TokenPattern.*NOT\_EQUAL*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*LT*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*EQUAL*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*LTE*;  
 \_tokenVals[i] = TokenPattern.*LTE*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*GT*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*EQUAL*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*GTE*;  
 \_tokenVals[i] = TokenPattern.*GTE*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*GT*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*GT*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*RIGHT\_SHIFT*;  
 \_tokenVals[i] = TokenPattern.*RIGHT\_SHIFT*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*LT*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*LT*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*LEFT\_SHIFT*;  
 ;  
 \_tokenVals[i] = TokenPattern.*RIGHT\_SHIFT*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*BITWISE\_AND*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*BITWISE\_AND*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*AND*;  
 \_tokenVals[i] = TokenPattern.*AND*.Field;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 else if (\_tokensExpressedInEnums[i].Equals(TokenPattern.*BITWISE\_OR*) &&  
 \_tokensExpressedInEnums[i + 1].Equals(TokenPattern.*BITWISE\_OR*))  
 {  
 \_tokensExpressedInEnums[i] = TokenPattern.*OR*;  
 \_tokenVals[i] = TokenPattern.*OR*;  
 \_tokensExpressedInEnums.RemoveAt(i + 1);  
 \_tokenVals.RemoveAt(i + 1);  
 }  
 }  
  
 Dictionary<string, List<dynamic>> res = new Dictionary<string, List<dynamic>>();  
 res.Add("tokenVals", \_tokenVals);  
 res.Add("tokensExpressedInEnums", \_tokensExpressedInEnums);  
 return res;  
 }  
  
 private string CommentHandler()  
 {  
 var incomingCWithoutComments = new StringBuilder();  
 var flag = true;  
  
 string[] split = \_textWithIncomingC.Split("");  
 for (var index = 0; index < split.Length; index++)  
 {  
 var symb = split[index];  
 if (symb.Equals(TokenPattern.*DIVISION*.Field) &&  
 \_textWithIncomingC[index + 1].ToString().Equals(TokenPattern.*MULTIPLY*.Field) &&  
 index != \_textWithIncomingC.Length - 1)  
 flag = false;  
  
 else if (symb.Equals(TokenPattern.*DIVISION*.Field) &&  
 \_textWithIncomingC[index - 1].ToString().Equals(TokenPattern.*MULTIPLY*.Field)) flag = true;  
 else if (flag)  
 {  
 if (\_patternsSymb.Contains(symb)) incomingCWithoutComments.Append(" ").Append(symb).Append(" ");  
 else if (symb.Equals("!")) incomingCWithoutComments.Append(symb).Append(" ");  
 else incomingCWithoutComments.Append(symb);  
 }  
 }  
  
 return incomingCWithoutComments.ToString();  
 }  
  
 public void FileSplitter()  
 {  
 var separatedIncomingC = new List<string>();  
 foreach (string item in CommentHandler().Split(" "))  
 {  
 if (item.Equals("")) continue;  
 if (item.Contains(TokenPattern.*OR\_BRACKET*.Field + TokenPattern.*CR\_BRACKET*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*OR\_BRACKET*.Field + TokenPattern.*CR\_BRACKET*.Field,  
 " " + TokenPattern.*OR\_BRACKET*.Field + " " + TokenPattern.*CR\_BRACKET*.Field + " ")  
 .Split(" "));  
 }  
  
 if (item.Length > 1)  
 {  
 if (item.Contains(TokenPattern.*ADDITION*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*ADDITION*.Field, " " + TokenPattern.*ADDITION*.Field + " ")  
 .Split(" "));  
 }  
  
 if (item.Contains(TokenPattern.*SEMICOLON*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*SEMICOLON*.Field, " " + TokenPattern.*SEMICOLON*.Field + " ")  
 .Split(" "));  
 }  
  
 if (item.Contains(TokenPattern.*OС\_BRACE*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*OС\_BRACE*.Field, " " + TokenPattern.*OС\_BRACE*.Field + " ")  
 .Split(" "));  
 }  
  
 if (item.Contains(TokenPattern.*CС\_BRACE*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*CС\_BRACE*.Field, " " + TokenPattern.*CС\_BRACE*.Field + " ")  
 .Split(" "));  
 }  
 }  
 else if (item.Contains(TokenPattern.*MAIN*.Field))  
 {  
 separatedIncomingC.AddRange(item  
 .Replace(TokenPattern.*MAIN*.Field, TokenPattern.*MAIN*.Field + " ")  
 .Split(" "));  
 }  
  
 else if (item.Contains(TokenPattern.*RETURN*.Field))  
 {  
 separatedIncomingC.AddRange(item.Replace(TokenPattern.*RETURN*.Field, " " + TokenPattern.*RETURN*.Field)  
 .Split(" "));  
 }  
 else separatedIncomingC.Add(item);  
 }  
  
 \_separatedTextOfIncomingC = separatedIncomingC;  
 }  
 }  
**}**

**Parser.cs**

using System;  
using System.Collections.Generic;  
  
namespace KR  
{  
 public class Parser  
 {  
 private List<dynamic> \_tokensExpressedInEnums;  
 private List<object> \_tokenVals;  
  
 public Parser(List<object> tokenVals, List<dynamic> tokensExpressedInEnums)  
 {  
 \_tokensExpressedInEnums = new List<dynamic>();  
 foreach (var token in tokensExpressedInEnums)  
 {  
 \_tokensExpressedInEnums.Add((string) token);  
 }  
  
 \_tokenVals = tokenVals;  
 }  
  
  
 public Dictionary<string, Dictionary<string, List<string>>> Parse()  
 {  
 for (var index = 0; index < \_tokensExpressedInEnums.Count; index++)  
 {  
 var tokenPattern = \_tokensExpressedInEnums[index];  
 if (tokenPattern.Equals(TokenPattern.*CONST\_FLOAT*))  
 {  
 \_tokenVals[index] = int.*Parse*((string) \_tokenVals[index]);  
 \_tokensExpressedInEnums[index] = TokenPattern.*CONST\_INT*;  
 }  
 else if (tokenPattern.Equals(TokenPattern.*CONST\_INT*))  
 {  
 \_tokenVals[index] = int.*Parse*((string) \_tokenVals[index]);  
 }  
 }  
  
 var operations = new List<TokenPattern>  
 {  
 TokenPattern.*EQUAL\_L*, TokenPattern.*AND*, TokenPattern.*OR*, TokenPattern.*NOT\_EQUAL*, TokenPattern.*LT*,  
 TokenPattern.*LTE*, TokenPattern.*GT*, TokenPattern.*GTE*, TokenPattern.*MODULO*, TokenPattern.*BITWISE\_AND*,  
 TokenPattern.*BITWISE\_OR*, TokenPattern.*BITWISE\_XOR*, TokenPattern.*LEFT\_SHIFT*, TokenPattern.*RIGHT\_SHIFT*,  
 TokenPattern.*MULTIPLY*, TokenPattern.*DIVISION*, TokenPattern.*ADDITION*, TokenPattern.*SUBTRACTION*,  
 TokenPattern.*QUEST\_MARK*, TokenPattern.*COLON* };  
  
  
 for (var index = 0; index < \_tokensExpressedInEnums.Count; index++)  
 {  
 var pattern = \_tokensExpressedInEnums[index];  
 var markedList = new List<string> {TokenPattern.*MAIN*.ValueOfIt(), TokenPattern.*FUNCTION*.ValueOfIt()};  
 if (pattern.Equals(TokenPattern.*INT*) || pattern.Equals(TokenPattern.*FLOAT*))  
 {  
 if (index == 0)  
 {  
 if (!markedList.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, 1);  
 }  
 else  
 {  
 var checkList = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*MAIN*.ValueOfIt(),  
 TokenPattern.*FUNCTION*.ValueOfIt()  
 };  
 var signs = new List<string>  
 {  
 TokenPattern.*COMMA*.ValueOfIt(), TokenPattern.*OR\_BRACKET*.ValueOfIt(),  
 TokenPattern.*CС\_BRACE*.ValueOfIt(), TokenPattern.*SEMICOLON*.ValueOfIt(),  
 TokenPattern.*OС\_BRACE*.ValueOfIt()  
 };  
 if (!(checkList.Contains(\_tokensExpressedInEnums[index + 1]) &&  
 signs.Contains(\_tokensExpressedInEnums[index - 1])))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 }  
 else if (markedList.Contains(\_tokensExpressedInEnums[index - 1]))  
 {  
 var checkList = new List<string>  
 {  
 TokenPattern.*INT*.ValueOfIt(), TokenPattern.*FLOAT*.ValueOfIt(), TokenPattern.*RETURN*.ValueOfIt()  
 };  
 if (!checkList.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !TokenPattern.*OR\_BRACKET*.Equals(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*OR\_BRACKET*.Equals(pattern))  
 {  
 var checkList = new List<string>  
 {  
 TokenPattern.*CR\_BRACKET*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt(),  
 TokenPattern.*CONST\_INT*.ValueOfIt(), TokenPattern.*INT*.ValueOfIt(), TokenPattern.*FLOAT*.ValueOfIt()  
 };  
 if (!markedList.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !checkList.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*CR\_BRACKET*.Equals(pattern))  
 {  
 List<string> checkList = new List<string>  
 {  
 TokenPattern.*OR\_BRACKET*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt(),  
 TokenPattern.*CONST\_INT*.ValueOfIt()  
 };  
 if (!checkList.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !new List<string>  
 {  
 TokenPattern.*OС\_BRACE*.ValueOfIt(), TokenPattern.*SEMICOLON*.ValueOfIt()  
 }.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*COMMA*.Equals(pattern))  
 {  
 List<string> checkList = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt()  
 };  
 List<string> typeVars = new List<string>  
 {  
 TokenPattern.*INT*.ValueOfIt(), TokenPattern.*FLOAT*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt(),  
 TokenPattern.*CONST\_INT*.ValueOfIt()  
 };  
 if (!checkList.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !typeVars.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*RETURN*.Equals(pattern))  
 {  
 List<string> arr1 = new List<string>  
 {TokenPattern.*SEMICOLON*.ValueOfIt(), TokenPattern.*OС\_BRACE*.ValueOfIt()};  
 List<string> arr2 = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt(),  
 TokenPattern.*FUNCTION*.ValueOfIt(),  
 TokenPattern.*NOT*.ValueOfIt()  
 };  
 if (!arr1.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !arr2.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*EQUAL*.Equals(pattern))  
 {  
 List<string> checkList = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt(),  
 TokenPattern.*FUNCTION*.ValueOfIt(),  
 TokenPattern.*NOT*.ValueOfIt()  
 };  
 if (!TokenPattern.*VARIABLE*.ValueOfIt().Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !checkList.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*SEMICOLON*.Equals(pattern))  
 {  
 List<string> checkList = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt(),  
 TokenPattern.*FUNCTION*.ValueOfIt(),  
 TokenPattern.*NOT*.ValueOfIt()  
 };  
 List<string> arr2 = new List<string>  
 {  
 TokenPattern.*RETURN*.ValueOfIt(), TokenPattern.*CС\_BRACE*.ValueOfIt(),  
 TokenPattern.*INT*.ValueOfIt(),  
 TokenPattern.*FLOAT*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt()  
 };  
 if (!checkList.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !arr2.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*OС\_BRACE*.ValueOfIt().Equals(pattern))  
 {  
 List<string> arr1 = new List<string>  
 {  
 TokenPattern.*RETURN*.ValueOfIt(), TokenPattern.*INT*.ValueOfIt(), TokenPattern.*FLOAT*.ValueOfIt()  
 };  
 if (!TokenPattern.*CR\_BRACKET*.Equals(\_tokensExpressedInEnums[index - 1]) ||  
 !arr1.Equals(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*CС\_BRACE*.ValueOfIt().Equals(pattern))  
 {  
 if (!TokenPattern.*SEMICOLON*.ValueOfIt().Equals(\_tokensExpressedInEnums[index - 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (new List<string> {TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt()}  
 .Contains(pattern))  
 {  
 if (!operations.Contains(\_tokensExpressedInEnums[index - 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (operations.Contains(pattern))  
 {  
 List<string> arr1 = new List<string>  
 {  
 TokenPattern.*CONST\_INT*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt()  
 };  
 List<string> arr2 = new List<string>  
 {  
 TokenPattern.*CONST\_INT*.ValueOfIt(), TokenPattern.*VARIABLE*.ValueOfIt(),  
 TokenPattern.*NOT*.ValueOfIt()  
 };  
 if (!arr1.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !arr2.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 else if (TokenPattern.*NOT*.ValueOfIt().Contains(pattern))  
 {  
 List<string> arr2 = new List<string>  
 {  
 TokenPattern.*VARIABLE*.ValueOfIt(), TokenPattern.*CONST\_INT*.ValueOfIt()  
 };  
 if (!operations.Contains(\_tokensExpressedInEnums[index - 1]) ||  
 !arr2.Contains(\_tokensExpressedInEnums[index + 1]))  
 throw new UnsuspectedTokenException(\_tokenVals, index + 1);  
 }  
 }  
  
 var result = new Dictionary<string, Dictionary<string, List<string>>>();  
  
 for (var indexToken = 0; indexToken < \_tokensExpressedInEnums.Count; indexToken++)  
 {  
 var inEnum = \_tokensExpressedInEnums[indexToken];  
 List<string> arr1 = new List<string> {TokenPattern.*FUNCTION*.ValueOfIt(), TokenPattern.*MAIN*.ValueOfIt()};  
 List<string> arr2 = new List<string> {TokenPattern.*FLOAT*.ValueOfIt(), TokenPattern.*INT*.ValueOfIt()};  
  
 if (arr1.Contains(inEnum) && arr2.Contains(\_tokensExpressedInEnums[indexToken - 1]))  
 {  
 if (result.ContainsKey((string) \_tokenVals[indexToken]))  
 **{** throw new InvalidStatementException("Invalid statement: func " + \_tokenVals[indexToken] +  
 " already exists");  
 **}** for (var indexTokenIn = 0; indexTokenIn < \_tokensExpressedInEnums.Count; indexTokenIn++)  
 {  
 var tokensExpressedInEnum = \_tokensExpressedInEnums[indexTokenIn];  
 if (indexTokenIn > indexToken)  
 {  
 if (tokensExpressedInEnum.Equals(TokenPattern.*VARIABLE*.ValueOfIt()) ||  
 tokensExpressedInEnum.Equals(TokenPattern.*RETURN*.ValueOfIt()))  
 {  
 List<string> checkList = new List<string>  
 {  
 TokenPattern.*FLOAT*.ValueOfIt(), TokenPattern.*INT*.ValueOfIt()  
 };  
 if (\_tokenVals[indexTokenIn].Equals(TokenPattern.*RETURN*) &&  
 result[(string) \_tokenVals[indexToken]].ContainsKey(TokenPattern.*RETURN*.Field))  
 {  
 result.Add((string) \_tokenVals[indexTokenIn],  
 new Dictionary<string, List<string>>());  
 }  
 else if (checkList.Contains(\_tokensExpressedInEnums[indexTokenIn - 1]))  
 {  
 if (result[(string) \_tokenVals[indexToken]]  
 .ContainsKey((string) \_tokenVals[indexTokenIn]))  
 result[(string) \_tokenVals[indexToken]].Add(  
 (string) \_tokenVals[indexTokenIn],  
 new List<string>());  
 }  
 else  
 {  
 throw new InvalidStatementException("Invalid statement: \nVariable " +  
 \_tokenVals[indexTokenIn] +  
 " already exists in func " +  
 \_tokenVals[indexToken]);  
 }  
 }  
  
 if (\_tokensExpressedInEnums[indexTokenIn + 1].Equals(TokenPattern.*EQUAL*))  
 {  
 if (result[(string) \_tokenVals[indexToken]]  
 .ContainsKey((string) \_tokenVals[indexTokenIn]))  
 {  
 List<string> val = new List<string>();  
 result.Add((string) \_tokenVals[indexToken],  
 new Dictionary<string, List<string>>());  
 for (var index = 0; index < \_tokensExpressedInEnums.Count; index++)  
 {  
 var item = \_tokensExpressedInEnums[index];  
 if (index > indexTokenIn + 1)  
 {  
 if (item.Equals(TokenPattern.*SEMICOLON*.ValueOfIt())) break;  
 }  
 else if (item.Equals(TokenPattern.*VARIABLE*.ValueOfIt()))  
 {  
 if (result[(string) \_tokenVals[indexToken]]  
 .ContainsKey((string) \_tokenVals[index]))  
 {  
 val.Add((string) \_tokenVals[index]);  
 }  
 else  
 {  
 throw new InvalidStatementException("Invalid statement: variable " +  
 \_tokenVals[index] +  
 " doesn`t exists in func " +  
 \_tokenVals[indexToken]);  
 }  
 }  
 else  
 {  
 val.Add((string) \_tokenVals[index]);  
 }  
  
 result[(string) \_tokenVals[indexToken]]  
 .Add((string) \_tokenVals[indexTokenIn], val);  
 }  
 }  
 else  
 {  
 throw new InvalidStatementException(  
 "Invalid statement: variable " + \_tokenVals[indexTokenIn] +  
 " doesn`t exist in func " + \_tokenVals[indexToken]);  
 }  
  
 if (\_tokensExpressedInEnums[indexTokenIn]  
 .Equals(TokenPattern.*RETURN*.ValueOfIt()))  
 {  
 List<string> val = new List<string>();  
 for (var index = 0; index < \_tokensExpressedInEnums.Count; index++)  
 {  
 var expressedInEnum = \_tokensExpressedInEnums[index];  
 if (index > indexTokenIn)  
 {  
 if (expressedInEnum.Equals(TokenPattern.*SEMICOLON*.ValueOfIt()))  
 break;  
 if (expressedInEnum.Equals(TokenPattern.*VARIABLE*.ValueOfIt()))  
 {  
 if (result[(string) \_tokenVals[indexToken]]  
 .ContainsKey((string) \_tokenVals[index]))  
 {  
 val.Add((string) \_tokenVals[index]);  
 }  
 else  
 {  
 throw new InvalidStatementException("Invalid code: variable " +  
 \_tokenVals[index] +  
 " doesn`t exists in func " +  
 \_tokenVals[indexToken]);  
 }  
 }  
 else  
 {  
 val.Add((string) \_tokenVals[index]);  
 }  
  
 result[(string) \_tokenVals[indexToken]]  
 .Add((string) \_tokenVals[indexTokenIn], val);  
 }  
 else if (tokensExpressedInEnum.Equals(TokenPattern.*CС\_BRACE* .ValueOfIt())) break;  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
  
 return result;  
 }  
 }  
}

**Generator.cs**

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace KR  
**{** public class Generator  
 {  
 private Dictionary<String, Dictionary<String, List<String>>> \_varsForGeneration;  
  
 public Generator(Dictionary<String, Dictionary<String, List<String>>> varsForGeneration)  
 {  
 this.\_varsForGeneration = varsForGeneration;  
 }  
  
 public string Generate()  
 {  
 var vars = new StringBuilder();  
 var outcomeMasm = new StringBuilder();  
 var valTokensForGeneration = new Dictionary<string, Dictionary<string, List<string>>>();  
  
 var keysVar = \_varsForGeneration.Keys;  
 foreach (var key in keysVar)  
 {  
 valTokensForGeneration.Add(key, new Dictionary<string, List<string>>());  
 var keysSetIn = \_varsForGeneration[key].Keys;  
 foreach (var keySetIn in keysSetIn)  
 {  
 List<string> value = \_varsForGeneration[key][keySetIn];  
 var arrayList = value  
 .Select(item => new TokenProcessor().MatchItemWithTokenPattern(item).ValueOfIt()).ToList();  
 valTokensForGeneration[key].Add(keySetIn, arrayList);  
 }  
 }  
  
 foreach (var keyValuePair in valTokensForGeneration)  
 {  
 Dictionary<string, List<string>> value = keyValuePair.Value;  
 foreach (var valuePair in value)  
 {  
 for (var index = 0; index < valuePair.Value.Count; index++)  
 {  
 var item = valuePair.Value[index];  
 if (item.Equals(TokenPattern.*VARIABLE*.Field) && valuePair.Value.Count > 1)  
 {  
 if (valuePair.Value[index + 1].Equals(TokenPattern.*OR\_BRACKET*.ValueOfIt()))  
 {  
 valuePair.Value[index] = TokenPattern.*FUNCTION*.ValueOfIt();  
 }  
 }  
 }  
 }  
 }  
  
 int amount = 0;  
 foreach (var keyValuePair in valTokensForGeneration)  
 {  
 var value = keyValuePair.Value;  
 foreach (var valuePair in value)  
 {  
 for (var a = 0; a < valuePair.Value.Count; a++)  
 {  
 var s = valuePair.Value[a];  
 if (s.Equals(TokenPattern.*VARIABLE*.ValueOfIt()))  
 {  
 List<string> list = \_varsForGeneration[keyValuePair.Key][valuePair.Key];  
 \_varsForGeneration[keyValuePair.Key][valuePair.Key][a] = list[a] + " " + keyValuePair.Key;  
 }  
 }  
  
 if (valuePair.Key.Equals(TokenPattern.*RETURN*.ValueOfIt()))  
 {  
 vars.Append(" solution\_").Append(keyValuePair.Key).Append(" dd 0\n");  
 }  
 else  
 {  
 vars.Append(keyValuePair.Key).Append("\_").Append(valuePair.Key).Append(" dd 0\n");  
 }  
 }  
 }  
  
 var n = 0;  
 var outcomeBlock = new StringBuilder();  
 foreach (var keyValuePair in valTokensForGeneration)  
 {  
 var key = "";  
 var appendix = 0;  
 if (keyValuePair.Key.Equals(TokenPattern.*MAIN*.Field))  
 {  
 outcomeBlock.Append(keyValuePair.Key).Append(" proc\n");  
 var value = keyValuePair.Value;  
 foreach (var stringListEntry in value)  
 {  
 if (stringListEntry.Value.Count != 0)  
 {  
 var flag = false;  
 if (stringListEntry.Value[0].Equals(TokenPattern.*VARIABLE*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov eax,")  
 .Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [0])  
 .Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*CONST\_INT*.ValueOfIt()))  
  
 {  
 outcomeBlock.Append("\tmov eax,")  
 .Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [0])  
 .Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*EQUAL\_L*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,")  
 .Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tje @start").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @end").Append(n)  
 .Append("\n\t@start").Append(n).Append(":\n\tmov eax,1\n\t@end").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*QUEST\_MARK*.ValueOfIt()))  
 {  
 appendix++;  
 outcomeBlock.Append("\tcmp eax,1\n\tjne @false").Append(appendix).Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*COLON*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tjmp @exit\n\t@false").Append(appendix).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 flag = true;  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*NOT\_EQUAL*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx, {")  
 .Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tje @true").Append(n)  
 .Append("\n\tcmp eax,ebx\n\tje @true").Append(n)  
 .Append("\n\t@true").Append(n).Append(":\n\tmov eax,0\n\t@exit").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*ADDITION*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tadd eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*SUBTRACTION*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tsub eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*MULTIPLY*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx, ").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmul ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*DIVISION*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ecx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcdq\n\tidiv ecx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*GT*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tjg @bigger").Append(n).Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_bigger")  
 .Append(n).Append("\n\t@bigger").Append(n)  
 .Append(":\n\tmov eax,1\n\t@exit\_bigger").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LT*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tjl @lesser").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_lesser").Append(n)  
 .Append("\n\t@lesser").Append(n).Append(":\n\tmov eax,1\n\t@exit\_lesser")  
 .Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LEFT\_SHIFT*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmov cl,ebx\n\tshl eax,cl\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*RIGHT\_SHIFT*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmov cl,ebx\n\tshr eax,cl\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_XOR*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("}\n\txor eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_OR*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tor eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_AND*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tand eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*GTE*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,ebx\n\tjge @big\_eq").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_big\_eq").Append(n)  
 .Append("\n\t@big\_eq").Append(n).Append(":\n\tmov eax,1\n\t@exit\_big\_eq")  
 .Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LTE*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,ebx\n\tjle @les\_eq").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_les\_eq")  
 .Append(n).Append(":\n\tmov eax,1\n\t@exit\_les\_eq").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*MODULO*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov ecx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcdq\n\tidiv ecx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*OR*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,0\n\tje @start")  
 .Append(n).Append("\n\tjmp @end").Append(n)  
 .Append("\n\t@start").Append(n).Append(":\n\tmov eax,ebx\n\t@end").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*AND*.ValueOfIt()))  
 {  
 n++;  
 outcomeBlock.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,0\n\tje @start").Append(n)  
 .Append("\n\tcmp ebx,0\n\tje @start").Append(n).Append("\_").Append(n)  
 .Append("\n\tmov eax,ebx\n\tjmp @end").Append(n).Append("\n\t@start")  
 .Append(n).Append(":\n\t@start").Append(n).Append("\_").Append(n)  
 .Append(":\n\tmov eax,0\n\t@end").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*FUNCTION*.ValueOfIt()))  
 {  
 amount = 0;  
 key = \_varsForGeneration  
 [keyValuePair.Key]  
 [stringListEntry.Key]  
 [0];  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*COMMA*.ValueOfIt()))  
 {  
 foreach (var listEntry in \_varsForGeneration[key])  
 {  
 for (var f = 0; f < listEntry.Value.Count; f++)  
 {  
 var l = listEntry.Value[f];  
 if (amount == f)  
 {  
 outcomeBlock.Append("\tmov ").Append(l).Append("\_").Append(key)  
 .Append(", eax\n");  
 }  
 }  
 }  
  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 amount++;  
 break;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*CR\_BRACKET*.ValueOfIt()))  
 {  
 foreach (var listEntry in \_varsForGeneration[key])  
 {  
 for (var f = 0; f < listEntry.Value.Count; f++)  
 {  
 var l = listEntry.Value[f];  
 if (f == 2)  
 {  
 outcomeBlock.Append("\tmov ").Append(l).Append("\_").Append(key)  
 .Append(",eax \n");  
 }  
 }  
 }  
  
 \_varsForGeneration[keyValuePair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 outcomeBlock.Append("\tcall ").Append(key).Append("\n\tmov eax, solution\_").Append(key)  
 .Append("\n");  
 break;  
 }  
  
 if (flag) outcomeBlock.Append("@exit:\n");  
 if (!keyValuePair.Key.Equals("return"))  
 outcomeBlock.Append("\tmov ")  
 .Append(keyValuePair.Key)  
 .Append("\_")  
 .Append(stringListEntry.Key)  
 .Append(",eax\n");  
 else outcomeBlock.Append("\tmov solution\_").Append(stringListEntry.Key).Append(",eax\n");  
 }  
  
 appendix++;  
 }  
  
 appendix++;  
 outcomeBlock.Append("\tfn MessageBox,0,").Append("str$(solution\_main)," + "Sliusarenko")  
 .Append(", MB\_OK\n\tret\n").Append(keyValuePair.Key).Append(" endp\n\n");  
 }  
 else  
 {  
 outcomeMasm.Append(keyValuePair.Key).Append(" proc\n");  
 foreach (var keyValPair in valTokensForGeneration)  
 {  
 foreach (var stringListEntry in keyValPair.Value)  
 {  
 if (stringListEntry.Value.Count != 0)  
 {  
 var flag = false;  
 if (stringListEntry.Value[0].Equals(TokenPattern.*CONST\_INT*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov eax,")  
 .Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [0])  
 .Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*VARIABLE*.ValueOfIt()))  
 {  
 outcomeBlock.Append("\tmov eax,")  
 .Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [0])  
 .Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*QUEST\_MARK*.ValueOfIt()))  
 {  
 appendix++;  
 outcomeMasm.Append("\tcmp eax,1\n\tjne @false").Append(appendix).Append("\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*EQUAL\_L*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,")  
 .Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tje @start").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @end").Append(n)  
 .Append("\n\t@start").Append(n).Append(":\n\tmov eax,1\n\t@end").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*NOT\_EQUAL*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx, {")  
 .Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tje @true").Append(n)  
 .Append("\n\tcmp eax,ebx\n\tje @true").Append(n)  
 .Append("\n\t@true").Append(n).Append(":\n\tmov eax,0\n\t@exit").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*COLON*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tjmp @exit\n\t@false").Append(appendix).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 flag = true;  
 }  
  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*ADDITION*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tadd eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*SUBTRACTION*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tsub eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*MULTIPLY*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx, ").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmul ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*DIVISION*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ecx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcdq\n\tidiv ecx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*GT*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tjg @bigger").Append(n).Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_bigger")  
 .Append(n).Append("\n\t@bigger").Append(n)  
 .Append(":\n\tmov eax,1\n\t@exit\_bigger").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LT*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tcmp eax,ebx\n\tjl @lesser").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_lesser").Append(n)  
 .Append("\n\t@lesser").Append(n).Append(":\n\tmov eax,1\n\t@exit\_lesser")  
 .Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*RIGHT\_SHIFT*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmov cl,ebx\n\tshr eax,cl\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_XOR*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("}\n\txor eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*MODULO*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ecx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcdq\n\tidiv ecx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_OR*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tor eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*BITWISE\_AND*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tand eax,ebx\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*GTE*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,ebx\n\tjge @big\_eq").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_big\_eq").Append(n)  
 .Append("\n\t@big\_eq").Append(n).Append(":\n\tmov eax,1\n\t@exit\_big\_eq")  
 .Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*OR*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,0\n\tje @start")  
 .Append(n).Append("\n\tjmp @end").Append(n)  
 .Append("\n\t@start").Append(n).Append(":\n\tmov eax,ebx\n\t@end").Append(n)  
 .Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LTE*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,ebx\n\tjle @les\_eq").Append(n)  
 .Append("\n\tmov eax,0\n\tjmp @exit\_les\_eq")  
 .Append(n).Append(":\n\tmov eax,1\n\t@exit\_les\_eq").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*LEFT\_SHIFT*.ValueOfIt()))  
 {  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1])  
 .Append("\n\tmov cl,ebx\n\tshl eax,cl\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*AND*.ValueOfIt()))  
 {  
 n++;  
 outcomeMasm.Append("\tmov ebx,").Append(\_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [1]).Append("\n\tcmp eax,0\n\tje @start").Append(n)  
 .Append("\n\tcmp ebx,0\n\tje @start").Append(n).Append("\_").Append(n)  
 .Append("\n\tmov eax,ebx\n\tjmp @end").Append(n).Append("\n\t@start")  
 .Append(n).Append(":\n\t@start").Append(n).Append("\_").Append(n)  
 .Append(":\n\tmov eax,0\n\t@end").Append(n).Append(":\n");  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*FUNCTION*.ValueOfIt()))  
 {  
 amount = 0;  
 key = \_varsForGeneration  
 [keyValPair.Key]  
 [stringListEntry.Key]  
 [0];  
 stringListEntry.Value.RemoveAt(0);  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*COMMA*.ValueOfIt()))  
 {  
 foreach (var listEntry in \_varsForGeneration[key])  
 {  
 for (var f = 0; f < listEntry.Value.Count; f++)  
 {  
 var l = listEntry.Value[f];  
 if (amount == f)  
 {  
 outcomeMasm.Append("\tmov ").Append(l).Append("\_").Append(key)  
 .Append(", eax\n");  
 }  
 }  
 }  
  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 amount++;  
 }  
 else if (stringListEntry.Value[0].Equals(TokenPattern.*CR\_BRACKET*))  
 {  
 foreach (var listEntry in \_varsForGeneration[key])  
 {  
 for (var index = 0; index < listEntry.Value.Count; index++)  
 {  
 var value = listEntry.Value[index];  
 if (index == 2)  
 {  
 outcomeMasm.Append("\tmov ").Append(value).Append("\_").Append(key)  
 .Append(",eax \n");  
 }  
 }  
 }  
  
 \_varsForGeneration[keyValPair.Key][stringListEntry.Key].RemoveAt(0);  
 stringListEntry.Value.RemoveAt(0);  
 outcomeMasm.Append("\tcall ").Append(key).Append("\n\tmov eax, solution\_")  
 .Append(key)  
 .Append("\n");  
 }  
  
 if (flag) outcomeMasm.Append("@exit:\n");  
 if (!keyValPair.Key.Equals("return"))  
 outcomeBlock.Append("\tmov ")  
 .Append(keyValPair.Key)  
 .Append("\_")  
 .Append(stringListEntry.Key)  
 .Append(",eax\n");  
 else  
 outcomeMasm.Append("\tmov solution\_").Append(stringListEntry.Key).Append(",eax\n");  
 }  
  
 appendix++;  
 }  
  
 appendix++;  
 outcomeMasm.Append("\tret\n").Append(keyValPair.Key).Append(" endp\n\n");  
 }  
 }  
 }  
  
 return *GenerateUsingTemplate*(vars, outcomeBlock, outcomeMasm);  
 }  
  
 private static string *GenerateUsingTemplate*(StringBuilder vars, StringBuilder outcomeBlock,  
 StringBuilder outcomeMasm)  
 {  
 var template = new StringBuilder();  
 template.Append(".386\n")  
 .Append(".model flat, stdcall\n")  
 .Append("option Dictionary:none\n")  
 .Append("include \\masm32\\include\\masm32rt.inc\n")  
 .Append(".data\n")  
 .Append(vars).Append("\n")  
 .Append(".code\n")  
 .Append("start:\n")  
 .Append(outcomeBlock).Append("\n")  
 .Append(outcomeMasm).Append("\n")  
 .Append("invoke main\n")  
 .Append("invoke ExitProcess, 0\n")  
 .Append("END start");  
 return template.ToString();  
 }  
 }  
**}**

**TokenPattern.cs**

using System;  
  
namespace KR  
{  
 public class TokenPattern  
 {  
 public static readonly TokenPattern *INT* = new("int", () => "type\_int");  
 public static readonly TokenPattern *FLOAT* = new("float", () => "type\_float");  
 public static readonly TokenPattern *DOUBLE* = new("double", () => "type\_double");  
 public static readonly TokenPattern *CHAR* = new("char", () => "type\_char");  
 public static readonly TokenPattern *STRING* = new("String", () => "type\_String");  
 public static readonly TokenPattern *VOID* = new("void", () => "type\_void");  
  
 public static readonly TokenPattern *DIVISION* = new("/", () => "division");  
 public static readonly TokenPattern *ADDITION* = new("+", () => "addition");  
 public static readonly TokenPattern *SUBTRACTION* = new("-", () => "minus");  
 public static readonly TokenPattern *MULTIPLY* = new("\*", () => "multiply");  
  
 public static readonly TokenPattern *BITWISE\_AND* = new("&", () => "&");  
 public static readonly TokenPattern *BITWISE\_OR* = new("|", () => "|");  
 public static readonly TokenPattern *BITWISE\_XOR* = new("^", () => "^");  
 public static readonly TokenPattern *MODULO* = new("%", () => "%");  
 public static readonly TokenPattern *LEFT\_SHIFT* = new("<<", () => "<<");  
 public static readonly TokenPattern *RIGHT\_SHIFT* = new(">>", () => ">>");  
 public static readonly TokenPattern *AND* = new("&&", () => "&&");  
 public static readonly TokenPattern *OR* = new("||", () => "||");  
  
 public static readonly TokenPattern *EQUAL* = new("=", () => "equal");  
 public static readonly TokenPattern *LT* = new("<", () => "<");  
 public static readonly TokenPattern *LTE* = new("<=", () => "<=");  
 public static readonly TokenPattern *GT* = new(">", () => ">");  
 public static readonly TokenPattern *GTE* = new(">=", () => ">=");  
  
 public static readonly TokenPattern *OR\_BRACKET* = new("(", () => "open\_round\_bracket");  
 public static readonly TokenPattern *CR\_BRACKET* = new(")", () => "close\_round\_bracket");  
 public static readonly TokenPattern *OС\_BRACE* = new("{", () => "open\_curly\_brace");  
 public static readonly TokenPattern *CС\_BRACE* = new("}", () => "close\_curly\_brace");  
  
 public static readonly TokenPattern *COMMA* = new(",", () => "comma");  
 public static readonly TokenPattern *QUEST\_MARK* = new("?", () => "question mark");  
 public static readonly TokenPattern *COLON* = new(":", () => "colon");  
 public static readonly TokenPattern *SEMICOLON* = new(";", () => "semicolon");  
  
 public static readonly TokenPattern *MAIN* = new("main", () => "identifier");  
 public static readonly TokenPattern *RETURN* = new("return", () => "return\_keyword");  
  
 public static readonly TokenPattern *VARIABLE* = new("variable", () => "variable");  
 public static readonly TokenPattern *CONST\_INT* = new("int\_constant", () => "int\_constant");  
 public static readonly TokenPattern *CONST\_FLOAT* = new("float\_constant", () => "float\_constant");  
  
 public static readonly TokenPattern *INVALID* = new("invalid", () => "invalid");  
 public static readonly TokenPattern *FUNCTION* = new("function", () => "function");  
 public static readonly TokenPattern *EQUAL\_L* = new("==", () => "logic equal");  
 public static readonly TokenPattern *NOT\_EQUAL* = new("!=", () => "not equal");  
 public static readonly TokenPattern *NOT* = new("!", () => "not");  
  
 public readonly string Field;  
 private readonly Func<string> \_valueOfIt;  
  
 private TokenPattern(string field, Func<string> valueOfIt)  
 {  
 Field = field;  
 \_valueOfIt = valueOfIt;  
 }  
  
 public string ValueOfIt()  
 {  
 return \_valueOfIt.Invoke();  
 }  
 }  
}

**UnsuspectedTokenException.cs**

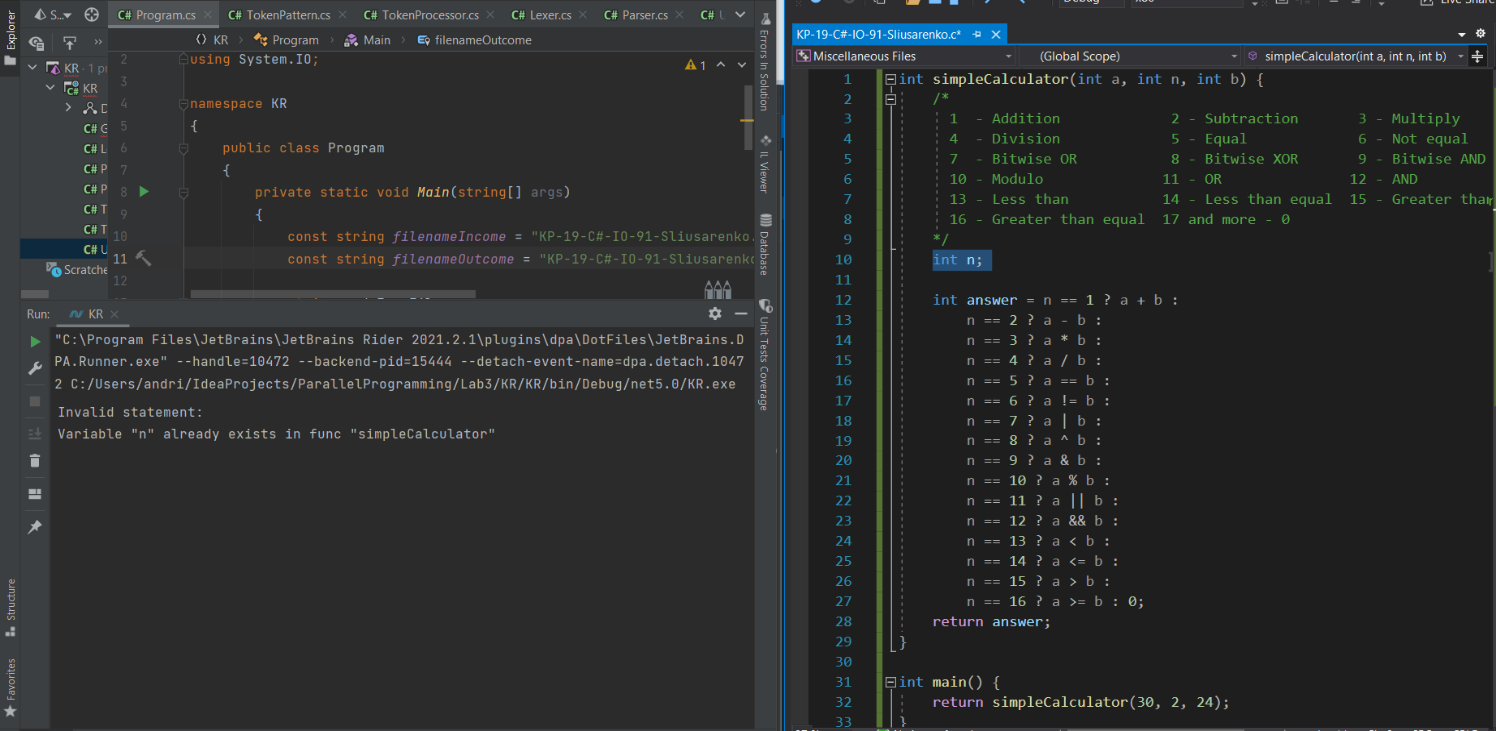
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace KR  
{  
 public class UnsuspectedTokenException : Exception  
 {  
 public UnsuspectedTokenException(List<object> list, int index)  
 {  
 var whereItHappened = new StringBuilder();  
 string signValue = null;  
   
 for (var i = 0; i < list.Count; i++)  
 {  
 var item = list[i];  
 if (i == index) {  
 signValue = (string) item;  
 if (signValue.Equals(TokenPattern.*SEMICOLON*.Field)) {  
 whereItHappened.Append(item).Append("\n");  
 } else {  
 whereItHappened.Append(item);  
 }  
 } else if (item.Equals(TokenPattern.*SEMICOLON*.Field) ||   
 item.Equals(TokenPattern.*CС\_BRACE*.Field)) {  
 whereItHappened.Append(item).Append("\n");  
 } else if (item.Equals(TokenPattern.*OС\_BRACE*.Field)) {  
 whereItHappened.Append("\n").Append(item).Append("\n");  
 } else {  
 whereItHappened.Append(" ").Append(item);  
 }  
 }  
  
 Console.*WriteLine*("Unsuspected symbol " + signValue);  
 Console.*WriteLine*("Caused at " + whereItHappened);  
 Console.*ReadLine*();  
 }  
 }  
  
 public class InvalidStatementException : Exception  
 {  
 public InvalidStatementException(string message)  
 {  
 Console.*WriteLine*(message);  
 Console.*ReadLine*();  
 }  
 }  
}

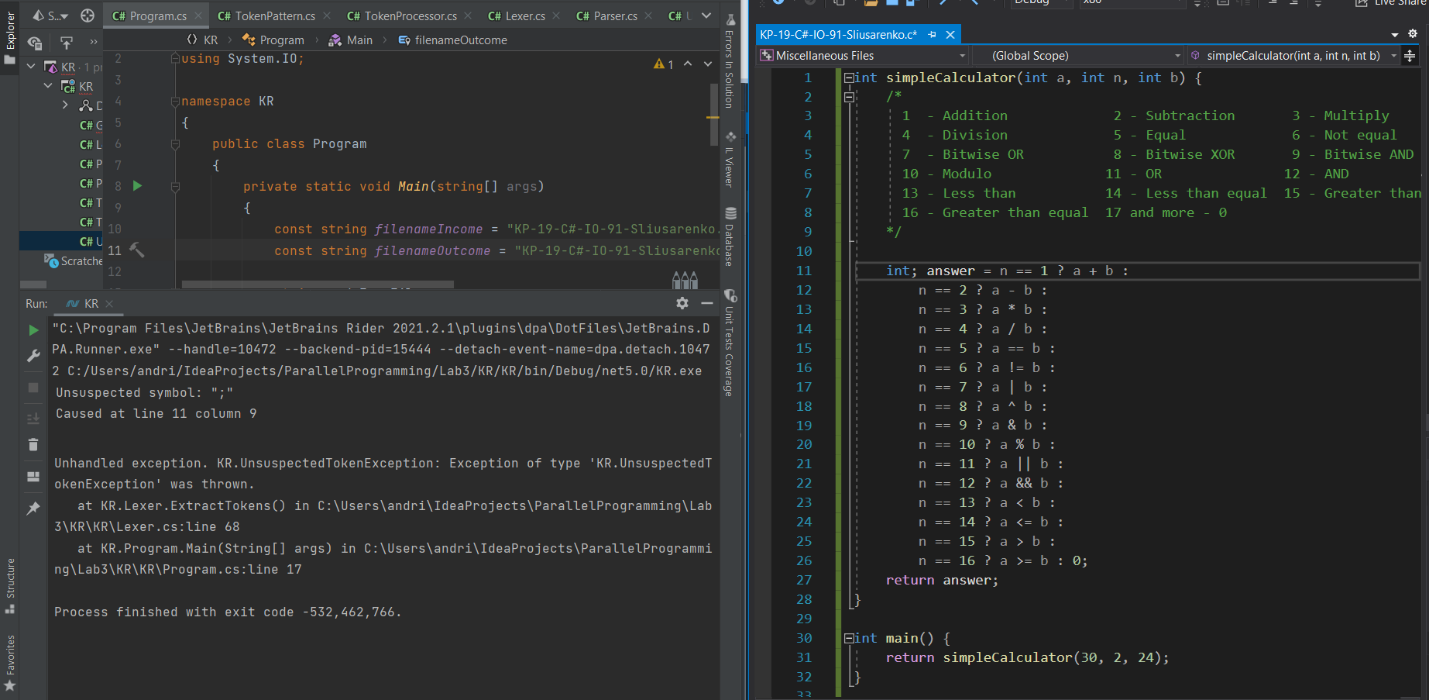
**TokenProcessor.cs**

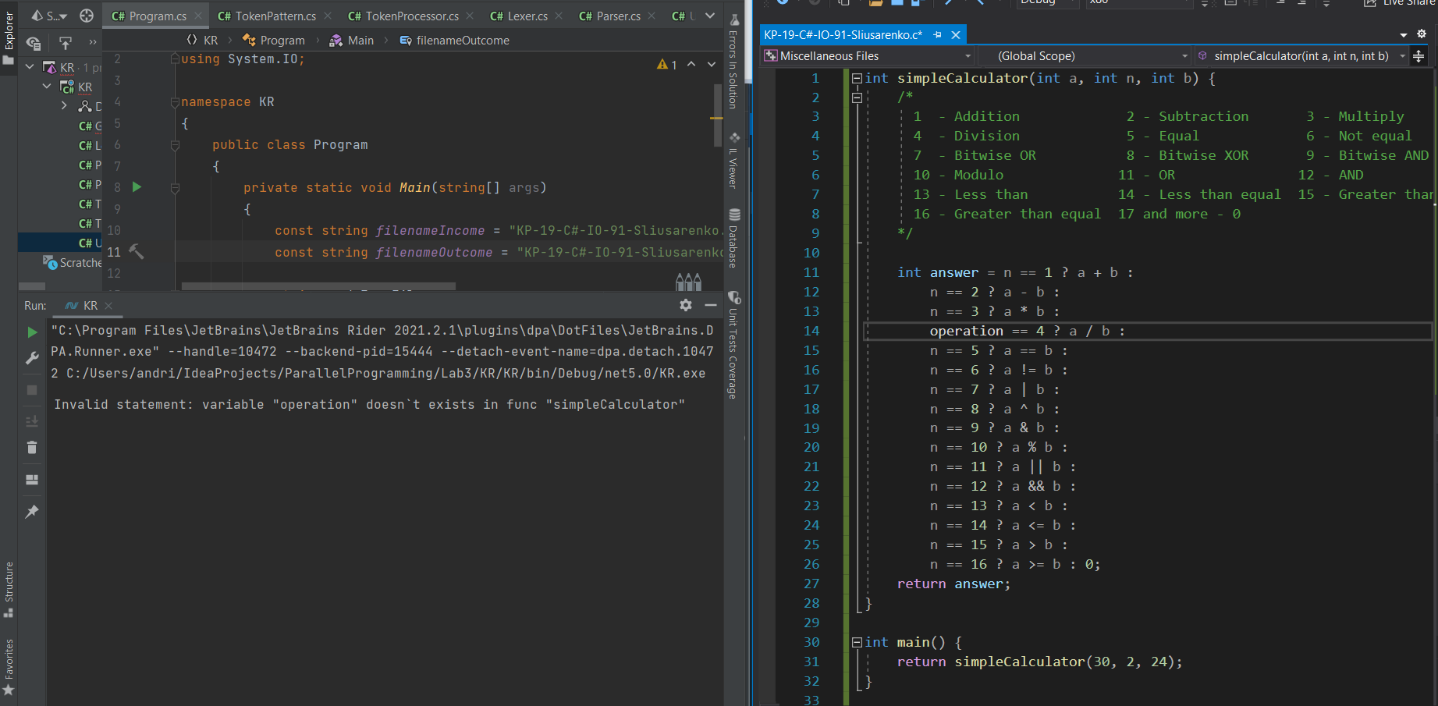
using System;  
using System.Collections.Generic;  
using System.Linq;  
  
namespace KR  
**{** public class TokenProcessor  
 {  
 private readonly List<TokenPattern> \_allTokenPatterns = new()  
 {  
 TokenPattern.*GT*, TokenPattern.*INT*, TokenPattern.*LT*, TokenPattern.*CHAR*, TokenPattern.*MAIN*, TokenPattern.*VOID*,  
 TokenPattern.*COLON*, TokenPattern.*COMMA*, TokenPattern.*EQUAL*, TokenPattern.*FLOAT*, TokenPattern.*DOUBLE*, TokenPattern.*MODULO*,  
 TokenPattern.*RETURN*, TokenPattern.*STRING*, TokenPattern.*CС\_BRACE*, TokenPattern.*OС\_BRACE*, TokenPattern.*ADDITION*,  
 TokenPattern.*DIVISION*, TokenPattern.*MULTIPLY*, TokenPattern.*BITWISE\_OR*,  
 TokenPattern.*CR\_BRACKET*, TokenPattern.*OR\_BRACKET*, TokenPattern.*QUEST\_MARK*, TokenPattern.*SEMICOLON*, TokenPattern.*BITWISE\_AND*,  
 TokenPattern.*BITWISE\_XOR*, TokenPattern.*SUBTRACTION* };  
  
  
  
 public TokenPattern MatchItemWithTokenPattern(string item)  
 {  
 foreach (var pattern in \_allTokenPatterns)  
 {  
 if (pattern.Field.Equals(item))  
 {  
 return pattern;  
 }  
  
 if (CheckIfTextSeemsLikeNum(item))  
 {  
 if (char.*IsLetter*(item[0]))  
 {  
 return TokenPattern.*VARIABLE*;  
 }  
 return TokenPattern.*CONST\_INT*;  
 }  
 if (item.Contains(".") && CheckIfTextSeemsLikeDigit(item.Replace(".", ""))) return TokenPattern.*CONST\_FLOAT*;  
 }  
  
 return item.Length == 0 ? null : TokenPattern.*INVALID*;  
 }  
   
 private bool CheckIfTextSeemsLikeNum(string text)  
 {  
 return text.Split("").SelectMany(s => s.ToCharArray()).All(char.*IsDigit*);  
 }  
 private bool CheckIfTextSeemsLikeDigit(string s)  
 {  
 foreach (char c in s)  
 {  
 if (!char.*IsDigit*(c))  
 {  
 return false;  
 }  
 }  
 return true;  
 }  
 }  
**}**

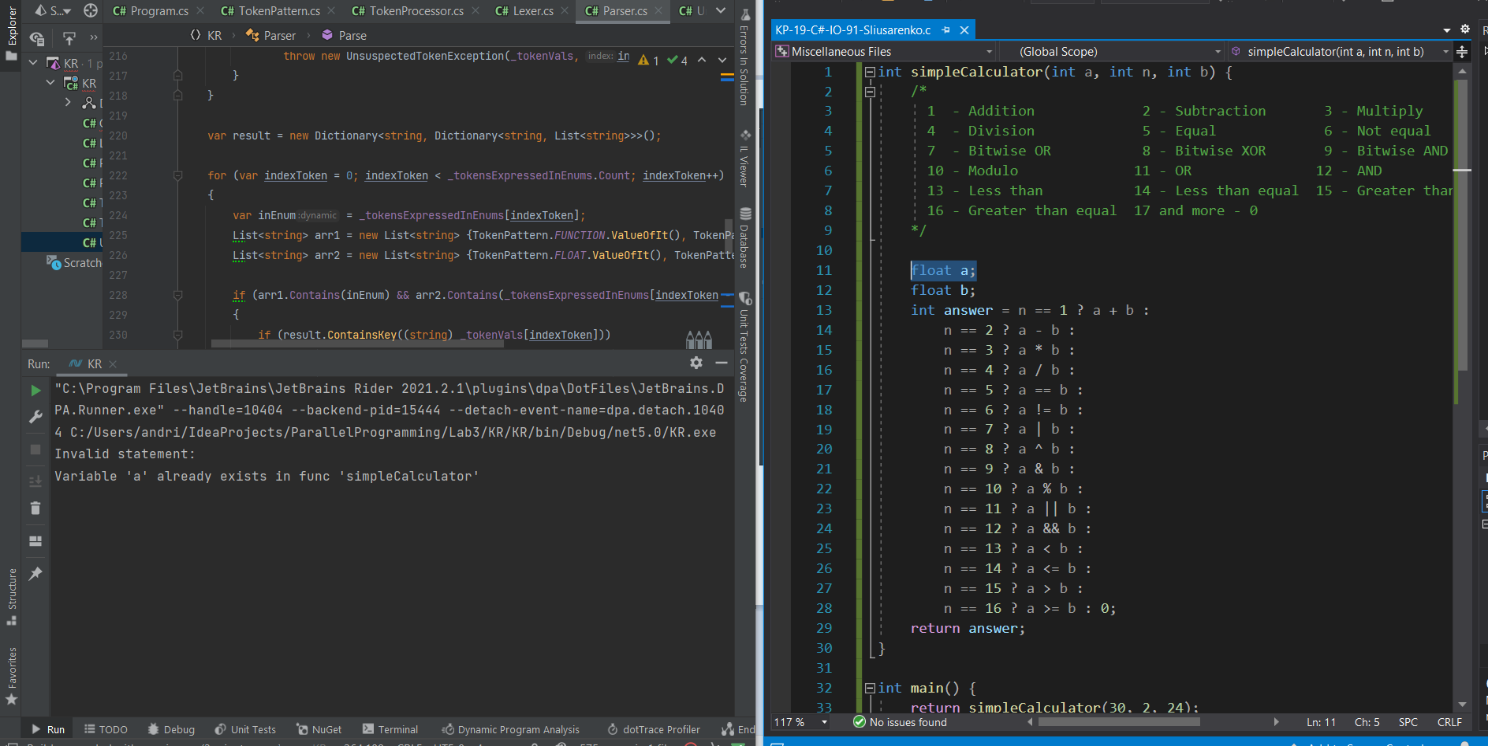
7. Результати перевірки працездатності програми

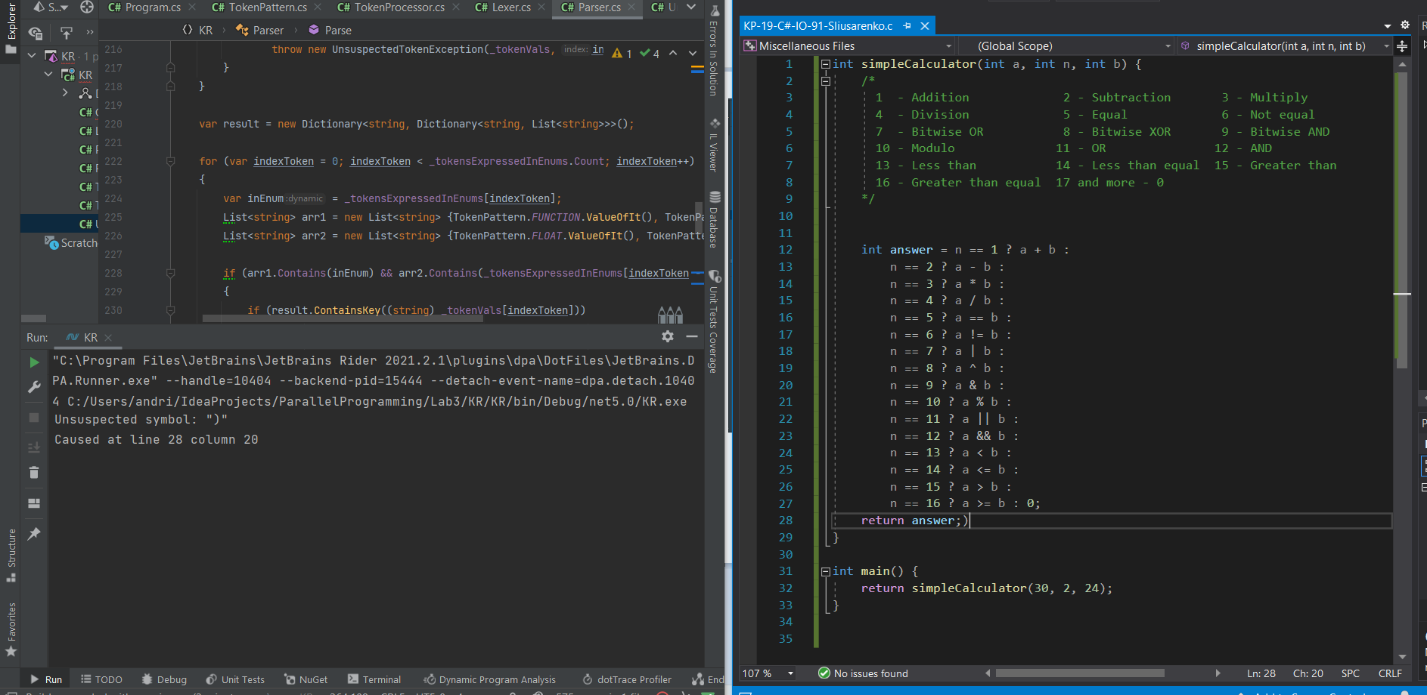
7.1 Приклади виконання з помилками



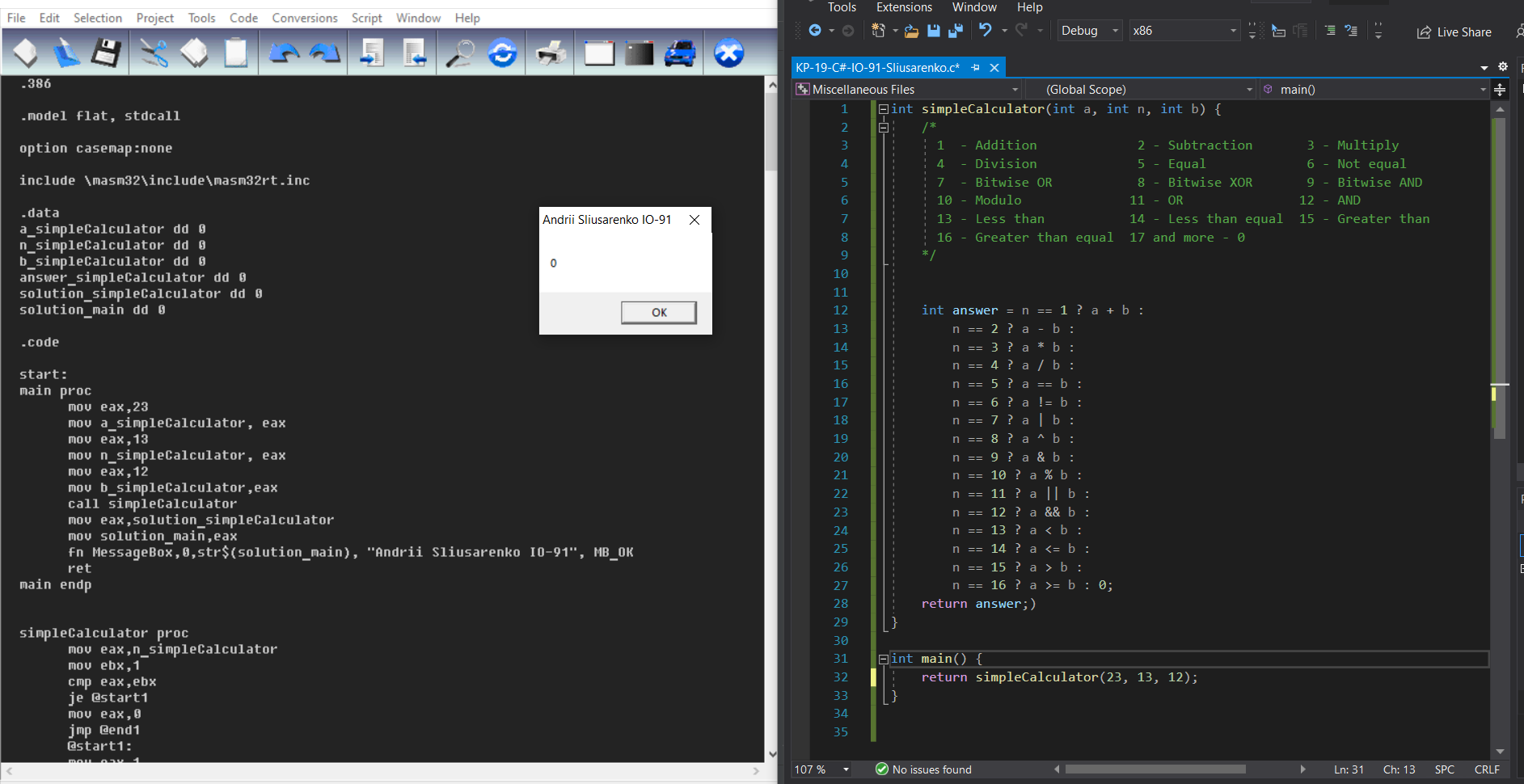






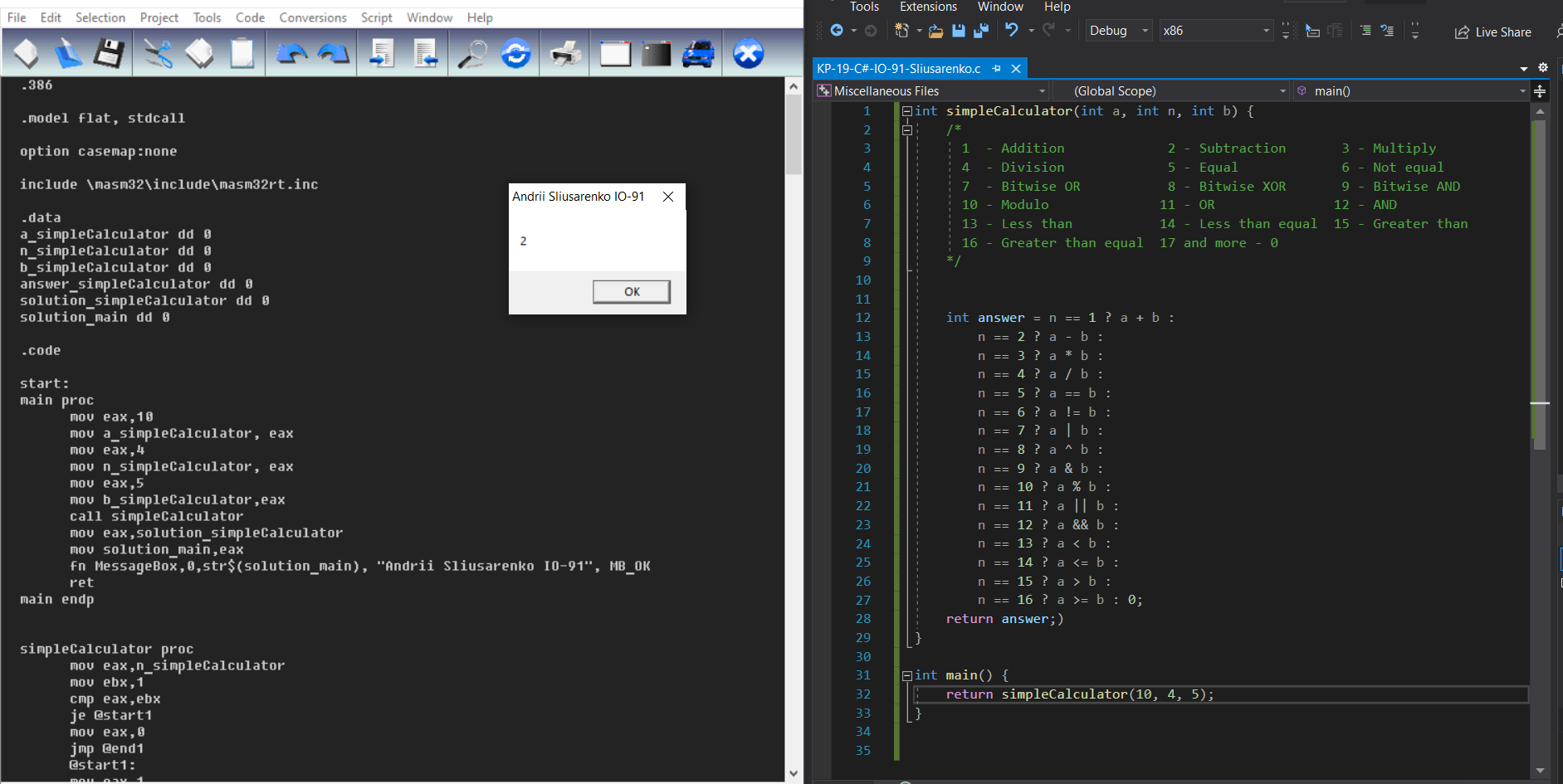


7.2 Приклади виконання без помилок:



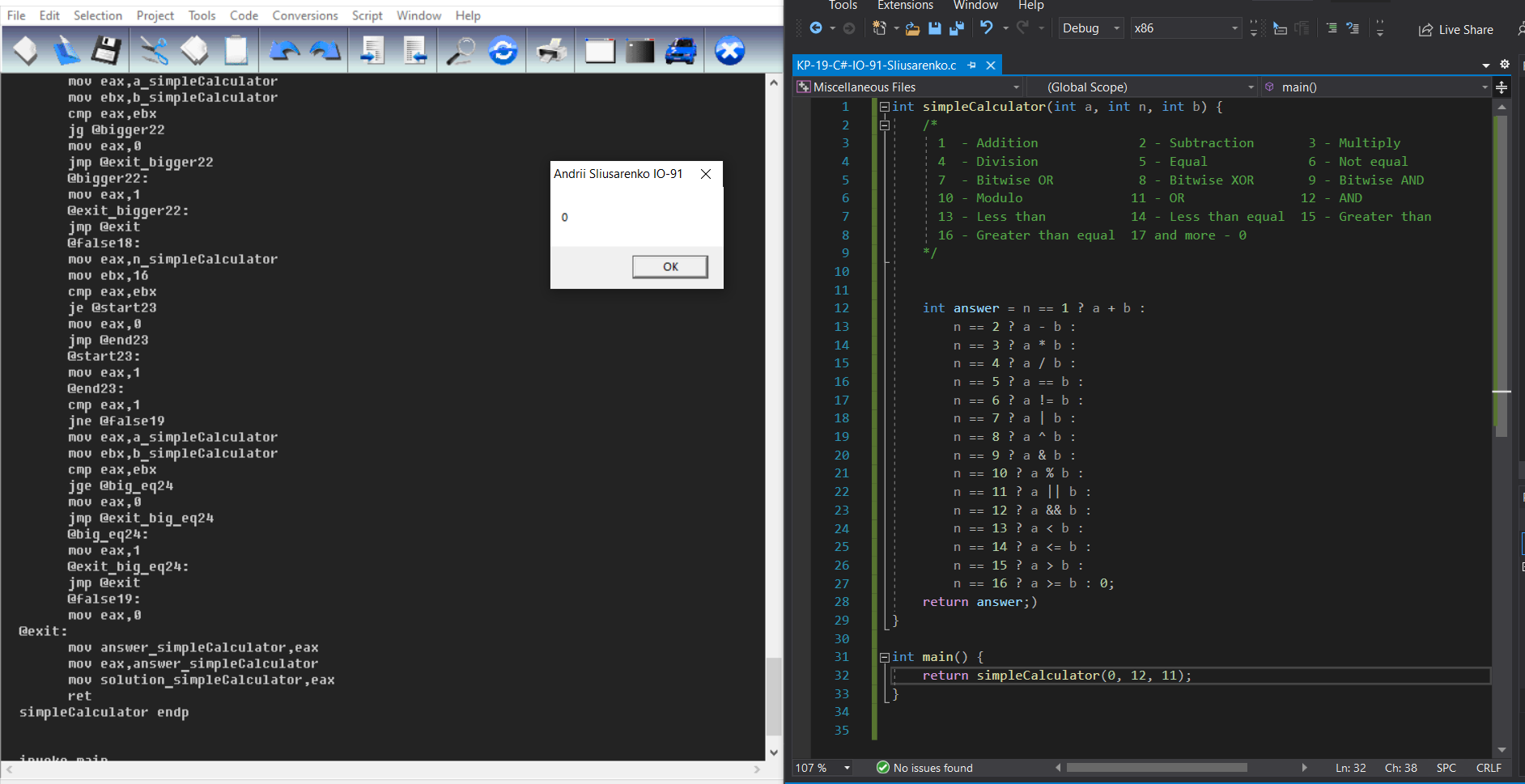
Перевірка: перший операнд – 23, другий – 12, оператор – менше.

23 <12 = 0, тобто false – результат вірний.



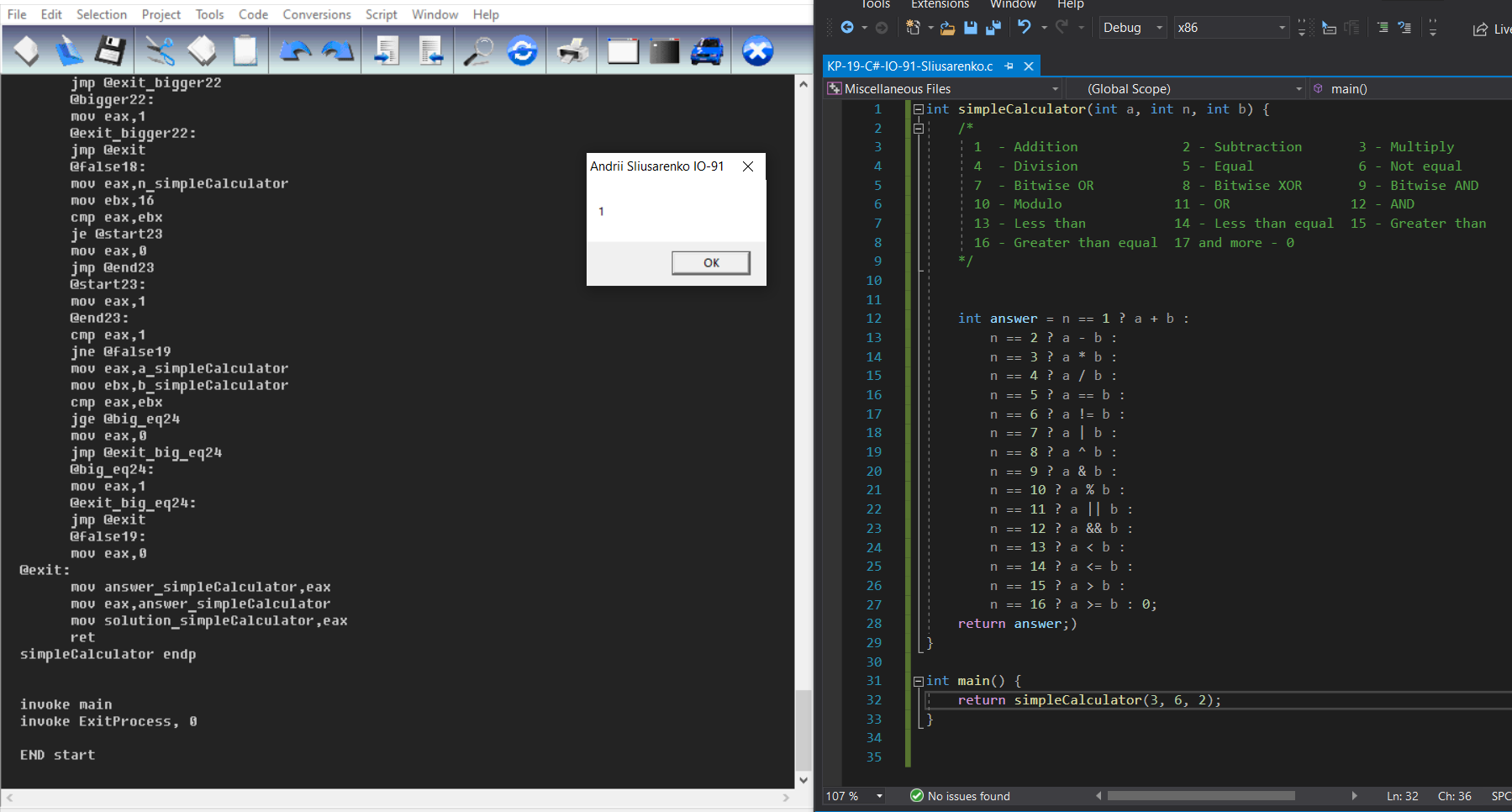
Перевірка: перший операнд – 10, другий – 5, операція – ділення.

10 / 5 = 2 – результат вірний.



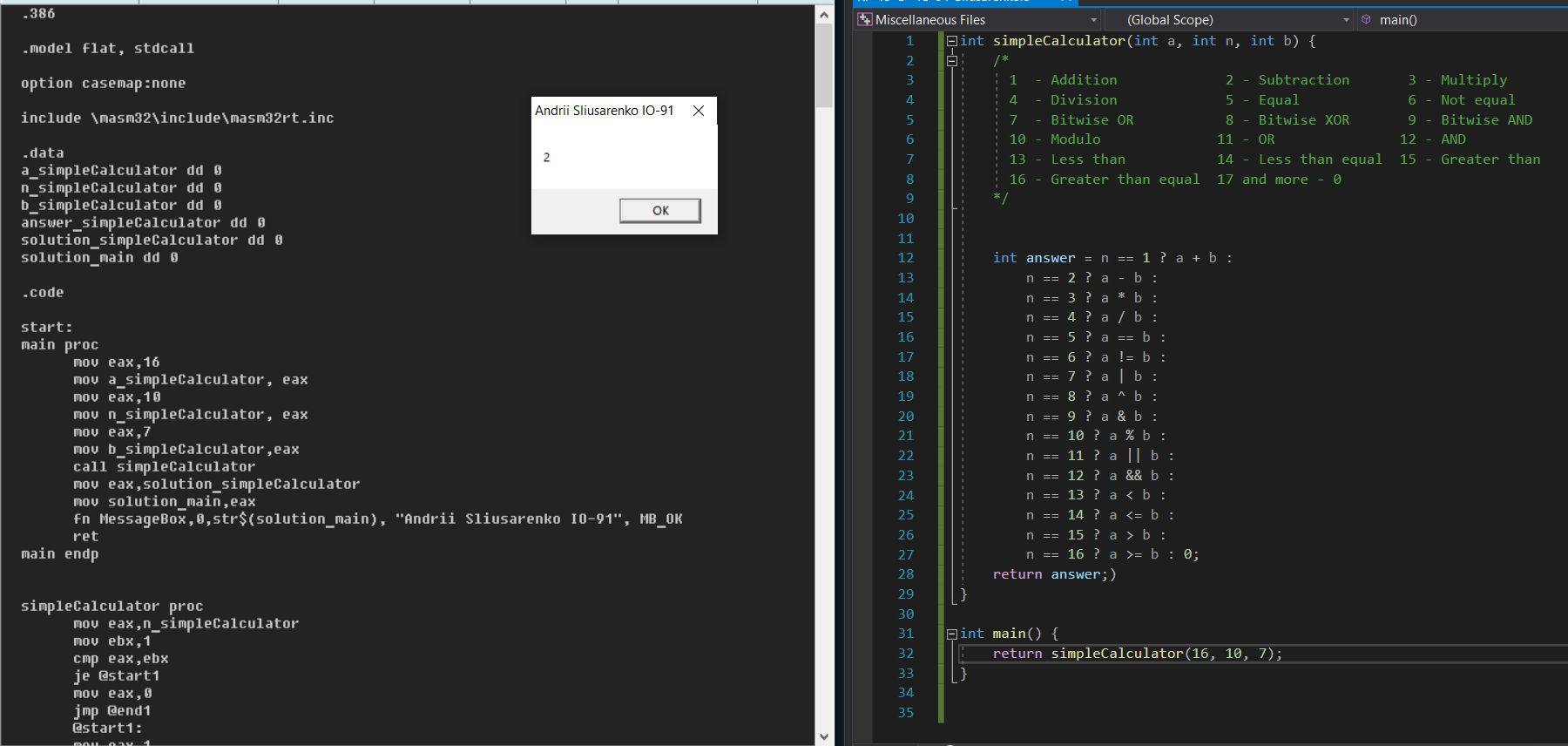
Перевірка: перший операнд – 0, другий – 11, операція – І (логічне).

0 && 11 = 0, тобто false – результат вірний.



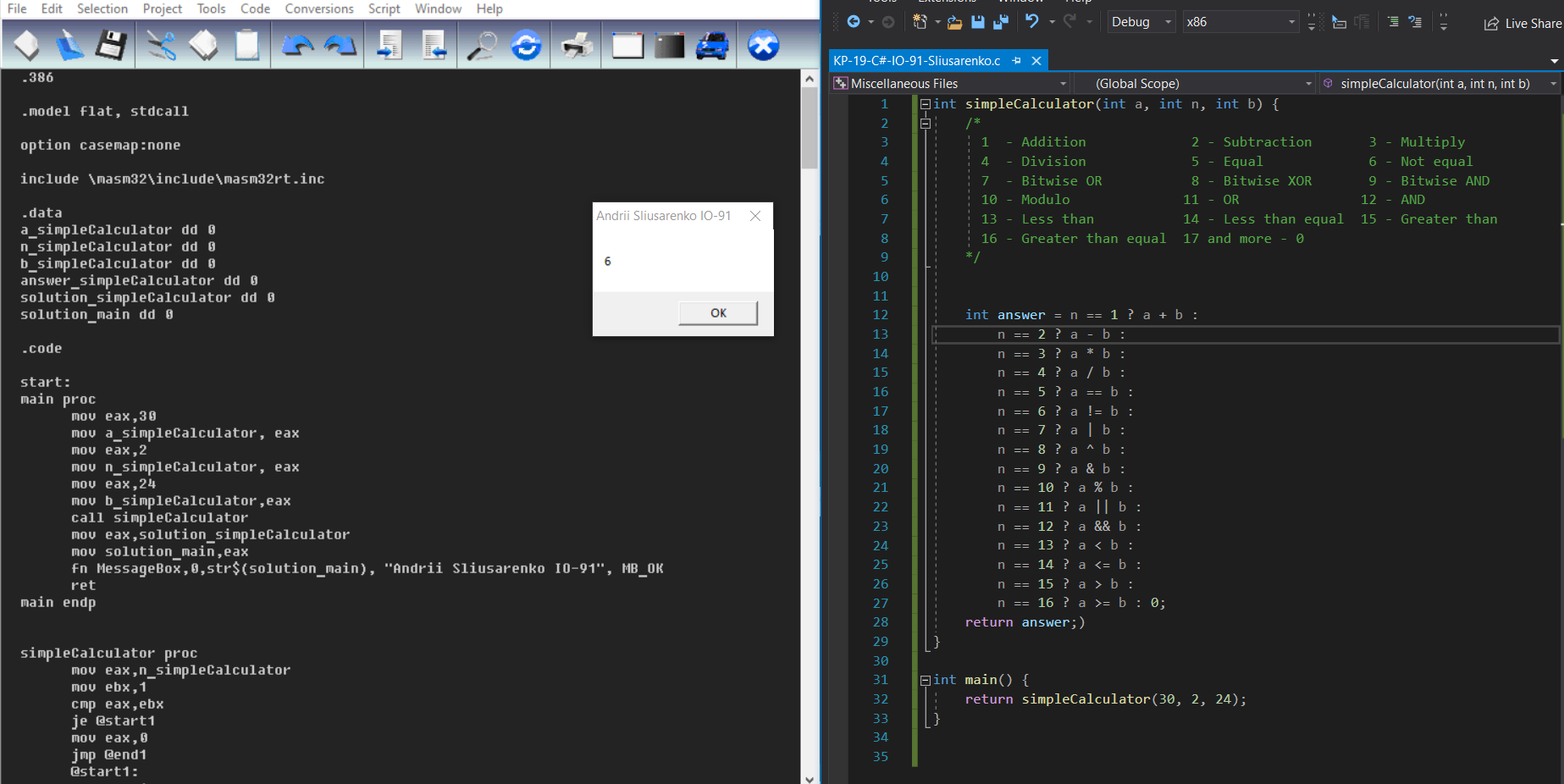
Перевірка: перший операнд – 3, другий – 2, оператор – Не рівності.

3 != 2 = 1, тобто true – результат вірний.



Перевірка: перший операнд – 16, другий – 7, операція – Mod(остача від ділення).

16 % 7 = 2 – результат вірний.



Перевірка: перший операнд – 30, другий – 24, операція – віднімання

30 – 24 = 6 – результат вірний.

8. Висновок

У ході виконання курсової роботи було розроблено програму компілятора мови програмування С, реалізовано алгоритм за варіантом, а саме простий калькулятор. Розроблений калькулятор підтримує операції додавання, віднімання, множення, ділення, остача від ділення, оператори: менше або рівне, більше або рівне, тотожності(рівності), логічні І та АБО, побітові І, АБО, виключне АБО.

9. Список використаної літератури

1. Compiler Construction. The Art of Niklaus Wirth. URL: <https://www.researchgate.net/profile/Hanspeter-Moessenboeck/publication/221350529_Compiler_Construction_-_The_Art_of_Niklaus_Wirth/links/0deec5213875d84735000000/Compiler-Construction-The-Art-of-Niklaus-Wirth.pdf>
2. Соломатин Д.И., Копытин А.В., Другалев А.И. Основы синтаксического разбора, построение синтаксических анализаторов. Воронеж, 2014. — 57 с.
3. Windows Assembly Programming Tutorial. URL: <https://doc.lagout.org/operating%20system%20/Windows/winasmtut.pdf>
4. Yale N. Patt, Sanjay J. Patel. Introduction To Computing Systems: From Bits And Gates To C And Beyond. Singapore, 2005. URL: <https://www.academia.edu/34254842/INTRODUCTION_TO_COMPUTING_SYSTEMS_FROM_BITS_AND_GATES_TO_C_AND_BEYOND_SECOND_EDITION_International_Edition_2005>
5. Орлов С.А. Теория и практика языков программирования. Учебник для вузов. — СПб.: Питер, 2013. — 688 с.