|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ | | | | | | | | | | |
|  | | | |  | | |  | | | |
| ФГБОУ ВПО «Пермский государственный национальный исследовательский университет» | | | | | | | | | | |
|  | |  | | | | | | |  | |
|  | | ОТЧЕТ  по дисциплине «Формальные грамматики и методы трансляции» | | | | | | |  | |
|  | | |  | | | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Васильевых М.Г. \_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021 | | | |  | Проверил  ассистент кафедры МОВС  Пономарёв Ф.А. \_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021 | | | |  |
|  | |  | | | | | | |  | |
|  | | Пермь 2021 | | | | | | |  | |

Оглавление

Оглавление

[Модуль ввода вывода 3](#_Toc90160151)

[Описание 3](#_Toc90160152)

[Проектирование 3](#_Toc90160153)

[Реализация 4](#_Toc90160154)

[Тестирование 7](#_Toc90160155)

[Лексический анализатор 8](#_Toc90160156)

[Описание 8](#_Toc90160157)

[Проектирование 8](#_Toc90160158)

[Реализация программы 9](#_Toc90160159)

[Тестирование 15](#_Toc90160160)

[Синтаксический анализатор 23](#_Toc90160161)

[Описание 23](#_Toc90160162)

[Проектирование 23](#_Toc90160163)

[Реализация 24](#_Toc90160164)

[Тестирование 34](#_Toc90160165)

[Семантический анализатор 38](#_Toc90160166)

[Описание 38](#_Toc90160167)

[Проектирование 38](#_Toc90160168)

[Реализация 39](#_Toc90160169)

[Тестирование 44](#_Toc90160170)

# Модуль ввода вывода

## Описание

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их анализатору.

## Проектирование

На данном этапе нам необходимо входной файл разбить на лексемы, в соответствии с синтаксисом языка, а именно его ключевыми словами и специальными символами, определив для каждой лексемы её тип и сохраняя их в определённый буфер.

Концепция данного модуля заключается на обработке отдельных групп символов. Набор всех допустимых символов, был разбит на 3 подмножества.

1. Множество букв в нижнем и вернем регистре и нижнее подчёркивание
2. Множество цифр
3. Множество символов

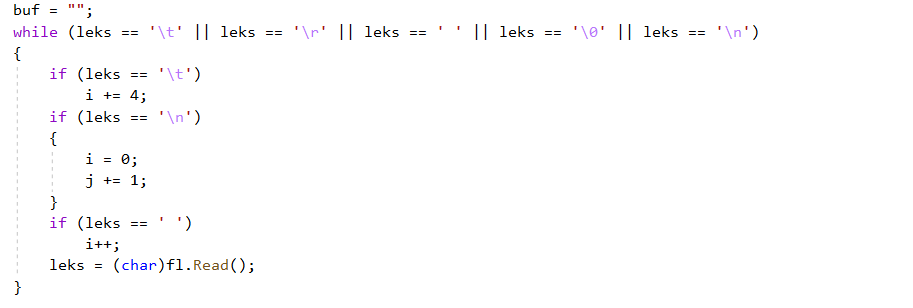
Для работы предполагается система считывающего модуля и буфера символов.

Что бы понять, что мы считали мы используем несколько групп ввода, а именно:

1. Ввод числа, целого и вещественного типа
2. Ввод символьной строки
3. Ввод символьной константы
4. Ввод строковой константы
5. Ввод специальных символов

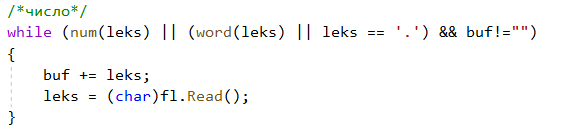
## Реализация

Модуль ввода вывода сначала будет проверять символы, связанные с переносом строки, табуляцию, пробелы то есть если при начале работы модуля ловим символ ‘\n’ то IO модуль его пропустит и считает следующий.

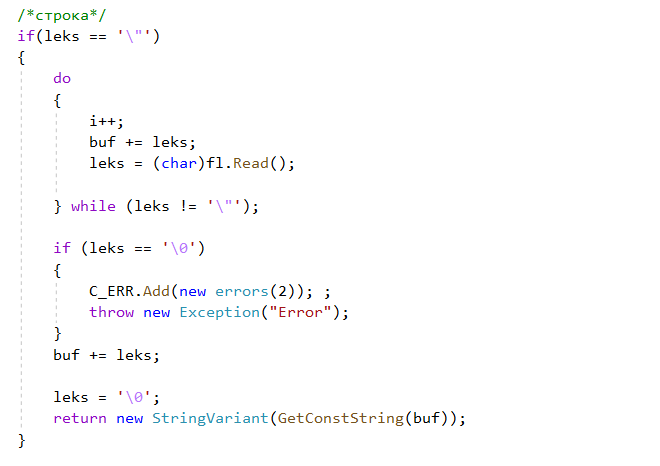


Зная что ни один идентификатор не может начинаться с цифры, начнём обработку лексемы с того число ли это.

А именно будем считать число это набор цифры, буквы или точка если перед ними есть цифры.

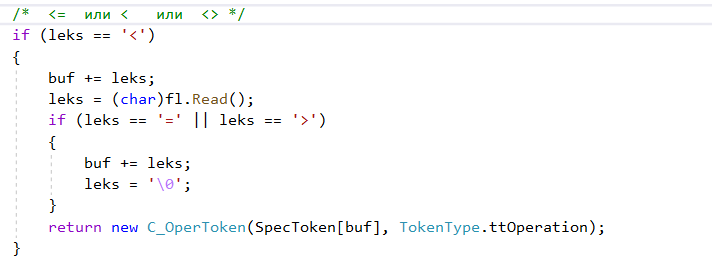


Считывание строковой константы

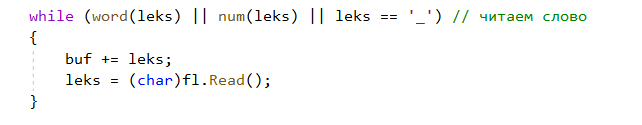


Считывание парных спец символов: ‘:=’, ‘<=’, ‘>=’, ‘<>’.





Читаем «слова»



## Тестирование

На данном этапе тестирование не имеет смысла, так как модуль ввода вывода тесно связан с лексическим анализатором.

# Лексический анализатор

## Описание

Лексический анализатор формирует символы исходной программы и строит их внутреннее представление. Кроме того, сканер распознает и исключает комментарии, которые не нужны для дальнейшей трансляции.

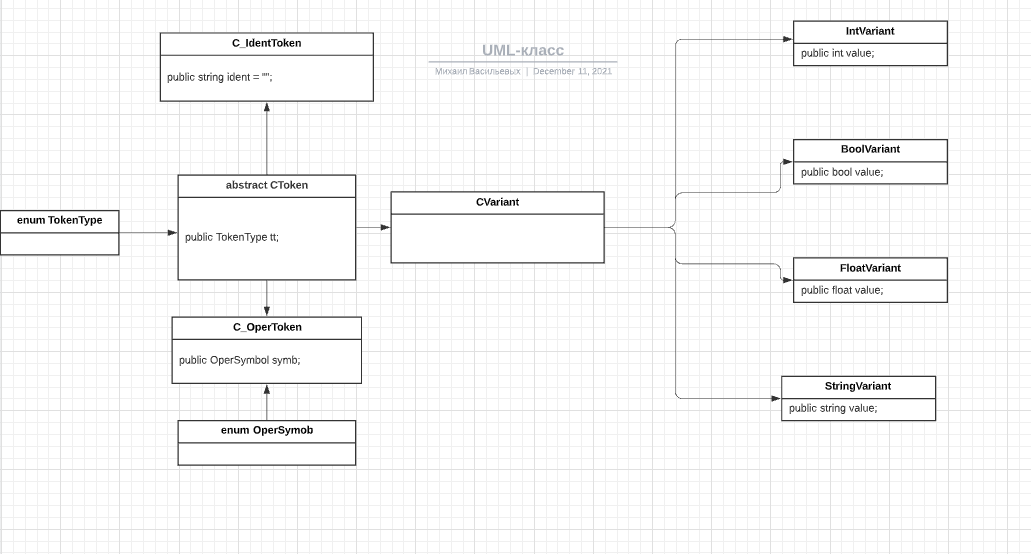
## Проектирование

Нам необходимо понять, как создать структуру классов и ошибок

Для каждой группы символов нужно будет создать свой отдельный класс, который будет связан иерархически

Я остановился на данной реализации, где есть абстрактный класс родитель для всех остальных типов (идентификатора, спец символа, константы).

И для символа констант я создал такой же родительский класс для типов констант(целого, логического, вещественного строкового)



Так же стоит проблема в определении ошибок.

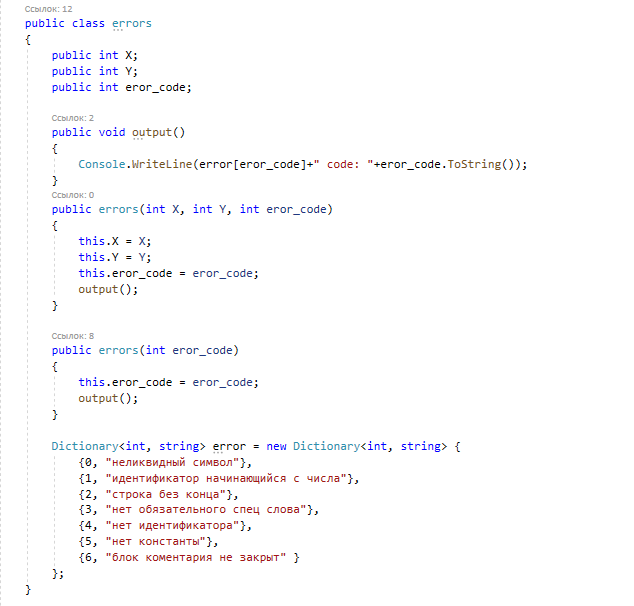
Я остановился на следующих ошибках:

1. Запрещенный символ, символ которого нет в языке.
2. Неправильный формат идентификатора, начинается с цифры.
3. Неправильный формат символьной константы, пустая константа или не закрытая.
4. Нарушен формат для вещественного числа, наличие в нём более одной точки например 2..0
5. Не завершен блок комментариев

## Реализация программы

Все спец слова должны хранится в специальном словаре, где ключ это строка слова, а значение его код(базовые типы как integer, float(real), bool, string не будут хранится в этом словаре а будут определятся как идентификаторы, это является особенностью языка Паскаль)

Так же был создан класс для хранения ошибок, а так же внутри него словарь для ошибок и их описаний.



И возникла проблема нейтрализация ошибок.

Для разрешения этой проблемы я прибег к использованию, исключений, но так как наш компилятор однопроходный работу с исключениями созданными в лексическом анализаторе я вынес в блок с синтаксическим анализатором

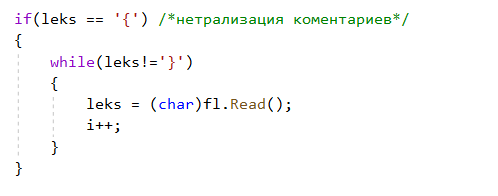


Классификация лексем на символы

Так как лексический анализатор содержится внутри модуля ввода вывода, некоторые их действия в программе напрямую взаимосвязаны, а иногда один и тот же программный блок выполняет действия для обоих модулей, поэтому некоторые фрагменты кода будут рассмотрены повторно.

Нейтрализация комментариев

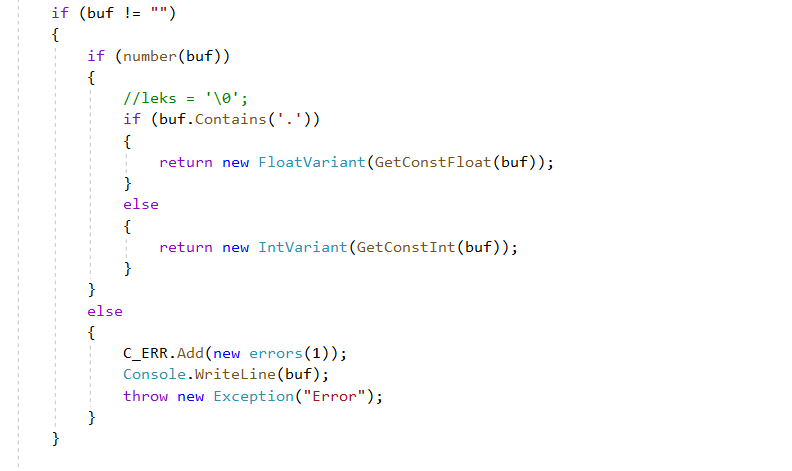
Модуль нейтрализует комментарий, пройдя циклом, пока не встретит модуль закрывающий комментарий.

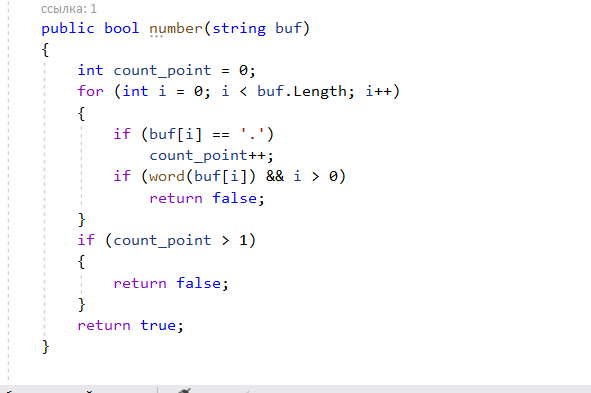


Затем проверим, считали ли мы что то в буфер и является ли это числом?

Если да то нам нужно понять, а какое число: целое или вещественное?

В определении этого факта нам поможет тот факт, что у вещественных чисел есть точка.

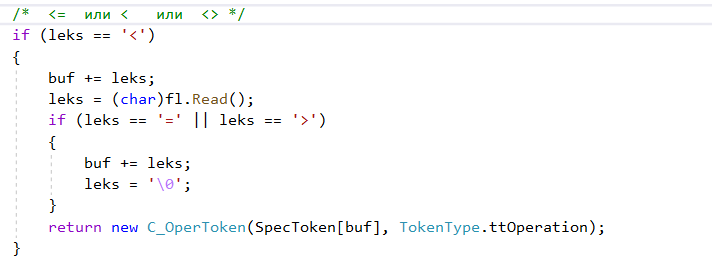


Для определения число ли это, был написан метод 

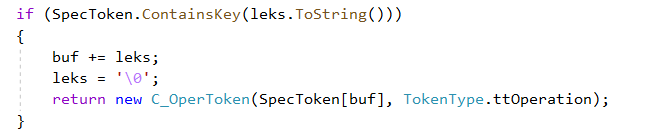
Проверка на спец символ

Проверка парных спец символов: ‘:=’, ‘<=’, ‘>=’, ‘<>’.

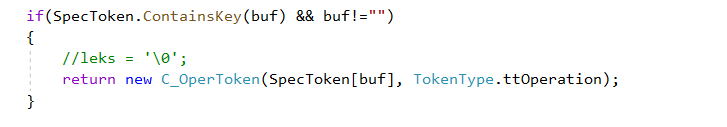




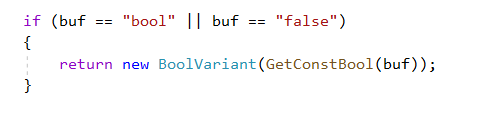
Не парные символы найдем, проверив их по словарю



Спец слова инициализируем по словарю



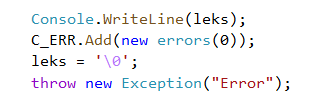
Логические константы



Идентификаторы



Иначе невалидный символ

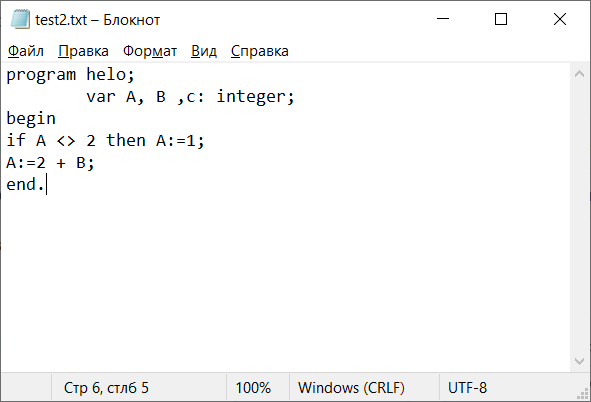


## Тестирование

Стоит обратить внимание, что листинг идет не с тем как введена проверяемая программа в файл, а по её кодам, но так как мы используем перечисления то выводятся имена этих кодов.

Лексически корректная программа, содержащая все виды символов

Входные данные:



Ожидаемый результат:

Sprogram

helo

Ssemicon

Svar

A

Scomma

B

Scomma

c

Scolon

integer

Ssemicon

Sbegin

Sif

A

Snotequals

ttConst

Sthen

A

Sassigment

ttConst

Ssemicon

A

Sassigment

ttConst

Sadd

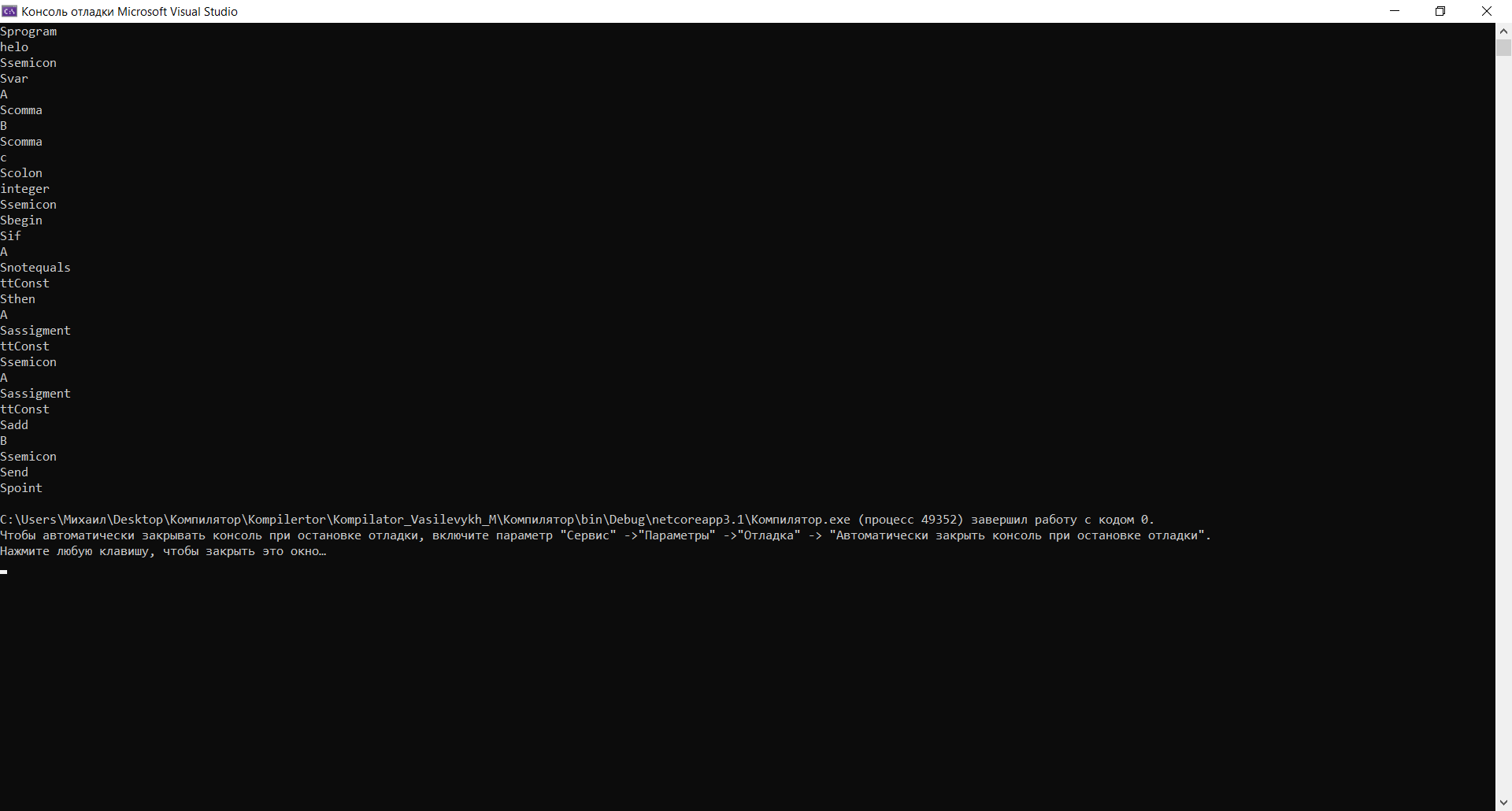
B

Ssemicon

Send

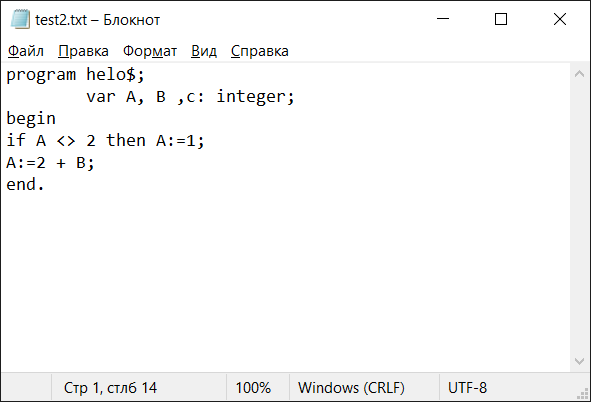
Spoint

Выходной результат:



Имеется ошибка, символ не входящий в область допустимых символов, так же проверяется нейтрализация ошибки

Входные данные:



Ожидаемый результат:

Sprogram

helo

$

неликвидный символ code: 0

Ssemicon

Svar

A

Scomma

B

Scomma

c

Scolon

integer

Ssemicon

Sbegin

Sif

A

Snotequals

ttConst

Sthen

A

Sassigment

ttConst

Ssemicon

A

Sassigment

ttConst

Sadd

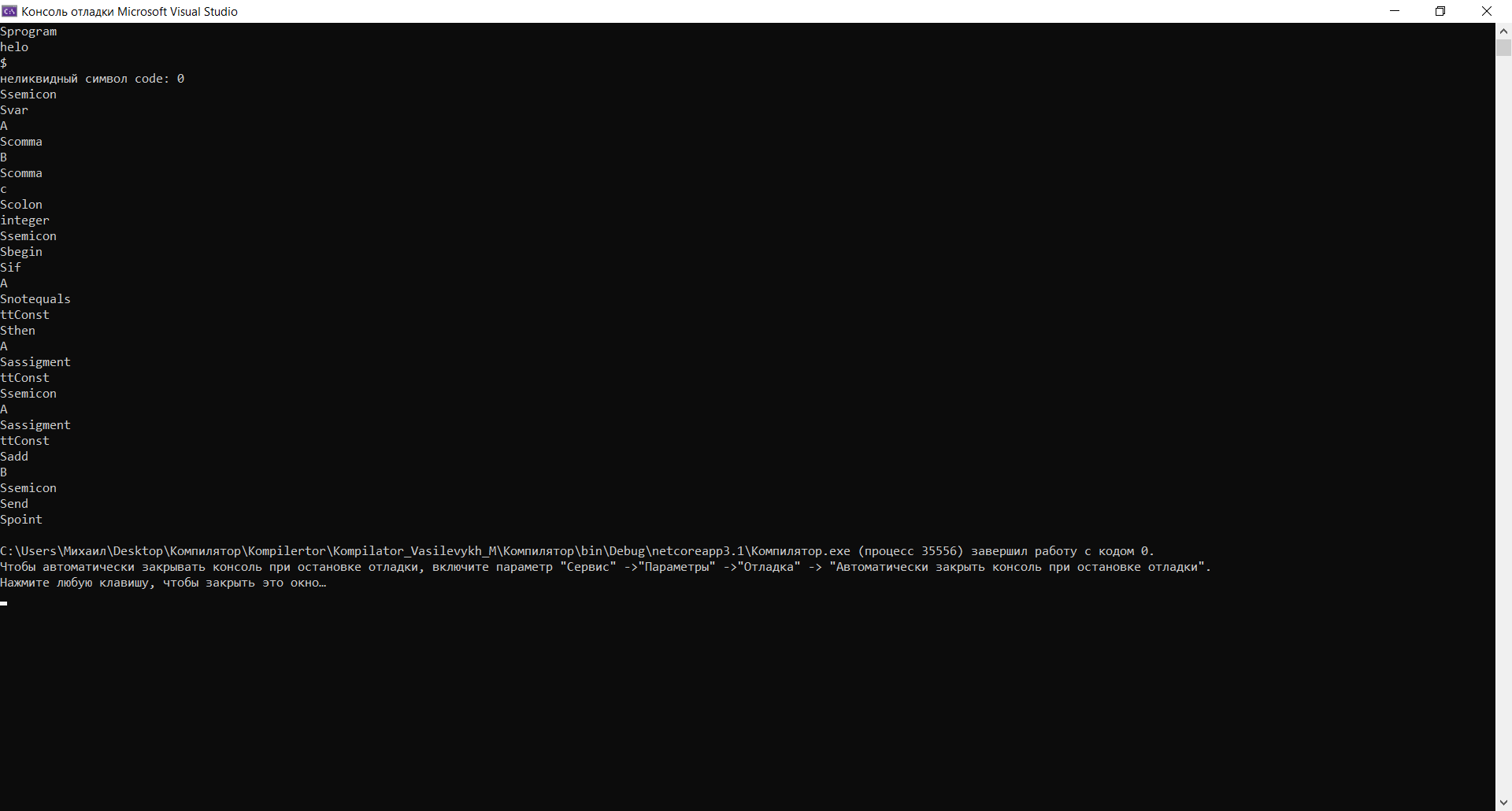
B

Ssemicon

Send

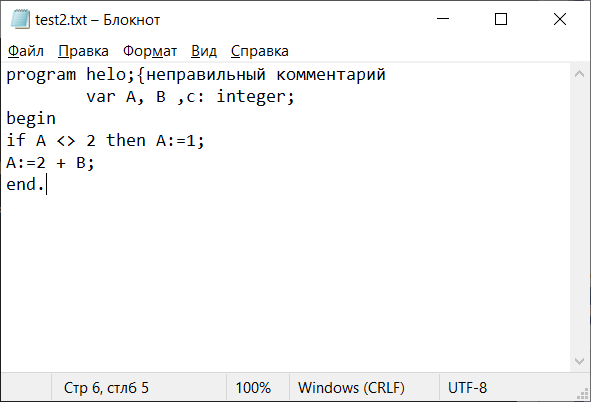
Spoint

Выходной результат:



Ошибка блок комментария не закрыт

Входные данные:



Ожидаемый результат:

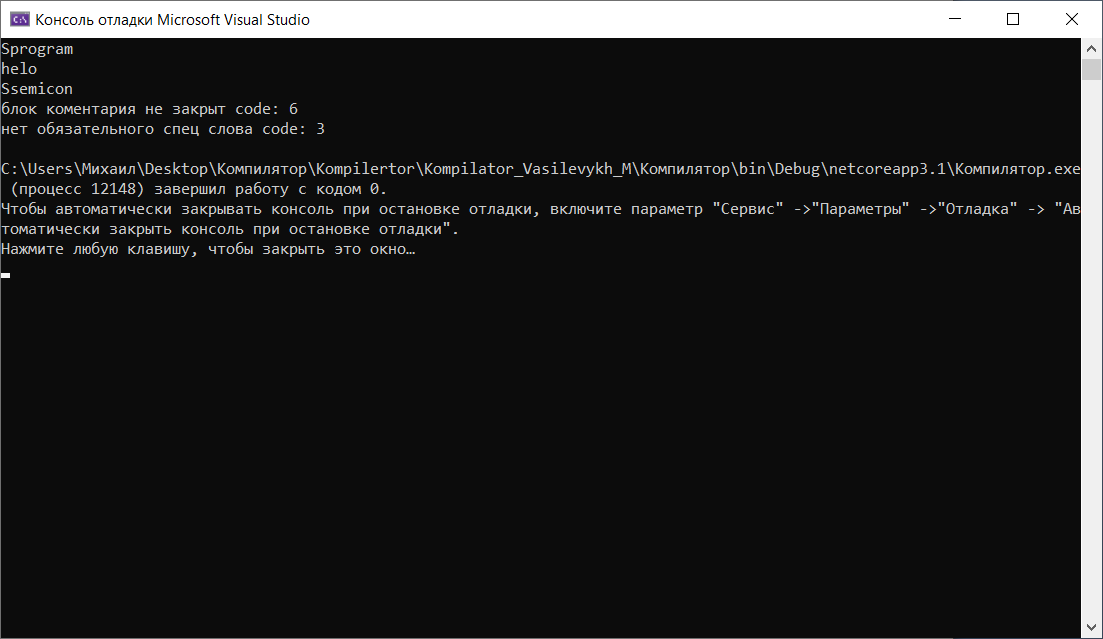
Sprogram

helo

Ssemicon

блок коментария не закрыт code: 6

Выходные данные



Выходные данные отличаются от предполагаемых, так как все модули тестируются совестно. В нашем случае сработало обработка ошибки синтаксического анализатора совестно с лексическим.

# Синтаксический анализатор

## Описание

Синтаксический анализатор проверяет правильность и порядок использование символов языка.

## Проектирование

Синтаксический анализатор строится на правил использования символов языка, в нашем случае используются БНФ.

Ниже приведены все БНФ использованные мной:

<программа>::=program <имя>;<блок>.

<имя>::=<буква>{<буква>|<цифра>}

<блок>::=<раздел констант><раздел переменных><оператор>

<раздел констант>::=<пусто>|const <определение константы>;{<определение константы>;}

<определение константы>::=<имя>=<константа>

<константа>::=<число без знака>|<знак><число без знака>|

<имя константы>|<знак><имя константы>|<строка>

<число без знака>::=<целое без знака>|<вещественное без знака>

<целое без знака>::=<цифра>{<цифра>}

<вещественное без знака>::=<целое без знака>.<цифра>{<цифра>}|<целое без знака>.<цифра>{<цифра>}E<порядок>|<целое без знака>E<порядок>

<порядок>::=<целое без знака>|<знак><целое без знака>

<знак>::=+|-

<имя константы>::=<имя>

<строка>::='<символ>{<символ>}'

<раздел переменных>::= var <описание однотипных переменных>;{<описание однотипных переменных>;}|<пусто>

<описание однотипных переменных>::=<имя>{,<имя>}:<тип>

<оператор>::=<простой оператор>|<сложный оператор>

<простой оператор>::=<оператор присваивания>|<пустой оператор>

<оператор присваивания>::=<имя>:=<выражение>

<выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение>

<операция отношения>::==|<>|<|<=|>=|>|in

<простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}

<аддитивная операция>::=+|-|or

<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}

<мультипликативная операция>::=\*|/|div|mod|and

<множитель>::=<имя>|<константа без знака>|(<выражение>)

<константа без знака>::=<число без знака>|<строка>|<имя константы>|nil

<пустой оператор>::=<пусто>

<пусто>::=

<сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>

<составной оператор>::= begin <оператор>{;<оператор>} end

<выбирающий оператор>::= if <выражение> then <оператор>|if <выражение> then <оператор> else <оператор>

<оператор цикла>::= while <выражение> do <оператор>

Для обработки событий, что в наша программа не соответствует БНФ я буду использовать исключения и обрабатывать их на верхних уровнях БНФ.

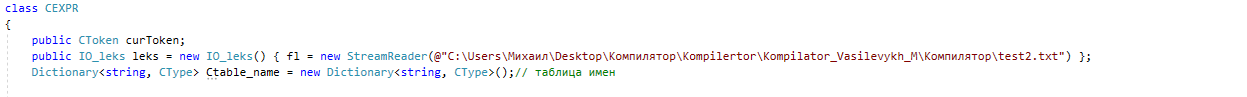
Будем считать что на БНФ приходит всегда обрабатываемый символ а выходит символ для следующей БНФ, это позволит избежать использования буферов.

Принятие решений при выборе альтернатив я положу на этом же уровне БНФ анализируя текущий символ.

Так как синтаксический анализатор тесно взаимодействует с семантическим, мы будем рассматривать в этой части отчёта части семантического анализатора.

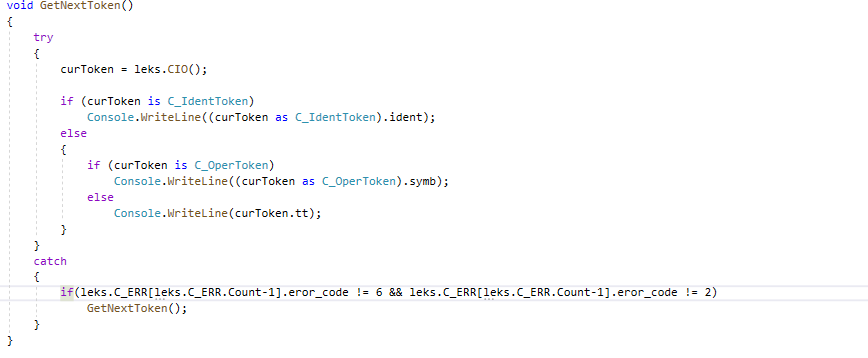
## Реализация

Я создал класс для реализации синтаксического анализатора, а все БНФ запрограммированы как его методы.

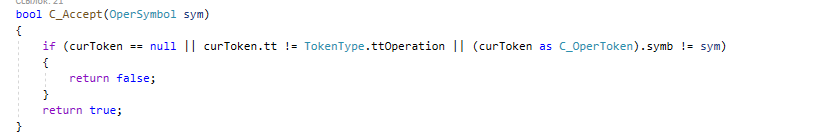


Поле curToken, обрабатываемый символ.

Leks, поле для доступа к лексическому анализатору.



Метод для получения следующего символа для обработки.



Проверка соответствия спец символа

Все методы запрограммированы в соответствии БНФ

**public** **void** **C\_Programm**()

{

GetNextToken();

**if** (!C\_Accept(OperSymbol.Sprogram))

leks.C\_ERR.Add(**new** errors(**3**));

GetNextToken();

**if** (curToken.tt != TokenType.ttIdent)

leks.C\_ERR.Add(**new** errors(**4**));

GetNextToken();

**if** (!C\_Accept(OperSymbol.Ssemicon))

leks.C\_ERR.Add(**new** errors(**3**));

GetNextToken();

C\_Block();

**if** (!C\_Accept(OperSymbol.Spoint))

leks.C\_ERR.Add(**new** errors(**3**));

}

**void** **C\_Block**() // <блок>::=<раздел констант><раздел переменных><оператор>

{

**try**

{

C\_ConstArea();

}

**catch**

{

GetNextToken();

}

**try**

{

C\_VarArea();

}

**catch**

{

GetNextToken();

}

**try**

{

C\_Operator();

}

**catch**

{

GetNextToken();

}

}

**void** **C\_ConstArea**() // <раздел констант>::=<пусто>|const <определение константы>;{<определение константы>;}

{

**if** (C\_Accept(OperSymbol.Sconst))

{

C\_ConstDeter();

**if** (!C\_Accept(OperSymbol.Ssemicon))

**throw** **new** **Exception**();

GetNextToken();

**while** (**true**)

{

**try**

{

C\_ConstDeter();

}

**catch**

{

**break**;

}

**if** (!C\_Accept(OperSymbol.Ssemicon))

**throw** **new** **Exception**();

GetNextToken();

}

}

}

**void** **C\_ConstDeter**() // <определение константы>::=<имя>=<константа>

{

**if** (curToken.tt != TokenType.ttIdent)

{

**throw** **new** **Exception**();

}

GetNextToken();

**if** (!C\_Accept(OperSymbol.Sequals))

{

**throw** **new** **Exception**();

}

GetNextToken();

C\_Const();

}

**void** **C\_Const**() // <константа>::=<число без знака>|<знак><число без знака>|

//<имя константы>|<знак><имя константы>|<строка>

//<число без знака>::=<целое без знака>|<вещественное без знака>

//<целое без знака>::=<цифра>{<цифра>}

//<вещественное без знака>::=<целое без знака>.<цифра>{<цифра>}|< целое без знака>.<цифра>{<цифра>}E<порядок> |< целое без знака>E<порядок>

//<порядок>::=<целое без знака>|<знак><целое без знака>

//<знак>::=+|-

//<имя константы>::=<имя>

//<строка>::='<символ>{<символ>}'

{

**if** (C\_Accept(OperSymbol.Ssubtraction))

{

GetNextToken();

**if** (curToken.tt == TokenType.ttConst || curToken.tt == TokenType.ttIdent)

{

GetNextToken();

}

**else**

{

**throw** **new** **Exception**();

}

}

**else**

{

**if** (curToken.tt == TokenType.ttConst || curToken.tt == TokenType.ttIdent)

{

GetNextToken();

}

**else**

{

**throw** **new** **Exception**();

}

}

}

**void** **C\_VarArea**() //<раздел переменных>::= var <описание однотипных переменных>;{<описание однотипных переменных>;}|<пусто>

{

**if** (C\_Accept(OperSymbol.Svar))

{

GetNextToken();

C\_var();

**if** (!C\_Accept(OperSymbol.Ssemicon))

**throw** **new** **Exception**();

GetNextToken();

**while** (**true**)

{

**try**

{

C\_var();

}

**catch**

{

**break**;

}

**if** (!C\_Accept(OperSymbol.Ssemicon))

**throw** **new** **Exception**();

GetNextToken();

}

}

}

**void** **C\_var**() // <описание однотипных переменных>::=<имя>{,<имя>}:<тип>

{

**if** (curToken.tt != TokenType.ttIdent)

**throw** **new** **Exception**();

/\*имена переменных\*/

List<**string**> name = **new** List<**string**>();

name.Add((curToken **as** C\_IdentToken).ident);

GetNextToken();

**while** (C\_Accept(OperSymbol.Scomma))

{

GetNextToken();

**if** (curToken.tt != TokenType.ttIdent)

**throw** **new** **Exception**();

/\*имена переменных\*/

name.Add((curToken **as** C\_IdentToken).ident);

GetNextToken();

}

**if** (!C\_Accept(OperSymbol.Scolon))

**throw** **new** **Exception**();

GetNextToken();

**if** (curToken.tt != TokenType.ttIdent)

**throw** **new** **Exception**();

/\*\*/

CType t;

/\*добавление в таблицу имен\*/

**switch** ((curToken **as** C\_IdentToken).ident)

{

**case** "integer": t = **new** CIntType(); **break**;

**case** "float": t = **new** CFloatType(); **break**;

**case** "string": t = **new** CStringType(); **break**;

**case** "bool": t = **new** CBoolType(); **break**;

**default**: **throw** **new** Exception();

}

**foreach**(**string** egge **in** name)

{

Ctable\_name.Add(egge, t);

}

GetNextToken();

}

**void** **C\_Operator**() //<оператор>::=<простой оператор>|<сложный оператор>

{

**if**(curToken **is** C\_OperToken)

{

**switch**((curToken **as** C\_OperToken).symb)

{

**case** OperSymbol.Sbegin: C\_CompositeOperator(); **return**;

**case** OperSymbol.Sif:C\_ChoiseOperator(); **return**;

**case** OperSymbol.Swhile: C\_CycleOpertor();**return**;

}

}

C\_SimpleOperator();

}

**void** **C\_SimpleOperator**() // <простой оператор>::=<оператор присваивания>|<пустой оператор>

{

**try**

{

C\_AssigmentOperator();

}

**catch**

{

}

}

/\*тут должны возвращать тип выражения\*/

/\* тип имени сравниваем с типом выражения\*/

**void** **C\_AssigmentOperator**() // <оператор присваивания>::=<имя>:=<выражение>

{

**if** (curToken.tt != TokenType.ttIdent)

**throw** **new** **Exception**();

CType left = Ctable\_name[(curToken **as** C\_IdentToken).ident];

GetNextToken();

**if** (!C\_Accept(OperSymbol.Sassigment))

**throw** **new** **Exception**();

GetNextToken();

CType right = C\_Excpression();

isDerivetToAssignemt(left, right);

}

/\*тут должны возвращать тип выражения\*/

CType **C\_Excpression**() // <выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение>

{

CType left = C\_SimpleExpression();

**if** (C\_realat())

{

GetNextToken();

CType right = C\_SimpleExpression();

**if** (!left.isDerivedTo(right))

{

leks.C\_ERR.Add(**new** errors(**7**));

**throw** **new** **Exception**();

}

**return** **new** **CBoolType**();

}

**return** left;

}

CType **C\_SimpleExpression**() // <простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}

{

**if** (C\_Accept(OperSymbol.Ssubtraction))

GetNextToken();

CType left = C\_Term();

**while** (**true**)

{

**if** (C\_add())

{

GetNextToken();

CType right = C\_Term();

**if** (!left.isDerivedTo(right))

{

leks.C\_ERR.Add(**new** errors(**7**));

**throw** **new** **Exception**();

}

left = DerivetTo(left, right);

}

**else**

{

**break**;

}

}

**return** left;

}

CType **C\_Term**() // <слагаемое>::=<множитель>{<мультипликативная операция><множитель>}

{

CType left = C\_Factor();

**while** (**true**)

{

**if** (C\_mult())

{

GetNextToken();

CType right = C\_Factor();

**if**(!left.isDerivedTo(right))

{

leks.C\_ERR.Add(**new** errors(**7**));

**throw** **new** **Exception**();

}

left = DerivetTo(left, right);

}

**else**

{

**break**;

}

}

**return** left;

}

**bool** **C\_mult**()

{

**if** ((curToken **is** C\_OperToken)&&((curToken **as** C\_OperToken).symb == OperSymbol.Smult ||

(curToken **as** C\_OperToken).symb == OperSymbol.Sdivision ||

(curToken **as** C\_OperToken).symb == OperSymbol.Sdiv ||

(curToken **as** C\_OperToken).symb == OperSymbol.Smod ||

(curToken **as** C\_OperToken).symb == OperSymbol.Sand))

{

**return** **true**;

}

**return** **false**;

}

**bool** **C\_add**() // <аддитивная операция>::=+|-|or \*

{

**if** (curToken.tt == TokenType.ttOperation)

{

**if** ((curToken **as** C\_OperToken).symb == OperSymbol.Sadd ||

(curToken **as** C\_OperToken).symb == OperSymbol.Ssubtraction ||

(curToken **as** C\_OperToken).symb == OperSymbol.Sor)

{

**return** **true**;

}

}

**return** **false**;

}

**bool** **C\_realat**() // <операция отношения>::==|<>|<|<=|>=|>|in \*

{

**if** (curToken.tt == TokenType.ttOperation)

{

**if** ((curToken **as** C\_OperToken).symb == OperSymbol.Sequals ||

(curToken **as** C\_OperToken).symb == OperSymbol.Snotequals ||

(curToken **as** C\_OperToken).symb == OperSymbol.Ssmall ||

(curToken **as** C\_OperToken).symb == OperSymbol.Smore ||

(curToken **as** C\_OperToken).symb == OperSymbol.Smorequals ||

(curToken **as** C\_OperToken).symb == OperSymbol.Ssmallequals ||

(curToken **as** C\_OperToken).symb == OperSymbol.Sin)

{

**return** **true**;

}

}

**return** **false**;

}

CType **C\_Factor**() //<множитель>::=<имя>|<константа без знака>|(<выражение>)

/\*если первые 2 фора прошли то если не пройдёт 3 то кинет исключение а если пройдёт то норм\*/

{

**if** (curToken.tt == TokenType.ttIdent)

{

CType a = Ctable\_name[(curToken **as** C\_IdentToken).ident];

GetNextToken();

**if** (a != **null**)

**return** a;

**else**

**throw** **new** **Exception**();

}

**if** (curToken.tt == TokenType.ttConst)

{

**if** (curToken **is** IntVariant)

{

GetNextToken();

**return** **new** **CIntType**();

}

**if** (curToken **is** FloatVariant)

{

GetNextToken();

**return** **new** **CFloatType**();

}

**if**(curToken **is** StringVariant)

{

GetNextToken();

**return** **new** **CStringType**();

}

**if**(curToken **is** BoolVariant)

{

GetNextToken();

**return** **new** **CBoolType**();

}

}

**return** **C\_Excpression**();

**throw** **new** **Exception**();

//return C\_Excpression();

}

**void** **C\_CompositeOperator**() // <составной оператор>::= begin <оператор>;{<оператор>;} end

{

GetNextToken();

C\_Operator();

**if** (!C\_Accept(OperSymbol.Ssemicon))

**throw** **new** **Exception**();

**while** (**true**)

{

**if** (C\_Accept(OperSymbol.Ssemicon))

{

GetNextToken();

C\_Operator();

}

**else**

{

**break**;

}

}

**if** (!C\_Accept(OperSymbol.Send))

{

**throw** **new** **Exception**();

}

GetNextToken();

}

**void** **C\_ChoiseOperator**() // <выбирающий оператор>::= if <выражение> then <оператор>|if <выражение> then <оператор> else <оператор>

{

GetNextToken();

CType left = C\_Excpression();

**if**(!(left **is** CBoolType))

**throw** **new** **Exception**();

**if** (!C\_Accept(OperSymbol.Sthen))

**throw** **new** **Exception**();

GetNextToken();

C\_Operator();

**if** (C\_Accept(OperSymbol.Selse))

{

GetNextToken();

C\_Operator();

}

}

**void** **C\_CycleOpertor**() // <оператор цикла>::= while <выражение> do <оператор>

{

GetNextToken();

CType left = C\_Excpression();

**if** (!(left **is** CBoolType))

**throw** **new** **Exception**();

**if** (!C\_Accept(OperSymbol.Sdo))

**throw** **new** **Exception**();

GetNextToken();

C\_Operator();

}

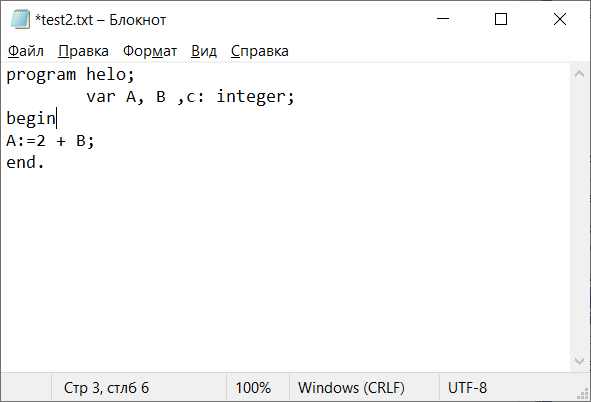
}

Стоит уделить внимание работе с альтернативами. Я использовал обработку символа на верхних уровнях анализа, что бы лишний раз не заходить не в те БНФ и не обрабатывать не критичные исключения.

## Тестирование

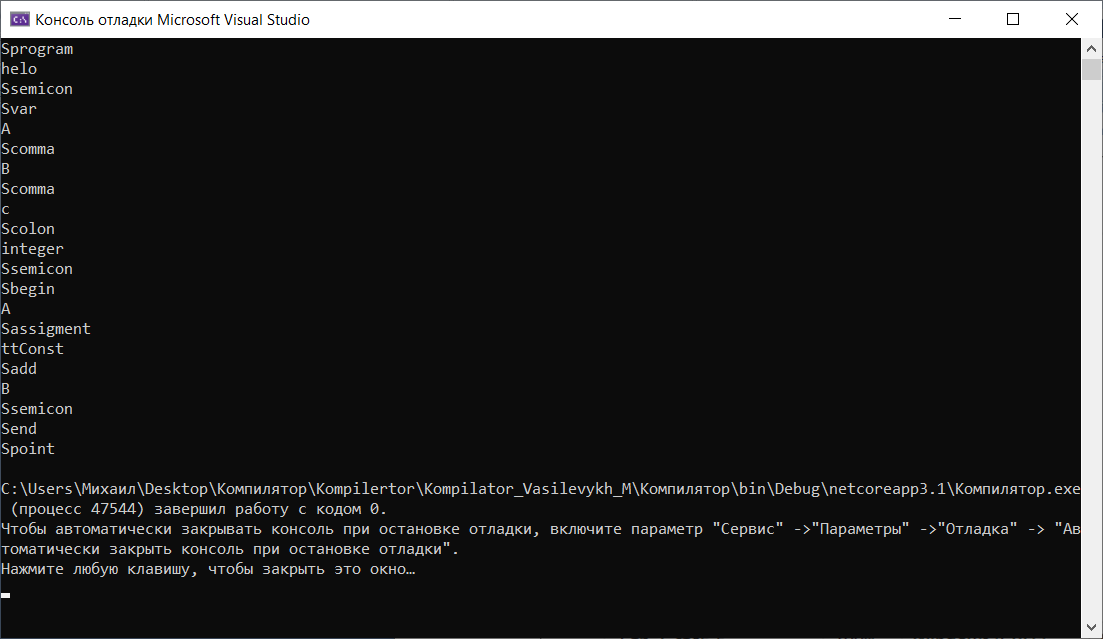
Работа с простыми арифметическими операторами и сложными операторами

Входные данные:



Ожидаемый результат: отсутствие ошибок и корректная работа программы

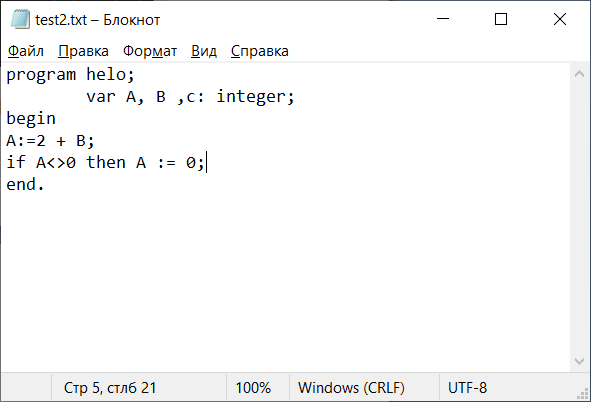
Выходной результат:



Работа с простыми арифметическими операторами

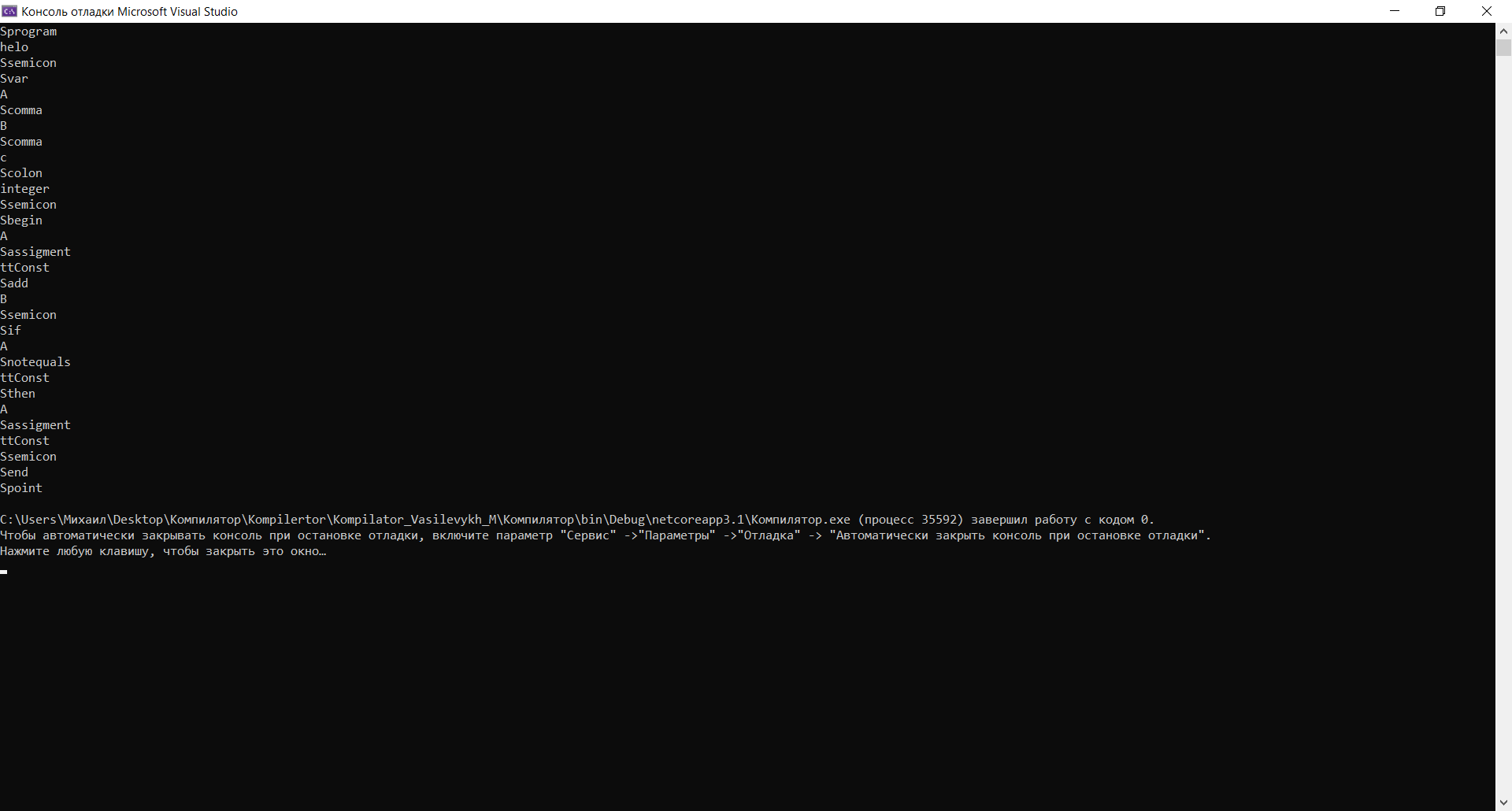
Работа с оператором ветвления

Входной результат:



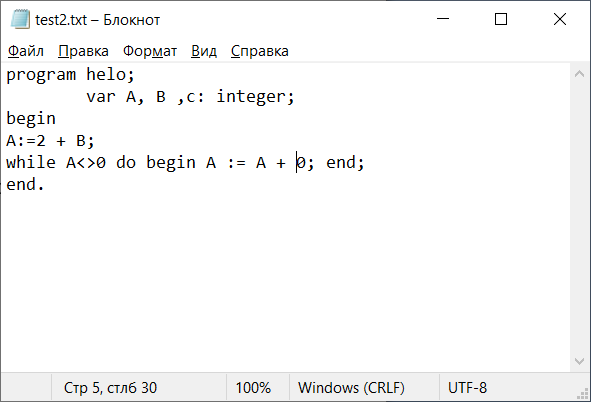
Ожидаемый результат: отсутствие ошибок и корректная работа программы

Выходной результат:



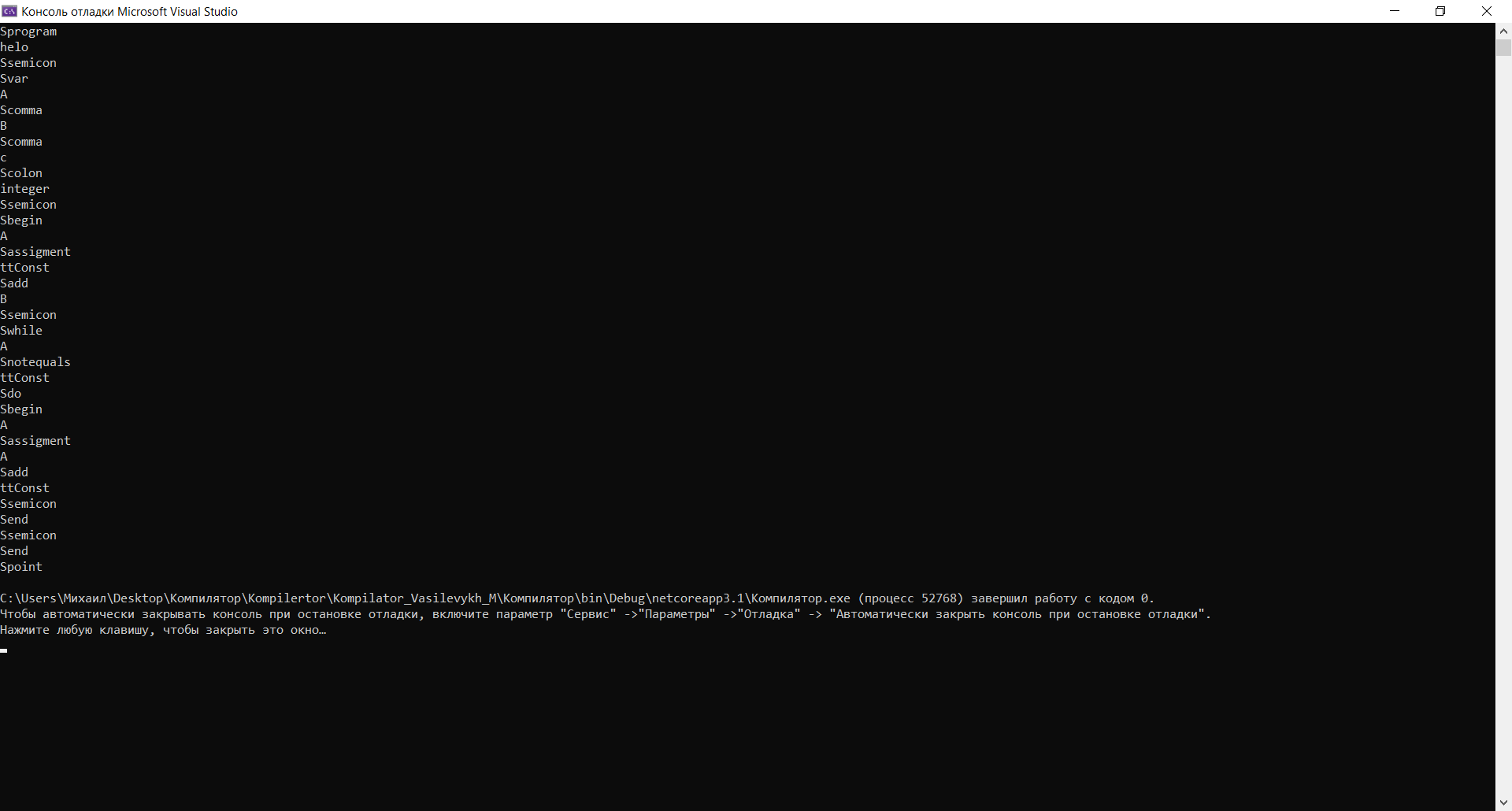
Работа с оператором цикла

Входной результат:



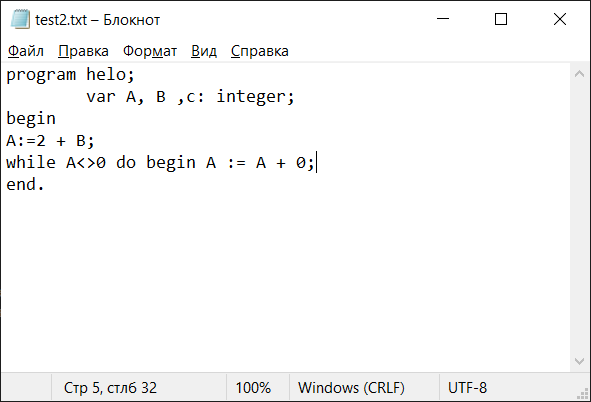
Ожидаемый результат: отсутствие ошибок и корректная работа программы

Выходной результат:



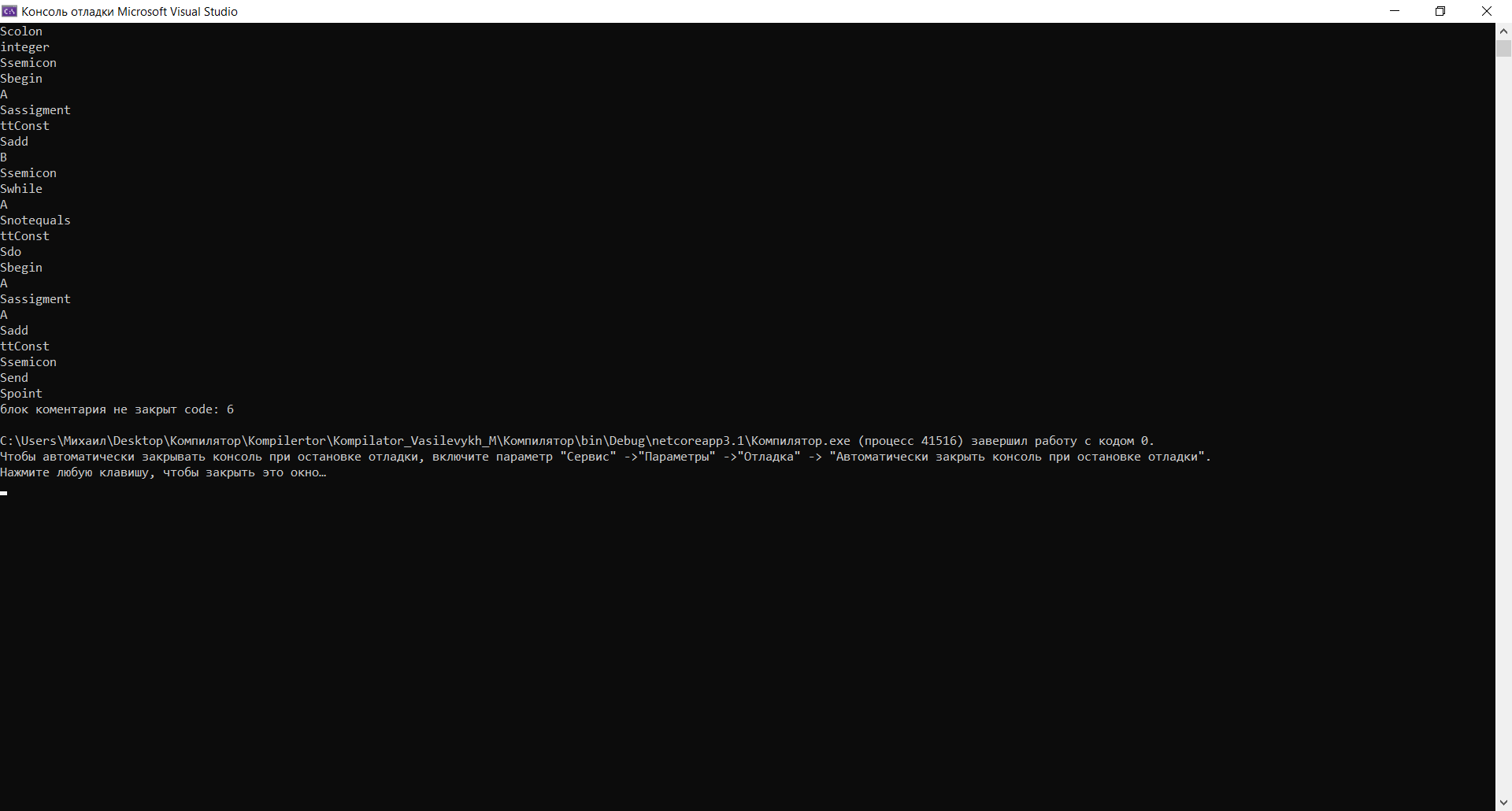
Есть синтаксическая ошибка

Входной результат:



Ожидаемый результат: сообщение об ошибке

Выходной результат:



# Семантический анализатор

## Описание

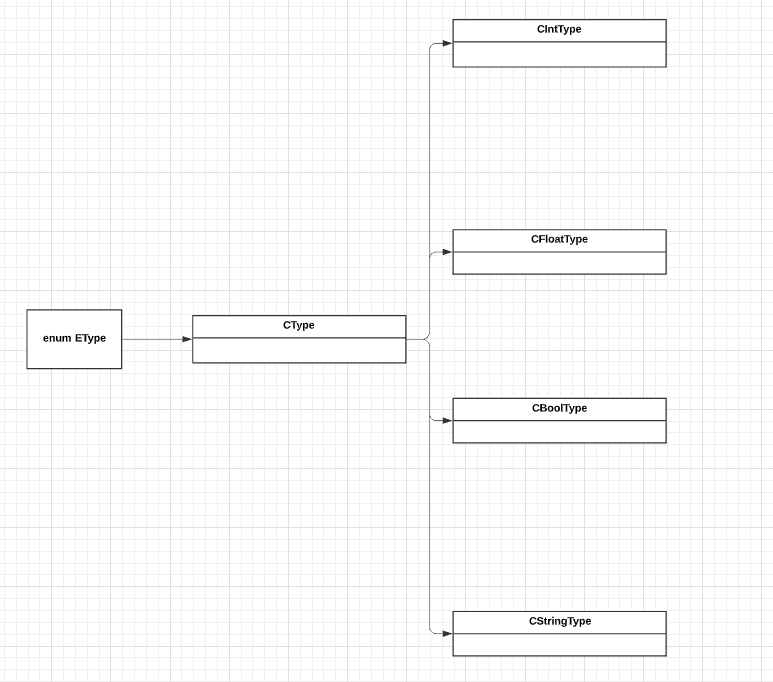
Синтаксический анализатор проверяет наличие соответствие типов, инициализацию всех переменных.

## Проектирование

Нам необходимо проверять приводимость типов и инициализацию переменных.

Для этого нам нужно понимать какой тип возвращает оператор.

Я определил 4 основных типа, так как я не реализовывал блок описания типов, то мы можем обойтись без таблицы типов



Так же мне необходимо создать таблицу идентификаторов для хранения идентификатора и его типа

Так как мы работаем в одной области видимости, то и таблица будет одна.

Таблицу идентификаторов мы заполняем на блоке описания переменных.

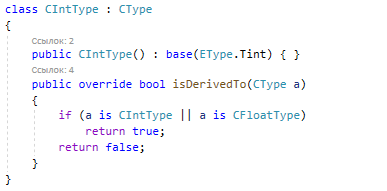
Правила приводимости типов

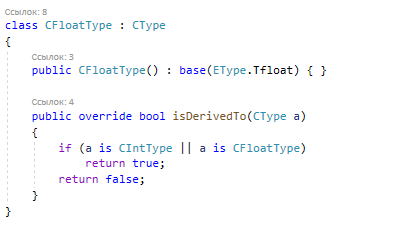
1. Присваивание
   1. Целому значению можно присвоить только целое значение
   2. Вещественному значению можно присвоить целое или вещественное значение
   3. Строковому типу можно присвоить только строковое значение
   4. Логическому типу можно присвоить только логическое значение
2. Операции арифметические и логические
   1. Проводить операции можно целыми и вещественным значениями
   2. Остальные только с такими же типами

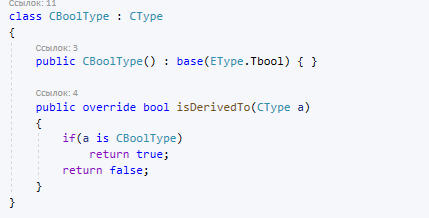
## Реализация

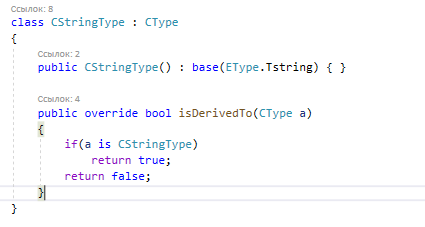
Приводимость типов мы определяем по 2 действиям: присваивание и арифметические и логические операции операции.

Анализ для арифметических операций хранится как метод для каждого класса отдельно

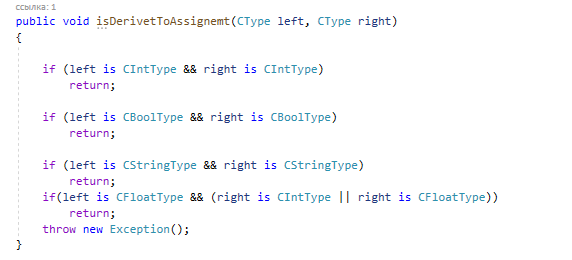




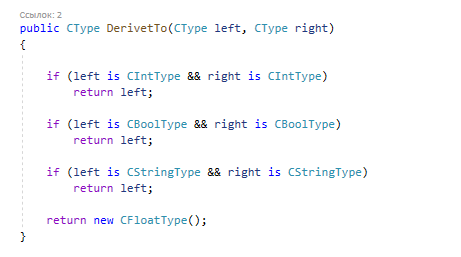




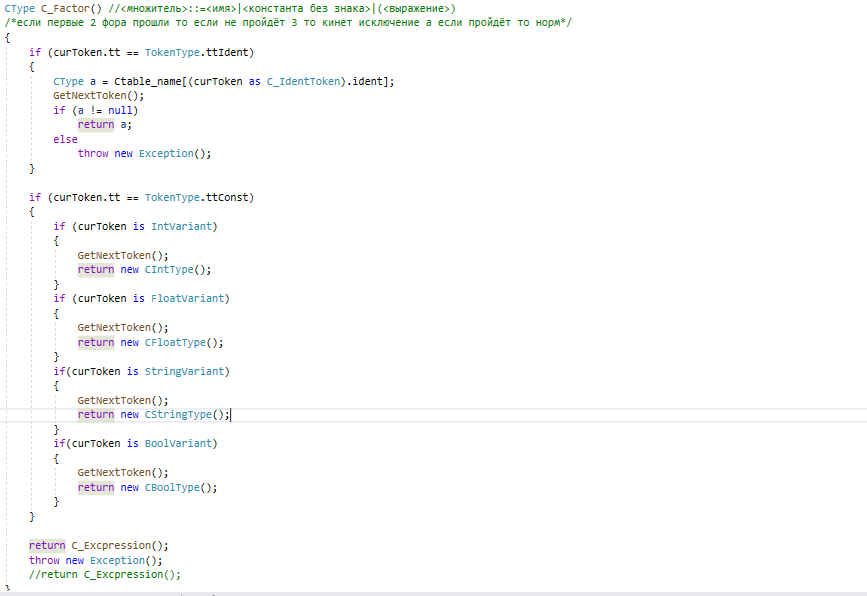
А приводимость для присваивания работает как метод анализатора

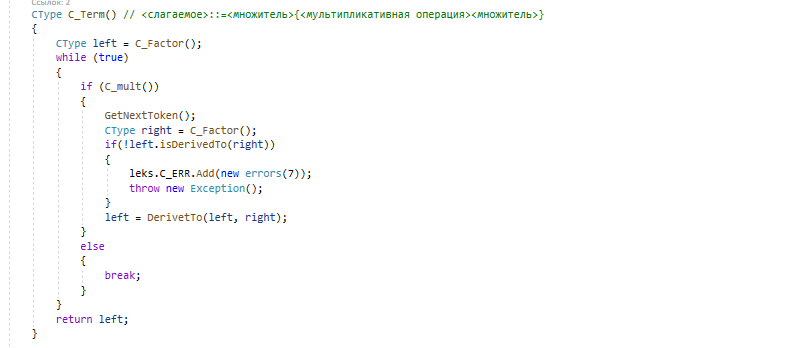


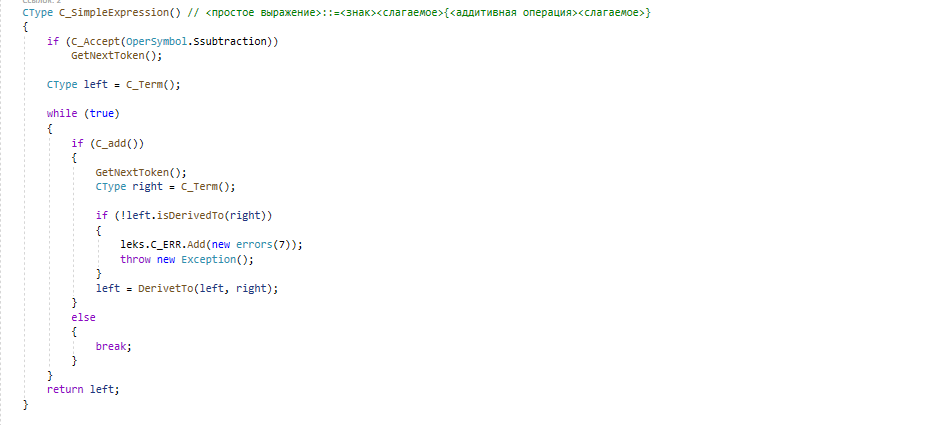
Так же есть метод для приведения типов

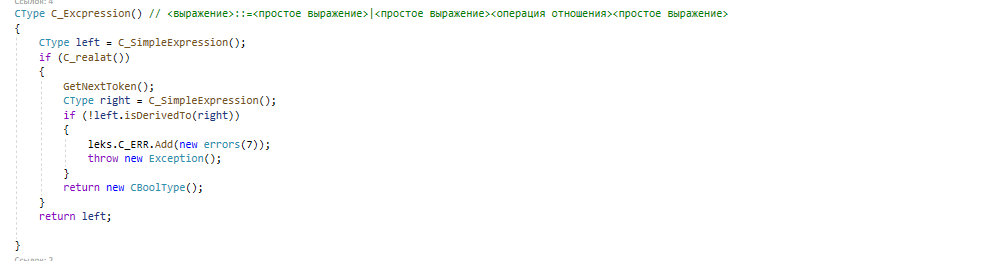


Так все операции связанные с типами происходят в БНФ связанных с операциями они должны возвращать тип значения, с которым работают (<слагаемое>, <простое выражение>, <выражение>, <множитель>).









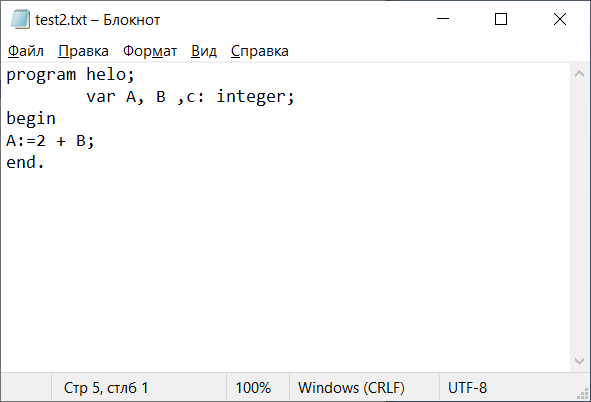
Проверка типа возвращенного нижней БНФ происходит в вызвавшей БНФ.

В БНФ «множитель» мы получаем тип символа, с которым работаем, а в БНФ «раздел переменных» мы заполняем таблицу идентификаторов.

## Тестирование

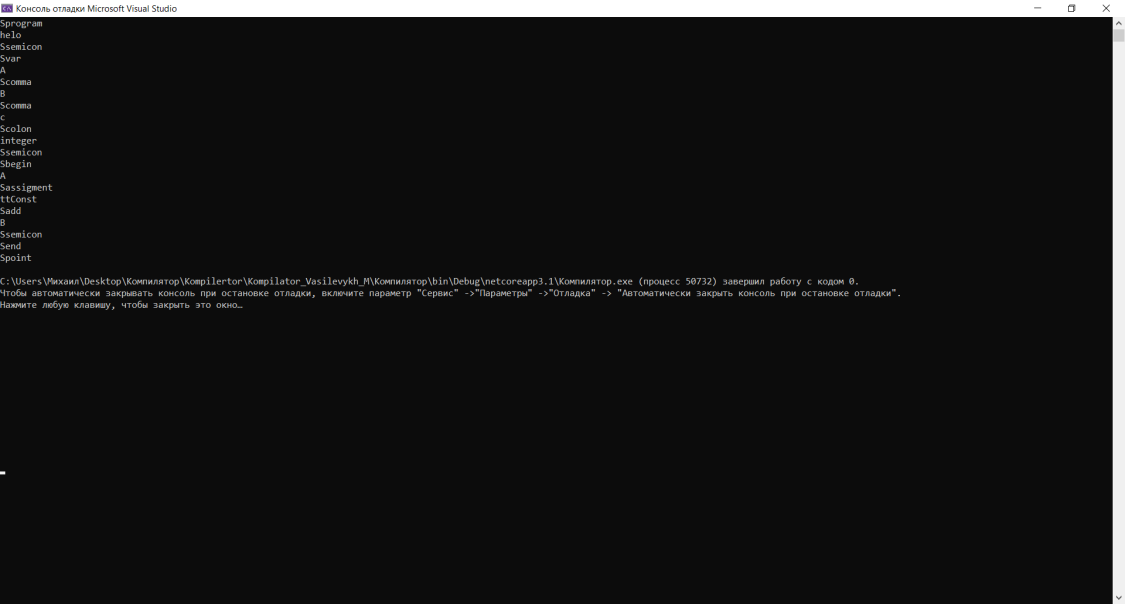
Проверка приводимости типов при арифметической операции и присваивания

Входные данные:



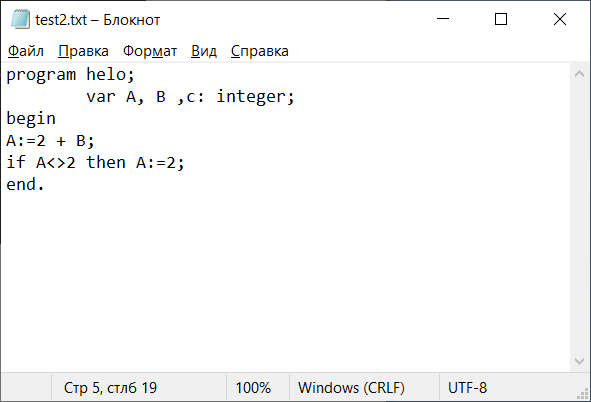
Ожидаемый результат: Отсутствие ошибок, нормальная работа программы без прерываний

Выходной результат:



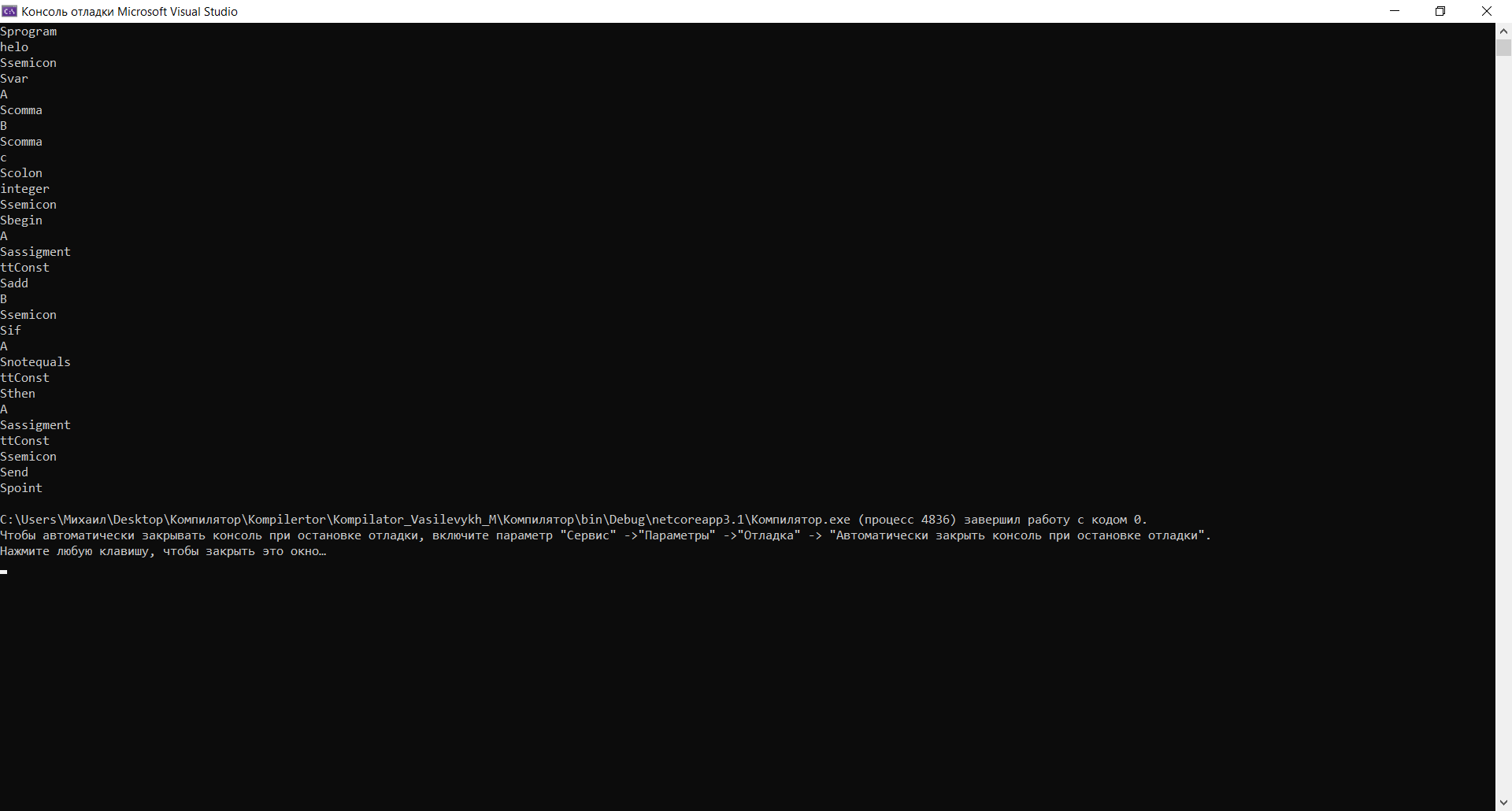
Проверка приводимости типов при арифметической операции и присваивания, и логической операции

Входные данные:



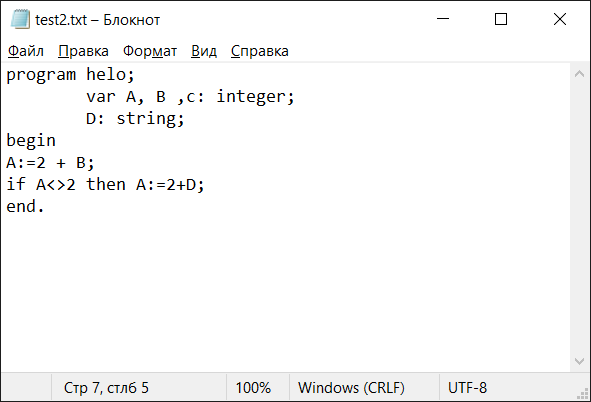
Ожидаемый результат: Отсутствие ошибок, нормальная работа программы без прерываний

Выходные данные:



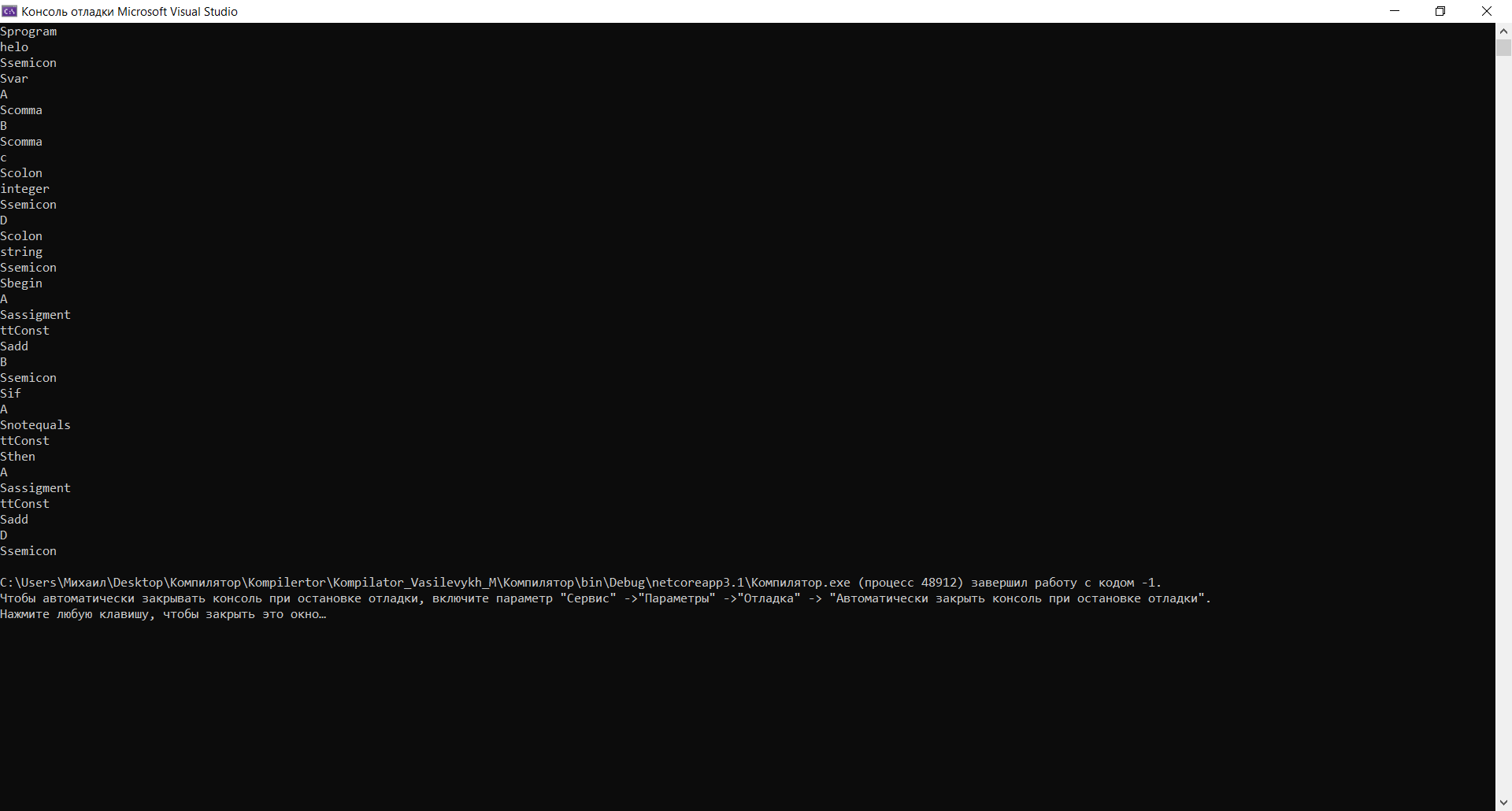
Ошибка в привидении типа(проверка нейтрализации ошибок)

Входные данные:



Ожидаемый результат: ошибка не приводимости типов

Выходной результат:



Так как ошибка проверяется на уровень выше то при листинге ошибка съезжает на 1 позицию вниз.

# Генератор

## Описание:

Генерация кода — это машинно-зависимая часть компилятора, так как она определяется архитектурой конкретной вычислительной машины (в нашем случае, так как мы пользуемся CLR, мы аппаратно независимы).

Результат генерации кода — объектная программа может представлять собой последовательность машинных команд или программу на языке ассемблера.

## Проектирование:

Генератор встроен в БНФ.

Изначально было 2 варианта реализации.

1. Сделать двухпроходный компилятор, сначала проход анализаторов, если все хорошо то снова прогоняем программу и генерируем код
2. Прогоняем программу сразу и генерируем тут же если не возникло ошибок то сохраняем иначе нет.

Я остановился на 2 варианте.

Нам нужно определится, каким образом и во что мы будем транслировать нашу программу, я выбрал язык MSIL, так как генерация кода в него аппаратно независимая, но он выше уровнем, чем любой машинный язык, но это не меняет его сути, как машинного языка, поскольку он может быть описан при помощи ассемблера.

Так как мы будем работать со стеком подразумевается работа с выражениями в польской записи, поэтому был немного изменен анализатор.

## Реализация

Так как изначально блоки анализаторов были написаны на Net Core, мне пришлось переносить всё на Net Framework, потому что не все методы оказались доступны.

Сначала мы определяем динамическую сборку(код сборки используется немедленно, а сборка также сохраняется на диск, чтобы ее можно было исследовать или использовать в другой программе)

Затем определяем модуль в динамической сборке

Определяем и создаем новые экземпляры классов во время выполнения.

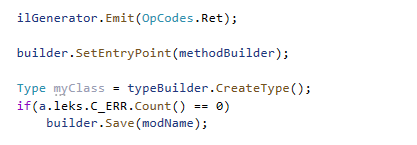
MethodBuilder используется для полного описания метода на языке MSIL, включая имя, атрибуты, сигнатуру и тело метода



При помощи IlGenerator мы создаём инструкции языка msil

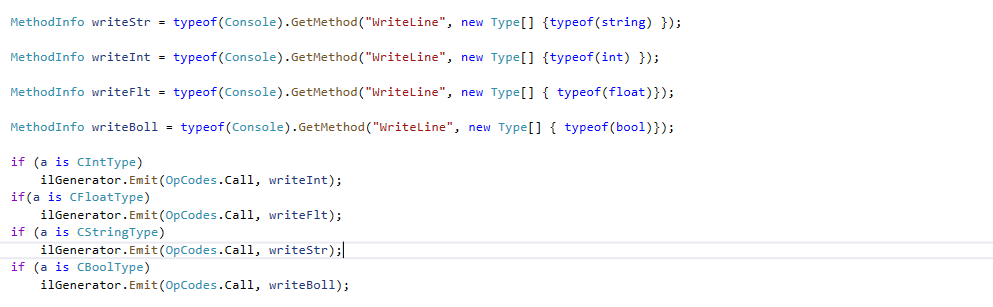
Вызывая SetEntryPoint мы взываем метод, что бы указать что он точка входа в сборку, для того что собрать исполняемый файл.

И сохраняем файл.



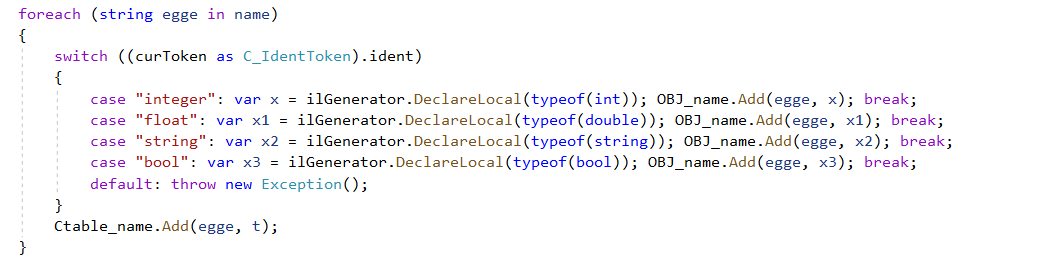
Так как нам необходимо организовать потоковый вывод, была добавлена новая БНФ <write>::=<writeln>’(‘<operaition>’)’;

Помещаем в стек инструкций языка MSIL заданную инструкцию, и маркер метаданных указанного метода, в нашем случае вывода в строку.

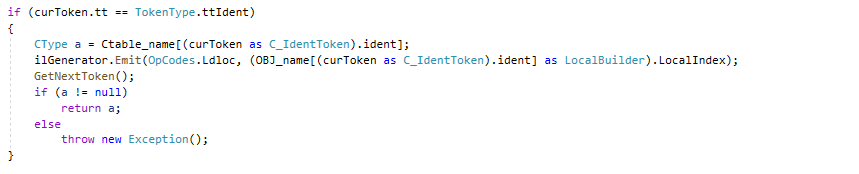


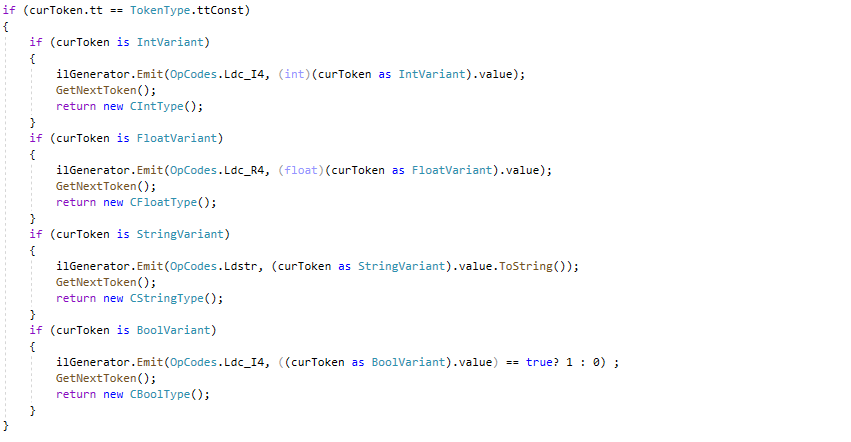
В блоке создания переменных мы объявляем локальные переменные.

Так как для дальнейшей работы нам необходимо точно указывать, что и в какой объект мы помещаем нам нужно создать словарь, где ключом будет символьное название переменной, а значение, объект LocalBuilder



В блоке множитель мы добавляем в стек значения наших множителей





Так как в msil нет как таковой логической константы, а вместо нее используется целочисленное значения, мне пришлось использовать условный оператор.

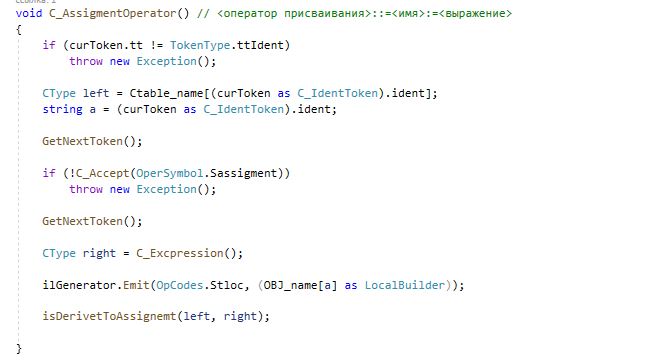






Так как в msil нет как таковых операций !=; >=; <= нам необходимо определить их совместным использованием других известных нам операций.

Вычисление промежуточных значений. Так как в стеке числа вычисляются как в польской нотации, поэтому пришлось изменять порядок добавления, а именно, сначала добавляем правый и левый множитель, а затем уде операцию.



В операторе присваивания нам нужно вычислять полученные выражения и передавать их значения в необходимую переменную.

Для реализации ветвления и циклов на необходимо обращаться к меткам

Метка отображает текущую инструкцию в стеке инструкций (по сути, она похожа на метки из с++ goto)

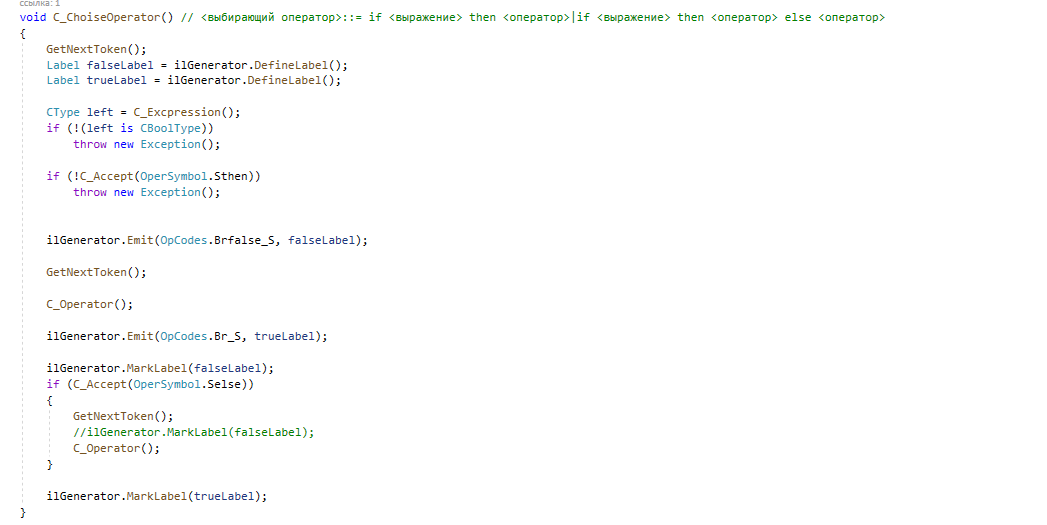
Для ветвления мы создаем 2 метки для выполнения условия и его не выполнения.

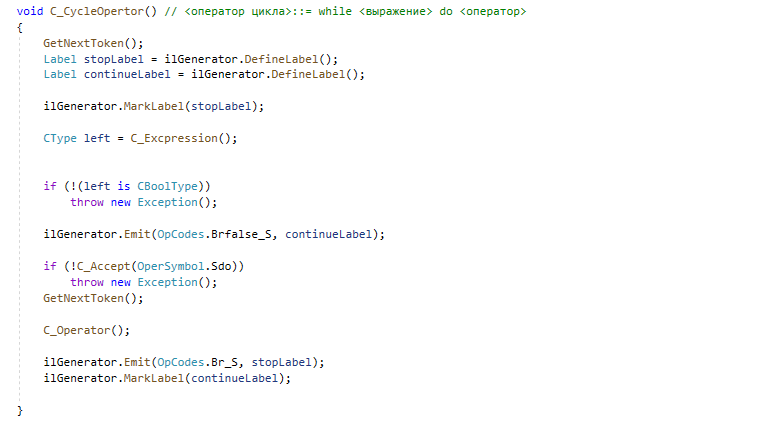
Если в стеке имеем значение true(1) то передаём в метку true

То есть проходим оператор и выходим

Иначе в метку false

Проскакиваем оператор и выполняем ветку else





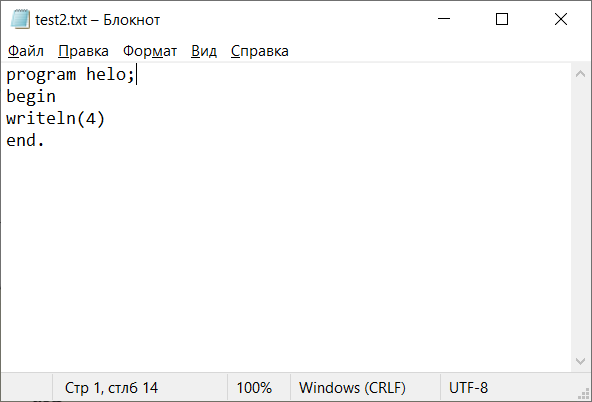
Тут ситуация аналогична, за исключение того что метка зациклена.

## Тестирование

Стоит отметить заранее что операция nop не является ошибкой это инструкция предписывающая ничего не делать, она ни коем образом не влияет на работы исходной программы.

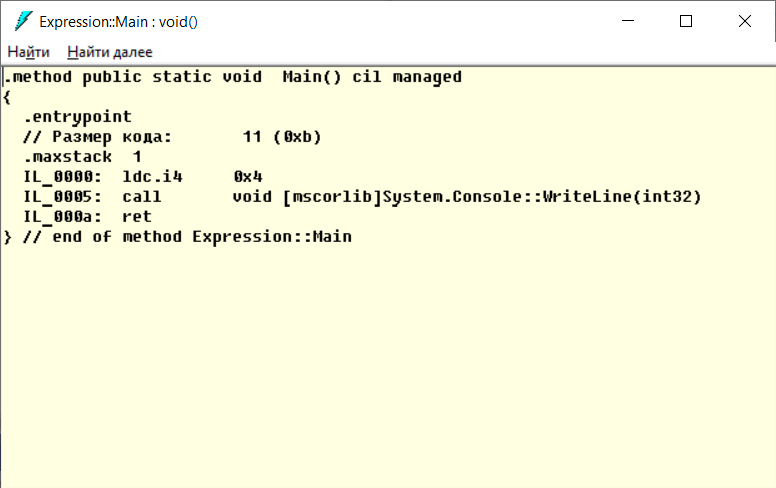
Проверка работы модуля вывода

Входные данные:



Ожидаемый результат: Вывод числа 4

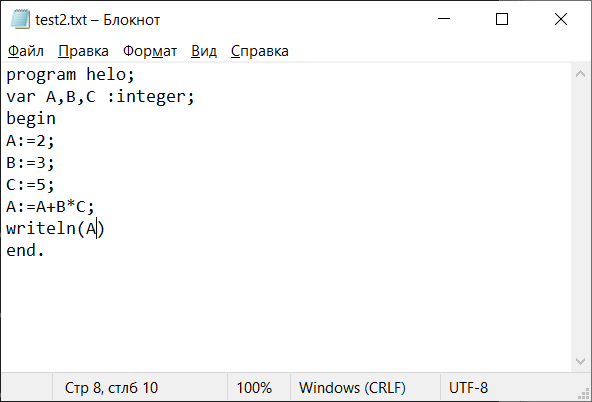
Выходные данные





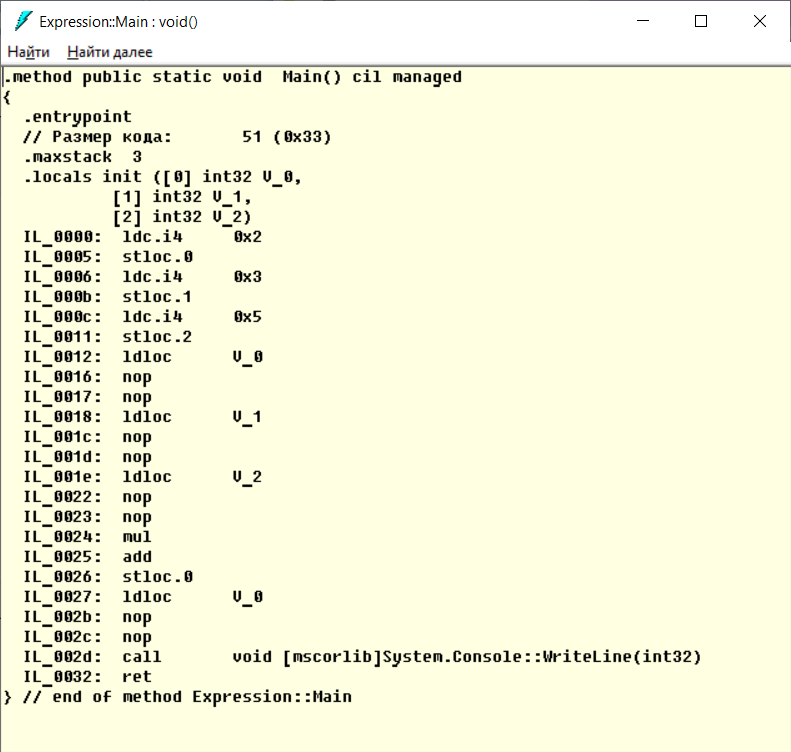
Работа с арифметическими операциями

Входные данные:



Ожидаемый результат: 17

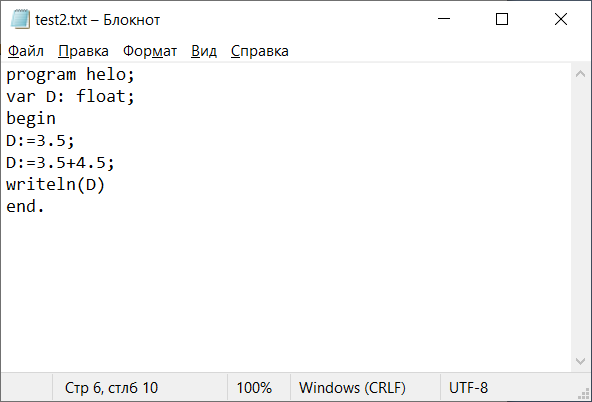
Выходные данные:





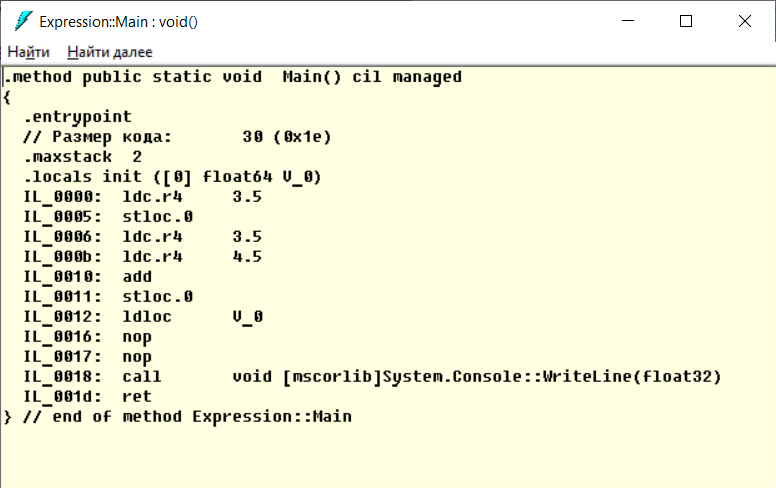
Работа с арифметическими операциями

Входные данные:



Ожидаемый результат: 8

Выходные данные:

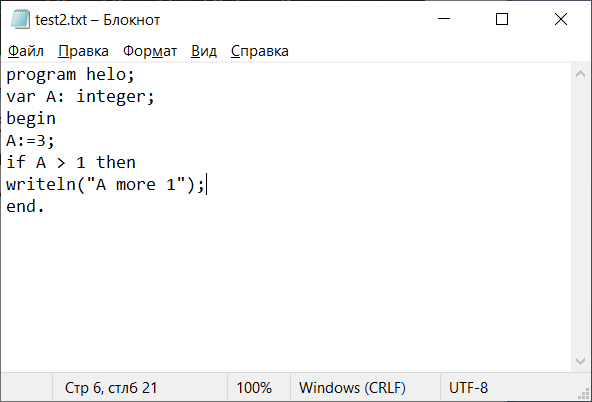




Так как msil не способен самостоятельно приводить типы то совместные операции int и float не возможны без дополнительного приведения, но так как, нам недостаточно имеющего контекста о том какая будет следующая константа, я принял решение о том, что ликвидной будут операции только внутри своих типов.

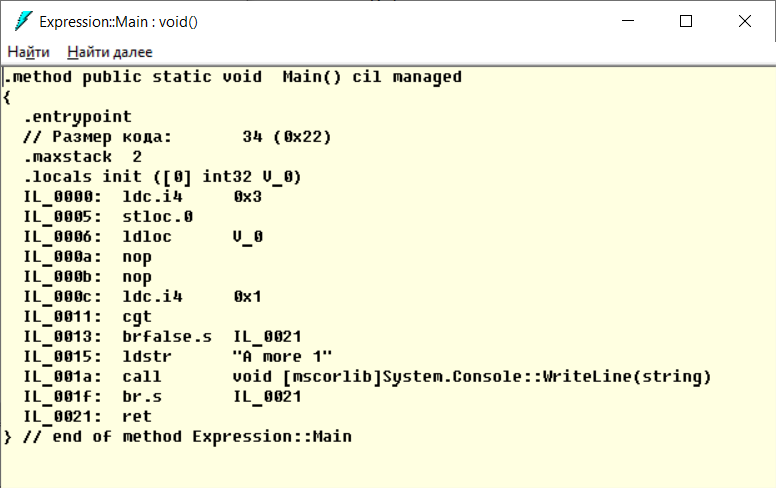
Работы с ветвлением

Входные данные:



Ожидаемый результат: A more 1

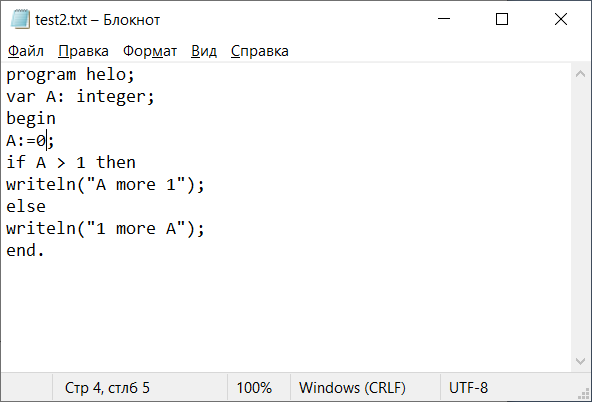
Выходные данные:





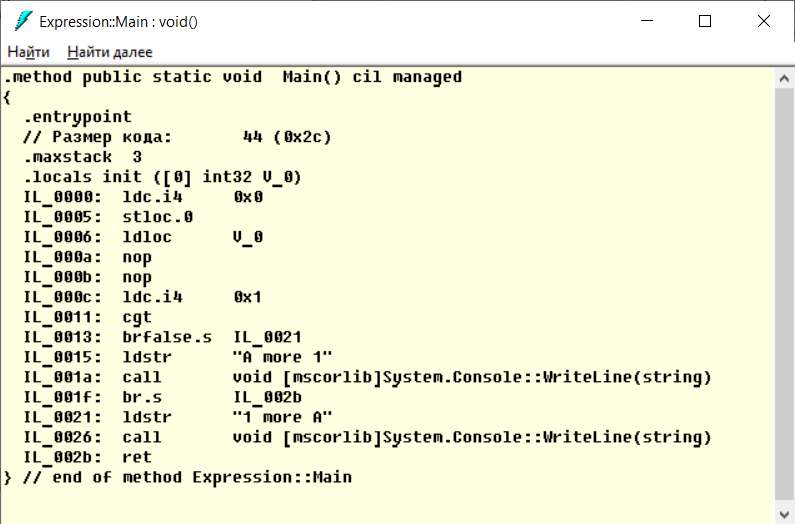
Работы с ветвлением

Входные данные:



Ожидаемый результат: 1 more A

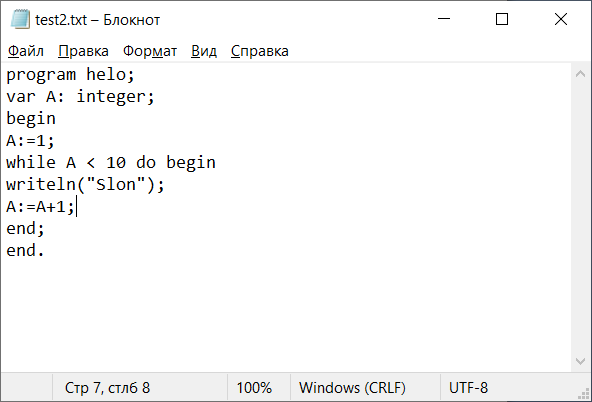
Выходные данные:





Работа с циклами

Входные данные:



Ожидаемый результат:

9 раз выведенная фраза “Slon”

Выходные данные:

