Министерство науки и высшего образования Российской Федерации

**Муромский институт (филиал)**

федерального государственного бюджетного образовательного учреждения высшего образования

**«Владимирский государственный университет**

**Имени Александра Григорьевича и Николая Григорьевича Столетовых»**

**(МИВлГУ)**

Факультет ИТР

Кафедра ПИн

КУРСОВАЯ

РАБОТА

По Теория автоматов и формальных языков

Тема Транслятор подмножества языка VB

Руководитель

. . Кульков Я.Ю. .

(оценка) (фамилия, инициалы)

. .

(подпись) (дата)

Члены комиссии Студент ПИн-120

(группа)

. . Цыбуцынин С.Н.

(подпись) (Ф.И.О.) (фамилия, инициалы)

. . .

( подпись) (Ф.И.О.) (подпись) (дата)

Муром 2022

Место для задания

Целью данной курсовой работы является разработка транслятора с подмножества языка Visual Basic. Программа реализована на языке C# в среде разработки «Microsoft Visual Studio 2017». Приложение реализовано для операционной системы Windows.

The purpose of this course work is to develop a translator from a subset of the Visual Basic language. The program is implemented in C# in the Microsoft Visual Studio 2017 development environment. The application is implemented for the Windows operating system.

Содержание

[Введение 6](#_Toc122914454)

[1 Анализ технического задания 7](#_Toc122914455)

[2 Описание грамматики языка 9](#_Toc122914456)

[3 Разработка архитектуры системы и алгоритмов 220](#_Toc122914457)

[4 Методика испытаний 25](#_Toc122914458)

[5 Руководство пользователя 33](#_Toc122914459)

[6 Руководство программиста 35](#_Toc122914460)

[Заключение 39](#_Toc122914461)

[Список литературы 40](#_Toc122914462)

[Приложение А. Ссылка на репозиторий 41](#_Toc122914463)

# Введение

Программа, написанная на языке высокого уровня, перед исполнением должна быть преобразована в программу на "машинном языке". Такой процесс называется трансляцией, или компиляцией. Для исходной программы в машинный код используется транслятор.

Транслятор (англ. translator — переводчик) — это программа-переводчик. Она преобразует программу, написанную на одном из языков высокого уровня, в программу, состоящую из машинных команд.

Исходными данными для работы транслятора служит текст входной программы – некоторая последовательность предложений входного языка программирования, удовлетворяющая синтаксическим требованиям.

Программу, написанную на языке программирования высокого уровня, называют исходной программой, а каждую самостоятельную программную единицу, образующую данную программу, - программным модулем. Для преобразования исходной программы в ее выполняемую форму (выполнимый файл) транслятор выполняет некоторую последовательность действий. Эта последовательность зависит как от языка программирования, так и от конкретной реализации самого транслятора.

В процессе трансляции выполняется анализ исходной программы, а затем синтез выполнимой формы данной программы. В зависимости от числа просмотров исходной программы, выполняемых компилятором, трансляторы разделяются на однопроходные, двухпроходные и трансляторы, использующие более двух проходов.

На основе проведенного анализа, актуальность темы работы находится на этапе учебных целей. Разбор реализации лексического, синтаксического анализа, сложного арифметического выражения, даёт возможность понять как устроен транслятор подмножества для языка программирования на стороне разработчика.

# 1 Анализ технического задания

Цель данной курсовой работы – разработка транслятора с подмножеством для языка Vusual Basic. Программная реализация осуществляется в среде разработки Visual Studio 2019 с использованием высокоуровневого объектно-ориентированного языка программирования C#. Среда разработки Visual Studio имеет приятный, интуитивно понятный интерфейс с множеством мощных вспомогательных средств для разработки, что ускоряет и упрощает разработку приложения. Разработка на языке С# основывается на том что он, простой, надежный и масштабируемый язык программирования. Динамически типизированный характер C# облегчает разработчикам поиск ошибок в коде, а также C# устраняет проблему утечки памяти.

Требования к приложения на основе технического задания:

* обеспечить развернутую диагностику ошибок;
* реализовать класс транслятора;
* реализовать синтаксический разбор - на основе LR(k)-грамматик;
* выполнить разбор арифметических выражений методом Дейкстры;
* в языке поддерживаются: у идентификаторов 8 символов значащие; не менее 3-х директив описания переменных;
* сложный арифметический оператор;
* простое логическое выражение; условный оператор if…then…else

Построение грамматики и все необходимые разборы, для реализации программы-транслятор происходят с использованием заранее заготовленного фрагмента кода языка Visual Basic:

Dim a as integer

a=8

if a>b then

x=4\*y+a/2

y=a

else

x=0

end if

x=0

end if

Разработка программы транслятора разделена на несколько этапов:

* Разработка лексического анализатора, который упрощает анализирование кода при синтаксическом анализе.
* Построение грамматики языка, на основе приведённого кода выше. Грамматика языка должна учитывать возможность вложенности условных операторов, операторов объявления переменных и операторов присваивания в условном операторе. На основе уже построенной грамматики, следующим подэтапом является построение грамматики LR(k) класса.
* На основе LR(k)-грамматики будет реализован синтаксический анализ. Для каждого правила грамматики будет соответствовать своя подпрограмма (метод), которая последовательно проверяет текущие лексемы с ожидаемыми. Если результат будет отличаться от ожидаемого, то анализ прекратиться, и будет выведена соответствующая ошибка.
* Конечным этапом разработки программы является разбор сложного арифметического выражения методом Дейкстры.

# 2 Описание грамматики языка

2.1 Описание языка Visual Basic

Visual Basic — компьютерный язык (точнее, семейство языков), созданный и развиваемый корпорацией Microsoft, а также интегрированная среды разработки. Является дальнейшим развитием языка QuickBasic (также разработанного Microsoft), от которого унаследовал общую концепцию, стиль и синтаксис.

Однако, развитие Visual Basic пошло в сторону процедурного, объектного, компонентного и событийного программирования. Язык активно используется как для разработки Windows-приложений, так и для создания ПО для других платформ. Visual Basic можно скачать и использовать как отдельно, так и в составе Microsoft Visual Studio.

2.2 Описание оператора объявления переменных

Структура объявления переменных в языке Visual Basic представлена в виде:

Dim <Имена переменных> As <тип> [=(Значение переменной)]

Блок <Имена переменных> может содержать в себе как одну переменную, так и несколько переменных – список, где каждая новая переменная отделяется запятой другой. Блок [= Значение] является необязательным, значение переменной или переменных по умолчанию может быть не определено.

2.3 Описание оператора присваивания

Структура оператора присваивания в языке Visual Basic представлена в виде:

<переменная>=<выражение>

В блоке <переменная>, находится имя той переменной, которое нужно изменить значение. Блок <выражение> может содержать в себе статичное значение или переменную, или арифметическое выражение, которое прежде нужно вычислить. Значение переменной в блоке <переменная>, становится равно значению в блоке <выражение>, которой может быть представлен как в виде сложного арифметического выражения, так и в виде простого.

2.4 Условный оператор

Структура условного оператора в языке Visual Basic представлена в виде:

If <логическое\_условие> then <список\_операций> End If

Блок <логическое\_условие> состоит из логического выражения, если оно верно то, выполняется список операций, определенный в блок <список\_операций>.

Структура условного оператора с двумя ветвями в языке Visual Basic представлена в виде:

If <логическое\_условие> then <список\_операций1> Else < список\_операций2> End If

Если блок <логическое\_условие> - истина, то выполнится блок <список\_операций1>, если же ложно, выполнился блок <список\_операций2>

2.5 Грамматика языка

На основе вышеописанных конструкций операторов языка Visual Basic, была разработана следующая грамматика языка:

G = (T, N, P, <программа>)

T = {Dim, as, integer, double, string, =, if, >, <, <= , >=, then, \* +, -, /, else, end, id, lit, expr, \n, , }

N = {<программа>, <список\_опис>, <список\_опер>, <список\_пер>, <пер>, <тип>, <опер>, <условн>, <присв>, <блок\_усл>, <операнд>, <знак>}

P= {<программа> ::= <список\_операторов>

<список\_операторов> ::=

<оператор> \n | <список\_операторов> <оператор> \n

<оператор> ::=

<условие> | <присвоение> | <объявление>

<объявление> ::=

Dim <список\_переменных> as <тип>

<список\_переменных> ::=

id | <список\_переменных>, id

<тип> ::=

integer | double | string

<условие> ::=

if <логическое\_условие> then \n

<список\_операторов> end if \n |

if <логическое\_условие> then \n <список\_операторов> else <список\_опер> end if \n

<логическое\_условие> ::= <операнд> <логический\_знак> <операнд >

<логический\_знак> ::= > | < | = | >= | <=

<присвоение> ::=

id = expr

<операнд> ::=

id | lit

}

/n- отвечает за конец строки, lit – литерал, id – идентификатор, expr - функция для анализа сложного арифметического выражения.

2.6 Грамматика языка для синтаксического анализа

В соответствии с техническим заданием синтаксическим разбор лексем следует выполнять восходящем анализатором на основе LR(k)-грамматики.

Таблица 1 – Присоединённая грамматика восходящего анализатора

|  |  |  |  |
| --- | --- | --- | --- |
| Состояние | Предыдущее состояние | Правила грамматики | Переход |
| 0 | - | <программа> ::= • <список\_операторов>  <список\_операторов> ::= • <оператор> \n  <список\_операторов> ::= • <список\_операторов> <оператор> \n <оператор> ::= • <условие>  <оператор> ::= • <присвоение>  <оператор> ::= • <объявление>  <условие> ::= • if <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = • if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if \n  <объявление> ::= • Dim <список\_переменных> as <тип>  <присвоение> ::= • id = expr | 1  2  1  3  4  5  6  6  7  8 |
| 1 | 0 | <программа> ::= <список\_операторов> •  <список\_операторов> ::= <список\_операторов> • <оператор> \n  <оператор> ::= • <условие>  <оператор> ::= • <присвоение>  <оператор> ::= • <объявление>  <условие> ::= • if <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = • if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if \n  <объявление> ::= • Dim <список\_переменных> as <тип>  <присвоение> ::= • id = expr | x  9  3  4  5  6  6  7  8 |
| 2 | 0,29 | <список\_операторов> ::= <оператор> • \n | 10 |
| 3 | 0,1,29, 36 | <оператор> ::= <условие> • | x |
| 4 | 0,1,29, 36 | <оператор> ::= <присвоение> • | x |
| 5 | 0,1,29, 36 | <оператор> ::= <объявление> • | x |

Продолжение таблицы 1

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 0,1,29, 36 | <условие> ::= if • <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = if • <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if \n  <логическое\_условие> ::= • <операнд> <логический\_знак> <операнд>  <операнд> ::= • id  <операнд> ::= • lit | 11  11  12  13  14 |
| 7 | 0,1,29, 36 | <объявление> ::= Dim • <список\_переменных> as <тип>  <список\_переменных> ::= • id  <список\_переменных> ::= • <список\_переменных>, id | 15  16  15 |
| 8 | 0,1,29, 36 | <присвоение> ::= id • = expr | 17 |
| 9 | 1,36 | <список\_операторов> ::= <список\_операторов> <оператор> • \n | 18 |
| 10 | 2 | <список\_операторов> ::= <оператор> \n • | x |
| 11 | 6 | <условие> ::= if <логическое\_условие> • then \n <список\_операторов> end if \n  <условие> :: = if <логическое\_условие> • then \n <список\_операторов> else \n <список\_операторов> end if \n | 19  19 |
| 12 | 6 | <логическое\_условие> ::= <операнд> • <логический\_знак> <операнд>  <логический\_знак> ::= • >  <логический\_знак> ::= • <  <логический\_знак> ::= • =  <логический\_знак> ::= • >=  <логический\_знак> ::= • <= | 20  21  22  23  24  25 |
| 13 | 6 | <операнд> ::= id • | x |
| 14 | 6 | <операнд> ::= lit • | x |
| 15 | 7 | <объявление> ::= Dim <список\_переменных> • as <тип>  <список\_переменных> ::= <список\_переменных> • , id | 26  27 |
| 16 | 7 | <список\_переменных> ::= id • | x |
| 17 | 8 | <присвоение> ::= id = • expr | 28 |
| 18 | 9 | <список\_операторов> ::= <список\_операторов> <оператор> \n • | x |
| 19 | 11 | <условие> ::= if <логическое\_условие> then • \n <список\_операторов> end if \n  <условие> :: = if <логическое\_условие> then • \n <список\_операторов> else \n <список\_операторов> end if \n | 29  29 |

Продолжение таблицы 1

|  |  |  |  |
| --- | --- | --- | --- |
| 20 | 12 | <логическое\_условие> ::= <операнд> <логический\_знак> • <операнд>  <операнд> ::= • id  <операнд> ::= • lit | 30  13  14 |
| 21 | 12 | <логический\_знак> ::= > • | x |
| 22 | 12 | <логический\_знак> ::= < • | x |
| 23 | 12 | <логический\_знак> ::= = • | x |
| 24 | 12 | <логический\_знак> ::= >= • | x |
| 25 | 12 | <логический\_знак> ::= <= • | x |
| 26 | 15 | <объявление> ::= Dim <список\_переменных> as • <тип>  <тип> ::= • integer  <тип> ::= • double  <тип> ::= • decimal | 31  32  33  34 |
| 27 | 15 | <список\_переменных> ::= <список\_переменных> , • id | 35 |
| 28 | 17 | <присвоение> ::= id = expr • | x |
| 29 | 19 | <условие> ::= if <логическое\_условие> then \n • <список\_операторов> end if \n  <условие> :: = if <логическое\_условие> then \n • <список\_операторов> else \n <список\_операторов> end if \n  <список\_операторов> ::= • <оператор> \n  <список\_операторов> ::= • <список\_операторов> <оператор> \n <оператор> ::= • <условие>  <оператор> ::= • <присвоение>  <оператор> ::= • <объявление>  <условие> ::= • if <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = • if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if \n  <объявление> ::= • Dim <список\_переменных> as <тип>  <присвоение> ::= • id = expr | 36  36  2  36  3  4  5  6  6  7  8 |
| 30 | 20 | <логическое\_условие> ::= <операнд> <логический\_знак> <операнд> • | x |
| 31 | 26 | <объявление> ::= Dim <список\_переменных> as <тип> • | x |
| 32 | 26 | <тип> ::= integer • | x |
| 33 | 26 | <тип> ::= double • | x |
| 34 | 26 | <тип> ::= decimal • | x |
| 35 | 27 | <список\_переменных> ::= <список\_переменных> , id • | x |

Продолжение таблицы 1

|  |  |  |  |
| --- | --- | --- | --- |
| 36 | 29 | <условие> ::= if <логическое\_условие> then \n <список\_операторов> • end if \n  <условие> :: = if <логическое\_условие> then \n <список\_операторов> • else \n <список\_операторов> end if \n  <список\_операторов> ::= <список\_операторов> • <оператор> \n  <оператор> ::= • <условие>  <оператор> ::= • <присвоение>  <оператор> ::= • <объявление>  <условие> ::= • if <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = • if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if \n  <объявление> ::= • Dim <список\_переменных> as <тип>  <присвоение> ::= • id = expr | 37  38  9  3  4  5  6  6  7  8 |
| 37 | 36 | <условие> ::= if <логическое\_условие> then \n <список\_операторов> end • if \n | 39 |
| 38 | 36 | <условие> :: = if <логическое\_условие> then \n <список\_операторов> else • \n <список\_операторов> end if \n | 40 |
| 39 | 37 | <условие> ::= if <логическое\_условие> then \n <список\_операторов> end if • | x |
| 40 | 38 | <условие> :: = if <логическое\_условие> then \n <список\_операторов> else \n • <список\_операторов> end if \n  <список\_операторов> ::= • <оператор> \n  <список\_операторов> ::= • <список\_операторов> <оператор> \n <оператор> ::= • <условие>  <оператор> ::= • <присвоение>  <оператор> ::= • <объявление>  <условие> ::= • if <логическое\_условие> then \n <список\_операторов> end if \n  <условие> :: = • if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if  <объявление> ::= • Dim <список\_переменных> as <тип>  <присвоение> ::= • id = expr | 41  2  41  3  4  5  6  6  7  8 |
| 41 | 40 | <условие> :: = if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> • end if | 42 |
| 42 | 42 | <условие> :: = if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end • if | 43 |
| 43 | 43 | <условие> :: = if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if • | x |

На основе таблицы 1, сформируем решающая таблицу восходящего анализатора, для разработки логики работы транслятора.

Таблица 2 – Решающая таблица LR-анализатора

|  |  |  |  |
| --- | --- | --- | --- |
| Состояние | Стек разбора | Вход | Действие |
| 0 | <программа>  E  <список\_операторов>  <оператор>  <условие>  <присвоение>  <объявление>  if  Dim  id |  | →Конец  Сдвиг  →1  →2  →3  →4  →5  →6  →7  →8 |
| 1 | <список\_операторов>  <список\_операторов>  <оператор>  <условие>  <присвоение>  <объявление>  if  Dim  id | $  if | Dim | id | Свёртка(-1, <программа>)  Сдвиг  →9  →3  →4  →5  →6  →7  →8 |
| 2 | <оператор>  \n |  | Сдвиг  →10 |
| 3 | <условие> |  | Свёртка (-1, <оператор>) |
| 4 | <присвоение> |  | Свёртка (-1, <оператор>) |
| 5 | <объявление> |  | Свёртка (-1, <оператор>) |
| 6 | if  <логическое\_условие>  <операнд>  id  lit |  | Сдвиг  →11  →12  →13  →14 |
| 7 | Dim  <список\_переменных>  id |  | Сдвиг  →15  →16 |
| 8 | id  = |  | Сдвиг  →17 |
| 9 | <оператор>  \n |  | Сдвиг  →18 |

Продолжение таблицы 2

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | <оператор>  \n |  | Свёртка (-2, < список\_операторов >) |
| 11 | <логическое\_условие>  then |  | Сдвиг  →19 |
| 12 | <операнд>  <логический\_знак>  >  <  =  >=  <= |  | Сдвиг  →20  →21  →22  →23  →24  →25 |
| 13 | id |  | Свёртка (-1, <операнд>) |
| 14 | lit |  | Свёртка (-1, <операнд>) |
| 15 | <список\_переменных>  as  , |  | Сдвиг  →26  →27 |
| 16 | id |  | Свёртка (-1, <список\_переменных>) |
| 17 | =  expr |  | Сдвиг  →28 |
| 18 | <список\_операторов> <оператор> \n |  | Свёртка (-3, <список\_операторов>) |
| 19 | then  \n |  | Сдвиг  →29 |
| 20 | <логический\_знак>  <операнд>  id  lit |  | Сдвиг  →30  →13  →14 |
| 21 | > |  | Свёртка (-1, <логический\_знак>) |
| 22 | < |  | Свёртка (-1, <логический\_знак>) |
| 23 | = |  | Свёртка (-1, <логический\_знак>) |
| 24 | >= |  | Свёртка (-1, <логический\_знак>) |

Продолжение таблицы 2

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | <= |  | Свёртка (-1, <логический\_знак>) |
| 26 | as  <тип>  integer  double  decimal |  | Сдвиг  →31  →32  →33  →34 |
| 27 | ,  id |  | Сдвиг  →35 |
| 28 | id = expr |  | Свёртка (-3, <присвоение>) |
| 29 | \n  <список\_операторов>  <оператор>  <условие>  <присвоение>  <объявление>  if  Dim  id |  | Сдвиг  →36  →2  →3  →4  →5  →6  →7  →8 |
| 30 | <операнд> <логический\_знак> <операнд> |  | Свёртка (-3, <логическое\_условие>) |
| 31 | Dim <список\_переменных> as <тип> |  | Свёртка (-4, <объявление>) |
| 32 | integer |  | Свёртка (-1, <тип>) |
| 33 | double |  | Свёртка (-1, <тип>) |
| 34 | decimal |  | Свёртка (-1, <тип>) |
| 35 | <список\_переменных> , id |  | Свёртка (-3, <список\_переменных>) |
| 36 | <список\_операторов>  end  else  <оператор>  <условие>  <присвоение>  <объявление>  if  Dim  id |  | Сдвиг  →37  →38  →9  →3  →4  →5  →6  →7  →8 |

Продолжение таблицы 2

|  |  |  |  |
| --- | --- | --- | --- |
| 37 | end  if |  | Сдвиг  →39 |
| 38 | else  \n |  | Сдвиг  →40 |
| 39 | if <логическое\_условие> then \n <список\_операторов> end if |  | Свёртка (-7, <условие>) |
| 40 | \n  <список\_операторов>  <оператор>  <условие>  <присвоение>  <объявление>  if  Dim  id |  | Сдвиг  →41  →2  →3  →4  →5  →6  →7  →8 |
| 41 | <список\_операторов>  end |  | Сдвиг  →42 |
| 42 | end  if |  | Сдвиг  →43 |
| 43 | if <логическое\_условие> then \n <список\_операторов> else \n <список\_операторов> end if |  | Свёртка (-10, <условие>) |

На основе построения решающей таблицы, можно сказать, что построенная грамматика относится к классу LR (1), так как для дальнейшего определения действия в некотором случае транслятору необходимо знать одну лексему наперёд.

3 Разработка архитектуры системы и алгоритмов

Для реализации транслятора с подмножеством для языка Visual Basic требуется разработать алгоритм лексического, синтаксического анализатора, а также разбор сложного арифметического выражения.

3.1 Описание работы лексического анализатора

1. На вход алгоритм получает исходную строку, которую в дальнейшем считывает посимвольно.

2. До тех пор, пока не будет достигнут конец входной строки или получена ошибка лексического анализа, считываем очередной символ анализируемой строки, определяем и переходим из текущего состояния диаграммы в другое, выполняя при этом соответствующие действия. Состояние, в которое попадаем, на данный момент, становится текущим.

3. Выходной строка с лексемой формируется в состояния, где выходная строка удовлетворяет определённым условиям. Тип лексемы изменятся в зависимости от текущего состояния. В каждом состояние выходная лексема проверяется на наличие ошибок, если встретилась ошибка – работа лексического анализатора прекращается.

При успешном проведенном лексическом анализе, анализатор вернёт список всех найденных лексем, в виде отношения: Лексема – тип. Для первого этапа разбора предварительные типы лексем: идентификатор, разделитель, литерал (Рисунок 1).

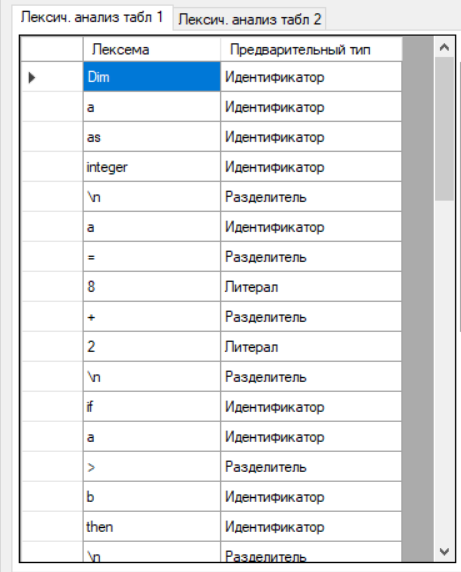


Рисунок 1 – Результат первого этапа лексического анализа

На следующем этапе происходит разбиение идентификаторов на ключевые слова и переменные, а также формирование двух таблиц: таблица стандартных символов, таблицу, содержащую отдельно ключевые слова, литералы, идентификаторы, разделители (Рисунок 2).

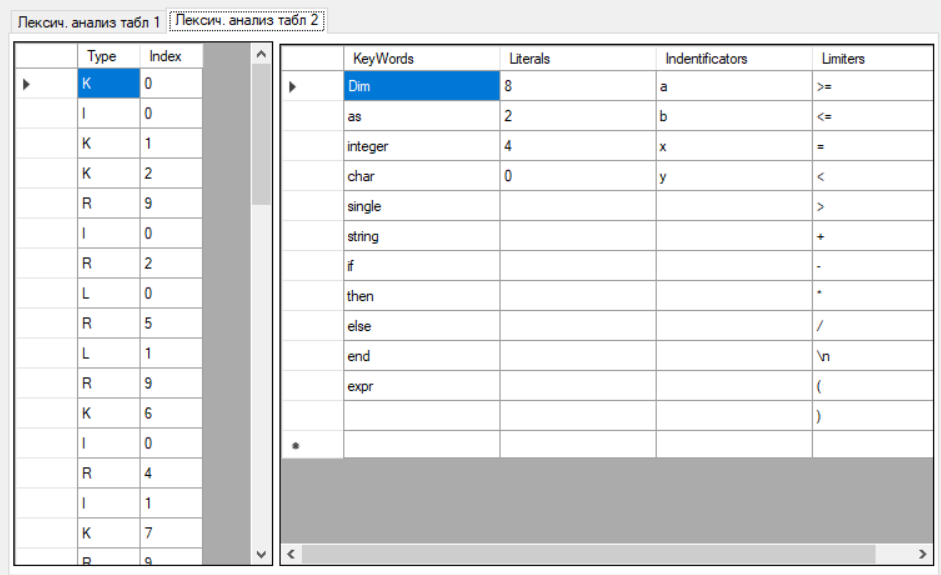


Рисунок 2 – Результат второго этапа лексического анализа

3.2 Синтаксический анализатор

Логика работы синтаксического анализатора вынесена в отдельный класс. В классе описаны методы состояний синтаксического анализатора, а также вспомогательные методы для его работы. Каждый метод-состояние имеет свою логику поведения в соответствии с решающей таблицей Lr-анализатора (таблица 2).

При LR(k)-анализе применяется метод "перенос-свертка" (shift-reduce). Этот метод использует магазинный автомат. Суть метода сводится к следующему. Символы входной цепочки переносятся в магазин до тех пор, пока на вершине магазина не накопится цепочка, совпадающая с правой частью какого-нибудь из правил (операция "перенос", "shift"). Далее все символы этой цепочки извлекаются из магазина и на их место помещается нетерминал, находящийся в левой части этого правила (операция "свертка", "reduce"). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнения операции свертка, в магазине окажется только аксиома грамматики.

3.3 Транслятор сложного арифметического выражений

Разработка транслятора сложных арифметических выражений в соответствии с техническим заданием курсовой работы происходит методом Дейкстры.

Данный метод, известный также как алгоритм сортировочной станции, основан на использовании стека с приоритетами. Выражение просматривается слева направо. Операнды переписываются в выходную строку, а знаки операции помещаются вначале в стек операций. Если приоритет входного знака равен нулю или больше приоритета знака, находящегося на вершине стека, то новый знак добавляется к вершине стека. В противном случае из стека выталкивается и переписывается в выходную строку знак находящийся на вершине, а также следующие за ним знаки с приоритетами большими или равными приоритету входного знака. После этого входной знак добавляется к вершине стека. Открывающаяся скобка просто записывается в стек и не выталкивает ни одного знака. Но и её не может вытолкнуть ни один знак, кроме закрывающейся скобки. Появление закрывающейся скобки вызывает выталкивание всех знаков до ближайшей отрывающейся скобки включительно. Скобки взаимно уничтожаются и в выходную строку не переносятся. После просмотра всех символов входной строки происходит выталкивание всех оставшихся в стеке символов и дописывание их к выходной строке. Таблица приоритетов для арифметических операций приведена на рисунке 3.



Рисунок 3 – Приоритеты арифметических операторов

Если выражение верно и удовлетворяет правилам написания выражения, то пользователь увидит этапы разбор арифметических выражений как на рисунке 4.

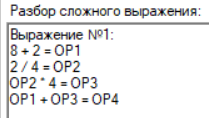


Рисунок 4 – Разбор арифметического выражений

4 Методика испытаний

Для выявления корректной работы всего транслятора потребуется отдельно проверить работу каждого анализатора, в частности.

4.1 Испытание лексического анализатора

Проверка лексического анализатора осуществляется путём ввода в кода на языке Visual Basic. Ввод корректных лексем, заполнит пустые таблицы приложения данными как на рисунке 5-6

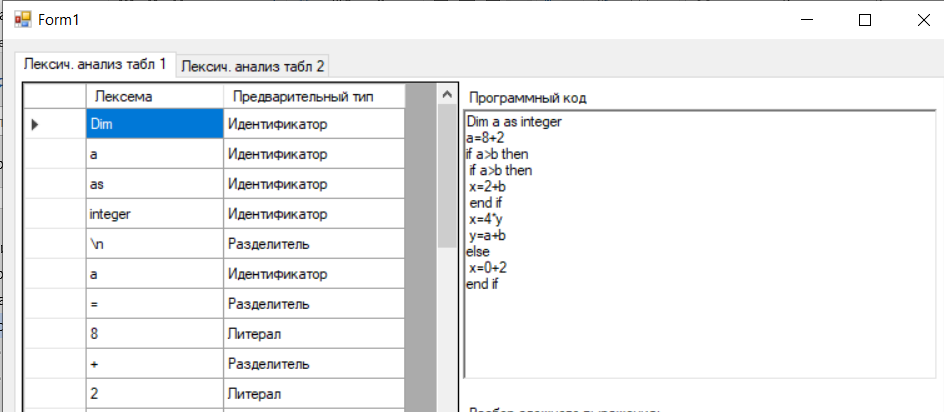


Рисунок 5 – успешный первый этап лексического разбора

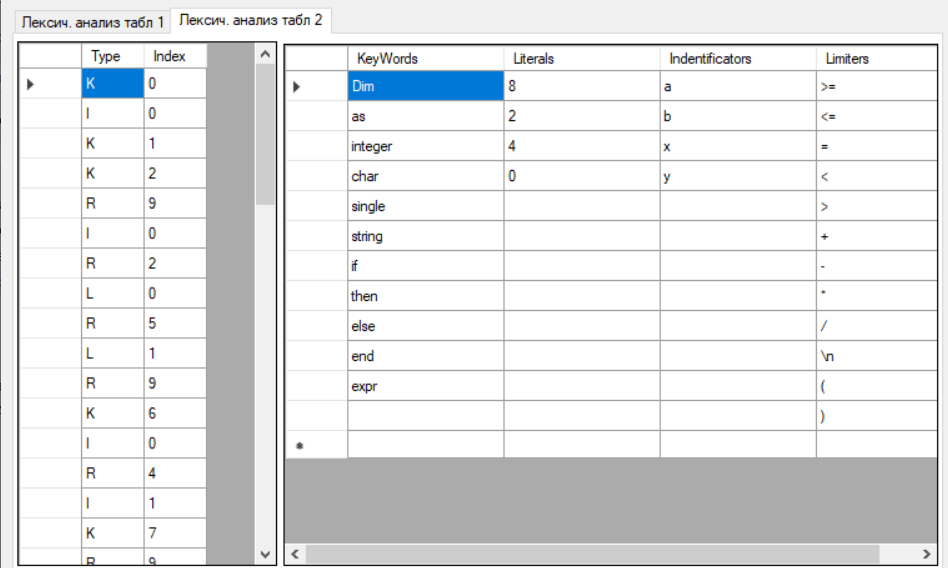


Рисунок 6 – Второй этап успешного лексического анализа

При вводе символа, неразрешённого грамматикой языка, выводится сообщение с ошибкой в виде диалогового окна (рисунок 7).

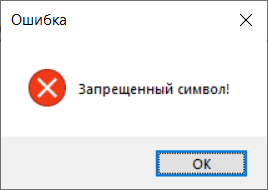


Рисунок 7 – Диалоговое окно с ошибкой

4.2 Испытание синтаксического анализатора

Успешное выполнение лексического анализатора запустит проверку синтаксическим анализатором. Если последовательность исходной строки соответствует грамматике анализатора и языка в целом, тогда появится диалоговое окно, которое свидетельствует об успешном синтаксическом анализе (рисунок 8).

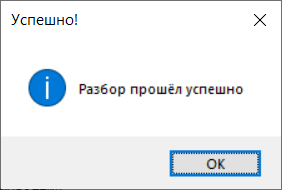


Рисунок 8 – Сообщение о успешном выполнении синтаксического анализа

Введение некорректной лексемы приведёт к ошибке, которая выведет номер строки где была обнаружена ошибка, а также что встретилось и что требовалось.

В ходе тестирования неверных лексем получены следующие ошибки:

1. Указание неверного типа инициализируемых переменных. Ошибка должна быть обнаружена в 26 состоянии:

Таблица 3 – Структура состояния 26

|  |  |  |  |
| --- | --- | --- | --- |
| 26 | as  <тип>  integer  double  decimal |  | Сдвиг  →31  →32  →33  →34 |

Транслятор ожидает на вход лексемы integer или double или string, но так как встретилась ошибочная лексема получаем окно с информацией об ошибке (рисунок 9).

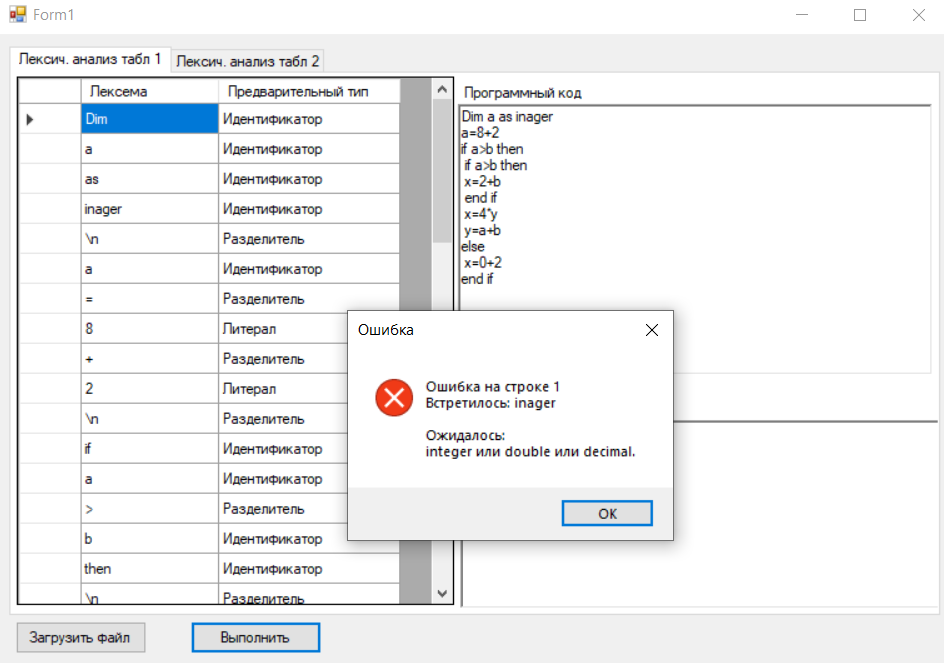


Рисунок 9 – Ошибка синтаксического анализатора

1. Удаление обязательной лексемы условной конструкции. Ошибка должна быть обнаружена в 8 состояние

Таблица 4 – Структура состояния 8

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | id  = |  | Сдвиг  →17 |

Транслятор ожидает на вход лексему =, но так как встретилась ошибочная лексема получаем окно с информацией об ошибке (рисунок 10).

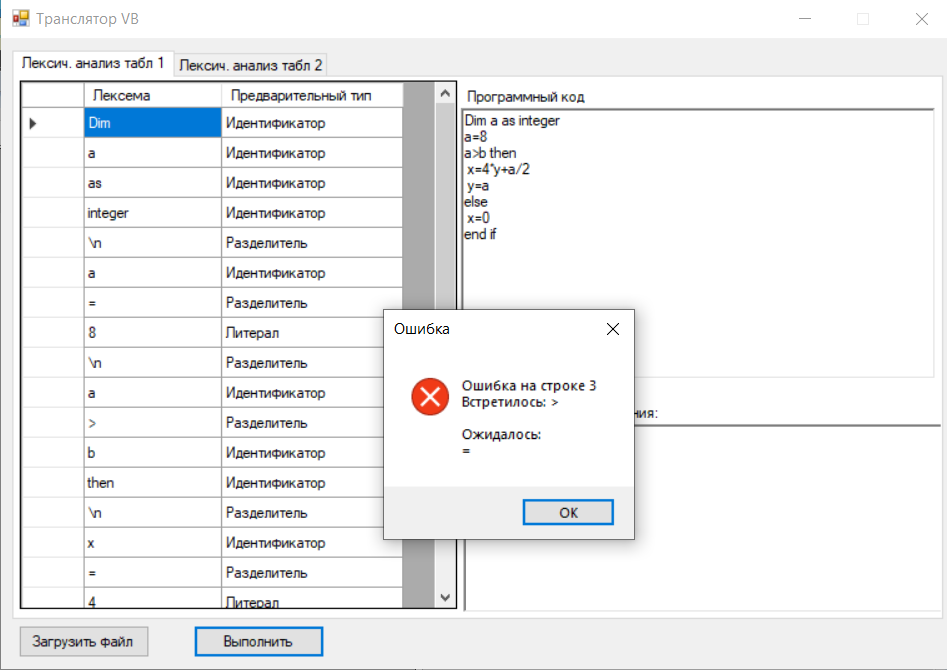


Рисунок 10 – Ошибка синтаксического анализатора

1. Нарушение целостности условной конструкции. Ошибка должна быть обнаружена в 11 состоянии:

Таблица 4 – Структура состояния 11

|  |  |  |  |
| --- | --- | --- | --- |
| 11 | <логическое\_условие>  then |  | Сдвиг  →19 |

Транслятор ожидает на вход лексемы then, но так как встретилась ошибочная лексема получаем окно с информацией об ошибке (рисунок 11).

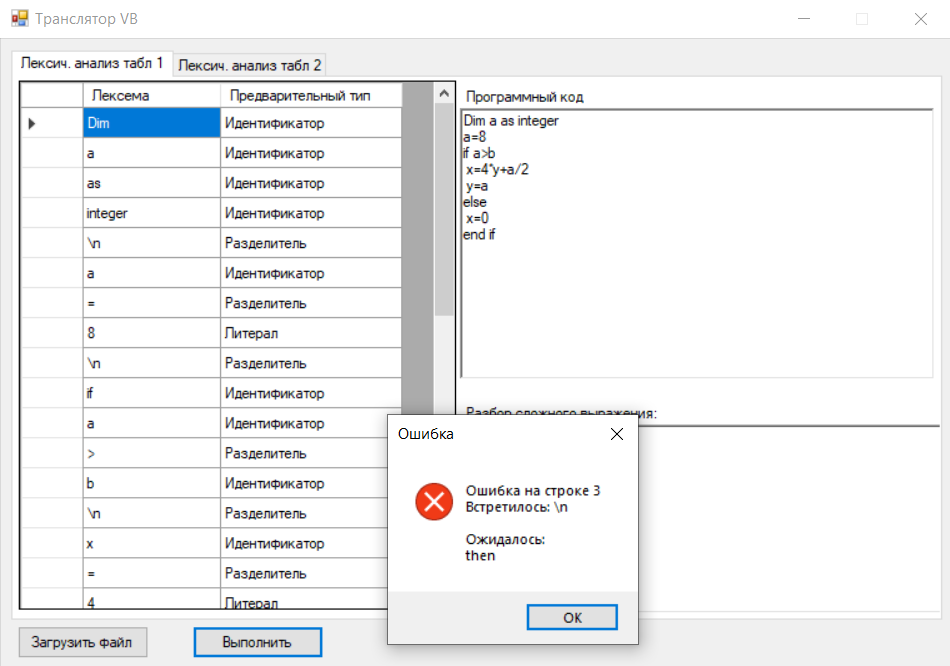


Рисунок 11 – Ошибка синтаксического анализатора

4.3 Испытание анализатора арифметических выражений

Проверка корректности работы анализатора арифметических выражений, осуществляется путём ввода сложного арифметического выражения, которое содержит случайный порядок операций и лексем.

Если введена правильная последовательность арифметического оператора, то анализатор выведет лог событий разбора арифметических операций (рисунок 12).

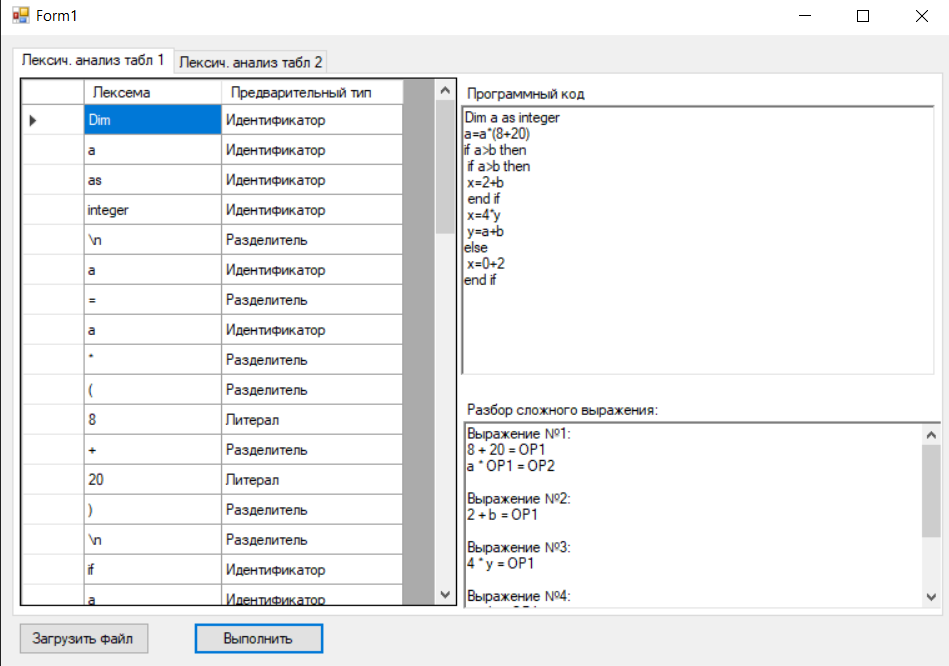


Рисунок 12– Успешный разбор арифметического выражения

Если последовательность арифметического выражения не верна, то выведется сообщение как на рисунке 10.

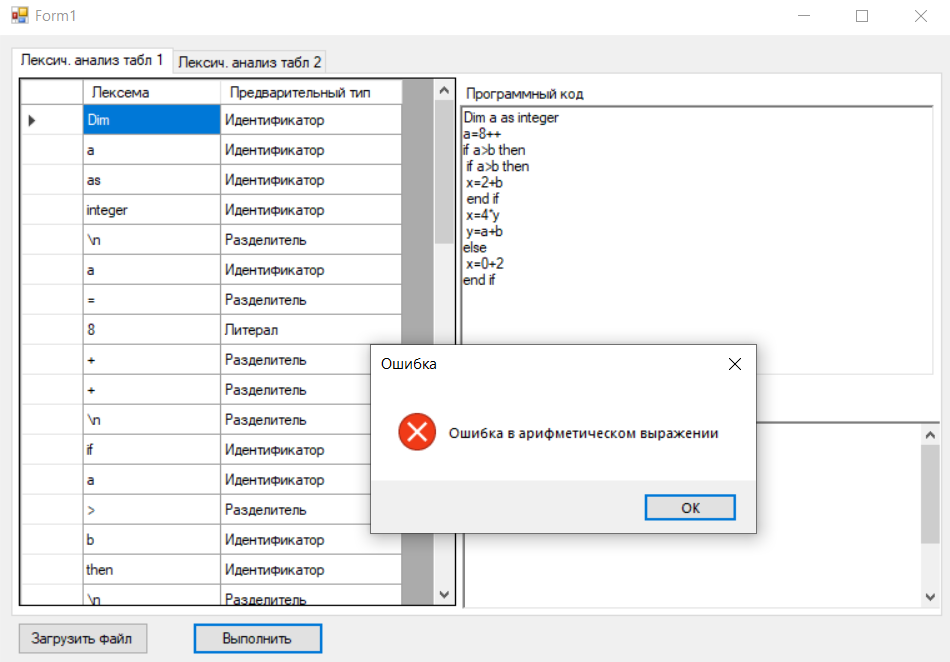


Рисунок 13 – Сообщение об ошибке в арифметическом выражени

Если при вводе сложного оператора была потеряна скобка появится сообщение с ошибкой о недостающей открывающейся или закрывающейся скобке, соответственно на рисунке 14 и 15.

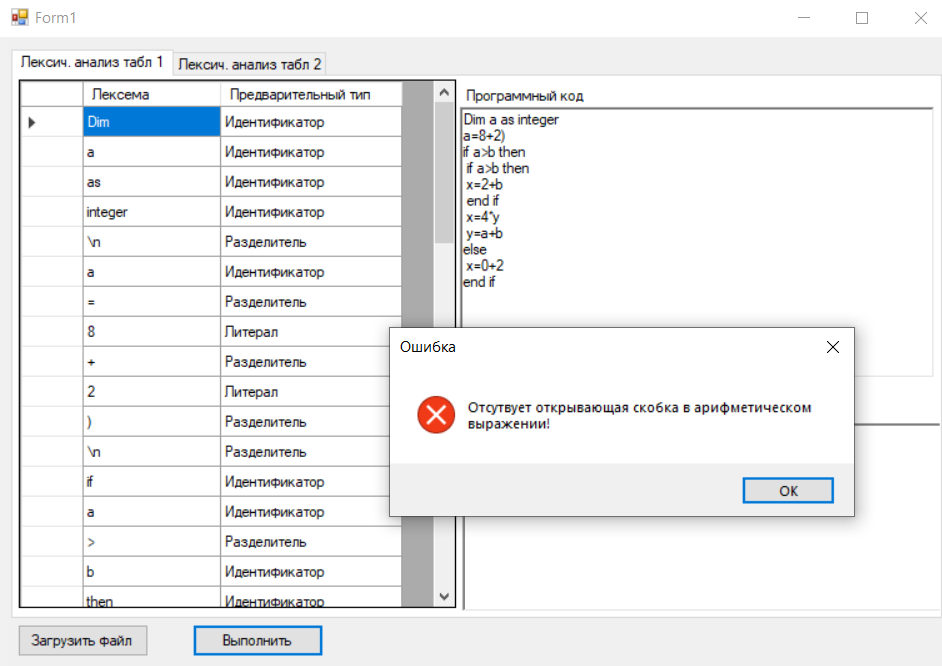


Рисунок 14 – Сообщение о недостающей открывающей скобке

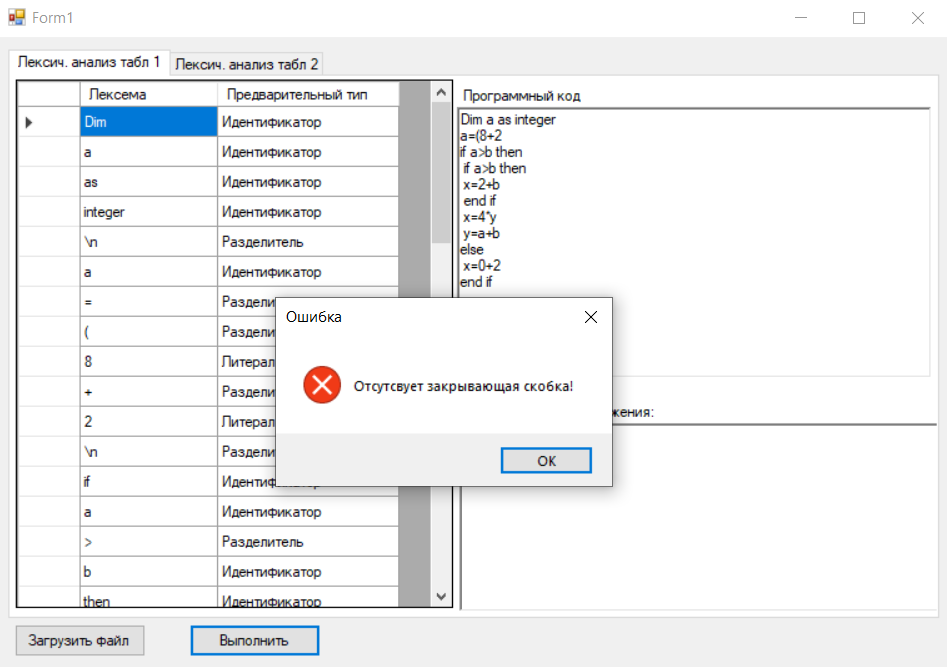


Рисунок 15 – Сообщение о недостающей закрывающей скобке

# 5 Руководство пользователя

Запуск исполняемого файла приложения откроет главное окно с программой (рисунок 16). Пользователь может взаимодействовать с приложением благодаря кнопкам «Загрузить файл», «Выполнить». Кнопка «Выполнить» запустит транслятор, если на разных этапах анализа введённого кода не произошло ошибок, то пользователь увидит заполненные таблицы, а также лог разбора сложного выражения, если таковое имеется.

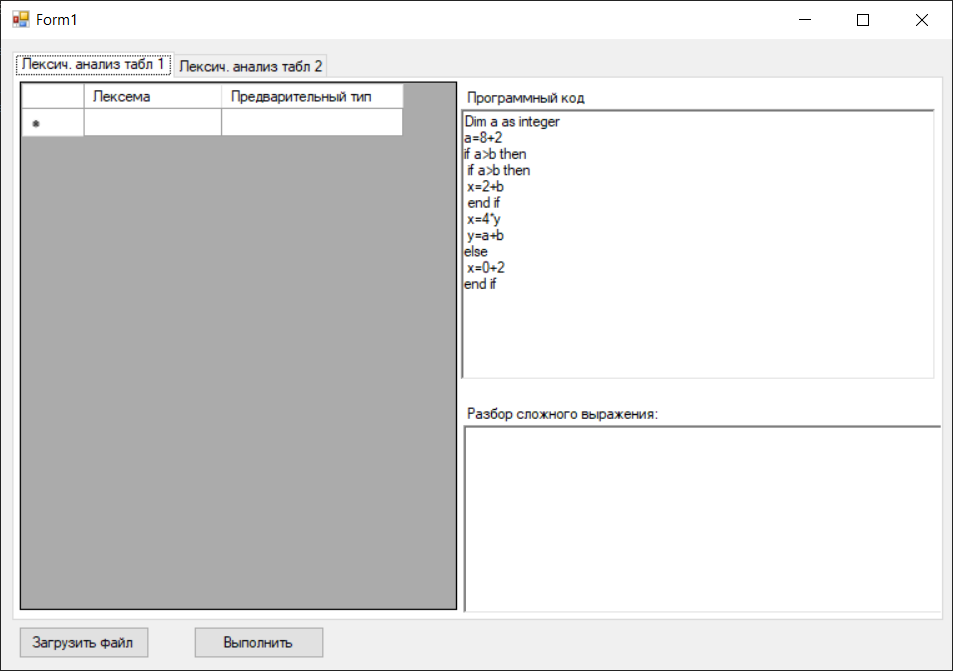


Рисунок 16 – Главное окно приложения

Для того чтобы посмотреть вторую часть (рисунок 17) лексического анализа следует нажать на вкладку «Лексич. Анализ табл 2» на рисунке 16.

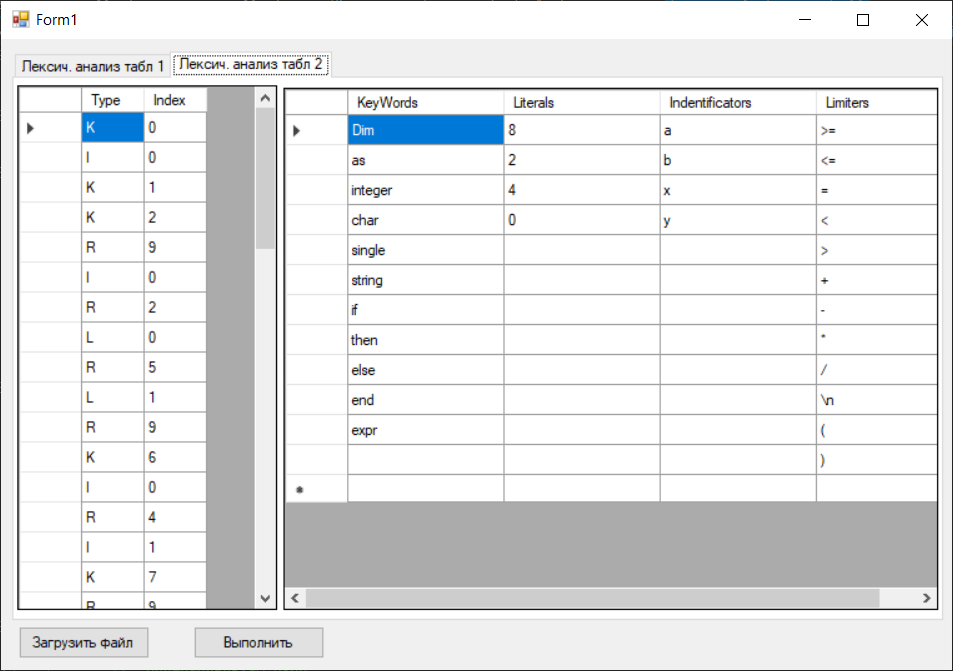


Рисунок 17 – Вторая часть лексического анализа

Кнопка «Загрузить файл» откроет новое диалоговое окно, где пользователь может считать код из файла формата txt (рисунок 18).

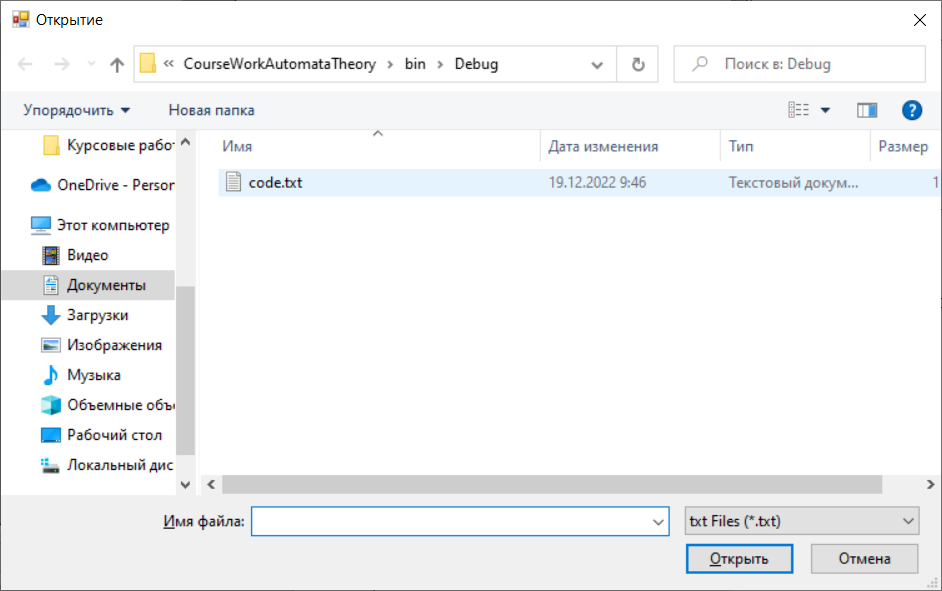


Рисунок 18 – Диалоговое окно выбора txt файла

# 6 Руководство программиста

Исходный код разбит на соответствующие классы. Внутри класса содержатся поля, методы, необходимые для работы приложения. В руководстве программиста представлено описание основных методов и полей класса.

6.1 Диаграмма классов

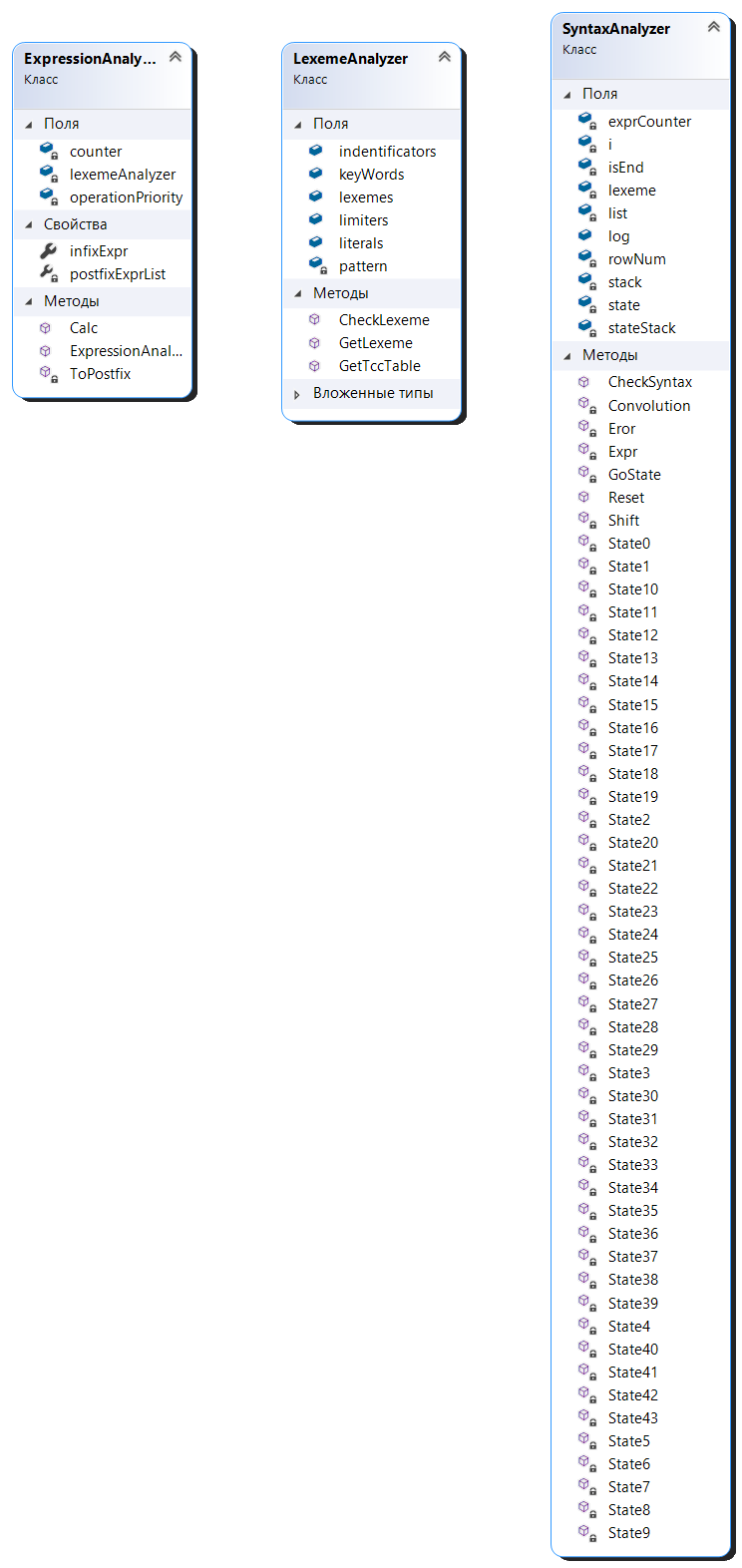


Рисунок 19 – Диаграмма классов

6.2 Класс LexemeAnalyzer.cs

Класс LexemeAnalyzer.cs отвечает за алгоритм работы лексического анализатора.

Поля класса:

readonly string pattern = @"\p{IsCyrillic}" – маска русских символов.

public readonly List<string> literals = new List<string>() - лист литералов.

public readonly List<string> indentificators = new List<string>() – лист идентификаторов.

public readonly List<string> limiters = new List<string>() {">=", "<=", "=", "<", ">", "+", "-", "\*", "/", "\n", "(",")"} – лист разделителей.

public readonly List<string> keyWords = new List<string>() { "Dim", "as", "integer", "char", "single", "string", "if", "then", "else", "end", "expr" } – лист ключевых слов.

public readonly List<Tuple<string, string>> lexemes = new List<Tuple<string, string>>() – лист выходных лексем.

private enum TypeOfLexe { I, L, R} – типы существующих лексем.

Методы класса:

public List<Tuple<string, string>> GetLexeme(string text) – метод для проведения первого этапа лексического анализа, на вход принимает строку с кодом, возвращает лист обработанных лексем.

private string CheckLexeme(char sym) – метод, принимающий на вход символ из входной строки, возвращает тип символа.

public List<Tuple<string, int>> GetTccTable() – метод реализующий второй этап лексического анализа, возвращает лист лексем в новом формате.

6.3 Класс SyntaxAnalyzer.cs

int exprCounter = 1 – счётчик сложных арифметических выражений

int i – индекс для прохода по листу лексем.

int state = 0 – текущее состояние транслятора.

int rowNum = 1 – счётчик строк.

public string log – хранит в себе информацию о разобранных сложных арифметических выражениях.

List<Tuple<string, int>> list – лист лексем.

LexemeAnalyzer lexeme – хранит в себе экземпляр класса LexemeAnalyzer.

bool isEnd = false – останавливает цикл, если синтаксический разбор прошёл успешно.

readonly Stack<string> stack = new Stack<string>() – стек терминалов и нетерминалов.

readonly Stack<int> stateStack = new Stack<int>() – стек состояний.

Методы класса:

public void CheckSyntax(List<Tuple<string, int>> list, LexemeAnalyzer lexeme) – метод проверки кода на верный синтаксис.

public void Reset() – метод сброса полей, если программа выдала ошибку или разбор был успешно завершён.

private void Shift() – метод, реализующий операцию «Сдвиг» в логике работы транслятора.

private void GoState(int state) – метод перехода к новому состоянию транслятора, принимает на вход новое состояние.

private void Convolution(int count, string NotATerminal) - метод, реализующий операцию «Свёртка» в логике работы транслятора, принимает на вход размер свёртки и название нетерминала.

private void Expr() – метод разбора арифметических выражений.

private void Eror(string text) – метод обработки ошибки.

private void State0-State43 – методы, реализующие логику работы транслятора в соответствии с решающей таблицей восходящего анализатора.

6.4 Класс ExpressionAnalayzer.cs

В данном классе реализована алгоритм разбора арифметического выражения.

Поля класса:

int counter = 0 – счётчик операций сложного арифметического выражения.

readonly LexemeAnalyzer lexemeAnalyzer = new LexemeAnalyzer() – экземпляр класса LexemeAnalyzer.

private List<string> infixExpr – лист, хранящий сложное арифметическое выражение.

List<string> postfixExprList лист, хранящий сложное арифметическое выражение в постфиксном формате.

private readonly Dictionary<string, int> operationPriority = new Dictionary<string, int>(){ { "(", 0 }, { "+", 1 }, { "-", 1 }, { "\*", 2 }, { "/", 2 }, { "^", 3}}; - словарь, хранящий приоритеты операндов.

Методы класса:

private List<string> ToPostfix(List<string> infixExpr) – метод преобразует арифметическое выражение в постфиксный формат, принимает на вход лист с арифметическим выражением.

public string Analyze() – метод, проводящий разбор арифметического выражения в постфиксном формате.

# Заключение

На основе проведенного анализа технического задания, в ходе курсовой работы был разработан транслятор с подмножества языка Visual Basic, соответствующий всем требования технического задания. Приложение реализовано в среде разработки Visual Studio при помощи типизированного, объектно-ориентированного языка программирования #C.

Разработанное приложение производит лексический и синтаксический анализ, на основе LR(1) грамматики, так же разбор сложного арифметического выражения методом Дейкстры.

Над разработанным приложением проведены тесты, результаты тестирования соответствуют ожидаемым результатам.

Программа разработана для операционной системы Windows и не требует дополнительного программного обеспечения для работы

# Список литературы

1. Ожиганов А.А. Теория автоматов. Учебное пособие. – СПб: НИУ ИТМО, 2013. – 84 с.

2. Выхованец В.С. Теория автоматов: Учеб. пособие для вузов. – Тирасполь, РИО ПГУ, 2001. - 87 с.: ил.

3. Малявко, А. А. Формальные языки и компиляторы: учебник / А. А. Малявко. — Новосибирск: Новосибирский государственный технический университет, 2014. — 431 c.

4. Карпов В.Э. К26 Теория компиляторов. Учебное пособие. 2-е изд., испр. и дополн. М., 2018. – 92 с.

# Приложение. Ссылка на репозиторий

Программный код проекта размещён на GitHub по следующей ссылке:

https://github.com/feniletilamin1/CourseWorkAutomataTheory