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-121

(группа)

. . Карпов Д.В.

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

. . .

( подпись) (Ф.И.О.) (подпись) (дата)

Муром 2023

Место для задания

В данной курсовой работе описан процесс создания транслятора с подмножества языка Visual Basic. Для создания приложения использовался язык программирования С#, а также среда разработки Visual Studio 2022. Данный транслятор создан под OC Windows.

This course work describes the process of creating a translator from a subset of the Visual Basic language. To create the application, the C# programming language was used, as well as the Visual Studio 2022 development environment. This translator was created 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 16](#_Toc122914457)

[4 Методика испытаний 21](#_Toc122914458)

[5 Руководство пользователя 27](#_Toc122914459)

[6 Руководство программиста 28](#_Toc122914460)

[Заключение 36](#_Toc122914461)

[Список литературы 37](#_Toc122914462)

[Приложение А. Ссылка на репозиторий 38](#_Toc122914463)

# Введение

С появлением первых компьютеров программисты серьезно задумались над проблемой кодирования компьютерных программ. Уже с конца 40-х годов стали появляться первые примитивные языки программирования высокого уровня. В них решаемая задача записывалась в виде математических формул, а затем переводилась символ за символом в коды. В дальнейшем специальные программы превращали эти коды в двоичный машинный код. Такой программой является транслятор.

Транслятор – это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на ре- зультирующем (выходном) языке. То есть транслятор – это часть программного обеспечения, который представляет собой набор машинных команд и данных и выполняется компьютером. Все составные части транслятора представляют собой фрагменты и модули программы со своими входными и выходными данными.

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

Целью данной курсовой работы является создание такого транслятора с подмножества языка Visual Basic.Чтобы создать транслятор, необходимо прежде всего построить грамматику входного языка, произвести лексический, синтаксический анализ, а также совершить разбор арифметических и логических выражений.

Актуальность данной темы заключается в учебных целях. Понять, как происходит процесс преобразования программы в программу на другом языке, как выполняется лексический анализ, как формируется таблица лексем, как осуществляется синтаксический разбор, узнать и реализовать метод разбора сложных логических выражений.

# 1 Анализ технического задания

В данной курсовой работе необходимо разработать транслятор с подмножества языка Vusual Basic. Программа будет разрабатываться в среде разработки Visual Studio 2022 с использованием высокоуровневого объектно-ориентированного языка программирования C#. Выбор именно этих инструментов для разработки указанного приложения основывается на том, что Visual Studio 2022 обладает удобным дизайнером форм, который позволяет распологать на форме различные компоненты, начиная с кнопок и текстовых полей, заканчивая таблицами и изображениями. Язык программирования C# выбран по следующим причинам: он бесплатный, а также ориентирован на платформу Windows.

Для создания графической интерфейса приложения будет использоваться технология Windows Forms. Эта технология имеет следующие особенности:

* наличие удобного дизайнера форм в среде разработки Microsoft Visual Studio 2022;
* наличие в .NET Framework достаточного набора графических компонентов для реализации пользовательского интерфейса, обеспечивающего требуемый функционал программы.

К разрабатываемому приложению были выдвинуты следующие требования:

* обеспечить развернутую диагностику ошибок;
* реализовать класс транслятора;
* реализовать синтаксический разбор - на основе LL(k)-грамматик;
* выполнить разбор логических выражений методом Бауэра-Замельзона;
* в языке поддерживаются: у идентификаторов 8 символов значащие; не менее 3-х директив описания переменных; простой арифметический оператор; сложное логическое выражение; оператор цикла DO WHILE … LOOP

Все необходимые разборы будут выполняться на заготовленном заранее фрагменте кода. Ниже представлен пример фрагмента кода:

Dim a as integer

b=1

do while (a < 10 or b > c)

b=b+a

loop

В начале работы будет написана часть программы, которая будет производить лексический анализ и классификацию лексем. Это поможет разобрать на лексические единицы анализируемый код, что в дальнейшем позволит проще производить синтаксический анализ.

Далее необходимо будет построить грамматику языка, для основы которой будет браться вышеуказанный фрагмент кода, но также грамматика должна учитывать возможность вложенности условных операторов, операторов объявления переменных и операторов присваивания в условном операторе. На основе построенной грамматики будет построена новая грамматика, которая должна будет принадлежать к классу LL(k). Эта грамматика требуется для реализации метода рекурсивного спуска.

На основе LL(k)-грамматики будет реализован синтаксический анализ. Данный анализ будет проводиться методом рекурсивного спуска. Каждому правилу грамматики будет соответствовать своя подпрограмма (метод), которая будет последовательно проверять текущие лексемы с ожидаемыми. Если результат будет отличаться от ожидаемого, то анализ прекратиться, и будет выведена соответствующая ошибка. Метод рекурсивного спуска будет также проверять правильность простого арифметического оператора.

Заключительным этапом работы будет являться разбор сложных логических выражений методом Бауэра-Замельзона. Результатом этого разбора будет являться матрица логического оператора.

# 2 Описание грамматики языка

2.1 Описание языка Visual Basic

Visual Basic - язык программирования, разработанный компанией Microsoft. Язык Visual Basic унаследовал дух, стиль и отчасти синтаксис своего предка — языка BASIC, у которого есть немало диалектов. В то же время Visual Basic сочетает в себе процедуры и элементы объектно-ориентированных и компонентно-ориентированных языков программирования. Интегрированная среда разработки VB включает инструменты для визуального проектирования пользовательского интерфейса, редактор кода с возможностью IntelliSense и подсветкой синтаксиса, а также инструменты для отладки приложений.

Visual Basic также является хорошим средством быстрой разработки (RAD) приложений баз данных для операционных систем семейства Microsoft Windows. Множество готовых компонентов, поставляемых вместе со средой, призваны помочь программисту сразу же начать разрабатывать бизнес-логику бизнес-приложения, не отвлекая его внимание на написание кода запуска программы, подписки на события и других механизмов, которые VB реализует автоматически.

2.2 Оператор объявления переменных

Объявление переменных – процесс введения новых переменных в программу. Объявление включает в себя указание идентификатора, типа и, при необходимости, указание начального значения.

Структура оператора в языке VB:

Dim <Названия переменных> As <Тип> [= Значение]

В блоке <Названия переменных> могут писаться одна переменная или список переменных, разделённый между собой запятой. Блок [= Значение] необязателен и может быть представлен переменной, значением, согласно типу, или выражением.

2.3 Оператор присваивания

Присваивание - это запись данных в участок памяти компьютера, отведенной для значения величины, тех данных, которые хранятся в другом участке памяти компьютера, где записано значение величины, то есть оператор присваивания служит для изменения значения переменной.

Структура оператора: <переменная>=<выражение>

Слева от знака «=» в операторе присваивания записывается имя той переменной, которой нужно присвоить новое значение, а справа - выражение, которое может быть представлено как переменная, определенное значение или арифметическое выражение, результат которого необходимо вычислить. Значение, хранящееся в блоке <выражение>, присваивается переменной.

2.4 Условный оператор

Оператор цикла DO WHILE … LOOP в языке Visual Basic позволяет выполнять определенные команды в цикле до тех пор, пока некоторое логическое выражение (условие) принимает значение «истина».

Структура оператора цикла DO WHILE … LOOP следующая:

Do While <условие><команды> Loop

При выполнении оператора цикла DO WHILE ... LOOP сначала проверяется логическое выражение <условие>. Если оно истинно, то выполняются команды, находящиеся между операторами DO WHILE и LOOP, а затем проверяется условие снова. Цикл продолжается до тех пор, пока условие остается истинным. Если условие становится ложным, то выполнение цикла прекращается и управление передается следующей инструкции после оператора LOOP.

Условие может быть представлено простым или сложным логическим выражением.

2.5 Грамматика языка

Для разработки решающего автомата необходимо продумать грамматику подмножества языка Visual Basic, включающую:

* Терминалы;
* Нетерминалы;
* Правила языка;
* Начальное правило языка.

В результате анализа вышеописанных конструкций языка Visual basic была создана следующая грамматика:

G = {T, N, P, <программа>}

T = { Short, Integer, Double, Dim, As, If, Then ,Else, End, Sub,Main, And, Or, Xor, lit, id, expr,<, >, =, +, -, \*, /, (, ) , /-, <=, >=, (), <>, ,}

N = {<программа>, <список\_действий>, <действие>, <список\_описаний>, <присваивание>, <условный\_оператор>, <операнд>, <знак>, <описание>, <список\_переменных>, <тип>, <блок\_действий>}

P = {<Программа> ::= Sub Main() /- <список\_действий>/- End Sub

<список\_действий> ::= <действие>| <список\_действий>/- <действие>

<действие> ::= <описание> | <присваивание> | <условный\_оператор>

<присваивание> ::= id = <операнд><знак><операнд>| id = <операнд>

<знак> ::= + | - | \* | /

<операнд> ::= id | lit

<описание> ::= Dim <список\_переменных> As <тип>| Dim <список\_переменных> As <тип> = <операнд><знак><операнд> | Dim <список\_переменных> As <тип> = <операнд>

<список\_переменных> ::= id | <список\_переменных>, id

<тип> ::= Short | Integer | Double

<условный\_оператор> ::= If expr Then/- <список\_действий>/- End If | If expr Then/- <список\_действий>/- Else/- <список\_действий>/- End If

Здесь знак /- отвечает за конец строки, lit – литерал, id – идентификатор, expr-это функция для анализа сложного логического выражения.

2.6 Грамматика языка для синтаксического анализа

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

Для построения LL(k)-грамматики обязательным условием является устранение леворекурсивных правил по специальному алгоритму, а также устранение левой факторизации – лишнего дублирования правил.

Процесс устранения леворекурсивных правил и левой факторизации для правил языка описан ниже:

Устранение левой рекурсии для <список\_действий> :

<список\_действий>::= <действие> | <действие><устранение\_рек\_действия>

<устранение\_рек\_действия>::=/-<действие>| /-<действие> <устранение\_рек\_действия>

Устранение левой факторизации:

<список\_действий>::= <действие><устр\_лев\_фактор\_спис\_действий>

<устр\_лев\_фактор\_спис\_действий>::= E | <устранение\_рек\_действия>

<устранение\_рек\_действия>::=/-<действие> <устр\_лев\_фактор\_спис\_действий>

Устранение левой факторизации <присваивание>:

<присваивание>::= id = <операнд><устр\_лев\_фактор\_присваивание>

<устр\_лев\_фактор\_присваивание>::= <знак><операнд> | E

Устранение левой факторизации <описание>:

<описание>::=Dim<список\_переменных>As<тип> <устр\_лев\_фактор\_описание>

<устр\_лев\_фактор\_описание> ::= E | = <операнд><знак><операнд> | = <операнд>

Устранение левой факторизации для <устр\_лев\_фактор\_описание>:

<устр\_лев\_фактор\_описание>::= =<операнд> <устр\_лев\_фактор\_присваивание> | E

Устранение левой рекурсии <список\_переменных>

<список\_переменных>::= id | id<устр\_рек\_список\_переменных>

<устр\_рек\_список\_переменных>::= ,id | ,id <устр\_рек\_список\_переменных>

Устранение левой факторизации:

<список\_переменных>::= id<устр\_фактор\_список\_переменных>

<устр\_фактор\_список\_переменных>::= E|<устр\_рек\_список\_переменных>

<устр\_рек\_список\_переменных>::= ,id<устр\_фактор\_список\_переменных>

Устранение левой факторизации <условный\_оператор>:

<условный\_оператор>::= If expr Then/- <список\_действий>/-<устр\_фактор\_условный\_оператор>

<устр\_фактор\_условный\_оператор>::= End If | Else/- <список\_действий>/- End If

После изменения правил под стандарты LL(k)-грамматики необходимо построить решающую таблицу нисходящего синтаксического анализатора (Таблица 1).

Таблица 1 – Решающая таблица нисходящего анализатора

|  |  |  |
| --- | --- | --- |
| Правила грамматики | FIRST1(a) & Follow1(a**)** | FIRST2(a) &  Follow1(a) |
| <программа>::=<спис\_опис><спис\_опер> | Dim |  |
| <спис\_опис>::=<опис>\n<спис\_опис>  <спис\_опис>::=<опис>\n | Dim  Dim | Dim  id,do |
| <опис>::=Dim <спис\_перем> as <тип> | Dim |  |
| <спис\_перем>::=<операнд><спис\_перем>  <спис\_перем>::=<операнд> | id  id | ,  as |

Продолжение таблицы 1

|  |  |  |
| --- | --- | --- |
| <спис\_опер>::=<опер>\n<спис\_опер>  <спис\_опер>::=<опер> | Id,do  Id,do | \n  $ |
| <опер>::=<присв>  <опер>::=<цикл> | id  Do |  |
| <присв>::=id=<матем> | id |  |
| <матем>::=<операнд><знак><матем>  <матем>::=<операнд> | id,lit  id,lit | +,-,\*,/  \n,$ |
| <логич>:=<матем><знак\_логич><логич>  <логич>:=<операнд> | id,lit  id,lit | >,<,<>,=  \n,$ |
| <цикл>::=do while<логич>\n<опер>\nloop | do |  |
| <операнд>::=id  <операнд>::=lit | id  lit |  |
| <тип>::=integer  <тип>::=double  <тип>::=string | Integer  double  string |  |
| <знак>::=+  <знак>::=-  <знак>::=\*  <знак>::=/ | +  -  \*  / |  |
| <знак\_логич>::=<  <знак\_логич>::=>  <знак\_логич>::=<>  <знак\_логич>::== | <  >  <>  = |  |

Продолжение таблицы 1

|  |  |  |
| --- | --- | --- |
| <знак> ::= +  <знак> ::= -  <знак>::= \* | +  -  \* |  |

В результате построения решающей таблицы было выявлено, что вышеописанная грамматика относится к классу LL(2), так как в одном правиле необходимо знать две лексемы, чтобы анализатор понял, что ему делать дальше.

3 Разработка архитектуры системы и алгоритмов

Для реализации транслятора с подмножества языка Visual Basic необходимо реализовать лексический анализатор, синтаксический анализатор, а после этого анализатор сложных логических выражений.

3.1 Работа лексического анализатора

Алгоритм работы лексического анализатора можно представить в виде диаграммы действий (Рисунок 1).

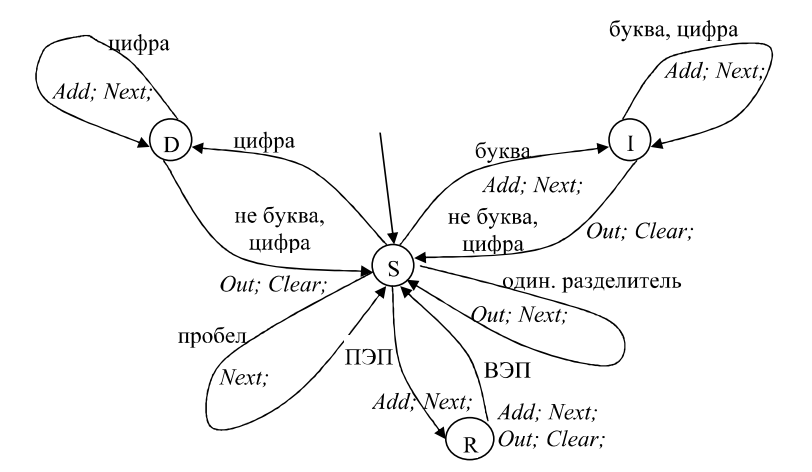


Рисунок 1 – Диаграмма состояний

Алгоритм лексического анализатора:

1. Объявляем текущим начальное состояние S диаграммы.

2. До тех пор, пока не будет достигнуто конечное состояние диаграммы на последнем элементе входного потока или получена ошибка лексического анализа, считываем очередной символ анализируемой строки и переходим из текущего состояния диаграммы в другое по дуге, помеченной этим символом, выполняя при этом соответствующие действия. Состояние, в которое попадаем, становится текущим.

3. Выходной поток формируется вызовом подпрограммы out. Out (лексема, тип) – подпрограмма (метод), которая выдаёт информацию о накопленной лексеме в выходной поток. Тип задаётся веткой диаграммы состояний по которой была собрана лексема. Данная подпрограмма также проверяет длину идентификатора и значение литерала. Если порог максимальной длины/значения превышен, то выводится сообщение с ошибкой, работа лексического анализатора прекращается.

Результатом работы лексического анализатора является перечень всех найденных в анализируемом коде лексем. Его можно представить в виде таблицы пар (лексема, её предварительный тип) или списка. Предварительными типами являются идентификатор (I), разделитель (R), литерал (D). После происходит разделение идентификаторов на ключевые слова и переменные и формирование следующих таблиц: таблица ключевых слов, таблица переменных, таблица разделителей, таблица литералов и таблица, содержащая в себе строки вида (номер таблицы определенного типа лексемы; номер лексемы в таблице, соответствующей её типу).

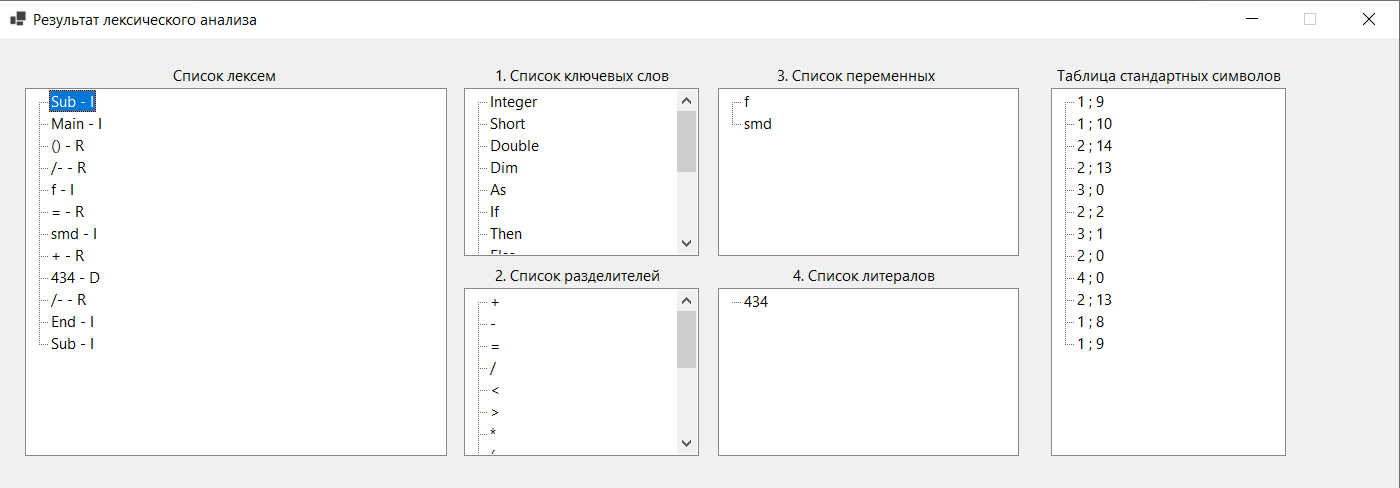


Рисунок 2 – Результат лексического анализа

3.2 Синтаксический анализатор

Для реализации синтаксического анализатора был реализован класс SyntacticAnalysis, который содержит поля и методы для реализации метода рекурсивного спуска. Каждый метод в классе реализует одно единственное правило, соответствующее решающей таблице.

Метод рекурсивного спуска – это нисходящиий метод синтаксического анализа, в котором входной поток лексем обрабатывается выполнением ряда рекурсивных процедур. Каждому нетерминалу грамматики соответствует одна подпрограмма, которая распознаёт цепочку, порождаемую этим нетерминалом. Последовательность вызовов процедур неявно определяет дерево разбора. Метод рекурсивного спуска применим в классе LL(k)-грамматик.

При разработке подпрограмм, необходимо придерживаться следующих правил:

1. Каждому нетерминалу соответствует подпрограмма. Если одному нетерминалу в левой части правила соответствует более одной правой части, то первый шаг – выбор нужного правила из списка альтернативных, для чего используется информация из второй колонки решающей таблицы, иначе первый шаг – пропустить. На этом этапе программируется анализ текущего элемента входного потока и следующих за ним (но не более k) и в случае совпадения элементов с эталонным набором из таблицы, программируется ветка, соответствующая этому эталону (см. правила 2 и 3). Смены текущего элемента при этом не происходит. Если нет совпадения ни с одним из возможных эталонных вариантов, то фиксируем ошибку и выходим из подпрограммы.

2. Если текущим символом правой части правила грамматики является нетерминал, ему соответствует вызов подпрограммы, имя которой совпадает с этим нетерминалом и которая распознаёт цепочку, порождаемую им.

3. Если текущим символом правой части правила грамматики является терминал, то проверяем текущий элемент цепочки на совпадение с этим терминалом. Если совпадения нет – фиксируем ошибку и выходим из подпрограммы, иначе перемещаемся к следующему символу входного потока (текущим становится следующий).

3.3 Транслятор логических выражений

Транслятор логических выражений (правило expr) в соответствии с техническим заданием следует реализовывать методом Бауэра-Замельзона.

Алгоритм работы транслятора следующий:

В методе используются два стека и таблица переходов. Один стек, обозначим его E, используется для хранения операндов, другой стек – Т, для хранения знаков операций. Над стеком Е выполняются две операции: К(id) – выбрать элемент с именем id из входного потока, положить на вершину стека Е, перейти к следующему элементу входного потока; К(ОР) – извлечь два верхних операнда из стека Е, записать тройку: (ОР, операнд, операнд) в матрицу арифметического оператора; записать результат на вершину стека Е. Над стеком Т выполняются операции согласно таблице переходов (Рисунок 3).

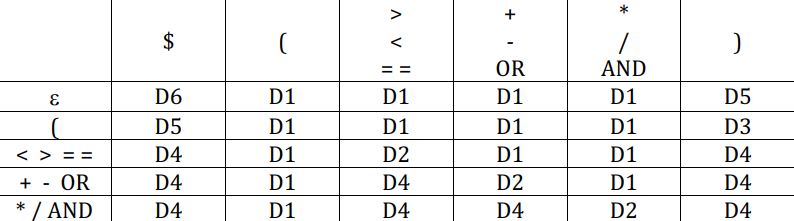


Рисунок 3 – таблица переходов

Описание правил в таблице переходов:

D1. Записать ОР в стек Т и читать следующий символ строки.

D2. Удалить ОР1 из стека Т и генерировать команду К(ОР1); Записать ОР в стек Т и читать следующий символ строки.

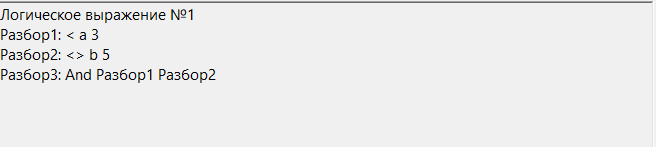
D3. Удалить ОР1 из стека Т и читать следующий символ строки.

D4. Удалить ОР1 из стека Т и генерировать команду К(ОР1);

D5. Ошибка в выражении. Конец разбора.

D6. Успешное завершение разбора.

В соответствии с описанным алгоритмом был реализован транслятор выражений, корректно разбирающий простые и сложные логические выражения. Матрица логического оператора в результате разбора выражения (a < 3) And (b <> 5) выглядит следующим образом (Рисунок 4):



4.1 Общее описание программы

Рисунок 4 – Разбор сложного логического выражения

4 Методика испытаний

Для проверки работы анализатора подмножества языка следует проверить работу лексического, синтаксического анализаторов и транслятора арифметических выражений.

4.1 Проверка лексического анализатора

Для проверки лексического анализатора необходимо ввести код на языке Visual Basic и начать анализ кода. Если были введены корректные лексемы, то высветиться окно с сообщением об успешном выполнении лексического анализа, а также станет доступна форма с данными о лексемах и её типах (Рисунок 5-6).

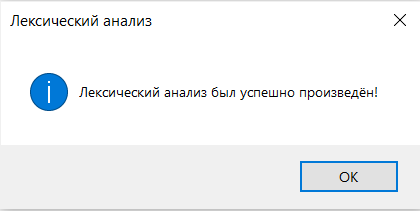


Рисунок 5 – Сообщение об успешном выполнении лексического анализа

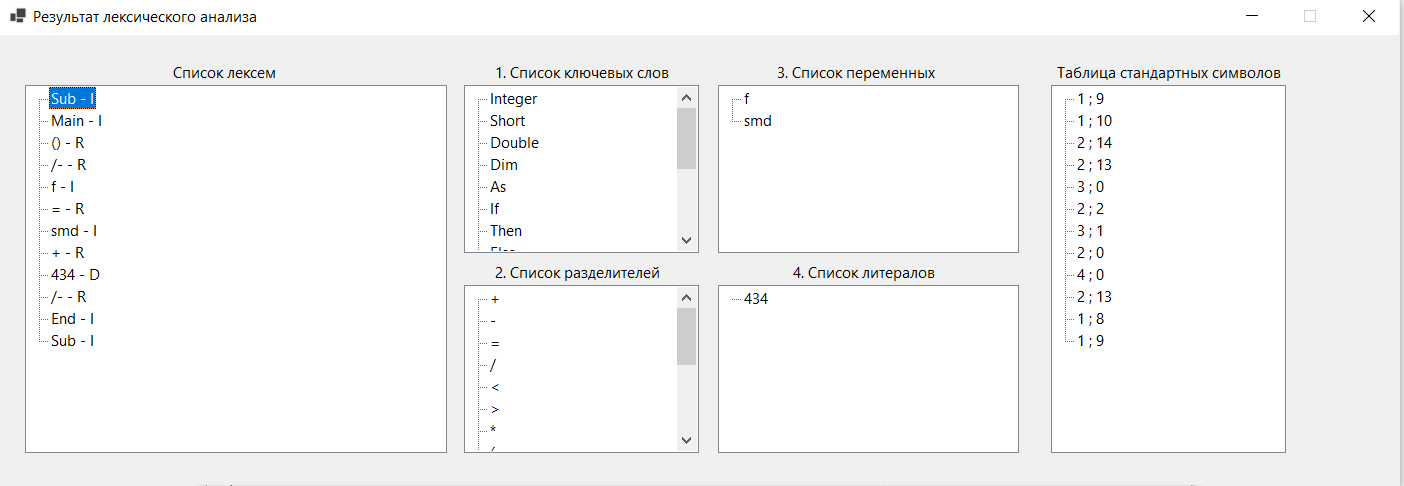


Рисунок 6 – Результат лексического анализа

Если же будет введён символ, который отсутствует в грамматике языка, то выведется сообщение с ошибкой, форма с данными о лексемах и её типах будет недоступна (Рисунок 7).

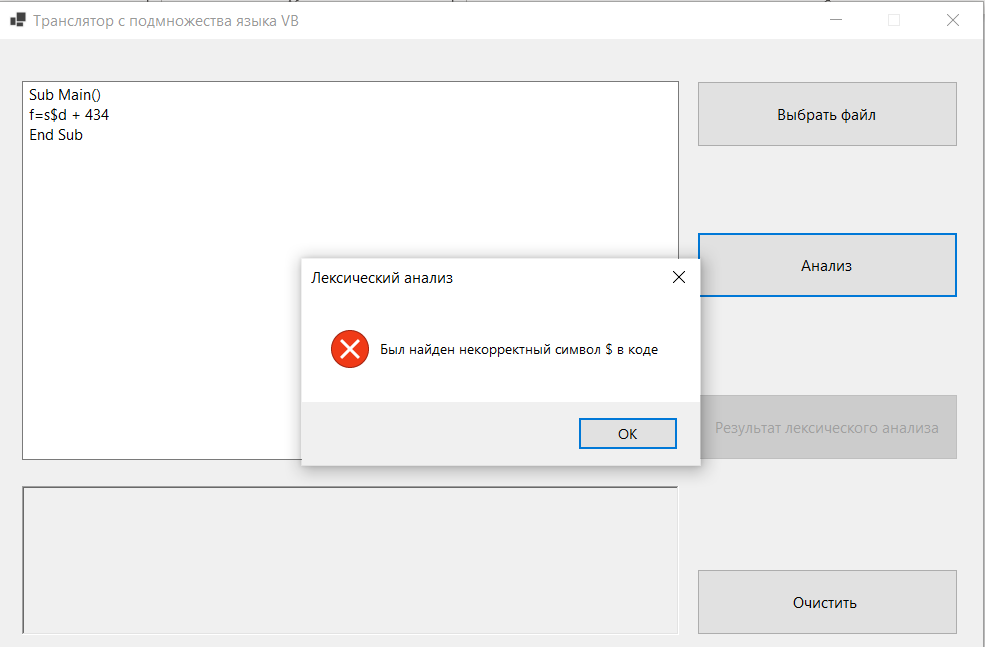


Рисунок 7 – Ошибка в лексическом анализе

4.2 Проверка синтаксического анализатора

Если лексический анализ успешно произведён, то после него начнётся синтаксический анализ. Если данный анализ успешно выполниться, то выведется соответствующее сообщение (Рисунок 8).

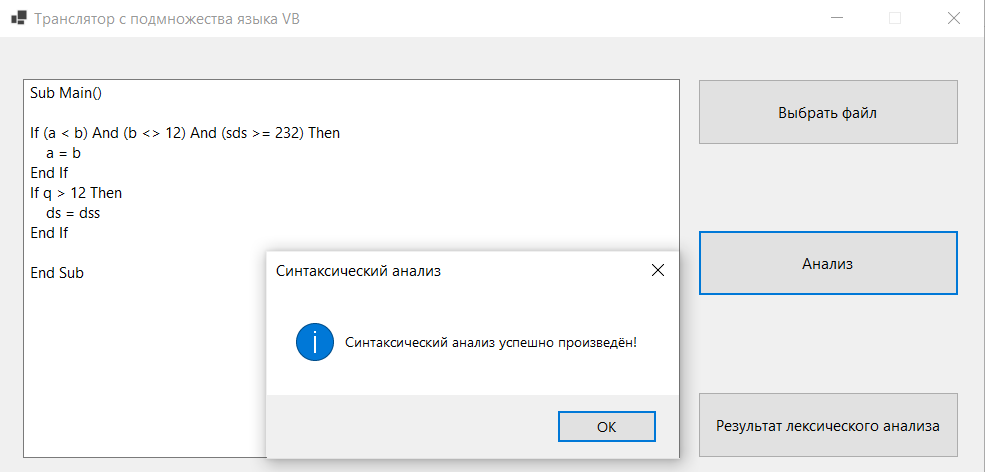


Рисунок 8 – Сообщение о успешном выполнении синтаксического анализа

Если же будет введена лексема, которая не соответствует ожидаемой лексеме в правиле, то выведется сообщение с ошибкой.

Примеры ошибок в синтаксическом анализе:

В правиле <устр\_лев\_фактор\_присваивание> может быть написано простое арифметическое действие (сложение, вычитание, умножение или деление). Если при выполнении данной операции вместо символов +,-,\*, /, /- встретиться другая лексема, то выведется сообщение с ошибкой (Рисунок 9):

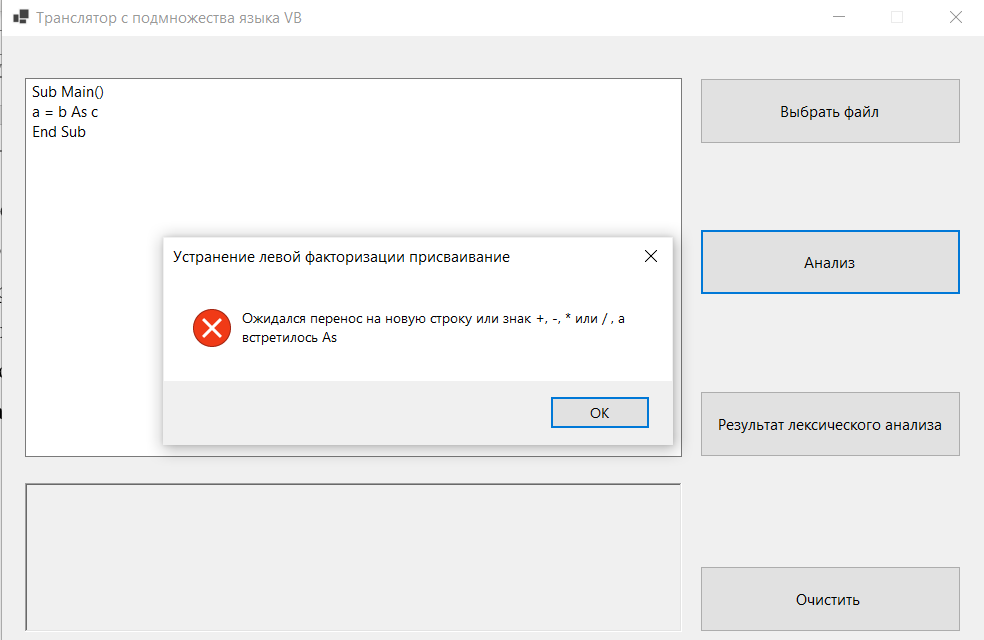


Рисунок 9 – Ошибка в правиле «устранение левой факторизации присваивание»

При описании переменных должен писаться тип данных Integer, или Double, Short(данные типы указаны в правиле «тип»).Если будет введена другая лексема, то выведется соответствующее сообщение (Рисунок 10).

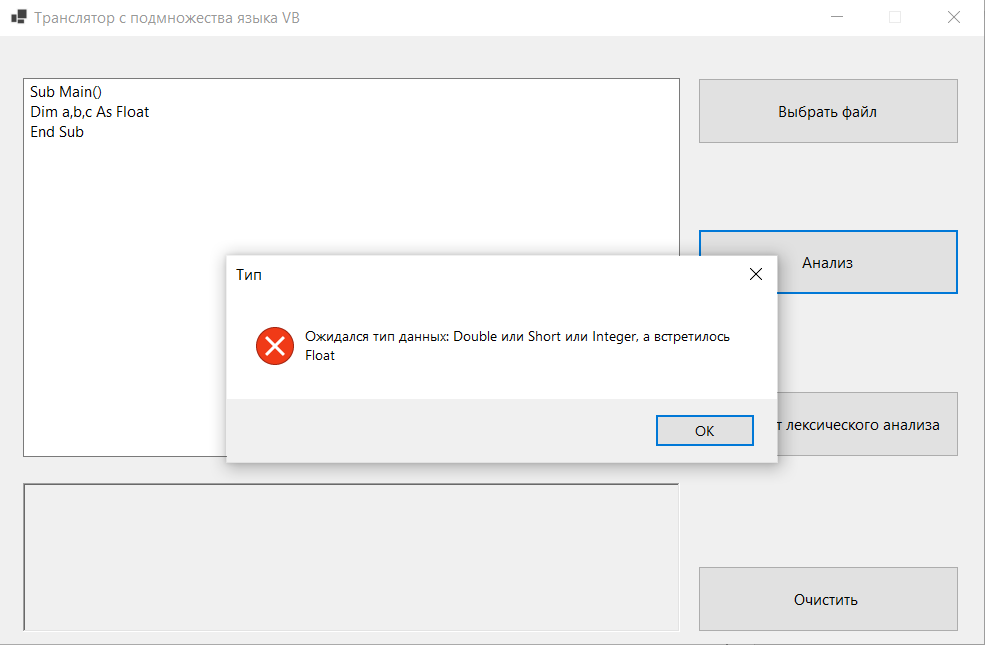


Рисунок 10 – Некорректный тип данных

Программа должна заканчиваться парой ключевых слов End Sub. Если будут введены другие лексемы, то выведется соответствующее сообщение (Рисунок 11):

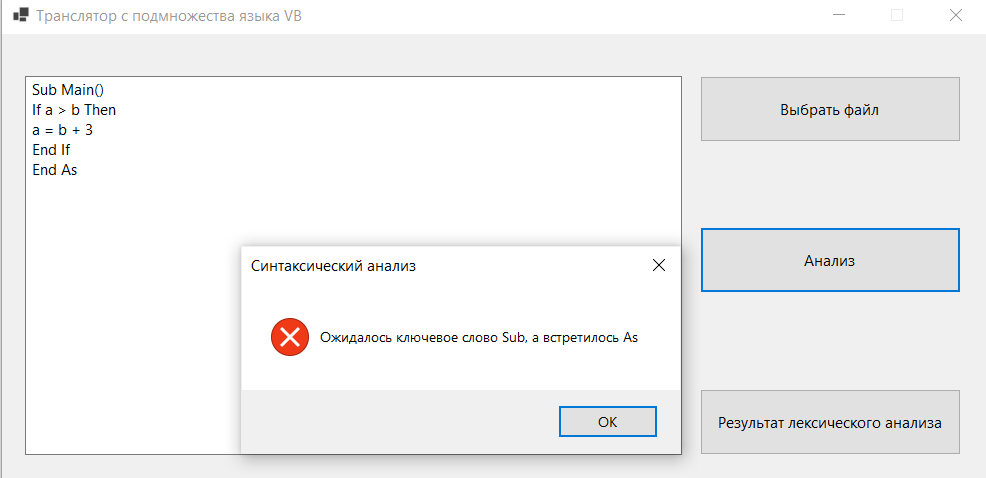


Рисунок 11 – Ошибка завершения программы

4.3 Проверка анализатора логических выражений

Для проверки корректной работы транслятора логических выражений следует передать ему сложное выражение со множеством условий и разрозненным порядком операций. Тест, состоящий из разбора сложного выражения, был успешно пройден (Рисунок 12).

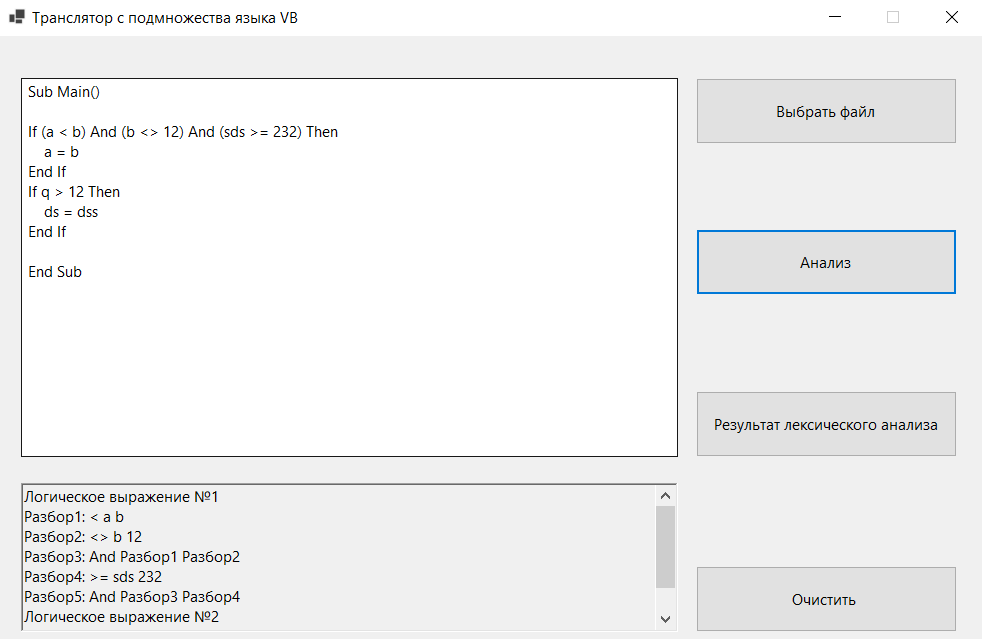


Рисунок 12 – Успешный разбор сложного логического выражения

Если будет введена лишняя скобка, то высветится соответствующее сообщение об ошибке (Рисунок 13).

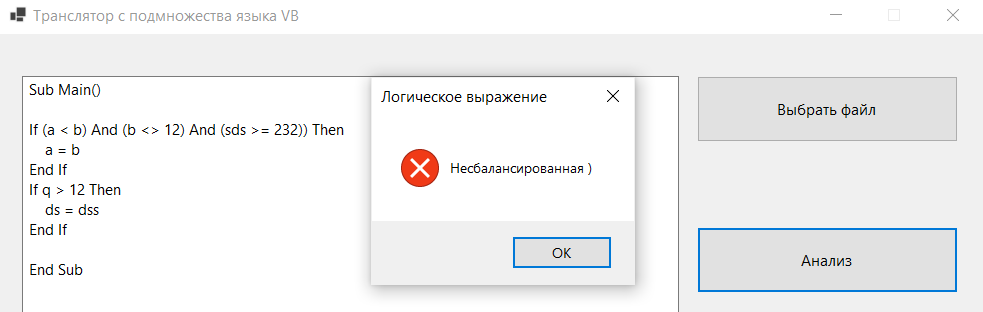


Рисунок 13 – Лишняя скобка

Если будет отсутствовать операнд в условии, то выведется соответствующее сообщение с ошибкой (Рисунок 14).

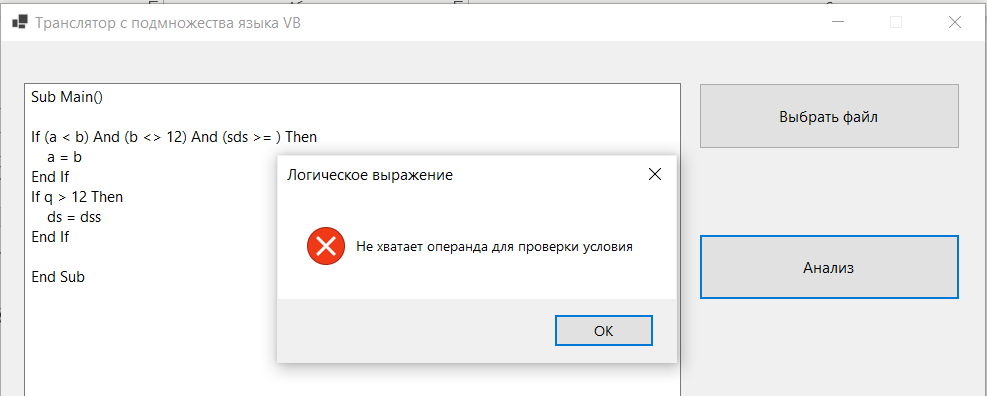


Рисунок 14 – Нехватка операнда

Если в условии встретиться некорректная лексема, например, ключевое слово, то выведется соответствующее сообщение об ошибке (Рисунок 15).

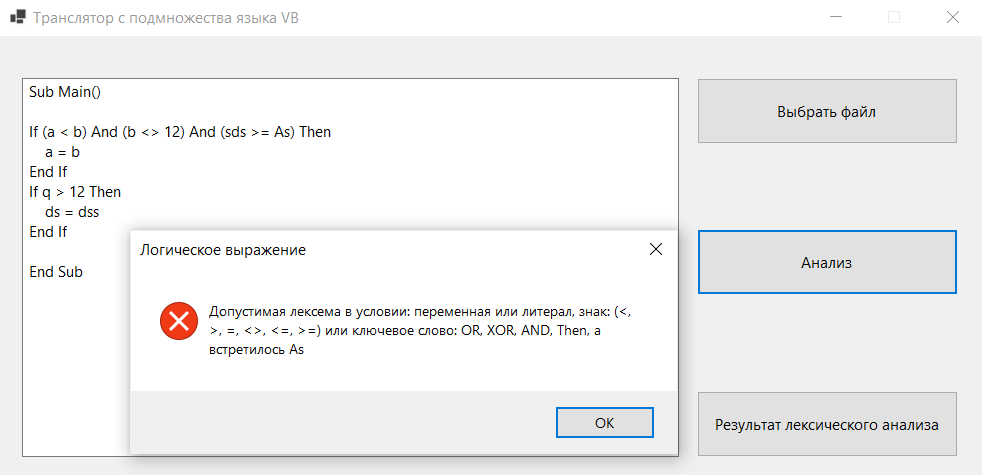


Рисунок 15 – Некорректная лексема в условии

# 5 Руководство пользователя

При запуске exe-файла открывается главное меню программы (Рисунок 16). Пользователю уже будет доступен фрагмент рабочего кода. Если пользователь захочет стереть данные из текстового поля и из поля для вывода матрицы логического оператора, то ему надо будет нажать на кнопку «Очистить».

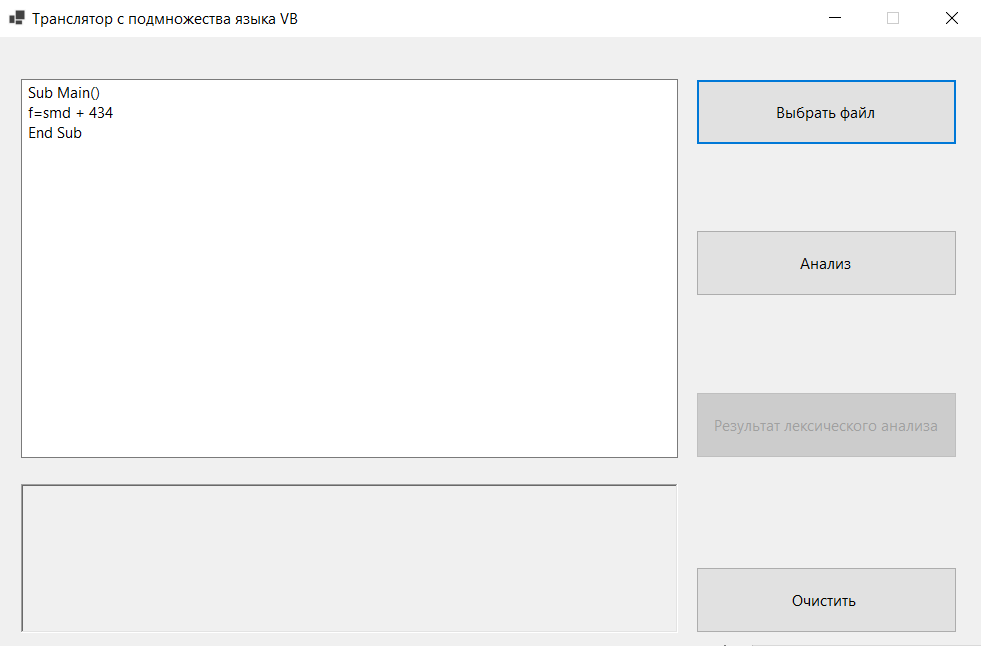


Рисунок 16 – Главное меню приложения

После ввода кода пользователь может его проанализировать, нажав на кнопку «Анализ». Если лексический анализ будет успешно произведен, то пользователь сможет просмотреть все лексемы и их типы (Рисунок 17).

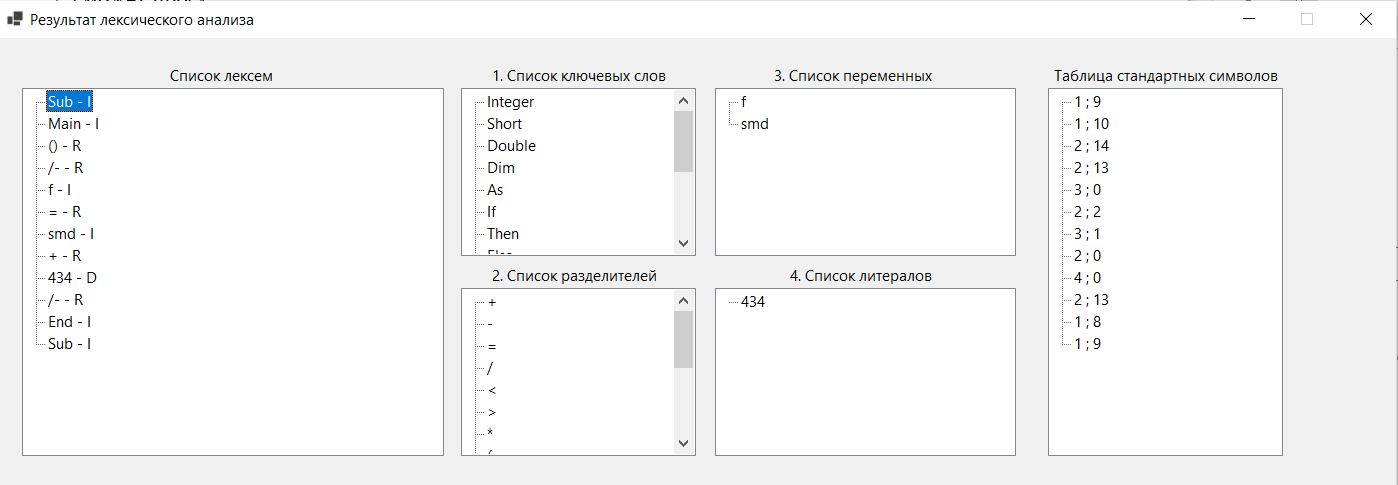


Рисунок 17 – Результат лексического анализа

# 6 Руководство программиста

Исходный код разработанного приложения содержит в себе множество классов, каждый из которых хранит в себе множество методов. Руководство включает в себя описание классов, методов и полей программы.

6.1 Класс Classification.cs

Класс Classification.cs предназначен для хранения и получения доступных лексем языка.

Поля и свойства класса:

static char singleSeparators – массив одинарных разделителей

static string[] doubleSeparators – массив двойных разделителей

public static List<Lexeme> lexemes – список лексем

public static List<string> variables – список переменных

public static List<string> literals – список литералов

public static List<string> separators – список разделителей

public static List<Token> tokens – список токенов

public static List<string> keyWords – список ключевых слов

public static string[] DoubleSeparators – свойство для массива двойных разделителей

public static char SingleSeparators – свойство для массива одинарных разделителей

public static List<string> KeyWords – свойство для списка ключевых слов

Методы класса:

private static List<string> GetListSeparators() – метод формирования списка разделителей

public static string GetLexeme(Token token) – метод получения лексемы. Принимает токен, возвращает лексему.

public static bool isId(string lexeme) – метод, проверяющий лексему. Принимает лексему, возвращает true, если лексема – идентификатор.

public static bool isLiteral(string lexeme) - метод, проверяющий лексему. Принимает лексему, возвращает true, если лексема – литерал.

public static void Clear() – метод очистки списков.

6.2 Класс Correctness.cs

Класс Correctness.cs нужен для проверки лексемы на правильность.

Методы класса:

public static bool IsErrorInLengthOfString(string str) – метод, проверяющий превышена ли длина строки заданному значению. Принимает строку, возвращает true, если у строки превышена длина.

public static bool IsErrorInValueOfNumber(string str) – метод, проверяющий превышено ли значение числа максимальному значению int. Принимает число в виде строки, возвращает true, если значение числа превышает максимальное значение int.

public static bool CharIsEnglishChar(char symbol) – метод, проверяющий символ на наличие его в английском алфавите. Принимает символ, возвращает true, если символ есть в английском алфавите.

6.3 Класс Lexeme.cs

Класс Lexeme.cs нужен для хранения данных о лексеме.

Поле и свойства класса:

private string name – название лексемы

private char type – тип лексемы

public string Name – свойство для названия лексемы

public char Type – свойство для типа лексемы

public Lexeme(string name, char type) – конструктор класса. Заполняет поля класса соответствующими данными.

6.4 Класс LexicalAnalysis.cs

Класс LexicalAnalysis.cs нужен для выполнения лексического анализа.

Поля класса:

string buffer – буфферная строка

Status status – статус состояния

bool errorInLength – ошибка превышения длины лексемы

Также в классе присутствует перечисление Status – статусы для лексического анализа.

Методы класса:

public bool Analysis(string text, Button ResultButton) – метод выполнения лексического анализа. Принимает код программы, кнопку для вывода данных о лексемах и её типах, возвращает true, если лексический анализ успешно произведен.

private bool SuccessfulLexemeAddition(string lexeme, char type) – метод добавления лексемы в список лексем. Принимает лексему и её тип, возвращает true, если лексема успешно добавлена.

private void ClearOfBuffer() – метод для очистки строки буффера и аннулирования статуса состояния.

6.5 Класс SyntacticAnalysis.cs

Класс SyntacticAnalysis.cs предназначен для выполнения синтаксического анализа и для разбора логического выражения.

Поля класса:

int success – успешное выполнение синтаксического анализа

int numLexeme – номер лексемы

int numLogicOperation – номер логической операции

string actualLexeme – текущая лексема

List<Token> tokens – список токенов

Stack<string> E – стек с операндами

Stack<string> T – стек с операциями

List<string> matrix – матрица логических выражений

RichTextBox textbox – поле для вывода матрицы логических выражений

Методы класса:

public SyntacticAnalysis(List<Token> tokens, RichTextBox richTextBox) – конструктор для заполнения полей класса соответствующими данными.

private void Next() – метод, который берет следующую лексему.

private void Past() – метод, который берет прошлую лексему.

public void StartAnalysis() - метод, реализующий стартовое правило: <Программа>

private bool ListOfActions() – метод, реализующий правило <список\_действий>. Возвращает true, если не встретились некорректные лексемы.

private bool Action() - метод, реализующий правило <действие>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationOfLeftFactorListOfAction() - метод, реализующий правило <устр\_лев\_фактор\_спис\_действий>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationOfLeftRecursionListOfAction() - метод, реализующий правило <устранение\_рек\_действия>. Возвращает true, если не встретились некорректные лексемы.

private bool Assignment() - метод, реализующий правило <присваивание>. Возвращает true, если не встретились некорректные лексемы.

private bool Description() - метод, реализующий правило <описание>. Возвращает true, если не встретились некорректные лексемы.

private bool ConditionalOperator() - метод, реализующий правило <условный\_оператор>. Возвращает true, если не встретились некорректные лексемы.

private void Expr() – метод разбора логического выражения.

private void D1() – метод, реализующий действие D1

private void D2() – метод, реализующий действие D2. Возвращает true, если действие успешно выполнено.

private void D3() – метод, реализующий действие D3.

private void D4() – метод, реализующий действие D4. Возвращает true, если действие успешно выполнено.

private bool CheckOperands() – метод проверки стека с операндами на наличие лишнего операнда. Возвращает true, если в стеке с операндами присутствует лишний операнд.

private bool EliminationLeftFactorConditionalOperator() - метод, реализующий правило <устр\_фактор\_условный\_оператор>. Возвращает true, если не встретились некорректные лексемы.

private bool ListOfVariables() - метод, реализующий правило < список\_переменных>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationLeftFactorListOfVariables() - метод, реализующий правило <устр\_фактор\_список\_переменных>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationLeftRecursionListOfVariables() - метод, реализующий правило <устр\_рек\_список\_переменных>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationLeftFactorDescription() - метод, реализующий правило <устр\_лев\_фактор\_описание>. Возвращает true, если не встретились некорректные лексемы.

private bool EliminationLeftFactorAssignment() - метод, реализующий правило <устр\_лев\_фактор\_присваивание>. Возвращает true, если не встретились некорректные лексемы.

private bool Sign() - метод, реализующий правило <знак>. Возвращает true, если не встретились некорректные лексемы.

private bool Operand() - метод, реализующий правило <операнд>. Возвращает true, если не встретились некорректные лексемы.

private bool Type() - метод, реализующий правило <тип>. Возвращает true, если не встретились некорректные лексемы.

private void GetMessageErrorKeyWords(string keyWord) - Метод вывода сообщения о том, что ожидалось ключевое слово. Принимает ключевое слово.

private void GetMessageErrorSymbol(string symbol) - Метод вывода сообщения о том, что ожидался символ. Принимает символ.

private void GetMessageErrorTransferNewLine() - Метод вывода сообщения о том, что ожидался переход на новую строку.

private void GetMessageErrorAction() - Метод вывода сообщения об ошибке в правиле "Действие".

private void GetMessageErrorId() - Метод вывода сообщения о том, что ожидалась переменная.

private void GetMessageErrorIdOrLiteral() - Метод вывода сообщения о том, что ожидалась переменная или литерал.

private void GetMessageErrorEliminationLeftFactorAssignment() - Метод вывода сообщения о ошибке в правиле "Устранение левой факторизации присваивание".

private void GetMessageErrorSign() - Метод вывода сообщения о ошибке в правиле "Знак".

private void GetMessageErrorEliminationLeftFactorListOfVariables() - Метод вывода сообщения об ошибке в правиле "Устранение левой факторизации список переменных".

private void GetMessageErrorType() - Метод вывода сообщения об ошибке в правиле "Тип".

private void GetMessageEliminationLeftFactorDescription() - Метод вывода сообщения об ошибке в правиле "Устранение левой факторизации описание".

private void GetMessageErrorEnd() - Метод вывода сообщения о невозможности продолжения выполнения кода.

private void GetMessageEliminationLeftFactorListOfAction() - Метод вывода сообщения об ошибке в правиле "Устранение левой факторизации список действий".

private void GetMessageErrorEliminationLeftFactorConditionalOperator() - Метод вывода сообщение об ошибке в правиле "Устранение левой факторизации условный оператор".

private void GetMessageErrorWaitingExpr() - Метод вывода сообщения о том, что ожидалось условие.

private void GetMessageErrorLogicalExpression() - Метод вывода сообщения о том, что введен некорректный символ для разбора сложного логического выражения.

private void GetMessageErrorWaitingOperationAndOperand() - Метод вывода сообщения о том, что ожидались оператор и операнд.

6.6 Класс Token.cs

Класс Token.cs предназначен для хранения токенов и формирования списка токенов.

Поля и свойства класса:

int numberTable – номер таблицы

int indexLexeme – индекс лексемы в таблице

public int NumberTable – свойство для номера таблицы

public int indexLexeme – свойство для индекса лексемы в таблице

Методы класса:

public Token(int numberTable, int indexLexeme) – конструктор класса. Заполняет поля класса соответствующими данными.

public static List<Token> GeneratingListTokens(List<Lexeme> lexemes, List<string> keyWords, List<string> separators, List<string> variables, List<string> literals) – метод создания списка токенов. Принимает список лексем, ключевых слов, разделителей, переменных литералов, возвращает список токенов.

# Заключение

В процессе выполнения курсового проекта был создан транслятор, способный переводить подмножество языка Visual Basic согласно требованиям, изложенным в ТЗ. Для создания транслятора использовались среда разработки Visual Studio и объектно-ориентированный язык программирования C#.

Для реализации транслятора был произведен лексический анализ, в результате которого были получены списки лексем и их типов. Затем была построена грамматика языка, приведенная к классу LL(k), и реализован синтаксический анализ. Для разбора сложных логических выражений был использован метод Бауэра-Замельзона.

Если потребуется расширить функционал программы, можно добавить проверку на соблюдение семантических соглашений языка, что позволит оценить состояния, которые не могут быть проверены на этапе синтаксического анализа. Программа занимает небольшой объем памяти и не требует высоких системных требований для работы.

# Список литературы

1. Шульга, Т. Э. Теория автоматов и формальных языков: учебное пособие / Т. Э. Шульга. — Саратов: Саратовский государственный технический университет имени Ю.А. Гагарина, ЭБС АСВ, 2015. — 104 c.

2. Алымова, Е. В. Конечные автоматы и формальные языки : учебник / Е. В.Алымова, В. М. Деундяк, А. М. Пеленицын. — Ростов-на-Дону, Таганрог: Издательство Южного федерального университета, 2018. — 292 c.

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

# Приложение А. Ссылка на репозиторий

Программный код проекта расположен в следующем репозитории:

https://github.com/Badom/translator