hМинистерство 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»

Факультет ИТР

Кафедра ПИн

*Курсовая*

*РАБОТА*

По Теория автоматов и формальных языков

Тема Транслятор с подмножеством языка VB

Руководитель

Кульков Я.Ю

(фамилия, инициалы)

(подпись) (дата)

Студент ПИН - 121

(группа)

Носков М.Ю

(фамилия, инициалы)

(подпись) (дата)

Муром 2023

В данной курсовой работе необходимо было спроектировать транслятор подмножества языка Visual Basic. В качестве средств разработки приложения была использована среда Microsoft Visual Studio 2022. Язык разработки: С#.

In this course work, it was necessary to design a translator for a subset of the Visual Basic language. The Microsoft Visual Studio 2022 environment was used as the application development tools. Development language: C#.

Содержание

Введение……………………………………………………………….…5

1. Анализ технического задания……………………………….………6
2. Описание грамматики языка………………………………………...8
3. Разработка архитектуры системы и алгоритмов…………….…….15
4. Тестирование……………...………………………………….….......19
5. Руководство пользователя…………………….……………….........25
6. Руководство программиста..…………………………………...........26
7. Заключение…………………………………….……………………..32
8. Список используемой литературы………………………...………..33
9. Приложение1…………………………………………………………35
10. Приложение2…………………………………………………………37

Введение

В настоящее время искусственные языки широко применяются в программировании, а так же в других областях. Для того, что бы корректно составлять структуры всевозможных документов, графических интерфейсов и т.д, используются трансляторы – особые системы, служащие для анализа и интерпретации текстов.

Несмотря на то , что в наше время разработан огромное количество различных языков и их трансляторов, процесс создания новых приложений не останавливается. Это связано с тем, что технологии производства вычислительных систем крайне быстро развиваются, и трансляторы быстро устаревают.

Исходными данными для транслятора служит текст входной программы, т.е некая последовательность символов входного языка программирования, удовлетворяющая синтаксическим требованиям.

Идея транслятора легла в основу создания многих языков программирования, например Visual Basic - процедурного языка с элементами компонентной и структурной парадигмой программирования.

1. Анализ технического задания.

В представленной курсовой работе необходимо спроектировать транслятор подмножества языка Visual Basic.

Анализируя тему данной курсовой работы, требуется, чтобы в созданной программе присутствовали:

1) развернутая диагностика ошибок.

2) реализация класса транслятора.

3) синтаксический разбор – на основе LL(k)-грамматик.

4) разбор выражений, выполненный методом Бауэра-Замельзона.

В языке поддерживаются:

1) идентификаторы, значащие первые 8 символов.

2) не менее трех директив описания переменных.

3) сложный арифметический оператор.

4) оператор выбора select … case

Представленная курсовая работа реализуется в несколько шагов:

Создание лексического анализа, который в свою очередь выполняет

анализ полученных данных из файла , а также распознает лексемы и их типы. Полученная информация обрабатывается на основе синтаксического анализа.

Синтаксический анализ обрабатывает данные, полученные в ходе работы лексического анализа, посредством нахождения синтаксических выражений и конструкций.

Разбор сложного логического выражения – последовательный перевод логического выражения в матричную форму записи.

В созданном приложении Windows Forms должно быть реализовано:

1. Возможность загрузки кода, как с файла, так и написание его в ручную с клавиатуры.
2. Вывод на экран результатов работы лексического анализа, классификатора лексем, нисходящего анализатора, а так же метода разбора сложных арифметических выражений

2. Описание грамматики языка

Для краткости название языка сократим до VB. Базовые символы языка VB - специализированные символы, цифры, а также буквы. Эти наборы символов представляют алфавит языка VB.

Как и любой другой язык, язык VB имеет свой алфавит, в который включены 26 букв латинского алфавита, цифры от 0 до 9, арифметические операции (+ , - , \*, /, \, ^ ), знаки отношений (<, >, =), разделительные знаки [: , ; ‘ “ . ( ) ] , другие знаки (# , $ , @ , ! , ? , & , \_ , %.).

Из алфавита данного языка можно сложить разнообразные конструкции языка, такие как: данные; операторы, выражения и функции.

Имена переменных в VB подчиняются ряду ограничений :

1. Первым символом имени должна быть *буква.*

2. Остальные символы - *буквы* и *цифры.* (Прописные и строчные буквы различаются.)

3. Можно использовать знак \_. Нельзя использовать точку.

4. Число символов не должно превышать 255.

5. Имя не должно быть *ключевым словом* VB.

В языке VB версии 6.0 переменная может иметь одиниз более чем десяти типов. Учитывая все вышеперечисленные правила, в ходе работы создана грамматика языка.

Конструкция Select Case позволяет обрабатывать сразу несколько условий. После слов Select Case указывается сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора Case. И в случае если значения совпали, то выполняется блок команд, помещенных после данного оператора Case. Конструкция завершается словами End Select. Если мы хотим определить действия, которые будут выполняться, если совпадений не выявлено, то мы можем использовать оператор Case Else, после которого помещаем блок действий по умолчанию. Блок Case Else необязателен и может не употребляться.

В данной грамматике языка существует множество конструкций, например:

Оператор <спис\_опер> нужен для того, чтобы размножить элемент списка. Оператор <опис> служит для перечисления переменных, а также для инициализации их типа.

Имея грамматику входного языка, необходимо выполнить ряд некоторых преобразований над этой грамматикой, облегчающих построение распознавателя. После этого надо проверить относится ли полученная грамматика к одному из известных классов КС-языков, для которых существуют линейные распознаватели. Целью работы считается построение грамматики, на базе которой можно реализовать нисходящий синтаксический анализатор.

Если грамматика является LL(k)-грамматикой, то просмотр вперед от текущего элемента не более k символов входной строки позволяет однозначно определить направление перехода и избежать возвратов. Любая LL(k)- грамматика однозначна.

Основная идея левой факторизации в том, что, когда неясно, какую из двух альтернатив надо использовать при выводе цепочки из нетерминала A, нужно переделать A-правила так, чтобы отложить решение до тех пор, пока не будет достаточно информации, для принятия правильного решения.

G=( N, T, P, <программа>)

Т = { Dim, as, select, case, else, end, +, -, //, \*, =>, <, ==, >=, <=, != ,=, (, ),

id,lit}

N={<программа>,<спис\_опис>,<спис\_опер>,<опис>,<тип>, <спис\_перем>,<опер>, <тип>,<условн.>, <блок.опер>,<присв.> ,<вариант> }

P ={ <программа>::=Dim <спис\_опис> \n <спис\_опер>\n

<спис\_опис> ::=<опис> | <спис\_опис> <опис>

<опис>::=<спис\_перем> as <тип>

<спис\_перем> ::= id | <спис\_перем>, id

<тип>::= integer | long | double

<спис\_опер>::= <опер>\n | <спис\_опер> \n<опер>\n

<опер>::=<условн.> | <присв.>

<условн.>::= select case id \n <вариант> <спис.вариантов> end select

<спис.вариантов>::= <вариант> | <вариант> <спис.вариантов> | Ɛ

<вариант>::= case lit \n < спис\_опер.> | case lit to lit \n < спис\_опер.> | case else \n < спис\_опер.>

<присв.>::=id= expr}

Вследствие создания цепочки вывода, было создано синтаксическое представление исходного кода. Итог работы приведен на таблице 1.

Таблица 1 - Пример формирования цепочки вывода

|  |  |
| --- | --- |
| Анализируемый фрагмент программы | Полученный синтаксический вывод |
| Dim a as integer  a=8  select case x  case 0  a=a+6\*(5+a)  case 2 to 6  a=9  case else  a=0  end select | Dim id as integer  id=lit  select case id  case lit  id=expr  case lit to lit  id=lit  case else  id=lit  end select |

Имея данную грамматику языка, произведём поиск леворекурсивных правил и правил с левой факторизацией.

Избавимся от левой рекурсии и левой факторизации:

1) <спис\_опис> ::=<опис> | <спис\_опис> <опис>

<спис\_опис>::= <опис> <доп.опер>

<доп.опер>::= Ɛ | <спис\_опис>

2) <спис\_перем> ::= id |<спис\_перем>, id

<спис\_перем> ::=id <U>

<U>::= Ɛ | , id <спис\_перем>

3) <спис\_опер>::= <опер>\n | <спис\_опер> \n <опер>\n

<спис\_опер>::=<опер>\n <Z>

<Z>::= Ɛ | <спис\_опер>

4) <спис.вариантов>::= <вариант> | <вариант> <спис.вариантов> | Ɛ

<спис.вариантов>::= <вариант> <X>

<X>::= Ɛ | <спис.вариантов>

5) <вариант>::= case lit \n <блок опер.> | case lit to lit \n <блок опер.> |

case else \n <блок опер.>

<вариант>::= case lit <W> | case else \n <блок опер.>

<W>::= \n <блок опер> | to lit \n <блок опер.>

<вариант>::=case <S>

<S>::= lit <W> | else \n <блок опер>

Полученная грамматика:

G=( N, T, P, <программа>)

Т = { Dim, as, select, case, else, end,+, -, //, \*, =>, <, ==, >=, <=, != ,=, (, ), id,lit,\n}

N={<программа>,<спис\_опис>,<опис>,<спис\_перем>,<U>,<тип>,<опер>,<спис\_опер>,<Z>,<условн.> , <вариант> , <S>, <W>,<блок опер>,<спис.вариантов> , <X> ,<присв.> ,<знак> }

P={<программа>::=Dim <спис\_опис> \n <спис\_опер>\n

<спис\_опис>::= <опис> <доп.опер>

<доп.опер>::= Ɛ | <спис\_опис>

<опис>::=<спис\_перем> as <тип>

<спис\_перем> ::=id <U>

<U>::= Ɛ | , id <спис\_перем>

<тип>::= integer | long | double

<спис\_опер>::=<опер>\n <Z>

<Z>::= Ɛ | <спис\_опер>

<опер>::=<условн.> | <присв.>

<условн.>::= select case id \n <вариант> <спис.вариантов> end select

<спис.вариантов>::= <вариант> <X>

<X>::= Ɛ | <спис.вариантов>

<вариант>::=case <S>

<S>::= lit <W> | else \n < спис\_опер>

<W>::= \n < спис\_опер > | to lit \n < спис\_опер>

<присв.>::=id= expr \n

Докажем, что полученная после преобразования грамматики является LL(k) и определим k: решающая таблица нисходящего анализатора представлена в приложении 2.

По данным полученным в результате построения таблицы множеств

FIRST и FOLLOW было доказано, что данная грамматика принадлежит к множеству LL (k). Так как выбрать нужное правило из двух альтернативных правил можно на основе анализа одного элемента входной строки грамматика является LL(1). На базе этой грамматики может быть построен нисходящий линейный анализатор.

3. Разработка архитектуры системы и алгоритмов

Во время работы с курсовым проектом был создан лексический анализатор. Лексический анализатор является частью компилятора, которая считывает литералы программы и строит из них лексемы. Лексический анализ обрабатывает исходный текст, полученный от пользователя, и распознает лексемы, а также классифицирует их.

Лексический анализатор выделяет из текста лексемы различных типов: идентификаторы, литералы (числовые и символьные константы), разделители. Выделение (сборка) лексемы сопровождается проверкои1 её правильности. Обнаруженные лексические ошибки фиксируются. Язык описания лексических единиц в большинстве случаев является регулярным, то есть может быть описан с помощью регулярных грамматик. Распознавателями регулярных языков являются конечные автоматы. Одним из способов описания конечного автомата является графическое его представление в виде маркированного однонаправленного графа, в котором узлы соответствуют состояниям конечного автомата, дуги отображают переходы из одного состояния в другое, а символы маркировки дуг соответствуют функции перехода конечного автомата.

Работа сканера заключается в моделировании различных конечных автоматов для распознавания идентификаторов, зарезервированных слов, констант и разделителей.

В процессе получения на вход символа, цикл производит проверку. Если символ не является ни буквой, и не цифрой, следовательно, цикл присваивает ему значение «Идентификатор». В случае, если на вход получена цифра, анализатор классифицирует ее как «Литерал». Если на вход получен символ «\n», программа инициализирует ее как «Конец строки». В противном случае присваивается значение «Разделитель».

Результатом работы сканера является последовательность кодов лексем. Эту последовательность обычно называют таблицей стандартных символов, так как в ней хранятся стандартизованные представления лексем. Информация в этой таблице расположена в том же порядке, что и в исходной программе.

Пример работы сканера на представлен на таблице 3 ниже:

Таблица 3-Пример работы сканера

|  |  |
| --- | --- |
| Dim | Идентификатор |
| \n | Конец строки |
| = | Разделитель |
| 8 | Литерал |
| select | Идентификатор |
| \* | Разделитель |

Таблица 4 -Пример работы лексического анализатора

|  |  |
| --- | --- |
| DIM | Dim |
| ENTER | \n |
| NUMBER | 8 |
| SELECT | select |
| \* | MULTIPLICATION |

Синтаксический анализ является частью компилятора, которая взаимодействует с синтаксическими конструкциями языка, с помощью токенов. Токен – некая структура данных, которая состоит из имени и набора необязательных произвольных атрибутов. Имя токена представляет собой абстрактный символ, который в свою очередь представляет тип лексической единицы, например, <ключевое слово>,<название переменной>, и т.п. Используя токены лексический анализ производит токенизацию, т.е процесс классификации разделов строки входных символов. Большая часть методов анализа введенных данных принадлежат либо нисходящим алгоритмам, либо восходящим.

Происхождение этих терминов связано с тем, каким образом строятся узлы синтаксического дерева: либо от корня (аксиомы грамматики) к листьям (терминальным символам), либо от листьев к корню. Нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. Нисходящий анализ связан с LL – грамматикой, которая обладает следующими особенностями:

Она может быть проанализирована без возвратов.

Первая буква L означает, что входная цепочка проверяется слева направо.

Вторая буква L означает, что строится левый вывод цепочки.

Алгоритм работы сканера:

На вход программы поступает список лексем, полученных при работе сканера. Согласно полученных грамматик языка происходит проверка полученного списка лексем на соответствие с грамматикой. При полном соответствии полученных лексем со списком грамматик выводится сообщение, что ошибок не обнаружено. При несоответствии полученных лексем со списком грамматик выводится сообщение об ошибке, в котором указывается ожидаемый и полученный символы.

1. Тестирование:

Целью проведения тестирования является подтверждение реализации требуемой функциональной системы. Случаем, когда тестирование прошло успешно является совпадение с ожидаемым результатом.

При прохождении всех элементов лексическим анализатором и при отсутствии ошибок, на экран выводится текстовое сообщение об успешном выполнении анализа.

При обнаружении ошибки анализатором, анализ завершает свою работу, и выводит сообщение на экран, содержащее ошибочный элемент, и элемент, ожидаемый в процессе анализа.

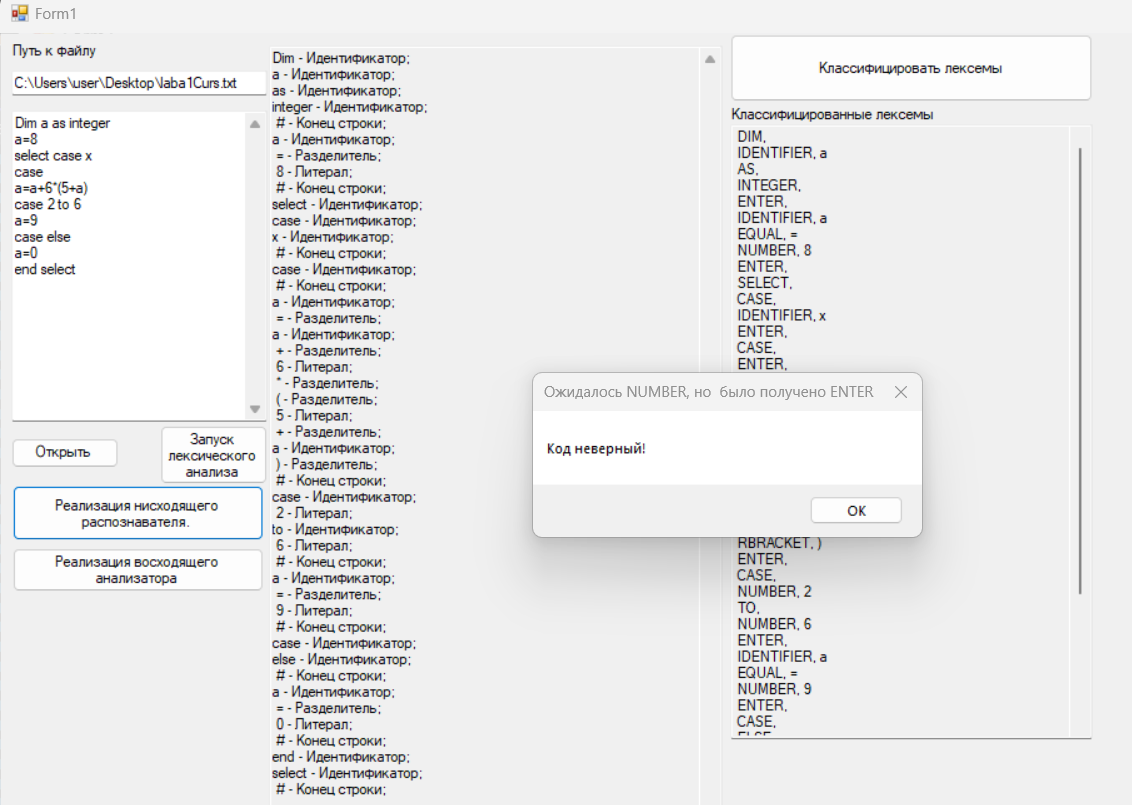


Рисунок 1 – Обнаружение ошибки лексическим анализатором

Данная ошибка определяется правилом <доп\_опер>:

public void AddDescriptions()// доп\_опер

{

switch (token.Type)

{

case Token.TokenType.ENTER:

Next();

break;

case Token.TokenType.IDENTIFIER:

ListOfDescriptions();

break;

default:

Expression(Token.TokenType.ENTER, token.Type);

break;

} }

Примеры вывода ошибок:

1. Ошибка в правиле <присв>:

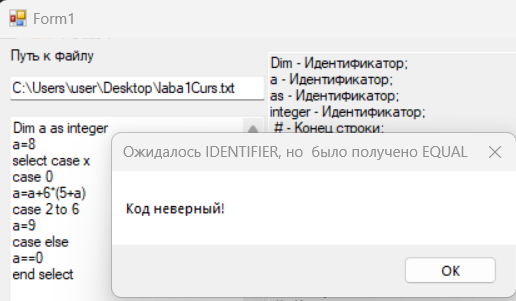


Рисунок 2 – Обнаружение ошибки в правиле <присв>

Данная ошибка возникает в следующем правиле грамматики:

<присв.>::=id= expr \n

Согласно грамматике, после «=» может идти только «expr».

Данная ошибка определяется в следующем методе:

public void Assignment()// присв

{

if (token.Type != Token.TokenType.IDENTIFIER )

{

Expression(Token.TokenType.IDENTIFIER, token.Type);

}

Next();

if (token.Type != Token.TokenType.EQUAL)

{

Expression(Token.TokenType.EQUAL, token.Type);

}

Next();

if(token.Type == Token.TokenType.IDENTIFIER || token.Type == Token.TokenType.NUMBER)

{

Skip();

Next();

}

else

{

Expression(Token.TokenType.IDENTIFIER, token.Type);

}

}

1. Ошибка в теле оператора <опис>:

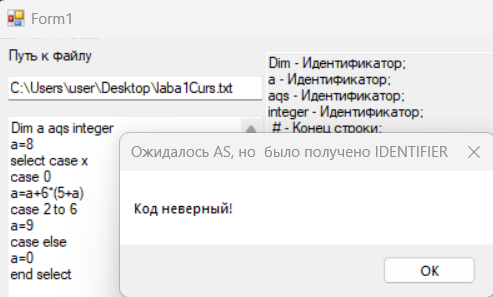


Рисунок 3 – Обнаружение ошибки в операторе <опис>

Данная ошибка возникает в следующем правиле грамматики:

<опис>::=<спис\_перем> as <тип>

После оператора <спис\_перем> должен быть элемент «as».

Данная ошибка генерируется следующим методом:

public void U()// U

{

switch (token.Type)

{

case Token.TokenType.AS:

Next();

break;

case Token.TokenType.COMMA:

Next();

ListOfVariables();

break;

default:

Expression(Token.TokenType.AS, token.Type);

break;

}

}

1. Ошибка в теле оператора <условн.>:

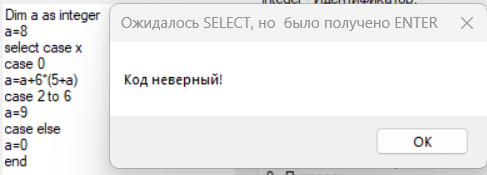


Рисунок 4 – Ошибка в теле оператора <условн.>

Данная ошибка возникает в нижеприведенном правиле грамматики:

<условн.>::= select case id \n <вариант> <спис.вариантов> end select

После элемента «end» должен следовать элемент «select». В данном примере программа отработала корректно. Данная ошибка генерируется следующим методом:

public void Conditional()// условн

{

if (token.Type != Token.TokenType.SELECT)

{

Expression(Token.TokenType.SELECT, token.Type);

}

}

1. Ошибка в операторе <вариант>:

<вариант>::=case <S>

<S>::= lit <W>

<S>::= else \n < спис\_опер>

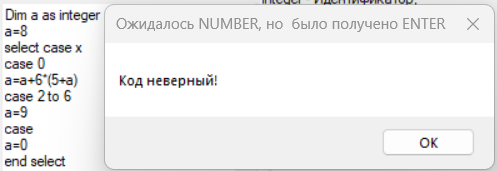


Рисунок 5 – ошибка в теле оператора <вариант>

После элемента «case» должен идти либо идентификатор, либо элемент «else». В случае, если это правило не выполняется, программа генерирует ошибку. В данном случае программа отработала корректно.

Данная ошибка генерируется следующим методом:

public void S()// S

{

switch (token.Type)

{

case Token.TokenType.NUMBER:

W();

break;

case Token.TokenType.ELSE:

Next();

break;

default:

Expression(Token.TokenType.NUMBER, token.Type);

break;

}

}

1. Ошибка при разборе сложного арифметического выражения.

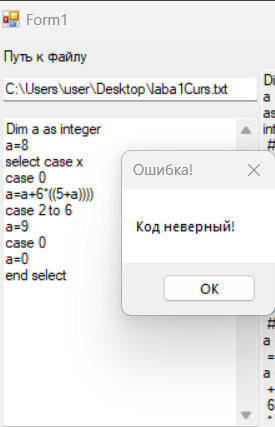


Рисунок 6 – ошибка при разборе сложного арифметического выражения

Данная ошибка генерируется методом:

private void RightBracket()

{

if (T.Count == 0)

{

D5();

}

else

{

switch (T.Peek().Type)

{

case Token.TokenType.LBRACKET:

D3();

break;

case Token.TokenType.PLUS:

D4();

break;

case Token.TokenType.MINUS:

D4();

break;

case Token.TokenType.MULTIPLICATION:

D4();

break;

case Token.TokenType.DIVISION:

D4();

break;

default:

Mistake("+, -, \*, /, ( , )");

break;

}

}

}

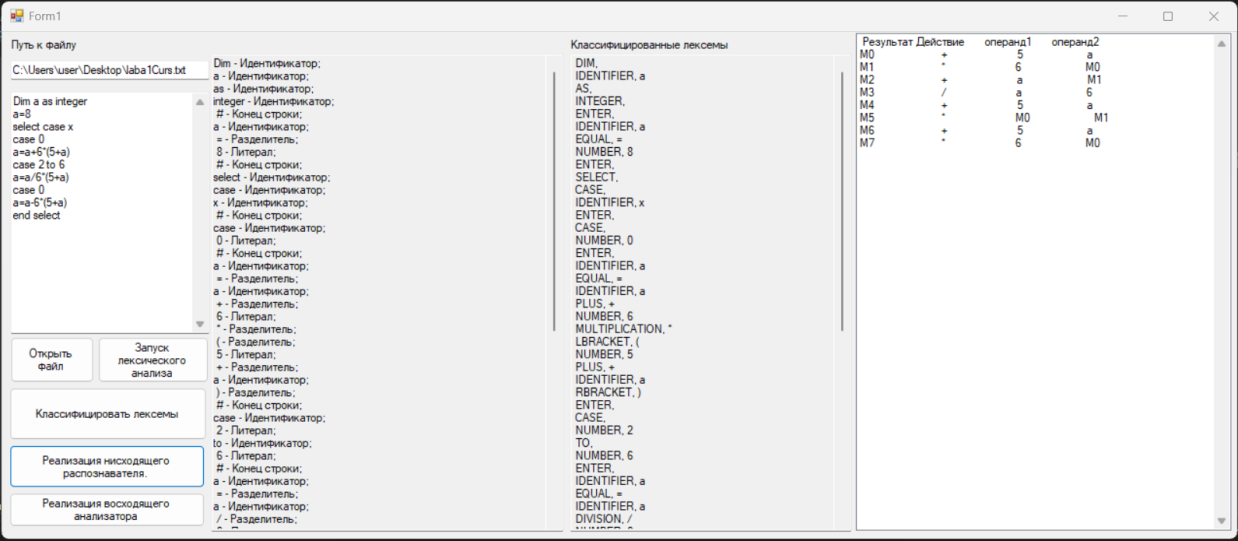


Рисунок 7 – успешный разбор сложного арифметического выражения

Результаты, полученные в ходе тестирования разработанного программного продукта, позволяют сделать вывод о том, что разработанная программа соответствует требованиям технического задания.

5. Руководство пользователя

Данное приложение создано для выполнения лексического и синтаксического анализа подмножества языка Visual Basic. Приложение имеет поддержку синтаксического разбора на основе LL(k)-грамматик, сложный арифметический оператор, оператор выбора select…..case.

При открытии приложения пользователь может видеть: кнопку «Открыть файл», с помощью которой пользователь может добавить код, используя файл. Также на форме присутствует кнопка «Запуск лексического анализа», которая проводит лексический анализ кода, введенного пользователем, стоит помнить, что это кнопка работает только после записи кода в программу. Кнопка «Классифицировать лексемы» выводит на экран весь список лексем, полученных при выполнении лексического анализа, и делит их на отдельные группы.

При нажатии кнопки «Реализация нисходящего анализа», проводится нисходящий анализ введенного кода, с использованием полученных при реализации классификации лексем. В случае, если список лексем соответствует грамматике выводится сообщение «Анализ проведен успешно», иначе выведется сообщение с ошибкой.

6. Руководство программиста.

Входными данными для транслятора является текст программы, полученный от пользователя, с синтаксисом подмножества языка программирования VB. Выходными данными транслятора являются таблица лексического разбора программы, список ключевых слов, разделителей, идентификаторов и литералов, используемых в программе после выполнения лексического анализа. В случае обнаружения ошибки в одном из анализаторов программа может вывести сообщение об обнаружении ошибки синтаксическим анализатором. Ниже представлены основные классы программы и их назначения:

Класс Recoznizer.cs: выполняет нисходящий анализ кода, полученного от пользователя. Все методы данного класса возвращают пустое значение. public void Begin() – запускает нисходящий анализ.

public void Programm() – выполняет первый шаг анализа, т.е проверку на соответствие текущего токена, токену DIM.

public void ListOfDescriptions() соответствует правилу <спис\_опис>.

public void ListOfOperations (), соответствует правилу <спис\_опер>.

public void Descriptions() соответствует правилу <опис>.

public void ListOfVariables() соответствует правилу <спис\_перем>.

public void Type() соответствует правилу <тип>.

public void U() - соответствует правилу <U>.

public void Operator() соответствует правилу <опер>.

public void Conditional() соответствует правилу <условн.>.

public void Assigment() соответствует методу <присв>.

public void Z() соответствует методу <Z>.

public void W() соответствует методу <W>.

public void Option() соответствует методу <вариант>.

public void X() соответствует методу <X>.

public void S() соответствует методу <S>.

public string Expression(Token.TokenType type, Token.TokenType type1). В случае ошибки генерирует сообщение, содержащее текущую лексему и ожидаемую лексему. Параметры: type - ожидаемый токен), type1(текущий токен).

Класс Validate.cs.

private char [] \_NumberOfSeparators – массив лексем, содержащиеся в грамматике.

public bool IsLetters(char symbol). Проверяет, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению.

Параметры – symbol char – символ из алфавита языка VB. Возвращаемое значение – true, в случае если соответствие найдено, и false если не найдено.

public bool IsNumbers(char symbol).

Проверяет, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению.

Параметры – symbol char – символ из алфавита языка VB. Возвращаемое значение – true, в случае если соответствие найдено, и false если не найдено.

public bool IsSeparator(char symbol). Проверяет, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению.

Параметры – symbol char – символ из алфавита языка VB. Возвращаемое значение – true, в случае если соответствие найдено, и false если не найдено**.**

Класс Token.cs.

Public TokenType Type - указывает различные типы токенов, которые могут быть заданы и возвращены из сканера языковой службы. Public string Value - указывает различные значения токенов, которые могут быть заданы и возвращены из сканера языковой службы.

Public Token(TokenType type). Представляет собой конструктор Token.

Параметры- type TokenType – тип токена.

private static bool IsDelimiter(Token token). Определяет, содержит ли последовательность указанный элемент. Параметры – token Token. Возвращаемое значение TokenType – токен из TokenType [] Delemiters.

public static bool IsSpecialWord (string word). Указывает, имеет ли указанная строка значение null, является ли она пустой строкой или строкой, состоящей только из символов-разделителей. Параметры – word string. Возвращаемое значение false, в случае если строка пустая, иначе word (токен, содержащийся в словаре SpecialWord).

public static bool IsSpecialSymbol (char ch). Определяет, содержится ли указанный ключ в словаре. Параметры – ch char. Возвращаемое значение ch (токен, содержащийся в словаре SpecialSymbols).

Класс ComplexExpression.cs, осуществляет разбор сложного логического выражения.

public struct Three – структура для хранения логических операций.

public Token operand – поле, содержащее токен, соответствующий операнду арифметической операции.

public Token operand1 - Поле, содержащее токен, соответствующий второму операнду арифметической операции.

Public Token action - Поле, содержащее токен, соответствующий оператору арифметической операции.

public Three(Token ac,Token op2,Token op1) - конструктор структуры Three.

Параметры

ac Token - поле, содержащее токен, соответствующий оператору логической

операции.

int index - целочисленное значение, содержит номер текущей операции сложного арифметического выражения.

public List<Three> threes - Список операций сложного арифметического выражения.

List<Token> tokens - список токенов для дальнейшего разбора.

Stack<Token> E - Стек для хранения операндов

Stack<Token> T - Стек для хранения операторов арифметических операций.

int nextlex - значение, которое содержит номер анализируемого токена из списка токенов.

public ComplexExpression (List<Token> x) - конструктор класса ComplexExpression.

Параметры - X List<Token>

public ComplexExpression (List<Token> inmet, int x)

Параметры - inmet List<Token>

Список токенов для разбора.

x int - значение, содержит номер текущей операции сложного арифметического выражения.

public int LastIndex { get { return index; } } - Свойство, возвращающее номер текущей операции сложного арифметического выражения.

private Token GetLexeme(int nextLex)

Метод, возвращающий токен из списка токенов.

Параметры - nextLex int

Целочисленное значение, содержит номер анализируемого токена из

списка токенов.

Возвращаемое значение – Token.

private void Operand() - метод, реализующий выбор элемента из списка токенов для разбора, после чего добавление его в стек операндов, затем переход к следующему элементу списка.

private void Action () - метод, реализующий извлечение двух элементов из стека операндов и элемента из стека операторов арифметических операций, запись этих элементов в структуру, которая в последствии отправлена в список операций сложного арифметического выражения.

Исключения – Exception

Невозможно выполнить операцию, т. к. количество операндов не

удовлетворяет условию.

public void Start()

Метод запуска разбора сложного арифметического выражения.

Исключение:

Вызов метода Mistake, в качестве параметра которому отдаётся строка с

ожидаемыми для разбора символами.

private void D1() - метод, реализующий запись анализируемого токена на вершину стека для хранения операторов арифметических операций, затем переход к следующему токену для дальнейшего анализа.

private void D2() - метод, реализующий вызов метода Action(), затем запись

анализируемого токена на вершину стека для хранения операторов

арифметических операций, затем переход к следующему токену для

дальнейшего анализа.

private void D3() - метод, реализующий извлечение элемента из стека для хранения операторов арифметических операций, затем переход к следующему токену для дальнейшего анализа.

private void D4() - метод, реализующий вызов метода Action().

private void D5() - метод, реализующий исключение Exception: Ошибка!

private void PlusMunis() - Метод, реализующий вызов метода в соответствии со столбцом «+» «-».

Исключения

Misteke()

Вызов метода Mistake, в качестве параметра которому отдаётся строка с

ожидаемыми для разбора символами.

private void MultiplicationDivision()

Метод, реализующий вызов метода в соответствии со столбцом \* /

Исключения

Mistake()

Вызов метода Mistake, в качестве параметра которому отдаётся строка с

ожидаемыми для разбора символами.

private void End()

Метод, реализующий вызов метода в соответствии со столбцом $

таблицы 3. Логика метода D6 реализована в методе Start, в связи с чем

отдельный метод D6 отсутствует.

Исключения

Mistake()

Вызов метода Mistake, в качестве параметра которому отдаётся строка с

ожидаемыми для разбора символами.

private void Mistake(string i)

Метод обработки ошибок.

Параметры

i string

Строка, содержащая ожидаемые для разбора символы или правила.

Исключения

Exception $"Ожидалось {i}, но получено {tokens[nextLexem].Value}"

7. Заключение:

В результате работы создан компилятор программы на подмножестве языка Visual Basic. Данная программа может выполнять лексический, синтаксический разборы, а так же разбор сложного арифметического выражения. Для синтаксического разбора был использован метод LR(k)-грамматик. Для разбора сложного арифметического выражения использован метод Бауэра-Замельзона.

В ходе работы была составлена грамматика подмножества языка Visual Basic, реализованы методы лексического и синтаксического разбора полученного кода, а также выполнено тестирование программы.

Подводя итоги, можно считать, что разработанный транслятор соответствует требованиям технического задания.

8. Список используемой 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. — ISBN 987-5-7433-  
2968-7. — Текст : электронный // Электронно-библиотечная система IPR BOOKS :  
[сайт]. — URL: [https://www.iprbookshop.ru/76519.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F76519.html&cc_key=) (дата обращения:  
15.04.2021). — Режим доступа: для авторизир. пользователей. - DOI:  
[https://doi.org/10.23682/76519](https://vk.com/away.php?to=https%3A%2F%2Fdoi.org%2F10.23682%2F76519&cc_key=) - [https://www.iprbookshop.ru/76519.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F76519.html&cc_key=)  
 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. — ISBN 978-5-  
9275-2397-9. — Текст : электронный // Электронно-библиотечная система IPR  
BOOKS : [сайт]. — URL: [https://www.iprbookshop.ru/87427.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F87427.html&cc_key=) (дата обращения:  
15.04.2021). — Режим доступа: для авторизир. пользователей -  
[https://www.iprbookshop.ru/87427.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F87427.html&cc_key=)  
 3. Пентус, А. Е. Математическая теория формальных языков : учебное  
пособие / А. Е. Пентус, М. Р. Пентус. — 3-е изд. — Москва : ИнтернетУниверситет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020.  
— 218 c. — ISBN 978-5-4497-0662-1. — Текст : электронный // Электроннобиблиотечная система IPR BOOKS : [сайт]. — URL:  
[https://www.iprbookshop.ru/97548.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F97548.html&cc_key=) (дата обращения: 15.04.2021). — Режим  
доступа: для авторизир. пользователей - [https://www.iprbookshop.ru/97548.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F97548.html&cc_key=)  
 4. Миронов, С. В. Формальные языки и грамматики : учебное пособие для  
студентов факультета компьютерных наук и информационных технологий / С. В.  
Миронов. — Саратов : Издательство Саратовского университета, 2019. — 80 c. —  
ISBN 978-5-292-04613-4. — Текст : электронный // Электронно-библиотечная  
система IPR BOOKS : [сайт]. — URL: [https://www.iprbookshop.ru/99047.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F99047.html&cc_key=) (дата  
обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей -  
[https://www.iprbookshop.ru/99047.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F99047.html&cc_key=)  
 5. Малявко, А. А. Формальные языки и компиляторы : учебник / А. А.  
Малявко. — Новосибирск : Новосибирский государственный технический  
университет, 2014. — 431 c. — ISBN 978-5-7782-2318-9. — Текст : электронный //  
Электронно-библиотечная система IPR BOOKS : [сайт]. — URL:  
[https://www.iprbookshop.ru/47725.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F47725.html&cc_key=) (дата обращения: 15.04.2021). — Режим  
доступа: для авторизированных пользователей - [https://www.iprbookshop.ru/47725.html](https://vk.com/away.php?to=https%3A%2F%2Fwww.iprbookshop.ru%2F47725.html&cc_key=).

Приложение 1:

Изображение работы программы

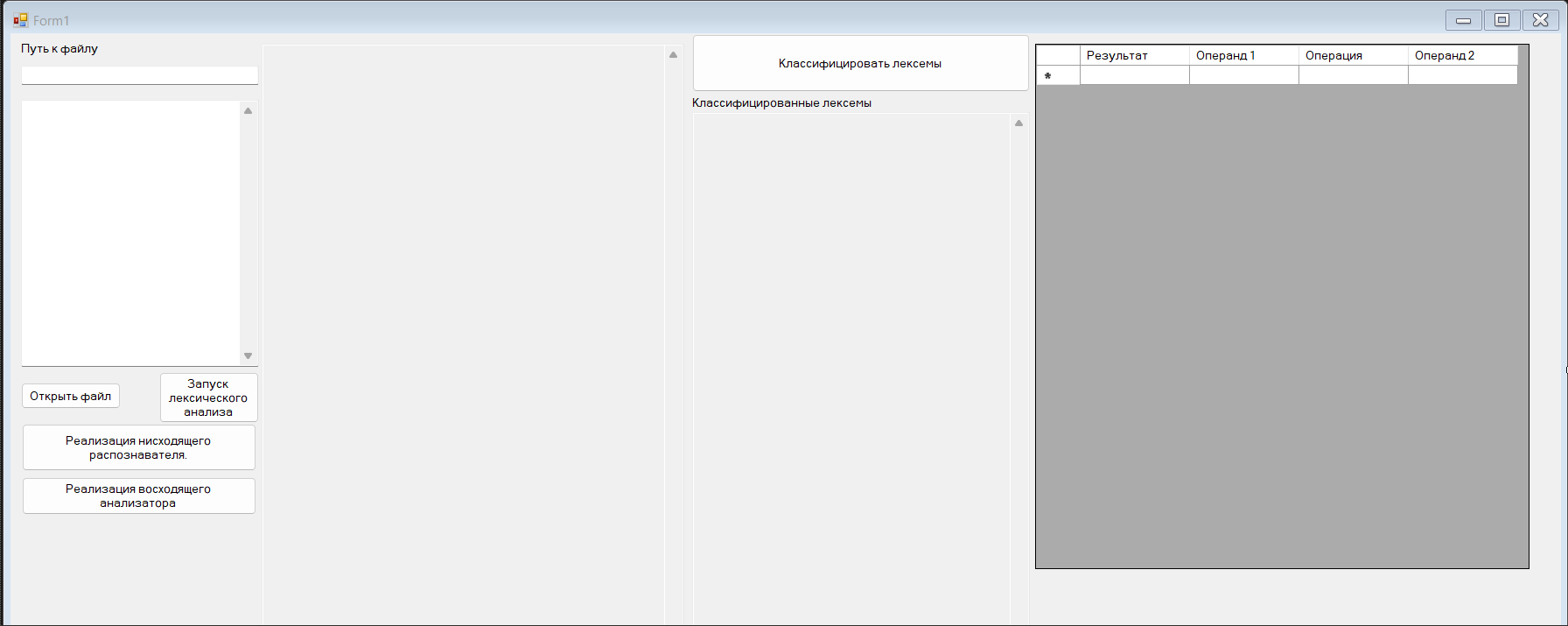


Рисунок 6 - Основное окно приложения

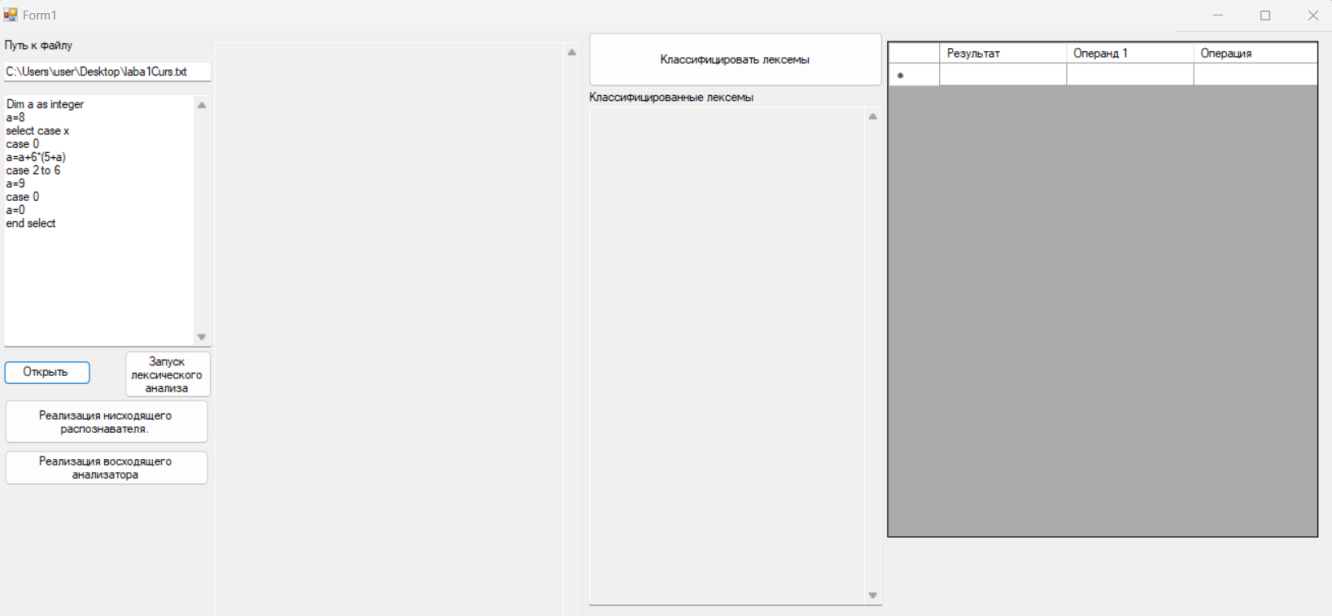


Рисунок 7 – Загрузка кода из внешнего файла

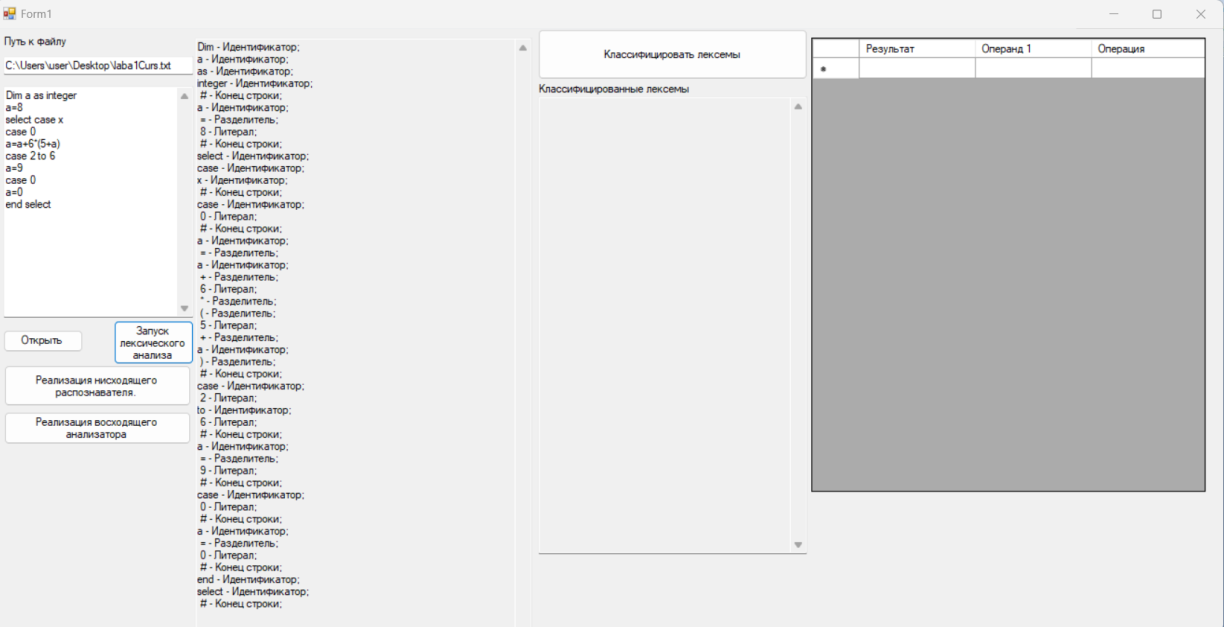


Рисунок 8 – Выполнение лексического анализа

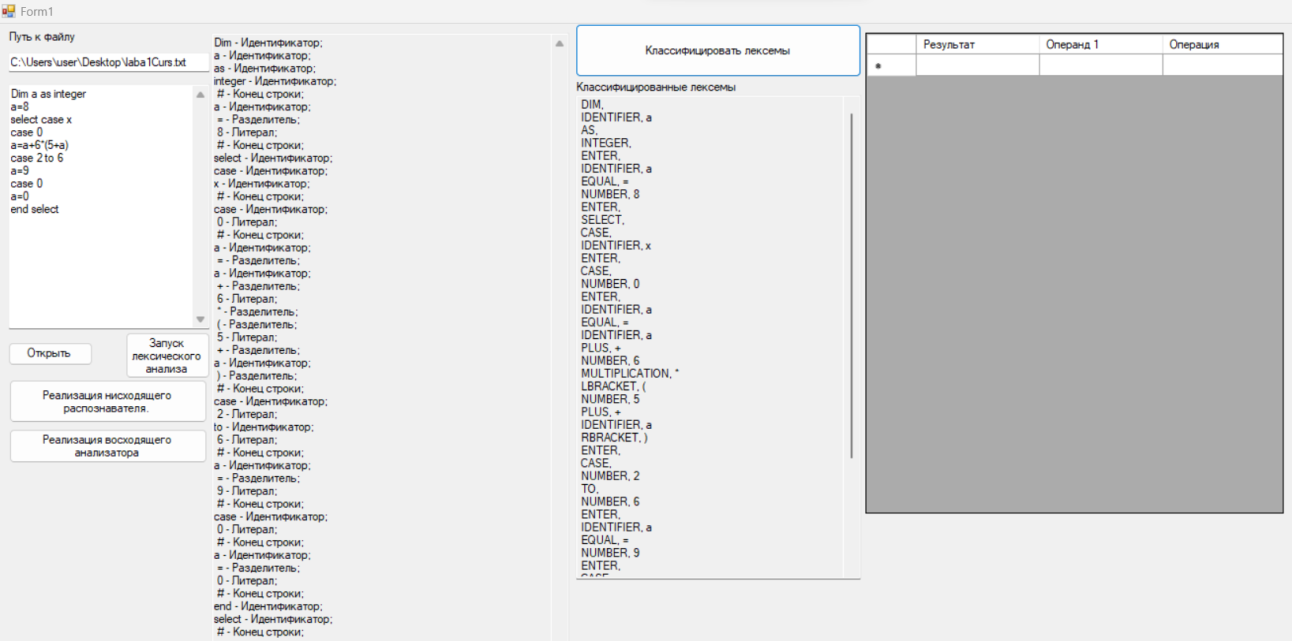


Рисунок 9 – Выполнение классификации лексем

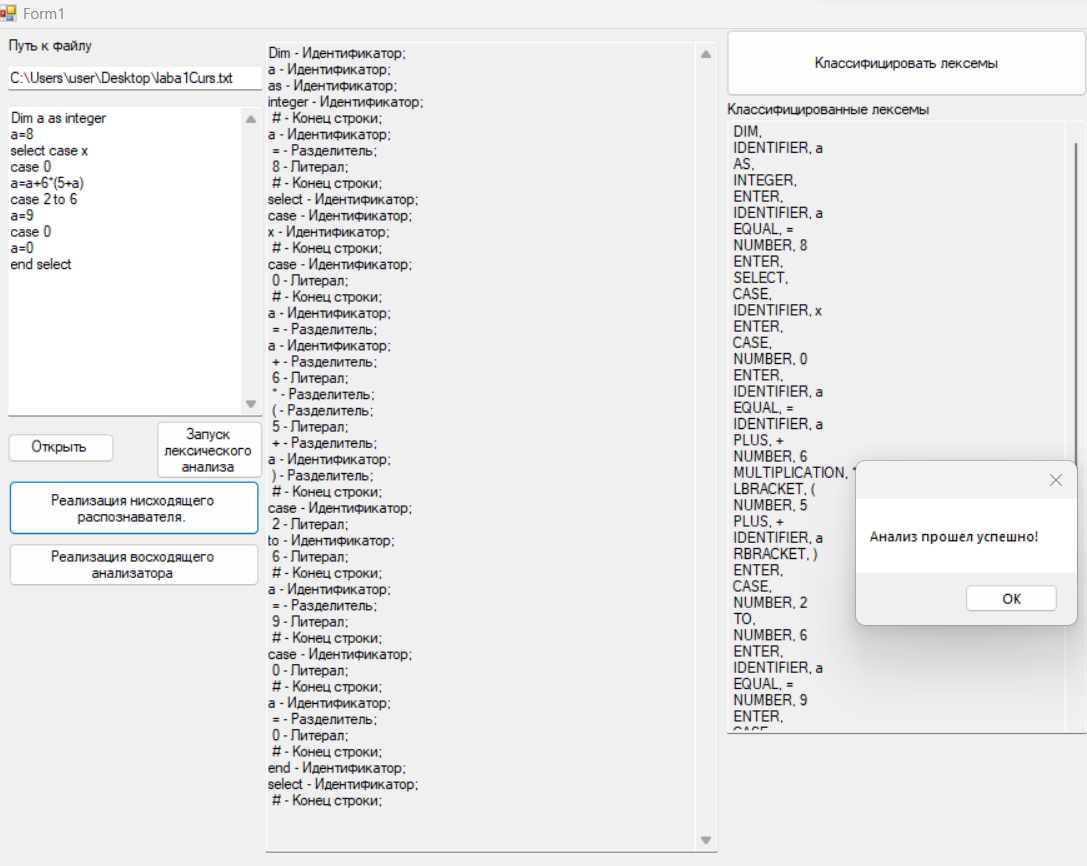


Рисунок 10 – Выполнение нисходящего анализатора и вывод сообщения об успешном разборе.

Приложение 2

Таблица 2 - решающая таблица нисходящего анализа

|  |  |
| --- | --- |
| Правила грамматики | ∪ |
|  |
| <программа>::=Dim <спис\_опис> \n <спис\_опер>\n | Dim |
| <спис\_опис>::= <опис> <доп.опер> | id |
| <доп.опер>::= Ɛ  <доп.опер>::= <спис\_опис> | \n,  id |
| <опис>::=<спис\_перем> as <тип> | id |
| <спис\_перем> ::=id <U> | id |
| <U>::= Ɛ  <U>::= , id <спис\_перем> | as  , id |
| <тип>::= integer  <тип>::= long  <тип>::= double | integer  long  double |
| <спис\_опер>::=<опер>\n <Z> | select,  id |
| <Z>::= Ɛ  <Z>::= <спис\_опер> | \n  select,id |
| <опер>::=<условн.>  <опер>::= <присв.> | select  id |
| <условн.>::= select case id \n <вариант> <спис.вариантов> end select | select |
| <спис.вариантов>::= <вариант> <X> | case |
| <X>::= Ɛ  <X>::= <спис.вариантов> | end  case |
| <вариант>::=case <S> | case |
| <S>::= lit <W>  <S>::= else \n < спис\_опер > | lit  else |
| <W>::= \n <спис\_опер>  <W>::= to lit \n < спис\_опер> | \n  to |
| <присв.>::=id= expr \n | id |