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»**

Факультет ИТР

Кафедра ПИн

Курсовая работа

По Теория автоматов и формальных языков

Тема: Транслятор с подмножества языка С

Руководитель:

Кульков Я.Ю.
(фамилия, инициалы)

(подпись) (дата)

Студент ПИн - 121
(группа)

Карасев Э.Ф.
(фамилия, инициалы)

(подпись) (дата)

Муром 2023

Содержание

Оглавление

Введение

С появлением первых высокоуровневых языков программирования (приближенных к человеческому языку) необходимо было создать программы, которые будут переводить программы в двоичный машинный код с которым работает компьютер. Данными программами стали трансляторы.

Транслятор – это программа или часть программного обеспечения, производящая трансляцию полученной программы. Транслятор преобразует программу написанную на одном из языков программирования, в программу, написанную на другом языке. Также транслятор осуществляет диагностику программы на ошибки, формирует словари лексем и их классификации.

Язык, на котором была представлена программа называется исходным языком, а код программы исходным кодом. То есть исходными данными транслятора будет текст программы, соответствующий синтаксическим требованиям исходного языка.

Все трансляторы состоят из следующих этапов:

- Лексический анализ/разбор – определение лексем из исходного кода по 4 типам: Ключевые слова языка, разделители, идентификаторы (Названия переменных) и литералы (Числовые переменные).
- Синтаксический анализ/разбор на основе LL или LR грамматик выполняет выделение синтаксических конструкций в тексте исходной
- программы, обработанной лексическим анализатором. На этом же этапе проверяется синтаксическая правильность программы.
- Разбор сложных арифметических и сложных логических выражений методами Дейкстры или Бауэра Замельзона.

Актуальность данной темы заключается в получении практических и теоретических навыков работы с транслятором подмножества языка С. Для успешного выполнения этой работы необходимо изучить работу синтаксического и нисходящего анализатора.

1. Анализ технического задания

В представленной курсовой работе необходимо спроектировать транслятор подмножества языка С. Анализируя тему данной курсовой работы, требуется, чтобы в созданной программе присутствовали:

– Язык для трансляции – С;

– Обеспечить развернутую диагностику ошибок;

– Реализовать класс транслятора;

– Синтаксический разбор - на основе LL(k)-грамматик;

– Разбор логических выражений выполнять методом Бауэра-Замельзона;

– В языке поддерживаются

– у идентификатора 8 символов значащие;

– не менее 3х директив описания переменных;

– простой арифметический оператор;

– сложное логическое выражение;

– условный оператор if (…){…}else{…}

Представленная курсовая работа реализуется в несколько шагов: Создание лексического анализа, который в свою очередь выполняет анализ полученных данных, 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.

2.Описание грамматики языка.

Язык С имеет свой алфавит, в который входят специализированные символы, цифры и буквы, составляющие базовые символы. Этот алфавит включает прописные и строчные буквы латиницы, арабские цифры, знаки арифметических и логических операций, ограничители, разделители и специальные символы, которые могут использоваться для формирования конструкций языка С, таких как данные, операторы, выражения и функции. При написании имен переменных необходимо соблюдать ограничения, например использовать только символы алфавита, цифры и знак подчёркивания, а также не использовать служебные слова в качестве идентификаторов. Для разбора логических выражений с множественными операторами и скобками может быть использован алгоритм Бауэра-Замельзона вместо синтаксического анализатора. Основываясь на правилах написания операторов, можно составить грамматику данного подмножества языка С.

Грамматика подмножества языка С основывается на правилах написания операторов. Например, одно из таких правил - операторы могут быть объединены в группы на основе их приоритетов, где операторы с более высоким приоритетом выполняются раньше, чем операторы с более низким приоритетом.

Также, грамматика подмножества языка С включает такие средства, как условные операторы, циклы, операторы присваивания, операторы ввода/вывода, функции и т.д. Различные операторы и конструкции могут быть комбинированы для создания более сложных программ.

Важно отметить, что конструкции языка С описываются с помощью синтаксических правил, которые могут быть формально определены в виде Бэкуса-Наура нотации (BNF). BNF - это формальный метаязык, позволяющий описывать грамматики любых формальных языков и их подмножеств.

Таким образом, грамматика подмножества языка С описывает синтаксис языка и позволяет создавать сложные программы, используя различные операторы и конструкции.

Имена переменных в С подчиняются ряду ограничений:

- Идентификатор может содержать символы английского алфавита, цифры и знак подчёркивания;

- Первым символом в идентификаторах может стоять или буква или знак подчёркивания. Использовать цифры в качестве первого символа недопустимо;

- Запрещено использовать в качестве имен идентификаторов служебные слова: имена встроенных процедур и функций, операций и т.д. Имя переменной должно начинаться с буквы (не важно маленькой или большой), после нее могут идти как буквы так и цифры;

- Не допускается использование символов (пробел, запятая и т.д.). Кроме того, нельзя использовать в качестве переменной служебные слова или операторы.

В курсовой работе предусматривается что математическое выражение будет иметь сложную структуру, то есть иметь

множественные операторы и скобки, следовательно разбор такого выражения целесообразно выполнять другим алгоритмом. Вместо синтаксического анализатора может использоваться построение дерева выражения при помощи алгоритма Бауэра-Замельзона.

Исходя из правил написания операторов языка, можно составить грамматику данного подмножества языка С.

Полученная грамматика:

G= T,N,P,<программа>

T= {main, int, string, bool, (,), {, }, int, =, ;, if, else, >, <, +, -, *, /, &&, id, lit, expr}

N={<программа>, <спис_опис.>, <спис_опер> <опис.>, <тип>,
<спис_перем.>, <опер.>, <условн.>, <блок_опер.>, <присв.>, <знак>,
<операнд> }

P = {

<программа> ::= main() { <спис_опис> <спис_опер> }

<спис_опис> ::= <опис> | <опис> <спис_опис>

<опис> ::= <тип> <спис_перем>;

<тип> ::= int | bool | string

<спис_перем> ::= id | <спис_перем>, id

<спис_опер> ::= <опер> | <опер> <спис_опер>

<опер.> ::= <условн.> | <присв.>

<условн> ::= if (expr) <блок. опер.> | if (expr) <блок_опер> else
<блок_опер>

<блок_опер> ::= <опер> | { <спис_опер> }

<присв> ::= id = <операнд> <знак> <операнд>; | id = <операнд>;

<знак> ::= + | - | \ | *

<операнд> ::= id |lit

<программа> ::= main() { <спис_опис> <спис_опер> }

<программа> ::= main() { <опис><спис_опис> <опер> <спис_опер> }

<программа> ::= main() { <опис> <опис> <опер> <опер> <спис_опер> }

<программа> ::= main() { <опис> <опис> <опер> <опер> <опер> }

<программа> ::= main() { <тип> <спис_перем>; <тип> <спис_перем>;
<присв.> <присв.> <условн.> }

<программа> ::= main() { int <спис_перем>, id; int id; id = <операнд>; id
= <операнд>; if (expr) <блок_опер> else <блок_опер>}

<программа> ::= main() { int id, id; int id; id = lit; id = lit if (expr)
{ <спис_опер> } else { <спис_опер> }}

<программа> ::= main() { int id, id; int id; id = lit; id = lit; if (expr)
{ <опер> } else { <опер> <спис_опер> }}

<программа> ::= main() { int id, id; int id; id = lit; id = lit; if (expr)
{ <присв.> } else { <опер> <опер> }}
<программа> ::= main() { int id, id; int id; id = lit; id = lit; if (expr) { id =
<операнд>; } else { id = <операнд> <знак> <операнд>; id =
<операнд>; }}
<программа> ::= main() { int id, id; int id; id = lit; id = lit; if (expr) { id =
id; } else { id = id + lit; id = id; }}

Таблица 1 - Пример формирования цепочки вывода

Анализируемый фрагмент программы	Полученный синтаксический вывод

main () {
int a, b;
int c;
b=0; a=5;
if (a>b && b<c) { c=a; }
else {c=b+5; b=a;}
}
	main() {
	int id, id;
	int id;
	id = lit; id = lit;
	if (expr) { id = id; }
	else {id = id + lit; id = id;}
	}

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

1. Поиск леворекурсивных правил.

1.1 <спис_опис.> ::= <опис.> | <опис.> ,<спис_опис>

Избавляемся от левой факторизации:

<спис_опис.> ::= <опис> | <опис><доп.опис.>

<доп.опис.> ::= E | <доп.опис.2>

<доп.опис> ::= ,<спис_опис> | ,<спис_опис><доп.опис.>

<доп.опис2> ::= , <опис><доп.опис.>

1.2 <спис_перем> ::=id | <спис_перем>, id

Избавляемся от левой факторизации:

<спис_перем> ::= id | id<альт_2>

<спис_перем> ::= id<X>

<X> ::= E | <альт_2>

<альт_2> ::= , id | , id<альт_2>

<альт_2> ::= , id<X>

1.3 <спис_опер.> ::= <опер> | <опер> ,<спис_опер.>

Избавляемся от левой факторизации:

<спис_опер.> ::= <опер> | <опер><доп.опер>

<спис_опер.> ::= <опер><доп.опер>

<доп.опер> ::= E | <доп.опер.2>

<доп.опер> ::= ,<спис_опер> | ,<спис_опер><доп.опер>

<доп.опер.2> ::= , <опер><доп.опер>

1.5 <условн.> ::= if expr <блок. опер.> | if expr <блок_опер.> else {<блок_опер.> }

Избавляемся от левой факторизации:

<условн.> ::= if expr <знак> expr <блок. опер.>< доп.условн.>

<доп.условн> ::= E | < доп.условн.2>

<доп.условн2> ::= else <блок_опер.>

1.4 <присв.> ::= id = <операнд> | id = <операнд> <знак> <операнд>

Избавляемся от левой факторизации:

<присв> ::= id = <операнд> <доп. присв.>

<доп. присв> ::= E | <доп.присв.2>

<доп. присв2> ::= <знак> <операнд>

Докажем, что полученная после преобразования грамматики

является LL(k) и определим k: (Табл.2)

Таблица 2 - Пример построения решающей таблицы

Правила грамматики	Терминальные символы
	FIRST1
<программа> ::= main() { <спис_опис.>; <спис_опер.>; }	main
<спис_опис.> ::= <опис> <доп_опис.>	int, bool, string
<доп_опис.> ::= Ɛ <доп_опис.> ::= <доп.опис.2>	; ,
<доп.опис_2> ::= , <опер><доп.опис.>	,
<опис.> ::= <тип> <спис_перем.>	int, bool, string
<тип> ::= int <тип> ::= bool <тип> ::= string	int bool string
<спис_перем.> ::= id <X>	id
<X>::= Ɛ <X>::= <альт_2>	;, int, bool, string ,
<альт_2> ::= , id<X>	,
<спис_опер.> ::= <опер><доп.опер.>	if, id
<доп.опер.> ::= E <доп.опер.> ::= <доп.опер.2>	; ,
<доп.опер.2> ::= , <опер><доп.опер.>	,
<опер.> ::= <условн.> <опер.> ::= <присв.>	If id
<условн.> ::= if expr <знак> expr <блок.опер.>< доп.условн.>	if
<доп.условн.> ::= E <доп.условн.> ::= < доп.условн.2>	; id
<доп.условн.2> ::= else <блок_опер.>	else
<блок_опер.> ::= <опер.>; <блок_опер.> ::= { <спис_опер.>; }	id {
<присв.> ::= id = <операнд>< доп. присв.>	id
<доп. присв.> ::= E <доп. присв.> ::= <доп. присв.2>	; id
<доп. присв.2> ::= <знак> <операнд>	&&, ||
<знак> ::= && <знак> ::= ||	&& ||
<операнд> ::= id <операнд> ::= lit	Id lit

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 проверки её правильности. Обнаруженные лексические ошибки фиксируются.

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». В случае получения знака 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 представлен на таблице 3 ниже:

Пример работы лексического анализатора представлен на таблице 4 ниже:

Таблица 3 - Пример работы сканера

Main	Идентификатор
;	Разделитель
5	Литерал
else	Идентификатор
,	Разделитель

Таблица 4 - Пример работы лексического анализатора

MAIN	main
LITERAL	5
ELSE	Else
SEMICOLON	;
COMMA	,

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.

Синтаксический анализ, взаимодействующий с синтаксическими конструкциями языка с помощью токенов, является частью компилятора. Токен - это имя и набор атрибутов, представляющий тип лексической единицы. Лексический анализ использует токены для токенизации, то есть классификации разделов строки входных символов. Большинство методов анализа данных используют нисходящие или восходящие алгоритмы. Нисходящие алгоритмы строят вывод, начиная от аксиомы грамматики и заканчивая терминальными символами, и связаны с LL-грамматикой.

Метод Бауэра-Замельзона был использован для анализа сложных выражений, где применяются два стека и таблица переходов. Стек E служит для хранения операндов, а стек T - для хранения знаков операций.

Для стека E доступны две операции:

K(id), которая выбирает элемент с указанным именем из входного потока, помещает его на вершину стека E и переходит к следующему элементу, а также K(OP), которая извлекает два верхних операнда из стека E, выполняет указанную операцию и записывает результат на вершину

стека E.

Операции над стеком T определяются таблицей переходов, где описаны действия, которые должен совершить транслятор при анализе выражения. В таблице 3 представлен полный перечень действий для арифметических и логических операторов.

Таблица 5 - Список действий для логических операторов

		Входной символ					
		$	(	<>=	+ - ||	* / &&	)
Символ на вершине стека	ε	D6	D1	D1	D1	D1	D5
	(	D5	D1	D1	D1	D1	D3
	<>=	D4	D1	D2	D1	D1	D4
	+ - ||	D4	D1	D4	D2	D1	D4
	* / &&	D4	D1	D4	D4	D2	D4

Алгоритм метода Бауэра-Замельзона состоит из следующих этапов:

1) Просматриваем входную строку слева направо;

2) Если текущий элемент – операнд, то выполняем операцию K (операнд).

3) Если текущий элемент – операция, то читаем элемент с вершины стека T, из таблицы переходов выбираем действие, соответствующее паре (элемент с вершины стека, символ входного потока). Выполняем выбранное действие. Возможны шесть действий при прочтении операции OP из входной строки и операции OP1 на вершине стека T:

D1. Записать OP в стек T и читать следующий символ строки.

D2. Удалить OP1 из стека T и генерировать команду K(OP1);

Записать OP в стек T и читать следующий символ строки.

D3. Удалить OP1 из стека T и читать следующий символ строки.

D4. Удалить OP1 из стека T и генерировать команду K(OP1);

D5. Ошибка в выражении. Конец разбора.

D6. Успешное завершение разбора.

4. Тестирование

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

Проверка лексического анализатора

Проверка на корректной лексике

Для проверки лексического анализатора вводится некоторый корректный текст программы на языке С и происходит запуск анализатора, который в результате своей работы формирует таблицы типов лексем с их нумерацией, а также таблицу токенов (Рисунок 1).

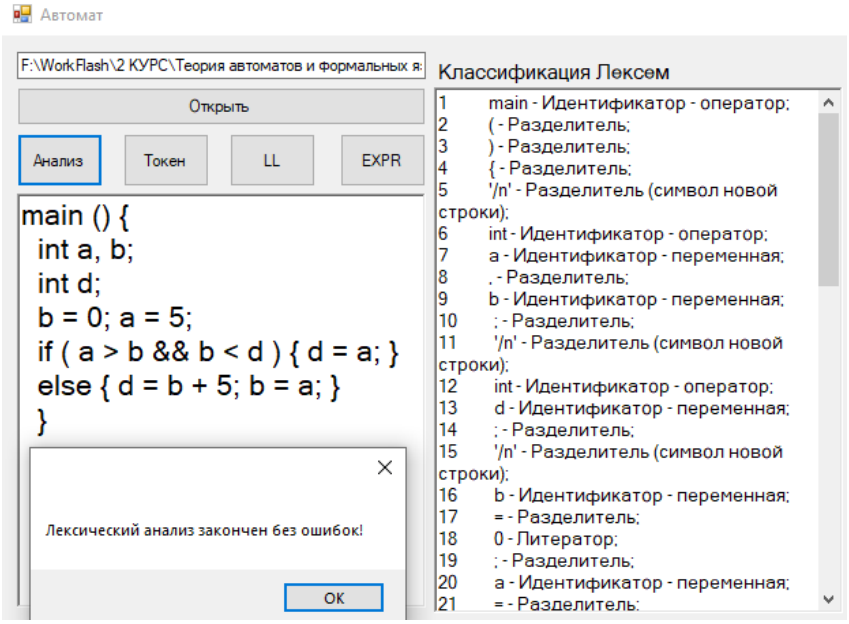

Рисунок 1 – Результат работы лексического анализатора

Проверка на некорректной лексике

Для проверки лексического анализатора на некорректной лексике вводится некорректный текст программы и выполняется запуск анализатора (Рисунок 2)

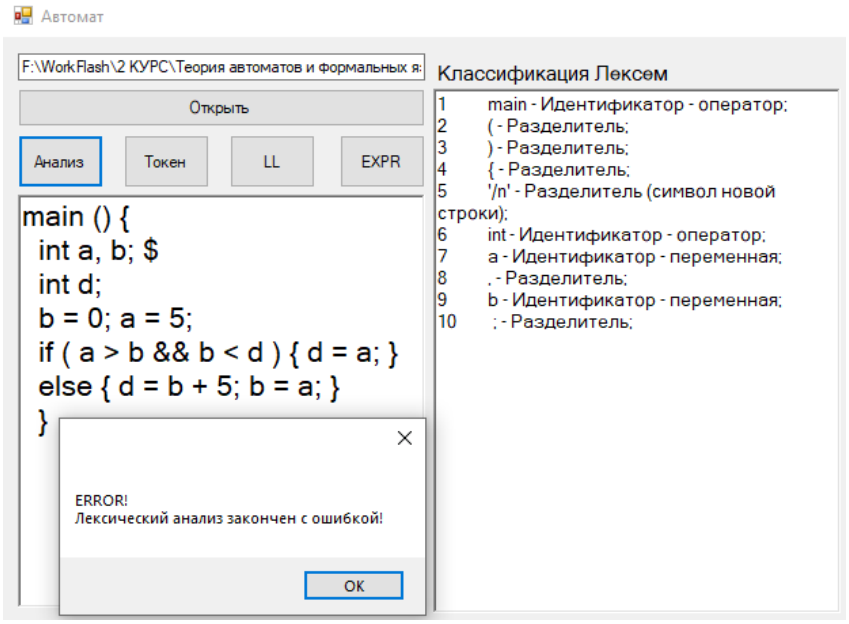

Рисунок 2 – Отправка некорректной лексики в лексический анализатор

Проверка синтаксического анализатора

Для проверки синтаксического анализатора следует протестировать его работу на корректно введённом в программном коде. В ходе теста были использованы все возможные конструкции подмножества языка С.

В результате работы синтаксического анализатора он успешно проанализировал программный код (Рисунок 4).

Проведена проверка работы синтаксического анализатора с дополнительным объявлением переменных, вводом вложенного условного оператора if (Рисунок 5), также проверка работоспособности анализатора без блока else (Рисунок 6).

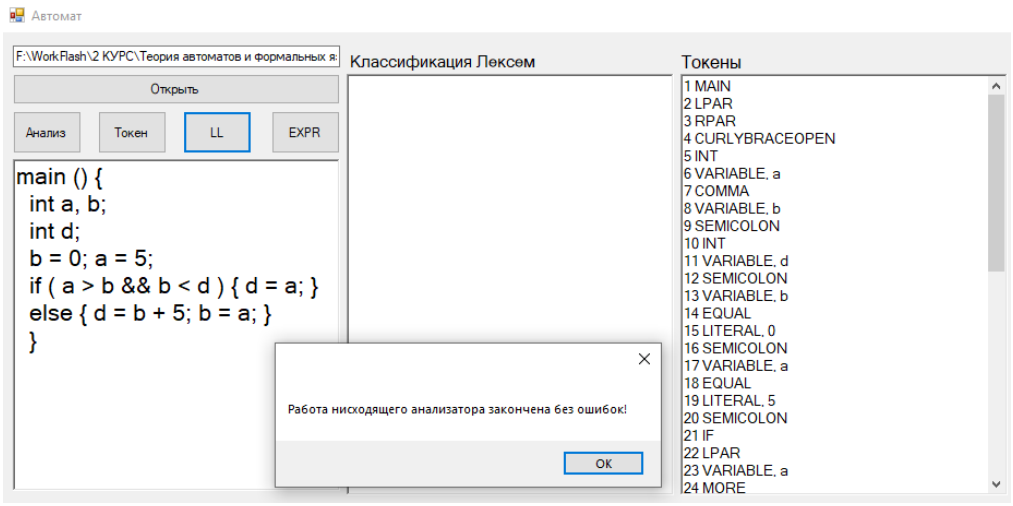

Рисунок 4 – Успешный разбор кода синтаксическим анализатором.

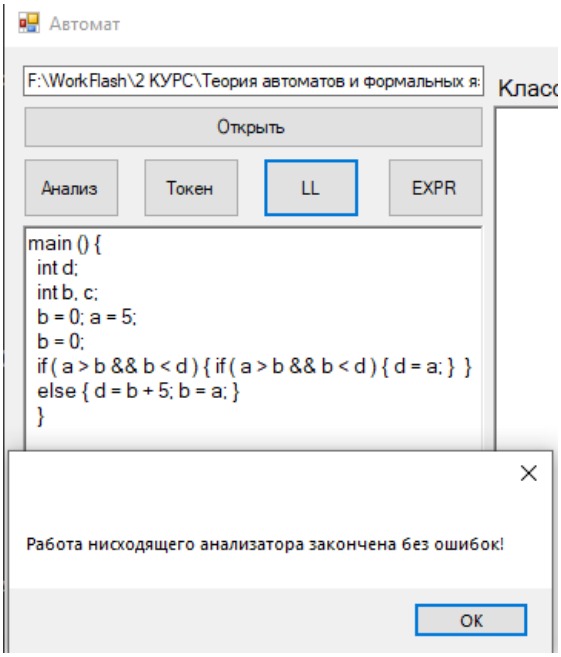

Рисунок 5 – Успешный разбор кода синтаксическим анализатором с двойным присвоением, вложенным условием if.

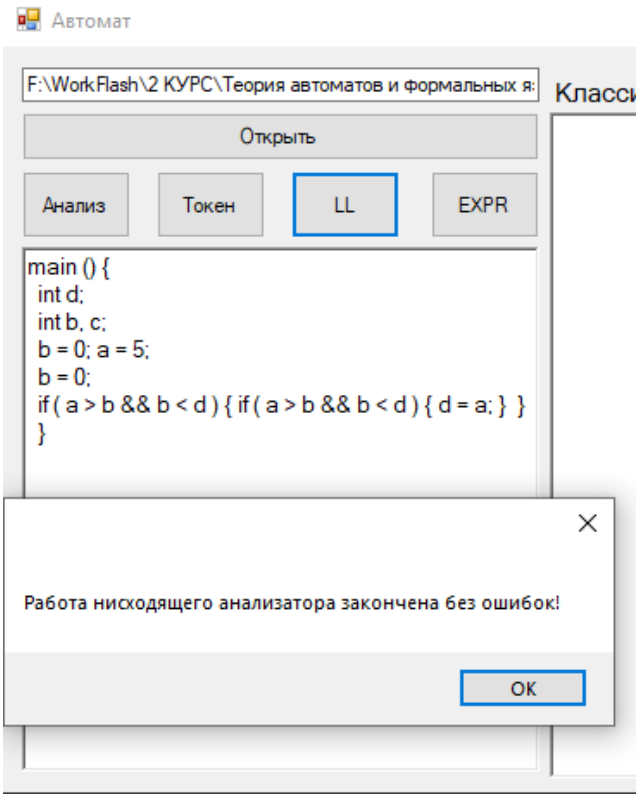

Рисунок 6 - Успешный разбор кода синтаксическим анализатором без “else”

Проверка на синтаксически некорректном коде

После проверки на корректном программном коде следует протестировать синтаксический анализатор на программном коде с ошибками.

В соответствии с правилом (табл.2) «<присв> ::= id = <операнд><X>, где <X> ::= Ɛ | , <спис_перем.>» при попытки ввода присвоения без “=”, то программа выдаст ошибку и сообщит об ожидаемой лексеме и о полученной (Рисунок 7).

В соответствии с правилом (табл.2) «<спис_перем.> ::= id <U>;, где <U> ::= Ɛ | <знак> <операнд>, где <операнд> ::= id | lit, где <знак> ::= + | - | \ | *» при попытки ввода присвоения без “;”, то программа выдаст ошибку и сообщит об ожидаемой лексеме “;” и о полученной (Рисунок 8).

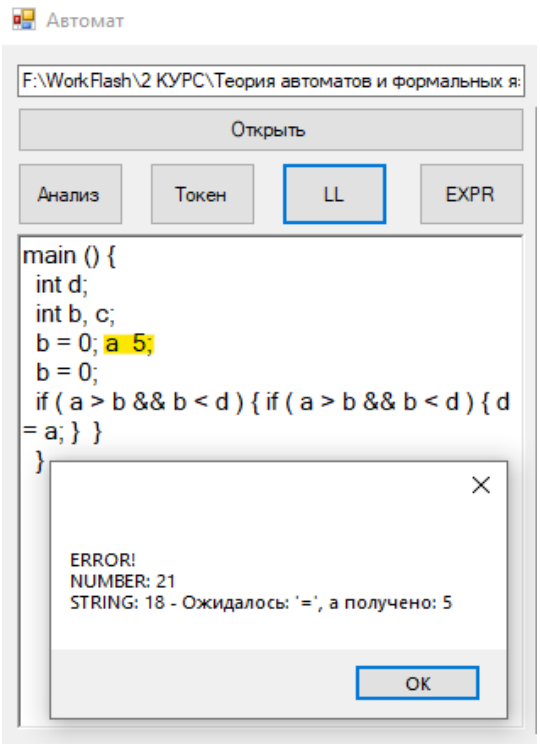

Рисунок 7 – Некорректное объявление присвоения

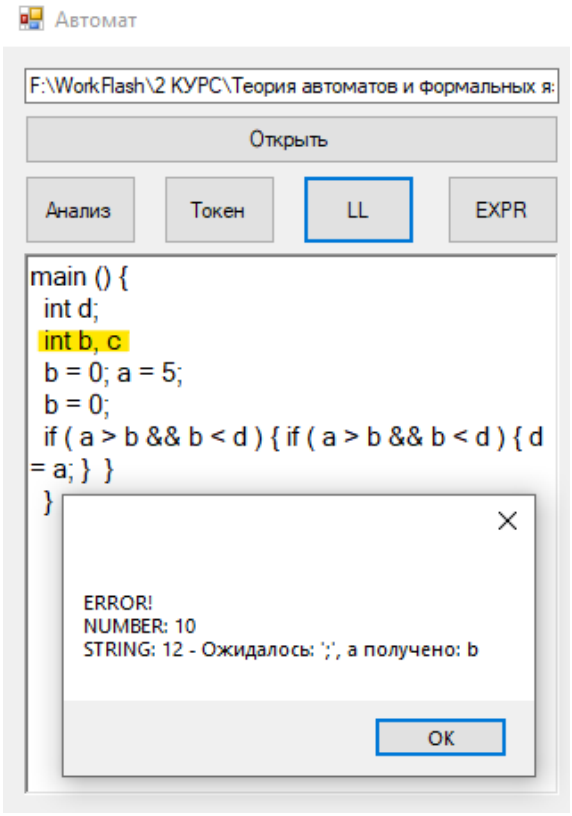

Рисунок 8 – Некорректное объявление переменных

Проверка разбора сложного выражения

Корректный разбор сложного выражения

В ходе правильного заполнения логической операции программа успешно проведёт анализ и отобразит его в текстовых полях. (Рисунок 9)

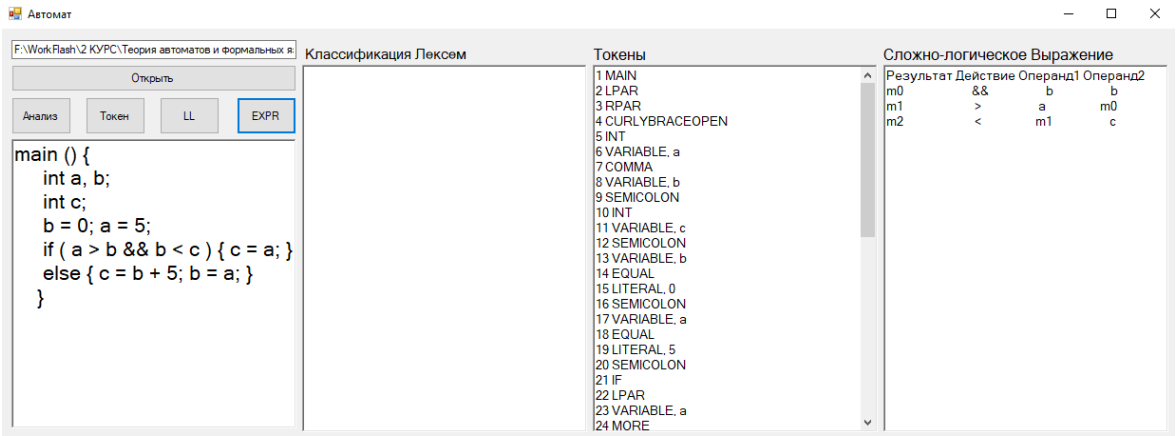

Рисунок 9 – Корректная работа программы

Некорректный разбор сложного выражения

В ходе ошибочного заполнения логического выражения (в данном примере повторное написание логического “и” (&&) Рисунок 10) программа выведет уведомление об ошибке и закончит разбор сложного выражения.

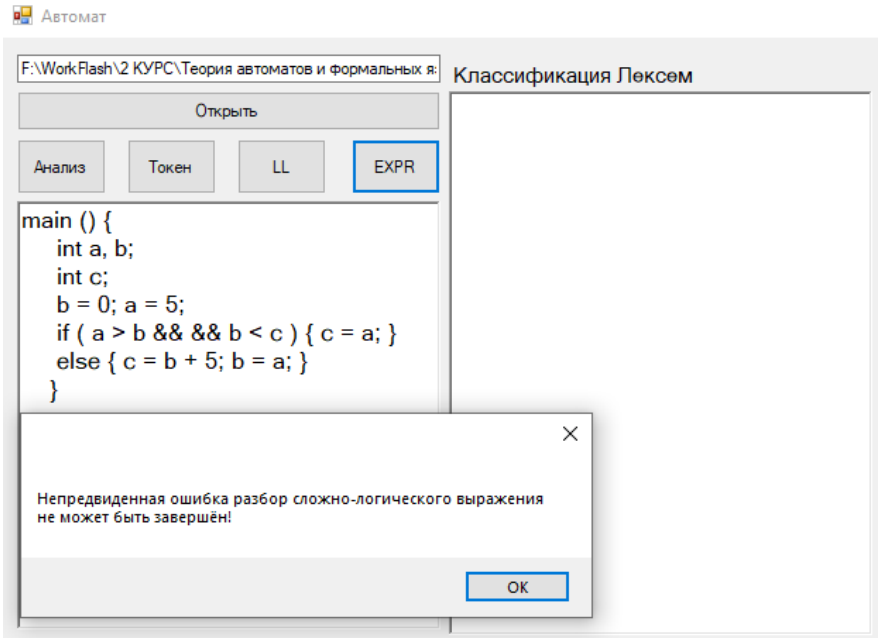

Рисунок 10 – Некорректная работа программы

5. Руководство пользователя

После запуска программы будет представлено окно (Рисунок 11).

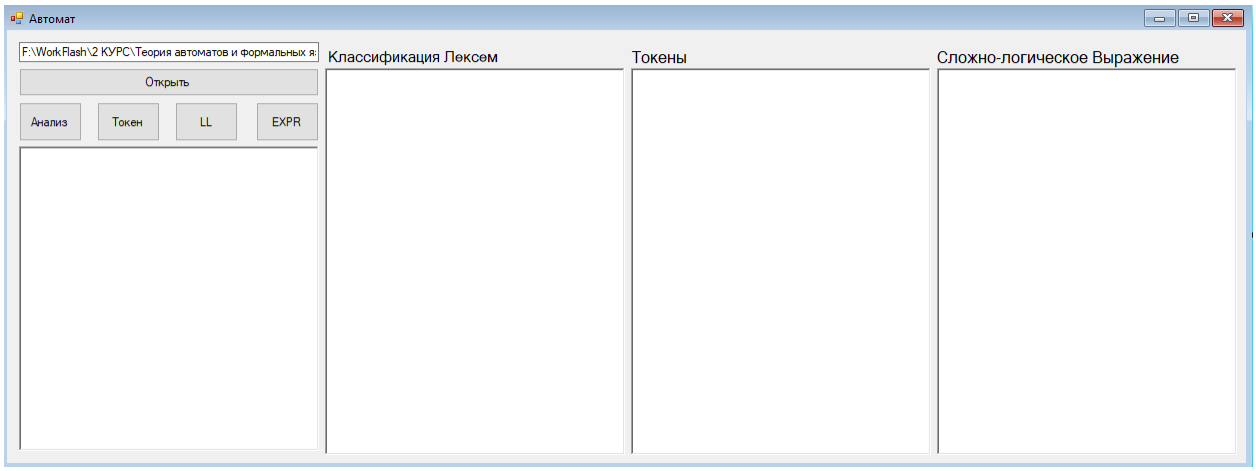

Рисунок 11 – интерфейс программы

Выделенная красным квадратом область окна – является приборной панелью данного приложения (Рисунок 12).

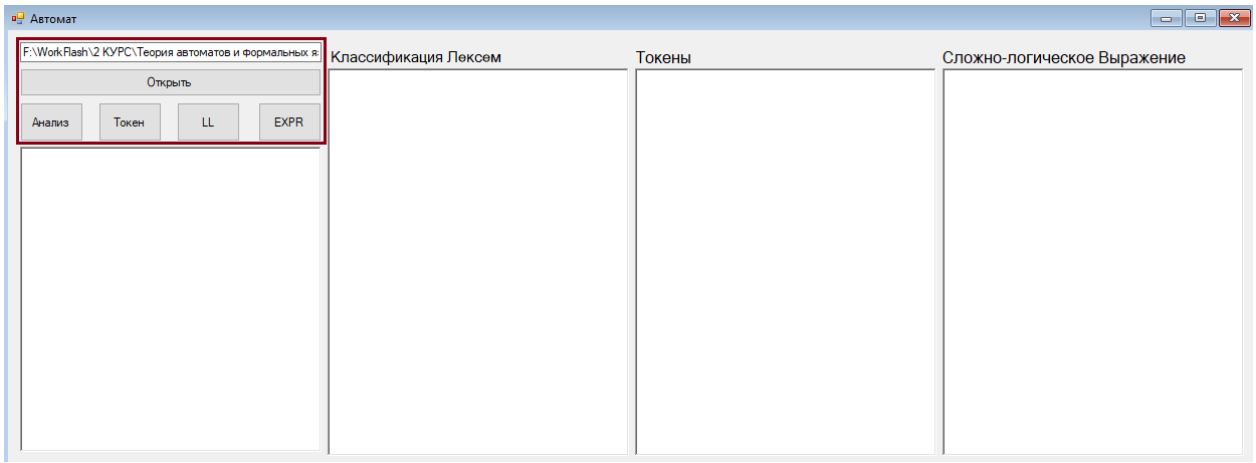

Рисунок 12 – приборная панель приложения

Для начала работы приложения потребуется указать путь к файлу и нажать кнопку “открыть” (Рисунок 13). Иначе ввести данные напрямую в поле (Рисунок 14).

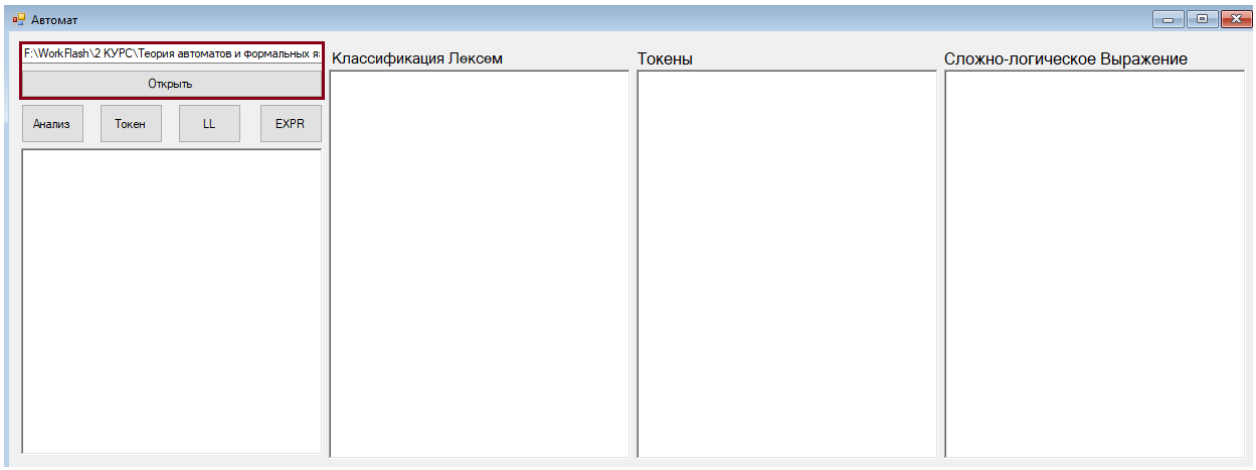

Рисунок 13 – Способ ввода данных в приложение

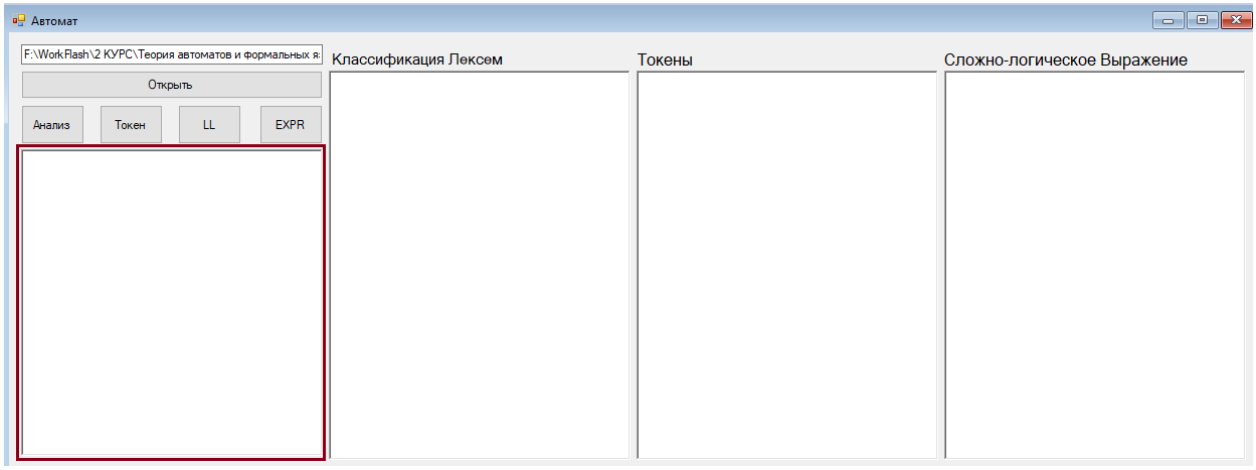

Рисунок 14 – Способ ввода данных в приложение

После успешного ввода данных мы можем провести лексический анализ кода по указанной кнопке “Анализ”, результат будет выведен в поле “Классификация Лексем” (Рисунок 15)

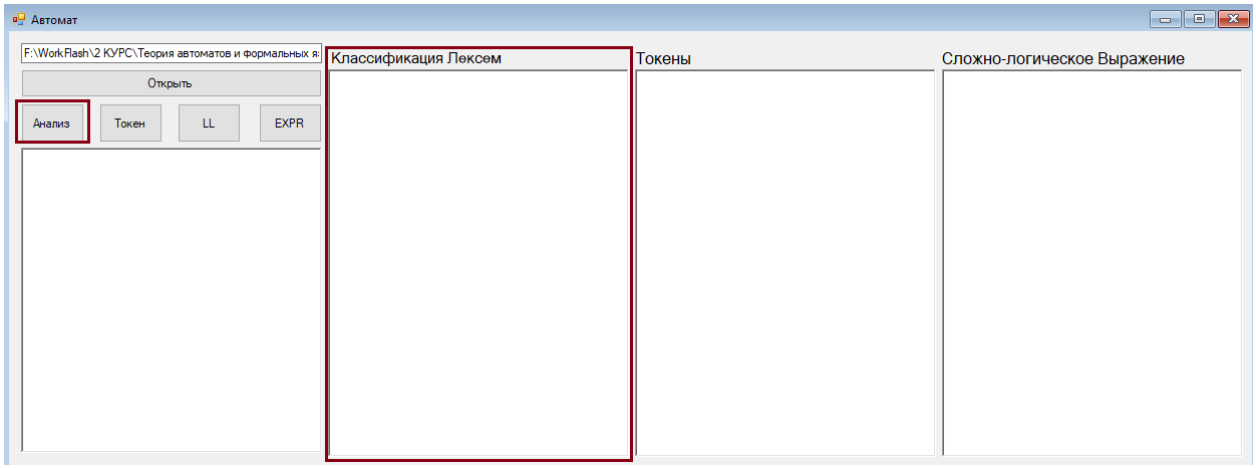

Рисунок 15 – Использования анализа лексем

Аналогично прошлому действию мы можем получить токены из ведённых данных нажав на кнопку “Токен” (Рисунок 16)

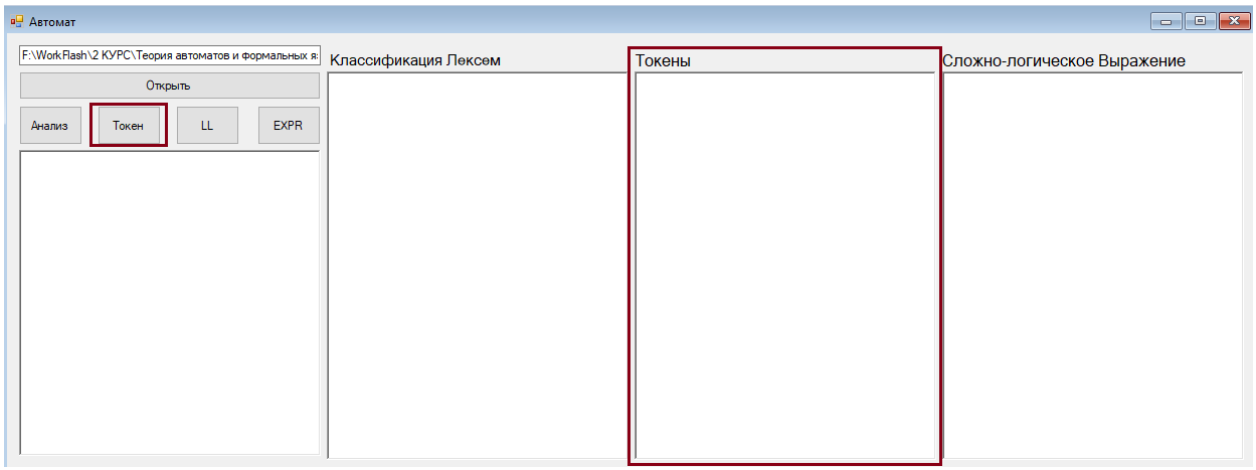

Рисунок 16 – Токенизация

Для запуска работы нисходящего анализатора следует нажать на кнопку “LL” (Рисунок 17), в случае успешной обработки будет выведено окно об успешном завершении работы нисходящего анализатора (Рисунок 18). В противном случае программа укажет неверно введённый символ (Рисунок 19).

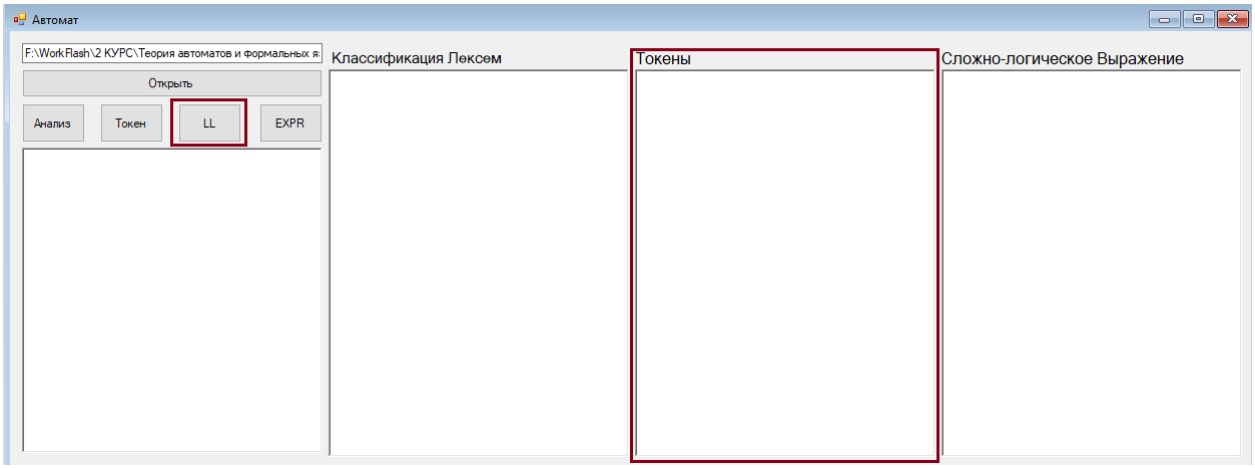

Рисунок 17 – Кнопка для запуска нисходящего анализатора и поле вывода токенов

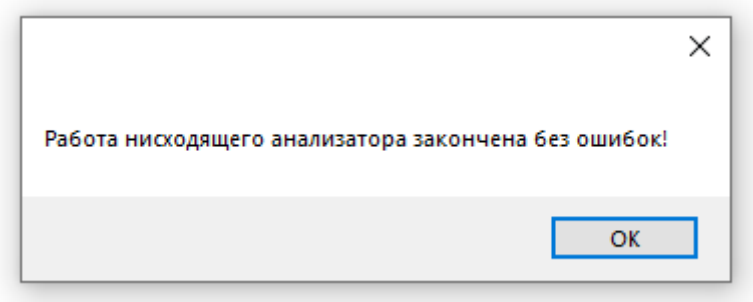

Рисунок 18 – Успешное окончание работы нисходящего анализатора.

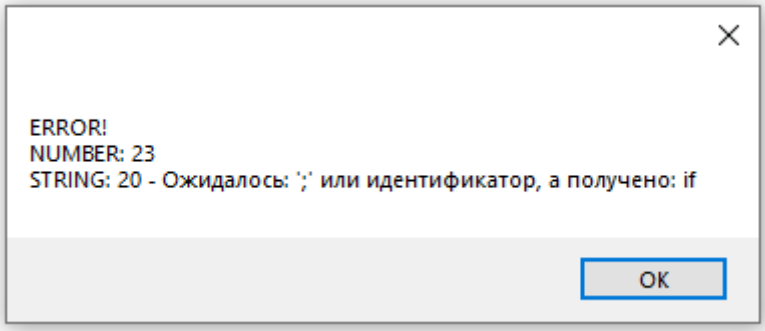

Рисунок 19 – Работа программы приостановлена из-за неправильных исходных данных.

Последняя кнопка “EXPR” нужна для запуска разбора сложно-логического выражения (Рисунок 20)

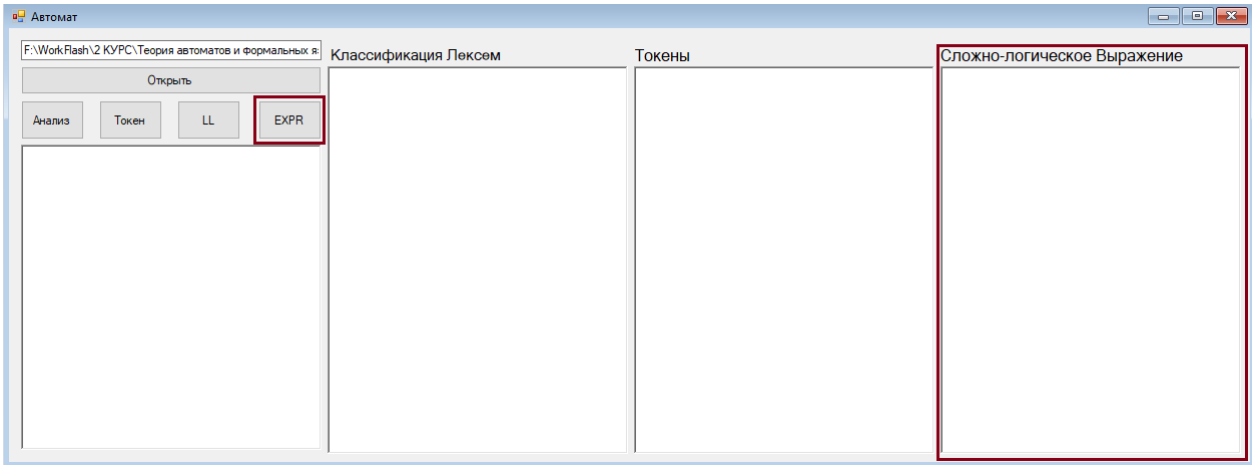

Рисунок 20 - Кнопка для запуска сложно-логического выражения и его поле вывода

6. Руководство программиста

Для транслятора входными данными является текст программы, представленный пользователем с использованием синтаксиса подмножества языка программирования С.

После выполнения лексического анализа, выходными данными транслятора являются таблица лексического разбора программы и список ключевых слов, разделителей, идентификаторов и литералов, используемых в программе.

Если при выполнении одного из анализаторов будет обнаружена ошибка, то программа выведет сообщение об этом синтаксическим анализатором.

Далее приведены основные методы, классы программы и их функциональность:

Класс Recognizer.cs включает в себя такие методы как: Recognizer, Start, Program, Next, Error, Description, xType, ListVariables, U, T, X, Y, Operator, Assignment, If, Sign, BlockOfOperations, AdditionalDescription, SkipEnter, HardOperand.

Параметры:

public List<Troyka> operatsii = new List<Troyka>(); - лист для хранения двух и более операндов.

Stack<Token> lexemStack = new Stack<Token>(); - стек для хранения лексем, используется в expr.

Public List<Token> tokens; - лист для хранения всех токенов.

Все вышеупомянутые методы данного класса реализовывают работы нисходящего анализатора. Каждый метод представляет собой определенное грамматическое правило.

Recognizer – конструктор класса Recognizer, принимающий List<token> tokens.

Start() – начальный метод запускающий работу класса. Вызывает метод Program() – метод, представляющий собой особое правило грамматики (табл.2) проверяет на верность корректного токена из List<token> tokens на соответствие токенам: MAIN, LPAR, RPAR, CURLYBRACEOPEN. Так же вызывает методы ListDescription() и ListOfOperators() после них проверяется на наличие токена CURLYBRACECLOSE

ListDescription() (спис_опис) - метод, представляющий собой особое правило грамматики (табл.2). Вызывает методы Description() (опис) и AdditionalDescription() (доп_опис).

Description() (опис) - метод, представляющий собой особое правило грамматики (табл.2). Вызывает методы: xType() (тип) и ListVariables() (спис_перем).

xType() (тип) - метод, представляющий собой особое правило грамматики (табл.2). Проверяет на соответствие конкретного токена из

List<token> tokens на соответствие её токену INT, BOOL, STRING.

ListVariables() (спис_перем) - метод, представляющий собой особое правило грамматики (табл.2). Проверяет текущий токен на VARIABLE и вызывает метод U()

U() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет на соответствие последовательности токенов VARIABLE, COMMA, VARIABLE, SEMICOLON.

Или VARIABLE, SEMICOLON.

И если в конце каждой последовательности обнаруживается токен INT, то вызывается повторный метод Description() для проверки повторной инициализации переменных.

ListOfOperators() (спис_опер) - метод, представляющий собой особое правило грамматики (табл.2). Вызывает методы Operator(), T().

Operator() (опер) - метод, представляющий собой особое правило грамматики (табл.2). Проверяет текущий токен на соответствие токенам VARIABLE или IF. Если токен соответствует токену VARIABLE, то вызывается метод Assignment(). Если токен соответствует токену IF, то вызывается метод If().

Assignment() (присв) - метод, представляющий собой особое правило грамматики (табл.2). Проверяет последовательность двух токенов на: VARIABLE, EQUEL. Вызывает метод X().

X() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет последовательность токенов на соответствие LITERAL или VARIABLE, PLUS, SEMICOLON.

Или если LITERAL или VARIABLE, SEMICOLON.

Если в конце каждой последовательности встречается: VARIABLE и следующий токен EQUAL, то вызывается метод Assignment() для повторной проверки присвоения.

If() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет последовательность двух токенов на соответствие

токенам: IF, LPAR. Вызывает метод ExpressionComplexLogical(), BlockOfOperations(), Y().

ExpressionComplexLogical() (expr) - метод, представляющий собой особое правило грамматики (табл.2). Вызывает методы: Operand() Sign() Operand(). Если после работы метода встречается токен VARIABLE – рекурсия, для повторной проверки expr.

Operand() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет последовательность двух токенов на соответствие токенам: VARIABLE, LITERAL.

Sign() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет текущий токен из List<token> tokens на соответствие токенам либо: PLUS, MINUS, MULTIPLY, MORE, LESS, DIVISION.

BlockOfOperations() – метод, представляющий собой особое правило грамматики (табл.2). Вызывает методы Operand(), ListOfOperators().

Y() (else) – метод, представляющий собой особое правило грамматики (табл.2). Проверяет последовательность токенов на соответствие: ELSE, CURLYBRACEOPEN. Вызывает метод BlockOfOperations(). После работы метода проверяет текущий токен на CURLYBRACECLOSE.

T() - метод, представляющий собой особое правило грамматики (табл.2). Проверяет текущий токен на SEMICOLON.

AdditionalDescription() - метод, представляющий собой особое правило грамматики (табл.2). Если токен соответствует MAIN, VARIABLE, IF – ошибка.

Next() – метод, который не принимает и не возвращает значения. Используется для коллекции List<token> tokens, делает шаг на +1 в списке.

SkipEnter() - метод, который не принимает и не возвращает значения. Если встречается токен ENTER – пропускает токен.

Класс Lexems.cs содержит следующие методы:
IsIDperator, IsSeparator, IsLiteral, IsIDVariable. Данный класс используется для проведения лексического анализа.

IsIDperator() – метод возвращающий значение bool, содержащий в себе идентификаторы лексем.

IsSeparator() - метод возвращающий значение bool, содержащий в себе знаки лексем.

IsLiteral() – метод возвращающий значение bool, проверяет текущую лексему на цифру.

IsIDVariable() - метод возвращающий значение bool, проверяет текущую лексему на переменную.

IsOperator() – метод возвращающий значение bool, содержащий в себе перечисление идентификаторов.

Класс Token.cs хранит две структуры данных, содержащих токены для классификации лексем (ключевые слова и разделители).

Параметры:

public string Value; - переменная в которую передаются значение токена (“SEMICOLON” = “ ; ”)

enum TokenType - содержит перечисление всех существующих токенов

public List<Troyka> operatsii - класс для восходящего синтаксического анализа.

Dictionary <string, TokenType> SpecialWords - структура данных содержащая в себе идентификаторы.

Dictionary<char, TokenType> SpecialSymbols - структура данных содержащая в себе символы.

IsSpecialSymbol(char ch) – метод принимающий символ char и возвращающий bool значение. Проверяет на соответствие лексемы токену находящийся в словаре SpecialSymbols.

IsSpecialWord(string word) - метод принимающий значение string и возвращает bool значение. Проверяет входящее значение на соответствие токену в словаре SpecialWords.

Класс Troyka – используется для трансляции логических выражений

public struct Troyka - Структура для хранения логических операций.

public Token operand1; - Поле, содержащее токен, соответствующий первому операнду логической операции.

public Token operand2; - Поле, содержащее токен, соответствующий второму операнду логической операции.

public Token deystvie; - Поле, содержащее токен, соответствующий оператору логической операции.

public Troyka(Token dy, Token op2, Token op1) - Конструктор структуры Troyka.

Параметры

dy Token - Поле, содержащее токен, соответствующий оператору логической операции.

op2 Token - Поле, содержащее токен, соответствующий второму операнду логической операции.

op1 Token - Поле, содержащее токен, соответствующий первому операнду логической операции.

public class Bower - Класс для разбора сложных логических выражений.

private int index - Целочисленное значение, содержит номер текущей операции сложного логического выражения.

public List<Troyka> troyka - Список операций сложного логического выражения.

private List<Token> tokens - Список токенов для разбора.

private Stack<Token> E - Стек для хранения операндов

private Stack<Token> T - Стек для хранения операторов логических операций

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

public Bower(List<Token> inmet) - Конструктор класса Bower.

Параметры

inmet List<Token> - Список токенов для разбора.

public Bower(List<Token> inmet, int index) - Параметры

inmet List<Token> - Список токенов для разбора.

index int - Целочисленное значение, содержит номер текущей операции сложного логического выражения.

public int Lastindex { get { return index; } } - Свойство, возвращающее номер текущей операции сложного логического 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 - Токен из списка токенов.

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 Deystv() - Метод, реализующий извлечение двух элементов из стека операндов и элемента из стека операторов логических операций, запись этих элементов в структуру, которая в последствии отправлена в список операций сложного логического выражения.

public void Start() - Метод запуска разбора сложного логического выражения.

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

private void D2() - Метод, реализующий вызов метода Deystv, затем запись анализируемого токена на вершину стека для хранения операторов

логических операций, затем переход к следующему токену для дальнейшего анализа.

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

private void D4() - Метод, реализующий вызов метода Deystv.

private void D5() - Метод, реализующий исключение Exception: Ошибка в выражении.

Конец разбора

private void PlusMinusOr() - Метод, реализующий вызов метода в соответствии со столбцом + - || таблицы 5.

Метод, реализующий вызов метода в соответствии со столбцом * / && (таблицы 3).

private void MoreLessEqual() - Метод, реализующий вызов метода в соответствии со столбцом < > = <=>= <> таблицы 3.

private void Lpar() - Метод, реализующий вызов метода в соответствии со столбцом (таблицы 5).

private void Rpar() - Метод, реализующий вызов метода в соответствии со столбцом (таблицы 5).

private void EndList() - Метод, реализующий вызов метода в соответствии со столбцом $ таблицы 5. Логика метода D6 реализована в методе Start, в связи с чем отдельный метод D6 отсутствует.

Класс Analysis – используется для лексического анализа.

Dictionary<string, string> parts = new Dictionary<string, string>() – хранит в себе лексему и описание этой лексемы

public static void PathToFile(TextBox box1, RichTextBox box2) – метод, принимающий файл и записывает в richTextBox1 содержание файла.

public void Gate(RichTextBox box1, RichTextBox box2) – метод, производящий лексический анализ. Берёт текст из richTextBox1 и разбивает его на лексемы.

Параметры

string subText – вспомогательная переменная, в которую кладутся символы лексем.

Класс AnalysisToken – используется для “токенизации” входящего кода.

public static void PathToFile(TextBox box1, RichTextBox box2) – метод, принимающий файл и записывает в richTextBox1 содержание файла.

public void ReWork(RichTextBox box2, RichTextBox box3, RichTextBox box4) – метод, производящий анализ кода, придаёт лексемам идентификаторы и по идентификаторам присваивает токены.

listBuf = new List<string>(); - структура данных, в которую кладутся лексемы

forToken = new List<string>(); - структура данных, в которую кладется идентификатор лексемы.

forChar = new List<char>(); - структура данных, в которую кладётся идентификатор лексемы разделителя.

Заключение

В результате выполнения данной курсовой работы были получены практические и теоретические навыки работы с автоматами и формальными языками.

Разработанное приложение полностью соответствует требованиям, установленным в техническом задании. Создание проекта проводилось на объектно-ориентированном языке программирования C# в Microsoft Visual Studio.

Интерфейс приложения был создан с использованием компонентов Windows Forms, таких как richTextBox, Button, label, TextBox. В ходе работы над приложением использовалась различная литература для решения проблем, возникших при разработке курсового проекта.

Приложение не занимает много места и не требовательно к системным требованиям оборудования.

Ссылка на GitHub проекта : https://github.com/E-Yokou/Course-work-1

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 с.

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 с.

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