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)

Зубова И.О.
(фамилия, инициалы)

(подпись) (дата)

Муром 2023

В данной курсовой работе был разработан транслятор с подмножества языка С. В ходе выполнения курсовой работы произведен анализ и сбор требований к разрабатываемому транслятору. Синтаксический разбор выполнен на основе LR(k)-грамматик; Разбор выражений выполнен методом Дейкстры. В качестве конструкции языка используется условный оператор if. Реализован класс транслятора.

In this course work, a translator was developed from a subset of the C language. In the course of the course work, an analysis and collection of requirements for the translator being developed was made. Syntactic parsing is based on LR(k)-grammars; The parsing of expressions was performed by the Dijkstra’s method. The conditional if statement is used as a language construct. Translator class implemented.

Содержание

Введение

Появление компьютеров требовало создания языков программирования для взаимодействия с ними. Одним из наиболее широко используемых языков программирования является C, который будет рассмотрен в этой курсовой работе. Для того, чтобы программы были корректно написаны и затем верно распознаны и интерпретированы, используются специальные методы их анализа и преобразования, которые основаны на теории языков и формальных грамматик, а также теории автоматов. Системы, которые используются для анализа и интерпретации программных текстов, называются трансляторами.

Целью курсовой работы является разработка транслятора с подмножества языка C.

Для выполнения цели курсовой работы, были выделены следующие задачи:

− Ознакомление с математическим аппаратом: формальными грамматиками, используемыми для описания искусственных языков;

− Проектирование искусственного языка;

− Формальное описание искусственного языка;

− Отладка лексического и синтаксического анализаторов, входящих в состав проектируемого транслятора;

− Разработка семантических программ транслятора;

− Генерация ассемблерного файла;

− Комплексная отладка транслятора на контрольных (тестовых) примерах.

1. Анализ технического задания

Для начала, нужно понять, что такое транслятор. Транслятор - это программа или инструмент, который выполняет перевод программы из одного языка программирования в другой. Он также выполняет диагностику ошибок, создает словарь идентификаторов и выдает текст программы для печати и т.д.

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 LR(k)-грамматик;

– Разбор логических выражений выполнять методом Дейкстры;

В языке поддерживаются

– у идентификатора 8 символов значащие;

– не менее 3х директив описания переменных;

– простой арифметический оператор;

– сложное логическое выражение;

– условный оператор if (…){…}else{…}

Для реализации транслятора будет выбран современный язык программирования C#, который является объектно-ориентированным и типобезопасным языком и позволяет разработчикам создавать безопасные и надежные приложения, работающие в .NET. Для создания транслятора будет использована среда разработки Visual Studio Community Edition 2022, которая позволяет создавать не только консольные приложения, но и приложения с графическим интерфейсом, включая поддержку технологии Windows Forms.

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 Дейкстры 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.

Рассматриваемое подмножество языка си включает в себя следующие операторы: объявление переменных, присвоение значений, арифметический оператор, логический оператор, условный оператор.

Рассмотрим синтаксис каждого из них:

- Объявление переменных: int a,b; int c;
- Присвоение значений: b=0; a=5; c=a; c=b+5; b=a;
- Арифметические операторы: b+5;
- Логический оператор: a>b && b<c;
- Условный оператор: if (…){…}else{…}

Для описания синтаксиса языка Си была использована форма Бэкуса-Наура. Чтобы построить соответствующую грамматику, нужно следовать следующим шагам:

1. Определяем множество терминальных символов T, которые включают в себя все ключевые слова и разделители языка.
2. Определяем общую структуру программы и начальный символ грамматики, который называется <программа>.
3. Программа в языке Си состоит из одной или более функций, каждая из которых ограничена фигурными скобками. Главной функцией в программе консольного приложения на языке Си является функция main(), которая является основным блоком.
4. Последовательно надо объяснить структуру всех нетерминалов, введенных в первом и в следующих правилах. Сама программа представляет собой множество операторов. Соответственно, необходимо ввести правила, позволяющее описать эту последовательность. Такие правила должны

содержать рекурсию, для того чтобы размножить "элемент списка" в количестве одной или более единиц.

5. Если требуется описать часть оператора, которая может принимать различные значения или может отсутствовать, то следует ввести новый нетерминал на месте этой части, а затем в следующем правиле описать ее структуру.

6. Все полученные правила формируют множество P, при этом все нетерминалы, используемые в правых частях правил, должны быть описаны как правила.

7. Набор нетерминалов образует множество N.

После выполнения всех шагов алгоритма была получена следующая грамматика:

G=T,N,P,<программа>

T={main, (,), {, int, =, ;, if, else, >, <, +, &&,}

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

P = {

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

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

Далее необходимо сравнить вывод с исходным кодом.

Таблица 1 – Сравнение полученной цепочки и анализируемого фрагмента

Анализируемый фрагмент программы	Полученная цепочка вывода

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;}
	}

Условие окончания цепочки вывода является получение сентенции, то есть строки, содержащей только символы множества терминалов.

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. 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». 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 на представлен на таблице 2 ниже:

Таблица 2 – Пример работы сканера

main	Идентификатор - оператор;
(	Разделитель;
)	Разделитель;
{	Разделитель;
int	Идентификатор - оператор;

При описании лексического анализа, важно понимать и разделять три связанных понятия.

Токен - структура, состоящая из имени и набора необязательных произвольных атрибутов. Имя токена – абстрактный символ,

представляющий тип лексической единицы, например, <ключевое слово>, <название переменной>, и т.п.

Лексема - представляет собой последовательность символов исходной программы, которая идентифицируется лексическим анализатором как экземпляр токена.

Шаблон - описание вида, который может принимать лексема токена. Лексический анализатор принимает решение о принадлежности лексемы токену на основе шаблона.

Лексический токен (или просто токен) – это строка с присвоенным и, таким образом, идентифицированным значением. Он структурирован как пара, состоящая из имени токена и необязательного значения токена. Имя токена — это категория лексической единицы.

На вход лексического анализатора поступает текст исходной программы,
а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Каждой выделенной из текста лексеме сканер ставит в соответствие токен вида: ‹имя_токена, значение_атрибута›.
<имя_токена>, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Токенизация — это процесс классификации разделов строки входных символов. Полученные токены затем передаются на следующий этап компиляции, то есть в синтаксический анализатор.

Итогом работы лексического анализатора должен быть список лексем, в котором каждой сопоставлен ее конкретный тип. Далее эта информация будет использована синтаксическим анализатором для проверки принадлежности грамматике. Пример работы лексического анализатора приведён в таблице 3.

Таблица 3 – Пример работы лексического анализатора

MAIN	main
LPAR	(
RPAR	)
CURLYBRACEOPEN	{
INT	int

Синтаксическим анализом (разбором) называется процесс, определяющий порождается ли данная строка лексем данной грамматикой.

Синтаксический анализатор может быть построен для любой грамматики. Для любой контекстно свободной грамматики существует анализатор, требующий самое большее O(n3) времени для разбора строки из n лексем. Но для языков, которые встречаются на практике, хватает линейных алгоритмов. Анализаторы языков программирования почти всегда проходят один раз слева направо входной текст, считывая за раз одну лексему.

Большинство методов синтаксического разбора относятся к «восходящим» или «нисходящим» методам. В первом случае построение дерева начинается с корня и продолжается в направлении листьев, в то время как во втором – построение начинается с листьев и развивается по направлению к корню. Популярность анализаторов «сверху-вниз» объясняется тем, что эффективный анализатор можно легко построить «вручную». Анализаторы «снизу-вверх» могут работать с большим классом грамматик и схем перевода, поэтому в программах-генераторах анализаторов всё больше используются восходящие алгоритмы разбора.

Восходящий анализ в классе LR(k)-грамматик.

Восходящий синтаксический анализ для грамматики G = (T, N, P, S) начинает разбор с конкретных слов (лексем) из множества T, связывая сначала пары слов, затем подсоединяет к этим парам новые слова или другие связанные пары образуя нетерминалы из множества N. Постепенно процесс связывания доходит до начального нетерминала S - то есть все лексемы оказываются связаны в единую структуру.

Восходящие методы синтаксического анализа состоят в том, что в цепочке (промежуточной или терминальной) ищется правая часть очередного правила, которое должно быть заменено своим нетерминалом. Т.е. синтаксическое дерево строится снизу-вверх: в текущем множества «незакрытых» вершин ищется подмножество потомков и над ними «надстраивается» вершина-предок. При этом обход вершин и, аналогичный, просмотр цепочки символов происходит слева- направо.

Задача восходящего анализа и состоит в поиске "редукции" исходной терминальной цепочки в аксиому (нетерминал), где на каждом шаге производится поиск основы в текущую цепочку с последующей заменой ее соответствующим нетерминалом, т.е. применяется "инверсия" некоторого правила данной КС - грамматики: правая часть правила заменяется левой. Нетрудно понять, что если мы заменяем на каждом шаге такой "редукции" самую левую основу, то восстанавливаем с конца к началу правый вывод исходной цепочки, а если самую правую основу, то "реконструируем" левый вывод.

Во - первых, по аналогии с нисходящим разбором можно предположить, что для обнаружения основы достаточно пары символов – последнего символа основы и следующего символа строки. Т.е. для каждой пары символов грамматики однозначно можно сформулировать утверждение, является ли эта пара концом основы или нет. Опять - таки это связано с глубиной просмотра вперед входной строки – она равна 1.

Во - вторых, распознавателю необходим стек, и для него требуется определить функциональное назначение. Поскольку просмотр строки в поисках основы требует сохранения пройденных символов, резонно это делать в стеке. Тогда замена правой части правила (основы) на левую будет также производиться в вершине стека. Сам стек будет хранить «недосвернутую» просмотренную часть цепочки, для которой еще не накоплена основа.

Теперь можно сформулировать основные принципы восходящего разбора с использованием магазинного автомата, именуемого также методом «свёртка-сдвиг»:

1) Первоначально в стек помещается первый символ входной строки, а второй становится текущим;

2) Автомат выполняет два основных действия:

– сдвиг очередного символа из входной строки в стек (с переходом к следующему);

– поиск правила, правая часть которого хранится в стеке. В таком случае производится замена ее на левую – свёртка;

Решение, какое из действий – перенос или свёртка должно быть выполнено на данном шаге, принимается на основе анализа пары символов

– символа в вершине стека и очередного символа входной строки. Свёртка соответствует наличию в стеке основы, при ее отсутствии выполняется перенос. Управляющими данными автомата является решающая таблица, содержащая для каждой пары символов грамматики указание на выполняемое действие (свёртка, сдвиг или переход в другое состояние) и сами правила грамматики.

Конфликт относится к типу сдвиг-свёртка, если для одной цепочки допустимы и сдвиг и свёртка. Конфликт относится к типу свёртка-свёртка, если допустимы свёртки по различным правилам.

Если в процессе LR-разбора принять детерминированное решение о сдвиге/свертке удаётся, рассматривая только цепочку α и первые k символов не просмотренной части входной цепочки, говорят, что грамматика обладает LR(k) - свойством.

В LR(k) разборе учитывается k-первых символов не просмотренной части входной цепочки.

Положительным результатом работы автомата будет наличие начального нетерминала грамматики в стеке при пустой входной строке

Построим граф состояний автомата, решающую таблицу детерминированного автомата. На основе анализа решающей таблицы и конфликтов сделаем вывод о принадлежности грамматики к классу LR(k) и определим k.

LR(k) означает, что:

- входная цепочка обрабатывается слева направо,
- выполняется правый вывод,
- для принятия решения используется не более k символов цепочки.

Для этого необходимо:

1. Создать граф состояний;

2. Построить таблицу решений для LR-анализатора.

Для создания графа состояний используется следующий процесс:

1. В начальное состояние вносятся все правила, которые можно вывести из генерирующего символа грамматики, и перед самым левым символом в правой части ставится маркер;

2. Для каждого нетерминала, который помечен маркером, вносятся правила, которые он может генерировать, с маркером перед самым левым символом в правой части;

3. Список правил, полученных выполнением п. 1-2, называется состоянием;

4. Если маркер в правиле состояния находится после последнего символа, то в столбец "переход" записывается "нет перехода", а состояние отмечается как конечное.

5. Если в строке, соответствующей состоянию D, ячейка "правила" заполнена, а ячейка "переход" пуста, то в нее записывается номер следующего свободного состояния n. Если в этом состоянии есть другие строки с таким же маркером, то в ячейки "переход" этих строк также записывается номер n.

6. Все строки из текущего состояния D, у которых в столбце "переход" записан n и маркер в них, переносятся на один символ вправо и записываются в следующее свободное состояние n.

7. Повторяются действия 2-3, пока все ячейки столбца "Переход" не будут заполнены.

Посредством выполнения этого алгоритма был построен граф состояний:

Таблица 4 – Граф состояний

Сост	Пред	Правила	переход
0	-	<программа> ::= • main() { <спис_опис>; <спис_опер> }	1
1	0	<программа> ::= main• () { <спис_опис>; <спис_опер> }	2
2	1	<программа> ::= main (•) { <спис_опис>; <спис_опер> }	3
3	2	<программа> ::= main ()•{ <спис_опис>; <спис_опер> }	4
4	3	<программа> ::= main (){•<спис_опис>; <спис_опер> } <спис_опис>::=• <опис.> <опис.> ::= • <тип> <спис_перем.> <тип> ::= • int <тип> ::= • bool <тип> ::= • string <спис_опис.> ::= • <спис_опис>; <опис.>	5 6 7 8 9 10 5
5	4	<программа> ::= main (){<спис_опис>•; <спис_опер> } <спис_опис.> ::= <спис_опис>•; <опис.>	11 11
6	4	<спис_опис>::=<опис.> •	X
7	4	<опис.> ::= <тип> •<спис_перем.> <спис_перем.> ::= •id <спис_перем.> ::= • <спис_перем.>, id	12 13 12
8	4	<тип> ::= int •	X
9	4	<тип> ::= bool •	x
10	4	<тип> ::= string•	x
11	5	<программа> ::= main (){<спис_опис>;• <спис_опер> } <спис_опис.> ::= <спис_опис>;• <опис.> <спис_опер.> ::= • <опер> <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд> <спис_опер.> ::= •<спис_опер.>; <опер.> <опис.> ::= • <тип> <спис_перем.> <тип> ::= • int <тип> ::= • bool <тип> ::= • string	14 15 16 17 18 18 19 20 20 14 7 8 9 10
12	7	<опис.> ::= <тип> <спис_перем.> • <спис_перем.> ::= <спис_перем.> •, id	X 21
13	7	<спис_перем.> ::= id•	x
14	11	<программа> ::= main (){<спис_опис>; <спис_опер>•} <спис_опер.> ::= <спис_опер.> •; <опер.>	26 22
15	11	<спис_опис.> ::= <спис_опис>; <опис.> •	x
16	11,30	<спис_опер.> ::= <опер> •	x

17	11,22,23,30,34	<опер.> ::= <условн.> •	x
18	11,22,23,30,34	<условн.> ::= if •expr <блок. опер.> <условн.> ::= if •expr <блок_опер.> else <блок_опер.>	23 23
19	11,22,23,30,34	<опер.> ::= <присв.> •	x
20	11,22,23,30,34	<присв.> ::= id •= <операнд> <знак> <операнд> <присв.> ::= id• = <операнд>	24 24
21	12	<спис_перем.> ::= <спис_перем.>, •id	25
22	14	<спис_опер.> ::= <спис_опер.>; • <опер.> <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд>	27 17 18 18 19 20 20
23	18	<условн.> ::= if expr •<блок. опер.> <условн.> ::= if expr •<блок_опер.> else <блок_опер.> <блок_опер.> ::= •<опер.>; <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд> <блок_опер.> ::= •{ <спис_опер.>; }	28 28 29 17 18 18 19 20 20 30
24	20	<присв.> ::= id = •<операнд> <знак> <операнд> <присв.> ::= id = •<операнд> <операнд> ::= • id <операнд> ::= • lit	31 31 32 33
25	21	<спис_перем.> ::= <спис_перем.>, id•	x
26	14	<программа> ::= main (){<спис_опис>; <спис_опер> }•	x
27	22	<спис_опер.> ::= <спис_опер.>; <опер.> •	x
28	23	<условн.> ::= if expr <блок. опер.> • <условн.> ::= if expr <блок_опер.> • else <блок_опер.>	X 34
29	23,34	<блок_опер.> ::= <опер.> •;	35
30	23,34	<блок_опер.> ::= { •<спис_опер.>; } <спис_опер.> ::= • <опер> <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд> <спис_опер.> ::= •<спис_опер.>; <опер.>	36 16 17 18 18 19 20 20 36
31	24	<присв.> ::= id = <операнд>• <знак> <операнд> <присв.> ::= id = <операнд>• <знак> ::= •+ <знак> ::= • - <знак> ::= •/ <знак> ::= • *	37 X 38 39 40 41
32	24,37	<операнд> ::= id •	x
33	24,37	<операнд> ::= lit•	x
34	28	<условн.> ::= if expr <блок_опер.> else •<блок_опер.>	42

		<блок_опер.> ::= •<опер.>; <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд> <блок_опер.> ::= •{ <спис_опер.>; }	29 17 18 18 19 20 20 30
35	29	<блок_опер.> ::= <опер.>; •	x
36	30	<блок_опер.> ::= { <спис_опер.> •; } <спис_опер.> ::= <спис_опер.> •; <опер.>	43 43
37	31	<присв.> ::= id = <операнд><знак>•<операнд> <операнд> ::= • id <операнд> ::= • lit	44 32 33
38	31	<знак> ::= +•	x
39	31	<знак> ::= - •	x
40	31	<знак> ::= /•	x
41	31	<знак> ::= *•	x
42	34	<условн.> ::= if expr <блок_опер.> else <блок_опер.> •	X
43	36	<блок_опер.> ::= { <спис_опер.>; • } <спис_опер.> ::= <спис_опер.>; • <опер.> <опер.> ::= • <условн.> <условн.> ::= •if expr <блок. опер.> <условн.> ::= • if expr <блок_опер.> else <блок_опер.> <опер.> ::= • <присв.> <присв.> ::= • id = <операнд> <знак> <операнд> <присв.> ::= • id = <операнд>	45 27 17 18 18 19 20 20
44	37	<присв.> ::= id = <операнд><знак><операнд>•	x
45	43	<блок_опер.> ::= { <спис_опер.>; }•	x

Следующим шагом в разработке восходящего анализатора является создание решающей таблицы. Это делается на основе анализа строк, связанных с состоянием с тем же именем. Чтобы создать таблицу принятия решений, нужно следовать следующему алгоритму:

1. В колонке «Стек разбора» для начального состояния должен быть указан порождающий символ грамматики с действием «КОНЕЦ», а также строка «ε» с действием «СДВИГ».

2. Все строки, связанные с анализируемым состоянием, просматриваются. В столбец «Стек разбора» записывается элемент, находящийся под маркером, а в столбец «Действие» записывается базовое действие – «СДВИГ» или «СВЕРТКА». Если все строки являются

переходами, то базовым действием будет «СДВИГ», а если все строки помечены как конечные, то базовым действием будет «СВЕРТКА».

3. Если в состоянии присутствуют и строки с переходами, и строки с признаком «нет перехода», то в этом состоянии должны быть указан оба действия – «СДВИГ» и «СВЕРТКА».

На основании вышеприведенного алгоритма была построена решающая таблица:

Таблица 5 – Решающая таблица восходящего анализатора

Сост	Стек	Вход	Действие
0	<программа> E main		Конец Сдвиг s1
1	(		S2
2	)		S3
3	{		S4
4	<спис_опис> <опис.> <тип> int bool string		S5 S6 S7 S8 S9 S10
5	;		S11
6	<опис.>		Свёртка(-1,< спис_опис >)
7	<спис_перем.> id		s12 s13
8	int		Свёртка(-1,<тип>)
9	bool		Свёртка(-1,<тип>)
10	string		Свёртка(-1,<тип>)
11	<спис_опер> <опис.> <опер> <условн.> If <присв.> Id <тип> Int Bool string		S14 S15 S16 S17 S18 S19 S20 S7 S8 S9 S10
12	<спис_перем.> <спис_перем.> ,	; ,	Свёртка(-2, <опис.>) Сдвиг S21
13	Id		Свёртка(-1, <спис_перем.>)
14	;		S22

	}		S26
15	<опис.>		Свёртка(-3, <спис_опис.>)
16	<опер>		Свёртка(-1,<спис_опер.>)
17	<условн.>		Свёртка(-1, <опер.>)
18	Expr		S23
19	<присв.>		Свёртка(-1, <опер.>)
20	=		S24
21	id		s25
22	} <опер.> <условн.> If <присв.> id		S26 S27 S17 S18 S19 S20
23	<блок. опер.> <опер.> <условн.> If <присв.> Id {		S28 S29 S17 S18 S19 S20 S30
24	<операнд> Id lit		S31 S32 S33
25	Id		Свёртка(-3, <спис_перем.>)
26	}		Свёртка(-8, <программа>)
27	<опер.>		Свёртка(-3, <спис_опер.>)
28	<блок. опер.> <блок. опер.> else	; else	Свёртка(-3,<условн.>) Сдвиг S34
29	;		S35
30	<спис_опер.> <опер> <условн.> If <присв.> Id		S36 S16 S17 S18 S19 S20
31	<операнд> <операнд> <знак> + - / *	+,-,/,* ;	Сдвиг Свёртка(-3,<присв.>) S37 S38 S39 S40 S41
32	Id		Свёртка(-1, <операнд>)
33	Lit		Свёртка(-1, <операнд>)
34	<блок_опер.> <опер.> <условн.> If <присв.> Id		S42 S29 S17 S18 S19 S20

	{		S30
35	;		Свёртка(-2, <блок_опер.>)
36	;		S43
37	<операнд> Id lit		S44 S32 S33
38	+		Свёртка(-1, <знак>)
39	-		Свёртка(-1, <знак>)
40	/		Свёртка(-1, <знак>)
41	*		Свёртка(-1, <знак>)
42	<блок_опер.>		Свёртка(-5, <условн.>)
43	} <опер.> <условн.> If <присв.> id		S45 S27 S17 S18 S19 S20
44	<операнд>		Свёртка(-5, <присв.>)
45	}		Свёртка(-4, <присв.>)

При построении решающей таблицы было выявлено, что в некоторых состояниях необходимо просматривать следующий символ входного потока для принятия решения. Таким образом k=1, а грамматика принадлежит к классу LR(1).

Разбор сложных логических выражений осуществляется методом Дейкстры:

1. Входная строка просматривается слева направо, для каждого символа:

1.1. если символ является операндом, то он добавляется к выходной строке;

1.2. если символ является открывающейся скобкой, она помещается в стек;

1.3. если символ является закрывающейся скобкой, выталкиваются элементы из стека в выходную строку до тех пор, пока на вершине стека не окажется открывающаяся скобка.

При этом открывающаяся скобка удаляется из стека, но в выходную строку не добавляется.

Если стек закончился раньше, чем встретилась открывающая скобка – в выражении несогласованны скобки.

1.4. если символ операция, то:

- если приоритет операции равен нулю или больше приоритета операции, находящейся на вершине стека или стек пуст, то входная операция записывается в стек,

- иначе из стека выталкивается и переписывается в выходную строку операция, находящаяся на вершине, а также следующие за ней операции с приоритетами большими или равными приоритету входной операции. После этого входная операция добавляется к вершине стека.

2. После просмотра всех символов входной строки происходит выталкивание всех оставшихся в стеке операций и дописывание их к выходной строке.

Далее осуществляется перевод обратной Польской нотации в матричный вид по правилам:

1. ОПН просматривается слева направо;

2. При чтении операнда – он записывается в стек операндов;

3. При чтении операции:

• из стека извлекается два верхних операнда;

• формируется тройка (операция, операнды) и записывается в матрицу операций;

• в стек записывается обозначение строки матрицы с полученной операцией.

Программирование системы

В разрабатываемом приложении был использован язык C# и среда разработки Visual Studio 2022. Графический интерфейс приложения будет реализован с помощью фреймворка WinForms. . Программа способна анализировать фрагменты кода, введенные пользователем

через компонент RichTextBox. Для управления функциями программы будут использованы компоненты Button и их Click-события. Результаты работы программы будут отображены на форме с помощью компонента RichTextBox.

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).

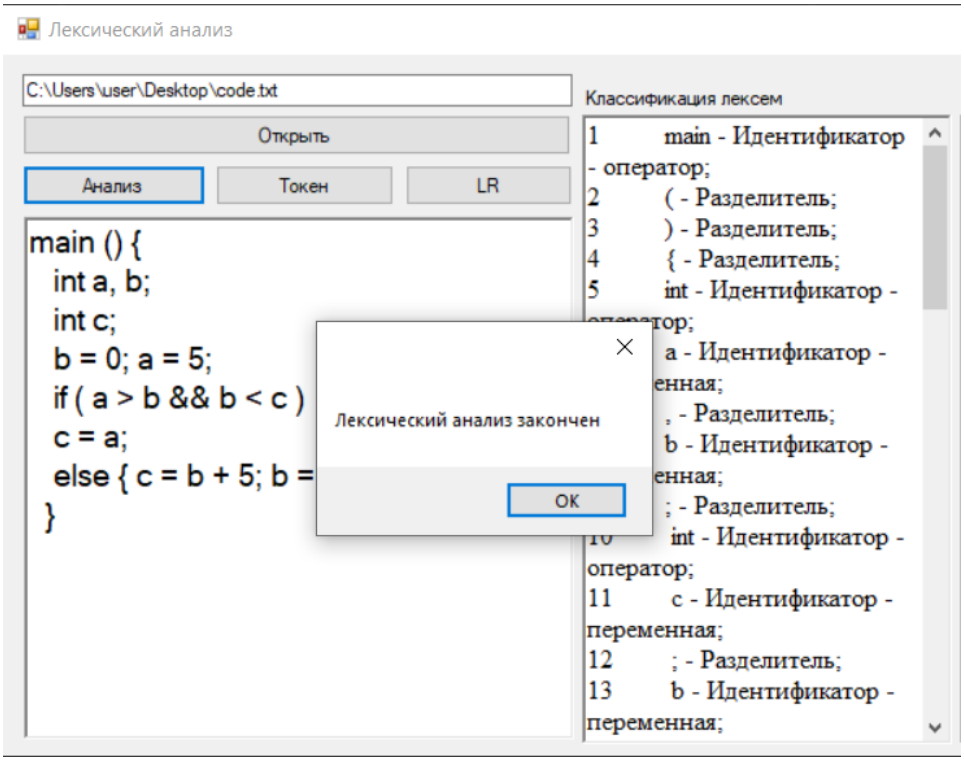

Рисунок 1 – Результат работы лексического анализатора

Проверка на некорректной лексике

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

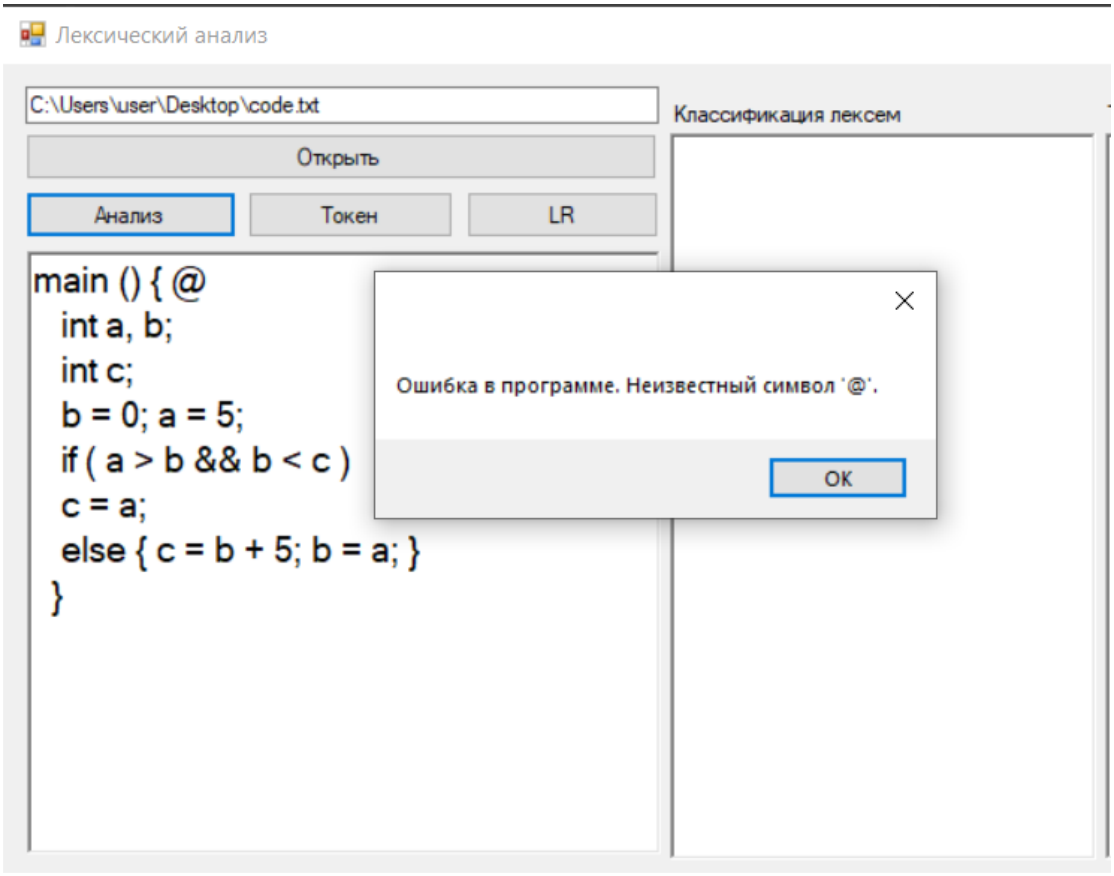

Рисунок 2 – Результат работы лексического анализатора

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 3).

Проведена проверка работы синтаксического анализатора с дополнительным объявлением переменных (Рисунок 4).

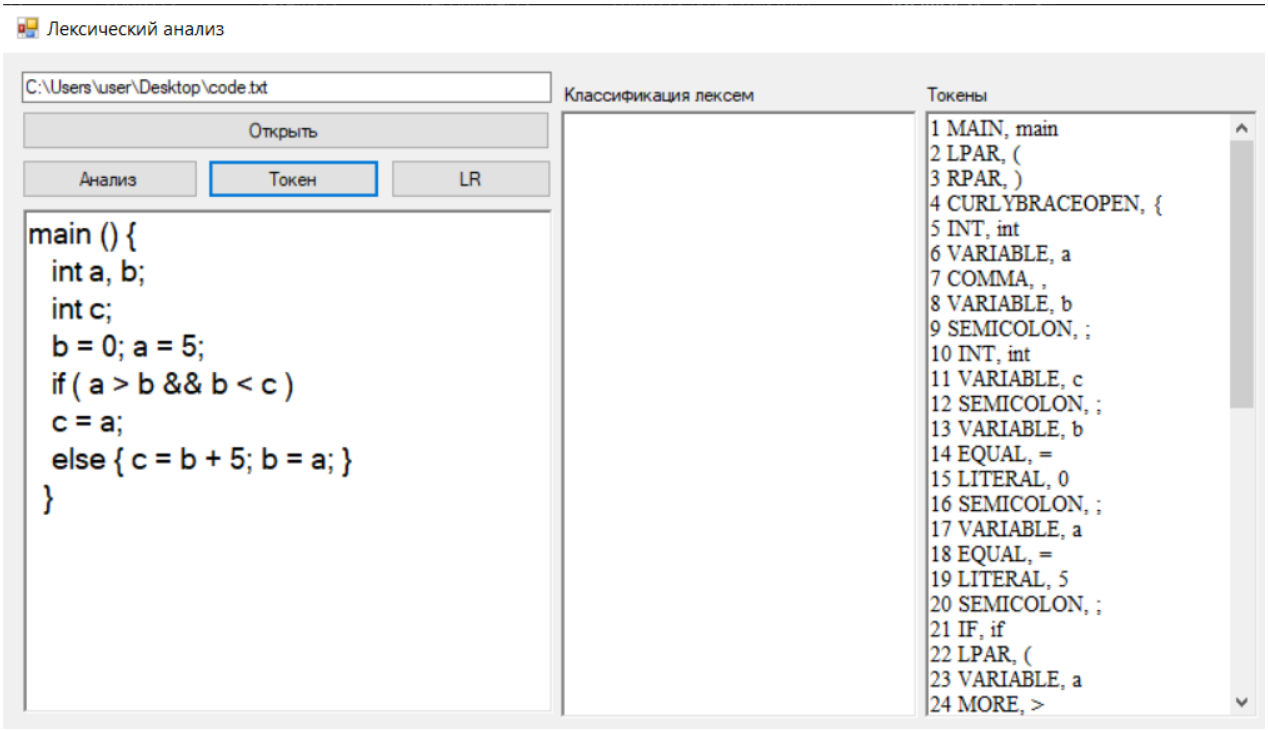

Рисунок 3 – Успешный разбор кода синтаксическим анализатором.

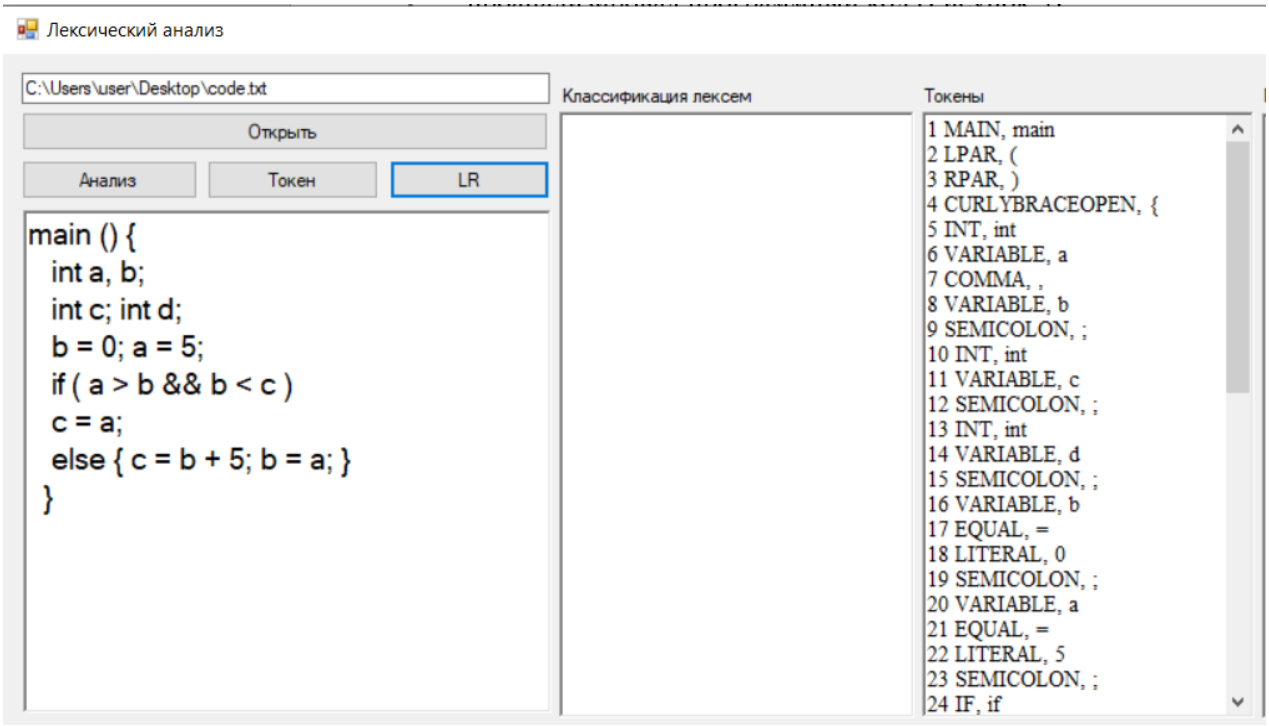

Рисунок 4 – Успешный разбор кода синтаксическим анализатором с дополнительным объявлением переменных.

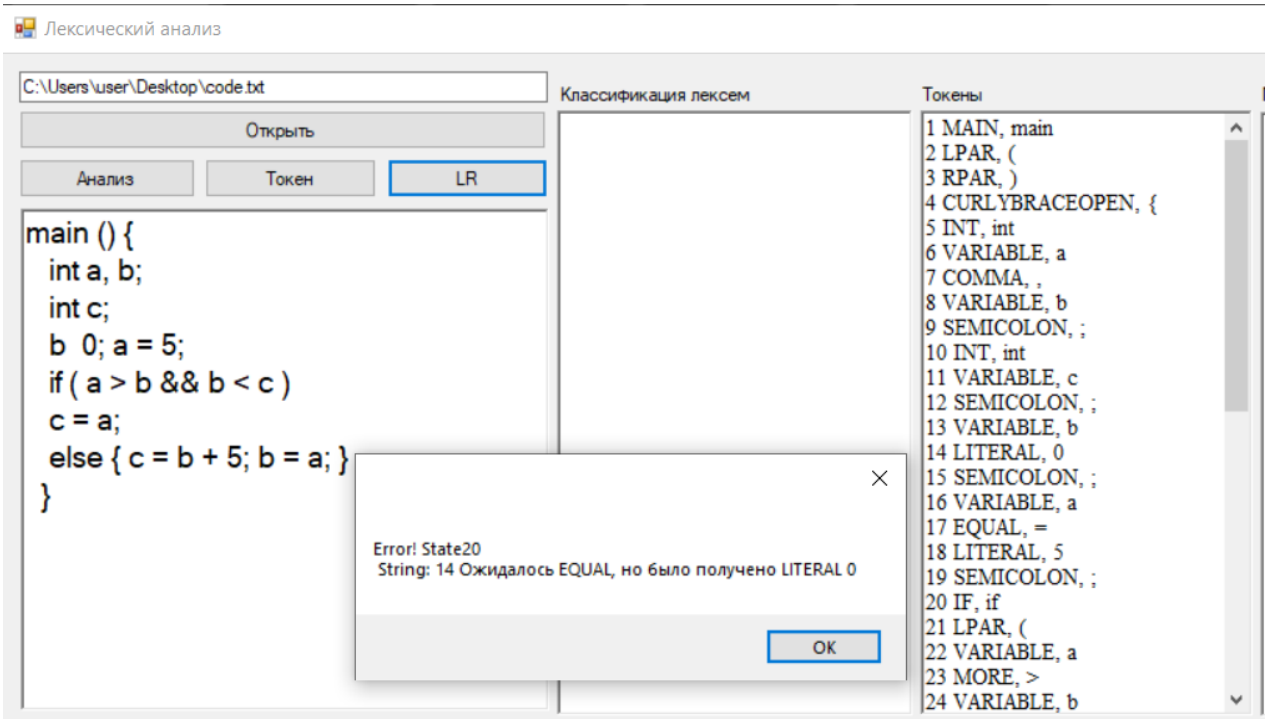

Рисунок 5 – Некорректное объявление переменных.

5. Руководство пользователя

После запуска программы будет представлено окно (Рисунок 6).

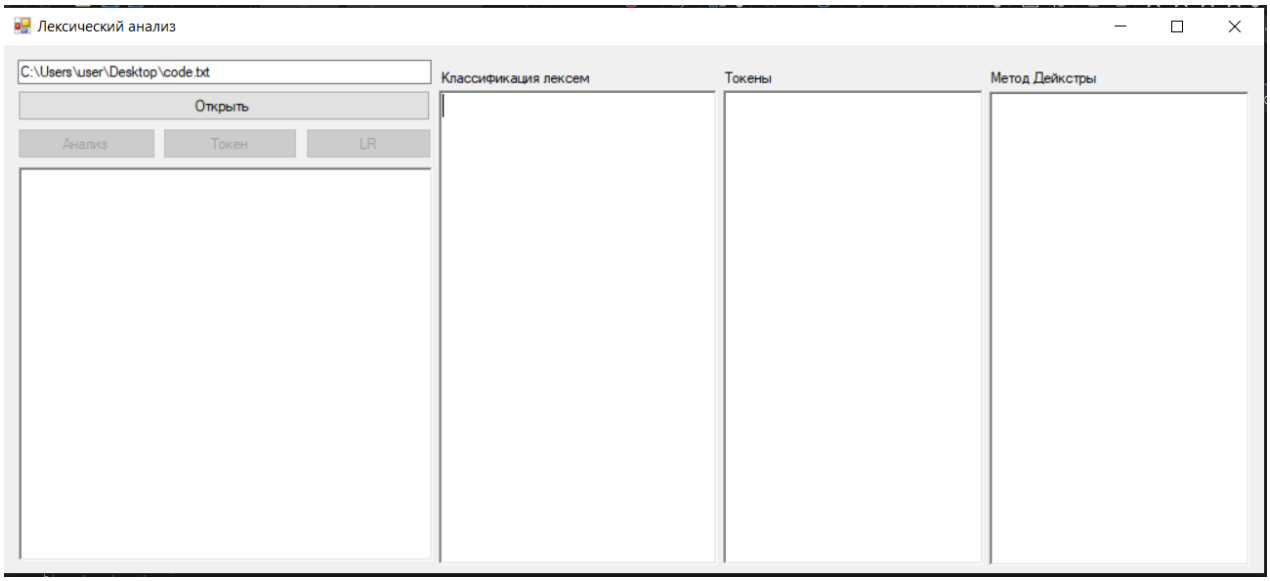

Рисунок 6 – интерфейс программы

Выделенная красным квадратом область окна – является приборной панелью данного приложения (Рисунок 7).

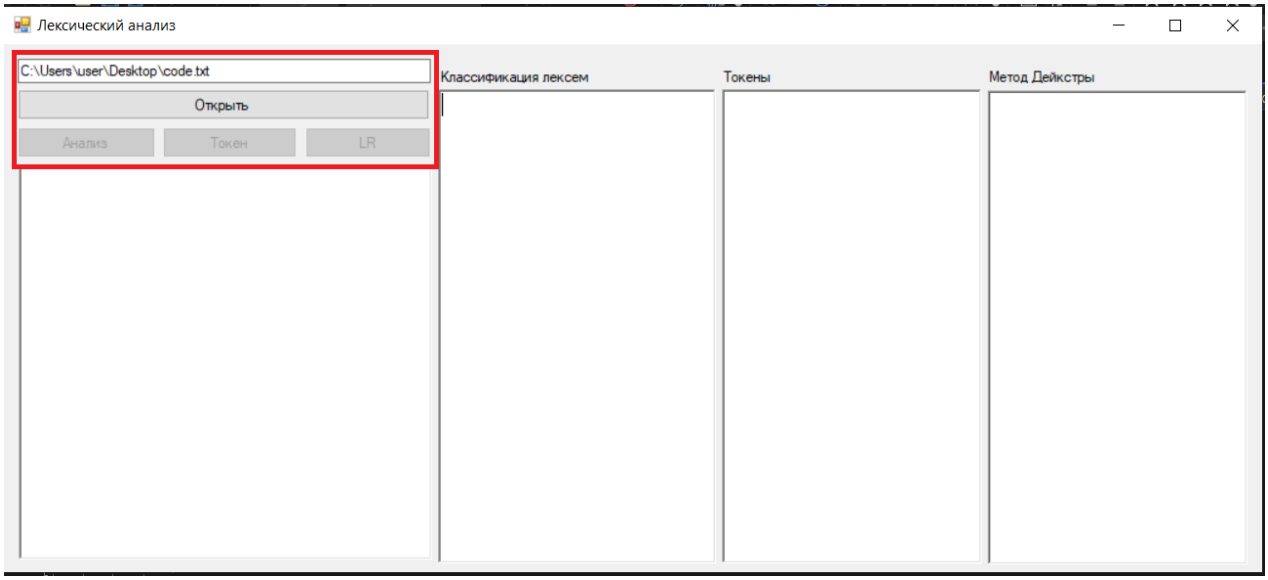

Рисунок 7 – приборная панель приложения

Для начала работы приложения потребуется указать путь к файлу и нажать кнопку “открыть” (Рисунок 8). Иначе ввести данные напрямую в поле (Рисунок 9).

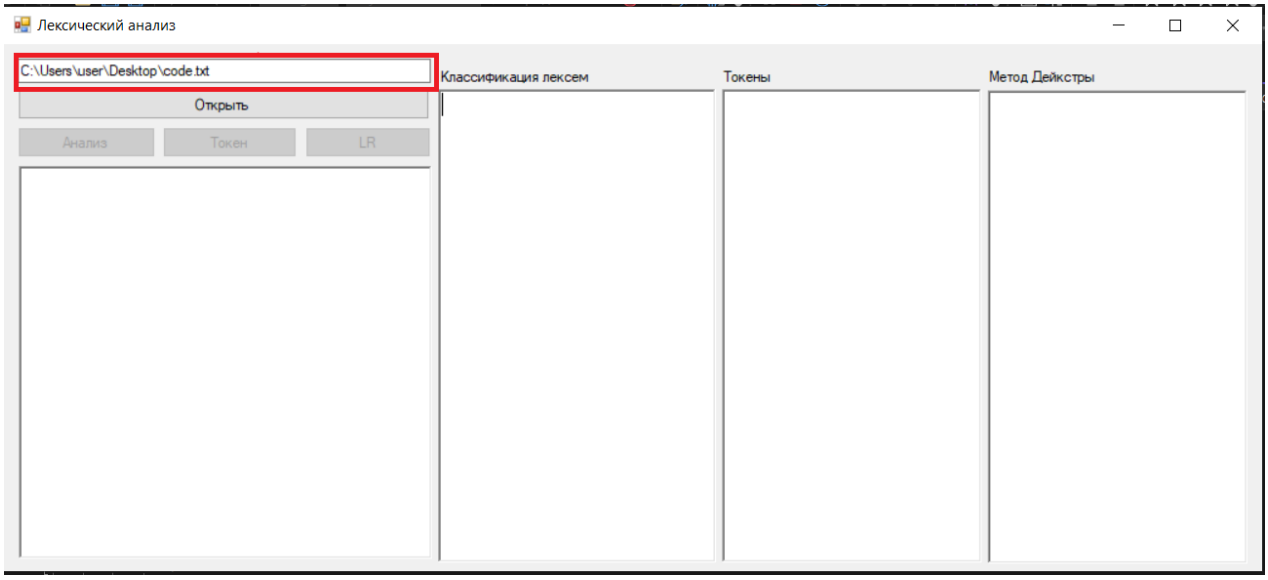

Рисунок 8 – Способ ввода данных в приложение

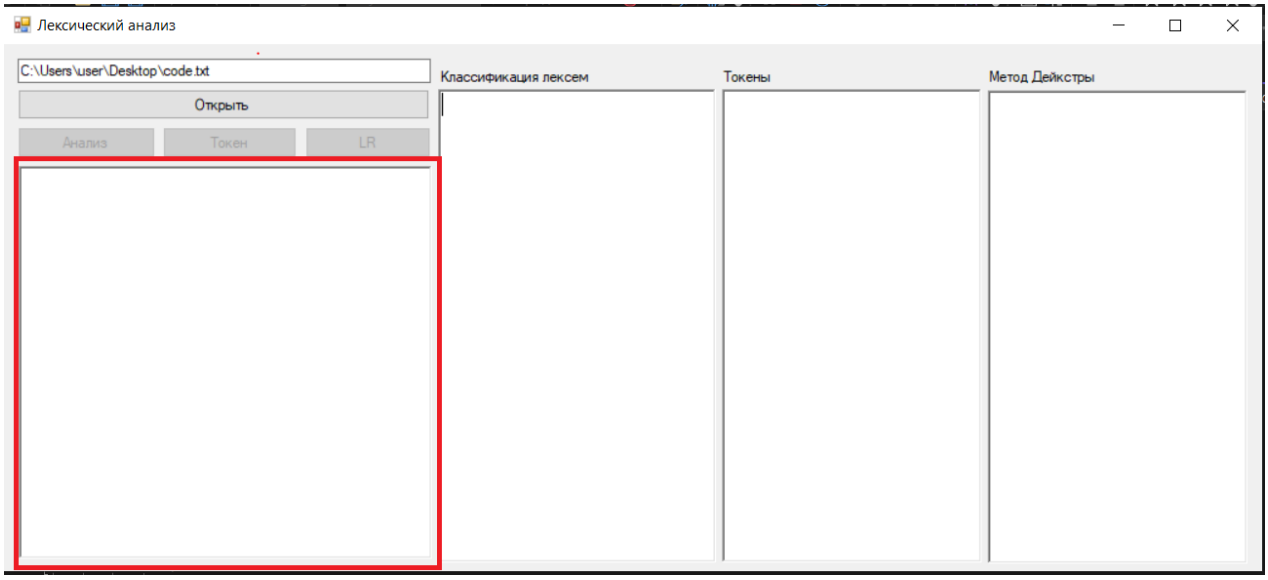

Рисунок 9 – Способ ввода данных в приложение

После успешного ввода данных мы можем провести лексический анализ кода по указанной кнопке “Анализ”, результат будет выведен в поле “Классификация Лексем” (Рисунок 10)

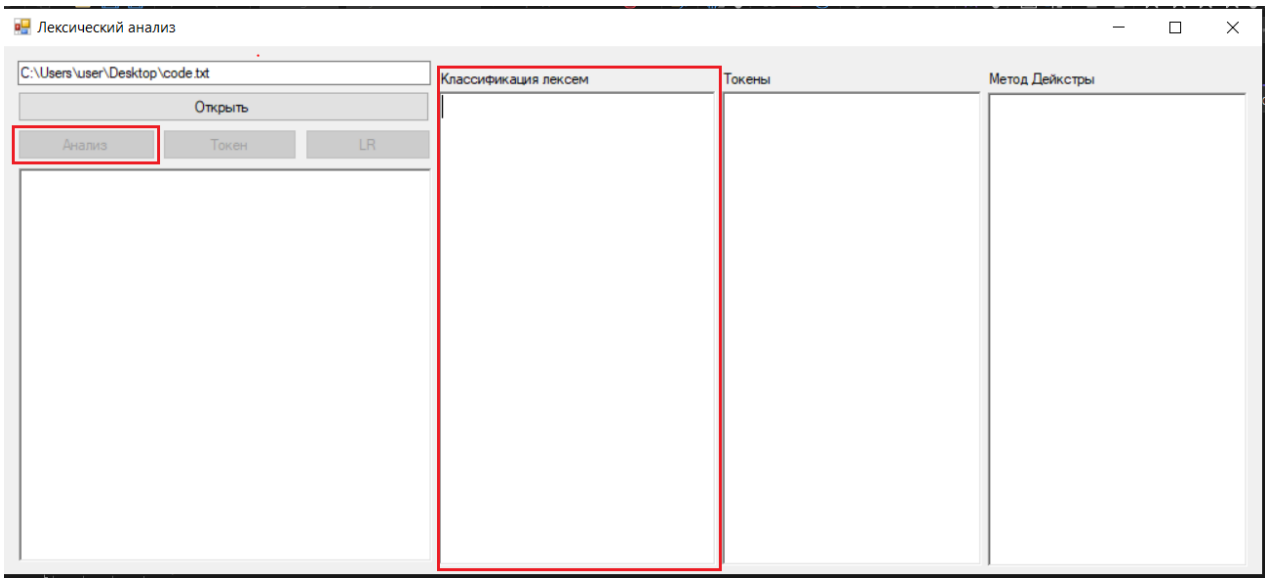

Рисунок 10 – Использование анализа лексем

Аналогично прошлому действию мы можем получить токены из ведённых данных нажав на кнопку “Токен” (Рисунок 11)

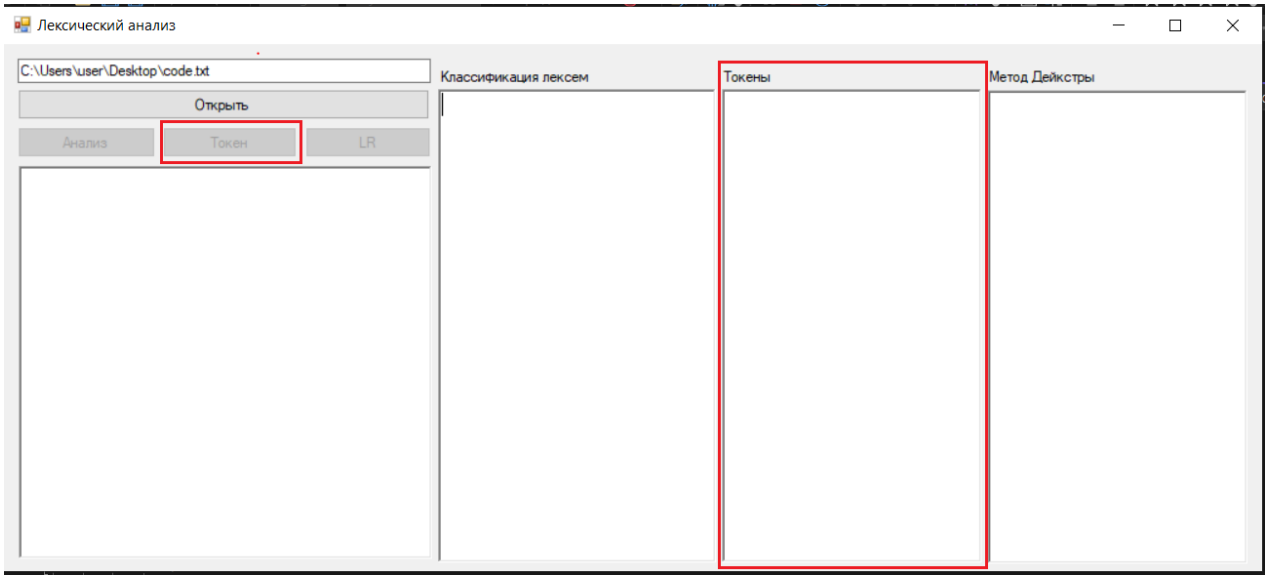

Рисунок 11 – Токенизация

Для запуска работы восходящего анализатора следует нажать на кнопку “LR” (Рисунок 12).

В случае ошибки программа укажет неверно введённый символ (Рисунок 13).

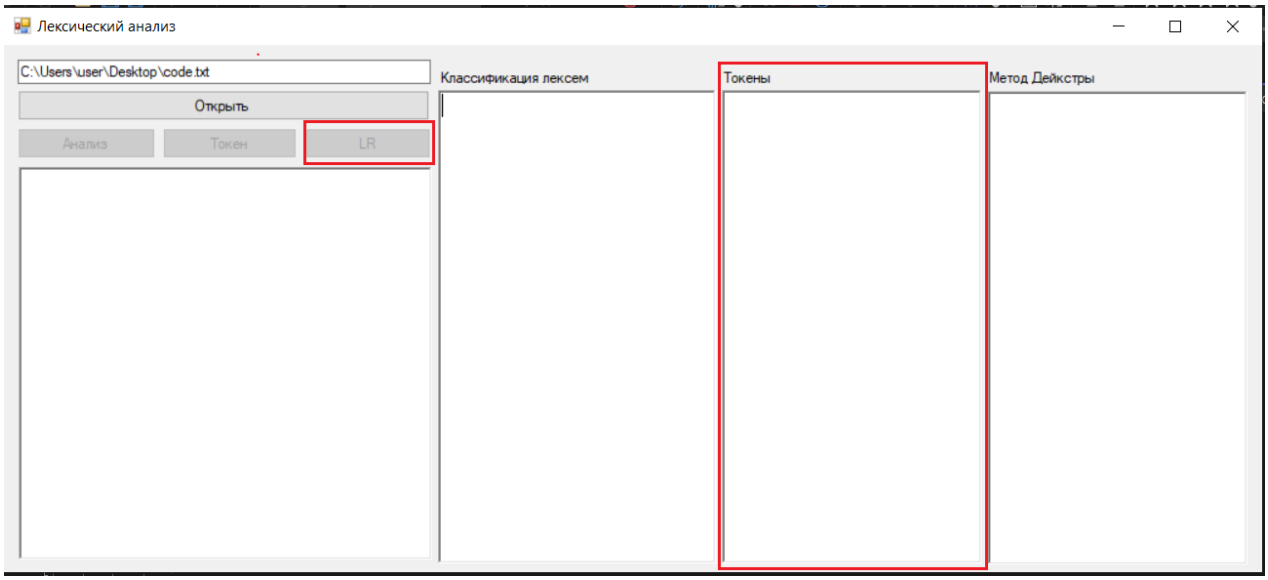

Рисунок 12 – Кнопка для запуска восходящего анализатора и поле вывода Токенов и запуска разбора сложно-логического выражения

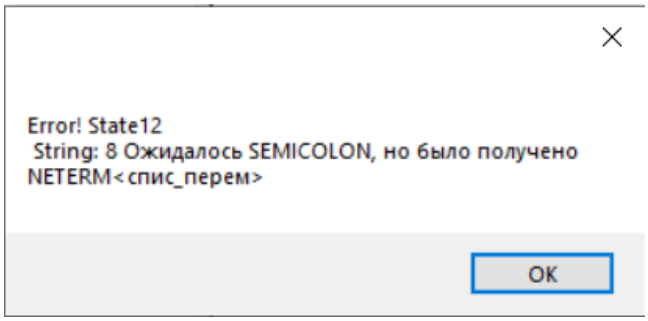

Рисунок 13 – Работа программы приостановлена из-за неправильных исходных данных.

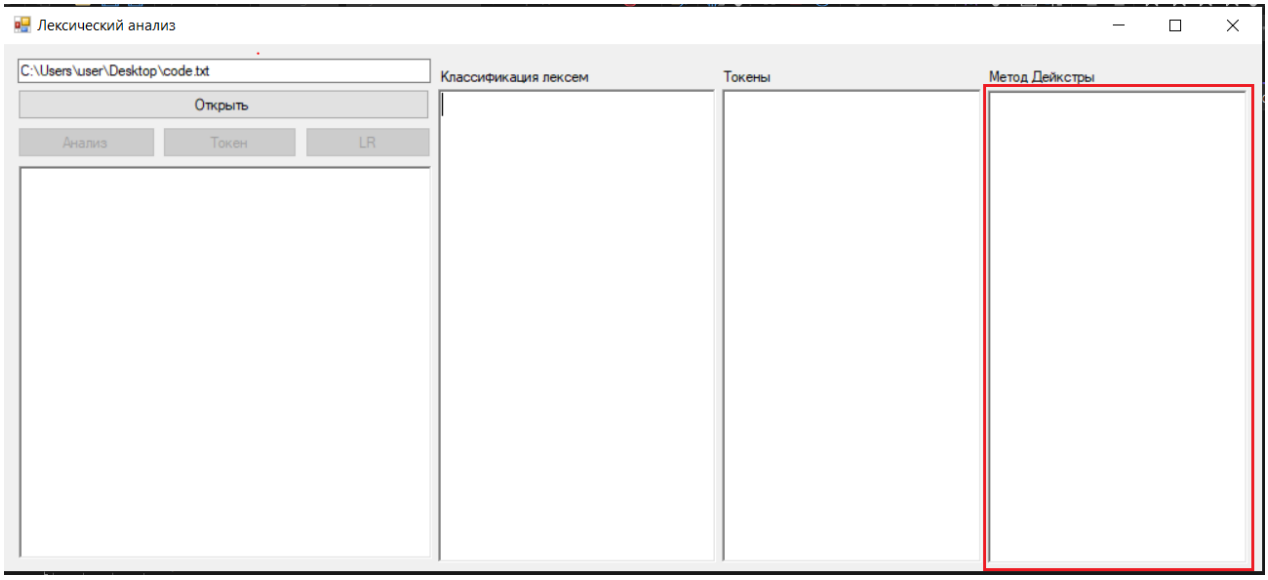

Рисунок 14 – Поле вывода разбора сложно-логического выражения

6. Руководство программиста

Входными данными для транслятора является текст программы, полученный от пользователя, с синтаксисом подмножества языка программирования C. Выходными данными транслятора являются лексический разбор 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 методы программы и их назначения:

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 LR.cs

List<Token> tokens – Список для хранения токенов

Stack<Token> lexemStack – Стек разбора

Stack<int> stateStack - Стек для хранения состояний анализатора.

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

private int state - Целочисленное значение, содержит номер текущего состояния анализатора.

private Token GetLexeme(int nextLex) – Метод, возвращаемый токен из списка токенов

private bool isEnd - Логическая переменная, содержит информацию о том, что список токенов пуст

public LR(List<Token> enterToken) – Конструктор класса LR

private void Shift – Метод сдвига очередного символа из входной строки в стек с переходом к следующему.

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

Возвращаемое значение

Token - Токен из списка токенов.

Исключения

Exception - Список лексем пуст, но анализ не был завершён.

private void GoToState(int state) - Метод перевода анализатора в другое состояние

state int - Целочисленное значение, содержит номер текущего состояния анализатора.

private void Reduce(int num, string neterm) - Свёртка, метод замены, хранимой в стеке разбора правой части правила на левую.

Параметры

num int - Целочисленное значение, содержит число удаляемых из стеков символов.

neterm string - Строка, содержащая название нетерминала, отправляемого в стек разбора.

private void Expression() - Метод разбора сложных логических выражений.

public void Start() - Метод запуска восходящего анализатора.

type TokenType - Поле для хранения типа токена.

Возвращаемое значение

string - Строка, содержащая ошибочно полученную лексему.

private void State1() … private void State45() - Методы состояний, действия которых соответствует решающей таблице 5.

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

Expression.cs

Метод TakeToken получает на вход список токинов, которые в последствие добавляются в список logicExpressionStack

Словарь priority содержит в себе приоритеты для каждой операции

Метод Start является входной точкой в программу и вызывает 2 метода Decstra() и ReversePolishNotationInMatrixView();

Метод Decstra выполняет метод Дейкстры и в переменную output записывают обратную Польскую нотацию

Метод PushesOutOperationsHighestPriority принимает на вход строковую форму операции, является вспомогательным методом для метода Decstra и выполняет выталкивание всех операций выше приоритета, чем входная операция из стека stackOfOperations в выходную строку output

Метод ReversePolishNotationInMatrixView переводит Обратную Польскую нотацию в матричный вид и выводит на форму

Заключение

В результате работы создан транслятор программы на подмножестве языка С. Данная программа может выполнять лексический, синтаксический анализы, а также разбор сложного логического выражения.

Для синтаксического разбора был использован метод LR(k)-грамматик. Для разбора сложного арифметического выражения использован метод Дейкстры.

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

Для более подробного ознакомления с приложением можно воспользоваться ссылкой на репозиторий данного проекта:

https://github.com/plkkz/TranslyatorC

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

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 с. — ISBN 987-5-7433-2968-7. — Текст: электронный // Электроннобиблиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/76519.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей. - DOI: https://doi.org/10.23682/76519 - https://www.iprbookshop.ru/76519.html

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. — ISBN 978-5-9275-2397-9. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/87427.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей -https://www.iprbookshop.ru/87427.html

3) Пентус, А. Е. Математическая теория формальных языков: учебное пособие / А. Е. Пентус, М. Р. Пентус. — 3-е изд. — Москва: Интернет Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. — 218 с. — ISBN 978-5-4497-0662-1. — Текст: электронный // Электроннобиблиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/97548.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей - https://www.iprbookshop.ru/97548.html

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 с. — ISBN 978-5-292-04613-4. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/99047.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей - https://www.iprbookshop.ru/99047.html

5) Малявко, А. А. Формальные языки и компиляторы: учебник / А. А. Малявко. — Новосибирск: Новосибирский государственный технический университет, 2014. — 431 с. — ISBN 978-5-7782-2318-9. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/47725.html (дата обращения: 15.04.2021). — Режим доступа: для авторизированных пользователей - https://www.iprbookshop.ru/47725.html.