

# Содержание

ВВЕДЕНИЕ .....	2
1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.....	4
1.1 Постановка задачи.....	4
1.2 Порядок выполнения.....	6
1.3 Грамматика языка.....	6
2 ПРАКТИЧЕСКАЯ ЧАСТЬ .....	10
2.1 Разработка лексического анализатора.....	10
2.2 Разработка синтаксического анализатора .....	11
2.3 Семантический анализ .....	13
3 ТЕСТИРОВАНИЕ ПРОГРАММЫ.....	14
3.1 Тест 1. Правильный код.....	15
3.2 Тест 2. Лексическая ошибка.....	17
3.3 Тест 3. Синтаксическая ошибка .....	18
3.4 Тест 4. Семантическая ошибка.....	20
ЗАКЛЮЧЕНИЕ.....	23
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	24
ПРИЛОЖЕНИЯ .....	25
Приложение А.....	26
Приложение Б .....	32

# ВВЕДЕНИЕ

В информатике под формальным языком подразумевается язык программирования, построенный по правилам некоторого логического исчисления и представляющий собой систему построения конечных знаков последовательностей в заданном алфавите. Теория формальных языков является разделом математической лингвистики – науки, исследующей языки и формальные методы их построения. Теория формальных языков включает в себя способы описания формальных грамматик языков, построение методов и алгоритмов анализа принадлежности цепочек языку, а ещё алгоритмов перевода (трансляции) алгоритмических языков на язык машины. Теория формальных языков зародилась в конце 50-х годов XX века, когда команда под руководством американского ученого Джона Бэкуса разработала первый высокоуровневый язык программирования Фортран. В своей деятельности команда Бэкуса базировалась на научных разработках лингвиста Ноама Хомского, создавшего классификацию формальных языков.

Разработка нового языка программирования требует креативного подхода и кропотливой работы, невзирая на наличие немалого числа алгоритмов для автоматизации процесса написания транслятора для формальных языков. Это касается синтаксиса языка, который должен быть комфортен в прикладном программировании, а также должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Целью представленной курсовой работы является разработка распознавателя модельного языка программирования, согласно установленной формальной грамматике, основанной на персональном варианте. Для достижения цели нужно реализовать следующие задачи:

- освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;

- приобретение практических навыков по написанию транслятора языка программирования;
- закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией.

# 1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## 1.1 Постановка задачи

Разработать распознаватель модельного языка программирования, согласно установленной формальной грамматике на основе персонального варианта.

Распознаватель представляет собой алгоритм, позволяющий вынести решение о принадлежности цепочки символов некоторому языку.

Распознаватель схематично представляется в виде совокупности входной ленты, читающей головки, указывающей на очередной символ на ленте, устройства управления (далее сокращение УУ) и дополнительной памяти.

Конфигурацией распознавателя является:

- состояние УУ;
- содержимое входной ленты;
- положение читающей головки;
- содержимое дополнительной памяти.

Трансляция исходного текста программы осуществляется в несколько этапов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация целевого кода.

Лексический анализ является наиболее простой фазой и выполняется с помощью регулярной грамматики. Регулярным грамматикам соответствуют конечные автоматы, следовательно, разработка и написание программы лексического анализатора эквивалентна разработке конечного автомата и его диаграммы состояний (далее сокращение ДС).

Алгоритм синтаксического анализа строится на базе контекстно-свободных (далее сокращение КС) грамматик. Задача синтаксического анализатора – провести анализ разбора текста программы и сопоставить его с формальным описанием языка.

Семантический анализ позволяет учесть особенности языка, которые не могут быть описаны правилами КС-грамматики. К таким особенностям относятся:

- обработка описаний;
- анализ выражений;
- проверка правильности операторов.

Обработка описаний позволяет убедиться в том, что каждая переменная в программе описана и только один раз.

Анализ выражений заключается в том, чтобы проверить описаны ли переменные, участвующие в выражении, и соответствуют ли типы операндов друг другу и типу операции.

Этапы синтаксического и семантического анализа обычно объединяют.

## 1.2 Порядок выполнения

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики, вариант № 19.
2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка.
3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня.
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня.
5. Построить программный продукт, читающий текст программы, написанной на модельном языке, в виде консольного приложения.
6. Протестировать работу программного продукта с помощью серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

## 1.3 Грамматика языка

Согласно персональному варианту № 27 курсовой работы, грамматика языка включает следующие синтаксические конструкции:

1.  $\langle \text{операции\_группы\_отношения} \rangle ::= != | = | < | <= | > | >=$
2.  $\langle \text{операции\_группы\_сложения} \rangle ::= + | - | ||$
3.  $\langle \text{операции\_группы\_умножения} \rangle ::= * | / | \&\&$
4.  $\langle \text{унарная\_операция} \rangle ::= !$

5.  $\langle \text{программа} \rangle ::= \{ / (\langle \text{описание} \rangle \mid \langle \text{оператор} \rangle) ( : \mid \text{переход строки}) / \}$   
end
6.  $\langle \text{описание} \rangle ::= \text{dim } \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} \langle \text{тип} \rangle$
7.  $\langle \text{тип} \rangle ::= \text{integer} \mid \text{real} \mid \text{boolean}$
8.  $\langle \text{составной} \rangle ::= \text{begin } \langle \text{оператор} \rangle \{ ; \langle \text{оператор} \rangle \} \text{end};$
9.  $\langle \text{присваивания} \rangle ::= \langle \text{идентификатор} \rangle := \langle \text{выражение} \rangle;$
10.  $\langle \text{условный} \rangle ::= \text{if } \langle \text{выражение} \rangle \langle \text{оператор} \rangle [ \text{else } \langle \text{оператор} \rangle ];$
11.  $\langle \text{фиксированного\_цикла} \rangle ::= \text{for } \langle \text{присваивания} \rangle \text{ to } \langle \text{выражение} \rangle$   
[step  $\langle \text{выражение} \rangle$ ]  $\langle \text{оператор} \rangle \text{ next};$
12.  $\langle \text{условного\_цикла} \rangle ::= \text{while } \langle \text{выражение} \rangle \langle \text{оператор} \rangle;$
13.  $\langle \text{ввода} \rangle ::= \text{readln } \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \};$
14.  $\langle \text{вывода} \rangle ::= \text{write } \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \};$
15.  $\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{операции\_группы\_отношения} \rangle \langle \text{операнд} \rangle \};$
16.  $\langle \text{операнд} \rangle ::= \langle \text{слагаемое} \rangle \{ \langle \text{операции\_группы\_сложения} \rangle \langle \text{слагаемое} \rangle \};$
17.  $\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{операции\_группы\_умножения} \rangle \langle \text{множитель} \rangle \};$
18.  $\langle \text{множитель} \rangle ::= \langle \text{идентификатор} \rangle \mid \langle \text{число} \rangle \mid$   
 $\langle \text{логическая\_константа} \rangle \mid \langle \text{унарная\_операция} \rangle \langle \text{множитель} \rangle \mid \langle \text{выражение} \rangle$   
 $\langle \text{выражение} \rangle;$

19.  $\langle \text{логическая\_константа} \rangle ::= \text{true} \mid \text{false};$
20.  $\langle \text{идентификатор} \rangle ::= \langle \text{буква} \rangle \{ \langle \text{буква} \rangle \mid \langle \text{цифра} \rangle \};$
21.  $\langle \text{число} \rangle ::= \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \};$
22.  $\langle \text{буква} \rangle ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid$   
 $w \mid x \mid y \mid z \mid A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid$   
 $W \mid X \mid Y \mid Z;$
23.  $\langle \text{цифра} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом « $::=$ », нетерминалы заключены в угловые скобки, а терминалы – просто символы, используемые в языке. Терминалы, представляющие собой ключевые слова языка:

- dim
- integer
- real
- boolean
- begin
- end
- if
- else
- for



- to
- step
- next
- while
- readln
- writeln
- true
- false

## 2 ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2.1 Разработка лексического анализатора

Лексический анализатор – это подпрограмма, принимающая на вход исходный код программы и выдающая последовательность лексем – минимальных элементов программы, несущих смысловую нагрузку. Исходный код программы записан в текстовом файле example.txt.

В модельном языке программирования были выделены следующие типы лексем:

- ключевые слова;
- ограничители;
- числа;
- идентификаторы.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел  $n, k$ , где  $n$  – номер таблицы, а  $k$  – номер лексемы в таблице.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «возврат каретки»), комментариев, завершение строки с помощью символа «точка с запятой» и комментарии, заключенные в фигурные скобки.

Лексический анализ текста проводится по регулярной грамматике.

Известно, что регулярная грамматика эквивалентна конечному автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата. Диаграмма состояний представлена на Рисунке 1.

Исходный код лексического анализатора приведен в Приложении А.

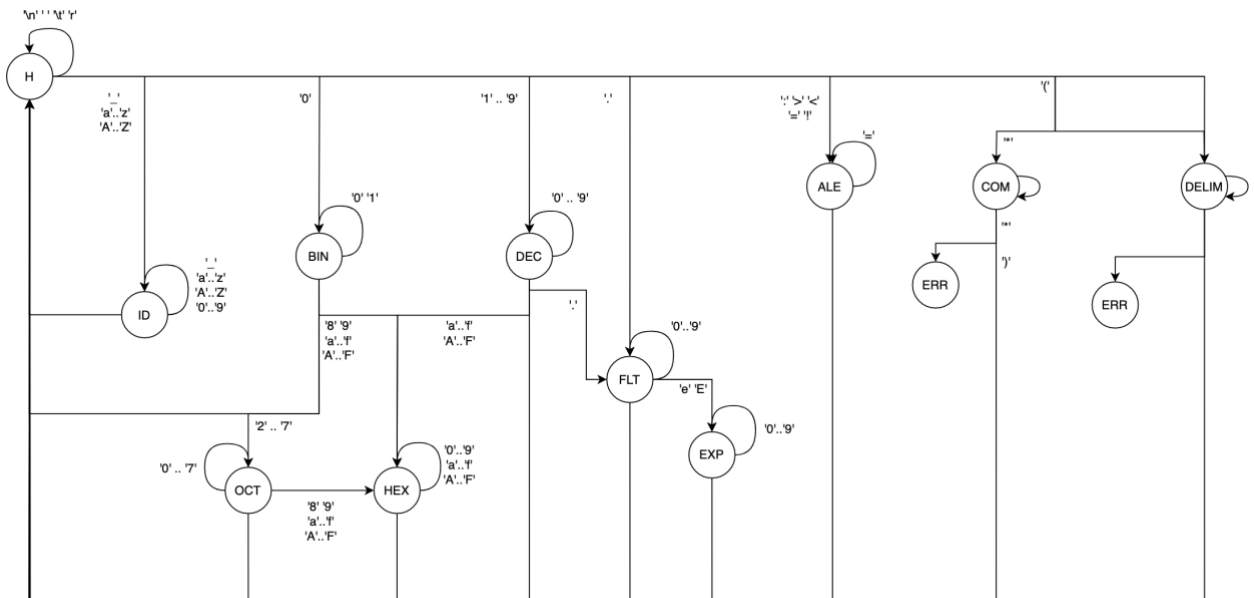


Рисунок 1 – Диаграмма состояний

## 2.2 Разработка синтаксического анализатора

Предположим, что лексический и синтаксический анализаторы взаимодействуют следующим образом: если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора.

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (далее сокращение РС). В основе метода лежит тот факт,

что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

- $P \rightarrow \{ / D \mid S / \} \text{ end}$
- $D \rightarrow I \{ , I \} [ \text{ integer } \mid \text{ boolean } \mid \text{ real } ]$
- $S \rightarrow I := E \mid \text{ if } "( E )" S [ \text{ else } S ] \mid \text{ for } I := E \text{ to } E [ \text{ step } E ] S \text{ next } \mid \text{ while } "( E )" S \mid \text{ begin } S \{ ; S \} \text{ end } \mid \text{ readln } I \{ , I \} \mid \text{ writeln } E \{ , E \}$
- $E \rightarrow E_1 \{ [ != \mid == \mid < \mid <= \mid > \mid >= ] E_1 \}$
- $E_1 \rightarrow T \{ [ + \mid - \mid \parallel ] T \}$
- $T \rightarrow F \{ [ * \mid / \mid \&\& ] F \}$
- $F \rightarrow I \mid N \mid L \mid !F \mid (E)$
- $L \rightarrow \text{ true } \mid \text{ false }$
- $I \rightarrow C \mid IC \mid IR$
- $N \rightarrow R \mid NR$
- $C \rightarrow \_ \mid a \dots z \mid A \dots Z$
- $R \rightarrow 0 \dots 9$

Здесь правила для нетерминалов L, I, N, C и R описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов P, D, S, E, E1, T, F.

Исходный код синтаксического анализатора приведен в Приложении А.

## 2.3 Семантический анализ

К сожалению, некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся:

- любой идентификатор, используемый в теле программы должен быть описан;
- повторное описание одного и того же идентификатора не разрешается.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в таблицу TID заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа при получении каждого идентификатора информация о нем заносится в буферную память, если он еще не объявлен. В случае использования идентификатора, отсутствующего в буферной памяти, ошибка об этом выводится пользователю. В случае повторной инициализации идентификатора, уже присутствующего в буферной памяти, ошибка об этом также выводится пользователю.

Описания функций семантических проверок приведены в листинге в Приложении Б.

### 3 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработано консольное приложение. Приложение принимает на вход исходный текст программы на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с номером некорректной лексемы. Список ошибок представлен в Таблице 1.

*Таблица 1 – Список ошибок*

<b>Номер ошибки</b>	<b>Суть ошибки</b>
001	Не удалось открыть файл
101	Ожидался символ ')'
102	Неизвестный разделитель
201	Отсутствует идентификатор

*Продолжение Таблицы 1*

202	Отсутствует тип идентификатора/ов
203	Отсутствует открывающая круглая скобка для условного оператор
204	Отсутствует открывающая круглая скобка для оператора условного цикла
210	Отсутствует закрывающая круглая скобка
211	Отсутствует оператор присваивания
212	Отсутствует выражение
250	Отсутствует ключевое слово end
299	Обнаружена неожиданная лексема. Неизвестная ошибка
301	Повторное объявление идентификатора
302	Присваивание к необъявленному идентификатору
320	Попытка чтения неинициализированного идентификатора

Рассмотрим примеры.

### **3.1 Тест 1. Правильный код**

*Листинг 1 – Код теста 1*

```
(*this is top commentary*)
```

```
dim i, j integer
```

```
dim temp_bool boolean
```

```

i := 12FF (*should be hex*)

j := 10 + 2

temp_bool := !true

dim middle_float real

middle_float := 1.1e2

(*this is middle commentary*)

if ( i < j )

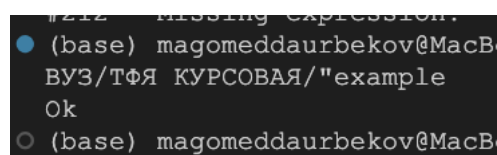
    writeln i, j

end

(*this is bottom commentary ( will not even be noticed by lexer )*)

```

Программа выдаст информацию об успешном прохождении теста. Результат работы программы при тесте 1 представлен на Рисунке 2.



```

#212 - missing expression.
● (base) magomeddaurbekov@MacB
  ВУЗ/ТФЯ КУРСОВАЯ/"example
  Ok
○ (base) magomeddaurbekov@MacB

```

**Рисунок 2 – Результат работы программы при тесте 1**



## 3.2 Тест 2. Лексическая ошибка

Для проверки правильности программы, во 2 тесте была допущена умышленная лексическая ошибка. В 6 строке кода были добавлены символы «@#\$^» не обусловленные правилами грамматики.

*Листинг 2 – Код теста 2*

```
(*this is top commentary*)

dim i, j integer

dim temp_bool boolean

@#$^

i := 12FF (*should be hex*)

j := 10 + 2

temp_bool := !true

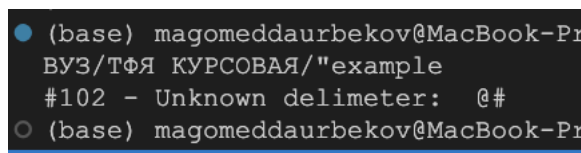
dim middle_float real

middle_float := 1.1e2

(*this is middle commentary*)
```

```
if ( i < j )  
  
    writeln i, j  
  
end  
  
(*this is bottom commentary ( will not even be noticed by lexer *))
```

Программа должна выдать ошибку при прохождении лексического анализа. Так как последнее, на что проверяет лексический анализатор, это разделители, то в случае успешного тестирования, программа должна выдать ошибку 102, что согласно Таблице 1 характеризуется как «Неизвестный разделитель». Результат работы программы при тесте 2 представлен на Рисунке 3.



```
(base) magomeddaurbekov@MacBook-Pro:~/ВУЗ/ТФЯ КУРСОВАЯ/"example"  
#102 - Unknown delimiter: @#  
(base) magomeddaurbekov@MacBook-Pro:~/ВУЗ/ТФЯ КУРСОВАЯ/"example"
```

Рисунок 3 – Результат работы программы при тесте 2

### 3.3 Тест 3. Синтаксическая ошибка

Для проверки правильности программы, в 3 тесте была допущена умышленная синтаксическая ошибка. В 13 строке кода было убрано ключевое слово «:=», что нарушает правила грамматики.

*Листинг 3 – Код теста 3*

```
(*this is top commentary*)

dim i, j integer

dim temp_bool boolean

i := 12FF (*should be hex*)

j := 10 + 2

temp_bool := !true

dim middle_float real

middle_float 1.1e2

(*this is middle commentary*)

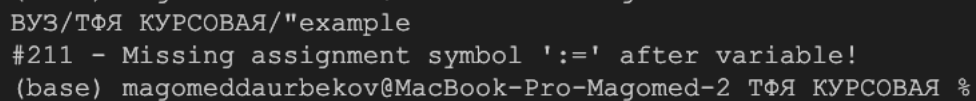
if ( i < j )

    writeln i, j

end

(*this is bottom commentary ( will not even be noticed by lexer )*)
```

Программа должна выдать ошибку при прохождении синтаксического анализа. В случае успешного тестирования, программа должна выдать ошибку 211, что согласно Таблице 1 характеризуется как "Отсутствует оператор присваивания". Результат работы программы при тесте 3 представлен на Рисунке 4.



```
ВУЗ/ТФЯ КУРСОВАЯ/"example
#211 - Missing assignment symbol ':= ' after variable!
(base) magomeddaurbekov@MacBook-Pro-Magomed-2 ТФЯ КУРСОВАЯ %
```

**Рисунок 4 – Результат работы программы при тесте 3**

### **3.4 Тест 4. Семантическая ошибка**

Для проверки правильности программы, в 4 тесте была допущена умышленная семантическая ошибка. Во 13 строке кода была добавлена повторная инициализация переменной «j», что нарушает семантические условия.

*Листинг 4 – Код теста 4*

```
(*this is top commentary*)

dim i, j integer

dim temp_bool boolean

i := 12FF (*should be hex*)

j := 10 + 2

temp_bool := !true
```

```

dim middle_float real

dim j real

middle_float := 1.1e2

(*this is middle commentary*)

if ( i < j )

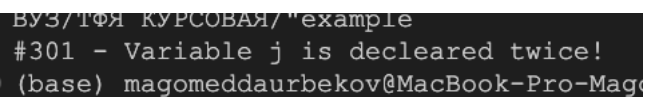
    writeln i, j

end

(*this is bottom commentary ( will not even be noticed by lexer )*)

```

Программа должна выдать ошибку при прохождении синтаксического анализа на этапе семантической проверки переменных. В случае успешного тестирования, программа должна выдать ошибку 301, что согласно Таблице 1 характеризуется как «Повторное объявление идентификатора». Результат работы программы при тесте 4 представлен на Рисунке 5.



```

ВУЗ/ТФЯ КУРСОВАЯ/"example
#301 - Variable j is declared twice!
(base) magomeddaurbekov@MacBook-Pro-Mag

```

**Рисунок 5 – Результат работы программы при тесте 4**

Таким образом, после прохождения всех приведённых проверок можно сказать, что программа работает корректно.

## ЗАКЛЮЧЕНИЕ

В процессе проектирования курсовой работы был разработан распознаватель модельного языка, который содержит в себе: лексический, синтаксический и семантический анализаторы. Лексический анализатор, разделяющий последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на высокоуровневом языке C++.

Анализ исходного текста программы был сделан при помощи синтаксического анализатора, который также реализован на языке C++. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости была перестроена грамматика, в частности, специальным образом обработаны встречающиеся итеративные синтаксически конструкции (нетерминалы D, S, E и T).

В код рекурсивных функций были интегрированы проверки семантических условий – проверка на повторное объявление ранее объявленной переменной и проверка на использование необъявленной переменной.

Тестирование приложения показало, что лексический, синтаксический и семантический анализаторы работают корректно, а написанная программа успешно распознается анализатором, а также, что программа, содержащая ошибки, выдает ошибки с кратким описанием их сути.

В процессе выполнения работы были изучены основные принципы построения систем на основе теории автоматов и формальных грамматик, были получены навыки лексического, синтаксического и семантического анализа предложений языков программирования.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Свердлов С.З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.
2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.
3. Миронов С.В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019.
4. Антик М.И., Казанцева Л.В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.
5. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.



## **ПРИЛОЖЕНИЯ**

Приложение А – Код лексического анализатора

Приложение Б – Код синтаксического анализатора (с семантическими проверками)

## Приложение А

### Код лексического анализатора

```
Lex Lexer::getLex()
{
    if (feof(fp))
        throw string("#250 - Missing 'end' keyword in the end of program!");

    CS = H;
    do {
        switch (CS) {
            case H:
                if (c == ' ' || c == '\\t' || c == '\\n' || c == '\\r')
                {
                    gc();
                }

                else if (c == '_' || isalpha(c))
                {
                    CS = ID;

                    clear();
                    add();
                    gc();
                }

                else if (isnumber(c))
                {
                    if (c == '0')
                        CS = BIN;
                    else
                        CS = DEC;

                    clear();
                    add();
                    gc();
                }

                else if (c == '.')
                {
                    CS = FLT;

                    clear();
                    add();
                }
            }
        }
    }
```

```

        gc();
    }

    else if(c == '(')
    {
        gc();
        if(c == '*')
        {
            CS = COM;
            gc();
        }
        else
        {
            int idx = look("(", TD);
            return Lex(LEX_LPAREN, idx);
        }
    }

    else if(c == ':' || c == '>' || c == '<' || c == '=' || c ==
'!')
    {
        CS = ALE;

        clear();
        add();
        gc();
    }

    else
    {
        CS = DELIM;

        clear();
        add();
        gc();
    }
    break;

case ID:
    if(c == '_' || isalpha(c) || isnumber(c))
    {
        add();
        gc();
    }
    else
    {
        int idx = look(buf, TW);

```

```

        if (idx)
            return Lex(words[idx], idx);
        else
            return Lex(LEX_ID, TID.put(buf));
    }
    break;

case BIN:
    if (isbin(c))
    {
        add();
        gc();
    }
    else if (isoct(c))
    {
        CS = OCT;

        add();
        gc();
    }
    else if (ishex(c))
    {
        CS = HEX;

        add();
        gc();
    }
    else
    {
        return Lex(LEX_NUM, bstoi(buf));
    }
    break;

case OCT:
    if (isoct(c))
    {
        add();
        gc();
    }
    else if (ishex(c))
    {
        CS = HEX;

        add();
        gc();
    }
    else

```

```

        {
            return Lex(LEX_NUM, ostoi(buf));
        }
        break;

case DEC:
    if(isalpha(c))
    {
        add();
        gc();
    }
    else if(c == '.')
    {
        CS = FLT;

        add();
        gc();
    }
    else if(ishex(c))
    {
        CS = HEX;

        add();
        gc();
    }
    else
    {
        return Lex(LEX_NUM, stoi(buf));
    }
    break;

case HEX:
    if(ishex(c))
    {
        add();
        gc();
    }
    else
    {
        return Lex(LEX_NUM, hstoi(buf));
    }
    break;

case FLT:
    if(isnumber(c))
    {
        add();
    }

```

```

        gc();
    }
    else if(c == 'e' || c == 'E')
    {
        CS = EXP;

        add();
        gc();
    }
    else
    {
        float temp = stof(buf);
        return Lex(LEX_NUMREAL, *(int*)&temp);
    }
    break;

case EXP:
    if(isnumber(c))
    {
        add();
        gc();
    }
    else
    {
        float temp = stof(buf);
        return Lex(LEX_NUMREAL, *(int*)&temp);
    }
    break;

case COM:
    if(c == '*')
    {
        gc();
        if(c == ')')
        {
            gc();
            CS = H;
        }
        else
        {
            throw string("#101 - Missing comment closing symbol
') '");
        }
    }
    gc();
    break;

```

```

        case ALE:
            if(c == '=')
            {
                add();
                gc();

                int idx = look(buf, TD);
                return Lex(dlms[idx], idx);
            }
            else
            {
                int idx = look(buf, TD);
                return Lex(dlms[idx], idx);
            }
            break;

        case DELIM:
            int idx = look(buf, TD);

            if(idx)
            {
                return Lex(dlms[idx], idx);
            }
            else
            {
                add();
                gc();

                idx = look(buf, TD);
                if(idx)
                    return Lex(dlms[idx], idx);

                throw string("#102 - Unknown delimiter: ") + " " + buf;
            }

            break;
    }
}
while(true);
}

```

## Приложение Б

### Код синтаксического анализатора (с семантическими проверками)

```
void Parser::P()
{
    do
    {
        if(c_type == LEX_VAR)
            D();
        else
            S();
    } while (c_type != LEX_END);
}

void Parser::D()
{
    reset();
    do
    {
        gl();

        if(c_type != LEX_ID)
            throw string("#201 - Missing variable name!");

        push(c_val);
        gl();
    } while (c_type == LEX_COMMA);

    if(c_type != LEX_INT && c_type != LEX_REAL && c_type != LEX_BOOL)
        throw string("#202 - Missing { integer | real | boolean } type!");

    dec(c_type);

    gl();
}

void Parser::S()
{
    if (c_type == LEX_IF) {
        gl();

        if(c_type != LEX_LPAREN)
            throw string("#203 - Missing '(' after 'if' statement!");
    }
}
```



```

        gl();
        E();

        if (c_type != LEX_RPAREN)
            throw string("#210 - Missing ')' after expression!");

        gl();
        S();

        if (c_type == LEX_ELSE)
        {
            gl();
            S();
        }
    }
    else if (c_type == LEX_WHILE) {
        gl();

        if(c_type != LEX_LPAREN)
            throw string("#204 - Missing '(' after 'while' statement!");

        gl();
        E();

        if (c_type == LEX_RPAREN)
            throw string("#210 - Missing ')' after expression!");

        gl();
        S();
    }
    else if (c_type == LEX_READ) {
        do
        {
            gl();

            if(c_type != LEX_ID)
                throw string("#201 - Missing variable name!");

            if(!TID[c_val].isAssigned())
                throw string("#320 - Reading unassigned variable '" +
TID[c_val].getName() + "' !");

        } while (c_type == LEX_COMMA);
    }
    else if (c_type == LEX_WRITE) {
        do

```

```

        {
            gl();
            E();

            } while (c_type == LEX_COMMA);
        }
    else if (c_type == LEX_ID) {
        auto& curr_id = TID[c_val];

        if(!curr_id.isDeclared())
            throw string("#302 - using undeclared variable '" +
curr_id.getName() + "' !");

        gl();

        if (c_type != LEX_ASSIGN)
            throw string("#211 - Missing assignment symbol ':' after
variable!");

        gl();
        E();

        curr_id.setAssigned();
    }
    else if(c_type == LEX_BEGIN) {
        do
        {
            gl();
            S();
        } while (c_type == LEX_SEMICOLON);

        if(c_type != LEX_END)
            throw string("#250 - Missing 'end' keyword!");

        gl();
    }
    else
        throw string("#299 - Unkown operator!");
}

void Parser::E()
{
    E1();
    if (c_type == LEX_EQ ||
        c_type == LEX_LSS ||
        c_type == LEX_GTR ||
        c_type == LEX_LEQ ||

```

```

        c_type == LEX_GEQ ||
        c_type == LEX_NEQ)
    {
        gl();
        E1();
    }
}

void Parser::E1()
{
    T();
    while (c_type == LEX_PLUS || c_type == LEX_MINUS || c_type == LEX_OR) {
        gl();
        T();
    }
}

void Parser::T()
{
    F();
    while (c_type == LEX_TIMES || c_type == LEX_SLASH || c_type == LEX_AND) {
        gl();
        F();
    }
}

void Parser::F()
{
    if (c_type == LEX_ID) {
        if(!TID[c_val].isDeclared())
            throw string("#302 - using undeclared variable '" +
TID[c_val].getName() + "' !");

        if(!TID[c_val].isAssigned())
            throw string("#320 - Reading unassigned variable '" +
TID[c_val].getName() + "' !");

        gl();
    }
    else if (c_type == LEX_NUM || c_type == LEX_NUMREAL) {
        gl();
    }
    else if (c_type == LEX_TRUE || c_type == LEX_FALSE) {
        gl();
    }
    else if (c_type == LEX_NOT) {
        gl();
    }
}

```

```

        F();
    }
    else if (c_type == LEX_LPAREN) {
        gl();
        E();
        if (c_type != LEX_RPAREN) {
            throw string("#210 - Missing symbol ')' after expression");
        }
        gl();
    }
    else {
        throw string("#212 - Missing expression!");
    }
}

```