



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

КУРСОВАЯ РАБОТА

по дисциплине

Теория формальных языков

(наименование дисциплины)

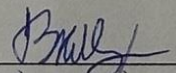
Тема курсовой работы

Разработка распознавателя модельного языка
программирования (вариант №2)

(наименование темы)


Студент группы ИКБО-04-22
(учебная группа)

Кликушин В.И.
(Фамилия И.О.)


(подпись студента)

Руководитель
курсовой работы доцент каф. ВТ, к.т.н.

Унгер А.Ю.


(подпись руководителя)

Консультант ст. преп. каф. ВТ

Боронников А.С.


(подпись консультанта)

Работа представлена к защите « » _____ 2023 г.

Допущен к защите « » _____ 2023 г.

Москва 2023



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

Утверждаю

Заведующий кафедрой

(подпись)

Платонова О.В.

«22» сентября 2023 г.

ЗАДАНИЕ

на выполнение курсовой работы по дисциплине

« Теория формальных языков »

Студент Кликушин Владислав Игоревич Группа ИКБО-04-22

Тема работы: Разработка распознавателя модельного языка программирования

Исходные данные: Грамматика модельного языка согласно варианту №2,
язык программирования – Python

Перечень вопросов, подлежащих разработке, и обязательного графического материала:

- 1) Проектирование диаграммы состояний лексического анализатора;
- 2) Разработка лексического анализатора;
- 3) Разработка синтаксического анализатора;
- 4) Разработка семантического анализатора;
- 5) Описание спецификации основных процедур и функций;
- 6) Исходный код с комментариями;
- 7) Тестирование распознавателя модельного языка программирования.

Срок представления к защите курсовой работы:

до «22» декабря 2023 г.

Задание на курсовую работу выдал

(подпись)

(Боронников А.С.)
ФИО консультанта

Задание на курсовую работу получил

«22» сентября 2023 г.

(подпись)

(Кликушин В.И.)
ФИО обучающегося

Москва 2023

ОТЗЫВ

на курсовую работу

по дисциплине «Теория формальных языков»

Студент Кликушин Владислав Игоревич группа ИКБО-04-22
(ФИО студента) (Группа)

Характеристика курсовой работы

Критерий	Да	Нет	Не полностью
1. Соответствие содержания курсовой работы указанной теме	+		
2. Соответствие курсовой работы заданию	+		
3. Соответствие рекомендациям по оформлению текста, таблиц, рисунков и пр.	+		
4. Полнота выполнения всех пунктов задания	+		
5. Логичность и системность содержания курсовой работы	+		
6. Отсутствие фактических грубых ошибок	+		

Замечания:

—

Рекомендуемая оценка:

отлично

а.при. Борзничев А.
доцент, к.т.н. Унгер А.Ю.

(Подпись руководителя)

(ФИО руководителя)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 ПОСТАНОВКА ЗАДАЧИ	7
2 ПОРЯДОК ВЫПОЛНЕНИЯ	8
3 ГРАММАТКА МОДЕЛЬНОГО ЯЗЫКА	9
4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА	11
5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА	13
6 СЕМАНТИЧЕСКИЙ АНАЛИЗ	15
7 ТЕСТИРОВАНИЕ ПРОГРАММЫ	16
ЗАКЛЮЧЕНИЕ	24
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	25
ПРИЛОЖЕНИЯ	26
Приложение А	27
Приложение Б	37
Приложение В	42

ВВЕДЕНИЕ

Развитие информационных технологий в последние десятилетия привело к увеличению объема программного кода, написанного на различных языках программирования. Для обеспечения правильной работы программ, необходимо не только их синтаксический анализ, но и оценка соответствия кода стандартам программирования и наличие правильных структурных элементов.

Первые языки программирования появились в середине XX века и имели свои особенности и ограничения. Вплоть до появления высокоуровневых языков программирования программирование было очень трудоёмким процессом, требующим внимательности и точности. С развитием компьютерных технологий важность языков программирования только возрастала, и разработчики начали активно стремиться к созданию более эффективных и удобных языков для решения сложных задач.

Однако, при разработке новых языков программирования возникает задача интерпретации и компиляции данных языков. Компиляторы — это программные средства, которые преобразуют программы на определенном языке программирования в машинный код, понятный компьютеру. Правильная работа компилятора обеспечивает эффективность и корректность выполнения программы.

Распознаватель модельного языка программирования играет важную роль в процессе разработки языков программирования и компиляторов. Он позволяет анализировать структуру и синтаксические правила языка программирования, что необходимо для создания компиляторов и других инструментов разработки. Без распознавателей модельного языка программирования процесс создания новых языков программирования и компиляторов стал бы гораздо более сложным и трудоемким.

Цель курсовой работы:

- закрепление теоретических знаний в области теории формальных языков, грамматик и автоматов;
- формирование практических умений и навыков разработки собственного распознавателя модельного языка программирования;
- закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

1 ПОСТАНОВКА ЗАДАЧИ

Разработать распознаватель модельного языка программирования согласно заданной грамматике варианта.

Распознаватель – это специальный алгоритм, который позволяет определить принадлежность цепочки символов некоторому языку. Распознаватель состоит из входной ленты, читающей головки, устройства управления и дополнительной памяти (стек). Конфигурацией распознавателя есть совокупность трех элементов: состояния управляющего устройства, содержимого входной ленты и положения входной головки, содержимого вспомогательной памяти.

В рамках курсовой работы были выполнены первые три этапа трансляции — лексический, синтаксический и семантический анализы.

Лексический анализ является необязательным этапом трансляции, однако крайне желательным по следующим причинам: замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки, лексический анализатор уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии. Лексический анализатор выполняется с использованием регулярных грамматик, поэтому принцип его работы эквивалентен разработке конечного автомата и его диаграммы состояний.

Для синтаксического разбора используются контекстно-свободные грамматики. Его задача - провести разбор текста программы, сопоставив его с эталоном, данным в описании языка.

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой.

2 ПОРЯДОК ВЫПОЛНЕНИЯ

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:
 - а) РБНФ;
 - б) диаграмм Вирта;
 - в) формальных грамматик;
2. Составить таблицу лексем и диаграмму состояний для распознавания и формирования лексем языка.
3. Разработать процедуру лексического анализа исходного текста программы на выбранном языке высокого уровня.
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на выбранном языке высокого уровня.
5. Построить программный продукт, анализирующий текст программы, написанной на модельном языке, в виде консольного приложения.
6. Протестировать работу программного продукта в полном объёме при помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки, которые могут быть допущены.

3 ГРАММАТКА МОДЕЛЬНОГО ЯЗЫКА

Для описания грамматики модельного языка воспользуемся расширенными формами Бэкуса-Наура (РБНФ), где символ «::=» отделяет левую часть правила от правой. Согласно индивидуальному варианту №2 задания на курсовую работы язык состоит из следующих синтаксических конструкций. Для удобства ключевые слова выделены полужирным шрифтом:

- $\langle \text{операции_группы_отношения} \rangle ::= = < > | = | < | < = | > | > =$
- $\langle \text{операции_группы_сложения} \rangle ::= + | - | \text{or}$
- $\langle \text{операции_группы_умножения} \rangle ::= * | / | \text{and}$
- $\langle \text{унарная_операция} \rangle ::= \text{not}$
- $\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{операции_группы_отношения} \rangle \langle \text{операнд} \rangle \}$
 - $\langle \text{операнд} \rangle ::= \langle \text{слагаемое} \rangle \{ \langle \text{операции_группы_сложения} \rangle \langle \text{слагаемое} \rangle \}$
 - $\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{операции_группы_умножения} \rangle \langle \text{множитель} \rangle \}$
 - $\langle \text{множитель} \rangle ::= \langle \text{идентификатор} \rangle \quad | \quad \langle \text{число} \rangle \quad | \quad \langle \text{логическая_константа} \rangle \quad | \quad \langle \text{унарная_операция} \rangle \quad \langle \text{множитель} \rangle \quad | \quad \langle \text{«}(\rangle \langle \text{выражение} \rangle \langle \text{«} \rangle \rangle$
 - $\langle \text{число} \rangle ::= \langle \text{целое} \rangle | \langle \text{действительное} \rangle$
 - $\langle \text{логическая_константа} \rangle ::= \text{true} | \text{false}$
 - $\langle \text{идентификатор} \rangle ::= \langle \text{буква} \rangle \{ \langle \text{буква} \rangle | \langle \text{цифра} \rangle \}$
 - $\langle \text{буква} \rangle ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z$
 - $\langle \text{цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 - $\langle \text{целое} \rangle ::= \langle \text{двоичное} \rangle \quad | \quad \langle \text{восьмеричное} \rangle \quad | \quad \langle \text{десятичное} \rangle \quad | \quad \langle \text{шестнадцатеричное} \rangle$

- $\langle \text{двоичное} \rangle ::= \{ / 0 \mid 1 / \} (B \mid b)$
- $\langle \text{восьмеричное} \rangle ::= \{ / 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 / \} (O \mid o)$
- $\langle \text{десятичное} \rangle ::= \{ / \langle \text{цифра} \rangle / \} [D \mid d]$
- $\langle \text{шестнадцатеричное} \rangle ::= \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \mid A \mid B \mid C \mid D \mid E \mid F \mid a \mid b \mid c \mid d \mid e \mid f \} (H \mid h)$
- $\langle \text{действительное} \rangle ::= \langle \text{числовая_строка} \rangle \langle \text{порядок} \rangle \mid [\langle \text{числовая_строка} \rangle] . \langle \text{числовая_строка} \rangle [\langle \text{порядок} \rangle]$
 - $\langle \text{числовая_строка} \rangle ::= \{ / \langle \text{цифра} \rangle / \}$
 - $\langle \text{порядок} \rangle ::= (E \mid e) [+ \mid -] \langle \text{числовая_строка} \rangle$
 - $\langle \text{программа} \rangle ::= \langle \{ \rangle \{ / (\langle \text{описание} \rangle \mid \langle \text{оператор} \rangle) ; / \} \langle \} \rangle$
 - $\langle \text{описание} \rangle ::= \mathbf{dim} \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} \langle \text{тип} \rangle$
 - $\langle \text{тип} \rangle ::= \mathbf{integer} \mid \mathbf{real} \mid \mathbf{boolean}$
 - $\langle \text{оператор} \rangle ::= \langle \text{составной} \rangle \mid \langle \text{присваивания} \rangle \mid \langle \text{условный} \rangle \mid \langle \text{фиксированного_цикла} \rangle \mid \langle \text{условного_цикла} \rangle \mid \langle \text{ввода} \rangle \mid \langle \text{вывода} \rangle$
 - $\langle \text{составной} \rangle ::= \langle [\rangle \langle \text{оператор} \rangle \{ (: \mid \text{перевод строки}) \langle \text{оператор} \rangle \} \langle] \rangle$
 - $\langle \text{присваивания} \rangle ::= \langle \text{идентификатор} \rangle \mathbf{as} \langle \text{выражение} \rangle$
 - $\langle \text{условный} \rangle ::= \mathbf{if} \langle \text{выражение} \rangle \mathbf{then} \langle \text{оператор} \rangle [\mathbf{else} \langle \text{оператор} \rangle]$
 - $\langle \text{фиксированного_цикла} \rangle ::= \mathbf{for} \langle \text{присваивания} \rangle \mathbf{to} \langle \text{выражение} \rangle \mathbf{do} \langle \text{оператор} \rangle$
 - $\langle \text{условного_цикла} \rangle ::= \mathbf{while} \langle \text{выражение} \rangle \mathbf{do} \langle \text{оператор} \rangle$
 - $\langle \text{ввода} \rangle ::= \mathbf{read} \langle (\rangle \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} \langle) \rangle$
 - $\langle \text{вывода} \rangle ::= \mathbf{write} \langle (\rangle \langle \text{выражение} \rangle \{ , \langle \text{выражение} \rangle \} \langle) \rangle$
 - $\langle \text{начало_комментария} \rangle ::= \langle /* \rangle$
 - $\langle \text{конец_комментария} \rangle ::= \langle */ \rangle$

4 ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР

Лексический анализатор – это подпрограмма, принимающая на вход исходный текст программы и выдающая последовательность лексем — минимальных элементов программы, несущих смысловую нагрузку.

Можно выделить следующие типы лексем:

- служебные (ключевые) слова;
- ограничители (разделители);
- числа;
- идентификаторы.

При разработке лексического анализатора, ключевые слова и ограничители известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для лексического анализа написана функция `Lexical`, которая читает поток символов из файла и разбивает его на лексемы. Особенности выбранного языка Python позволяют создавать вложенные функции – этот подход используется для написания вспомогательных функций для работы лексического анализатора, например функция считывания символа.

Для хранения лексем введён массив, который содержит словари и является результатом работы лексического анализатора. Каждый словарь имеет четыре элемента:

- а) Ключ «Тип лексемы» и значение по ключу - идентификатор значения типа лексемы
- б) Ключ «Значение» и значение по ключу – значение лексемы
- в) Ключ «Номер строки» и значение по ключу – порядковый номер строки, где находится текущая лексема
- г) Ключ «Позиция в строке» и значение по ключу – порядковый номер текущей лексемы в строке.

В массив лексем не сохраняется информация о символах перехода на новую строку, символах табуляции и пробела, а также весь текст,

5 СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Синтаксический анализатор — это подпрограмма, отвечающая за проверку текста программы на соответствие установленным синтаксическим правилам. На вход ему подаётся таблица лексем, сформированная на этапе лексического анализа, а результатом работы является сообщение об успешном завершении анализа или сообщение об ошибке с её описанием.

Синтаксический анализатор по аналогии с лексическим представлен в виде функции, которая работает с массивом лексем, полученным при лексическом анализе. Функция содержит внутренние функции, отвечающие за считывание лексемы и сам синтаксический анализ.

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (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:

- $\text{program} \rightarrow \langle\{ \rangle \text{body} \langle\{ \rangle$
- $\text{body} \rightarrow \{ / (\text{description} \mid \text{operator}) \langle ; \rangle / \}$
- $\text{description} \rightarrow \text{id_sequence} (\text{integer} \mid \text{real} \mid \text{boolean})$
- $\text{id_sequence} \rightarrow \text{id} \{ , \text{id} \}$
- $\text{operator} \rightarrow \langle [\rangle \text{operator} \{ (\langle : \rangle \mid \text{перевод строки}) \text{operator} \} \langle] \rangle \mid \text{ID as expression} \mid \text{if expression then operator [else operator]} \mid \text{while expression do operator} \mid \text{for ID as expression to expression do operator} \mid \text{read} \langle (\rangle \text{id_sequence} \langle) \rangle \mid \text{write} \langle (\rangle \text{expression_sequence} \langle) \rangle$
- $\text{expression_sequence} \rightarrow \text{expression} \{ , \text{expression} \}$
- $\text{expression} \rightarrow \text{operand} [(> \mid < \mid > = \mid < = \mid < > \mid =) \text{operand}]$
- $\text{operand} \rightarrow \text{summand} [(+ \mid - \mid \text{or}) \text{summand}]$
- $\text{summand} \rightarrow \text{multiplier} [(* \mid / \mid \text{and}) \text{multiplier}]$

- multiplier \rightarrow ID | NUMBER | **true** | **false** | **not** multiplier | «(» expression «)»

Код синтаксического анализатора представлен в Приложении Б.

6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

Некоторые особенности языка нельзя описать контекстно-свободной грамматикой. К таким особенностям относятся:

- любой идентификатор, используемый в теле программы должен быть описан;
- повторное описание одного и того же идентификатора не разрешается;
- в операторе присваивания типы идентификаторов должны совпадать;
- в условном операторе и операторе цикла в качестве условия допустимы только логические выражения;
- операнды операций отношения должны быть целочисленными.

Эти особенности проверяются на этапе семантического анализа, который удобно соvestить с синтаксическим анализом.

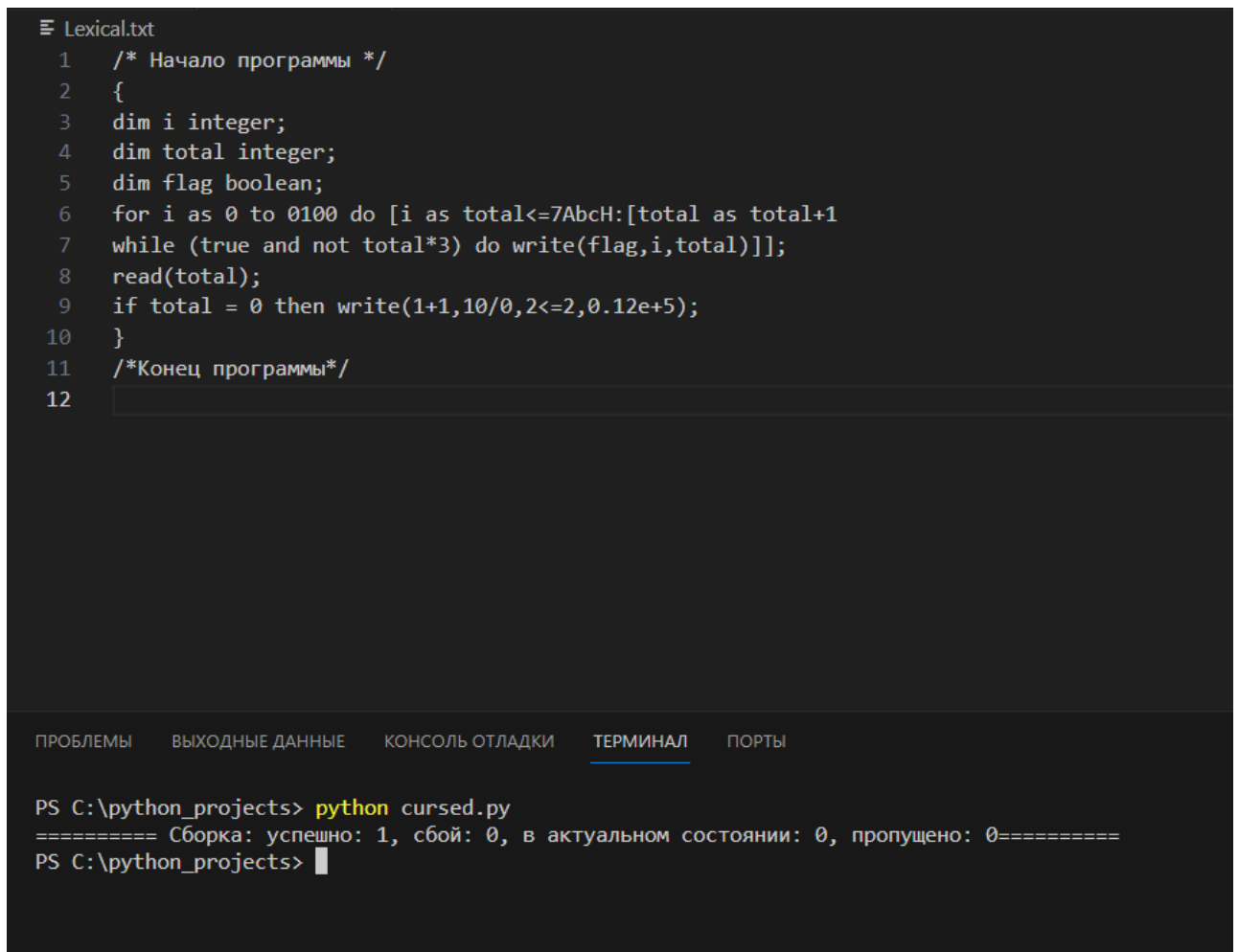
Подход к каждой семантической проверке может кардинально отличаться. Например, для проверки объявленной переменной необходимо на этапе лексического анализа сохранять для каждого идентификатора информацию об объявлении. Информация же будет обновляться в синтаксическом анализаторе при каждом новом разборе описания. Важно помнить, что повторное объявление переменной недопустимо.

7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Программным продуктом выступает программа *main.py*, которая запускается в консоли командой *python main.py* на Windows или *python3 main.py* на Linux или MacOS. После запуска программа предлагает пользователю ввести путь до анализируемой программы. Исходный код программы *main.py* представлен в Приложении В.

Программа выводит сообщение об успешном запуске в случае отсутствия каких-либо ошибок. Если на этапе лексического анализа появилась неизвестная лексема, синтаксический анализ не будет произведен и выведется сообщение об ошибке. Если лексический анализ прошел успешно, в дело вступает синтаксический, который сообщает об ошибках или их отсутствии. Для начала проверим работу распознавателя на полностью корректной программе, написанной на модельном языке.

1. Исходный код синтаксически верной программы (не несущей какой-либо смысл) представлен на Рисунке 7.1 совместно с сообщением об успешной проверке.



```
Lexical.txt
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1
7  while (true and not total*3) do write(flag,i,total)]];
8  read(total);
9  if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
10 }
11 /*Конец программы*/
12
```

ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ

```
PS C:\python_projects> python cursed.py
===== Сборка: успешно: 1, сбой: 0, в актуальном состоянии: 0, пропущено: 0=====
PS C:\python_projects>
```

Рисунок 7.1 — Пример синтаксически верной программы

2. Исходный текст синтаксически верной программы представлен на Рисунке 7.2 совместно с сообщением о неизвестной лексеме с указанием номера строки и позиции в строке.

The screenshot shows a code editor with two tabs: 'cursed.py' and 'Lexical.txt'. The 'Lexical.txt' tab is active, displaying a Python program with several lines of code. Line 6 contains the declaration 'dim 1C real;', which is a lexical error because '1C' is not a valid identifier. The program includes comments in Russian, variable declarations, a loop, conditional logic, and a write statement. Below the code editor, there is a terminal window with tabs: 'ПРОБЛЕМЫ', 'ВЫХОДНЫЕ ДАННЫЕ', 'КОНСОЛЬ ОТЛАДКИ', 'ТЕРМИНАЛ' (selected), and 'ПОРТЫ'. The terminal shows the command 'python cursed.py' being executed, followed by the error message 'Неизвестная лексема: 1C' and 'line 6, position 5'.

```
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  dim 1C real;
7  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1
8  while (true and not total*3) do write(flag,i,total)]];
9  read(total);
10 if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
11 }
12 /*Конец программы*/
13
```

ПРОБЛЕМЫ Выходные данные Консоль отладки **ТЕРМИНАЛ** Порты

```
PS C:\python_projects> python cursed.py
Неизвестная лексема: 1C
line 6, position 5
PS C:\python_projects>
```

Рисунок 7.2 — Пример программы с лексической ошибкой

Здесь ошибка заключается в лексеме «1C». Идентификатор не может начинаться с цифры. Но и число не содержит «C» в конце.

3. Исходный текст программы, содержащей лексическую ошибку, представлен на Рисунке 7.3 совместно с сообщением об неизвестной лексеме.

```
cursed.py Lexical.txt X
Lexical.txt
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1
7  while (true and not total*3) do write(flag,i,total)]];
8  read(total);
9  if total = 0 then write(1+1,10/0,2<=2,0.12e+5e);
10 }
11 /*Конец программы*/
12

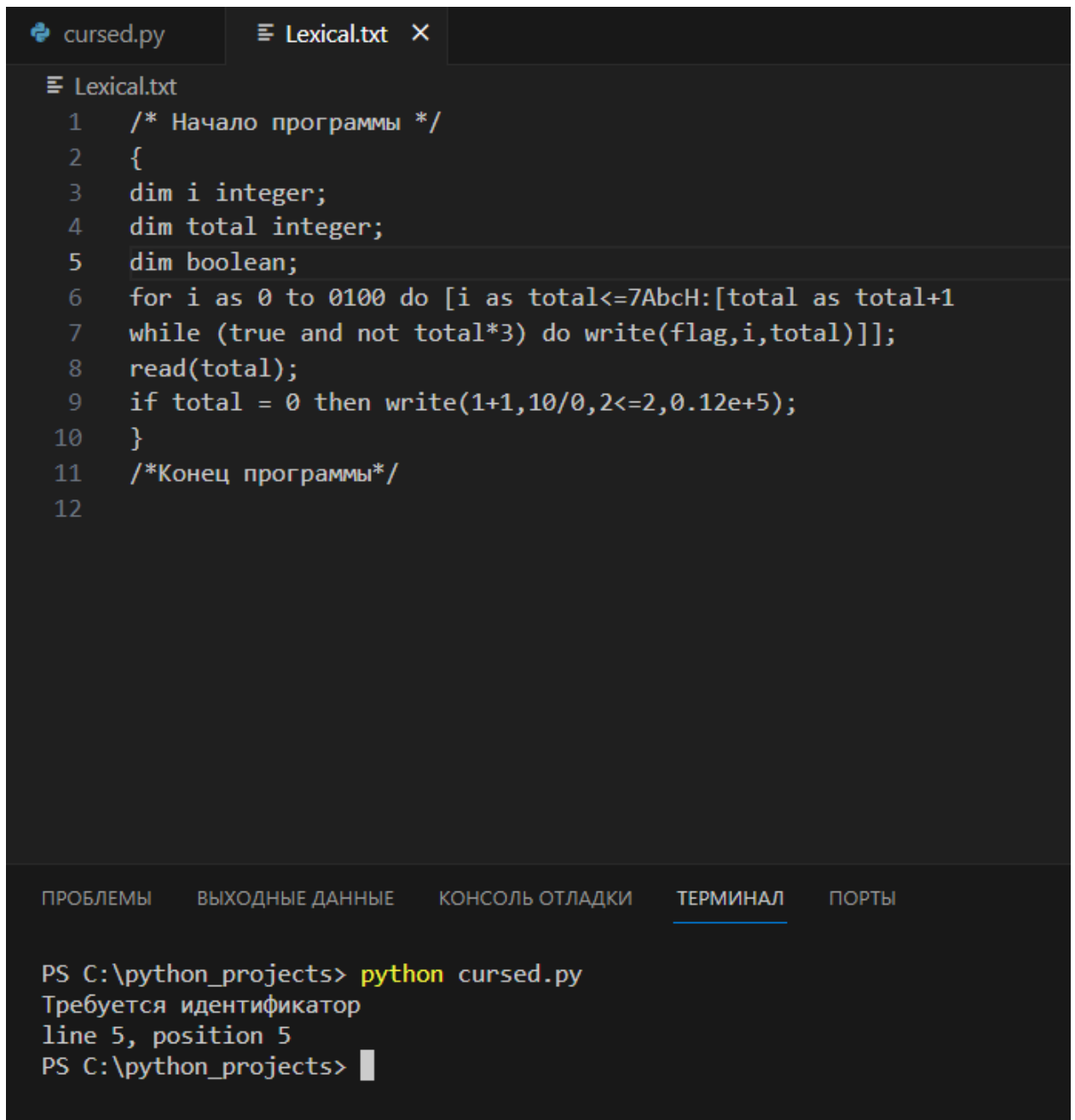
ПРОБЛЕМЫ  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ  ПОРТЫ

PS C:\python_projects> python cursed.py
Неизвестная лексема: 0.12e+5e
line 9, position 39
PS C:\python_projects> 
```

Рисунок 7.3 — Пример программы с лексической ошибкой

Данное действительное число не является корректным, потому что содержит дополнительный символ «е» в конце.

4. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.4 совместно с сообщением об ошибке.



```
cursed.py Lexical.txt X
Lexical.txt
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim boolean;
6  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1
7  while (true and not total*3) do write(flag,i,total)]];
8  read(total);
9  if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
10 }
11 /*Конец программы*/
12

ПРОБЛЕМЫ Выходные данные Консоль отладки Терминал Порты

PS C:\python_projects> python cursed.py
Требуется идентификатор
line 5, position 5
PS C:\python_projects> 
```

Рисунок 7.4 — Пример программы с синтаксической ошибкой

После ключевого слова `dim`, использующегося для объявления переменной был пропущен идентификатор.

5. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.5 совместно с сообщением об ошибке.

The screenshot shows a code editor with a file named `Lexical.txt` open. The code is a Python program with several lines of comments and logic. Line 7 contains a `while` loop with a syntax error: `while (true and not total*3) do write(flag,i,total,))];`. The error is a closing parenthesis `)` without a corresponding opening parenthesis `(`. The terminal at the bottom shows the command `python cursed.py` being executed, followed by an error message: `Ошибка в выражении`, `line 7, position 52`. The terminal prompt is `PS C:\python_projects>`.

```
cursed.py Lexical.txt X
Lexical.txt
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1
7  while (true and not total*3) do write(flag,i,total,))];
8  read(total);
9  if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
10 }
11 /*Конец программы*/
12

ПРОБЛЕМЫ  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ  ПОРТЫ

PS C:\python_projects> python cursed.py
Ошибка в выражении
line 7, position 52
PS C:\python_projects> 
```

Рисунок 7.5 — Пример программы с синтаксической ошибкой

В операторе `while` была поставлена лишняя запятая, требующая еще один идентификатор для вывода.

6. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.6 совместно с сообщением об ошибке.

```
Lexical.txt
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  for i as 0 to 0100 do [i as total<=7AbcH:[total as total+1;while (true and not total*3) do write(flag,i,total)]];
7  read(total);
8  if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
9  }
10 /*Конец программы*/
11
```

ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ

```
PS C:\python_projects> python cursed.py
Отсутствует символ ]
line 6, position 59
PS C:\python_projects>
```

Рисунок 7.6 — Пример программы с синтаксической ошибкой

В составном операторе отсутствует разделитель между операторами. Ожидался символ перехода на новую строку или двоеточие.

7. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.7 совместно с сообщением об ошибке.

The image shows a code editor with a file named 'Lexical.txt' open. The code is written in a Python-like syntax and contains a syntax error on line 6. The code is as follows:

```
1  /* Начало программы */
2  {
3  dim i integer;
4  dim total integer;
5  dim flag boolean;
6  for i as 0 to 0100 [i as total<=7AbcH:[total as total+1
7  while (true and not total*3) do write(flag,i,total)]];
8  read(total);
9  if total = 0 then write(1+1,10/0,2<=2,0.12e+5);
10 }
11 /*Конец программы*/
12
```

Below the code editor is a terminal window with the following output:

```
PS C:\python_projects> python cursed.py
Отсутствует оператор do
line 6, position 20
PS C:\python_projects> |
```

Рисунок 7.7 — Пример программы с синтаксической ошибкой
Пропущено ключевое слово `do` в операторе `for`.

ЗАКЛЮЧЕНИЕ

В работе представлены результаты разработки анализатора языка программирования. Грамматика языка задана с помощью правил вывода и описана в форме Бэкуса-Наура (БНФ). Согласно грамматике, в языке присутствуют лексемы следующих базовых типов: числовые константы, переменные, разделители и ключевые слова.

Разработан лексический анализатор, позволяющий разделить последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на языке высокого уровня Python в виде функции Lexical.

Разбором исходного текста программы занимается синтаксический анализатор, принимающий на вход массив лексем, сгенерированный при лексической проверке, который реализован в виде функции Syntactic на языке Python. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образом обрабатывать встречающиеся итеративные синтаксически конструкции.

Тестирование программного продукта показало, что синтаксически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается. Программа выдаёт сообщение об успешной сборке при отсутствии ошибок и указывает на ошибку с информацией о номере строки и позиции в строке.

В ходе работы изучены основные принципы построения интеллектуальных систем на основе теории автоматов и формальных грамматик, приобретены навыки лексического, синтаксического и семантического анализа предложений языков программирования.

Подводя итог, курс теории формальных языков был усвоен в полном объёме, я получил ценный опыт и знания, которые будут полезны в моей дальнейшей профессиональной деятельности.

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. Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.
5. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.
6. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.
7. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007.

ПРИЛОЖЕНИЯ

Приложение А — Исходный код лексического анализатора

Приложение Б — Исходный код синтаксического анализатора

Приложение В — Исходный код запуска анализа программы

Приложение А

Исходный код лексического анализатора

Листинг А — Код функции *Lexical*

```
def Lexical():
    def gc():
        nonlocal total, line_number, on_line_number, letter
        if total < len(data)-1:
            if data[total] == '\n':
                line_number += 1
                on_line_number = 1
            else:
                on_line_number += 1
            total += 1
            letter = data[total]
            return True
        else:
            return False
    flag, FLAG_READ_FILE = False, True
    Lex_result = list()
    buffer = str()
    status = 'H'
    total, line_number, on_line_number = 0, 1, 1
    key_words = ("not", "or", "and", "dim", "integer", "real", "boolean",
"while",
                "as", "if", "else", "then", "for", "to", "do", "read",
"write", "true", "false")
    separators = ('{', '}', '<>', '=', '<', '>', '<=', '[', ']', ':', '/*',
'*/',
                '>=', '+', '-', '*', '/', ',', ';', '(', ')')
    define_dict = {"not": "NOT", "or": "OR", "and": "AND", "dim": "DIM",
"integer": "INTEGER", "real": "REAL", "boolean": "BOOLEAN", "while": "WHILE",
                "as": "ASSIGN", "if": "IF", "else": "ELSE", "then":
"THEN", "for": "FOR", "to": "TO", "do": "DO", "read": "READ", "write":
"WRITE",
                "true": "TRUE", "false": "FALSE", "{": "BEGIN_PROG", "}":
"END_PROG", "<>": "NEQ", "=": "EQUAL", "<": "LESS", ">": "GREATER",
                "<=": "LEQ", ">=": "GEQ", "[": "BEGIN_COMPLEX", "]":
"END_COMPLEX", ":": "COLON", "+": "PLUS", "-": "MINUS", "*": "MULT",
                "/": "DIVIDE", ",": "COMMA", ";": "SEMICOLON", "(":
"LEFT_PAREN", ")": "RIGHT_PAREN"}
    # ID, KEYWORD, TYPE, LINE_BREAK, BEGIN_COMMENT, END_COMMENT, ERR, TYPE_I,
TYPE_F, TYPE_B, EMPTY
    data = file.read()
    while True:
        letter = data[total]
        match status:
            case 'H':
                while letter in (' ', '\n', '\t') and FLAG_READ_FILE:
                    if letter == '\n' and flag:
                        Lex_result.append({"Тип лексемы": "LINE_BREAK",
"Значение": "LINE_BREAK",
                                                "Номер строки": line_number,
"Позиция в строке": on_line_number})
                        FLAG_READ_FILE = gc()
                    if not FLAG_READ_FILE:
                        return Lex_result
                    if letter.isalpha():
                        buffer = letter
                        FLAG_READ_FILE = gc()
```

```

        if not FLAG_READ_FILE:
            return Lex_result
        status = 'ID'
    elif letter in ('0', '1'):
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = "BINARY NUMBER"
    elif letter in map(str, list(range(2, 8))):
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = "OCTAL NUMBER"
    elif letter in ('8', '9'):
        status = 'DECIMAL NUMBER'
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
    elif letter == '.':
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'P1'
    elif letter == '/':
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'C1'
    elif letter == '<':
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'M1'
    elif letter == '>':
        buffer = letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'M2'
    elif letter == '}':
        buffer = letter
        Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                                                    "Номер строки": line_number, "Позиция в
строке": on_line_number}))
        FLAG_READ_FILE = gc()
    else:
        status = 'SEPARATOR'
case 'ID':
    while letter.isalnum():
        buffer += letter
        FLAG_READ_FILE = gc()
        if not FLAG_READ_FILE:
            return Lex_result
    if buffer in key_words:

```

Продолжение листинга А

```
Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
"Номер строки": line_number, "Позиция в строке": on_line_number-len(buffer)})
else:
Lex_result.append({"Тип лексемы": "ID", "Значение":
buffer,
"Номер строки": line_number, "Позиция в строке": on_line_number-len(buffer)})
status = 'H'
case 'BINARY NUMBER':
while letter in ('0', '1'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
if letter in map(str, list(range(2, 8))):
status = 'OCTAL NUMBER'
elif letter in ('8', '9'):
status = 'DECIMAL NUMBER'
elif letter in ('a', 'c', 'f', 'A', 'C', 'F'):
status = 'HEX NUMBER'
elif letter in ('e', 'E'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'Ell'
elif letter in ('d', 'D'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'D'
elif letter in ('o', 'O'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'O'
elif letter in ('H', 'h'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'HX'
elif letter == '.':
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'Pl'
elif letter in ('b', 'B'):
buffer += letter
FLAG_READ_FILE = gc()
if not FLAG_READ_FILE:
return Lex_result
status = 'B'
elif letter.isalpha():
status = 'ERROR'
else:
```

```
        status = 'DECIMAL NUMBER'
    case 'OCTAL NUMBER':
        while letter in map(str, list(range(0, 8))):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
        if letter in ('8', '9'):
            status = 'DECIMAL NUMBER'
        elif letter in ('a', 'c', 'b', 'f', 'A', 'B', 'C', 'F'):
            status = 'HEX NUMBER'
        elif letter in ('e', 'E'):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'E11'
        elif letter in ('d', 'D'):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'D'
        elif letter in ('H', 'h'):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'HX'
        elif letter == '.':
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'P1'
        elif letter in ('o', 'O'):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'O'
        elif letter.isalpha():
            status = 'ERROR'
        else:
            status = 'DECIMAL NUMBER'
    case 'DECIMAL NUMBER':
        while letter in map(str, list(range(0, 10))):
            buffer += letter
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
        if letter in ('a', 'b', 'c', 'f', 'A', 'B', 'C', 'F'):
            status = 'HEX NUMBER'
        elif letter in ('e', 'E'):
            buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'E11'
        elif letter in ('h', 'H'):
            buffer += letter
```

```

        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'HX'
    elif letter == '.':
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'Pl'
    elif letter in ('d', 'D'):
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'D'
    elif letter.isalpha():
        status = 'ERROR'
    else:
        Lex_result.append({"Тип лексемы": "десятичное число",
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
        status = 'H'
        case 'HEX NUMBER':
            while letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e',
'f', 'A', 'B', 'C', 'D', 'E', 'F'):
                buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            if letter in ('h', 'H'):
                buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'HX'
        else:
            status = 'ERROR'
        case 'B':
            if letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e',
'f', 'A', 'B', 'C', 'D', 'E', 'F'):
                status = 'HEX NUMBER'
            elif letter in ('h', 'H'):
                buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'HX'
            elif letter.isalpha():
                status = 'ERROR'
            else:
                Lex_result.append({"Тип лексемы": "число в двоичной
системе", "Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
                status = 'H'
            case 'O':
                if letter.isalnum():
                    status = 'ERROR'
            else:

```


Продолжение листинга А

```
Lex_result.append({"Тип лексемы": "число в восьмеричной
системе", "Значение": buffer,
                  "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
    status = 'H'
    case 'D':
        if letter in ('h', 'H'):
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            status = 'HX'
        elif letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e',
'f', 'A', 'B', 'C', 'D', 'E', 'F'):
            status = 'HEX NUMBER'
        elif letter.isdigit():
            status = 'ERROR'
        else:
            Lex_result.append({"Тип лексемы": "десятичное число",
"Значение": buffer,
                  "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
            case 'HX':
                if letter.isalnum():
                    status = 'ERROR'
                else:
                    Lex_result.append({"Тип лексемы": "число в
шестнадцатеричной системе", "Значение": buffer,
                  "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
                    status = 'H'
                    case 'E11':
                        if letter.isdigit():
                            buffer += letter
                            gc()
                            if not FLAG_READ_FILE:
                                return Lex_result
                            status = 'E12'
                        elif letter in ('+', '-'):
                            buffer += letter
                            gc()
                            if not FLAG_READ_FILE:
                                return Lex_result
                            status = 'ZN'
                        elif letter in ('h', 'H'):
                            buffer += letter
                            gc()
                            if not FLAG_READ_FILE:
                                return Lex_result
                            status = 'HX'
                        elif letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e',
'f', 'A', 'B', 'C', 'D', 'E', 'F'):
                            buffer += letter
                            gc()
                            if not FLAG_READ_FILE:
                                return Lex_result
                            status = 'HEX NUMBER'
                        else:
                            status = 'ERROR'
                    case 'ZN':
                        if letter.isdigit():
```

```
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'E13'
    else:
        status = 'ERROR'
case 'E12':
    while letter.isdigit():
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    if letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e',
'f', 'A', 'B', 'C', 'D', 'E', 'F'):
        status = 'HEX NUMBER'
    elif letter in ('h', 'H'):
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'HX'
    elif letter.isalpha():
        status = 'ERROR'
    else:
        Lex_result.append({"Тип лексемы": "действительное число",
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
        status = 'H'
case 'E13':
    while letter.isdigit():
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    if letter.isalpha() or letter == '.':
        status = 'ERROR'
    else:
        Lex_result.append({"Тип лексемы": "действительное число",
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
        status = 'H'
case 'P1':
    if letter.isdigit():
        status = 'P2'
    else:
        status = 'ERROR'
case 'P2':
    while letter.isdigit():
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    if letter in ('e', 'E'):
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    status = 'E21'
```

```
elif letter.isalpha() or letter == '.':
    status = 'ERROR'
else:
    Lex_result.append({"Тип лексемы": "действительное число",
"Значение": buffer,
"Номер строки": line_number, "Позиция в строке": on_line_number-len(buffer)})
    status = 'H'
case 'E21':
    if letter in ('+', '-'):
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'ZN'
    elif letter.isdigit():
        status = 'E22'
    else:
        status = 'ERROR'
case 'E22':
    while letter.isdigit():
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    if (letter.isalpha() or letter == '.'):
        status = 'ERROR'
    else:
        Lex_result.append({"Тип лексемы": "действительное число",
"Значение": buffer,
"Номер строки": line_number, "Позиция в строке": on_line_number-len(buffer)})
        status = 'H'
case 'C1':
    if letter == '*':
        buffer += letter
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'C2'
    else:
        Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
"Номер строки": line_number, "Позиция в строке": on_line_number-len(buffer)})
        status = 'H'
case 'C2':
    while total != len(data) and letter != '*':
        gc()
        if not FLAG_READ_FILE:
            return Lex_result
    buffer = ''
    buffer += letter
    gc()
    if not FLAG_READ_FILE:
        return Lex_result
    status = 'C3'
case 'C3':
    if letter == '/':
        buffer += letter
        gc()
```

```
        if not FLAG_READ_FILE:
            return Lex_result
        status = 'H'
    else:
        status = 'C2'
    case 'M1':
        if letter == '>':
            buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
        elif letter == '=':
            buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
        else:
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
    case 'M2':
        if letter == '>':
            buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
        elif letter == '=':
            buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
        else:
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
    case 'SEPARATOR':
        buffer = letter
```

Продолжение листинга А

```
        if buffer in separators:
            FLAG_READ_FILE = gc()
            if not FLAG_READ_FILE:
                return Lex_result
            if buffer == '[':
                flag = True
            elif buffer == ']':
                flag = False
            Lex_result.append({"Тип лексемы": define_dict[buffer],
"Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
        else:
            status = 'ERROR'
        case 'ERROR':
            while FLAG_READ_FILE and letter not in (' ', '\n', '\t') and
letter not in separators:
                buffer += letter
            gc()
            if not FLAG_READ_FILE:
                return Lex_result
            Lex_result.append({"Тип лексемы": "ERR", "Значение": buffer,
                                "Номер строки": line_number, "Позиция в
строке": on_line_number-len(buffer)})
            status = 'H'
```

Приложение Б

Исходный код синтаксического анализатора

Листинг Б — Код функции Syntactic

```
def Syntactic():

    def GL():
        nonlocal total, prev_lex, current_lex
        if total < len(data):
            prev_lex = current_lex
            current_lex = data[total]
            total += 1
            return True
        else:
            print(
                "===== Сборка: успешно: 1, сбой: 0, в актуальном
состоянии: 0, пропущено: 0=====")
            exit(0)

    def EXPRESSION_SEQUENCE():
        GL()
        EXPRESSION()
        while current_lex['Тип лексемы'] == 'COMMA':
            GL()
            EXPRESSION()

    def ID_SEQUENCE():
        if current_lex['Тип лексемы'] == 'ID':
            GL()
            while current_lex['Тип лексемы'] == 'COMMA':
                GL()
                if current_lex['Тип лексемы'] == 'ID':
                    GL()
                else:
                    ERROR_HANDLER(5)
            else:
                ERROR_HANDLER(6)

    def ERROR_HANDLER(n: int):
        match n:
            case 1:
                print("Отсутствует символ }")
            case 2:
                print("Отсутствует символ {")
            case 3:
                print("Требуется оператор или описание переменной")
            case 4:
                print("Ожидалась точка с запятой")
            case 5:
                print("Ошибка в последовательности идентификаторов")
            case 6:
                print("Требуется идентификатор")
            case 7:
                print("Отсутствует символ ")
            case 8:
                print("Требуется тип переменной")
            case 9:
                print("Отсутствует оператор then")
            case 10:
                print("Отсутствует оператор do")
```

```
        case 11:
            print("Отсутствует оператор do")
        case 12:
            print("Отсутствует оператор to")
        case 13:
            print("Отсутствует символ ")
        case 14:
            print("Отсутствует символ (")
        case 15:
            print("Отсутствует символ ")
        case 16:
            print("Отсутствует символ (")
        case 17:
            print("Отсутствует символ ]")
        case 18:
            print("Ошибка в выражении")
        case 19:
            pass
        case 20:
            print("Неизвестный оператор")
        case 21:
            print("Требовался идентификатор")
    print(
        f"line {current_lex['Номер строки']], position
{current_lex['Позиция в строке']})")
    exit(n)

def DESCRIPTION():
    if current_lex['Тип лексемы'] == "DIM":
        GL()
        ID_SEQUENCE()
    if current_lex['Тип лексемы'] not in ('INTEGER', 'REAL',
'BOOLEAN'):
        ERROR_HANDLER(8)

def OPERATOR():
    match current_lex['Тип лексемы']:
        case "IF":
            GL()
            CONDITION_OPERATOR()
        case 'WHILE':
            GL()
            WHILE_OPERATOR()
        case "FOR":
            GL()
            FOR_OPERATOR()
        case "READ":
            GL()
            READ_OPERATOR()
        case "WRITE":
            GL()
            WRITE_OPERATOR()
        case "ID": # ASSIGN -> ID as EXPR
            GL()
            ASSIGN_OPERATOR()
        case "BEGIN_COMPLEX":
            GL()
            COMPLEX_OPERATOR()
        case _:
            ERROR_HANDLER(20)
```


Продолжение листинга Б

```
def EXPRESSION():
    OPERAND()
    if current_lex['Тип лексемы'] in ('EQUAL', 'LESS', 'GREATER', 'LEQ',
    'GEQ', 'NEQ'):
        GL()
        EXPRESSION()

def OPERAND():
    SUMMAND()
    if current_lex['Тип лексемы'] in ('PLUS', 'MINUS', 'OR'):
        GL()
        OPERAND()

def SUMMAND():
    MULTIPLIER()
    if current_lex['Тип лексемы'] in ('DIVIDE', 'MULT', 'AND'):
        GL()
        SUMMAND()

def MULTIPLIER():
    if current_lex['Тип лексемы'] in ('ID', 'действительное число',
    'десятичное число', 'число в шестнадцатеричной системе',
    'число в восьмеричной системе',
    'число в двоичной системе', 'TRUE', 'FALSE'):
        GL()
    elif current_lex['Тип лексемы'] == 'NOT':
        GL()
        MULTIPLIER()
    elif current_lex['Тип лексемы'] == 'LEFT_PAREN':
        GL()
        EXPRESSION()
    if current_lex['Тип лексемы'] == 'RIGHT_PAREN':
        GL()
    else:
        ERROR_HANDLER(7)
    else:
        ERROR_HANDLER(18)

def CONDITION_OPERATOR():
    EXPRESSION()
    if current_lex['Тип лексемы'] == 'THEN':
        GL()
        OPERATOR()
    if current_lex['Тип лексемы'] == 'ELSE':
        GL()
        OPERATOR()
    else:
        ERROR_HANDLER(9)

def WHILE_OPERATOR():
    EXPRESSION()
    if current_lex['Тип лексемы'] == 'DO':
        GL()
        OPERATOR()
    else:
        ERROR_HANDLER(10)

def FOR_OPERATOR():
    if current_lex['Тип лексемы'] != 'ID':
        ERROR_HANDLER(21)
    Return
```

```
GL()
ASSIGN_OPERATOR()
if current_lex['Тип лексемы'] == 'TO':
    GL()
    EXPRESSION()
    if current_lex['Тип лексемы'] == 'DO':
        GL()
        OPERATOR()
    else:
        ERROR_HANDLER(11)
else:
    ERROR_HANDLER(12)

def ASSIGN_OPERATOR():
    if current_lex['Тип лексемы'] == 'ASSIGN':
        GL()
        EXPRESSION()

def READ_OPERATOR():
    if current_lex['Тип лексемы'] == 'LEFT_PAREN':
        GL()
        ID_SEQUENCE()
        if current_lex['Тип лексемы'] != 'RIGHT_PAREN':
            ERROR_HANDLER(13)
        else:
            GL()
    else:
        ERROR_HANDLER(14)

def WRITE_OPERATOR():
    if current_lex['Тип лексемы'] == 'LEFT_PAREN':
        EXPRESSION_SEQUENCE()
        if current_lex['Тип лексемы'] == 'RIGHT_PAREN':
            GL()
        else:
            ERROR_HANDLER(15)
    else:
        ERROR_HANDLER(16)

def COMPLEX_OPERATOR():
    OPERATOR()
    while current_lex['Тип лексемы'] in ('COLON', 'LINE_BREAK'):
        GL()
        OPERATOR()
    if current_lex['Тип лексемы'] != 'END_COMPLEX':
        ERROR_HANDLER(17)
    else:
        GL()

def ANALYZE():
    PROGRAM()

def PROGRAM():
    GL()
    if current_lex['Тип лексемы'] == "BEGIN_PROG":
        GL()
        BODY()
        ERROR_HANDLER(1)
    else:
        ERROR_HANDLER(2)
```

Продолжение листинга Б

```
def BODY():
    if current_lex['Тип лексемы'] == "DIM":
        DESCRIPTION()
        GL()
    elif current_lex['Тип лексемы'] in ('ID', 'FOR', 'WHILE', 'IF',
'BEGIN_COMPLEX', 'READ', 'WRITE'):
        OPERATOR()
    else:
        ERROR_HANDLER(3)
    while current_lex['Тип лексемы'] == "SEMICOLON":
        GL()
        if current_lex['Тип лексемы'] == "DIM":
            DESCRIPTION()
            GL()
        elif current_lex['Тип лексемы'] in ('ID', 'FOR', 'WHILE', 'IF',
'BEGIN_COMPLEX', 'READ', 'WRITE'):
            OPERATOR()
        elif current_lex['Тип лексемы'] == "END_PROG":
            GL()
            break
    else:
        ERROR_HANDLER(4)

    if current_lex['Тип лексемы'] != "SEMICOLON":
        ERROR_HANDLER(4)

total = 0
current_lex = data[total]
prev_lex = dict()
ANALYZE()
```

Приложение В

Исходный код запуска анализа программы

Листинг В — Исходный код

```
with open('Lexical.txt', encoding='utf-8') as file:
    data = Lexical()
    for d in data:
        if d['Тип лексемы'] == "ERR":
            print(f"Неизвестная лексема: {d['Значение']}\nline {d['Номер строки']}, position {d['Позиция в строке']}")
            exit(0)
    Syntactic()
```