

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	2
1 ПОСТАНОВКА ЗАДАЧИ.....	3
2 ПОРЯДОК ВЫПОЛНЕНИЯ.....	5
3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА.....	6
4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА.....	8
5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА.....	9
6 СЕМАНТИЧЕСКИЙ АНАЛИЗ.....	11
7 ТЕСТИРОВАНИЕ ПРОГРАММЫ.....	12
ЗАКЛЮЧЕНИЕ.....	15
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	16
ПРИЛОЖЕНИЯ.....	17

ВВЕДЕНИЕ

Несмотря на более чем полувековую историю вычислительной техники, рождение теории формальных языков ведет отсчет с 1957 года. В этот год американский ученый Джон Бэкус разработал первый компилятор языка Фортран. Он применил теорию формальных языков, во многом опирающуюся на работы известного ученого-лингвиста Н. Хомского – автора классификации формальных языков. Хомский в основном занимался изучением естественных языков, Бэкус применил его теорию для разработки языка программирования. Это дало толчок к разработке сотен языков программирования.

Несмотря на наличие большого количества алгоритмов, позволяющих автоматизировать процесс написания транслятора для формального языка, создание нового языка требует творческого подхода. В основном это относится к синтаксису языка, который, с одной стороны, должен быть удобен в прикладном программировании, а с другой, должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа

Основы теории формальных языков и практические методы разработки распознавателей формальных языков составляют неотъемлемую часть образования современного инженера-программиста.

Целью данной курсовой работы является:

- освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;
- приобретение практических навыков написания транслятора языка программирования;
- закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией

1 ПОСТАНОВКА ЗАДАЧИ

Разработать распознаватель модельного языка программирования согласно заданной формальной грамматике.

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

Распознаватель можно схематично представить в виде совокупности входной ленты, читающей головки, которая указывает на очередной символ на ленте, устройства управления (УУ) и дополнительной памяти (стек).

Конфигурацией распознавателя является:

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

Трансляция исходного текста программы происходит в несколько этапов. Основными этапами являются следующие:

- 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 (ДС).

Синтаксический анализатор строится на базе контекстно-свободных (КС) грамматик. Задача синтаксического анализатора – провести разбор текста программы и сопоставить его с формальным описанием языка.

Семантический анализ позволяет учесть особенности языка программирования, 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 можно объединить.

2 ПОРЯДОК ВЫПОЛНЕНИЯ

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики;
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.

3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

- $\langle \text{операции_группы_отношения} \rangle ::= \text{NE} \mid \text{EQ} \mid \text{LT} \mid \text{LE} \mid \text{GT} \mid \text{GE}$
- $\langle \text{операции_группы_сложения} \rangle ::= \text{plus} \mid \text{min} \mid \text{or}$
- $\langle \text{операции_группы_умножения} \rangle ::= \text{mult} \mid \text{div} \mid \text{and}$
- $\langle \text{унарная_операция} \rangle ::= \sim$
- $\langle \text{программа} \rangle ::= \text{program var } \langle \text{описание} \rangle \text{ begin } \langle \text{оператор} \rangle \{ ; \langle \text{оператор} \rangle \} \text{ end.}$
 - $\langle \text{описание} \rangle ::= \{ \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} : \langle \text{тип} \rangle ; \}$
 - $\langle \text{тип} \rangle ::= \text{integer} \mid \text{real} \mid \text{boolean}$
 - $\langle \text{оператор} \rangle ::= \langle \text{составной} \rangle \mid \langle \text{присваивания} \rangle \mid \langle \text{условный} \rangle \mid \langle \text{фиксированного_цикла} \rangle \mid \langle \text{условного_цикла} \rangle \mid \langle \text{ввода} \rangle \mid \langle \text{вывода} \rangle$
 - 1) $\langle \text{составной} \rangle ::= \langle [\rangle \langle \text{оператор} \rangle \{ (: \mid \text{перевод строки}) \langle \text{оператор} \rangle \} \langle] \rangle$
 - $\langle \text{присваивания} \rangle ::= \langle \text{идентификатор} \rangle \text{ as } \langle \text{выражение} \rangle$
 - $\langle \text{условный} \rangle ::= \text{if } \langle \text{выражение} \rangle \text{ then } \langle \text{оператор} \rangle [\text{else } \langle \text{оператор} \rangle]$
 - $\langle \text{фиксированного_цикла} \rangle ::= \text{for } \langle \text{присваивания} \rangle \text{ to } \langle \text{выражение} \rangle \text{ do } \langle \text{оператор} \rangle$
 - $\langle \text{условного_цикла} \rangle ::= \text{while } \langle \text{выражение} \rangle \text{ do } \langle \text{оператор} \rangle$
 - $\langle \text{ввода} \rangle ::= \text{read } \langle (\rangle \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} \langle) \rangle$
 - $\langle \text{вывода} \rangle ::= \text{write } \langle (\rangle \langle \text{выражение} \rangle \{ , \langle \text{выражение} \rangle \} \langle) \rangle$
 - Многострочные комментарии в программе (шестая цифра варианта)?
- $\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{операции_группы_отношения} \rangle \langle \text{операнд} \rangle \}$
 - $\langle \text{операнд} \rangle ::= \langle \text{слагаемое} \rangle \{ \langle \text{операции_группы_сложения} \rangle \langle \text{слагаемое} \rangle \}$
 - $\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{операции_группы_умножения} \rangle \langle \text{множитель} \rangle \}$

- $\langle \text{множитель} \rangle ::= \langle \text{идентификатор} \rangle \mid \langle \text{число} \rangle \mid \langle \text{логическая_константа} \rangle \mid$
- $\langle \text{унарная_операция} \rangle \langle \text{множитель} \rangle \mid \langle \text{«}(\rangle \langle \text{выражение} \rangle \langle \text{«} \rangle \rangle$
- $\langle \text{число} \rangle ::= \langle \text{целое} \rangle \mid \langle \text{действительное} \rangle$
- $\langle \text{логическая_константа} \rangle ::= \text{true} \mid \text{false}$
- $\langle \text{идентификатор} \rangle ::= \langle \text{буква} \rangle \{ \langle \text{буква} \rangle \mid \langle \text{цифра} \rangle \}$
- $\langle \text{буква} \rangle ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z \mid a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
- $\langle \text{цифра} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- $\langle \text{целое} \rangle ::= \langle \text{двоичное} \rangle \mid \langle \text{восьмеричное} \rangle \mid \langle \text{десятичное} \rangle \mid$
- $\langle \text{шестнадцатеричное} \rangle$
- $\langle \text{двоичное} \rangle ::= \{ / 0 \mid 1 / \} (B \mid b)$
- $\langle \text{восьмеричное} \rangle ::= \{ / 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 / \} (O \mid o)$
- $\langle \text{десятичное} \rangle ::= \{ / \langle \text{цифра} \rangle / \} [D \mid d]$
- $\langle \text{шестнадцатеричное} \rangle ::= \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \mid A \mid B \mid C \mid D \mid E \mid F \mid a \mid b \mid c \mid d \mid e \mid f \} (H \mid h)$
- $\langle \text{действительное} \rangle ::= \langle \text{числовая_строка} \rangle \langle \text{порядок} \rangle \mid$
- $\langle \text{числовая_строка} \rangle \mid \langle \text{числовая_строка} \rangle [\langle \text{порядок} \rangle]$
- $\langle \text{числовая_строка} \rangle ::= \{ / \langle \text{цифра} \rangle / \}$
- $\langle \text{порядок} \rangle ::= (E \mid e) [+ \mid -] \langle \text{числовая_строка} \rangle$

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. Жирным выделены терминалы, представляющие собой ключевые слова языка.

4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Лексический анализатор – подпрограмма, которая принимает на вход исходный текст программы и выдает последовательность лексем – минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования выделяют следующие типы лексем:

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

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

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел (n, k) , где n – номер таблицы лексем, 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», «перенос строки», «возврат каретки») и комментариев, заключенных 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 (рис. 1).

Исходный код лексического анализатора приведен в Приложении А.

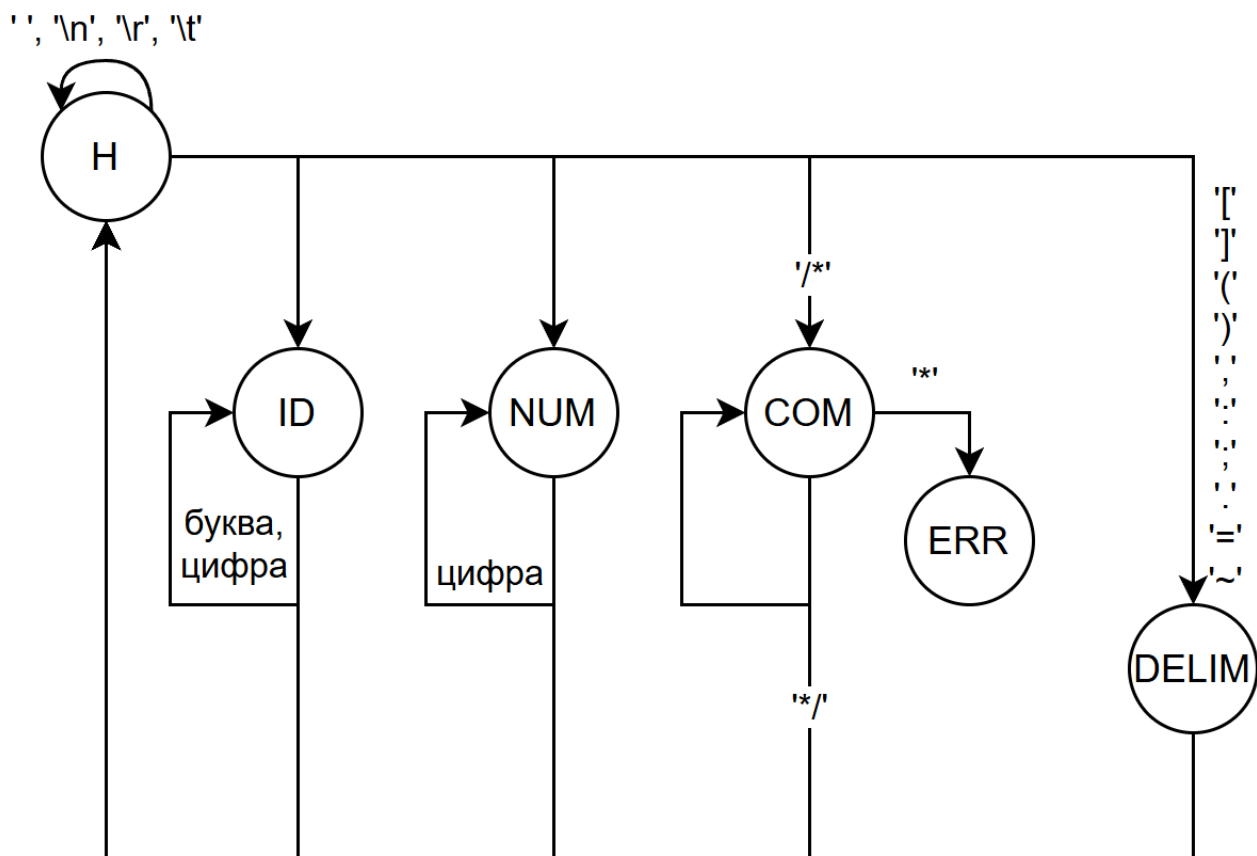


Рисунок 1 – Диаграмма состояний лексического анализатора

5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Будем считать, 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 (parser).

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:

$$P \rightarrow \textit{program } D1; B \perp$$
$$D1 \rightarrow \textit{var } D \{,D\}$$
$$D \rightarrow I \{,I\}:[\textit{integer|boolean}]$$
$$B \rightarrow \textit{begin } S \{; S\} \textit{end}.$$
$$S \rightarrow I = E | \textit{if } E \textit{ then } S \textit{ else } S | \textit{while } E \textit{ do } S | B | \textit{read}(I) | \textit{write}(E)$$
$$E \rightarrow E1 \{[EQ | GT | LT | GE | LE | NE]E1\}$$
$$E1 \rightarrow T \{[+ | - | \textit{or }] T\}$$
$$T \rightarrow F \{[* | / | \textit{and }] F\}$$
$$F \rightarrow I | N | L | \textit{not } F | (E)$$
$$L \rightarrow \textit{true} | \textit{false}$$
$$I \rightarrow C | IC | IR$$
$$N \rightarrow R | NR$$
$$C \rightarrow a | b | \dots | z | A | B | \dots | Z$$
$$R \rightarrow 0 | 1 | \dots | 9$$

Здесь правила для нетерминалов L, I, N, C и R описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов P, D1, D, B, S, E, E1, T, F.

Исходный код синтаксического анализатора приведен в Приложении Б.

6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

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;
- 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.

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 в ту же таблицу заносятся данные о типе идентификатора (поле `type`) и о наличии для него описания (поле `declared`).

С учетом сказанного, правила вывода для нетерминала D (раздел описаний) принимают вид:

$D \rightarrow \text{stack.reset() } I \text{ stack.push(c_val) } \{, I \text{ stack.push(c_val)}\} : [\text{integer dec(LEX_INT)} \mid \text{boolean dec(LEX_BOOL)}]$

Здесь `stack` – структура данных, в которую запоминаются идентификаторы (номера строк в таблице TID), `dec` – функция, задача которой заключается в занесении информации об идентификаторах (поля `type` и `declared`), а также контроль повторного объявления идентификатора.

роль повторного объявления идентификатора. Описания функций семантических проверок приведены в листинге в Приложении Б.

7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

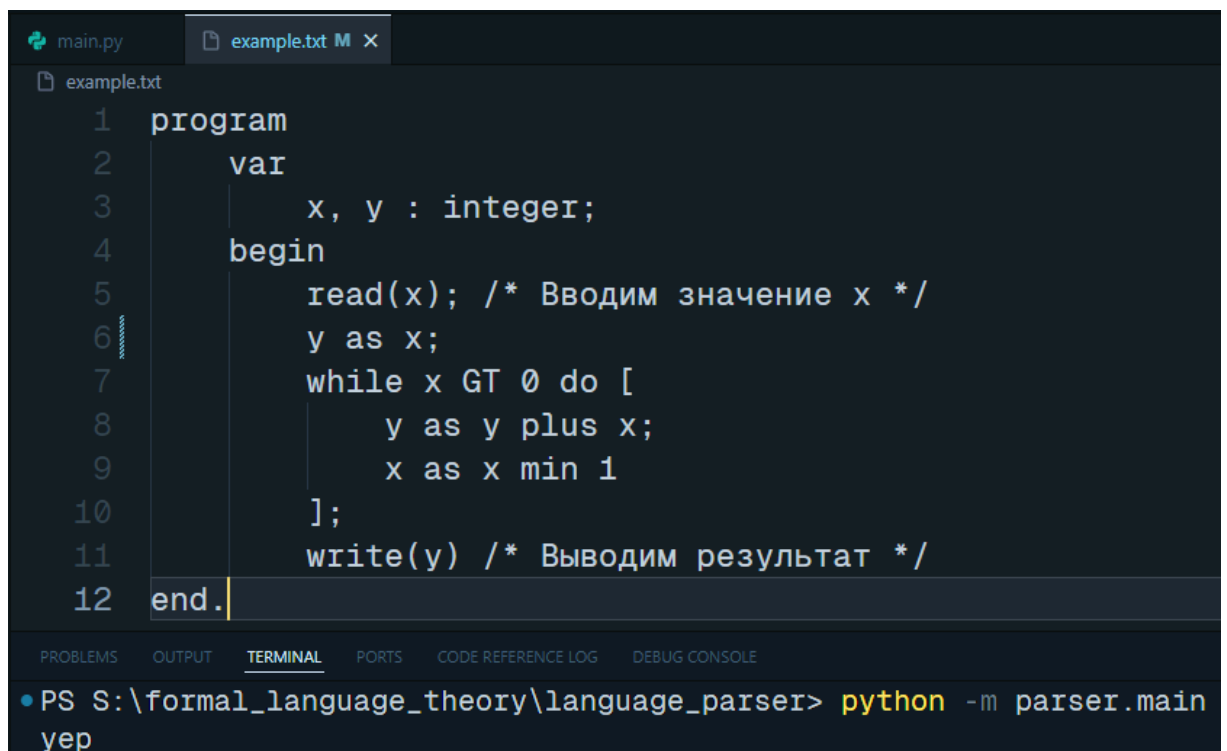
В качестве программного продукта разработано консольное приложение parser.exe, Приложение принимает на вход исходный текст программы на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с номером некорректной лексемы. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1

Листинг 1 – Тестовая программа

```
program
  var
    x, y : integer;
  begin
    read(x); /* Вводим значение x */
    y as x;
    while x GT 0 do [
      y as y plus x;
      x as x min 1
    ];
    write(y) /* Выводим результат */
  end.
```

Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение (рис. 2).



The image shows a screenshot of a code editor with a dark theme. The editor has two tabs: 'main.py' and 'example.txt'. The 'example.txt' tab is active, showing the following code:

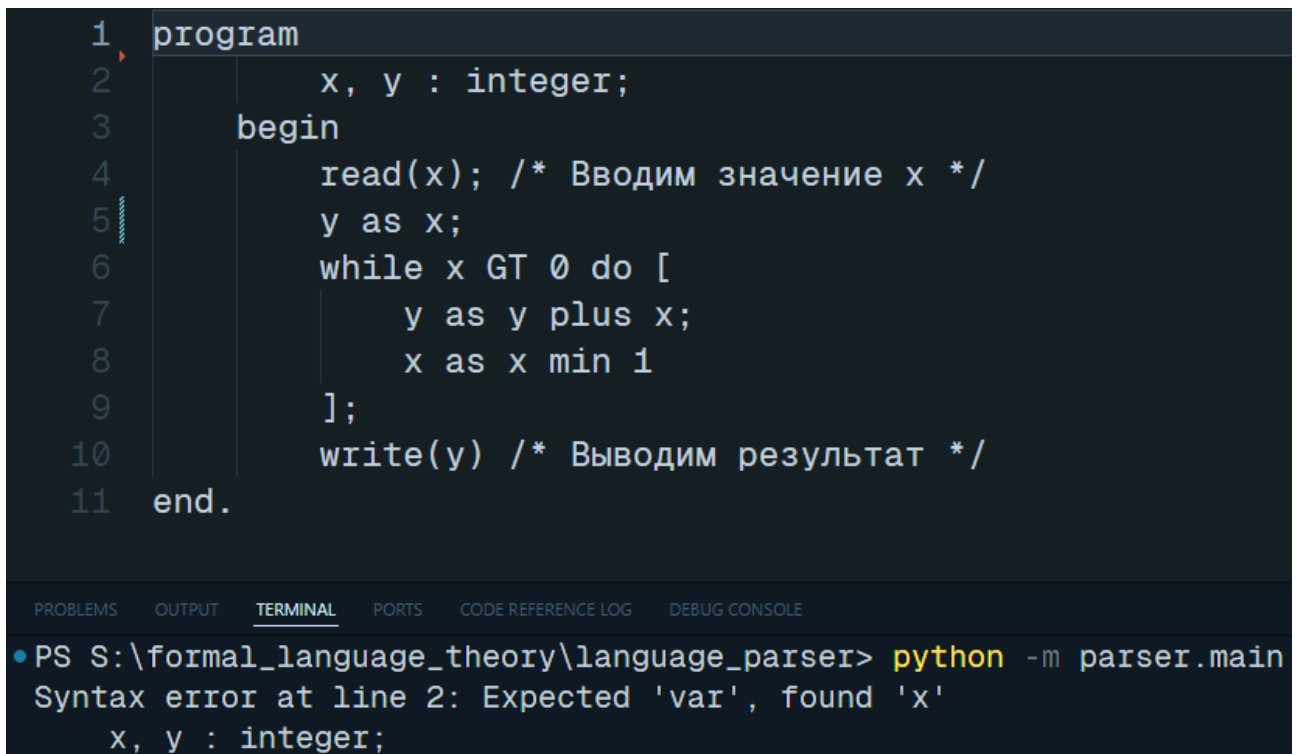
```
1 program
2   var
3     x, y : integer;
4   begin
5     read(x); /* Вводим значение x */
6     y as x;
7     while x GT 0 do [
8       y as y plus x;
9       x as x min 1
10    ];
11    write(y) /* Выводим результат */
12  end.
```

Below the code editor is a terminal window. The terminal shows the command being executed:

```
PS S:\formal_language_theory\language_parser> python -m parser.main
yer
```

Рисунок 2 – Пример синтаксически корректной программы

2. Исходный код программы, содержащий синтаксическую ошибку, приведен на рис. 3 совместно с сообщением об ошибке.



```
1 program
2     x, y : integer;
3 begin
4     read(x); /* Вводим значение x */
5     y as x;
6     while x GT 0 do [
7         y as y plus x;
8         x as x min 1
9     ];
10    write(y) /* Выводим результат */
11 end.
```

PROBLEMS OUTPUT TERMINAL PORTS CODE REFERENCE LOG DEBUG CONSOLE

• PS S:\formal_language_theory\language_parser> python -m parser.main
Syntax error at line 2: Expected 'var', found 'x'
x, y : integer;

Рисунок 3 – Пример программы, содержащей синтаксическую ошибку

Здесь ошибка допущена в строке 2: отсутствует ключевое слово **var** перед объявлением переменных.

3. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь переменная *y* объявлена дважды.

```
1 ~ program
2 ~   var
3 |   x, y, y : integer;
4 ~   begin
5 |     read(x); /* Вводим значение x */
6 |     y as x;
7 ~     while x GT 0 do [
8 |       y as y plus x;
9 |       x as x min 1
10 |     ];
11 |     write(y) /* Выводим результат */
12 end.
```

PROBLEMS OUTPUT TERMINAL PORTS CODE REFERENCE LOG DEBUG CONSOLE

```
• PS S:\formal_language_theory\language_parser> python -m parser.main
Syntax error at line 3: Variable 'y' already declared
x, y, y : integer;
```

Рисунок 4 – Пример программы, содержащей семантическую ошибку

3. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 5 вместе с сообщением об ошибке. Здесь переменная *y* не объявлена, но используется в коде.

```
example.txt
1 program
2   var
3 |   x: integer;
4 |   begin
5 |     read(x); /* Вводим значение x */
6 |     y as x;
7 |     while x GT 0 do [
8 |       y as y plus x;
9 |       x as x min 1
10 |     ];
11 |     write(y) /* Выводим результат */
12 end.
```

PROBLEMS OUTPUT TERMINAL PORTS CODE REFERENCE LOG DEBUG CONSOLE

```
• PS S:\formal_language_theory\language_parser> python -m parser.main
Syntax error at line 6: Variable 'y' not declared
y as x;
```

Рисунок 5 – Пример программы, содержащей семантическую ошибку

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 класса `Lexer`.

Разбором исходного текста программы занимается синтаксический анализатор, который реализован в виде класса `Parser` 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.

В код рекурсивных функций включены проверки дополнительных семантических условий, в частности, проверка на повторное объявление одной и той же переменной.

Тестирование программного продукта показало, что синтаксически и семантически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается.

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.

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 – Класс лексического анализатора

Приложение Б – Класс синтаксического анализатора

Приложение А

Класс лексического анализатора

Листинг А.1 – Lexer.py

```
class Token:
    def __init__(self, table_num, lexeme_num, line, column, value=None):
        self.table_num = table_num # n
        self.lexeme_num = lexeme_num # k
        self.line = line
        self.column = column
        self.value = value

    def __str__(self):
        return f"({self.table_num}, {self.lexeme_num}): {self.value}"

class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.line = 1
        self.column = 1
        self.current_char = self.text[self.pos] if self.pos < len(self.text)
    else None
        self.tokens = []
        self.text_lines = self.text.split("\n")

        # Таблица служебных слов (n = 1)
        self.keywords_table = [
            "program",
            "var",
            "begin",
            "end",
            "if",
            "else",
            "while",
            "for",
            "to",
            "then",
            "do",
            "read",
            "write",
            "true",
            "false",
            "integer",
            "real",
            "boolean",
            "as",
        ]

        # Таблица операторов отношений (n = 2)
        self.rel_op_table = ["NE", "EQ", "LT", "LE", "GT", "GE"]

        # Таблица операций сложения (n = 3)
        self.add_ops_table = ["plus", "min", "or"]

        # Таблица операций умножения (n = 4)
        self.mul_ops_table = ["mult", "div", "and"]

        # Таблица унарных операций (n = 5)
        self.uops_table = ["~"]
```

```
# Таблица разделителей (n = 6)
self.delimiters_table = ["[", "]", "(", ")", ",", ":", ";", ".", "=",
"<", ">"]

# Таблица для чисел (n = 7)
self.numbers_table = []

# Таблица для идентификаторов (n = 8)
self.identifiers_table = []

def advance(self):
    """Переход к следующему символу."""
    if self.current_char == "\n":
        self.line += 1
        self.column = 1
    else:
        self.column += 1

    self.pos += 1
    if self.pos < len(self.text):
        self.current_char = self.text[self.pos]
    else:
        self.current_char = None

def add_token(self, n, k, value=None):
    """Добавление токена в список токенов."""
    self.tokens.append(
        Token(n, k, self.line, self.column - (len(value) if value else 0),
value)
    )

def skip_whitespace(self):
    """Пропуск пробельных символов."""
    while self.current_char and self.current_char.isspace():
        self.advance()

def parse_identifier_or_keyword(self):
    """Разбор идентификаторов и ключевых слов."""
    start = self.pos
    while self.current_char and (
        self.current_char.isalnum() or self.current_char == "_"
    ):
        self.advance()
    text = self.text[start : self.pos]

    # Проверяем, является ли текст служебным словом
    if text.lower() in self.keywords_table:
        self.add_token(1, self.keywords_table.index(text.lower()) + 1,
value=text)
    # Проверяем, является ли текст оператором отношений
    elif text.upper() in self.rel_op_table:
        self.add_token(2, self.rel_op_table.index(text.upper()) + 1,
value=text)
    # Проверяем, является ли текст операцией сложения
    elif text.lower() in self.add_ops_table:
        self.add_token(3, self.add_ops_table.index(text.lower()) + 1,
value=text)
    # Проверяем, является ли текст операцией умножения
    elif text.lower() in self.mul_ops_table:
        self.add_token(4, self.mul_ops_table.index(text.lower()) + 1,
value=text)
```

```
        else:
            # Иначе считаем текст идентификатором
            if text not in self.identifiers_table:
                self.identifiers_table.append(text)
            self.add_token(8, self.identifiers_table.index(text) + 1,
value=text)

    def parse_number(self):
        """Разбор числовых литералов, включая поддержку суффиксов."""
        text = ""
        has_decimal_point = False
        is_float = False

        while self.current_char and (self.current_char.isdigit() or
self.current_char.upper() in 'ABCDEF' or self.current_char == '.'):
            if self.current_char == '.':
                if has_decimal_point:
                    # Две десятичные точки - считаем за идентификатор
                    is_float = False
                    break
                has_decimal_point = True
                is_float = True
            text += self.current_char
            self.advance()

        if self.current_char and self.current_char.upper() == 'E':
            is_float = True
            text += self.current_char
            self.advance()
            if self.current_char in '+-':
                text += self.current_char
                self.advance()
            if self.current_char and self.current_char.isdigit():
                while self.current_char and self.current_char.isdigit():
                    text += self.current_char
                    self.advance()
            else:
                is_float = False

        suffix = ""
        if self.current_char and self.current_char.lower() in 'bohd':
            suffix = self.current_char.lower()
            text += self.current_char
            self.advance()

        # Если после числа идет буква, и это не суффикс системы счисления, то
        # это идентификатор
        if self.current_char is not None and self.current_char.isalpha() and
len(suffix) != 1:
            while self.current_char is not None and self.current_char.isalnum():
                text += self.current_char
                self.advance()
            if text not in self.identifiers_table:
                self.identifiers_table.append(text)
            self.add_token(8, self.identifiers_table.index(text) + 1,
value=text)
            return

        if text not in self.numbers_table:
            self.numbers_table.append(text)
```

```

        self.add_token(7, self.numbers_table.index(text) + 1, value=text)

def parse_comment(self):
    """Разбор комментариев вида /* ... */."""
    self.advance() # Пропускаем '/'
    self.advance() # Пропускаем '*'
    while self.current_char:
        if self.current_char == "*":
            self.advance()
            if self.current_char == "/":
                self.advance()
                return
            else:
                raise Exception(
                    f"Syntax error at line {self.line}: An incomplete
multi-line comment\n    {self.text_lines[self.line - 1].strip()}"
                )
        else:
            self.advance()

def parse_delimiter_or_operator(self):
    """Разбор разделителей и операторов."""
    char = self.current_char
    self.advance()
    text = char

    # Проверка на многосимвольные операторы
    if char == "<" and self.current_char == "=":
        text += self.current_char
        self.advance()
    elif char == ">" and self.current_char == "=":
        text += self.current_char
        self.advance()
    elif char == "a" and self.current_char == "s":
        text += self.current_char
        self.advance()

    # Определение типа токена
    if text.upper() in self.rel_op_table:
        self.add_token(2, self.rel_op_table.index(text.upper()) + 1,
value=text)
    elif text.lower() in self.add_ops_table:
        self.add_token(3, self.add_ops_table.index(text.lower()) + 1,
value=text)
    elif text.lower() in self.mul_ops_table:
        self.add_token(4, self.mul_ops_table.index(text.lower()) + 1,
value=text)
    elif text in self.uops_table:
        self.add_token(5, self.uops_table.index(text) + 1, value=text)
    elif text in self.delimiters_table:
        self.add_token(6, self.delimiters_table.index(text) + 1, value=text)
    else:
        self.add_token(0, 0, value=text) # Неизвестный токен
def tokenize(self):
    """Основной метод лексического анализа."""
    while self.current_char:
        self.skip_whitespace()
        if not self.current_char:
            break

```

Продолжение листинга А.1

```

        if self.current_char == "/" and self.peek() == "*":
            self.parse_comment()
        elif self.current_char.isalpha() or self.current_char == "_":
            self.parse_identifier_or_keyword()
        elif self.current_char.isdigit():
            self.parse_number()
        elif self.current_char in self.delimiters_table or self.current_char
in [
            "<",
            ">",
            ":",
            "!",
            "=",
            "~",
            "+",
            "-",
            "*",
            "/",
            "a",
        ]:
            self.parse_delimiter_or_operator()
        else:
            # Неизвестный символ, можно обработать как ошибку или пропустить
            self.add_token(0, 0, value=self.current_char)
            self.advance()

    self.add_token(0, 0, "EOF") # Добавляем токен конца файла
    return self.tokens

def peek(self):
    """Вспомогательный метод для просмотра следующего символа без его
извлечения."""
    if self.pos + 1 < len(self.text):
        return self.text[self.pos + 1]
    else:
        return None

```

Приложение Б

Класс синтаксического анализатора

Листинг Б.1 – Parser.py

```
class SymbolTable:
    def __init__(self):
        self.symbols = {}
        self.scopes = []

    def enter_scope(self):
        """Вход в новую область видимости."""
        self.scopes.append({})

    def exit_scope(self):
        """Выход из текущей области видимости."""
        self.scopes.pop()

    def define(self, name, type):
        """Определить переменную в текущей области видимости."""
        current_scope = self.scopes[-1]
        if name in current_scope:
            return False # Переменная уже определена в этой области видимости
        current_scope[name] = type
        return True

    def lookup(self, name):
        """Найти переменную в таблице символов."""
        for scope in reversed(self.scopes):
            if name in scope:
                return scope[name]
        return None

class Parser:
    def __init__(self, lexer):
        self.lexer = lexer
        self.tokens = lexer.tokenize() # Получаем список токенов сразу
        self.current_token_index = 0
        self.current_token = (
            self.tokens[self.current_token_index] if self.tokens else None
        )
        self.symbol_table = SymbolTable()
        self.text_lines = lexer.text.split("\n")

        # Словари для быстрого доступа к номерам лексем по их именам
        self.keywords_dict = {
            name: i + 1 for i, name in enumerate(lexer.keywords_table)
        }
        self.rel_op_dict = {name: i + 1 for i, name in
            enumerate(lexer.rel_op_table)}
        self.add_ops_dict = {name: i + 1 for i, name in
            enumerate(lexer.add_ops_table)}
        self.mul_ops_dict = {name: i + 1 for i, name in
            enumerate(lexer.mul_ops_table)}
        self.uops_dict = {name: i + 1 for i, name in
            enumerate(lexer.uops_table)}
        self.delimiters_dict = {
            name: i + 1 for i, name in enumerate(lexer.delimiters_table)
        }
```

```
def advance(self):
    """Переход к следующему токenu."""
    self.current_token_index += 1
    if self.current_token_index < len(self.tokens):
        self.current_token = self.tokens[self.current_token_index]
    else:
        self.current_token = None

def error(self, message, context=None):
    line = self.text_lines[self.current_token.line - 1]
    stript_line = line.strip()
    if context:
        message = f"In rule '{context}', " + message
    raise Exception(
        f"Syntax error at line {self.current_token.line}: {
            message}\n    {stript_line}"
        )

def get_token_name(self, token):
    """Вспомогательная функция для получения имени токена по его типу и
значение."""
    if token.table_num == 0:
        return token.value
    if token.table_num == 1:
        try:
            return self.lexer.keywords_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown keyword"
    elif token.table_num == 2:
        try:
            return self.lexer.rel_op_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown rel_op"
    elif token.table_num == 3:
        try:
            return self.lexer.add_ops_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown add_op"
    elif token.table_num == 4:
        try:
            return self.lexer.mul_ops_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown mul_op"
    elif token.table_num == 5:
        try:
            return self.lexer.uops_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown uop"
    elif token.table_num == 6:
        try:
            return self.lexer.delimiters_table[token.lexeme_num - 1]
        except IndexError:
            return "unknown delimiter"
    elif token.table_num in (7, 8):
        return token.value # Для чисел и идентификаторов выводим их
значение
    else:
        return f"({token.table_num}, {token.lexeme_num})"
def eat(self, table_num, lexeme_num):
    if (
        self.current_token.table_num == table_num
```



```
    ):
        self.advance()
    else:
        # Поиск имени токена для вывода ошибки:
        if table_num == 1:
            expected_token_name = (
                self.lexer.keywords_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.keywords_table)
                else "unknown keyword"
            )
        elif table_num == 2:
            expected_token_name = (
                self.lexer.rel_op_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.rel_op_table)
                else "unknown rel_op"
            )
        elif table_num == 3:
            expected_token_name = (
                self.lexer.add_ops_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.add_ops_table)
                else "unknown add_op"
            )
        elif table_num == 4:
            expected_token_name = (
                self.lexer.mul_ops_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.mul_ops_table)
                else "unknown mul_op"
            )
        elif table_num == 5:
            expected_token_name = (
                self.lexer.uops_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.uops_table)
                else "unknown uop"
            )
        elif table_num == 6:
            expected_token_name = (
                self.lexer.delimiters_table[lexeme_num - 1]
                if lexeme_num - 1 < len(self.lexer.delimiters_table)
                else "unknown delimiter"
            )
        else:
            expected_token_name = f"({table_num}, {lexeme_num})"

        current_token_name = self.get_token_name(self.current_token)

        if (table_num, lexeme_num) == (
            6,
            self.delimiters_dict[";"],
        ):
            # Если ожидался конец составного оператора, но нашелся другой
            # разделитель, указываем на него
            self.error(
                f"Expected '{expected_token_name}', found '{current_token_name}'"
            )
        else:
            self.error(
                f"Expected '{expected_token_name}', found '{current_token_name}'"
            )
```

```

def program(self):
    """
    <программа> ::= program var <описание> begin <оператор> {; <оператор>}
end.
    """
    self.eat(1, self.keywords_dict["program"])
    self.eat(1, self.keywords_dict["var"])
    self.symbol_table.enter_scope() # Входим в глобальную область видимости
    self.description()
    self.eat(1, self.keywords_dict["begin"])
    self.operator_list()
    self.eat(1, self.keywords_dict["end"])
    self.eat(6, self.delimiters_dict["."])
    self.symbol_table.exit_scope() # Выходим из глобальной области
ВИДИМОСТИ
    if not (
        self.current_token.table_num == 0 and self.current_token.lexeme_num
== 0
    ):
        self.error("Expected end of program")

def operator_list(self):
    """
    <оператор> {; <оператор>}
    """
    self.operator_()
    while (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict[";"]
    ):
        self.eat(6, self.delimiters_dict[";"])
        if (
            self.current_token.table_num == 1
            and self.current_token.lexeme_num == self.keywords_dict["end"]
        ):
            return # Обрабатываем END
        self.operator_()

def description(self):
    """
    <описание> ::= {<идентификатор> {, <идентификатор>} : <тип> ;}
    """
    while self.current_token.table_num == 8:
        ids = self.id_list()
        self.eat(6, self.delimiters_dict[":"])
        type_token = self.type()
        for id_token in ids:
            if not self.symbol_table.define(id_token.value, type_token):
                self.error(f"Variable '{id_token.value}' already declared")
        self.eat(6, self.delimiters_dict[";"])

def id_list(self):
    """
    <идентификатор> {, <идентификатор>}
    """
    id_tokens = [self.current_token]
    if not self.current_token.value[0].isalpha():
        self.error(

```

```

        f"Identifier '{self.current_token.value}' must start with a
letter",
        context="id_list",
    )
    self.eat(8, self.current_token.lexeme_num)
    while (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict[","]
    ):
        self.eat(6, self.delimiters_dict[","])
        id_tokens.append(self.current_token)
        if not self.current_token.value[0].isalpha():
            self.error(
                f"Identifier '{self.current_token.value}' must start with a
letter",
                context="id_list",
            )
        self.eat(8, self.current_token.lexeme_num)
    return id_tokens

def type(self):
    """
    <тип> ::= integer | real | boolean
    """
    if (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["integer"]
    ):
        type_token = self.current_token
        self.eat(1, self.keywords_dict["integer"])
        return type_token
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["real"]
    ):
        type_token = self.current_token
        self.eat(1, self.keywords_dict["real"])
        return type_token
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["boolean"]
    ):
        type_token = self.current_token
        self.eat(1, self.keywords_dict["boolean"])
        return type_token
    else:
        self.error("Expected type (integer, real, boolean)")

def operator_(self):
    """
    <оператор> ::= <присваивания> | <условный> | <фиксированного_цикла> |
<условного_цикла> | <составной> | <ввода> | <вывода>
    """
    if self.current_token.table_num == 8:
        self.assignment()
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["if"]
    ):
        self.conditional()
    elif (
        self.current_token.table_num == 1

```

Продолжение листинга Б.1

```

        and self.current_token.lexeme_num == self.keywords_dict["for"]
    ):
        self.fixed_loop()
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["while"]
    ):
        self.conditional_loop()
    elif (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict["["]
    ):
        self.compound()
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["read"]
    ):
        self.input_op()
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["write"]
    ):
        self.output_op()
    else:
        self.error("Expected operator")

def assignment(self):
    """
    <присваивания> ::= <идентификатор> as <выражение>
    """
    id_token = self.current_token
    self.eat(8, self.current_token.lexeme_num)
    if not self.symbol_table.lookup(id_token.value):
        self.error(f"Variable '{id_token.value}' not declared")
    self.eat(1, self.keywords_dict["as"])
    self.expression()

def conditional(self):
    """
    <условный> ::= if <выражение> then <оператор> [ else <оператор> ]
    """
    self.eat(1, self.keywords_dict["if"])
    self.expression()
    self.eat(1, self.keywords_dict["then"])
    self.operator_()
    if (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["else"]
    ):
        self.eat(1, self.keywords_dict["else"])
        self.operator_()

def fixed_loop(self):
    """
    <фиксированного цикла> ::= for <присваивания> to <выражение> do
    <оператор>
    """
    self.eat(1, self.keywords_dict["for"])
    self.symbol_table.enter_scope() # Входим в область видимости цикла
    self.assignment()
    self.eat(1, self.keywords_dict["to"])
    self.expression()

```

Продолжение листинга Б.1

```

self.eat(1, self.keywords_dict["do"])
self.operator_()
self.symbol_table.exit_scope() # Выходим из области видимости цикла

def conditional_loop(self):
    """
    <условного_цикла> ::= while <выражение> do <оператор>
    """
    self.eat(1, self.keywords_dict["while"])
    self.expression()
    self.eat(1, self.keywords_dict["do"])
    self.operator_()

def compound(self):
    """
    <составной> ::= «[» <оператор> { ( : | \n) <оператор> } «]»
    """
    self.eat(6, self.delimiters_dict["["])
    self.symbol_table.enter_scope() # Входим в область видимости составного
оператора
    self.operator_()
    while self.current_token.table_num == 6 and (
        self.current_token.lexeme_num == self.delimiters_dict[":"]
        or self.current_token.lexeme_num == self.delimiters_dict[";"]
    ):
        if self.current_token.lexeme_num == self.delimiters_dict[";"]:
            self.eat(6, self.delimiters_dict[";"])
        else:
            self.eat(6, self.delimiters_dict[":"])
            self.operator_()
    if (
        self.current_token.table_num != 6
        or self.current_token.lexeme_num != self.delimiters_dict[""]
    ):
        self.error(
            f"Expected ']', found '{self.get_token_name(self.current_token)}'",
            context="compound",
        )
    self.symbol_table.exit_scope() # Выходим из области видимости
составного оператора
    self.eat(6, self.delimiters_dict[""])

def input_op(self):
    """
    <ввода> ::= read «(»<идентификатор> {, <идентификатор> } «)»
    """
    self.eat(1, self.keywords_dict["read"])
    self.eat(6, self.delimiters_dict["("])
    id_token = self.current_token
    self.eat(8, self.current_token.lexeme_num)
    if not self.symbol_table.lookup(id_token.value):
        self.error(f"Variable '{id_token.value}' not declared")
    while (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict[","]
    ):
        self.eat(6, self.delimiters_dict[","])
        id_token = self.current_token
    self.eat(8, self.current_token.lexeme_num)
    if not self.symbol_table.lookup(id_token.value):

```

Продолжение листинга Б.1

```

        self.error(f"Variable '{id_token.value}' not declared")
    self.eat(6, self.delimiters_dict[""])

def output_op(self):
    """
    <вывода> ::= write «(<выражение> {, <выражение> } <)>»
    """
    self.eat(1, self.keywords_dict["write"])
    self.eat(6, self.delimiters_dict["("])
    self.expression()
    while (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict[","]
    ):
        self.eat(6, self.delimiters_dict[","])
        self.expression()
    self.eat(6, self.delimiters_dict[")"])

def expression(self):
    """
    <выражение> ::= <сумма> | <выражение> (<>|<|<=<|>|>=) <сумма>
    """
    self.sum()
    while self.current_token.table_num == 2:
        if self.current_token.lexeme_num == self.rel_op_dict["NE"]:
            self.eat(2, self.rel_op_dict["NE"])
        elif self.current_token.lexeme_num == self.rel_op_dict["EQ"]:
            self.eat(2, self.rel_op_dict["EQ"])
        elif self.current_token.lexeme_num == self.rel_op_dict["LT"]:
            self.eat(2, self.rel_op_dict["LT"])
        elif self.current_token.lexeme_num == self.rel_op_dict["LE"]:
            self.eat(2, self.rel_op_dict["LE"])
        elif self.current_token.lexeme_num == self.rel_op_dict["GT"]:
            self.eat(2, self.rel_op_dict["GT"])
        elif self.current_token.lexeme_num == self.rel_op_dict["GE"]:
            self.eat(2, self.rel_op_dict["GE"])
        else:
            break
    self.sum()

def sum(self):
    """
    <сумма> ::= <произведение> { (+ | - | or) <произведение>}
    """
    self.product()
    while self.current_token.table_num == 3:
        if self.current_token.lexeme_num == self.add_ops_dict["plus"]:
            self.eat(3, self.add_ops_dict["plus"])
        elif self.current_token.lexeme_num == self.add_ops_dict["min"]:
            self.eat(3, self.add_ops_dict["min"])
        elif self.current_token.lexeme_num == self.add_ops_dict["or"]:
            self.eat(3, self.add_ops_dict["or"])
        else:
            break
    self.product()

def product(self):
    """
    <произведение> ::= <множитель> { (* | / | and) <множитель>}
    """

```

```

self.multiplier()
while self.current_token.table_num == 4:
    if self.current_token.lexeme_num == self.mul_ops_dict["mult"]:
        self.eat(4, self.mul_ops_dict["mult"])
    elif self.current_token.lexeme_num == self.mul_ops_dict["div"]:
        self.eat(4, self.mul_ops_dict["div"])
    elif self.current_token.lexeme_num == self.mul_ops_dict["and"]:
        self.eat(4, self.mul_ops_dict["and"])
    else:
        break
self.multiplier()

def multiplier(self):
    """
    <множитель> ::= <идентификатор> | <число> | <логическая_константа> | not
    <множитель> | «(><выражение><)>»
    """
    if self.current_token.table_num == 8:
        id_token = self.current_token
        self.eat(8, self.current_token.lexeme_num)
        if not self.symbol_table.lookup(id_token.value):
            self.error(f"Variable '{id_token.value}' not declared")
    elif self.current_token.table_num == 7:
        self.number()
    elif self.current_token.table_num == 1 and (
        self.current_token.lexeme_num == self.keywords_dict["true"]
        or self.current_token.lexeme_num == self.keywords_dict["false"]
    ):
        self.logical_constant()
    elif (
        self.current_token.table_num == 5
        and self.current_token.lexeme_num == self.uops_dict["~"]
    ):
        self.eat(5, self.uops_dict["~"])
        self.multiplier()
    elif (
        self.current_token.table_num == 6
        and self.current_token.lexeme_num == self.delimiters_dict["("]
    ):
        self.eat(6, self.delimiters_dict["("])
        self.expression()
        self.eat(6, self.delimiters_dict[")"])
    else:
        self.error("Expected identifier, number, logical constant, 'not', or
'(')")

def number(self):
    """
    <число> ::= <целое> | <действительное>
    """
    if self.current_token.table_num == 7:
        number_token = self.current_token
        number_value = number_token.value
        self.eat(7, self.current_token.lexeme_num)

        if (
            number_value.endswith("b")
            and number_value.startswith("0")
            and len(number_value) > 2
        ):

```

Продолжение листинга Б.1

```

        self.error(f"Invalid binary number '{number_value}'",
context="number")
        elif (
            number_value.endswith("o")
            and number_value.startswith("0")
            and len(number_value) > 2
        ):
            self.error(f"Invalid octal number '{number_value}'",
context="number")
        elif (
            number_value.endswith("h")
            and number_value.startswith("0")
            and len(number_value) > 2
        ):
            self.error(
                f"Invalid hexadecimal number '{number_value}'",
context="number"
            )
        elif (
            number_value.endswith("d")
            and number_value.startswith("0")
            and len(number_value) > 2
        ):
            self.error(f"Invalid decimal number '{number_value}'",
context="number")
        elif any(
            c not in "01" for c in number_value[:-1]
        ) and number_value.endswith("b"):
            self.error(f"Invalid binary number '{number_value}'",
context="number")
        elif any(
            c not in "01234567" for c in number_value[:-1]
        ) and number_value.endswith("o"):
            self.error(f"Invalid octal number '{number_value}'",
context="number")
        elif any(
            c not in "0123456789abcdefABCDEF" for c in number_value[:-1]
        ) and number_value.endswith("h"):
            self.error(
                f"Invalid hexadecimal number '{number_value}'",
context="number"
            )
        elif any(
            c not in "0123456789" for c in number_value[:-1]
        ) and number_value.endswith("d"):
            self.error(f"Invalid decimal number '{number_value}'",
context="number")
        elif any(c not in "0123456789.eE+-" for c in number_value) and (
            "e" in number_value or "E" in number_value or "." in
number_value
        ):
            self.error(f"Invalid float number '{number_value}'",
context="number")
        elif (
            any(c not in "0123456789" for c in number_value)
            and not (
                "e" in number_value or "E" in number_value or "." in
number_value
            )
            and (number_value[-1] not in "bohhd")
        ):

```

Продолжение листинга Б.1


```

        self.error(f"Invalid decimal number '{number_value}'",
context="number")
    else:
        self.error("Expected number")

def logical_constant(self):
    """
    <логическая_константа> ::= true | false
    """
    if (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["true"]
    ):
        self.eat(1, self.keywords_dict["true"])
    elif (
        self.current_token.table_num == 1
        and self.current_token.lexeme_num == self.keywords_dict["false"]
    ):
        self.eat(1, self.keywords_dict["false"])
    else:
        self.error("Expected 'true' or 'false'")

def parse(self):
    """Запуск синтаксического анализа."""
    self.program()
    if not (
        self.current_token.table_num == 0 and self.current_token.lexeme_num
== 0
    ):
        self.error("Expected end of program")

```