

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
1 ПОСТАНОВКА ЗАДАЧИ.....	5
2 ПОРЯДОК ВЫПОЛНЕНИЯ .....	6
3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА.....	7
4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА.....	10
5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА.....	12
6 СЕМАНТИЧЕСКИЙ АНАЛИЗ .....	13
7 ТЕСТИРОВАНИЕ ПРОГРАММЫ.....	14
ЗАКЛЮЧЕНИЕ .....	18
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	19
ПРИЛОЖЕНИЯ.....	20

# 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 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.

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.

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

## 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 помощи 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:

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{операции\_группы\_отношения} \rangle \langle \text{операнд} \rangle \}$

$\langle \text{операнд} \rangle ::= \langle \text{слагаемое} \rangle \{ \langle \text{операции\_группы\_сложения} \rangle \langle \text{слагаемое} \rangle \}$

$\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{операции\_группы\_умножения} \rangle \langle \text{множитель} \rangle \}$

$\langle \text{множитель} \rangle ::= \langle \text{идентификатор} \rangle \mid \langle \text{число} \rangle \mid \langle \text{логическая\_константа} \rangle \mid$

$\langle \text{унарная\_операция} \rangle \langle \text{множитель} \rangle \mid \langle \text{«} \rangle \langle \text{«} \rangle \langle \text{выражение} \rangle \langle \text{«} \rangle \langle \text{«} \rangle$

$\langle \text{число} \rangle ::= \langle \text{целое} \rangle \mid \langle \text{действительное} \rangle$

$\langle \text{логическая\_константа} \rangle ::= \text{true} \mid \text{false}$

$\langle \text{идентификатор} \rangle ::= \langle \text{буква} \rangle \{ \langle \text{буква} \rangle \mid \langle \text{цифра} \rangle \}$

$\langle \text{буква} \rangle ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K$   
 $\mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid$   
 $Z \mid a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n$   
 $\mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$

$\langle \text{цифра} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{целое} \rangle ::= \langle \text{двоичное} \rangle \mid \langle \text{восьмеричное} \rangle \mid \langle \text{десятичное} \rangle \mid \langle \text{шес-}$   
 $\text{тнадцатеричное} \rangle$

$\langle \text{двоичное} \rangle ::= \{ / 0 \mid 1 / \} (B \mid b)$

$\langle \text{восьмеричное} \rangle ::= \{ / 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 / \}$   
 $(O \mid o)$

$\langle \text{десятичное} \rangle ::= \{ / \langle \text{цифра} \rangle / \} [D \mid d]$

$\langle \text{шестнадцатеричное} \rangle ::= \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \mid A \mid B \mid C \mid$   
 $D \mid E \mid F \mid a \mid b \mid c \mid d \mid e \mid f \} (H \mid h)$

$\langle \text{действительное} \rangle ::= \langle \text{числовая\_строка} \rangle \langle \text{порядок} \rangle \mid$   
 $[\langle \text{числовая\_строка} \rangle] . \langle \text{числовая\_строка} \rangle [\langle \text{порядок} \rangle]$   
 $\langle \text{числовая\_строка} \rangle ::= \{ / \langle \text{цифра} \rangle /\}$   
 $\langle \text{порядок} \rangle ::= ( E \mid e ) [ + \mid - ] \langle \text{числовая\_строка} \rangle$   
 Операции языка -  $\langle \text{операции\_группы\_отношения} \rangle ::= != \mid$   
 $= \mid < \mid <= \mid > \mid >=$   
 $\langle \text{операции\_группы\_сложения} \rangle ::= + \mid - \mid \mid \mid$   
 $\langle \text{операции\_группы\_умножения} \rangle ::= * \mid / \mid \&\&$   
 $\langle \text{унарная\_операция} \rangle ::= \text{not}$   
 Структура программы -  $\langle \text{программа} \rangle ::= \langle \{ \rangle \{ /$   
 $(\langle \text{описание} \rangle \mid \langle \text{оператор} \rangle) ; / \} \langle \} \rangle$   
 Синтаксис команд описания данных -  $\langle \text{описание} \rangle ::= \langle \text{тип} \rangle$   
 $\langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \}$   
 Описание типов (в порядке следования: целый, действительный, логический) -  $\langle \text{тип} \rangle ::= \% \mid ! \mid \$$   
 Синтаксис оператора -  $\langle \text{составной} \rangle ::= \text{begin} \langle \text{оператор} \rangle$   
 $\{ ; \langle \text{оператор} \rangle \} \text{end}$   
 Оператор присваивания -  $\langle \text{присваивания} \rangle ::=$   
 $\langle \text{идентификатор} \rangle := \langle \text{выражение} \rangle$   
 Оператор условного перехода -  $\langle \text{условный} \rangle ::= \text{if}$   
 $\langle ( \rangle \langle \text{выражение} \rangle \langle ) \rangle \langle \text{оператор} \rangle [\text{else} \langle \text{оператор} \rangle]$   
 Синтаксис оператора цикла с фиксированным числом повторений -  $\langle \text{фиксированного\_цикла} \rangle ::= \text{for} \langle \text{присваивания} \rangle$   
 $\text{to} \langle \text{выражение} \rangle [\text{step} \langle \text{выражение} \rangle ] \langle \text{оператор} \rangle \text{next}$   
 Синтаксис условного оператора цикла -  $\langle \text{условного\_цикла} \rangle ::= \text{while} \langle ( \rangle \langle \text{выражение} \rangle \langle ) \rangle \langle \text{оператор} \rangle$   
 Синтаксис оператора ввода -  $\langle \text{ввода} \rangle ::= \text{readln}$   
 $\langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \}$   
 Синтаксис оператора вывода -  $\langle \text{вывода} \rangle ::= \text{writeln}$   
 $\langle \text{выражение} \rangle \{ , \langle \text{выражение} \rangle \}$

Синтаксис многострочных комментариев-Начало ::= /\*  
Конец ::= \*/

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 конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата (рис. 1).

Исходные код лексического анализатора приведен в Приложении А.



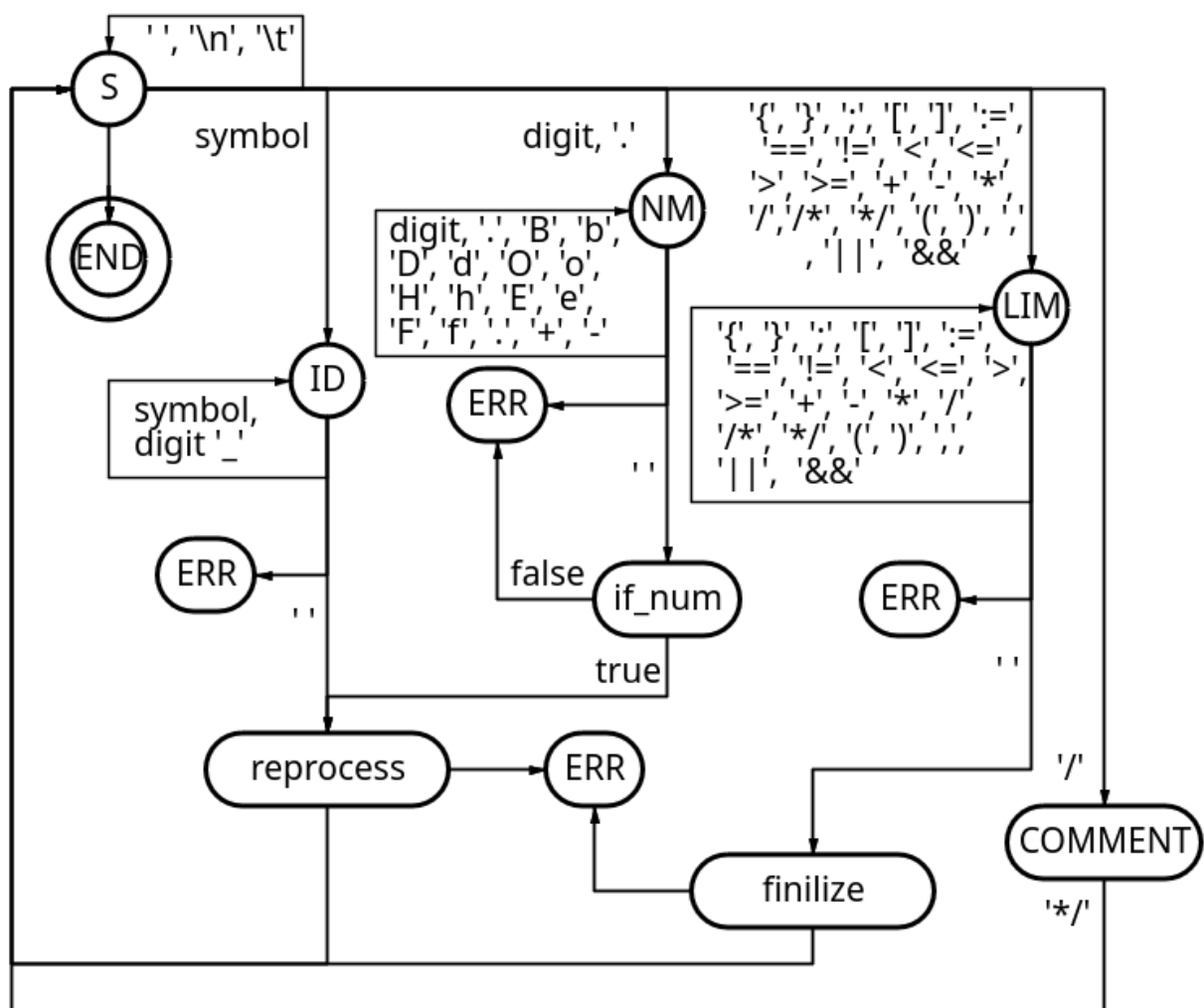


Рисунок 1 – Диаграмма состояний лексического анализатора

## 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:

$$RP \rightarrow \{ D1 B \}$$

$$D1 \rightarrow \{ D \}^*$$

$$D \rightarrow \% IN ; | ! IN ; | \$ IN ;$$

$$B \rightarrow S1$$

$$S1 \rightarrow \{ S \}^*$$

$$S \rightarrow I = E ; | \# H ;$$

$$E \rightarrow E + T | E - T | T$$

$$T \rightarrow T * F | T / F | F$$

$$F \rightarrow ( E ) | N | I$$

$$H \rightarrow ( L )$$

$$L \rightarrow E \{ , E \}^*$$

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:

- проверка наличия повторного объявления идентификаторов;
- проверка наличия неопределенных идентификаторов;
- проверка совпадения типов выражений и переменных;
- проверка наличия логического выражения в качестве условия.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. Рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в таблицу `symbol_table` заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы.

С учетом сказанного, правила вывода для нетерминала `D` (раздел описаний) можно представить следующим образом:

```
D → symbol_table.clear() I symbol_table.add(identifier) { , I symbol_table.add(identifier) } : [ % dec('int') | ! dec('bool') | $ dec('string') ]
```

Где `symbol_table` — структура данных, используемая для хранения идентификаторов и их типов. `dec` — функция, которая заносит информацию об идентификаторах в `symbol_table` (поля `type` и `declared`) и контролирует повторное объявление идентификаторов. Таким образом, при парсинге раздела описаний парсер очищает таблицу символов, читает идентификаторы `I` и добавляет их в `symbol_table`, обрабатывает возможные повторные идентификаторы в списке через запятую и в зависимости от символа устанавливает тип переменной.

Описания функций семантических проверок приведены в листинге в Приложении Б.

## 7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработан скрипт для запуска продукта. Приложение принимает на вход файл 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 строки и символа ошибки, а так же её краткое описание. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1. Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение (рис. 2).

*Листинг 1 – Тестовая программа*

```
{ % xa ; % y , x ; ! z ;  
    /* ab abab */  
    begin  
        x := 10 ;  
        if ( x <= 5 ) begin  
            writeln ( x ) ;  
        end  
        else begin  
            y := 2 ;  
        end ;  
    end  
}
```



**Рисунок 2 – Пример синтаксически корректной программы**

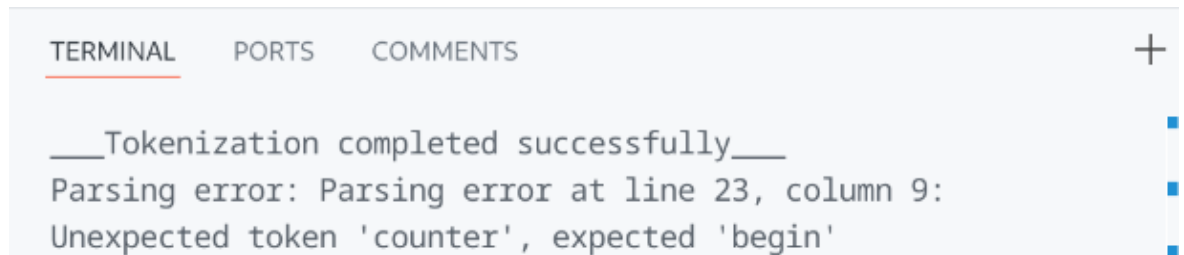
2. Исходный код программы приведен в листинге 2. Вывод парсера, приведен на рис. 3 совместно с сообщением об ошибке.

*Листинг 2 – Тестовая программа*

```
{ % counter ; % limit ; % result ;  
    begin  
        counter := 0 ;  
        limit := 10 ;
```

### Продолжение листинга 2

```
        result := 1 ;
        while ( counter < limit )
            counter := counter + 1 ;
            result := result * counter ;
        end ;
        writeln ( result ) ;
    end
}
```



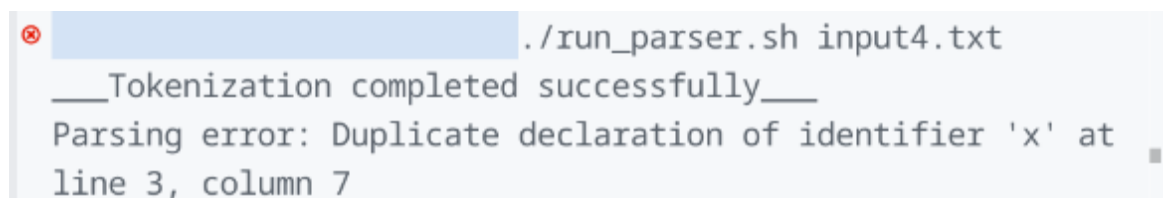
**Рисунок 3 – Пример программы, содержащей синтаксическую ошибку**

Здесь ошибка допущена в строке 23 столбце 9: отсутствует открывающее `begin` после объявления `while`. В сообщении об ошибке указан адрес ошибки, а так же ожидаемая лексема `begin`, вместо полученной `counter`.

3. Исходный код программы с семантической ошибкой приведен в листинге 3 и вывод парсера на рисунке 4.

### Листинг 3 – Тестовая программа

```
{
    % x ;
    % x ;
    begin
        x := 5 ;
    end
}
```

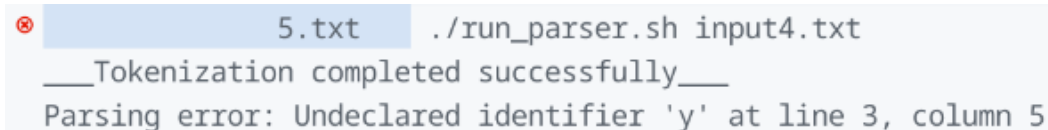


**Рисунок 4 – Пример программы, содержащей семантическую ошибку 1**

4. Исходный код программы с семантической ошибкой приведен в листинге 4 и вывод парсера на рисунке 5.

*Листинг 4 – Тестовая программа*

```
{ % x ;  
  begin  
    y := 10 ;  
  end  
}
```



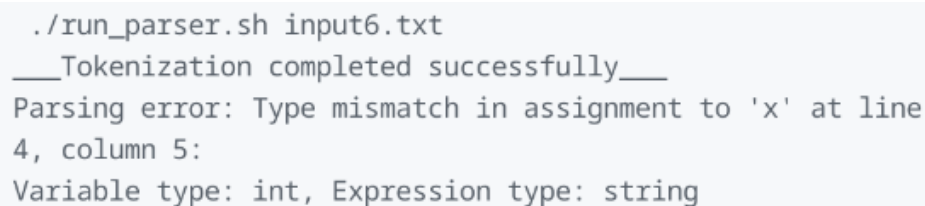
5.txt ./run\_parser.sh input4.txt  
\_\_Tokenization completed successfully\_\_  
Parsing error: Undeclared identifier 'y' at line 3, column 5

**Рисунок 5 – Пример программы, содержащей семантическую ошибку 2**

5. Исходный код программы с семантической ошибкой приведен в листинге 5 и вывод парсера на рисунке 6.

*Листинг 5 – Тестовая программа*

```
{ % x ;  
  $ y ;  
  begin  
    x := y ;  
  end  
}
```



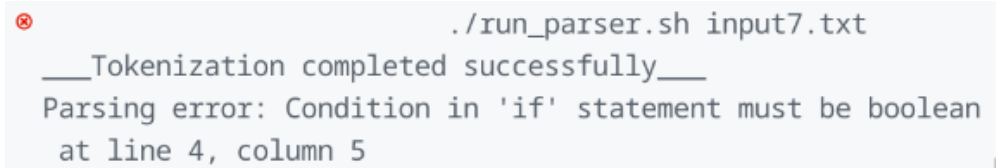
./run\_parser.sh input6.txt  
\_\_Tokenization completed successfully\_\_  
Parsing error: Type mismatch in assignment to 'x' at line 4, column 5:  
Variable type: int, Expression type: string

**Рисунок 6 – Пример программы, содержащей семантическую ошибку 3**

6. Исходный код программы с семантической ошибкой приведен в листинге 6 и вывод парсера на рисунке 7.

*Листинг 6 – Тестовая программа*

```
{ % y , x ;  
  begin  
    x := 10 ;  
    if ( x ) begin  
      writeln ( x ) ;  
    end ;  
  end  
}
```

A terminal window with a light blue background. It shows the command `./run_parser.sh input7.txt` being executed. The output consists of two lines: `___Tokenization completed successfully___` and `Parsing error: Condition in 'if' statement must be boolean at line 4, column 5`. A red 'x' icon is visible in the top left corner of the terminal window.

```
./run_parser.sh input7.txt
___Tokenization completed successfully___
Parsing error: Condition in 'if' statement must be boolean
at line 4, column 5
```

**Рисунок 7 – Пример программы, содержащей семантическую ошибку 4**

## 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*

```
import sys

class LexerError(Exception):
    pass

class Lexer:
    def __init__(self, filepath):
        self.filepath = filepath
        self.tokens = []
        self.state = 'S'
        self.word = ''
        self.letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_"
        self.digits = "0123456789"
        self.service_words = ['while', 'readln', 'for', 'to', 'step',
                              'next',
                              '%', '!', '$', 'writeln', 'if', 'else',
                              'true', 'false', 'begin', 'end', 'not']
        self.limiters = ['{', '}', ';', '[', ']', ':=', '==', '!=', '<',
                          '<=', '>', '>=',
                          '+', '-', '*', '/', '/*', '*/', '(', ')', ',',
                          '||', '&&']
        self.identifiers = []
        self.numbers = []
        self.outputs = []
        self.backtrack = False
        self.last_char_was_star = False

    def tokenize(self):
        with open(self.filepath, 'r') as file:
            line_num = 1
            while True:
                line = file.readline()
                if not line:
                    break
                i = 0
                line_length = len(line)
                while i < line_length:
                    char = line[i]
                    proceed = self.process_char(char, line_num, i + 1)
```

```
        if self.backtrack:
            self.backtrack = False
            # Do not increment 'i', reprocess this character
        else:
            i += 1
            line_num += 1
        # Handle any remaining word
        if self.word:
            self.finalize_token(line_num, 1) # Assume column 1 at end

def process_char(self, char, line_num, col_num):
    match self.state:
        case 'S':
            if char in [' ', '\t', '\n']:
                pass # Ignore whitespace
            elif char in self.service_words and len(char) == 1:
                # Directly add as service word
                group = 1
                index = self.get_service_word_index(char)
                self.add_token(group, index, line_num, col_num, char)
            elif char.isalpha() or char == '_':
                self.word += char
                self.state = 'ID'
                self.token_start = (line_num, col_num)
            elif char.isdigit() or char == '.':
                self.word += char
                self.state = 'NM'
                self.token_start = (line_num, col_num)
            elif char == '/':
                self.state = 'DIV_OR_COMMENT'
            else:
                self.word += char
                self.state = 'LIM'
                self.token_start = (line_num, col_num)
        case 'ID':
            if char.isalnum() or char == '_':
                self.word += char
            else:
                self.reprocess_char(char, line_num, col_num)
        case 'NM':
            if char.isdigit() or char == '.':
```

```

        self.word += char
    else:
        self.reprocess_char(char, line_num, col_num)
    case 'LIM':
        potential_two_char = self.word + char
        if potential_two_char in self.limiters:
            self.word = potential_two_char
            self.state = 'LIM2'
            self.token_start = (line_num, col_num - 1)
        else:
            self.reprocess_char(char, line_num, col_num)
    case 'LIM2':
        self.finalize_token(line_num, col_num)
    case 'COMMENT':
        if self.last_char_was_star and char == '/':
            self.state = 'S'
            self.word = ''
            self.last_char_was_star = False
        elif char == '*':
            self.last_char_was_star = True
        else:
            self.last_char_was_star = False
            # Stay in COMMENT
    case 'DIV_OR_COMMENT':
        if char == '*':
            self.state = 'COMMENT'
            self.last_char_was_star = False
        else:
            self.add_token(2, self.get_limiter_index(
                '/'), line_num, self.token_start[1], '/')
            self.reprocess_char(char, line_num, col_num)
    return True

def reprocess_char(self, char, line_num, col_num):
    # Push back the character for next token processing
    self.finalize_token(line_num, col_num - 1)
    self.backtrack = True

def finalize_limiter(self, line_num, col_num):
def finalize_token(self, line_num, col_num):
def add_token(self, group, index, line_num, col_num, word)

```

```
self.tokens.append((group, index, line_num, col_num, word))

def add_identifier(self, identifier):
    if identifier not in self.identifiers:
        self.identifiers.append(identifier)
    return self.identifiers.index(identifier) + 1

def add_number(self, number):
    if number not in self.numbers:
        self.numbers.append(number)
    return self.numbers.index(number) + 1

def is_number(self, word):
    try:
        float(word)
        return True

def get_service_word_index(self, word):
    if word in self.service_words:
        return self.service_words.index(word) + 1
    raise ValueError(f"Unknown service word: {word}")

def get_limiter_index(self, limiter):
    if limiter in self.limiters:
        return self.limiters.index(limiter) + 1
    raise ValueError(f"Unknown limiter: {limiter}")

def save_tokens(self, output_path):
    with open(output_path, 'w') as outfile:
        # Save identifiers
        outfile.write("Identifiers:\n")
        for identifier in self.identifiers:
            outfile.write(f"{identifier}\n")
        outfile.write("\nNumbers:\n")
        # Save numbers
        for number in self.numbers:
            outfile.write(f"{number}\n")
        outfile.write("\nTokens with Words and Positions:\n")
        # Save tokens as (group, index, line_num, col_num, word)
        for token in self.tokens:
            outfile.write(f"{token}\n")

def main():
    if len(sys.argv) != 2:
        print("Usage: python lecser2.py <source_file>")
        sys.exit(1)
    source_file = sys.argv[1]
```

*Окончание листинга A.1*

```
output_token_file = "lecsems2.txt"
lexer = Lexer(source_file)
try:
    lexer.tokenize()
    lexer.save_tokens(output_token_file)
    print("__Tokenization completed successfully__")
except LexerError as e:
    print(f"Lexer error: {e}")
    sys.exit(1)

if __name__ == "__main__":
    main()
```

## Приложение Б

### Класс синтаксического анализатора

Листинг Б.1 – Parser.py

```
import sys

class ParserError(Exception):
    pass

class Parser:
    def __init__(self, token_file):
        self.identifiers = []
        self.numbers = []
        self.tokens = []
        self.current = 0
        self.symbol_table = {} # For semantic checks
        # Declaration symbols to types
        self.declaration_types = {'%': 'int', '!!': 'float', '$': 'string'}
        self.load_tokens(token_file)

        # Define service words and limiters as per Lexer
        self.service_words = ['while', 'readln', 'for', 'to', 'step',
                              'next',
                              '%', '!!', '$', 'writeln', 'if', 'else',
                              'true', 'false', 'begin', 'end', 'not', 'do']
        self.limiters = ['{', '}', ';', '[', ']', ':=', '==', '!=', '<',
                        '<=', '>', '>=',
                        '+', '-', '*', '/', '/*', '*/', '(', ')', ',',
                        '||', '&&']

    def load_tokens(self, token_file):
    def current_token(self):

    def get_word(self, token):
        group, index, line_num, col_num, word = token
        if group == 1:
            return self.service_words[index - 1]
        elif group == 2:
            return self.limiters[index - 1]
        elif group == 3:
            if 0 < index <= len(self.identifiers):
                return self.identifiers[index - 1]
```



```

{index}")

        elif group == 4:
            if 0 < index <= len(self.numbers):
                return self.numbers[index - 1]
            else:
                raise ParserError(f"Number index out of range: {index}")
        else:
            return "Unknown"

    def consume(self, expected_group, expected_index):
        token = self.current_token()
        if token is None:
            raise ParserError("Unexpected end of input")
        group, index, line_num, col_num, word = token
        if group != expected_group or index != expected_index:
            expected_word = self.get_expected_word(
                expected_group, expected_index)
            raise ParserError(
                f"Parsing error at line {line_num}, column {col_num}:\n"
                f"Unexpected token '{self.get_word(token)}', expected '{'
                    expected_word}'"
            )
        self.current += 1
        return token

    def match(self, expected_group, expected_index):
        token = self.current_token()
        if token and token[0] == expected_group and token[1] ==
expected_index:
            self.current += 1
            return True
        return False

    def analyze(self):
        self.P()
        if self.current < len(self.tokens):
            token = self.current_token()
            if len(token) == 5:
                group, index, line_num, col_num, word = token
                remaining = self.get_word(token)
                raise ParserError(
                    f"Extra tokens after parsing complete at line {

```

```

        line_num}, column {col_num}:\n"
        f"Unexpected token '{remaining}'"
    )
else:
    raise ParserError("Extra tokens after parsing complete")

# Grammar Rules
#  $P \rightarrow \{ D1 B \}$ 
def P(self):
    self.consume(2, self.get_limiter_index('{'))
    self.D1()
    self.B()
    self.consume(2, self.get_limiter_index('}'))

#  $D1 \rightarrow \{ D \}^*$ 
def D1(self):
    while True:
        token = self.current_token()
        if token and token[0] == 1 and token[1] in [
            self.get_service_word_index('%'),
            self.get_service_word_index('!'),
            self.get_service_word_index('$')
        ]:
            self.D()
        else:
            break

#  $D \rightarrow \text{DeclarationWord } I \{ , I \} ;$ 
def D(self):
    token = self.current_token()
    if token and token[0] == 1 and token[1] in [
        self.get_service_word_index('%'),
        self.get_service_word_index('!'),
        self.get_service_word_index('$')
    ]:
        decl_word = self.get_word(token)
        var_type = self.declaration_types[decl_word]
        self.consume(token[0], token[1])
        identifier_token = self.current_token()
        if identifier_token and identifier_token[0] == 3:
            identifier = self.get_word(identifier_token)
            if identifier in self.symbol_table:

```

```

        line_num, col_num = identifier_token[2], identifier_token[3]
        raise ParserError(
            f"Duplicate declaration of identifier '{
                identifier}' at line {line_num}, column
{col_num}"
        )
        self.symbol_table[identifier] = var_type
        self.consume(3, identifier_token[1])
    else:
        self.I()
    while self.match(2, self.get_limiter_index(',')):
        identifier_token = self.current_token()
        if identifier_token and identifier_token[0] == 3:
            identifier = self.get_word(identifier_token)
            if identifier in self.symbol_table:
                line_num, col_num = identifier_token[2],
identifier_token[3]
                raise ParserError(
                    f"Duplicate declaration of identifier '{
                        identifier}' at line {line_num}, column
{col_num}"
                )
            self.symbol_table[identifier] = var_type
            self.consume(3, identifier_token[1])
        else:
            self.I()
        self.consume(2, self.get_limiter_index(';'))
    else:
        raise ParserError("Expected variable declaration")
# B → begin { S ; } end
def B(self):
    self.consume(1, self.get_service_word_index('begin'))
    while True:
        token = self.current_token()
        if token and (token[0] == 1 and token[1] ==
self.get_service_word_index('end')):
            break
        self.S()
        self.consume(2, self.get_limiter_index(';'))
    self.consume(1, self.get_service_word_index('end'))

# S → I := E | if E B else B | while E B | readln ( I { , I } ) |

```

*Продолжение листинга Б.1*

```
writeln ( E { , E } ) | B

def S(self):
    token = self.current_token()
    if token is None:
        raise ParserError("Unexpected end of input in statement")

    group, index, line_num, col_num, word = token

    if group == 3: # Identifier
        identifier = self.get_word(token)
        var_type = self.get_identifier_type(identifier, line_num,
col_num)
        self.consume(3, token[1])
        self.consume(2, self.get_limiter_index(':='))
        expr_type = self.E()
        if not self.types_compatible(var_type, expr_type):
            raise ParserError(
                f"Type mismatch in assignment to '{
                identifier}' at line {line_num}, column
{col_num}:\n"
                f"Variable type: {var_type}, Expression type:
{expr_type}"
            )
    elif group == 1 and index == self.get_service_word_index('if'):
        self.consume(1, self.get_service_word_index('if'))
        condition_type = self.E()
        if condition_type != 'boolean':
            raise ParserError(
                f"Condition in 'if' statement must be boolean at line
{
                line_num}, column {col_num}"
            )
        self.B()
        if self.match(1, self.get_service_word_index('else')):
            self.B()
    elif group == 1 and index == self.get_service_word_index('while'):
        self.consume(1, self.get_service_word_index('while'))
        condition_type = self.E()
        if condition_type != 'boolean':
            raise ParserError(
                f"Condition in 'while' statement must be boolean at
line {
                line_num}, column {col_num}"
            )
    )
```

```
self.B()

    elif group == 1 and index == self.get_service_word_index('for'):
        self.consume(1, self.get_service_word_index('for'))
        self.I()
        self.consume(2, self.get_limiter_index(':='))
        self.E()
        self.consume(1, self.get_service_word_index('to'))
        condition_type = self.E()
        if condition_type != 'boolean':
            raise ParserError(
                f"Condition in 'for' statement must be integer at line
{
                line_num}, column {col_num}")
        self.consume(1, self.get_service_word_index('step'))
        self.I()
        self.consume(2, self.get_limiter_index(':='))
        self.E()
        self.B()
        self.consume(1, self.get_service_word_index('next'))
    elif group == 1 and index ==
self.get_service_word_index('readln'):
        self.consume(1, self.get_service_word_index('readln'))
        self.consume(2, self.get_limiter_index('('))
        identifier = self.expect_identifier()
        while self.match(2, self.get_limiter_index(',')'):
            identifier = self.expect_identifier()
        self.consume(2, self.get_limiter_index(')'))
    elif group == 1 and index ==
self.get_service_word_index('writeln'):
        self.consume(1, self.get_service_word_index('writeln'))
        self.consume(2, self.get_limiter_index('('))
        self.E()
        while self.match(2, self.get_limiter_index(',')'):
            self.E()
        self.consume(2, self.get_limiter_index(')'))
    elif group == 1 and index == self.get_service_word_index('begin'):
        self.B()
    else:
        raise ParserError(
            f"Invalid statement starting with token
'{self.get_word(token)}' at line {
            line_num}, column {col_num}"
```

```

    )

    # E → E1 [ RelationOp E1 ]
    def E(self):
        type_e1 = self.E1()
        token = self.current_token()
        relation_ops = ['<', '>', '<=', '>=', '==', '!=']
        relation_indices = [self.get_limiter_index(op) for op in
relation_ops]
        if token and token[0] == 2 and token[1] in relation_indices:
            operator = self.get_word(token)
            self.consume(token[0], token[1])
            type_e2 = self.E1()
            if not self.types_compatible(type_e1, type_e2):
                raise ParserError(
                    f"Type mismatch in relation operation '{
                        operator}' at line {token[2]}, column {token[3]}"
                )
            return 'boolean'
        else:
            return type_e1
    # E1 → T { [ + | - | || ] T }
    def E1(self):
        type_t = self.T()
        while True:
            token = self.current_token()
            if token and token[0] == 2 and token[1] in [
                self.get_limiter_index('+'),
                self.get_limiter_index('-')]:
                operator = self.get_word(token)
                self.consume(token[0], token[1])
                type_t2 = self.T()
                type_t = self.combine_types(type_t, type_t2, operator)
            elif token and token[0] == 2 and token[1] ==
self.get_limiter_index('||'):
                operator = self.get_word(token)
                self.consume(token[0], token[1])
                type_t2 = self.T()
                if type_t != 'boolean' or type_t2 != 'boolean':
                    raise ParserError(
                        f"Logical operator '{

```

```

        operator}' requires boolean operands"
    )
    type_t = 'boolean'
else:
    break
return type_t

# T → F { [ * | / | && ] F }
def T(self):
    type_f = self.F()
    while True:
        token = self.current_token()
        if token and token[0] == 2 and token[1] in [
            self.get_limiter_index('*'),
            self.get_limiter_index('/')
        ]:
            operator = self.get_word(token)
            self.consume(token[0], token[1])
            type_f2 = self.F()
            type_f = self.combine_types(type_f, type_f2, operator)
        elif token and token[0] == 2 and token[1] == self.get_limiter_index('&&'):
            operator = self.get_word(token)
            self.consume(token[0], token[1])
            type_f2 = self.F()
            if type_f != 'boolean' or type_f2 != 'boolean':
                raise ParserError(
                    f"Logical operator '{
                        operator}' requires boolean operands"
                )
            type_f = 'boolean'
        else:
            break
    return type_f

# F → I | N | L | not F | ( E )
def F(self):
    token = self.current_token()
    if token is None:
        raise ParserError("Unexpected end of input in expression")
    group, index, line_num, col_num, word = token
    if group == 3: # Identifier

```

```

        identifier = self.get_word(token)
        var_type = self.get_identifier_type(identifier, line_num,
col_num)
        self.consume(3, token[1])
        return var_type
    elif group == 4: # Number
        num_value = self.get_word(token)
        if '.' in num_value:
            num_type = 'float'
        else:
            num_type = 'int'
        self.consume(4, token[1])
        return num_type
    elif group == 1 and index in [
        self.get_service_word_index('true'),
        self.get_service_word_index('false')
    ]:
        self.L()
        return 'boolean'
    elif group == 1 and index == self.get_service_word_index('not'):
        self.consume(1, self.get_service_word_index('not'))
        type_f = self.F()
        if type_f != 'boolean':
            raise ParserError(
                f"Operator 'not' requires boolean operand at line {
                    line_num}, column {col_num}")
        return 'boolean'
    elif group == 2 and index == self.get_limiter_index('('):
        self.consume(2, self.get_limiter_index('('))
        type_e = self.E()
        self.consume(2, self.get_limiter_index(''))
        return type_e
    else:
        raise ParserError(
            f"Invalid factor starting with token
'{self.get_word(token)}' at line {
                line_num}, column {col_num}"
        )
    )
# L → true | false
def L(self):
    token = self.current_token()

```



```
        if token and (token[0], token[1]) == (1,
self.get_service_word_index('true')):
            self.consume(1, self.get_service_word_index('true'))
        elif token and (token[0], token[1]) == (1,
self.get_service_word_index('false')):
            self.consume(1, self.get_service_word_index('false'))
        else:
            if token and len(token) == 5:
                line_num, col_num = token[2], token[3]
                raise ParserError(
                    f"Expected 'true' or 'false' at line {
                        line_num}, column {col_num}"
                )
            else:
                raise ParserError("Expected 'true' or 'false'")

# I → Identifier
def I(self):
    token = self.current_token()
    if token and token[0] == 3:
        identifier = self.get_word(token)
        if identifier not in self.symbol_table:
            line_num, col_num = token[2], token[3]
            raise ParserError(
                f"Undeclared identifier '{identifier}' at line {
                    line_num}, column {col_num}"
            )
        self.consume(3, token[1])
    else:
        if token and len(token) == 5:
            line_num, col_num = token[2], token[3]
            raise ParserError(
                f"Expected identifier at line {line_num}, column {
                    col_num}, found '{self.get_word(token)}'"
            )
        else:
            raise ParserError(
                "Expected identifier but reached end of input")
def expect_identifier(self):
    token = self.current_token()
    if token and token[0] == 3:
        identifier = self.get_word(token)
```

```
        if identifier not in self.symbol_table:
            line_num, col_num = token[2], token[3]
            raise ParserError(
                f"Undeclared identifier '{identifier}' at line {
                    line_num}, column {col_num}"
            )
        self.consume(3, token[1])
        return identifier
    else:
        if token and len(token) == 5:
            line_num, col_num = token[2], token[3]
            raise ParserError(
                f"Expected identifier at line {line_num}, column {
                    col_num}, found '{self.get_word(token)}'"
            )
        else:
            raise ParserError(
                "Expected identifier but reached end of input")

    def get_identifier_type(self, identifier, line_num, col_num):
        if identifier in self.symbol_table:
            return self.symbol_table[identifier]
        else:
            raise ParserError(
                f"Undeclared identifier '{identifier}' at line {
                    line_num}, column {col_num}"
            )

    # N → Number
    def N(self):
        token = self.current_token()
        if token and token[0] == 4:
            self.consume(4, token[1])
        else:
            if token and len(token) == 5:
                line_num, col_num = token[2], token[3]
                raise ParserError(
                    f"Expected number at line {line_num}, column {
                        col_num}, found '{self.get_word(token)}'"
                )
            else:
                raise ParserError("Expected number but reached end of
```

```
input")

# Helper methods to get service word and limiter indices
def get_service_word_index(self, word):
    if word in self.service_words:
        return self.service_words.index(word) + 1
    else:
        raise ValueError(f"Unknown service word: {word}")

def get_limiter_index(self, limiter):
    if limiter in self.limiters:
        return self.limiters.index(limiter) + 1
    else:
        raise ValueError(f"Unknown limiter: {limiter}")

def get_expected_word(self, group, index):
    if group == 1:
        return self.service_words[index - 1]
    elif group == 2:
        return self.limiters[index - 1]
    elif group == 3:
        if 0 < index <= len(self.identifiers):
            return self.identifiers[index - 1]
        else:
            return "identifier"
    elif group == 4:
        if 0 < index <= len(self.numbers):
            return self.numbers[index - 1]
        else:
            return "number"
    else:
        return "unknown"

# Semantic Helper Methods
def combine_types(self, type1, type2, operator):

def types_compatible(self, var_type, expr_type):
    if var_type == expr_type:
        return True
    elif var_type == 'float' and expr_type == 'int':
        return True
    else:
```

```
        return False

def main():
    if len(sys.argv) != 2:
        print("Usage: python parser2.py <token_file>")
        sys.exit(1)

    token_file = sys.argv[1]
    parser = Parser(token_file)
    try:
        parser.analyze()
        print("__Parsing completed successfully__")
    except ParserError as e:
        print(f"Parsing error: {e}")
        sys.exit(1)

if __name__ == "__main__":
    main()
```