



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

КУРСОВАЯ РАБОТА

по дисциплине

Теория формальных языков

(наименование дисциплины)

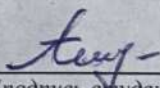
Тема курсовой работы

Разработка распознавателя модельного языка
программирования (вариант №9)

(наименование темы)


Студент группы ИКБО-42-23
(учебная группа)

Туляшева А.Т.
(Фамилия И.О.)


(подпись студента)


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

Унгер А.Ю.


(подпись руководителя)

Консультант ст. преп. каф. ВТ

Боронников А.С.


(подпись консультанта)

Работа представлена к защите « » _____ 2024 г.

Допущен к защите « » _____ 2024 г.

Москва 2024



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

Утверждаю

Заведующий кафедрой

(подпись)

Платонова О.В.

«24» сентября 2024 г.

ЗАДАНИЕ

на выполнение курсовой работы по дисциплине

« Теория формальных языков »

Студент Туляшева Арина Тимуровна Группа ИКБО-42-23

Тема работы: Разработка распознавателя модельного языка программирования

Исходные данные: Грамматика модельного языка согласно варианту №9,
язык программирования – Python

Перечень вопросов, подлежащих разработке, и обязательного графического материала: _____

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

Срок представления к защите курсовой работы: до «23» декабря 2024 г.

Задание на курсовую работу выдал

(подпись)

(Боронников А.С.)
ФИО консультанта

Задание на курсовую работу получил

«19» сентября 2024 г.

(подпись)

(Туляшева А.Т.)
ФИО обучающегося

Москва 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 ПОСТАНОВКА ЗАДАЧИ	5
2 ПОРЯДОК ВЫПОЛНЕНИЯ	7
3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА.....	8
4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА.....	10
5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА.....	12
6 СЕМАНТИЧЕСКИЙ АНАЛИЗ	14
7 ТЕСТИРОВАНИЕ ПРОГРАММЫ.....	15
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 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 Python;
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке Python;
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 9 варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

$\langle \text{операции_группы_отношения} \rangle ::= \langle \rangle \mid = \mid < \mid \leq \mid > \mid \geq$

$\langle \text{операции_группы_сложения} \rangle ::= + \mid - \mid \text{or}$

$\langle \text{операции_группы_умножения} \rangle ::= * \mid / \mid \text{and}$

$\langle \text{унарная_операция} \rangle ::= \text{not}$

$\langle \text{программа} \rangle = \{ / (\langle \text{описание} \rangle \mid \langle \text{оператор} \rangle) (: \mid \text{переход строки}) / \} \text{ end}$

$\langle \text{описание} \rangle ::= \{ \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} : \langle \text{тип} \rangle ; \}$

$\langle \text{тип} \rangle ::= \% \mid ! \mid \$$

$\langle \text{оператор} \rangle ::= \langle \text{составной} \rangle \mid \langle \text{присваивания} \rangle \mid \langle \text{условный} \rangle \mid$

$\langle \text{фиксированного_цикла} \rangle \mid \langle \text{условного_цикла} \rangle \mid \langle \text{ввода} \rangle \mid \langle \text{вывода} \rangle$

$\langle \text{составной} \rangle ::= \langle [\rangle \langle \text{оператор} \rangle \{ (: \mid \text{перевод строки}) \langle \text{оператор} \rangle \} \langle] \rangle$

$\langle \text{присваивания} \rangle ::= \langle \text{идентификатор} \rangle \text{ as } \langle \text{выражение} \rangle$

$\langle \text{условный} \rangle ::= \text{if } \langle \text{выражение} \rangle \text{ then } \langle \text{оператор} \rangle [\text{ else } \langle \text{оператор} \rangle]$

$\langle \text{фиксированного_цикла} \rangle ::= \text{for } \langle \text{присваивания} \rangle \text{ to } \langle \text{выражение} \rangle \text{ do}$

$\langle \text{оператор} \rangle$

$\langle \text{условного_цикла} \rangle ::= \text{while } \langle \text{выражение} \rangle \text{ do } \langle \text{оператор} \rangle$

$\langle \text{ввода} \rangle ::= \text{read } \langle \langle \rangle \langle \text{идентификатор} \rangle \{ , \langle \text{идентификатор} \rangle \} \langle \rangle \rangle$

$\langle \text{вывода} \rangle ::= \text{write } \langle \langle \rangle \langle \text{выражение} \rangle \{ , \langle \text{выражение} \rangle \} \langle \rangle \rangle$

Многострочные комментарии в программе { ... }

Выражения языка задаются правилами:

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{операции_группы_отношения} \rangle \langle \text{операнд} \rangle \}$

$\langle \text{операнд} \rangle ::= \langle \text{слагаемое} \rangle \{ \langle \text{операции_группы_сложения} \rangle \langle \text{слагаемое} \rangle \}$

$\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{операции_группы_умножения} \rangle$

$\langle \text{множитель} \rangle \}$

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

$\langle \text{унарная_операция} \rangle \langle \text{множитель} \rangle \mid \langle \langle \rangle \langle \text{выражение} \rangle \langle \rangle \rangle$

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

<логическая_константа>::= true | false

Правила, определяющие идентификатор, букву и цифру:

<идентификатор>::= <буква> {<буква> | <цифра>}

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

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Правила, определяющие целые числа:

<целое>::= <двоичное> | <восьмеричное> | <десятичное>
| <шестнадцатеричное>

<двоичное>::= { / 0 | 1 / } (B | b)

<восьмеричное>::= { / 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 / } (O | o)

<десятичное>::= { / <цифра> / } [D | d]

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a | b | c |
d | e | f} (H | h)

Правила, описывающие действительные числа:

<действительное>::= <числовая_строка> <порядок> |

[<числовая_строка>] . <числовая_строка> [порядок]

<числовая_строка>::= { / n <цифра> / }

<порядок>::= (E | e) [+ | -] <числовая_строка>

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.

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. 4.1).

Код лексического анализатора приведен в Приложении А.

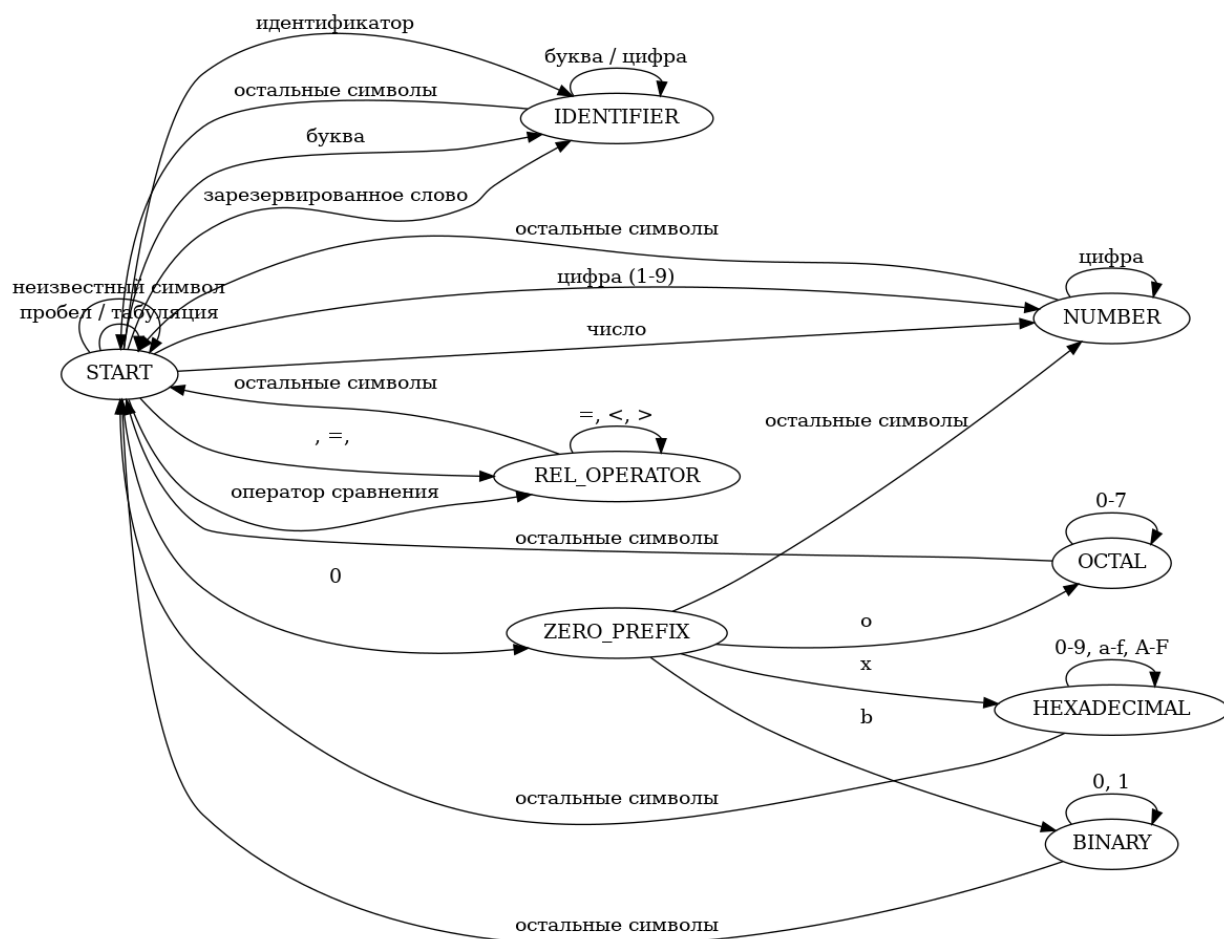


Рисунок 4.1 – Диаграмма состояний лексического анализатора

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.

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:

$$P \rightarrow \{ D S \}$$

$$D \rightarrow I \text{ as TYPE } ; D \mid \varepsilon$$

$$S \rightarrow [S']$$

$$S' \rightarrow \text{IF } E \text{ then } S S'' \mid \text{FOR } A \text{ to } E \text{ do } S \mid \text{WHILE } E \text{ do } S \mid \text{READ } (I) ; \mid \text{WRITE } (E) ; \mid I \text{ as } E ; \mid S$$

$$S'' \rightarrow \text{else } S \mid \varepsilon$$

$$E \rightarrow E \text{ and } R \mid E \text{ or } R \mid R$$

$$R \rightarrow R \text{ REL_OP } A \mid A$$

$$A \rightarrow A \text{ ADD_OP } M \mid M$$

$$M \rightarrow M \text{ MUL_OP } U \mid U$$

$$U \rightarrow (E) \mid \text{not } U \mid \text{TRUE} \mid \text{FALSE} \mid \text{NUMBER} \mid I$$

$$\text{TYPE} \rightarrow \% \mid ! \mid \$$$

Где: P — программа (корневой блок), D — декларации, S — оператор (или последовательность операторов), S' — операторы, включая ветвления, циклы, ввод/вывод, и присваивание, S'' — ветвь else, E — выражение, R — выражение с операторами отношений, A — аддитивное выражение, M — мультипликативное

выражение, U — унарное выражение или базовые элементы, TYPE — тип данных (% для целых, ! для вещественных, \$ для булевых).

Грамматические правила разделены на более мелкие части, что облегчает понимание структуры программы. Исходный код синтаксического анализатора приведен в Приложении Б.

6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

Некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. Основные этапы семантического анализа:

1. Сканирование программы: на этом этапе происходит разбор исходного кода программы на токены, которые соответствуют грамматическим правилам (например, DIM, IDENTIFIER, ASSIGN, NUMBER, и т.д.).

2. Проверка деклараций и типов данных: каждое объявление переменной должно содержать тип, например: “dim x : !;”. При этом x должно быть идентификатором, а ! — одним из допустимых типов данных. Для этого нужно собрать все объявления в структуру данных (например, таблицу символов), чтобы проверять, что переменные использованы в программе правильно, с учетом их типов.

3. Проверка присваиваний: присваивание должно быть совместимо с типом переменной. Например: “x as 10;”. Если x объявлена как %, то присваивание значения типа % или выражения, которое вычисляется в %, будет корректным.

4. Проверка использования переменных: необходимо проверять, что переменные используются после их объявления. Например: “y as x + 10;”. Для этого можно поддерживать список всех объявленных переменных в ходе анализа программы и проверять, что каждая переменная используется только после ее объявления.

5. Проверка выражений и совместимости типов: все выражения должны быть проверены с точки зрения типов операндов.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. В данном варианте все идентификаторы объявляются только в начале программы, и все идентификаторы имеют один тип, что существенно облегчает семантический анализ.

7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Программа 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 некорректной. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1.

Листинг 1 – Тестовая программа

```
{
  dim x : %;
  dim y, z : !;
  dim flag : $;
  x as 132h;
  y as 101b;
  z as 3.14;
  if x < y then
    write(x);
  else
    write(y);
  dim i : %;
  for i as 0 to 10 do
  {
    write(i);
  }
  while x <= 100 do
  {
    x as x + 10;
    write(x);
  }
  read(x, y, flag);
  if flag then
  {
    write(flag);
  }
}
end
```

Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение (рис. 6.1).

```
{
    dim x : %;
    dim y, z : !;
    dim flag : $;
    x as 132h;
    y as 101b;
    z as 3.14;
    if x < y then
        write(x);
    else
        write(y);
    dim i : %;
    for i as 0 to 10 do
    {
        write(i);
    }
    while x <= 100 do
    {
        x as x + 10;
        write(x);
    }
    read(x, y, flag);
    if flag then
    {
        write(flag);
    }
}
end
```

main ×
('IDENTIFIER', 'end')

Синтаксический и семантический анализы успешны

Рисунок 6.1 – Пример синтаксически корректной программы

2. Исходный код программы, содержащий ошибку, приведен на рис. 6.2 совместно с сообщением об ошибке.

```
{
  dim x : %;
  dim y, z : !;
  dim flag : $;
  x as 132h;
  y as 101b;
  z as 3.14;
  if x < y then
    write(x);
  else
    write(y);
  dim i : %;
  for i as 0 to 10 do
  {
    write(i)
  }
  while x <= 100 do
  {
    x as x + 10;
    write(x);
  }
  read(x, y, flag);
  if flag then
  {
    write(flag);
  }
}
end
```

main × |

('RBRACE', '}')

('IDENTIFIER', 'end')

Ошибка синтаксического анализа: Синтаксическая ошибка в позиции 64: ожидался ';', но найден 'RBRACE' ('}').

Рисунок 6.2 – Пример некорректной программы

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 Python.

Разбором исходного текста программы занимается синтаксический анализатор на языке Python. На вход синтаксическому анализатору подаются лексемы, с помощью которых он делает вывод о корректности текста программы.

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. Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.
4. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.
5. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.

ПРИЛОЖЕНИЯ

Приложение А – Класс лексического анализатора

Приложение Б – Класс синтаксического анализатора

Приложение А

Класс лексического анализатора

Листинг А.1 – Лексический анализатор

```
class Lexer:
    RESERVED_WORDS = { # ключевые слова
        "if": "IF",
        "then": "THEN",
        "else": "ELSE",
        "for": "FOR",
        "to": "TO",
        "do": "DO",
        "while": "WHILE",
        "read": "READ",
        "write": "WRITE",
        "as": "ASSIGN",
        "true": "TRUE",
        "false": "FALSE",
        "and": "AND",
        "or": "OR",
        "not": "NOT",
        "dim": "DIM",
    }
    SYMBOLS = { # разделители
        "{": "LBRACE",
        "}": "RBRACE",
        "[": "LBRACKET",
        "]": "RBRACKET",
        ";": "SEMICOLON",
        ",": "COMMA",
        "(": "LPAREN",
        ")": "RPAREN",
        ":": "COLON",
    }
    TYPE = { # типы данных
        "%": "INT",
        "!": "REAL",
        "$": "BOOL",
    }

    OPERATORS = { # операторы
        "+": "ADD_OP",
        "-": "ADD_OP",
        "*": "MUL_OP",
        "/": "MUL_OP",
        "=": "REL_OP",
        "<": "REL_OP",
        ">": "REL_OP",
        "<=": "REL_OP",
        ">=": "REL_OP",
        "<=": "REL_OP",
        "<": "REL_OP",
        ">": "REL_OP",
    }

    def __init__(self, program_text):
        self.program_lines = program_text.splitlines() # текст разбивается на
строки
        self.lexemes = [] # список лексем
        self.column_number = 0 # текущая колонка
        self.line_number = 0 # текущая строка
```

Продолжение листинга A.1

```
def analyze_line(self, line): # анализ одной строки
    accumulator = "" # символы лексемы
    current_state = "START" # начальное состояние
    for idx, char in enumerate(line, start=1):
        self.column_number = idx # номер колонки

        if current_state == "START":
            if char.isspace(): # пропуск пробелов
                continue
            elif char.isalpha():
                accumulator += char
                current_state = "IDENTIFIER" # обработка идентификаторов
            elif char.isdigit() or (char == '.' and accumulator):
                accumulator += char
                current_state = "NUMBER" # обработка чисел
            elif char in self.SYMBOLS:
                self.add_lexeme(self.SYMBOLS[char], char) # добавление
СИМВОЛА

            elif char in "<=>":
                accumulator += char
                current_state = "REL_OPERATOR" # операторы сравнения
            elif char in self.OPERATORS: # добавление оператора
                self.add_lexeme(self.OPERATORS[char], char)
            elif char in self.TYPE: # добавление типа данных
                self.add_lexeme(self.TYPE[char], char)
            else:
                print(f"Ошибка: Неизвестный символ '{char}' в строке
{self.line_number}, колонке {self.column_number}.")
            elif current_state == "IDENTIFIER": # обработка идентификаторов
                if char.isalnum():
                    accumulator += char
                else:
                    if accumulator in self.RESERVED_WORDS:
                        self.add_lexeme(self.RESERVED_WORDS[accumulator],
accumulator)
                    else:
                        self.add_lexeme("IDENTIFIER", accumulator)
                        accumulator = ""
                        current_state = "START"
                        self.analyze_line(char) # повторный вызов для текущего
СИМВОЛА

        # аналогично для других состояний:
        elif current_state == "NUMBER":
            if char.isdigit():
                accumulator += char # собираем цифры числа
            elif char == "." and '.' not in accumulator: # Проверяем на
точку
                accumulator += char
            elif char.lower() in 'bosh': # проверяем на возможные суффиксы
для систем счисления
                if char.lower() == 'b': # двоичная система
                    if accumulator and all(c in '01' for c in accumulator):
# проверка на допустимые цифры
                        self.add_lexeme("NUMBER", accumulator + 'b') #
добавляем лексему
                        accumulator = ""
                        current_state = "START"
                    elif char.lower() == 'o': # восьмеричная система
                        if accumulator and all(c in '01234567' for c in
accumulator): # проверка на допустимые цифры
```

```

добавляем лексему
        self.add_lexeme("NUMBER", accumulator + 'o') #
        accumulator = ""
        current_state = "START"
        self.add_lexeme("SEMICOLON", ";") # Добавляем
точку с запятой
    else:
        print(f"Ошибка: некорректное восьмеричное число
'{accumulator}'))
        accumulator = ""
        current_state = "START"
elif char.lower() == 'd': # десятичная система
    if accumulator and accumulator.isdigit(): # проверка
на допустимые цифры
        self.add_lexeme("NUMBER", accumulator + 'd') #
добавляем лексему
        accumulator = ""
        current_state = "START"
        self.add_lexeme("SEMICOLON", ";") # Добавляем
точку с запятой
    else:
        print(f"Ошибка: некорректное десятичное число
'{accumulator}'))
        accumulator = ""
        current_state = "START"
elif char.lower() == 'h': # шестнадцатеричная система
    if accumulator and all(c in '0123456789abcdefABCDEF'
for c in accumulator): # проверка на допустимые цифры
        self.add_lexeme("NUMBER", accumulator + 'h') #
добавляем лексему
        accumulator = ""
        current_state = "START"
    else:
        self.add_lexeme("NUMBER", accumulator) # если это обычное
число
        accumulator = ""
        current_state = "START"
        self.analyze_line(char) # повторно анализируем текущий
СИМВОЛ

elif current_state == "REL_OPERATOR":
    if char in ">=":
        accumulator += char
        if accumulator in self.OPERATORS:
            self.add_lexeme(self.OPERATORS[accumulator],
accumulator)
            accumulator = ""
            current_state = "START"
    else:
        if accumulator in self.OPERATORS:
            self.add_lexeme(self.OPERATORS[accumulator],
accumulator)
            accumulator = ""
            current_state = "START"
        self.analyze_line(char)
if current_state == "IDENTIFIER" and accumulator:
    if accumulator in self.RESERVED_WORDS:
        self.add_lexeme(self.RESERVED_WORDS[accumulator], accumulator)
    else:
        self.add_lexeme("IDENTIFIER", accumulator)

```

Окончание листинга A.1

```
def add_lexeme(self, lexeme_type, lexeme_value): # сохраняет лексемы в
    список
    self.lexemes.append((lexeme_type, lexeme_value))

def display_lexemes(self): # вывод лексем
    for lexeme in self.lexemes:
        print(lexeme)

def execute(self): # запуск анализа строк
    for line_idx, line in enumerate(self.program_lines, start=1):
        self.line_number = line_idx # обновление номера строки
        self.analyze_line(line) # анализ строки

def retrieve_lexemes(self):
    return self.lexemes # список лексем
```

Приложение Б

Класс синтаксического анализатора

Листинг Б.1 – Синтаксический анализатор

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.position = 0
        self.declared_variables = set()

    def parse(self):
        self.program()

    def current_token(self):
        if self.position < len(self.tokens):
            return self.tokens[self.position]
        return None

    def eat(self, token_type):
        token = self.current_token()
        if token and token[0] == token_type:
            self.position += 1
        else:
            self.raise_syntax_error(token_type)

    def raise_syntax_error(self, expected):
        token = self.current_token()
        position = self.position + 1
        if token:
            raise SyntaxError(f"Синтаксическая ошибка в позиции {position}:  
ожидался '{expected}', но найден '{token[0]}' ('{token[1]}').")
        else:
            raise SyntaxError(f"Синтаксическая ошибка в позиции {position}:  
ожидался '{expected}', но достигнут конец программы.")

    def program(self):
        self.eat('LBRACE')
        while self.current_token() and self.current_token()[0] != 'RBRACE':
            self.declaration_or_statement()
        self.eat('RBRACE')

    def declaration_or_statement(self):
        while self.current_token() and self.current_token()[0] == 'COMMENT':
            self.eat('COMMENT')
        if self.current_token()[0] == 'DIM':
            self.declaration()
        else:
            self.statement()

    def declaration(self):
        self.eat('DIM') # "DIM" как начало объявления переменной
        identifiers = self.identifier_list() # Считываем список идентификаторов
        self.declared_variables.update(identifiers) # Добавляем их в список  
объявленных переменных
        self.eat('COLON') # Ожидаем ":"

        # Ожидаем один из типов: %, ! или $
        if self.current_token()[0] in ('INT', 'REAL', 'BOOL'):
            self.eat(self.current_token()[0]) # Съедаем этот тип
        else:
```

```
        self.raise_syntax_error('тип переменной') # Если тип не найден,
        вызываем ошибку

        self.optional_semicolon(mandatory=True) # Ожидаем или не ожидаем точки
с запятой
    def identifier_list(self):
        identifiers = []
        identifiers.append(self.current_token()[1]) # Добавляем первый
идентификатор
        self.eat('IDENTIFIER') # Переходим к следующему токenu
        while self.current_token() and self.current_token()[0] == 'COMMA': #
Если есть запятая
            self.eat('COMMA') # Пропускаем запятую
            identifiers.append(self.current_token()[1]) # Добавляем следующий
идентификатор
        self.eat('IDENTIFIER') # Переходим к следующему токenu
        return identifiers

    def statement(self):
        while self.current_token() and self.current_token()[0] == 'COMMENT':
            self.eat('COMMENT')
        token = self.current_token()
        if token[0] == 'IF':
            self.if_statement()
        elif token[0] == 'FOR':
            self.for_statement()
        elif token[0] == 'WHILE':
            self.while_statement()
        elif token[0] == 'READ':
            self.read_statement()
            self.optional_semicolon(mandatory=True)
        elif token[0] == 'WRITE':
            self.write_statement()
            self.optional_semicolon(mandatory=True)
        elif token[0] == 'LBRACKET':
            self.composite_statement()
        elif token[0] == 'IDENTIFIER':
            self.assignment_statement()
            self.optional_semicolon(mandatory=True)
        else:
            self.raise_syntax_error("оператор или ключевое слово")

    def if_statement(self):
        self.eat('IF') # Съедаем ключевое слово 'IF'
        self.expression() # Обрабатываем условие
        self.eat('THEN') # Съедаем ключевое слово 'THEN'
        # Проверяем, есть ли начало блока
        if self.current_token() and self.current_token()[0] == 'LBRACE':
            self.composite_statement() # Обрабатываем блок инструкций
        else:
            self.statement() # Обрабатываем одиночный оператор

        # Обрабатываем конструкцию 'else'
        if self.current_token() and self.current_token()[0] == 'ELSE':
            self.eat('ELSE') # Съедаем ключевое слово 'ELSE'
            if self.current_token() and self.current_token()[0] == 'LBRACE':
                self.composite_statement() # Обрабатываем блок инструкций в
else
            else:
                self.statement() # Обрабатываем одиночный оператор в else
```



```
def for_statement(self):
    self.eat('FOR') # Съедаем ключевое слово 'FOR'

    identifier = self.current_token()[1]
    if identifier not in self.declared_variables:
        self.raise_syntax_error(f"Переменная '{identifier}' не была
объявлена.")
    self.eat('IDENTIFIER') # Съедаем идентификатор переменной цикла

    self.eat('ASSIGN') # Ожидаем оператора присваивания 'as'

    # Обрабатываем начальное значение
    self.expression() # Ожидаем выражение, например, 0
    # Ожидаем 'TO' после выражения
    if self.current_token()[0] != 'TO':
        self.raise_syntax_error(
            f"Ожидается 'TO' после присваивания переменной '{identifier}',
найдено {self.current_token()[0]}".)

    self.eat('TO') # Съедаем 'TO'
    # Ожидаем выражение для конца диапазона (например, 10)
    self.expression() # Обрабатываем число или выражение

    self.eat('DO') # Съедаем ключевое слово 'DO'

    if self.current_token() and self.current_token()[0] == 'LBRACE':
        self.composite_statement() # Обрабатываем блок инструкций
    else:
        self.statement() # Если нет блока, обрабатываем один оператор

def while_statement(self):
    self.eat('WHILE') # Съедаем ключевое слово 'WHILE'
    self.expression() # Обрабатываем условие цикла
    self.eat('DO') # Съедаем ключевое слово 'DO'

    # Проверяем, есть ли начало блока
    if self.current_token() and self.current_token()[0] == 'LBRACE':
        self.composite_statement() # Обрабатываем блок инструкций
    else:
        # Если нет блока, то обрабатываем одиночный оператор
        self.statement()

def read_statement(self):
    self.eat('READ')
    self.eat('LPAREN')
    identifiers = self.identifier_list()
    self.check_identifiers_declared(identifiers)
    self.eat('RPAREN')

def write_statement(self):
    self.eat('WRITE')
    self.eat('LPAREN')
    self.expression_list()
    self.eat('RPAREN')

def composite_statement(self):
    self.eat('LBRACE')
    while self.current_token() and self.current_token()[0] != 'RBRACE':
        self.statement()
    self.eat('RBRACE')
```

```
def assignment_statement(self):
    identifier = self.current_token()[1]
    if identifier not in self.declared_variables:
        self.raise_syntax_error(f"Переменная '{identifier}' не была
объявлена.")
    self.eat('IDENTIFIER') # Съедаем идентификатор
    self.eat('ASSIGN')     # Ожидаем 'as'
    self.expression()      # Присваиваем значение

def expression_list(self):
    self.expression()
    while self.current_token() and self.current_token()[0] == 'COMMA':
        self.eat('COMMA')
        self.expression()

def expression(self):
    self.logical_expression()

def logical_expression(self):
    self.relational_expression()
    while self.current_token() and self.current_token()[0] in ('AND',
'OR'):
        self.eat(self.current_token()[0])
        self.relational_expression()

def relational_expression(self):
    self.additive_expression()
    while self.current_token() and self.current_token()[0] == 'REL_OP':
        self.eat('REL_OP')
        self.additive_expression()

def additive_expression(self):
    self.multiplicative_expression()
    while self.current_token() and self.current_token()[0] == 'ADD_OP':
        self.eat('ADD_OP')
        self.multiplicative_expression()

def multiplicative_expression(self):
    self.operand()
    while self.current_token() and self.current_token()[0] == 'MUL_OP':
        self.eat('MUL_OP')
        self.operand()

def operand(self):
    token = self.current_token()
    if token[0] == 'IDENTIFIER': # Это идентификатор
        self.check_identifiers_declared([token[1]])
        self.eat('IDENTIFIER')
    elif token[0] == 'NUMBER': # Если это число, то обрабатываем как число
        # Здесь проверяем, является ли число целым или с плавающей точкой
        number_value = token[1]
        if '.' in number_value or 'e' in number_value or 'E' in number_value:
            # Это вещественное число (FLOAT)
            self.eat('NUMBER')
        else:
            # Целое число
            self.eat('NUMBER')
    elif token[0] in ('TRUE', 'FALSE'): # Логические литералы
        self.eat(token[0])
    elif token[0] == 'NOT': # Логическое отрицание
        self.eat('NOT')
```

Окончание листинга Б.1

```
        self.operand()
    elif token[0] == 'LPAREN': # Если это скобки, то рекурсивно обрабатываем
выражение
        self.eat('LPAREN')
        self.expression()
        self.eat('RPAREN')
    else:
        self.raise_syntax_error("идентификатор, число или выражение")
def optional_semicolon(self, mandatory=False):
    if self.current_token() and self.current_token()[0] == 'SEMICOLON':
        self.eat('SEMICOLON')
    elif mandatory:
        self.raise_syntax_error("';'")

def check_identifiers_declared(self, identifiers):
    for identifier in identifiers:
        if identifier not in self.declared_variables:
            raise SyntaxError(f"Семантическая ошибка: переменная
'{identifier}' не объявлена.")
```