СОДЕРЖАНИЕ

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, Бекус 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 описание 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.

Обработки 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 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:

<операции_группы_отношения> ::= > | < | >= | <= | = |!=

<операции_группы_сложения> ::= + | - | ***or***

<операции_группы_умножения> ::= * | / | ***and***

<унарная_операция> ::= ***not***

<программа> ::= ***program var*** <описание> ***begin*** <оператор> {**;** <оператор>} ***end***

<описание> ::= <идентификатор> {**,** <идентификатор> } **:** <тип>

<тип> ::= ***int*** | ***bool***

<оператор> ::= <составной> | <присваивания> | <условный> | <цикла> | <ввода> | <вывода>

<составной> ::= ***begin*** <оператор> {**;** <оператор> } ***end***

<присваивания> ::= <идентификатор> := <выражение>

<условный> ::= ***if*** <выражение> ***then*** <оператор> [***else*** <оператор>]

<цикла> ::= ***while*** <выражение> ***do*** <оператор>

<ввода> ::= ***read*****(**<идентификатор>**)**

<вывода> ::= ***write*****(**<выражение>**)**

<выражение> ::= <операнд> {<операции_группы_отношения> <операнд>}

<операнд> ::= <слагаемое> {<операции_группы_сложения> <слагаемое>}

<слагаемое> ::= <множитель> {<операции_группы_умножения> <множитель>}

<множитель> ::= <идентификатор> | <число> | <логическая_константа> | <унарная_операция> <множитель> | **(**<выражение>**)**

<логическая_константа> ::= ***true*** | ***false***

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

<число> ::= <цифра>{<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

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 ограничителя 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.

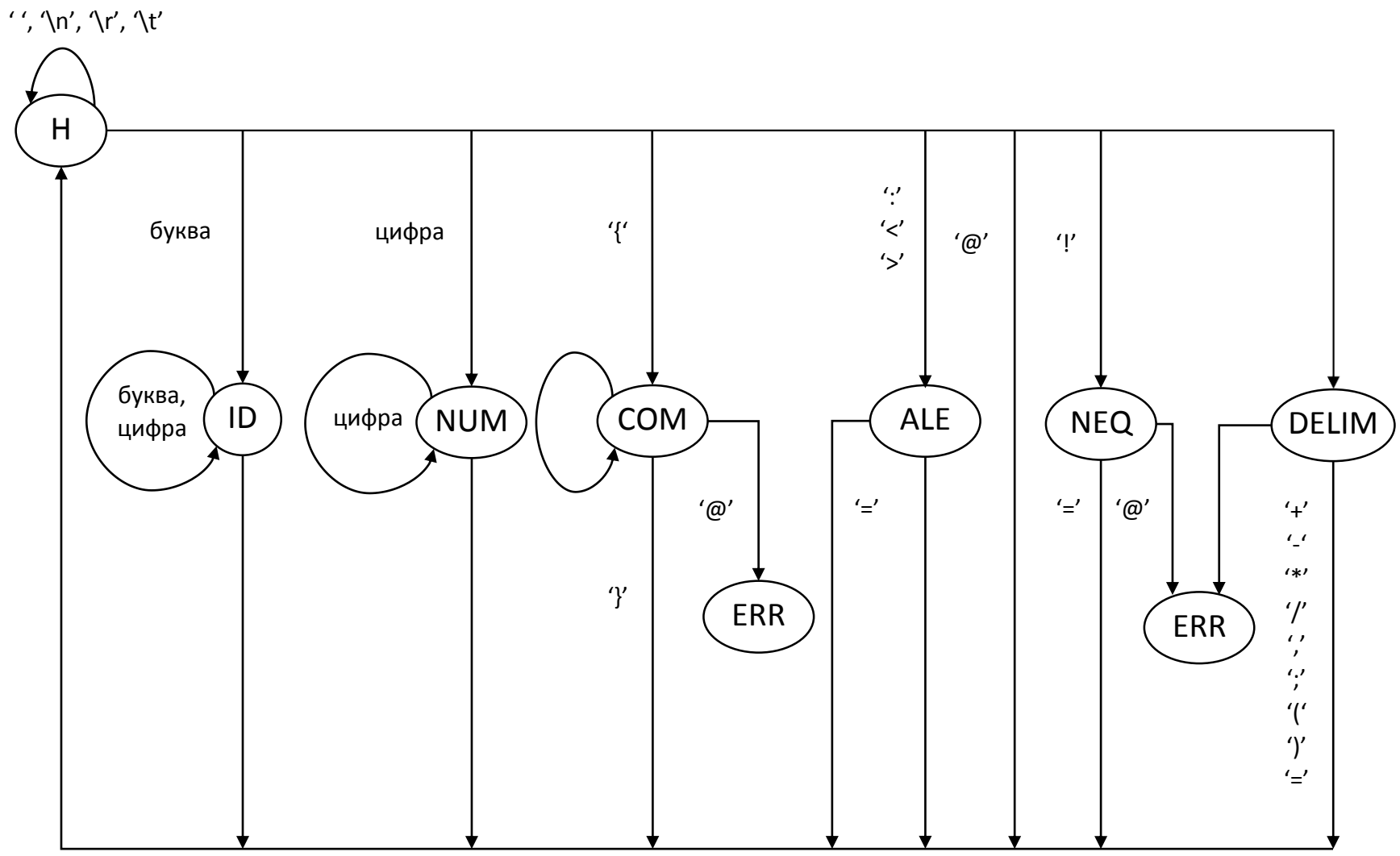

Рисунок 1 – Диаграмма состояний лексического анализатора

5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Будем считать, что лексический и синтаксической 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:

$P \rightarrow \boldsymbol{program}\ D1; B\ \perp$
$D1 \rightarrow \boldsymbol{var}\ D\ \{, D\}$
$D \rightarrow I\ \{, I\}: [\boldsymbol{int}|\boldsymbol{bool}]$
$B \rightarrow \boldsymbol{begin}\ S\ \{; S\}\ \boldsymbol{end}$
$S \rightarrow I \coloneqq E| \ \boldsymbol{if}\ E\ \boldsymbol{then}\ S\ \boldsymbol{else}\ S \mid \boldsymbol{while}\ E\ \boldsymbol{do}\ S \mid B \mid \boldsymbol{read}(I) \mid \boldsymbol{write}(E)$
$E \rightarrow E1\{[= \mid > \mid < \mid >= \mid <= \mid !=]E1\}$
$E1 \rightarrow T\{[+ \mid - \mid \boldsymbol{or}\ ]\ T\}$
$T \rightarrow F\{[* \mid / \mid \boldsymbol{and}\ ]\ F\}$
$F \rightarrow I \mid N \mid L| \ \boldsymbol{not}\ F \mid (E)$
$L \rightarrow \boldsymbol{true} \mid \boldsymbol{false}$
$I \rightarrow C \mid IC \mid IR$
$N \rightarrow R \mid NR$
$C \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$
$R \rightarrow 0 \mid 1 \mid \ldots \mid 9$

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→stack.reset() I stack.push(*c_val*) {, *I stack.push*(*c_val*)} : [***int*** *dec*(*LEX_INT*) | ***bool*** *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.

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 a1 : int,
b1 : bool;
begin
    a1 := 1;
    b1 := false
end
@
```

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

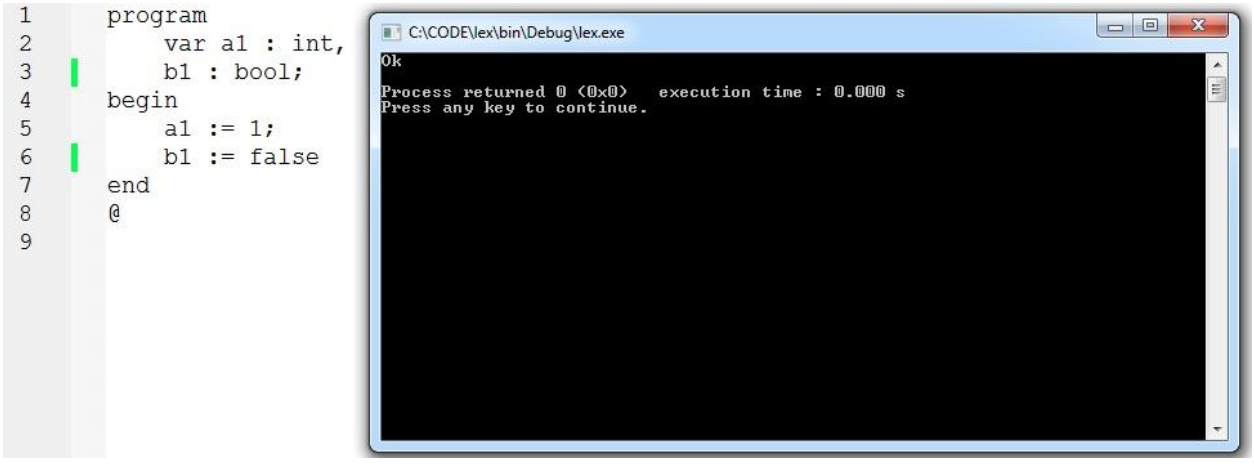

Рисунок 2 – Пример синтаксически корректной программы

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

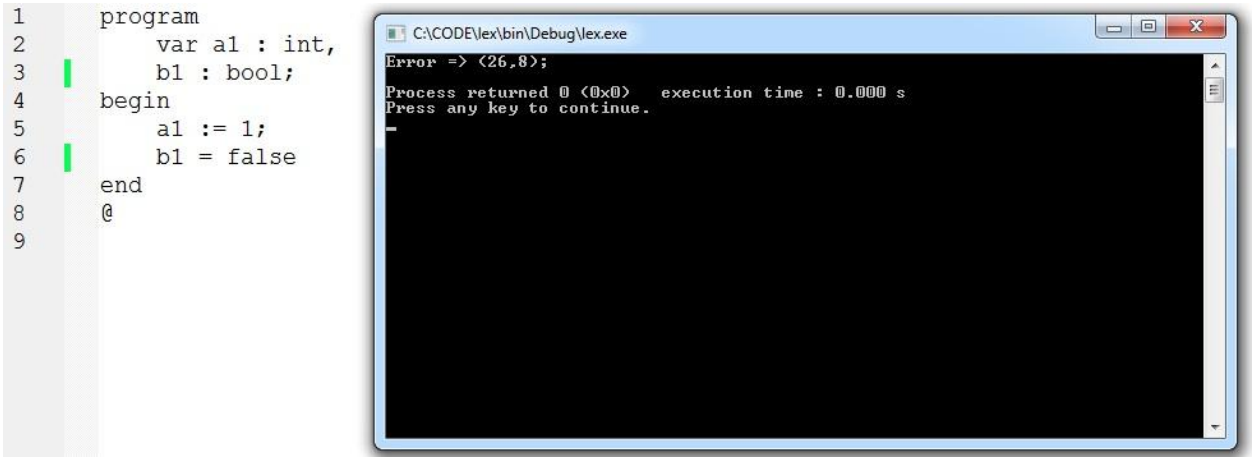

Рисунок 3 – Пример программы, содержащей ошибку

Здесь ошибка допущена в строке 6: неправильное использование оператора сравнения (=). В сообщении об ошибке указана ошибочная лексема (26,8). Тип лексемы, соответствует *LEX_EQ*, т.е. символу «равно».

3. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь переменная *a1* объявлена дважды (*declared twice*).

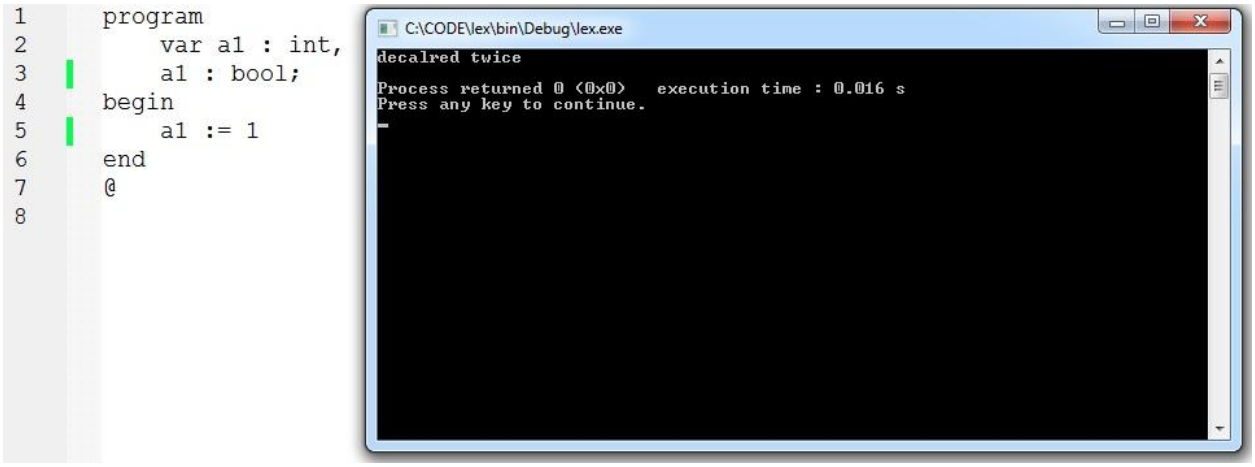

Рисунок 4 – Пример программы, содержащей семантическую ошибку

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 C++ в виде класса *Lexer*.

Разбором исходного текста программы занимается синтаксический анализатор, который реализован в виде класса *Parser* на языке C++. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образом обрабатывать встречающиеся итеративные синтаксически конструкции (нетерминалы *D*, *D1*, *B*, *E1* и *T*).

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.cpp

```
class Lexer
{
    enum state {
        H,
        ID,
        NUM,
        COM,
        ALE,
        NEQ,
        DELIM
    };
    state CS;
    static const char * TW[];
    static const char * TD[];
    static lex_type words[];
    static lex_type dlms[];
    FILE * fp;
    char c;
    char buf[80];
    int buf_top;

    void clear()
    {
        buf_top = 0;
        for(int i = 0; i < 80; ++i) {
            buf[i] = '\0';
        }
    }

    void add()
    {
        buf[buf_top++] = c;
    }

    int look(const char * buf, const char ** list)
    {
        int i = 0;
        while(list[i]) {
            if (!strcmp(buf, list[i])) {
```

Продолжение листинга А.1

```
                return i;
            }
            i++;
        }
        return 0;
    }

    void gc()
    {
        c = (char) fgetc(fp);
    }

public:
    Lexer(const char * filename)
    {
        fp = fopen(filename, "r");
        CS = H;
        clear();
        gc();
    }

    Lex getLex();
};

// Таблица ключевых слов
const char * Lexer::TW[] = {
    "",
    "and",
    "begin",
    "bool",
    "do",
    "else",
    "end",
    "if",
    "false",
    "int",
    "not",
    "or",
    "program",
    "read",
    "then",
    "true",
    "var",
```

Продолжение листинга А.1

```
    "while",
    "write",
    NULL
};

// Таблица разделителей
const char * Lexer::TD[] = {
    "",
    "@",
    ";",
    ",",
    ":",
    ":=",
    "(",
    ")",
    "=",
    "<",
    ">",
    "+",
    "-",
    "*",
    "/",
    "<=",
    "!=",
    ">=",
    NULL
};

// Таблица идентификаторов
TableId TID(100);

// Таблица типов ключевых слов
lex_type Lexer::words[] = {
    LEX_NULL,
    LEX_AND,
    LEX_BEGIN,
    LEX_BOOL,
    LEX_DO,
    LEX_ELSE,
    LEX_END,
    LEX_IF,
    LEX_FALSE,
    LEX_INT,
```

Продолжение листинга А.1

```
    LEX_NOT,
    LEX_OR,
    LEX_PROGRAM,
    LEX_READ,
    LEX_THEN,
    LEX_TRUE,
    LEX_VAR,
    LEX_WHILE,
    LEX_WRITE,
    LEX_NULL
};

// Таблица типов разделителей
lex_type Lexer::dlms[] = {
    LEX_NULL,
    LEX_FIN,
    LEX_SEMICOLON,
    LEX_COMMA,
    LEX_COLON,
    LEX_ASSIGN,
    LEX_LPAREN,
    LEX_RPAREN,
    LEX_EQ,
    LEX_LSS,
    LEX_GTR,
    LEX_PLUS,
    LEX_MINUS,
    LEX_TIMES,
    LEX_SLASH,
    LEX_LEQ,
    LEX_NEQ,
    LEX_GEQ,
    LEX_NULL
};

// Основная функция лексического разбора
Lex Lexer::getLex()
{
    int d, j;
    CS = H;
    do {
        switch(CS) {
        case H:
```

Продолжение листинга А.1

```
            if (c == ' ' || c == '\n' || c == '\r' || c
== '\t') {
                gc();
            }
            else if (isalpha(c)) {
                clear();
                add();
                gc();
                CS = ID;
            }
            else if (isdigit(c)) {
                d = c - '0';
                gc();
                CS = NUM;
            }
            else if (c == '{') {
                gc();
                CS = COM;
            }
            else if (c == ':' || c == '<' || c == '>') {
                clear();
                add();
                gc();
                CS = ALE;
            }
            else if (c == '@') {
                return Lex(LEX_FIN);
            }
            else if (c == '!') {
                clear();
                add();
                gc();
                CS = NEQ;
            }
            else {
                CS = DELIM;
            }
            break;

        case ID:
            if (isalpha(c) || isdigit(c)) {
                add();
                gc();
```

Продолжение листинга А.1

```
            }
            else {
                j = look(buf, TW);
                if (j) {
                    return Lex(words[j], j);
                }
                else {
                    j = TID.put(buf);
                    return Lex(LEX_ID, j);
                }
            }
            break;

        case NUM:
            if (isdigit(c)) {
                d = d * 10 + (c - '0');
                gc();
            }
            else {
                return Lex (LEX_NUM, d);
            }
            break;

        case COM:
            if (c == '}') {
                gc();
                CS = H;
            }
            else if (c == '@' || c == '{') {
                throw c;
            }
            else {
                gc();
            }
            break;

        case ALE:
            if (c == '=') {
                add();
                gc();
                j = look(buf, TD);
                return Lex(dlms[j], j);
            }
```

Окончание листинга А.1

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

        case NEQ:
            if (c == '=') {
                add();
                gc();
                j = look(buf, TD);
                return Lex(LEX_NEQ, j);
            }
            else {
                throw '!';
            }
            break;

        case DELIM:
            clear();
            add();
            j = look(buf, TD);
            if (j) {
                gc();
                return Lex(dlms[j], j);
            }
            else {
                throw c;
            }
            break;
        }
    }
    while(true);
}
```

Приложение Б

Класс синтаксического анализатора

Листинг Б.1 – Parser.cpp

```
class Parser
{
    Lex curr_lex;        // текущая лексема
    lex_type c_type;     // её тип
    int c_val;           // её значение
    Lexer lexer;

    int stack[100];      // стек переменных для контроля
повторного объявления
    int top = 0;

    // Рекурсивные функции
    void P();
    void D1();
    void D();
    void B();
    void S();
    void E();
    void E1();
    void T();
    void F();

    // Получить очередную лексему
    void gl()
    {
        curr_lex = lexer.getLex();
        c_type = curr_lex.getType();
        c_val = curr_lex.getValue();
    }

    void reset()
    {
        top = 0;
    }

    void push(int i)
    {
        stack[top] = i;
        ++top;
```

Продолжение листинга Б.1

```
    }

    int pop()
    {
        --top;
        return stack[top];
    }

    void dec(lex_type type)
    {
        int i;
        while(top) {
            i = pop();
            if (TID[i].isDeclared()) {
                throw "decalred twice";
            }
            TID[i].setDeclared();
            TID[i].setType(type);
        }
    }

public:

    // Провести синтаксический разбор
    void analyze()
    {
        gl();
        P();
    }

    // Конструктор
    Parser(const char * filename) : lexer(filename),
top(0) {}
};

// P→program D1;B@
void Parser::P()
{
    if (c_type == LEX_PROGRAM) {
        gl();
    }
    else {
        throw curr_lex;
```

Продолжение листинга Б.1

```
    }
    D1();
    if (c_type == LEX_SEMICOLON) {
        gl();
    }
    else {
        throw curr_lex;
    }
    B();
    if (c_type != LEX_FIN) {
        throw curr_lex;
    }
}

// D1→var D {,D}
void Parser::D1()
{
    if (c_type != LEX_VAR) {
        throw curr_lex;
    }
    gl();
    D();
    while (c_type == LEX_COMMA) {
        gl();
        D();
    }
}

// D→I {,I}:[int|bool]
void Parser::D()
{
    reset();
    if (c_type != LEX_ID) {
        throw curr_lex;
    }
    push(c_val);
    gl();
    while (c_type == LEX_COMMA) {
        gl();
        if (c_type != LEX_ID) {
            throw curr_lex;
        }
        else {
```

Продолжение листинга Б.1

```
            push(c_val);
            gl();
        }
    }
    if (c_type !=LEX_COLON) {
        throw curr_lex;
    }
    gl();
    if (c_type == LEX_INT) {
        this->dec(LEX_INT);
        gl();
    }
    else if (c_type == LEX_BOOL) {
        this->dec(LEX_BOOL);
        gl();
    }
    else {
        throw curr_lex;
    }
}

//B→begin S {;S} end
void Parser::B()
{
    if (c_type != LEX_BEGIN) {
        throw curr_lex;
    }
    gl();
    S();
    while (c_type == LEX_SEMICOLON) {
        gl();
        S();
    }
    if (c_type == LEX_END) {
        gl();
    }
    else {
        throw curr_lex;
    }
}
```

Продолжение листинга Б.1

```
// S→I::=E | if E then S else S | while E do S | B |
read(I) | write(E)
void Parser::S()
{
    if (c_type == LEX_IF) {
        gl();
        E();
        if (c_type == LEX_THEN) {
            gl();
            S();
            if (c_type == LEX_ELSE) {
                gl();
                S();
            }
        }
        else {
            throw curr_lex;
        }
    }
    else if (c_type == LEX_WHILE) {
        gl();
        E();
        if (c_type != LEX_DO) {
            throw curr_lex;
        }
        gl();
        S();
    }
    else if (c_type == LEX_READ) {
        gl();
        if (c_type != LEX_LPAREN) {
            throw curr_lex;
        }
        gl();
        if (c_type != LEX_ID) {
            throw curr_lex;
        }
        gl();
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
        gl();
    }
```

Продолжение листинга Б.1

```
    else if (c_type == LEX_WRITE) {
        gl();
        if (c_type != LEX_LPAREN) {
            throw curr_lex;
        }
        gl();
        E();
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
        gl();
    }
    else if (c_type == LEX_ID) {
        gl();
        if (c_type != LEX_ASSIGN) {
            throw curr_lex;
        }
        gl();
        E();
    }
    else {
        B();
    }
}

// E→E1{[=|>|<|>=|<=|!=]E1}
void Parser::E()
{
    E1();
    if (c_type == LEX_EQ ||
        c_type == LEX_LSS ||
        c_type == LEX_GTR ||
        c_type == LEX_LEQ ||
        c_type == LEX_GEQ ||
        c_type == LEX_NEQ)
    {
        gl();
        E1();
    }
}

// E1→T{[ + | - | or ] T}
void Parser::E1()
```

Продолжение листинга Б.1

```
{
    T();
    while (c_type == LEX_PLUS || c_type == LEX_MINUS ||
c_type == LEX_OR) {
        gl();
        T();
    }
}

// T→F{[ * | / | and ] F}
void Parser::T()
{
    F();
    while (c_type == LEX_TIMES || c_type == LEX_SLASH ||
c_type == LEX_AND) {
        gl();
        F();
    }
}

// F→I | N | L | not F | (E)
void Parser::F()
{
    if (c_type == LEX_ID) {
        gl();
    }
    else if (c_type == LEX_NUM) {
        gl();
    }
    else if (c_type == LEX_TRUE || c_type == LEX_FALSE) {
        gl();
    }
    else if (c_type == LEX_NOT) {
        gl();
        F();
    }
    else if (c_type == LEX_LPAREN) {
        gl();
        E();
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
        gl();
```

Окончание листинга Б.1

```
    }
    else {
        throw curr_lex;
    }
}
```