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 №15)

(наименование темы)

Студент группы ИКБО-41-23 *(учебная группа)* Попов А.В. *(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 Попов Алексей Валерьевич Группа ИКБО-41-23
Тема работы: Разработка распознавателя модельного языка программирования
Исходные данные: Грамматика модельного языка согласно варианту №15,
язык программирования – C++
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 _____ (Боронников А.С.)
Подпись *ФИО консультанта*

Задание на курсовую работу получил «19» сентября 2024 г.

_____ (Попов А.В.)
Подпись *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.....13

7 ТЕСТИРОВАНИЕ ПРОГРАММЫ.....14

ЗАКЛЮЧЕНИЕ.....16

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....17

ПРИЛОЖЕНИЯ.....18

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> ::= + | - | ||

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

<унарная_операция> ::= !

<программа> ::= «{» {/ (<описание> | <оператор>) ; /} «}»

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

<тип> ::= **integer** | **real** | **boolean**

<оператор> ::= <составной> | <присваивания> | <условный> | <фиксированного_цикла> | <условного_цикла> | <ввода> | <вывода>

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

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

<условный> ::= **if** «(»<выражение> «)» <оператор> [**else** <оператор>]

<фиксированного_цикла> ::= **for** <присваивания> **to** <выражение> [**step** <выражение>] <оператор> **next**

<условного_цикла> ::= **while** «(»<выражение> «)» <оператор>

<ввода> ::= **readln** идентификатор {, <идентификатор> }

<вывода> ::= **writeln** <выражение> {, <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.

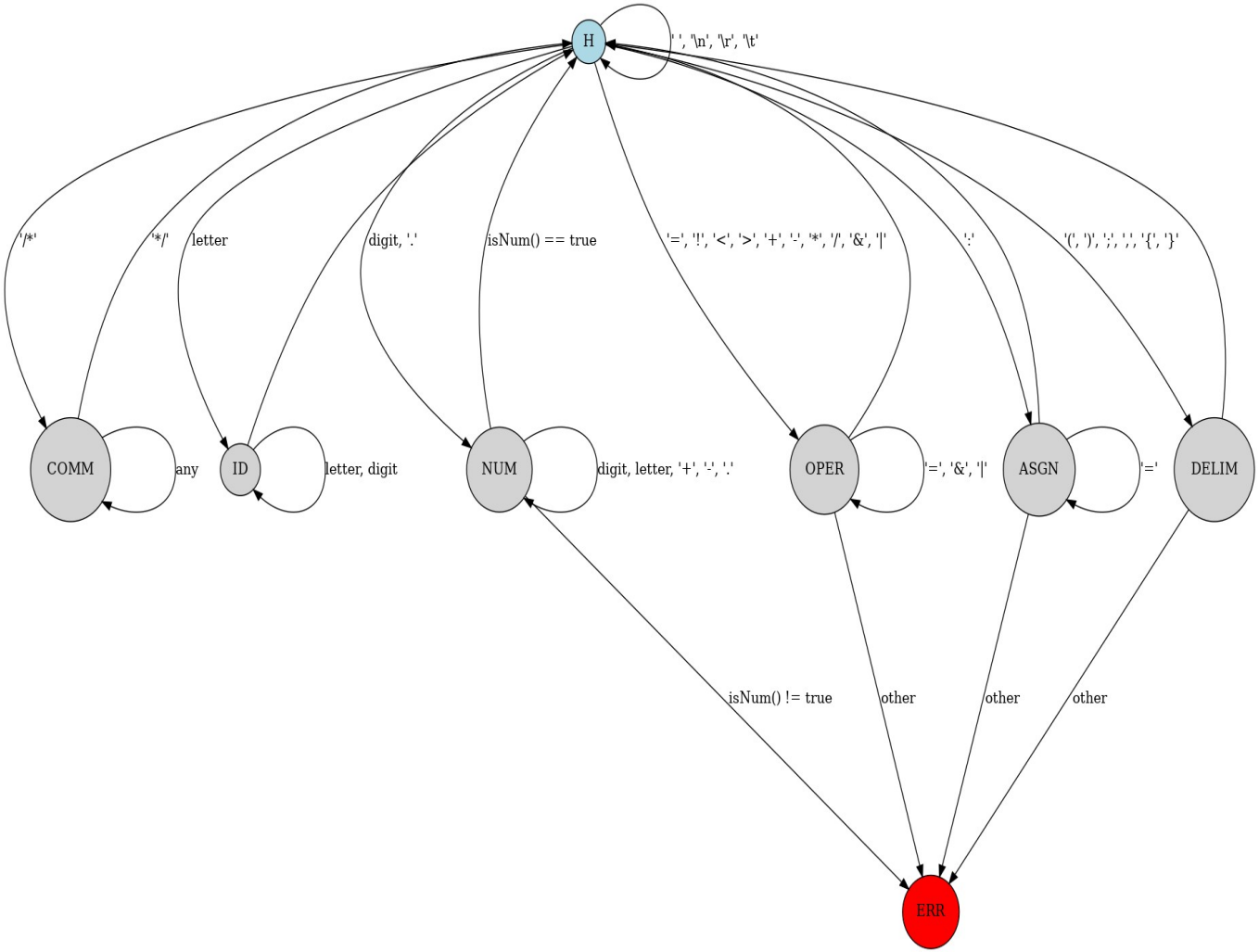


Рисунок 1 – Диаграмма состояний лексического анализатор

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 → «{» D1 | S «}» ⊥
D1 → dim D
D → I {, I} integer | real | boolean
B → begin S {; S} end
S → I := E | if (E) S [else S] | while (E) S | for I := E to E [step E] S next | readln I {, I} | writeln I {, I}
E → E1{[!= | = = | < | <= | > | >=] E1}
E1 → T{[+ | − | ||] T}
T → F{[* | / | &&] F}
F → I | N | L | != F | (E)
$L \rightarrow true \mid false$
$I \rightarrow C \{ C \mid R \}$
$N \rightarrow R \{ N \mid R \}$
$C \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$
$R \rightarrow 0 \mid 1 \mid \ldots \mid 9$

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 структуру *Lex* заносятся данные обо всех лексемах- идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа в структуру Identifier заносятся данные о типе идентификатора (поле *type*) и его имени (поле name).

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 некорректной лексемой. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1.

Листинг 1 – Тестовая программа

```
{
dim a, b integer
a := 10
b := a + 5
if (a < b)
    writeln a
else
    writeln b
}
```

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

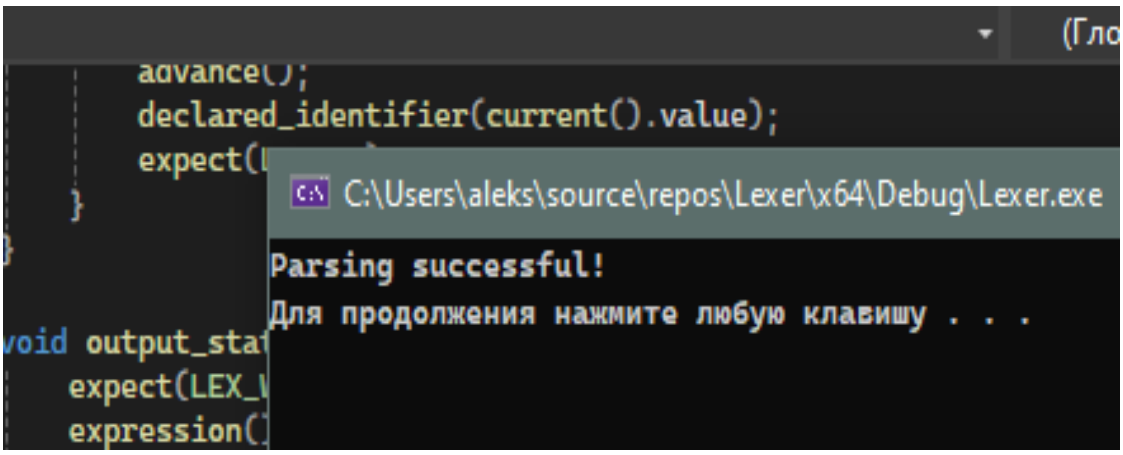


Рисунок 2 – Пример синтаксически корректной программы

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

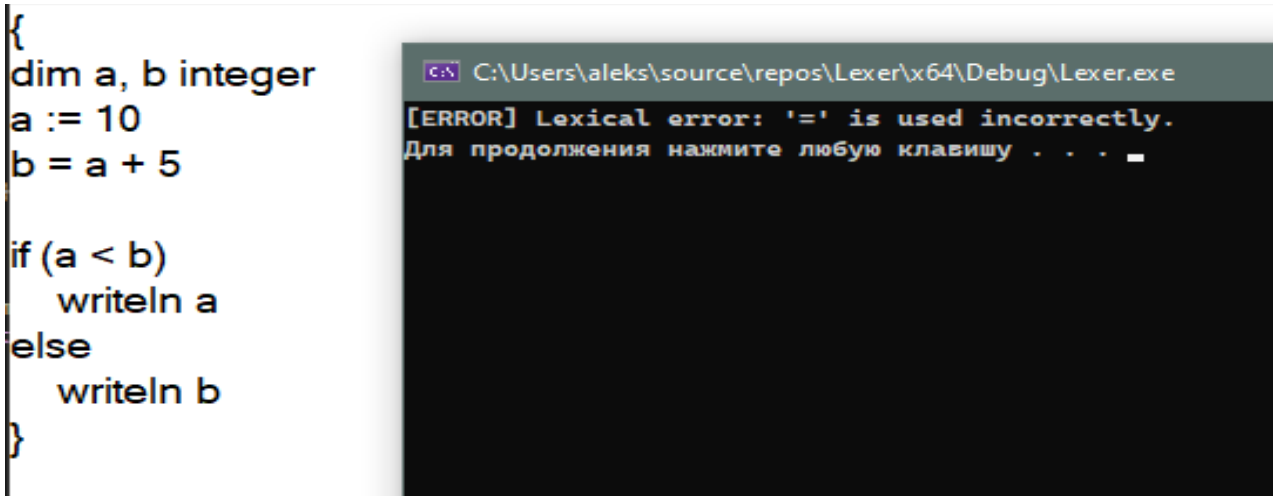
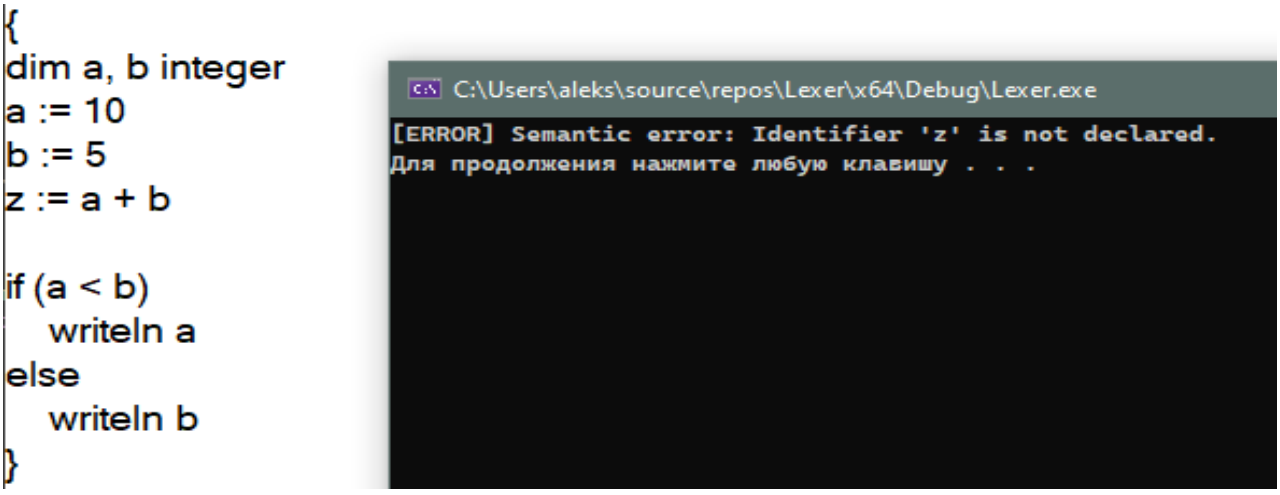


Рисунок 3 – Пример программы, содержащей ошибку

Здесь ошибка допущена в строке 3: неправильное использование оператора присвоения (:=). В сообщении об ошибке указана ошибочная лексема.

2. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь к необъявленной переменной (z) был использован оператор присваивания.



(as).

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 среднего 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 – класс Lexer

```
class Lexer {
private:
    ifstream file;
    char current_char;
    unordered_map<string, lex_type> keywords = {
    {"dim", LEX_DIM}, {"integer", LEX_INTEGER}, {"real", LEX_REAL}, {"boolean",
LEX_BOOLEAN},
    {"if", LEX_IF}, {"else", LEX_ELSE}, {"for", LEX_FOR}, {"to", LEX_TO},
{"step", LEX_STEP},
    {"next", LEX_NEXT}, {"while", LEX_WHILE}, {"readln", LEX_READLN},
{"writeln", LEX_WRITELN},
    {"true", LEX_TRUE}, {"false", LEX_FALSE}, {"begin", LEX_BEGIN}, {"end",
LEX_END}
    };

    void advance() { current_char = file.get(); }
    void skip_whitespace() { while (isspace(current_char)) advance(); }
    void skip_comment() {
        advance(); advance();
        while (current_char != EOF) {
            if (current_char == '*' && file.peek() == '/') {
                advance(); advance();
                return;
            }
            advance();
        }
    }

    Lex parse_identifier() {
        string result;
        while (isalnum(current_char)) {
            result += current_char;
            advance();
        }
        return keywords.count(result) ? Lex(keywords[result], result) :
Lex(LEX_ID, result);
    }

    Lex parse_number() {
        string result;

        while (isalnum(current_char) || current_char == '.' || current_char ==
'+' || current_char == '-') {
            result += current_char;
            advance();
        }

        if (isNum(result)) {
            if (result.find(".") != -1) {
                return Lex(LEX_RNUM, result);
            }
            else {
                return Lex(LEX_NUM, result);
            }
        }
        else {
```

Продолжение листинга А.1

```
            throw runtime_error("Lexical error: Invalid number '" + result +
"'.");
        }
    }

    Lex parse_operator() {
        string result(1, current_char);
        advance();

        if (result == ":") {
            if (current_char == '=') {
                result += current_char;
                advance();
                return Lex(LEX_ASSIGN, result);
            }
            else {
                throw runtime_error("Lexical error: ':' is used incorrectly.");
            }
        }

        if (result == "=") {
            if (current_char == '=') {
                result += current_char;
                advance();
                return Lex(LEX_EQ, result);
            }
            else {
                throw runtime_error("Lexical error: '=' is used incorrectly.");
            }
        }

        if (result == "!") {
            if (current_char == '=') {
                result += current_char;
                advance();
                return Lex(LEX_NEQ, result);
            }
            return Lex(LEX_NOT, result);
        }

        if (result == "<") {
            if (current_char == '=') {
                result += current_char;
                advance();
                return Lex(LEX_LEQ, result);
            }
            return Lex(LEX_LT, result);
        }

        if (result == ">") {
            if (current_char == '=') {
                result += current_char;
                advance();
                return Lex(LEX_GEQ, result);
            }
            return Lex(LEX_GT, result);
        }

        if (result == "|") {
            if (current_char == '|') {
                result += current_char;
                advance();
                return Lex(LEX_OR, result);
```

Продолжение листинга А.1

```
            }
        }

        if (result == "&") {
            if (current_char == '&') {
                result += current_char;
                advance();
                return Lex(LEX_AND, result);
            }
        }

        if (result == "+") return Lex(LEX_PLUS, result);
        if (result == "-") return Lex(LEX_MINUS, result);
        if (result == "*") return Lex(LEX_TIMES, result);
        if (result == "/") return Lex(LEX_DIV, result);

        throw runtime_error("Lexical error: Unknown operator: '" + result +
"'.");
    }

    Lex parse_delimiter() {
        string result(1, current_char);
        advance();
        if (result == "(" || result == ")" || result == ";" || result == "{" ||
result == "}" || result == ",") {
            if (result == "(") return Lex(LEX_LPAREN, result);
            if (result == ")") return Lex(LEX_RPAREN, result);
            if (result == ";") return Lex(LEX_SEMICOLON, result);
            if (result == ",") return Lex(LEX_COMMA, result);
            if (result == "{") return Lex(LEX_LBRACE, result);
            if (result == "}") return Lex(LEX_RBRACE, result);
        }
        else {
            throw runtime_error("Lexical error: Unknown symbol: '" + result +
"'.");
        }
    }

public:
    explicit Lexer(const string& filename) {
        file.open(filename);
        if (!file.is_open()) {
            throw runtime_error("File not found or cannot be opened.");
        }
        advance();
    }

    vector<Lex> get_lex_table() {
        vector<Lex> lex_table;
        Lex token;
        while ((token = get_next_lex()).type != LEX_FIN) {
            lex_table.push_back(token);
        }
        return lex_table;
    }

    Lex get_next_lex() {
        while (current_char != EOF) {
            if (current_char == '/' && file.peek() == '*') {
                skip_comment();
                continue;
            }
```

Окончание листинга А.1

```
            if (isspace(current_char)) {
                skip_whitespace();
                continue;
            }
            if (isalpha(current_char)) return parse_identifier();
            if (isdigit(current_char) || current_char == '.') return
parse_number();
            if (current_char == '=' || current_char == '!' || current_char ==
'<' || current_char == '>' ||
                current_char == '+' || current_char == '-' || current_char ==
'*' || current_char == '/' ||
                current_char == '&' || current_char == '|' || current_char ==
':') {
                return parse_operator();
            }
            return parse_delimiter();
        }
        return Lex(LEX_FIN, "");
    }
};
```

Приложение Б

Класс синтаксического анализатора

Листинг Б.1 – класс Parser

```
class Parser {
private:
    vector<Lex> lex_table;
    unordered_map<string, lex_type> symbol_table; // Таблица идентификаторов
    size_t pos;

    Lex current() {
        if (pos < lex_table.size()) {
            return lex_table[pos];
        }
        return Lex(LEX_FIN, "");
    }

    void advance() {
        if (pos < lex_table.size()) {
            pos++;
        }
    }

    void expect(lex_type type) {
        if (current().type != type) {
            throw runtime_error("Syntax error: Expected " +
string(lex_type_description(type)) + ", but found " +
lex_type_description(current().type) + ".");
        }
        advance();
    }

    void declared_identifier(const string& identifier) {
        if (symbol_table.find(identifier) == symbol_table.end()) {
            throw runtime_error("Semantic error: Identifier '" + identifier + "'
is not declared.");
        }
    }

    void validate_identifier(const string& identifier) {
        if (identifier.empty()) {
            throw runtime_error("Semantic error: Identifier cannot be empty.");
        }
        if (!isalpha(identifier[0])) {
            throw runtime_error("Semantic error: Identifier must start with a
letter.");
        }
        for (char c : identifier) {
            if (!isalnum(c)) {
                throw runtime_error("Semantic error: Identifier can only contain
alphanumeric characters.");
            }
        }
    }

    void declare_identifier(const string& identifier, lex_type type) {
        if (symbol_table.find(identifier) != symbol_table.end()) {
            throw runtime_error("Semantic error: Identifier '" + identifier + "'
is already declared.");
        }
```

Продолжение листинга Б.1

```
        symbol_table[identifier] = type;
    }

    lex_type get_identifier_type(const string& identifier) {
        auto it = symbol_table.find(identifier);
        if (it == symbol_table.end()) {
            throw runtime_error("Semantic error: Identifier '" + identifier + "'
is not declared.");
        }
        return it->second;
    }

    void check_boolean_expression() {
        lex_type expr_type = expression();
        if (expr_type != LEX_BOOLEAN) {
            throw runtime_error("Semantic error: Condition must be a boolean
expression.");
        }
    }

    void check_relation_operands(lex_type left, lex_type right) {
        if (left != LEX_INTEGER || right != LEX_INTEGER) {
            throw runtime_error("Semantic error: Relational operators require
integer operands.");
        }
    }

public:
    Parser(const vector<Lex>& lex_table) : lex_table(lex_table), pos(0) {}

    void parse() {
        program();
        if (current().type != LEX_FIN) {
            throw runtime_error("Syntax error: Unexpected token after program
end.");
        }
    }

    void program() {
        expect(LEX_LBRACE);
        while (current().type != LEX_RBRACE) {
            if (current().type == LEX_DIM) {
                description();
            }
            else {
                statement();
            }
        }
        expect(LEX_RBRACE);
    }

    void description() {
        vector<string> temp;
        expect(LEX_DIM);
        temp.push_back(current().value);
        expect(LEX_ID);
        while (current().type == LEX_COMMA) {
            advance();
            validate_identifier(current().value);
            temp.push_back(current().value);
            expect(LEX_ID);
        }
```

Продолжение листинга Б.1

```
        if (current().type != LEX_INTEGER && current().type != LEX_REAL &&
current().type != LEX_BOOLEAN) {
            throw runtime_error("Syntax error: Expected type (integer, real,
boolean).\n");
        }
        lex_type type = current().type;
        advance();
        for (auto identifier : temp) {
            declare_identifier(identifier, type);
        }
    }

    void statement() {
        if (current().type == LEX_ID) {
            assignment();
        }
        else if (current().type == LEX_IF) {
            conditional();
        }
        else if (current().type == LEX_FOR) {
            for_loop();
        }
        else if (current().type == LEX_WHILE) {
            while_loop();
        }
        else if (current().type == LEX_READLN) {
            input_statement();
        }
        else if (current().type == LEX_WRITELN) {
            output_statement();
        }
        else if (current().type == LEX_BEGIN) {
            compound_statement();
        }
        else {
            throw runtime_error("Syntax error: Unexpected token " +
string(lex_type_description(current().type)));
        }
    }

    void assignment() {
        string identifier = current().value;
        declared_identifier(identifier);
        expect(LEX_ID);
        expect(LEX_ASSIGN);
        lex_type left_type = get_identifier_type(identifier);
        lex_type right_type = expression();
        if (left_type != right_type) {
            throw runtime_error("Semantic error: Type mismatch in assignment to
'" + identifier + "'.");
        }
    }

    void conditional() {
        expect(LEX_IF);
        expect(LEX_LPAREN);
        check_boolean_expression();
        expect(LEX_RPAREN);
        statement();
        if (current().type == LEX_ELSE) {
            advance();
            statement();
```

Продолжение листинга Б.1

```
        }
    }

    void for_loop() {
        expect(LEX_FOR);
        assignment();
        expect(LEX_TO);
        check_boolean_expression();
        if (current().type == LEX_STEP) {
            advance();
            lex_type expr_type = expression();
            if (expr_type != LEX_INTEGER) {
                throw runtime_error("Semantic error: Loop step must be an
integer.");
            }
        }
        statement();
        expect(LEX_NEXT);
    }

    void while_loop() {
        expect(LEX_WHILE);
        expect(LEX_LPAREN);
        check_boolean_expression();
        expect(LEX_RPAREN);
        statement();
    }

    lex_type expression() {
        lex_type result = operand();
        while (current().type == LEX_EQ || current().type == LEX_NEQ ||
            current().type == LEX_LT || current().type == LEX_LEQ ||
            current().type == LEX_GT || current().type == LEX_GEQ) {
            lex_type left = result;
            advance();
            lex_type right = operand();
            check_relation_operands(left, right);
            result = LEX_BOOLEAN;
        }
        return result;
    }

    lex_type operand() {
        lex_type result = term();
        while (current().type == LEX_PLUS || current().type == LEX_MINUS ||
current().type == LEX_OR) {
            advance();
            lex_type temp_result = term();
            if (temp_result != result) {
                result = LEX_NULL;
            }
        }
        return result;
    }

    lex_type term() {
        lex_type result = factor();
        while (current().type == LEX_TIMES || current().type == LEX_DIV ||
current().type == LEX_AND) {
            advance();
            lex_type temp_result = factor();
            if (temp_result != result) {
```

Продолжение листинга Б.1

```
                result = LEX_NULL;
            }
        }
        return result;
    }

    lex_type factor() {
        if (current().type == LEX_ID) {
            string identifier = current().value;
            declared_identifier(identifier);
            advance();
            return get_identifier_type(identifier);
        }
        else if (current().type == LEX_NUM) {
            advance();
            return LEX_INTEGER;
        }
        else if (current().type == LEX_RNUM) {
            advance();
            return LEX_REAL;
        }
        else if (current().type == LEX_LPAREN) {
            advance();
            lex_type result = expression();
            expect(LEX_RPAREN);
            return result;
        }
        else if (current().type == LEX_TRUE || current().type == LEX_FALSE) {
            advance();
            return LEX_BOOLEAN;
        }
        else if (current().type == LEX_NOT) {
            advance();
            lex_type result = factor();
            if (result != LEX_BOOLEAN) {
                throw runtime_error("Semantic error: NOT operator requires a
boolean operand.");
            }
            return LEX_BOOLEAN;
        }
        else {
            throw runtime_error("Syntax error: Unexpected token " +
string(lex_type_description(current().type)));
        }
    }

    void compound_statement() {
        expect(LEX_BEGIN);
        statement();
        while (current().type == LEX_SEMICOLON) {
            advance();
            statement();
        }
        expect(LEX_END);
    }

    void input_statement() {
        expect(LEX_READLN);
        declared_identifier(current().value);
        expect(LEX_ID);
        while (current().type == LEX_COMMA) {
            advance();
```

Окончание листинга Б.1

```
            declared_identifier(current().value);
            expect(LEX_ID);
        }
    }

    void output_statement() {
        expect(LEX_WRITELN);
        expression();
        while (current().type == LEX_COMMA) {
            advance();
            expression();
        }
    }
};
```