СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Теория Формальных языков берет свое начало из Америки. В 1957 году Джон Бэкус разработал компилятор языка программирования Фортран, с помощью ученого Н. Хомского – автора классификаций формального языка. Хомский занимался естественными языками, и благодаря его теории Бэкус разработал язык программирования. Это дало огромный толчок к развитию программирования и разработке сотен языков программирования.

Разработка нового языка программирования требует творческого подхода, несмотря на существование большого количества алгоритмов для автоматизации процесса написания транслятора для формальных языков. Это касается синтаксиса языка, который должен быть как удобен в прикладном программировании, так и должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Теория формальных языков и практические методы разработки распознавателей этих языков составляют большую часть обучения современного программиста.

Целью курсовой работы является разработка распознавателя модельного языка программирования, согласно заданной формальной грамматике. Для достижения цели необходимо выполнить следующие задачи:

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

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Постановка задачи

Разработать распознаватель модельного языка программирования, согласно заданной формальной грамматике.

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

Распознаватель схематично представляется в виде совокупности входной ленты, читающей головки, указывающей на очередной символ на ленте, устройства управления (далее УУ), и дополнительной памяти.

Конфигурацией распознавателя является:

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

Трансляция исходного текста программы осуществляется в несколько этапов:

- 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 (далее ДС).

Алгоритм синтаксического анализа строится на базе контекстно-свободных (далее КС) грамматик. Задача синтаксического анализатора —

провести анализ разбор текста программы и сопоставить его с формальным описанием языка.

Семантический анализ позволяет учесть особенности языка, 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 объединяют.

1.2 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.

1.3 Грамматика языка

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

1. <операции_группы_отношения>:: = < > | = | < | <= | > | >=,
2. <операции_группы_сложения>:: = + | - | or,
3. <операции_группы_умножения>::= * | / | and,
4. <унарная_операция>::= not,
5. <программа> = {/ (<описание> | <оператор>) (: | переход строки) /} end,
6. <описание>::= dim <идентификатор> {, <идентификатор> } <тип>,
7. <тип>::= % | ! | $,
8. <составной>::= «[» <оператор> { (: | перевод строки) <оператор> } «]»,
9. <присваивания>::= <идентификатор> as <выражение>,
10. <условный>::= if <выражение> then <оператор> [else <оператор>],
11. <фиксированного_цикла>::= for <присваивания> to <выражение> do <оператор>,
12. <условного_цикла>::= while <выражение> do <оператор>,
13. <ввода>::= read «(»<идентификатор> {, <идентификатор> } «)»,
14. <вывода>::= write «(»<выражение> {, <выражение> } «)»,
15. <выражение> ::= <операнд> {<операции_группы_отношения> <операнд>},
16. <операнд> ::= <слагаемое> {<операции_группы_сложения> <слагаемое>},

17. <слагаемое> ::= <множитель> {<операции_группы_умножения> <множитель>},
18. <множитель> ::= <идентификатор> | <число> | <логическая_константа> | <унарная_операция> <множитель> | «(» <выражение> «)»,
19. <логическая_константа> ::= true | false,
20. <идентификатор> ::= <буква>{<буква> | <цифра>},
21. <число> ::= <цифра>{<цифра>},
22. <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,
23. <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. Терминалы, представляющие собой ключевые слова языка:

- or;
- and;
- not;
- end;
- dim;
- as;
- if;
- then;
- else;
- for;
- to;
- do;

- while;
- read;
- write;
- true;
- false.

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Разработка лексического анализатора

Лексический анализатор — подпрограмма, принимающая на вход исходный код программы, и выдающая последовательность лексем — минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования были выделены следующие типы лексем:

- ключевые слова,
- ограничители,
- числа,
- идентификаторы.

Во время разработки лексического анализатора ключевые слова и разделители выделяются заранее, идентификаторы и числа в момент разбора исходного кода.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы — пара чисел n,k, где n — номер таблицы, а 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», «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. Диаграмма состояний представлена на рисунке 1.

Исходный код лексического анализатора приведен в Приложении А.

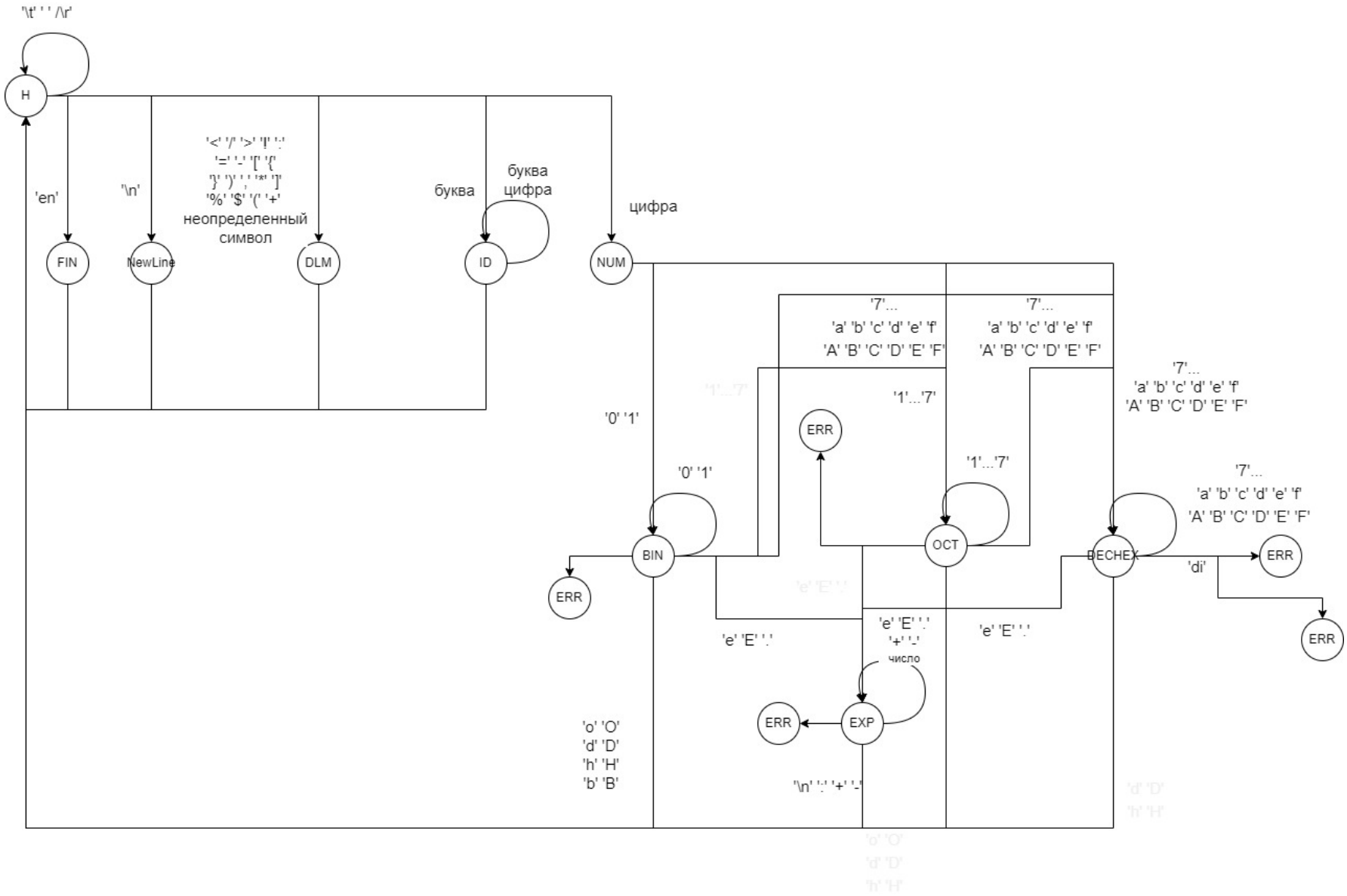

Рисунок 1. Диаграмма состояний

2.2 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 (далее 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|B {: | \n D1|B} end;
- D1 → dim D {,D};
- D → I {,I}[% | ! | $];
- B → S ;
- S → I as *E*| if E then S [else S | ε] | for I as E to E do B | while E do S | B | read(I{,I}) | write(E{,E}) | [S {: | \n S}];
- E → E1{[= | > | < | >= | <= |<>]E1};
- E1 → T{[+ | − | or] T};
- T → F{[∗ | / | and] F};
- F → I | N | L| not F | (E);
- L → true | false;
- I → C | IC | IR;
- N → R | NR;
- C → a | b | … | z | A | B | … | Z;
- R → 0 | 1 | … | 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.

2.3 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.

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 при получении каждого идентификатора информация о нем заносится в буферную память, если он еще не объявлен. В случае использования идентификатора, отсутствующего в буферной памяти, ошибка об этом выводится пользователю. В случае повторной инициализации идентификатора, уже присутствующего в буфернйо памяти, ошибка об этом также выводится пользователю.

Описания функций семантических проверок приведены в листинге в Приложении Б.

3 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработано консольное приложение. 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. Список ошибок представлен в таблице 1.

Таблица 1

Список ошибок

Номер ошибки	Суть ошибки
101	Ожидалось число
102	Ожидалось end
103	Ожидалось число или e или E или. или b или B
104	Ожидалось число или e или E или. или o или O
105	Ожидалось отсутствие точки после e
106	Ожидалось e или E
107	Ожидалось отсутствие i после d
108	Ожидалось число или e или E или. или d или D или h или H
109	Ожидалось отсутствие < после >
110	Ожидался разделитель
111	Ожидалась буква для описания cc
112	Ожидалось продолжение действительного числа
113	Ожидалось продолжение действительного числа

Таблица 1 (Продолжение)

114	Ожидалось D или d или B или b или H или h или O или o или . или E или e или число
201	Ожидался оператор группы умножения
202	Ожидался оператор или описание
203	Ожидалось : или перенос строки
204	Ожидалось end
205	Ожидался идентификатор
206	Ожидался dim
207	Ожидался тип переменной
208	Ожидался]
209	Ожидался as
210	Ожидался then
211	Ожидался to
212	Ожидалось выражение
213	Ожидался do
214	Ожидался оператор
215	Ожидалась (
216	Ожидалась операция группы сложения
217	Ожидалась)
218	Неопознанная ошибка
219	Ожидалась операция группы отношения
221	Ожидалась унарная операция
301	Повторное объявление переменной

Таблица 1 (Продолжение)

302	Использование необъявленной переменной

Рассмотрим примеры.

Тест 1. Правильный код.

```
dim k, l % {тест коммента}
dim a, o !:a as 5d-3o
l as 5.55e-5
if a <= 3d then read ( k, l) else write(2d+3h,7d-1b){проверка 2}
end
```

Программа должна выдать информацию об успешном прохождении теста. Результат работы программы при тесте 1 представлен на рисунке 2.

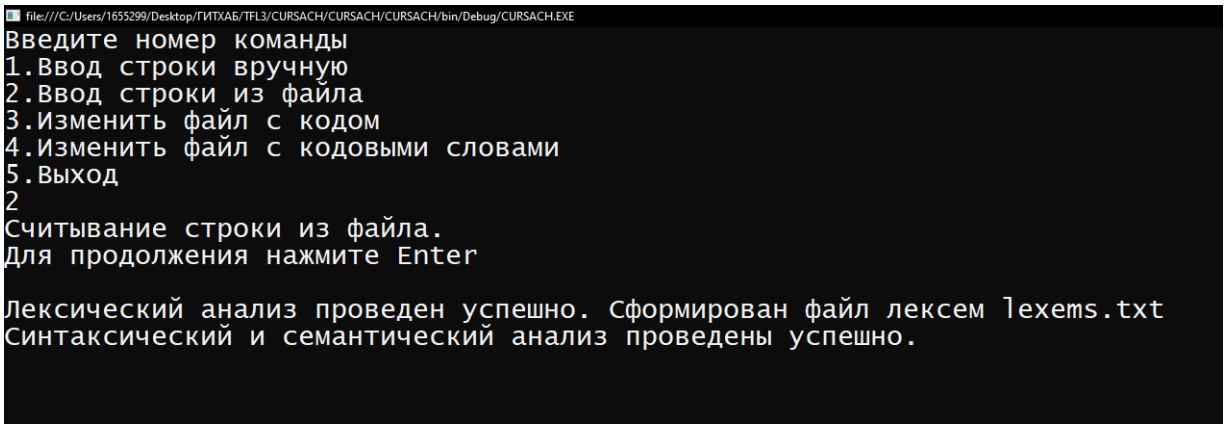

Рисунок 2. Результат работы программы при тесте 1

Тест 2. Лексическая ошибка.

Во 2 тесте была допущена умышленная лексическая ошибка. В 3 строке кода был добавлен символ «@» не обусловленный правилами грамматики.

```
dim k, l % {тест коммента}
dim a, o !:a as 5d-3o
@
l as 5.55e-5
```

```
if a <= 3d then read ( k, l) else write(2d+3h,7d-1b){проверка 2}
end
```

Программа должна выдать ошибку при прохождении лексического анализа. Так как последнее, на что проверяет лексический анализатор, это разделители, то в случае успешного тестирования, программа должна выдать ошибку 110, что согласно Таблице 1 характеризуется как «Ожидался разделитель». Результат работы программы при тесте 2 представлен на рисунке 3.

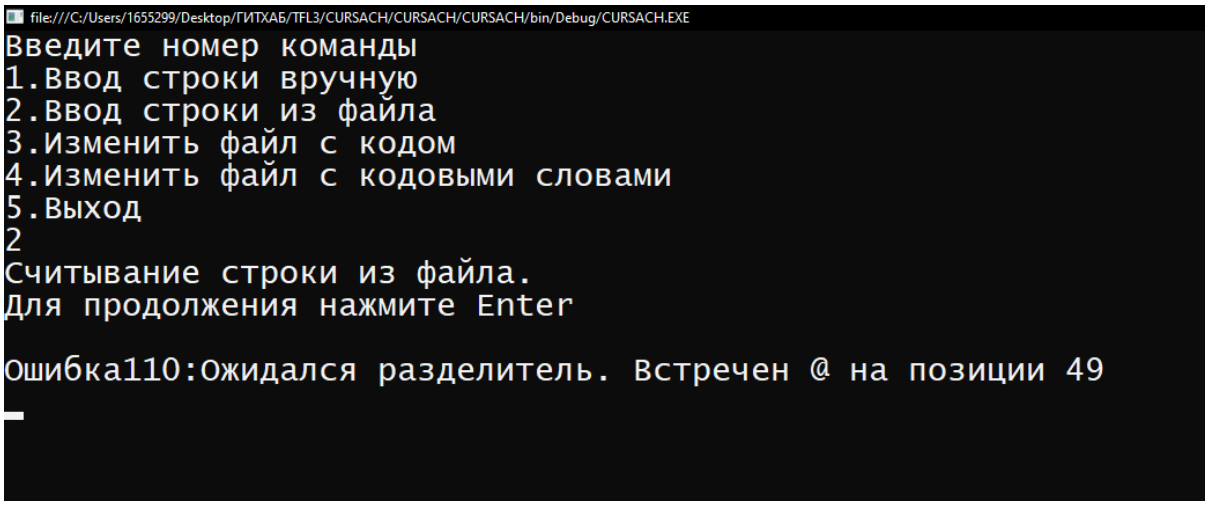

Рисунок 3. Результат работы программ ы при тесте 2

Тест 3. Синтаксическая ошибка

В 3 тесте была допущена умышленная синтаксическая ошибка. В 4 строке кода было убрано слово «then», что нарушает правила грамматики.

```
dim k, l % {тест коммента}
dim a, o !:a as 5d-3o
l as 5.55e-5
if a <= 3d read ( k, l) else write(2d+3h,7d-1b){проверка 2}
end
```

Программа должна выдать ошибку при прохождении синтаксического анализа. В случае успешного тестирования, программа должна выдать ошибку 210, что согласно Таблице 1 характеризуется как «Ожидался then». Результат работы программы при тесте 3 представлен на рисунке 4.

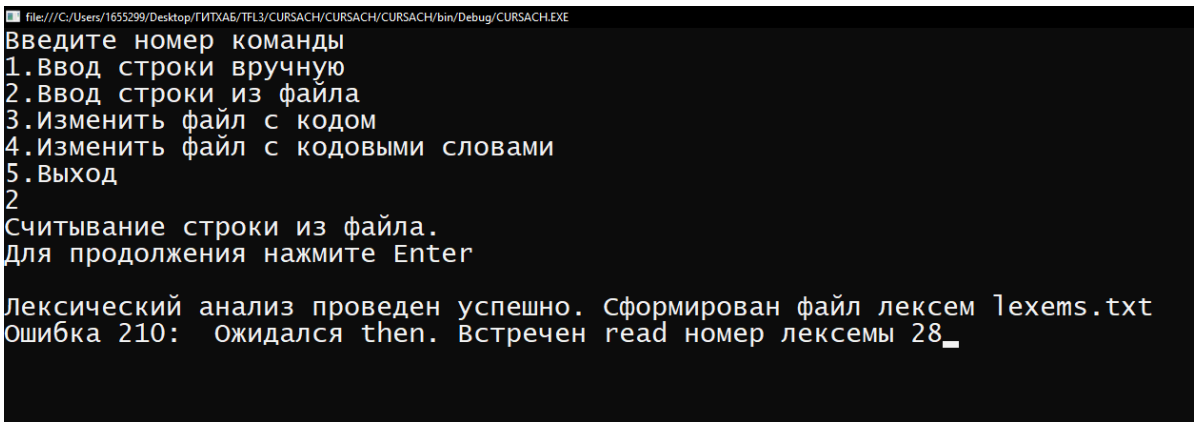

Рисунок 4. Результат работы программ ы при тесте 3

Тест 4. Семантическая ошибка

В 4 тесте была допущена умышленная семантическая ошибка. В 2 строке кода была добавлена повторная инициализация переменной «k», что нарушает семантические условия.

```
dim k, l % {тест коммента}
dim a, o, k !:a as 5d-3o
l as 5.55e-5
if a <= 3d then read ( k, l) else write(2d+3h,7d-1b){проверка 2}
end
```

Программа должна выдать ошибку при прохождении синтаксического анализа на этапе семантической проверки переменных. В случае успешного тестирования, программа должна выдать ошибку 301, что согласно Таблице 1 характеризуется как «Повторное объявление переменной». Результат работы программы при тесте 4 представлен на рисунке 5.

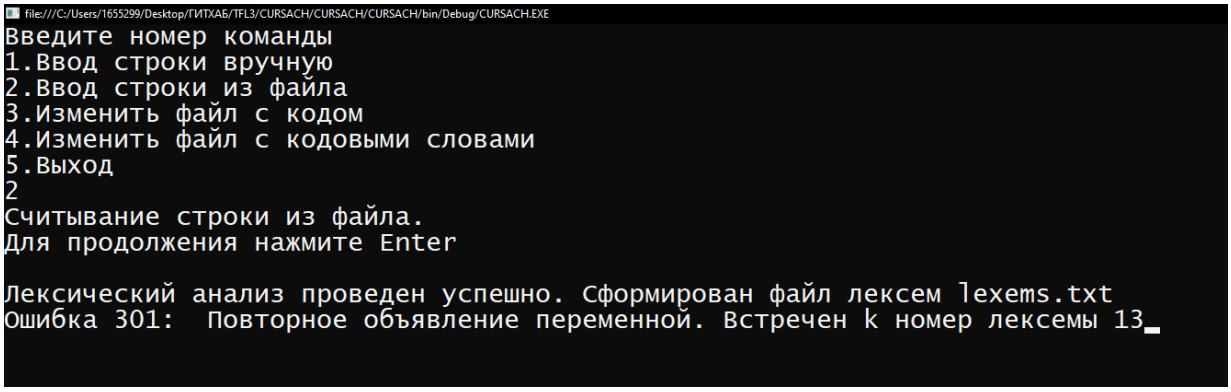

Рисунок 5. Результат работы программ ы при тесте 4

Таким образом, после прохождения всех проверок можно сказать, что программа работает корректно.

ЗАКЛЮЧЕНИЕ

В ходе курсового проектирования был разработан лексический анализатор, разделяющий последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на языке высокого уровня C#.

Разбор исходного текста программы был сделан с помощью синтаксического анализатора, который реализован также на языке C#. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости была преобразована грамматика, в частности, специальным образом обработаны встречающиеся итеративные синтаксически конструкции (нетерминалы D, D1, B, E1 и T).

В код рекурсивных функций включены проверки семантических условий ―проверка на повторное объявление одной и той же переменной и проверка на использование необъявленной переменной.

Тестирование приложения показало, что лексически, синтаксически и семантически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, выдает ошибки с кратким описанием сути ошибки.

В ходе работы изучены основные принципы построения 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. 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.
6. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007.

Приложение А

Код лексического анализатора

```
public static void Lexer()
{
    bool isId = true;
    string buf_word = "";
    string buf_num = "";
    string[,] lexems = new string[0, 2];
    string[] key_word_read;
    string[] delimeters_read;
    int cur_lexem = 0;
    string CurCond = "H";
    word_char = word.ToCharArray();
    key_word_read = File.ReadAllLines("KeyWords.txt");
    key_word = new string[key_word_read.Length];
    for (int i = 0; i < key_word_read.Length; i++)
    {
        key_word[i] = key_word_read[i].Split(' ')[0];
    }
    delimeters_read = File.ReadAllLines("Delimeters.txt");
    delimeters = new string[delimeters_read.Length];
    for (int i = 0; i < delimeters_read.Length; i++)
    {
        delimeters[i] = delimeters_read[i].Split(' ')[0];
    }
    while (cur_pos < word_char.Length)
    {
        switch (CurCond)
        {
            case "H":
                if (word_char[cur_pos] == '\t' ||
word_char[cur_pos] == ' ' || word_char[cur_pos] == '\r')
                {
```

```
                    cur_pos++;
                }
                else if (word_char[cur_pos] == 'e' &&
word_char[cur_pos + 1] == 'n')
                {
                    CurCond = "FIN";
                }
                else if (word_char[cur_pos] == '\n')
                {
                    CurCond = "NewLine";
                }
                else if (word_char[cur_pos] == '<' ||
word_char[cur_pos] == '/' || word_char[cur_pos] == '>' ||
word_char[cur_pos] == '!' || word_char[cur_pos] == ':' ||
word_char[cur_pos] == '=' || word_char[cur_pos] == '-' ||
word_char[cur_pos] == '[' || word_char[cur_pos] == '{' ||
word_char[cur_pos] == '}' || word_char[cur_pos] == ')' ||
word_char[cur_pos] == ',' || word_char[cur_pos] == '*' ||
word_char[cur_pos] == ']' || word_char[cur_pos] == '%' ||
word_char[cur_pos] == '$' || word_char[cur_pos] == '(' ||
word_char[cur_pos] == '+')
                {
                    sost = false;
                    CurCond = "DLM";
                }
                else if (Char.IsLetter(word_char[cur_pos]))
                {
                    buf_word = "";
                    CurCond = "ID";
                }
                else if (Char.IsDigit(word_char[cur_pos]))
                {
                    isLetter = false;
                    buf_num = "";
                    CurCond = "NM";
```

```
                    }
                    else
                    {
                        sost = false;
                        CurCond = "DLM";
                    }
                    break;
                case "NewLine":
                    ResizeArray(ref lexems, cur_lexem + 1, 2);
                    lexems[cur_lexem, 0] = 2.ToString();
                    lexems[cur_lexem, 1] = 22.ToString();
                    cur_lexem++;
                    cur_pos++;
                    CurCond = "H";
                    break;
                case "FIN":
                    if (word_char[cur_pos + 1] == 'n' &&
word_char[cur_pos + 2] == 'd' && word_char.Length == cur_pos +
3)
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        lexems[cur_lexem, 0] = 1.ToString();
                        lexems[cur_lexem, 1] = 7.ToString();
                        cur_lexem++;
                        cur_pos = cur_pos + 3;
                        CurCond = "H";
                    }
                    else
                    {
                        errorLex(102);
                    }
                    break;
                case "ID":
```

```
                    if    (Char.IsDigit(word_char[cur_pos])    ||
Char.IsLetter(word_char[cur_pos]))
                    {
                        buf_word += word_char[cur_pos];
                        if (GetLexem(buf_word)[0] != 0)
                        {
                            ResizeArray(ref  lexems,  cur_lexem +
1, 2);
                            cur_pos++;
                            CurCond = "ID";
                            lexems[cur_lexem,         0]         =
GetLexem(buf_word)[0].ToString();
                            lexems[cur_lexem,         1]         =
GetLexem(buf_word)[1].ToString();
                            isId = true;
                        }
                        else
                        {
                            ResizeArray(ref  lexems,  cur_lexem +
1, 2);
                            isId = false;
                            lexems[cur_lexem, 0] = 3.ToString();
                            lexems[cur_lexem,  1]  =  (cur_ID  +
1).ToString();
                            cur_pos++;
                            CurCond = "ID";
                        }
                    }
                    else
                    {
                        if (!isId)
                        {
                            Array.Resize(ref     identificators,
cur_ID + 1);
                            identificators[cur_ID] = buf_word;
```

```
                        cur_ID++;
                    }
                    cur_lexem++;
                    CurCond = "H";
                }
                break;
            case "NM":
                if (word_char[cur_pos] == '0' ||
word_char[cur_pos] == '1')
                {
                    CurCond = "BIN";
                }
                else if
(Convert.ToInt32(Convert.ToString(word_char[cur_pos])) > 1 &&
Convert.ToInt32(Convert.ToString(word_char[cur_pos])) < 8)
                {
                    CurCond = "OCT";
                }
                else if (Convert.ToInt32(word_char[cur_pos])
> 7 || IsHex(word_char[cur_pos]))
                {
                    CurCond = "DECHEX";
                }
                else
                {
                    errorLex(101);
                }
                break;
            case "BIN":
                buf_num += word_char[cur_pos];
                if (word_char[cur_pos] != 'O' &&
word_char[cur_pos] != 'o' && word_char[cur_pos] != 'd' &&
word_char[cur_pos] != 'D' && word_char[cur_pos] != 'H' &&
word_char[cur_pos] != 'h' && word_char[cur_pos] != 'B' &&
word_char[cur_pos] != 'b' && word_char[cur_pos] != 'E' &&
```

```
word_char[cur_pos] != 'e' && word_char[cur_pos] != '.' &&
!Char.IsDigit(word_char[cur_pos]))
                    {
                        errorLex(114);
                    }
                    if (word_char[cur_pos] == 'O' ||
word_char[cur_pos] == 'o')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem, 1] = (cur_num +
1).ToString(); ;
                        numbers[cur_num] = GetNum(buf_num,
ss.OCT);
                        cur_lexem++;
                        cur_num++;
                    }
                    else if (word_char[cur_pos] == 'd' ||
word_char[cur_pos] == 'D')
                    {
                        if (word_char[cur_pos] == 'd' &&
word_char[cur_pos + 1] == 'i')
                        {
                            errorLex(107);
                        }
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
```

```
                        lexems[cur_lexem,   1]   =   (cur_num   +
1).ToString();
                        numbers[cur_num]                        =
buf_num.Remove(buf_num.Length - 1);
                        cur_lexem++;
                        cur_num++;
                    }
                    else   if   (word_char[cur_pos]   ==   'H'   ||
word_char[cur_pos] == 'h')
                    {
                        ResizeArray(ref  lexems,  cur_lexem  +  1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem,   1]   =   (cur_num   +
1).ToString(); ;
                        numbers[cur_num]    =    GetNum(buf_num,
ss.HEX);
                        cur_lexem++;
                        cur_num++;
                    }
                    else   if   (word_char[cur_pos]   ==   'B'   ||
word_char[cur_pos] == 'b')
                    {
                        ResizeArray(ref  lexems,  cur_lexem  +  1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem,   1]   =   (cur_num   +
1).ToString(); ;
```

```
                        numbers[cur_num]    =    GetNum(buf_num,
ss.BIN);
                        cur_lexem++;
                        cur_num++;
                    }
                    else   if   (word_char[cur_pos]   ==   'e'   ||
word_char[cur_pos] == 'E' || word_char[cur_pos] == '.')
                    {
                        cur_pos--;
                        buf_num  =  buf_num.Remove(buf_num.Length
- 1);
                        CurCond = "EXP";
                    }
                    else                                        if
(Convert.ToInt32(Convert.ToString(word_char[cur_pos]))   >   1   &&
Convert.ToInt32(Convert.ToString(word_char[cur_pos])) < 8)
                    {
                        CurCond = "OCT";
                    }
                    else   if   (word_char[cur_pos]   ==   '0'   ||
word_char[cur_pos] == '1')
                    {
                        CurCond = "BIN";
                    }
                    else                                        if
(Convert.ToInt32(Convert.ToString(word_char[cur_pos]))   >   7   ||
IsHex(word_char[cur_pos]))
                    {
                        CurCond = "DECHEX";
                    }
                    else if (Char.IsDigit(word_char[cur_pos]))
                    {

                    }
                    else
```

```
                    {
                        errorLex(103);
                    }
                    cur_pos++;
                    break;
                case "OCT":
                    buf_num += word_char[cur_pos];
                    if (word_char[cur_pos] != 'O' &&
word_char[cur_pos] != 'o' && word_char[cur_pos] != 'd' &&
word_char[cur_pos] != 'D' && word_char[cur_pos] != 'H' &&
word_char[cur_pos] != 'h' && word_char[cur_pos] != 'B' &&
word_char[cur_pos] != 'b' && word_char[cur_pos] != 'E' &&
word_char[cur_pos] != 'e' && word_char[cur_pos] != '.' &&
!Char.IsDigit(word_char[cur_pos]))
                    {
                        errorLex(114);
                    }
                    if (word_char[cur_pos] == 'O' ||
word_char[cur_pos] == 'o')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem, 1] = (cur_num +
1).ToString(); ;
                        numbers[cur_num] = GetNum(buf_num,
ss.OCT);
                        cur_lexem++;
                        cur_num++;
                    }
                    else if (word_char[cur_pos] == 'd' ||
word_char[cur_pos] == 'D')
```

```
                    {
                        if (word_char[cur_pos] == 'd' &&
word_char[cur_pos + 1] == 'i')
                        {
                            errorLex(107);
                        }
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem, 1] = (cur_num +
1).ToString();
                        numbers[cur_num] =
buf_num.Remove(buf_num.Length - 1);
                        cur_lexem++;
                        cur_num++;
                    }
                    else if (word_char[cur_pos] == 'H' ||
word_char[cur_pos] == 'h')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem, 1] = (cur_num +
1).ToString();
                        numbers[cur_num] = GetNum(buf_num,
ss.HEX);
                        cur_lexem++;
                        cur_num++;
                    }
```

```
                    else if (word_char[cur_pos] == 'e' ||
word_char[cur_pos] == 'E' || word_char[cur_pos] == '.')
                    {
                        cur_pos--;
                        buf_num = buf_num.Remove(buf_num.Length
- 1);
                        CurCond = "EXP";
                    }
                    else                                    if
(Convert.ToInt32(Convert.ToString(word_char[cur_pos])) > 1 &&
Convert.ToInt32(Convert.ToString(word_char[cur_pos])) < 8)
                    {
                        CurCond = "OCT";
                    }
                    else                                    if
(Convert.ToInt32(Convert.ToString(word_char[cur_pos])) > 7 ||
IsHex(word_char[cur_pos]))
                    {
                        CurCond = "DECHEX";
                    }
                    else if (Char.IsDigit(word_char[cur_pos]))
                    {

                    }
                    else
                    {
                        errorLex(104);
                    }
                    cur_pos++;
                    break;
                case "EXP":
                    if    (word_char[cur_pos]    ==    'd'    ||
word_char[cur_pos] == 'D' || word_char[cur_pos] == 'h' ||
word_char[cur_pos] == 'H' || word_char[cur_pos] == 'o' ||
```

```
word_char[cur_pos] == 'O' || word_char[cur_pos] == 'b' ||
word_char[cur_pos] == 'B')
                    {
                        errorLex(112);
                    }
                    if (word_char[cur_pos] == 'e' ||
word_char[cur_pos] == 'E')
                    {
                        isLetter = true;
                        buf_num += word_char[cur_pos];
                        exp = true;
                    }
                    else if (word_char[cur_pos] == '.' && exp ==
true)
                    {
                        errorLex(105);
                    }
                    else if (word_char[cur_pos] == '.' && exp ==
false)
                    {
                        buf_num += word_char[cur_pos];
                    }
                    else if (Char.IsDigit(word_char[cur_pos]))
                    {
                        buf_num += word_char[cur_pos];
                    }
                    else if ((word_char[cur_pos] == '+' ||
word_char[cur_pos] == '-') && exp == false)
                    {
                        errorLex(106);
                    }
                    else if ((word_char[cur_pos] == '+' ||
word_char[cur_pos] == '-'))
                    {
                        if (!symb)
```

```
                        {
                            buf_num += word_char[cur_pos];
                            symb = true;
                        }
                        else
                        {
                            ResizeArray(ref lexems, cur_lexem +
1, 2);
                            Array.Resize(ref numbers, cur_num +
1);
                            lexems[cur_lexem, 0] = 4.ToString();
                            lexems[cur_lexem, 1] = (cur_num +
1).ToString();
                            numbers[cur_num] = buf_num;
                            cur_lexem++;
                            cur_num++;
                            CurCond = "H";
                            cur_pos--;
                        }
                    }
                    else if (word_char[cur_pos] == '\n' ||
word_char[cur_pos] == ':')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem, 1] = (cur_num +
1).ToString();
                        numbers[cur_num] =
buf_num.Remove(buf_num.Length - 1);
                        cur_lexem++;
                        cur_num++;
                        if (word_char[cur_pos] == '\n')
                        {
```

```
                            ResizeArray(ref lexems, cur_lexem +
1, 2);
                            lexems[cur_lexem, 0] = 2.ToString();
                            lexems[cur_lexem,        1]        =
22.ToString();
                            cur_lexem++;
                        }
                        CurCond = "H";
                    }
                    else  if  (Char.IsLetter(word_char[cur_pos])
&& (word_char[cur_pos] != 'e' || word_char[cur_pos] != 'E'))
                    {
                        errorLex(113);
                    }
                    else
                    {
                        buf_num += word_char[cur_pos];
                    }
                    cur_pos++;
                    break;
                case "DECHEX":
                    buf_num += word_char[cur_pos];
                    if    (word_char[cur_pos]    !=    'O'    &&
word_char[cur_pos]  !=  'o'  &&  word_char[cur_pos]  !=  'd'  &&
word_char[cur_pos]  !=  'D'  &&  word_char[cur_pos]  !=  'H'  &&
word_char[cur_pos]  !=  'h'  &&  word_char[cur_pos]  !=  'B'  &&
word_char[cur_pos]  !=  'b'  &&  word_char[cur_pos]  !=  'E'  &&
word_char[cur_pos]  !=  'e'  &&  word_char[cur_pos]  !=  '.'  &&
!Char.IsDigit(word_char[cur_pos]))
                    {
                        errorLex(114);
                    }
                    if    (word_char[cur_pos]    ==    'H'    ||
word_char[cur_pos] == 'h')
                    {
```

```
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem,  1]  =  (cur_num  +
1).ToString(); ;
                        numbers[cur_num]   =   GetNum(buf_num,
ss.HEX);
                        cur_lexem++;
                        cur_num++;
                    }
                    else  if  (word_char[cur_pos]  ==  'd'  ||
word_char[cur_pos] == 'D')
                    {
                        if  (word_char[cur_pos]  ==  'd'  &&
word_char[cur_pos + 1] == 'i')
                        {
                            errorLex(107);
                        }
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        Array.Resize(ref numbers, cur_num + 1);
                        isLetter = true;
                        CurCond = "H";
                        lexems[cur_lexem, 0] = 4.ToString();
                        lexems[cur_lexem,  1]  =  (cur_num  +
1).ToString();
                        numbers[cur_num]                      =
buf_num.Remove(buf_num.Length - 1);
                        cur_lexem++;
                        cur_num++;
                    }
```

```
                    else if (word_char[cur_pos] == 'e' ||
word_char[cur_pos] == 'E' || word_char[cur_pos] == '.')
                    {
                        cur_pos--;
                        buf_num = buf_num.Remove(buf_num.Length
- 1);
                        CurCond = "EXP";
                    }
                    else if (Char.IsDigit(word_char[cur_pos]))
                    {

                    }
                    else
                    {
                        errorLex(108);
                    }
                    cur_pos++;
                    break;
                case "DLM":
                    if (word_char[cur_pos] == '(' ||
word_char[cur_pos] == ')' || word_char[cur_pos] == ':' ||
word_char[cur_pos] == '[' || word_char[cur_pos] == ',' ||
word_char[cur_pos] == ']')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        CurCond = "H";
                        lexems[cur_lexem, 0] =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
                        lexems[cur_lexem, 1] =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                        cur_pos++;
                        cur_lexem++;
                    }
```

```
                    else if (word_char[cur_pos] == '$' ||
word_char[cur_pos] == '!' || word_char[cur_pos] == '%')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        CurCond = "H";
                        lexems[cur_lexem,          0]          =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
                        lexems[cur_lexem,          1]          =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                        cur_pos++;
                        cur_lexem++;
                    }
                    else if (word_char[cur_pos] == '+' ||
word_char[cur_pos] == '-' || word_char[cur_pos] == '*' ||
word_char[cur_pos] == '/')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        CurCond = "H";
                        lexems[cur_lexem,          0]          =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
                        lexems[cur_lexem,          1]          =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                        cur_pos++;
                        cur_lexem++;
                    }
                    else if (word_char[cur_pos] == '<' ||
word_char[cur_pos] == '>')
                    {
                        if (sost == true)
                        {
                            if (word_char[cur_pos - 1] == '<' &&
word_char[cur_pos] == '>')
                            {
```

```
                                ResizeArray(ref lexems,
cur_lexem + 1, 2);
                                CurCond = "H";
                                lexems[cur_lexem, 0] =
GetLexem(word_char[cur_pos - 1] + "" +
word_char[cur_pos])[0].ToString();
                                lexems[cur_lexem, 1] =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                                cur_pos++;
                                cur_lexem++;
                            }
                            else if (word_char[cur_pos - 1] ==
'>' && word_char[cur_pos] == '<')
                            {
                                errorLex(109);
                            }
                        }
                        else
                        {
                            sost = true;
                            cur_pos++;
                        }
                    }
                    else if (word_char[cur_pos] == '=')
                    {
                        if (sost == true)
                        {
                            ResizeArray(ref lexems, cur_lexem +
1, 2);
                            CurCond = "H";
                            lexems[cur_lexem, 0] =
GetLexem(word_char[cur_pos - 1] + "" +
word_char[cur_pos])[0].ToString();
                            lexems[cur_lexem, 1] =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
```

```
                            cur_pos++;
                            cur_lexem++;
                            sost = false;
                        }
                        else
                        {
                            ResizeArray(ref lexems, cur_lexem +
1, 2);
                            CurCond = "H";
                            lexems[cur_lexem,        0]        =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
                            lexems[cur_lexem,        1]        =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                            cur_pos++;
                            cur_lexem++;
                        }
                    }
                    else if (word_char[cur_pos] == '{')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        lexems[cur_lexem,          0]          =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
                        lexems[cur_lexem,          1]          =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                        cur_pos++;
                        cur_lexem++;
                    }
                    else if (word_char[cur_pos] == '}')
                    {
                        ResizeArray(ref lexems, cur_lexem + 1,
2);
                        CurCond = "H";
                        lexems[cur_lexem,          0]          =
GetLexem(word_char[cur_pos].ToString())[0].ToString();
```

```
                        lexems[cur_lexem,          1]          =
GetLexem(word_char[cur_pos].ToString())[1].ToString();
                        cur_pos++;
                        cur_lexem++;
                    }
                    else if (Char.IsDigit(word_char[cur_pos]) ||
Char.IsLetter(word_char[cur_pos]) || word_char[cur_pos] == ' '
|| word_char[cur_pos] == '\n' || word_char[cur_pos] == '\r' ||
word_char[cur_pos] == '\t')
                    {
                        cur_pos++;
                    }
                    else
                    {
                        errorLex(110);
                    }
                    break;
                }
            }
            if (isLetter == false)
            {
                errorLex(111);
            }
            for (int i = 0; i < cur_lexem; i++)
            {
                string str = lexems[i, 0] + " " + lexems[i, 1] +
"\n";
                File.AppendAllText("lexems.txt", str);
            }
            for (int i = 0; i < numbers.Length; i++)
            {
                if (numbers[i] != null)
                {
                    string str = numbers[i] + " " + (i + 1) + "\n";
                    File.AppendAllText("Numbers.txt", str);
```

```
            }
        }
        for (int i = 0; i < identificators.Length; i++)
        {
            if (identificators[i] != null)
            {
                string str = identificators[i] + " " + (i + 1) +
"\n";
                File.AppendAllText("Identificators.txt", str);
            }
        }
    }
```

Приложение Б

Код синтаксического анализатора

(с дополнительными семантическими проверками)

```
public static void Parser()
{
    Fill();
    prog();
}
public static void Fill()
{
    string[] cur_lex = File.ReadAllLines("lexems.txt");
    num = new string[cur_lex.Length];
    val = new string[cur_lex.Length]; ;
    for (int i = 0; i < cur_lex.Length; i++)
    {
        num[i] = cur_lex[i].Split(' ')[0];
        val[i] = cur_lex[i].Split(' ')[1];
    }
}
public static void get_lexem()
{
    int counter = 1;
    try
    {
        switch (num[cur_lex_lexem])
        {
            case "1":
                foreach (var s in key_word)
                {
```

```
                    if      (counter.ToString()      ==
val[cur_lex_lexem])
                    {
                        cur_lex_lexem_word = s;
                    }
                    counter++;
                }
                break;
            case "2":
                foreach (var s in delimeters)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem])
                    {
                        cur_lex_lexem_word = s;
                    }
                    counter++;
                }
                break;
            case "3":
                foreach (var s in identificators)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem])
                    {
                        cur_lex_lexem_word = s;
                    }
                    counter++;
                }
                break;
```

```
            case "4":
                foreach (var s in numbers)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem])
                    {
                        cur_lex_lexem_word = s;
                    }
                    counter++;
                }
                break;
            default:
                ErrorParser(201);
                break;

        }
    }
    catch (IndexOutOfRangeException)
    {
        ErrorParser(204);
    }
    cur_lexem_number = cur_lex_lexem;
    cur_lex_lexem++;
}
public static bool equals(string S)
{
    int counter = 1;
    switch (num[cur_lex_lexem - 1])
    {
        case "1":
```

```
            foreach (var i in key_word)
            {
                if (i == S)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem - 1])
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                counter++;
            }
            break;
        case "2":
            foreach (var i in delimeters)
            {
                if (i == S)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem - 1])
                    {
                        return true;
                    }
                    else
                    {
                        return false;
```

```
                    }
                }
                counter++;
            }
            break;
        case "3":
            foreach (var i in identificators)
            {
                if (i == S)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem - 1])
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                counter++;
            }
            break;
        case "4":
            foreach (var i in numbers)
            {
                if (i == S)
                {
                    if      (counter.ToString()      ==
val[cur_lex_lexem - 1])
```

```
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                counter++;
            }
            break;
        default:
            return false;
    }
    return false;
}
public static bool ID()
{
    if (num[cur_lex_lexem - 1] == "3")
    {
        return true;
    }
    else
    {
        return false;
    }
}
public static bool numer()
{
    if (num[cur_lex_lexem - 1] == "4")
```

```
    {
        return true;
    }
    else
    {
        return false;
    }
}
public static void add()
{
    Array.Resize(ref declared_identificators,
declared_identificators.Length + 1);

declared_identificators[declared_identificators.Length
- 1] = val[cur_lex_lexem - 1];
}
public static bool check()
{
    foreach (var i in declared_identificators)
    {
        if
(identificators[Convert.ToInt32(val[cur_lex_lexem - 1])
- 1] == identificators[Convert.ToInt32(i) - 1] &&
num[cur_lex_lexem - 1] == "3")
        {
            return true;
        }
    }
    return false;
}
```

```
public static void prog() //<программа> = {/
(<описание> | <оператор>) ( : | переход строки) /} end
{
    do
    {
        get_lexem();
        if (equals("{"))
        {
            comment();
        }
        if (equals("dim"))
        {
            descrip();
        }
        else if (equals("[") || equals("if") ||
equals("for") || equals("while") || equals("read") ||
equals("write") || ID())
        {
            oper();
        }
        else if (equals("end"))
        {
            break;
        }
        else
        {
            ErrorParser(202);
        }
        if (equals("{"))
        {
```

```
            comment();
        }
        if (!equals(":") && !equals("\\n"))
        {

            ErrorParser(203);
        }
    } while (equals(":") || equals("\\n"));
    if (equals("{"))
    {
        comment();
    }
    if (!equals("end"))
    {
        ErrorParser(204);
    }
}
public   static   void   descrip()//<описание>::=   dim
<идентификатор> {, <идентификатор> } <тип>
{
    if (equals("{"))
    {
        comment();
    }
    if (equals("dim"))
    {
        get_lexem();
        if (check())
        {
            ErrorParser(301);
```

```
}
else
{
    add();
}
get_lexem();
if (equals("{"))
{
    comment();
}
while (equals(","))
{
    if (equals("{"))
    {
        comment();
    }
    get_lexem();
    if (!ID())
    {
        ErrorParser(205);
    }
    else
    {
        if (check())
        {
            ErrorParser(301);
        }
        else
        {
            add();
```

```
                }
            }
            get_lexem();
        }
        type();
    }
    else
    {
        ErrorParser(206);
    }
}
public static void oper()//<оператор>::=  <составной> | <присваивания> | <условный> |<фиксированного_цикла> | <условного_цикла> | <ввода> |<вывода>
{
    if (equals("{"))
    {
        comment();
    }
    if (equals("["))
    {
        compare_oper();
    }
    else if (equals("if"))
    {
        if_oper();
    }
    else if (equals("for"))
    {
        for_cicle();
```

```
    }
    else if (equals("while"))
    {
        while_cicle();
    }
    else if (equals("read"))
    {
        input();
    }
    else if (equals("write"))
    {
        output();
    }
    else if (ID())
    {
        assign_oper();
    }
}
public static void type()//<тип>::= % | ! | $
{
    if (equals("{"))
    {
        comment();
    }
    if (!equals("%") && !equals("!") && !equals("$"))
    {
        ErrorParser(207);
    }
    get_lexem();
}
```

```
public static void compare_oper()//<составной>::= «[»
<оператор> { ( : | перевод строки) <оператор> } «]»
{
    do
    {
        if (equals("{"))
        {
            comment();
        }
        get_lexem();
        oper();
    } while (equals(":") || equals("\\n"));
    if (equals("{"))
    {
        comment();
    }
    if (!equals("]"))
        ErrorParser(208);
    get_lexem();
}
public static void assign_oper()//<присваивания>::=
<идентификатор> as <выражение>
{
    if (equals("{"))
    {
        comment();
    }
    if (!check())
    {
        ErrorParser(302);
```

```
    }
    get_lexem();
    if (equals("{"))
    {
        comment();
    }
    if (!equals("as"))
    {
        ErrorParser(209);
    }
    get_lexem();
    expression();
}
public   static   void   if_oper()//<условный>::=   if
<выражение> then <оператор> [ else <оператор>]
{
    if (equals("{"))
    {
        comment();
    }
    get_lexem();
    expression();
    if (equals("{"))
    {
        comment();
    }
    if (!equals("then"))
    {
        ErrorParser(210);
    }
```

```
    get_lexem();
    oper();
    if (equals("{"))
    {
        comment();
    }
    if (equals("else"))
    {
        get_lexem();
        oper();
    }
}
public static void for_cicle()//<фиксированного_цикла>::= for <присваивания> to <выражение> do <оператор>
{
    if (equals("{"))
    {
        comment();
    }
    get_lexem();
    if (ID())
    {
        assign_oper();
    }
    if (equals("{"))
    {
        comment();
    }
    if (!equals("to"))
```

```
    {
        ErrorParser(211);
    }
    get_lexem();
    if (equals("{"))
    {
        comment();
    }
    if   (ID()   ||   numer()   ||   equals("true")   ||
equals("false") || equals("not") || equals("("))
    {
        if (ID() && !check())
            ErrorParser(302);
        expression();
    }
    else
    {
        ErrorParser(212);
    }
    if (equals("{"))
    {
        comment();
    }
    if (!equals("do"))
    {
        ErrorParser(213);
    }
    get_lexem();
    oper();
}
```

```
public static void while_cicle()//<условного_цикла>::=
while <выражение> do <оператор>
{
    if (equals("{"))
    {
        comment();
    }
    if (ID() || numer() || equals("true") ||
equals("false") || equals("not") || equals("("))
    {
        if (ID() && !check())
            ErrorParser(302);
        expression();
    }
    else
    {
        ErrorParser(212);
    }
    get_lexem();
    if (equals("{"))
    {
        comment();
    }
    if (!equals("do"))
    {
        ErrorParser(213);
    }
    if (equals("[") || equals("if") || equals("for") ||
equals("while") || equals("read") || equals("write") ||
ID())
```

```
    {
        oper();
    }
    else
    {
        ErrorParser(214);
    }
}
public    static    void    input()//<ввода>::=    read
«(»<идентификатор> {, <идентификатор> } «)»
{
    if (equals("{"))
    {
        comment();
    }
    get_lexem();
    if (!equals("("))
        ErrorParser(215);
    get_lexem();
    if (equals("{"))
    {
        comment();
    }
    if (!ID())
    {
        ErrorParser(205);
    }
    get_lexem();
    if (equals("{"))
    {
```

```
        comment();
    }
    while (equals(","))
    {
        if (equals("{"))
        {
            comment();
        }
        get_lexem();
        if (ID() && !check())
            ErrorParser(302);
        get_lexem();
        if (equals("{"))
        {
            comment();
        }
    }
    if (!equals(")"))
        ErrorParser(217);
    get_lexem();
}
public   static   void   output()//<вывода>::=   write
«(»<выражение> {, <выражение> } «)»
{
    if (equals("{"))
    {
        comment();
    }
    get_lexem();
    if (equals("{"))
```

```
    {
        comment();
    }
    if (!equals("("))
        ErrorParser(215);
    get_lexem();
    if   (ID()   ||   numer()   ||   equals("true")   ||
equals("false") || equals("not") || equals("("))
    {
        if (ID() && !check())
            ErrorParser(302);
        expression();
        if (equals("{"))
        {
            comment();
        }
        while (equals(","))
        {
            if (equals("{"))
            {
                comment();
            }
            get_lexem();
            if (equals("{"))
            {
                comment();
            }
            if  (ID()  ||  numer()  ||  equals("true")  ||
equals("false") || equals("not") || equals("("))
            {
```

```
                if (ID() && !check())
                    ErrorParser(302);
                expression();
            }
            else
            {
                ErrorParser(212);
            }

        }
    }
    else
    {
        ErrorParser(212);
    }
    if (equals("{"))
    {
        comment();
    }
    if (!equals(")"))
        ErrorParser(217);
    get_lexem();
}
public   static   void   expression()//<выражение>::   =
    <операнд>{ <операции_группы_отношения> <операнд> }
{
    if (equals("{"))
    {
        comment();
    }
```

```
    if   (ID()   ||   numer()   ||   equals("true")   ||
equals("false") || equals("not") || equals("("))
    {
        if (equals("{"))
        {
            comment();
        }
        if (ID() && !check())
            ErrorParser(302);
        operand();
    }
    else ErrorParser(214);

    if (equals("{"))
    {
        comment();
    }
    if (equals("< >") || equals("=") || equals("<") ||
equals("<=") || equals(">") || equals(">="))
    {
        ratio();
        if (equals("{"))
        {
            comment();
        }
        if  (ID()  ||  numer()  ||  equals("true")  ||
equals("false") || equals("not") || equals("("))
        {
            if (equals("{"))
            {
```

```
                comment();
            }
            if (ID() && !check())
                ErrorParser(302);
            operand();
        }
        else ErrorParser(214);
    }
}
public static void operand()//<операнд>::=
    <слагаемое>              {<операции_группы_сложения>
<слагаемое>}
{
    if (equals("{"))
    {
        comment();
    }
    if   (ID()   ||   numer()   ||   equals("true")   ||
equals("false") || equals("not") || equals("("))
    {
        if (ID() && !check())
            ErrorParser(302);
        summand();
    }
    else ErrorParser(214);

    if (equals("{"))
    {
        comment();
    }
```

```
    if (equals("+") || equals("-") || equals("or"))
    {
        addition();
        if (equals("{"))
        {
            comment();
        }
        if (ID() || numer() || equals("true") ||
equals("false") || equals("not") || equals("("))
        {
            if (ID() && !check())
                ErrorParser(302);
            summand();
        }
        else ErrorParser(214);
    }
}
public static void summand()//<слагаемое>::=
    <множитель> {<операции_группы_умножения>
<множитель>}
{
    if (equals("{"))
    {
        comment();
    }
    if (ID() || numer() || equals("true") ||
equals("false") || equals("not") || equals("("))
    {
        if (ID() && !check())
            ErrorParser(302);
```

```
        multiply();
    }
    else ErrorParser(214);
    if (equals("{"))
    {
        comment();
    }
    if (equals("*") || equals("/") || equals("and"))
    {
        multiplicate();
        if (equals("{"))
        {
            comment();
        }
        if (ID() || numer() || equals("true") ||
equals("false") || equals("not") || equals("("))
        {
            if (ID() && !check())
                ErrorParser(302);
            multiply();
        }
        else ErrorParser(214);
    }
}
public static void multiply()//<множитель>::=
    <идентификатор> | <число> | <логическая_константа>
|<унарная_операция> <множитель> | «(»<выражение>«)»
{
    if (equals("{"))
    {
```

```
        comment();
    }
    if (equals("("))
    {
        get_lexem();
        expression();
        if (equals("{"))
        {
            comment();
        }
        if (!equals(")"))
            ErrorParser(217);

    }
    if (equals("{"))
    {
        comment();
    }
    else if (equals("not"))
    {
        unar();
    }
    get_lexem();
}
public static void number()//<число>:: =       <целое> |
<действительное>
{
    if (equals("{"))
    {
        comment();
```

```
    }
    if (!numer())
        ErrorParser(219);
    get_lexem();
}
public                  static                  void
ratio()//<операции_группы_отношения>:: = < > | = | < |
<= | > | >=
{
    if (equals("{"))
    {
        comment();
    }
    if (!(equals("< >") || equals("=") || equals("<")
|| equals("<=") || equals(">") || equals(">=")))
        ErrorParser(219);
    get_lexem();
}
public                  static                  void
addition()//<операции_группы_сложения>:: = + | - | or
{
    if (equals("{"))
    {
        comment();
    }
    if (!(equals("+") || equals("-") || equals("or")))
        ErrorParser(216);
    get_lexem();
}
```

```
public                    static                    void
multiplicate()//<операции_группы_умножения>::= * | / |
and
{
    if (equals("{"))
    {
        comment();
    }
    if (!(equals("*") || equals("/") || equals("and")))
        ErrorParser(201);
    get_lexem();
}
public static void unar()//<унарная_операция>::= not
{
    if (equals("{"))
    {
        comment();
    }
    if (!equals("not"))
        ErrorParser(223);
    get_lexem();
}
public                    static                    void
comment()//<комментарий>::="{"<символ>"}"
{
    while (!equals("}"))
    {
        get_lexem();
    }
    get_lexem();
```

```
}
```