МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Ижевский государственный технический университет имени М. Т. Калашникова» (ФГБОУ ВО «ИжГТУ имени М.Т. Калашникова»)

Факультет «Информатика и вычислительная техника»

Кафедра «Автоматизированные системы обработки информации и управления»

ОТЧЁТ

По лабораторной работе №2

по дисциплине «Формальные языки и трансляторы»

на тему: «Разработка синтаксического анализатора.»

Выполнил

Студент группы Б19-782-2 И.С. Пономарёв

Руководитель

К.т.н. доцент кафедры «АСОИУ» Д. Р. Касимов

г. Ижевск 2022

# Постановка задачи

Задание по лабораторной работе заключается в разработке синтаксического анализатора для предложенного варианта контекстно-свободной грамматики (КС-грамматики) методом предсказывающего рекурсивного спуска.

Данная КС-грамматика:

D → D ∨ K | K

K → K ∧ A | A

A → ¬ A | ( D ) | O

O → <1>=<2> | <2>=<1> | <2>=<2>

Где <1> и <2> слова из первой лабораторной работы.

# Преобразование грамматики

D → K T

T → ∨ K T | ε

K → A E

E → ∧ A E | ε

A → ¬ A | ( D ) | O

O → <1>=<2> | <2>=<1> | <2>=<2>

Рассчитаем nullable и First для нашей получившейся КС-грамматики

|  |  |
| --- | --- |
|  | Nullable? |
| D | Нет |
| T | Да |
| K | Нет |
| E | Да |
| A | Нет |
| O | Нет |

First(KT) ⊇ First(K) ⊇ First(AE) ⊇ First(A) ⊇ First(¬A) ⊇ First(¬)={¬}

⊇ First((D)) ⊇ First(()={(}

⊇ First(O) ⊇ First(<1>=<2>)={<1>}

⊇ First(<2>=<1>)={<2>}

⊇ First(<2>=<2>)={<2>}

First(∨ K) ⊇ First(∨) = {∨}

First(∧A) ⊇ First(∧) = {∧}

Это правила соответствуют грамматике класса LL(1)

# Код программы

Ниже представлен код программы

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LAB2

{

// Класс исключительных ситуаций синтаксического анализа.

class SynAnException : Exception

{

// Позиция возникновения исключительной ситуации в анализируемом тексте.

private int lineIndex; // Индекс строки.

private int symIndex; // Индекс символа.

// Индекс строки, где возникла исключительная ситуация - свойство только для чтения.

public int LineIndex

{

get { return lineIndex; }

}

// Индекс символа, на котором возникла исключительная ситуация - свойство только для чтения.

public int SymIndex

{

get { return symIndex; }

}

// Конструктор исключительной ситуации.

// message - описание исключительной ситуации.

// lineIndex и symIndex - позиция возникновения исключительной ситуации в анализируемом тексте.

public SynAnException(string message, int lineIndex, int symIndex)

: base(message)

{

this.lineIndex = lineIndex;

this.symIndex = symIndex;

}

}

// Класс "Синтаксический анализатор".

// При обнаружении ошибки в исходном тексте он генерирует исключительную ситуацию SynAnException или LexAnException.

class SyntaxAnalyzer

{

private LexicalAnalyzer lexAn; // Лексический анализатор.

// Конструктор синтаксического анализатора.

// В качестве параметра передается исходный текст.

public SyntaxAnalyzer(string inputLines)

{

// Создаем лексический анализатор.

// Передаем ему текст.

lexAn = new LexicalAnalyzer(inputLines);

}

// Обработать синтаксическую ошибку.

// msg - описание ошибки.

private void SyntaxError(string msg)

{

// Генерируем исключительную ситуацию, тем самым полностью прерывая процесс анализа текста.

throw new SynAnException(msg, lexAn.CurLineIndex, lexAn.CurSymIndex);

}

// Проверить, что тип текущего распознанного токена совпадает с заданным.

// Если совпадает, то распознать следующий токен, иначе синтаксическая ошибка.

private void Match(TokenKind tkn)

{

if (lexAn.Token.Type == tkn) // Сравниваем.

{

lexAn.RecognizeNextToken(); // Распознаем следующий токен.

}

else

{

SyntaxError("Ожидалось " + tkn.ToString()); // Обнаружена синтаксическая ошибка.

}

}

// Проверить, что текущий распознанный токен совпадает с заданным (сравнение производится в нижнем регистре).

// Если совпадает, то распознать следующий токен, иначе синтаксическая ошибка.

private void Match(string tkn)

{

if (lexAn.Token.Value.ToLower() == tkn.ToLower()) // Сравниваем.

{

lexAn.RecognizeNextToken(); // Распознаем следующий токен.

}

else

{

SyntaxError("Ожидалось " + tkn); // Обнаружена синтаксическая ошибка.

}

}

// Процедура разбора для стартового нетерминала E.

private void D()

{

K();

T(); // Вызываем процедуру разбора для нетерминала T.

}

private void K()

{

A();

E();

}

private void E()

{

if (lexAn.Token.Type == TokenKind.LogicMultiply)

{

lexAn.RecognizeNextToken();

A();

E();

}

}

private void A()

{

switch(lexAn.Token.Type)

{

case TokenKind.Negative:

lexAn.RecognizeNextToken();

A();

break;

case TokenKind.LeftParenthesis:

lexAn.RecognizeNextToken();

D();

Match(TokenKind.RightParenthesis);

break;

default:

O();

break;

}

}

private void O()

{

switch (lexAn.Token.Type)

{

case TokenKind.FirstWord:

lexAn.RecognizeNextToken();

Match(TokenKind.Equal);

Match(TokenKind.SecondWord);

break;

case TokenKind.SecondWord:

lexAn.RecognizeNextToken();

Match(TokenKind.Equal);

if (lexAn.Token.Type != TokenKind.FirstWord

&& lexAn.Token.Type != TokenKind.SecondWord)

SyntaxError("Ожидалось слово типа 1 или типа 2");

lexAn.RecognizeNextToken();

break;

default:

SyntaxError("Ожидалось слово типа 1 или 2");

break;

}

}

// Процедура разбора для нетерминала T.

private void T()

{

if (lexAn.Token.Type == TokenKind.LogicSum)

{

lexAn.RecognizeNextToken();

K();

T();

}

}

// Провести синтаксический анализ текста.

public void ParseText()

{

lexAn.RecognizeNextToken(); // Распознаем первый токен в тексте.

D(); // Вызываем процедуру разбора для стартового нетерминала E.

if (lexAn.Token.Type != TokenKind.EndOfText) // Если текущий токен не является концом текста.

{

SyntaxError("После арифметического выражения идет еще какой-то текст"); // Обнаружена синтаксическая ошибка.

}

}

}

}

using System;

using System.Data;

namespace LAB2

{

// Тип токена.

enum TokenKind

{

FirstWord, // Первое слово.

SecondWord, // Второе слово.

EndOfText, // Конец текста.

Unknown, // Неизвестный.

LogicSum,

LogicMultiply,

LeftParenthesis,

RightParenthesis,

Negative,

Equal

};

// Класс "Токен".

class Token

{

private string value; // Значение токена (само слово).

private TokenKind type; // Тип токена.

// Позиция токена в исходном тексте.

private int lineIndex; // Индекс строки.

private int symStartIndex; // Индекс символа в строке lineIndex, с которого начинается токен.

// Значение токена (само слово).

public string Value

{

get { return value; }

set { this.value = value; }

}

// Тип токена.

public TokenKind Type

{

get { return type; }

set { this.type = value; }

}

// Индекс строки в исходном тексте, на которой расположен токен.

public int LineIndex

{

get { return lineIndex; }

set { this.lineIndex = value; }

}

// Индекс символа в строке LineIndex в исходном тексте, с которого начинается токен.

public int SymStartIndex

{

get { return symStartIndex; }

set { this.symStartIndex = value; }

}

// Сбросить значения полей токена.

public void Reset()

{

this.value = "";

this.type = TokenKind.Unknown;

this.lineIndex = -1;

this.symStartIndex = -1;

}

// Конструктор токена.

public Token()

{

Reset(); // Сбрасываем значения полей токена.

}

}

// Класс исключительных ситуаций лексического анализа.

class LexAnException : Exception

{

// Позиция возникновения исключительной ситуации в анализируемом тексте.

private int lineIndex; // Индекс строки.

private int symIndex; // Индекс символа.

// Индекс строки, где возникла исключительная ситуация - свойство только для чтения.

public int LineIndex

{

get { return lineIndex; }

}

// Индекс символа, на котором возникла исключительная ситуация - свойство только для чтения.

public int SymIndex

{

get { return symIndex; }

}

// Конструктор исключительной ситуации.

// message - описание исключительной ситуации.

// lineIndex и symIndex - позиция возникновения исключительной ситуации в анализируемом тексте.

public LexAnException(string message, int lineIndex, int symIndex) : base(message)

{

this.lineIndex = lineIndex;

this.symIndex = symIndex;

}

}

// Класс "Лексический анализатор".

// При обнаружении ошибки в исходном тексте он генерирует исключительную ситуацию LexAnException.

class LexicalAnalyzer

{

// Тип символа.

enum SymbolKind

{

Letter, // Буква от а до d.

Digit, // Цифра 0 или 1.

Space, // Пробел.

Reserved, // Зарезервированный.

Other, // Другой.

EndOfLine, // Конец строки.

EndOfText // Конец текста.

};

private const char commentSymbol1 = '{'; // Первый символ комментария.

private const char commentSymbol2 = '\*'; // Второй символ комментария.

private const char commentSymbol3 = '}'; // символ комментария.

private const char logicalMultiply = '\u2227';

private const char logicalSum = '\u2228';

private string inputLines; // Входной текст - массив строк.

private int curLineIndex; // Индекс текущей строки.

private int curSymIndex; // Индекс текущего символа в текущей строке.

private int pointer;

public int CurLineIndex { get=> curLineIndex; }

public int CurSymIndex { get => curSymIndex; }

private char curSym; // Текущий символ.

private SymbolKind curSymKind; // Тип текущего символа.

private Token token; // Токен, распознанный при последнем вызове метода RecognizeNextToken().

// Обработать лексическую ошибку.

// msg - описание ошибки.

private void LexicalError(string msg)

{

// Генерируем исключительную ситуацию, тем самым полностью прерывая процесс анализа текста.

throw new LexAnException(msg, curLineIndex, curSymIndex);

}

// Классифицировать текущий символ.

private void ClassifyCurrentSymbol()

{

if (((int)curSym >= (int)'a') && ((int)curSym <= (int)'d')) // Если текущий символ лежит в диапазоне строчных латинских букв.

{

curSymKind = SymbolKind.Letter; // Тип текущего символа - буква.

}

else if (((int)curSym >= (int)'0') && ((int)curSym <= (int)'1')) // Если текущий символ лежит в диапазоне цифр.

{

curSymKind = SymbolKind.Digit; // Тип текущего символа - цифра.

}

else

{

switch (curSym)

{

case ' ': // Если текущий символ - пробел.

curSymKind = SymbolKind.Space; // Тип текущего символа - пробел.

break;

// Если текущий символ - точка или первый символ комментария или второй символ комментария или символ подчеркивания.

case commentSymbol1:

case commentSymbol2:

case commentSymbol3:

case logicalMultiply:

case logicalSum:

case '(':

case ')':

case '\u00ac':

case '=':

curSymKind = SymbolKind.Reserved; // Тип текущего символа - зарезервированный.

break;

default:

curSymKind = SymbolKind.Other; // Тип текущего символа - другой.

break;

}

}

}

// Считать следующий символ.

private void ReadNextSymbol()

{

if (pointer >= inputLines.Length-1) // Если индекс текущей строки выходит за пределы текстового поля.

{

curSym = (char)0; // Обнуляем значение текущего символа.

curSymKind = SymbolKind.EndOfText; // Тип текущего символа - конец текста.

return;

}

curSymIndex++; // Увеличиваем индекс текущего символа.

pointer++; // Увеличиваем указатель

curSym = inputLines[pointer]; // Считываем текущий символ.

if (curSym == '\n')

{

curSymIndex = -1;

curSym = (char)0;

curSymKind = SymbolKind.EndOfLine;

curLineIndex++;

return;

}

ClassifyCurrentSymbol(); // Классифицируем текущий символ.

}

enum State

{

S,

A,

B,

C,

D,

E,

F,

J

}

// Распознать идентификатор.

private void RecognizeFirstWord()

{

var curentState = State.S;

var flag = false;

while (true)

{

switch (curentState)

{

case State.S:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym;

curentState = State.A;

}

else if (curSymKind == SymbolKind.Digit && curSym == '0')

{

token.Value += curSym;

curentState = State.C;

}

else LexicalError("Ожидалась 0 или 1");

break;

case State.A:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym;

curentState = State.B;

}

else LexicalError("Ожидалась 1");

break;

case State.B:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym;

curentState = State.S;

}

else LexicalError("Ожидалась 1");

break;

case State.C:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym; // Наращиваем значение текущего токена.

curentState = State.D;

}

else LexicalError("Ожидалась 1");

break;

case State.D:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym; // Наращиваем значение текущего токена.

curentState = State.E; // Переходим в указанное состояние.

}

else LexicalError("Ожидалась 1");

break;

case State.E:

if (curSymKind == SymbolKind.Digit && curSym == '1')

{

token.Value += curSym; // Наращиваем значение текущего токена.

curentState = State.F; // Переходим в указанное состояние.

}

else flag = true;

break;

case State.F:

if (curSymKind == SymbolKind.Digit && curSym == '0')

{

token.Value += curSym; // Наращиваем значение текущего токена.

curentState = State.J; // Переходим в указанное состояние.

}

else LexicalError("Ожидался 0");

break;

case State.J:

if (curSymKind == SymbolKind.Digit && curSym == '0')

{

token.Value += curSym; // Наращиваем значение текущего токена.

curentState = State.E;

}

else LexicalError("Ожидался 0");

break;

}

if (flag) break;

ReadNextSymbol();

}

token.Type = TokenKind.FirstWord;

return;

}

enum State2

{

S,

A,

B,

Empty

}

// Распознать число (целое или вещественное).

private void RecognizeSecondWord()

{

var table = new DataTable();

table.Columns.AddRange(new DataColumn[] {

new DataColumn("CurrentState", typeof(State2)),

new DataColumn("a",typeof(State2)),

new DataColumn("b", typeof(State2)),

new DataColumn("c", typeof(State2)),

new DataColumn("d", typeof(State2)),

new DataColumn("Final?", typeof(bool))

}

);

table.PrimaryKey = new DataColumn[] { table.Columns[0] };

table.Rows.Add(State2.S, State2.A, State2.B, State.B, State.B, false);

table.Rows.Add(State2.A, State2.A, State2.Empty, State.B, State.B, true);

table.Rows.Add(State2.B, State2.A, State2.B, State.B, State.B, true);

var currentState = State2.S;

while (true)

{

var row = table.Rows.Find(currentState);

if (!table.Columns.Contains(curSym.ToString()) && (bool)row["Final?"] == true)

{

token.Type = TokenKind.SecondWord;

return;

}

else if (!table.Columns.Contains(curSym.ToString()) && (bool)row["Final?"] == false) throw new LexAnException("Встречен неизвестный символ", curLineIndex, curSymIndex);

currentState = (State2)row[curSym.ToString()];

token.Value += curSym;

if (currentState == State2.Empty) throw new LexAnException("Не может быть подстроки ab", curLineIndex, curSymIndex);

ReadNextSymbol();

}

}

private void RecognizeReservedSymbol()

{

switch (curSym)

{

case logicalSum:

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.LogicSum; // Тип распознанного токена - "+".

ReadNextSymbol(); // Читаем следующий символ в тексте.

break;

case logicalMultiply:

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.LogicMultiply; // Тип распознанного токена - "-".

ReadNextSymbol(); // Читаем следующий символ в тексте.

break;

case '(':

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.LeftParenthesis; // Тип распознанного токена - "(".

ReadNextSymbol(); // Читаем следующий символ в тексте.

break;

case ')':

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.RightParenthesis; // Тип распознанного токена - ")".

ReadNextSymbol(); // Читаем следующий символ в тексте.

break;

case '\u00Ac':

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.Negative; // Тип распознанного токена - ")".

ReadNextSymbol();

break;

case '=':

token.Value += curSym; // Наращиваем значение текущего токена.

token.Type = TokenKind.Equal; // Тип распознанного токена - ")".

ReadNextSymbol();

break;

default:

LexicalError("Неизвестный зарезервированный символ '" + curSym + "'"); // Обнаружена ошибка в тексте.

break;

}

}

// Пропустить комментарий.

private void SkipComment()

{

goto S; // Запускаем конечный автомат.

// Конечный автомат для комментария.

// S - начальное состояние.

//----------------------------------------------------//

S:

if (curSym == commentSymbol1)

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto A; // Переходим в указанное состояние.

}

else

LexicalError("Ожидалось " + commentSymbol1); // Обнаружена ошибка в тексте.

A:

if (curSym == commentSymbol2)

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto B; // Переходим в указанное состояние.

}

else

LexicalError("Ожидалось " + commentSymbol2); // Обнаружена ошибка в тексте.

B:

if (curSym == commentSymbol2)

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto C; // Переходим в указанное состояние.

}

else if (curSymKind == SymbolKind.EndOfText)

LexicalError("Незаконченный комментарий"); // Обнаружена ошибка в тексте.

else

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto B; // Переходим в указанное состояние.

}

C:

if (curSym == commentSymbol3)

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto Fin; // Переходим в указанное состояние.

}

else if (curSym == commentSymbol2)

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto C; // Переходим в указанное состояние.

}

else if (curSymKind == SymbolKind.EndOfText)

LexicalError("Незаконченный комментарий"); // Обнаружена ошибка в тексте.

else

{

ReadNextSymbol(); // Читаем следующий символ в тексте.

goto B; // Переходим в указанное состояние.

}

Fin:

goto Quit; // Выходим из конечного автомата.

//----------------------------------------------------//

// Конец конечного автомата для комментария.

Quit:

return;

}

// Конструктор лексического анализатора.

// В качестве параметра передается исходный текст.

public LexicalAnalyzer(string inputLines)

{

this.inputLines = inputLines;

// Обнуляем поля.

curLineIndex = 0;

curSymIndex = -1;

pointer = -1;

curSym = (char)0;

token = null;

// Считываем первый символ входного текста.

ReadNextSymbol();

}

// Токен, распознанный при последнем вызове метода RecognizeNextToken() - свойство только для чтения.

public Token Token

{

get { return token; }

}

// Распознать следующий токен в тексте.

public void RecognizeNextToken()

{

// На данный момент уже прочитан символ, следующий за токеном, распознанным в прошлом вызове этого метода.

// Если же это первый вызов, то на данный момент уже прочитан первый символ текста (в конструкторе).

// Цикл пропуска пробелов, переходов на новую строку, комментариев.

while ( (curSymKind == SymbolKind.Space) ||

(curSymKind == SymbolKind.EndOfLine) ||

(curSym == commentSymbol1) )

{

if (curSym == commentSymbol1) // Если текущий символ - первый символ комментария.

SkipComment(); // Пропускаем комментарий.

else

ReadNextSymbol(); // Пропускаем пробел или переход на новую строку.

}

// Создаем новый экземпляр токена.

token = new Token();

// Запоминаем позицию начала токена в исходном тексте.

token.LineIndex = curLineIndex;

token.SymStartIndex = curSymIndex;

switch (curSymKind) // Анализируем текущий символ.

{

case SymbolKind.Digit: // Если текущий символ - буква.

RecognizeFirstWord(); // Вызываем процедуру распознавания идентификатора.

break;

case SymbolKind.Letter: // Если текущий символ - цифра.

RecognizeSecondWord(); // Вызываем процедуру распознавания числа (целого или вещественного).

break;

case SymbolKind.EndOfText: // Если текущий символ - конец текста.

token.Type = TokenKind.EndOfText; // Тип распознанного токена - конец текста.

break;

case SymbolKind.Reserved:

RecognizeReservedSymbol();

break;

default: // Если текущий символ - какой-то другой.

LexicalError("Недопустимый символ"); // Обнаружена ошибка в тексте.

break;

}

}

}

}

# Пример работы

На рисунке 4.1 показан пример работы программы.

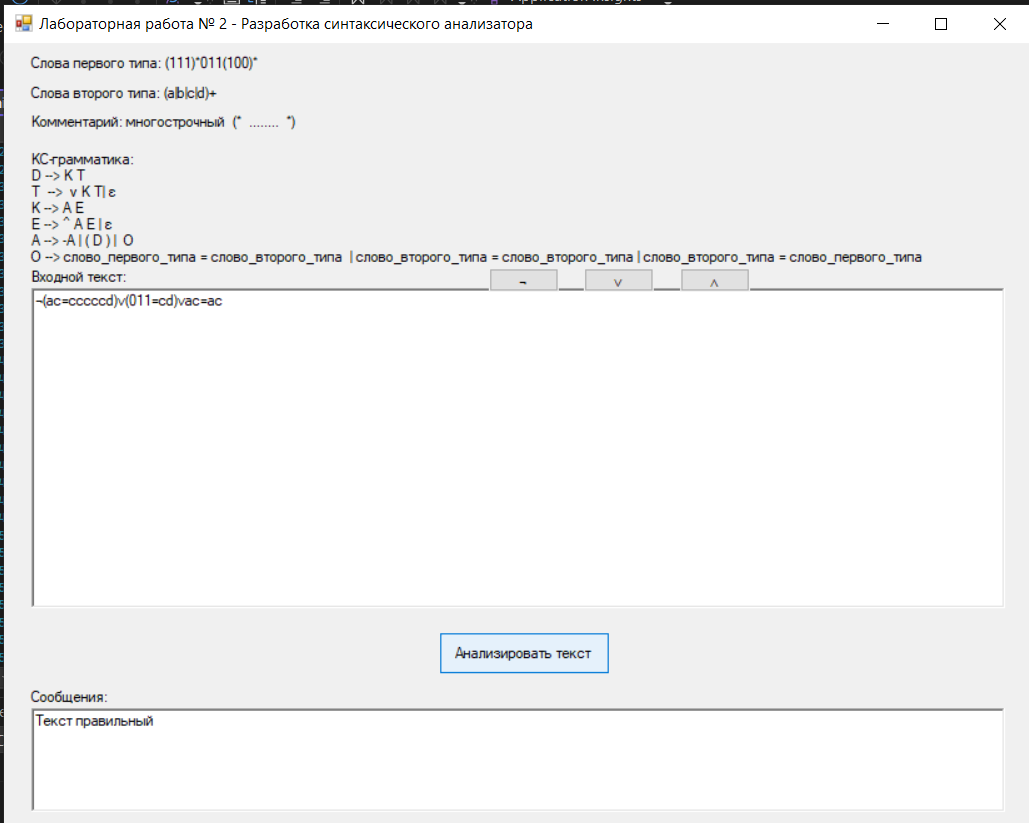


Рисунок 4.1 – Пример работы программы

На рисунке 4.2 показан пример работы с некорректными входными данными.

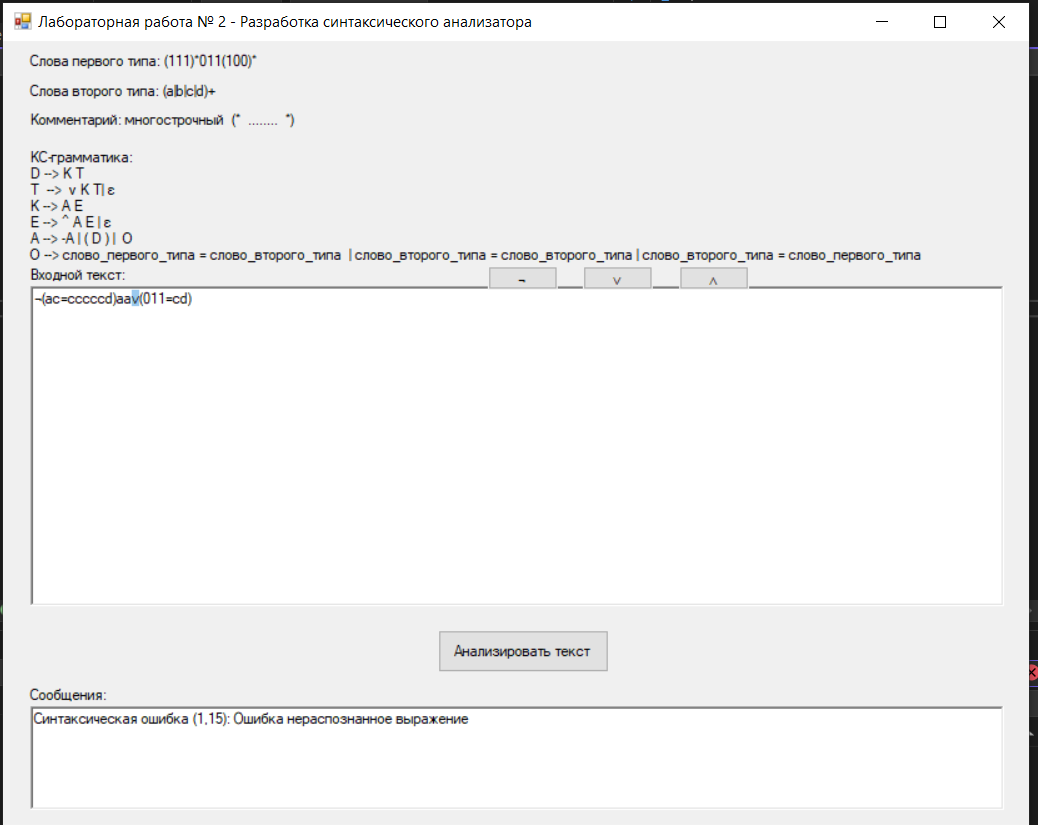


Рисунок 4.2 – Пример работы с некорректными данными