Федеральное агентство связи

Ордена Трудового Красного знамени

Федеральное государственное бюджетное образовательное учреждение

Высшего образования

«Московский технический университет связи и информатики»

Кафедра Математической кибернетики и информационных технологий

Отчет по лабораторной работе №3

по дисциплине «Системное программное обеспечение»

по теме «Построение простейшего дерева вывода»

Выполнил

Студент группы БФИ1701

Неретин И.Г.

Вариант 10

Проверила:

Алексанян Д.А.

Москва 2020

1. **Задание**

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике с построением дерева разбора. Текст на входном языке задается в виде символьного (текстового) файла. Синтаксис входного языка и перечень допустимых лексем указаны в задании. Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

При наличии во входном файле текста, соответствующего заданному языку, программа должна строить и отображать дерево синтаксического разбора. Если же текст во входном файле содержит ошибки (лексические или синтаксические), программа должна выдавать сообщения о наличии ошибок во входном тексте и корректно завершать свое выполнение.

1. **Цель работы**

Цель работы – изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализатора (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

1. **Исходная грамматика**

G({**a, or, xor, and, not, (, ), :=, ;**}, {S, F, T, E}, S, P)

P:

S -> a:= F;

F -> F or T | F xor T | T

T -> T and E | E

E -> (F) | not (F) | a

a – лексемы (идентификаторы или шестнадцатеричные числа)

1. **Множества крайних правых и левых символов**

В таблицах ниже показан пошаговый процесс поиска крайних левых и крайних правых символов грамматики.

Таблица – Поиск крайних правых и левых символов. Шаг 1

|  |  |  |
| --- | --- | --- |
| **Правило U** | **L(U) – левый символ** | **R(U) – правый символ** |
| S | a | ; |
| F | F, T | T |
| T | T, E | E |
| E | (, not, a | ), a |

Таблица – Поиск крайних правых и левых символов. Шаг 2

|  |  |  |
| --- | --- | --- |
| **Правило U** | **L(U) – левый символ** | **R(U) – правый символ** |
| S | a | ; |
| F | F, T, E | T, E |
| T | T, E, (, not, a | E, ), a |
| E | (, not, a | ), a |

Таблица – Поиск крайних правых и левых символов. Шаг 3

|  |  |  |
| --- | --- | --- |
| **Правило U** | **L(U) – левый символ** | **R(U) – правый символ** |
| S | a | ; |
| F | F, T, E, (, not, a | T, E, ), a |
| T | T, E, (, not, a | E, ), a |
| E | (, not, a | ), a |

Помимо фактических крайних правых и крайних левых символов правила, в их множества L(U) и R(U) были добавлены крайние правые и левые символы других правил, если правило содержало в себе нетерминальный символ, соответствующий другому правилу.

Например, правило F содержало в себе нетерминал T в множестве L(F), поэтому в него было добавлены все символы из множества L(T), которые были у правила T.

В Таблице 3 указан конечный вариант построения таблицы, поскольку больше никаких возможностей дополнения множеств не обнаружено.

1. **Множества крайних правых и левых терминальных символов**

Составление таблицы терминальных символов схоже с составлением таблицы символов в целом, за тем исключением, что нетерминальные символы игнорируются. Первоначальная версия таблицы указана ниже.

Таблица – Поиск крайних правых и левых терминальных символов. Шаг 1

|  |  |  |
| --- | --- | --- |
| **Правило U** | **L(U) – левый символ** | **R(U) – правый символ** |
| S | a | ; |
| F | or, xor | or, xor |
| T | and | and |
| E | (, not, a | ), a |

Теперь таблицу нетерминалов необходимо дополнить следующим образом: если в множестве крайних левых и правых символов (L(U), R(U)) правила U находится нетерминал X, то все терминалы правила X из таблицы терминальных символов, находящиеся в L(X) и R(X) этой таблицы, добавляются в таблицу терминальных символов в L(U) и R(U) для правила U соответственно.

Иначе говоря, в таблице терминальных символов все символы, которые есть у X, но нет у U, добавляются в U в соответствующее множество (крайнее правое или крайнее левое), при условии, что X есть в этом множестве у U в таблице символов, включающих нетерминалы.

Поступив таким образом, мы получаем следующую таблицу:

Таблица – Поиск крайних правых и левых терминальных символов. Шаг 2

|  |  |  |
| --- | --- | --- |
| **Правило U** | **L(U) – левый символ** | **R(U) – правый символ** |
| S | a | ; |
| F | or, xor, and, (, not, a | or, xor, and, ), a |
| T | and, (, not, a | and, ), a |
| E | (, not, a | ), a |

Теперь, на основе данных таблиц, а также правил грамматики, можно построить матрицу предшествования.

1. **Матрица предшествования**

Упрощенно алгоритм составления матрицы предшествования можно описать следующими действиями:

1. Заголовки строк и столбцов матрицы заполняются терминальными символами. Добавляется строка с символом начала, а также столбец с символом конца;
2. Берется символ строки. Если этот символ в правилах грамматики стоит перед каким-то терминалом, то ставится символ «=-» (составляет основу) в столбце с этим терминалом на текущей строке;
3. Если этот символ стоит перед каким-то нетерминалом X, то ставится символ «<-» (предшествует) в столбце со всеми терминалами, которые включены в множество L(X);
4. Шаги 2-3 выполнить для всех строк, кроме строки с символом начала;
5. Берется символ столбца. Если этот символ стоит после какого-то нетерминала X, то в текущем столбце и в каждой строке с терминалами, которые включены в R(X), ставится знак «->» (следует);
6. Шаг 5 повторить для каждого столбца;
7. Для целевого символа (стартового нетерминала) S берем множество L(S) из таблицы терминальных символов. На пересечении столбца с каждым терминальным символом из L(S) и строки с символом начала ставим знак «<-» (предшествует);
8. На пересечении каждой строки с символами из R(S) и столбцом со знаком конца ставим знак «->» (следует);
9. Если в матрице имеются коллизии (в одну ячейку записано 2 или 3 символа), то грамматика не подходит для построения матрицы операторного предшествования;
10. Если коллизий нет, то построение матрицы предшествования завершено.

В соответствии с заданным алгоритмом была построена матрица операторного предшествования (Таблица 6):

Таблица – Матрица операторного предшествования

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Символ | a | or | xor | and | not | ( | ) | := | ; | Кон |
| a |  | -> | -> | -> |  |  | -> | =- | -> |  |
| or | <- | -> | -> | <- | <- | <- | -> |  | -> |  |
| xor | <- | -> | -> | <- | <- | <- | -> |  | -> |  |
| and | <- | -> | -> | -> | <- | <- | -> |  | -> |  |
| not |  |  |  |  |  | =- |  |  |  |  |
| ( | <- | <- | <- | <- | <- | <- | =- |  |  |  |
| ) |  | -> | -> | -> |  |  | -> |  | -> |  |
| := | <- | <- | <- | <- | <- | <- |  |  | =- |  |
| ; |  |  |  |  |  |  |  |  |  | -> |
| Нач | <- |  |  |  |  |  |  |  |  |  |

Эту таблицу можно использовать для синтаксического анализа строки при помощи алгоритма «сдвиг-свертка». Но сперва, для программной реализации, необходимо составить остовную грамматику.

1. **Остовная грамматика**

Остовная грамматика представляет собой грамматику, в которой все нетерминалы заменены единственным нетерминалом (как правило, целевым).

При выполнении алгоритма «сдвиг-свертка» будет получена последовательность правил, примененных для построения строки (если она соответствует синтаксису языка). С помощью этой последовательности и исходных правил можно будет построить дерево синтаксического разбора, и не столь важно, какой именно символ исходной грамматики был использован. Поэтому, для упрощения вычислений в алгоритме, все нетерминальные символы были заменены нетерминальным символом «S», а повторяющиеся правила удалены. В итоге получилась следующая остовная грамматика:

G’({**a, or, xor, and, not, (, ), :=, ;**}, {S}, S, P)

P:

S -> a:= S; 1

S -> S or S | S xor S | S 2 | 3 | 4

S -> S and S 5

S -> (S) | not (S) | a 6 | 7 | 8

Сбоку подписаны номера правил.

1. **Пример разбора**

Например, имеется строка a := a or not(a);

{входная цепочка}{стэк}{последовательность правил} операция

{a := a or not(a); END}{START}{}

{ := a or not(a); END }{a START}{} сдвиг

{ a or not(a); END }{:= a START}{} сдвиг

{or not(a); END }{a := a START}{} сдвиг

{or not(a); END }{S := a START}{8} свертка

{not(a); END }{or S := a START}{8} сдвиг

{(a); END }{not or S := a START}{8} сдвиг

{a); END }{ ( not or S := a START}{8} сдвиг

{); END }{a ( not or S := a START}{8} сдвиг

{); END }{S ( not or S := a START}{8 8} свертка

{; END }{) S ( not or S := a START}{8 8} сдвиг

{; END }{S or S := a START}{8 8 7} свертка

{; END }{S := a START}{8 8 7 2} свертка

{; END }{S := a START}{8 8 7 2} сдвиг

{ END }{; S := a START}{8 8 7 2} сдвиг

{ END }{S START}{8 8 7 2 1} свертка

Разбор завершен.

Итоговая последовательность правил: 1 -> 2 -> 7 -> 8 -> 8

S

a:=S;

a:=S or S;

a:=S or not(S);

a:=a or not(S);

a:=a or not(a);

1. **Скриншоты программы**

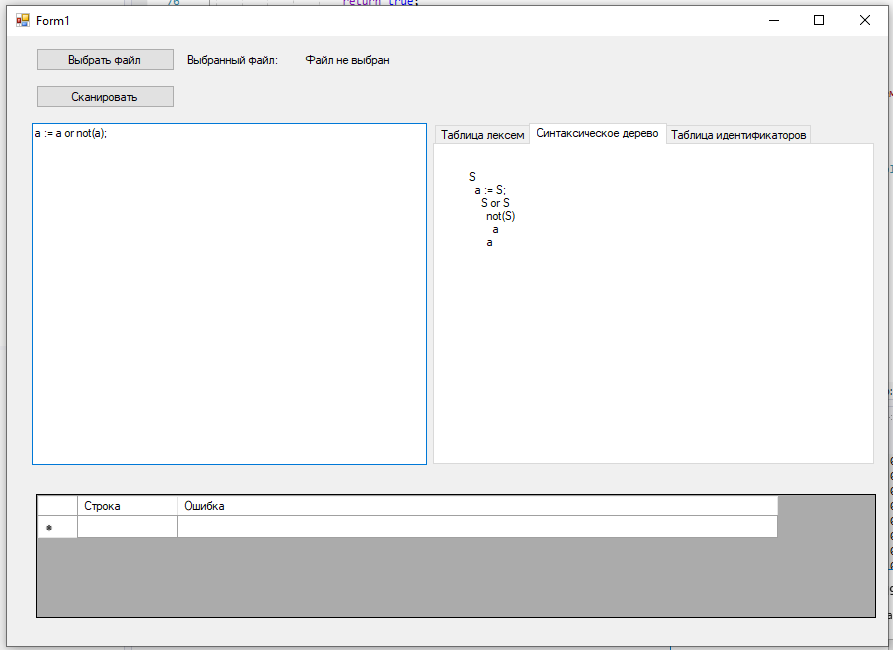


Рисунок – Результат разбора одной строки

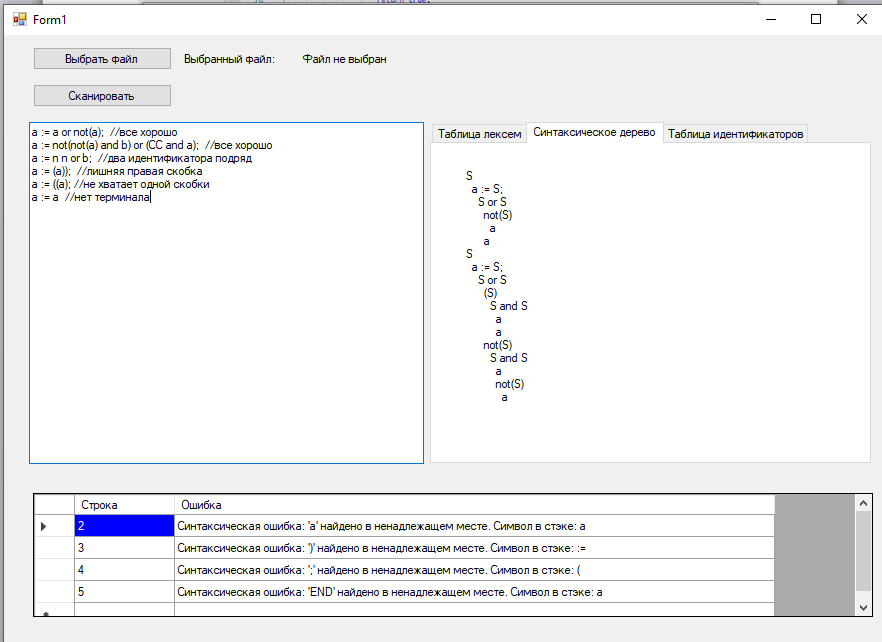


Рисунок – Результат разбора нескольких строк

1. **Текст программы**

Программа написана на языке C# в среде Visual Studio Community 2019 под операционную систему Windows.

Листинг – SyntaxAnalyser.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using SPOLab2;  namespace SPOLab3  {  class SyntaxAnalyser  {  private Stack<string> stack;  Dictionary<string, Dictionary<string, int>> precedenceTable;  RuleTable ruleTable;  public List<ErrorInfo> errorInfos;  public List<string> ruleSequense;  public List<string> stackHistory;  int currentRow;  public List<LexemeTree> lexemeTrees;  public SyntaxAnalyser()  {  PrecedenceTable table = new PrecedenceTable();  stack = new Stack<string>();  ruleTable = new RuleTable();  precedenceTable = table.getTable();  errorInfos = new List<ErrorInfo>();  ruleSequense = new List<string>();  stackHistory = new List<string>();  lexemeTrees = new List<LexemeTree>();  }  public void analyse(List<Lexeme> listOfLexemes)  {  List<string> currentList = new List<string>();  int i = 0;  currentRow = 0;  errorInfos.Clear();  bool result;  while (i < listOfLexemes.Count)  {  currentRow = listOfLexemes[i].stringNumber;  while (i < listOfLexemes.Count)  {  currentList.Add(listOfLexemes[i].value);  if (Language.isItTerminal(listOfLexemes[i].value)) break;  i++;  }  if (analyseThisString(currentList))  {  //reportError("very", "good");  TreeConstructor constructor = new TreeConstructor();  lexemeTrees.Add(constructor.CreateTree(ruleSequense));  }  currentList.Clear();  i++;  }  }  private bool analyseThisString(List<string> entrySequence)  {  string anotherLexeme;  string terminalFoundInStack;  int stateGotten;  Dictionary<string, int> PrecedenceTableCol;  stack.Push(PrecedenceTable.START\_SYMBOL);  entrySequence.Add(PrecedenceTable.END\_SYMBOL);  for (int i = 0; i < entrySequence.Count;)  {  anotherLexeme = entrySequence[i];    terminalFoundInStack = findNonTerminalFromStack();  if ((anotherLexeme == PrecedenceTable.END\_SYMBOL) && (terminalFoundInStack == PrecedenceTable.START\_SYMBOL))  {  return true;  }  if(precedenceTable.TryGetValue(terminalFoundInStack, out PrecedenceTableCol) != false)  {  if (PrecedenceTableCol.TryGetValue(anotherLexeme, out stateGotten) == false)  {  reportError("Синтаксическая ошибка: '" + anotherLexeme + "' найдено в ненадлежащем месте. Символ в стэке: ", terminalFoundInStack);  return false;  }  else  {  if ((stateGotten == PrecedenceTable.STATE\_EQUAL) || (stateGotten == PrecedenceTable.STATE\_PREVIOUS))  {  //сдвиг  stack.Push(anotherLexeme);  i++;  }  if (stateGotten == PrecedenceTable.STATE\_NEXT)  {  //свертка  //перемещение каретки не производится  List<string> chainGotten = getRelatedLexemesFromStack();  if (!convoluteLexemesIntoRule(chainGotten))  {  string chain = "";  for (int k = 0; k < chainGotten.Count; k++)  {  chain += chainGotten[k] + " ";  }  reportError("Синтаксическая ошибка при формировании конструкции: '" + chain + "'", "");  }  }  }  }  }  return false;  }  private string findNonTerminalFromStack()  {  Stack<string> currentStack = copyStack(stack);  int i;  while (currentStack.Count > 0)  {  if (itIsTerminal(currentStack.Peek()))  {  return currentStack.Peek();  }  currentStack.Pop();  }  return null;  }  private bool itIsTerminal(string a)  {  if (Language.isItLegalIdentifier(a) || Language.isItLegalNumber(a)  || Language.isItKeyword(a) || Language.isItOperationsWord(a)  || Language.isItTerminal(a) || (a == PrecedenceTable.START\_SYMBOL))  {  return true;  }  return false;  }  private void reportError(string wordFound, string terminalFoundInStack)  {  errorInfos.Add(new ErrorInfo(currentRow, wordFound + terminalFoundInStack));  }  private List<string> getRelatedLexemesFromStack()  {  List<string> chain = new List<string>();  Stack<string> buferStack = new Stack<string>();  List<string> LexemesInStack = new List<string>();  List<int> removeThis = new List<int>();  buferStack = copyStack(stack);  int i;  while (buferStack.Count > 0)  {  LexemesInStack.Add(buferStack.Pop());  }  string lastTerminal = "";  string lastNonTerminal = "";  string currentLexeme = "";  bool EqualChainNotFound = true;  for (i = 0; i < LexemesInStack.Count; i++)  {  currentLexeme = LexemesInStack[i];  if (itIsTerminal(currentLexeme))  {  if (lastTerminal == "")  {  chain.Add(currentLexeme);  removeThis.Add(i);  lastTerminal = currentLexeme;  }  else  {  if (!itHasStateInPrecedenceTable(currentLexeme, lastTerminal, PrecedenceTable.STATE\_EQUAL))  {  break;  }  else  {  chain.Add(currentLexeme);  removeThis.Add(i);  EqualChainNotFound = false;  }  lastTerminal = currentLexeme;  }  }  else  {  chain.Add(currentLexeme);  removeThis.Add(i);  lastNonTerminal = currentLexeme;  }  }  //Составная цепочка не найдена  if (EqualChainNotFound)  {  lastTerminal = "";  chain.Clear();  removeThis.Clear();  for (i = 0; i < LexemesInStack.Count; i++)  {  currentLexeme = LexemesInStack[i];  if (itIsTerminal(currentLexeme))  {  if (lastTerminal == "")  {  lastTerminal = currentLexeme;  chain.Add(currentLexeme);  removeThis.Add(i);  }  else  {  break;  }  }  else  {  chain.Add(currentLexeme);  removeThis.Add(i);  }  }  }  LexemesInStack = removeIndexesFromList(removeThis, LexemesInStack);  makeThisListAsCurrentStack(LexemesInStack);  return revertList(chain);  }  private Stack<string> copyStack(Stack<string> originalStack)  {  Stack<string> buferStack = new Stack<string>();  Stack<string> returnLater = new Stack<string>();  Stack<string> currentStack = new Stack<string>();  int i;  while (originalStack.Count > 0)  {  buferStack.Push(originalStack.Pop());  returnLater.Push(buferStack.Peek());  }  while (buferStack.Count > 0)  {  currentStack.Push(buferStack.Pop());  }  while (returnLater.Count > 0)  {  originalStack.Push(returnLater.Pop());  }  return currentStack;  }  private List<string> revertList(List<string> list)  {  string bufer;  for (int i = 0; i < list.Count / 2; i++)  {  bufer = list[i];  list[i] = list[list.Count - i - 1];  list[list.Count - i - 1] = bufer;  }  return list;  }  private List<string> removeIndexesFromList(List<int> removeIndexes, List<string> fromList)  {  for (int i = removeIndexes.Count - 1; i >= 0; i--)  {  if (removeIndexes.Contains(i))  {  removeIndexes.Remove(i);  fromList.RemoveAt(i);  }  }  return fromList;  }  private void makeThisListAsCurrentStack(List<string> list)  {  stack.Clear();  for (int i = list.Count - 1; i >= 0; i--)  {  stack.Push(list[i]);  }  }  private bool itHasStateInPrecedenceTable(string row, string col, int expected)  {  int valueGotten;  Dictionary<string, int> rowGotten;  if (precedenceTable.TryGetValue(row, out rowGotten))  {  if (rowGotten.TryGetValue(col, out valueGotten))  {  if (valueGotten == expected)  {  return true;  }  }  }  return false;  }  private bool convoluteLexemesIntoRule(List<string> listOfLexemes)  {  string numberOfRuleGotten;  string symbolGotten = ruleTable.convolution\_getRuleSymbol(listOfLexemes, out numberOfRuleGotten);  if (symbolGotten == null)  {  return false;  }  else  {  stack.Push(symbolGotten);  ruleSequense.Add(numberOfRuleGotten);  Stack<string> buferStack = copyStack(stack);  string containedInStack = "";  while (buferStack.Count > 0)  {  containedInStack += buferStack.Pop() + " ";  }  stackHistory.Add(containedInStack);  return true;  }    }  }  } |

Листинг 1 – RuleTable.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace SPOLab3  {  class RuleTable  {  public static string S\_NONTERMINAL\_SYMBOL = "$S";  public static string F\_NONTERMINAL\_SYMBOL = "$F";  public static string T\_NONTERMINAL\_SYMBOL = "$T";  public static string E\_NONTERMINAL\_SYMBOL = "$E";  public static string SYMBOL\_SEPARATOR = " ";  Dictionary<string, string> ruleTable;  List<string> NonTerminalSymbolsAllowed;  public RuleTable()  {  ruleTable = new Dictionary<string, string>();  //правило | символ-родитель  //S  ruleTable.Add("a" + SYMBOL\_SEPARATOR + ":=" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + ";", S\_NONTERMINAL\_SYMBOL + "1");  ruleTable.Add(S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + "or" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL, S\_NONTERMINAL\_SYMBOL + "2");  ruleTable.Add(S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + "xor" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL, S\_NONTERMINAL\_SYMBOL + "3");  ruleTable.Add(S\_NONTERMINAL\_SYMBOL, S\_NONTERMINAL\_SYMBOL + "4");  ruleTable.Add(S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + "and" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL, S\_NONTERMINAL\_SYMBOL + "5");  ruleTable.Add("(" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + ")", S\_NONTERMINAL\_SYMBOL + "6");  ruleTable.Add("not" + SYMBOL\_SEPARATOR + "(" + SYMBOL\_SEPARATOR + S\_NONTERMINAL\_SYMBOL + SYMBOL\_SEPARATOR + ")", S\_NONTERMINAL\_SYMBOL + "7");  ruleTable.Add("a", S\_NONTERMINAL\_SYMBOL + "8");  NonTerminalSymbolsAllowed = new List<string>();  NonTerminalSymbolsAllowed.Add(S\_NONTERMINAL\_SYMBOL);  NonTerminalSymbolsAllowed.Add(SYMBOL\_SEPARATOR);  }  public string convolution\_getRuleSymbol(string ruleResultGotten, out string numberOfRule)  {  string answer;  if (ruleTable.TryGetValue(ruleResultGotten, out answer))  {  answer = extractNumber(answer, out numberOfRule);  return answer;  }  else  {  numberOfRule = "error";  return null;  }  }  public string convolution\_getRuleSymbol(List<string> ruleResultGotten, out string numberOfRule)  {  return convolution\_getRuleSymbol(prepareList(ruleResultGotten), out numberOfRule);  }    private string extractNumber(string rule, out string number)  {  number = "";  for (int i = rule.Length - 1; i >= 0; i--)  {  if (itIsNumber(rule[i]))  {  number += rule[i];  rule = rule.Remove(i, 1);  }  }  number = revertString(number);  return rule;  }  private bool itIsNumber(char number)  {  for (int i = 0; i < 10; i++)  {  if (Convert.ToString(number) == Convert.ToString(i))  {  return true;  }  }  return false;  }  private string revertString(string str)  {  char bufer;  char[] charArray = str.ToCharArray();  for (int i = 0; i < charArray.Length / 2; i++)  {  bufer = charArray[i];  charArray[i] = charArray[charArray.Length - 1 - i];  charArray[charArray.Length - 1 - i] = bufer;  }  return new string(charArray);  }  private string prepareList(List<string> listOfLexemes)  {  string answer = "";  if (listOfLexemes.Count < 1) return "";  answer += listOfLexemes[0];  for (int i = 1; i < listOfLexemes.Count; i++)  {  answer += SYMBOL\_SEPARATOR + listOfLexemes[i];  }  return answer;  }  }  } |

Листинг 1 – PrecedenceTable.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace SPOLab3  {  class PrecedenceTable  {  public static int STATE\_PREVIOUS = 1;  public static int STATE\_NEXT = 2;  public static int STATE\_EQUAL = 3;  public static string START\_SYMBOL = "START";  public static string END\_SYMBOL = "END";  private Dictionary<string, Dictionary<string, int>> table;  public PrecedenceTable()  {  table = new Dictionary<string, Dictionary<string, int>>();  //a  Dictionary<string, int> str1 = new Dictionary<string, int>();  str1.Add("or", STATE\_NEXT);  str1.Add("xor", STATE\_NEXT);  str1.Add("and", STATE\_NEXT);  str1.Add(")", STATE\_NEXT);  str1.Add(":=", STATE\_EQUAL);  str1.Add(";", STATE\_NEXT);  //or  Dictionary<string, int> str2 = new Dictionary<string, int>();  str2.Add("a", STATE\_PREVIOUS);  str2.Add("or", STATE\_NEXT);  str2.Add("xor", STATE\_NEXT);  str2.Add("and", STATE\_PREVIOUS);  str2.Add("not", STATE\_PREVIOUS);  str2.Add("(", STATE\_PREVIOUS);  str2.Add(")", STATE\_NEXT);  str2.Add(";", STATE\_NEXT);  //xor  Dictionary<string, int> str3 = new Dictionary<string, int>();  str3.Add("a", STATE\_PREVIOUS);  str3.Add("or", STATE\_NEXT);  str3.Add("xor", STATE\_NEXT);  str3.Add("and", STATE\_PREVIOUS);  str3.Add("not", STATE\_PREVIOUS);  str3.Add("(", STATE\_PREVIOUS);  str3.Add(")", STATE\_NEXT);  str3.Add(";", STATE\_NEXT);  //and  Dictionary<string, int> str4 = new Dictionary<string, int>();  str4.Add("a", STATE\_PREVIOUS);  str4.Add("or", STATE\_NEXT);  str4.Add("xor", STATE\_NEXT);  str4.Add("and", STATE\_NEXT);  str4.Add("not", STATE\_PREVIOUS);  str4.Add("(", STATE\_PREVIOUS);  str4.Add(")", STATE\_NEXT);  str4.Add(";", STATE\_NEXT);  //not  Dictionary<string, int> str5 = new Dictionary<string, int>();  str5.Add("(", STATE\_EQUAL);  //(  Dictionary<string, int> str6 = new Dictionary<string, int>();  str6.Add("a", STATE\_PREVIOUS);  str6.Add("or", STATE\_PREVIOUS);  str6.Add("xor", STATE\_PREVIOUS);  str6.Add("and", STATE\_PREVIOUS);  str6.Add("not", STATE\_PREVIOUS);  str6.Add("(", STATE\_PREVIOUS);  str6.Add(")", STATE\_EQUAL);  //)  Dictionary<string, int> str7 = new Dictionary<string, int>();  str7.Add("or", STATE\_NEXT);  str7.Add("xor", STATE\_NEXT);  str7.Add("and", STATE\_NEXT);  str7.Add(")", STATE\_NEXT);  str7.Add(";", STATE\_NEXT);  //:=  Dictionary<string, int> str8 = new Dictionary<string, int>();  str8.Add("a", STATE\_PREVIOUS);  str8.Add("or", STATE\_PREVIOUS);  str8.Add("xor", STATE\_PREVIOUS);  str8.Add("and", STATE\_PREVIOUS);  str8.Add("not", STATE\_PREVIOUS);  str8.Add("(", STATE\_PREVIOUS);  str8.Add(";", STATE\_EQUAL);  //;  Dictionary<string, int> str9 = new Dictionary<string, int>();  str9.Add(END\_SYMBOL, STATE\_NEXT);  //START  Dictionary<string, int> str10 = new Dictionary<string, int>();  str10.Add("a", STATE\_PREVIOUS);  table.Add("a", str1);  table.Add("or", str2);  table.Add("xor", str3);  table.Add("and", str4);  table.Add("not", str5);  table.Add("(", str6);  table.Add(")", str7);  table.Add(":=", str8);  table.Add(";", str9);  table.Add(START\_SYMBOL, str10);  }  public Dictionary<string, Dictionary<string, int>> getTable()  {  return table;  }  }  } |

Листинг 1 – TreeConstructor.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace SPOLab3  {  class TreeConstructor  {  string[] numberToRuleResult;  Stack<int> rules;  public TreeConstructor()  {  numberToRuleResult = new string[9];  numberToRuleResult[0] = "";  numberToRuleResult[1] = "a := S;";  numberToRuleResult[2] = "S or S";  numberToRuleResult[3] = "S xor S";  numberToRuleResult[4] = "S";  numberToRuleResult[5] = "S and S";  numberToRuleResult[6] = "(S)";  numberToRuleResult[7] = "not(S)";  numberToRuleResult[8] = "a";  rules = new Stack<int>();  }  public LexemeTree CreateTree(List<string> ruleSequence)  {  for (int i = 0; i < ruleSequence.Count; i++)  {  rules.Push(Convert.ToInt32(ruleSequence[i]));  }  LexemeTree primalTree = new LexemeTree();  primalTree.contains = "S";  string StringGotten = CreateNewBranch(ref primalTree);  return primalTree;  }  private string CreateNewBranch(ref LexemeTree tree)  {  LexemeTree newChild = new LexemeTree();  tree.addChild(newChild);  if (rules.Count == 0) return "";  string stringGottenFromRule = numberToRuleResult[rules.Pop()];  newChild.contains = stringGottenFromRule;  while (stringGottenFromRule.Contains('S'))  {  int indexS = stringGottenFromRule.IndexOf('S');  stringGottenFromRule = stringGottenFromRule.Remove(indexS, 1);  stringGottenFromRule.Insert(indexS, CreateNewBranch(ref newChild));  }  return stringGottenFromRule;  }    }  } |

Листинг 1 – LexemeTree.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace SPOLab3  {  class LexemeTree  {  public string contains;  public List<LexemeTree> childs;  private LexemeTree parent;  public LexemeTree()  {  childs = new List<LexemeTree>();  }  public void addChild(LexemeTree newChild)  {  newChild.parent = this;  childs.Add(newChild);  }  public LexemeTree getParent()  {  return parent;  }  public void setParent(LexemeTree newParent)  {  parent = newParent;  }  public List<string> toLines()  {  List<string> result = new List<string>();  int depth = 0;  printBranch(ref result, this, depth);  return result;  }  private void printBranch(ref List<string> list, LexemeTree node, int depth)  {  if (node == null) return;  list.Add(indentViaDepth(depth) + node.contains);  for (int i = 0; i < node.childs.Count; i++)  {  printBranch(ref list, node.childs[i], depth + 1);  }  }  private string indentViaDepth(int depth)  {  string result = "";  for (int i = 0; i <= depth; i++)  {  result += " ";  }  return result;  }  }  } |

Листинг 1 – Form1.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;  using System.IO;  namespace SPOLab1  {  public partial class Form1 : Form  {  //глобальные переменные  String[] readedStrings;  String currentFile;  binaryTree Tree;    public Form1()  {  InitializeComponent();  currentFile = "";  }  private void button1\_Click(object sender, EventArgs e)  {    try  {  //выбор файла  openFileDialog1.Filter = "Файлы txt|\*.txt";  String fileName = "";  if (openFileDialog1.ShowDialog() == DialogResult.OK)  {  fileName = openFileDialog1.FileName;  }  label2.Text = fileName;  currentFile = fileName;  //чтение из файла  if (fileName != "")  {  StreamReader reader = new StreamReader(fileName);  int stringCount = 0;  String bufer;  //количество строк в файле  while ((bufer = reader.ReadLine()) != null)  {  stringCount++;  }  //создание новой коллекции строк  readedStrings = new string[stringCount];  //обнуление значений для считывания  reader.BaseStream.Position = 0;  stringCount = 0;  while ((bufer = reader.ReadLine()) != null){  readedStrings[stringCount] = bufer;  stringCount++;  }  //вывод содержимого файла  textBox1.Clear();  for (int i = 0; i < stringCount; i++)  {  textBox1.AppendText(readedStrings[i] + Environment.NewLine);  }  reader.Close();  }  }  catch (Exception E)  {  MessageBox.Show(E.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);  }        }  private void button2\_Click(object sender, EventArgs e)  {    String[] strings = new string[textBox1.Lines.Count()];  for (int i = 0; i < textBox1.Lines.Count(); i++)  {  strings[i] = textBox1.Lines[i];  }  //лексический анализатор  SPOLab2.LexicalAnalyser lexicalAnalyser = new SPOLab2.LexicalAnalyser();  List<SPOLab2.Lexeme> lexemes = lexicalAnalyser.analyseThisText(strings);  dataGridView1.Rows.Clear();  for (int i = 0; i < lexemes.Count; i++)  {  dataGridView1.Rows.Add(lexemes[i].stringNumber, lexemes[i].name, lexemes[i].type, lexemes[i].value);  }  dataGridView2.Rows.Clear();  for (int j = 0; j < lexicalAnalyser.errorInfos.Count; j++)  {  dataGridView2.Rows.Add(lexicalAnalyser.errorInfos[j].stringNumber, lexicalAnalyser.errorInfos[j].info);  }  //создание таблицы идентификаторов  binaryTree identifierTable = new binaryTree(lexemes.Count);  int operations = 0;  Dictionary<string, int> idCounts = new Dictionary<string, int>();  List<SPOLab2.Lexeme> identifiers = new List<SPOLab2.Lexeme>();  for (int i = 0; i < lexemes.Count; i++)  {  if (!identifierTable.isItExist(lexemes[i].name, ref operations))  {  identifierTable.put(lexemes[i].name, ref operations);  identifiers.Add(lexemes[i]);  idCounts.Add(lexemes[i].name, 1);  }  else  {  int countGotten;  if(idCounts.TryGetValue(lexemes[i].name, out countGotten))  {  idCounts[lexemes[i].name]++;  }  }  }  dataGridView3.Rows.Clear();  for (int i = 0; i < identifiers.Count; i++)  {  dataGridView3.Rows.Add(identifiers[i].name, identifiers[i].stringNumber, idCounts[identifiers[i].name]);  }  //синтаксический анализатор  SPOLab3.SyntaxAnalyser syntaxAnalyser = new SPOLab3.SyntaxAnalyser();  syntaxAnalyser.analyse(lexemes);  for (int k = 0; k < syntaxAnalyser.errorInfos.Count; k++)  {  dataGridView2.Rows.Add(syntaxAnalyser.errorInfos[k].stringNumber, syntaxAnalyser.errorInfos[k].info);  }  syntaxTreeLabel.Text = "";  for (int i = 0; i < syntaxAnalyser.lexemeTrees.Count; i++)  {  List<string> linesGotten = syntaxAnalyser.lexemeTrees[i].toLines();  for (int j = 0; j < linesGotten.Count; j++)  {  syntaxTreeLabel.Text += linesGotten[j] + Environment.NewLine;  }  }    }  private void button3\_Click(object sender, EventArgs e)  {  saveFileDialog1.DefaultExt = ".txt";  if (saveFileDialog1.ShowDialog() == DialogResult.OK)  {  String savedFile = saveFileDialog1.FileName;  StreamWriter writer = new StreamWriter(savedFile);  Random rng = new Random();  int randomValue = 0;  for (int i = 0; i < 1000000; i++)  {  randomValue = rng.Next(0, 1000000000);  writer.WriteLine("id" + randomValue);  }  writer.Close();  }  }  private void button4\_Click(object sender, EventArgs e)  {    }  }  } |

Код лексического анализатора и языка представлен в предыдущей работе.

1. **Заключение**

В ходе данной работы был разработан синтаксический анализатор для языка, описанного грамматикой. Для его реализации были построены таблицы крайних левых и крайних правых символов и терминальных символов. На основе исходных правил грамматики и построенных таблиц, была построена таблица операторного предшествования и остовная грамматика. При помощи алгоритма «сдвиг-свертка» для таблицы операторного предшествования и правил остовной грамматики был программно реализован синтаксический анализатор, определяющий правильность написания синтаксиса входной строки, а также восстанавливающий последовательность использованных правил грамматики и строящий простейшее дерево синтаксического вывода. Таким образом, цель данной лабораторной работы была успешно выполнена.