Московский авиационный институт

(национальный исследовательский университет)

Институт «Компьютерные науки и прикладная математика»

**Курсовой проект по курсу**

**«Системы программирования»**

**IV семестр**

Конечный Автомат с операционными символами

Студент: Мерц Савелий

Группа: М8О-207Б-20

Руководитель: Семёнов А. С.

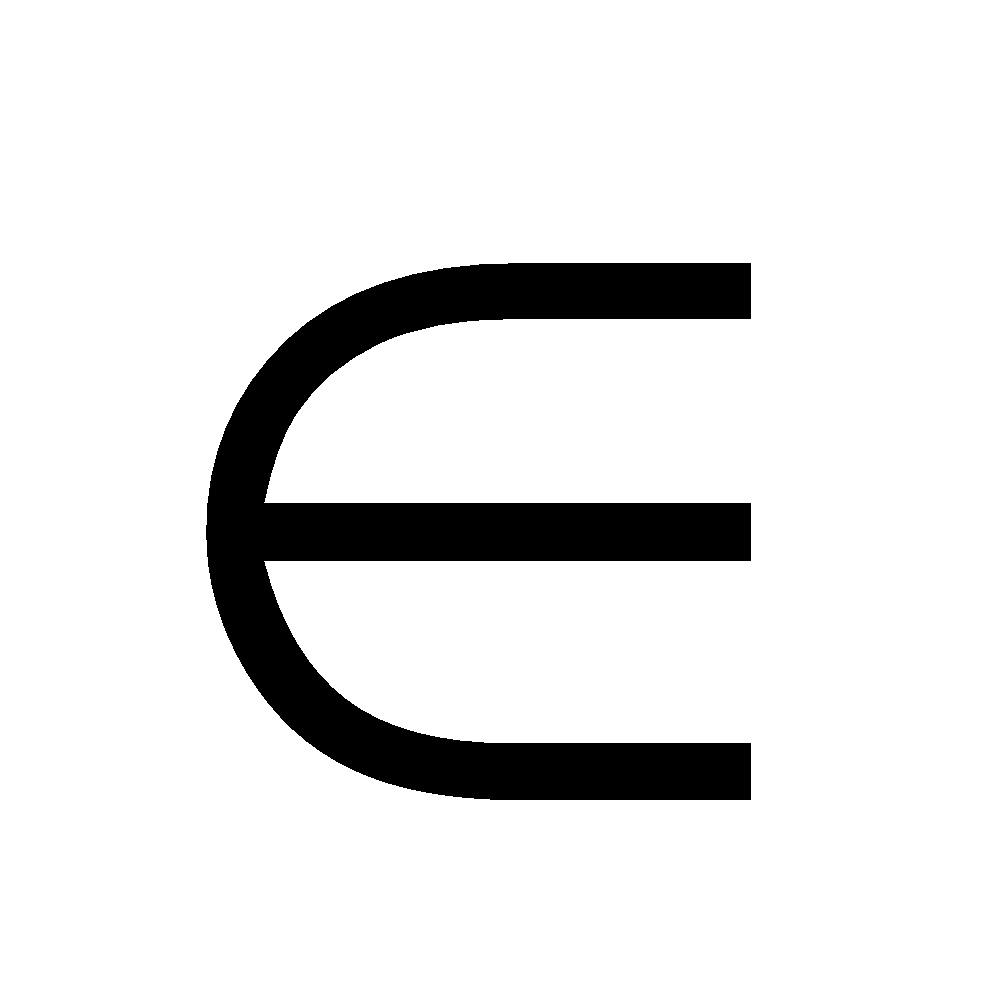
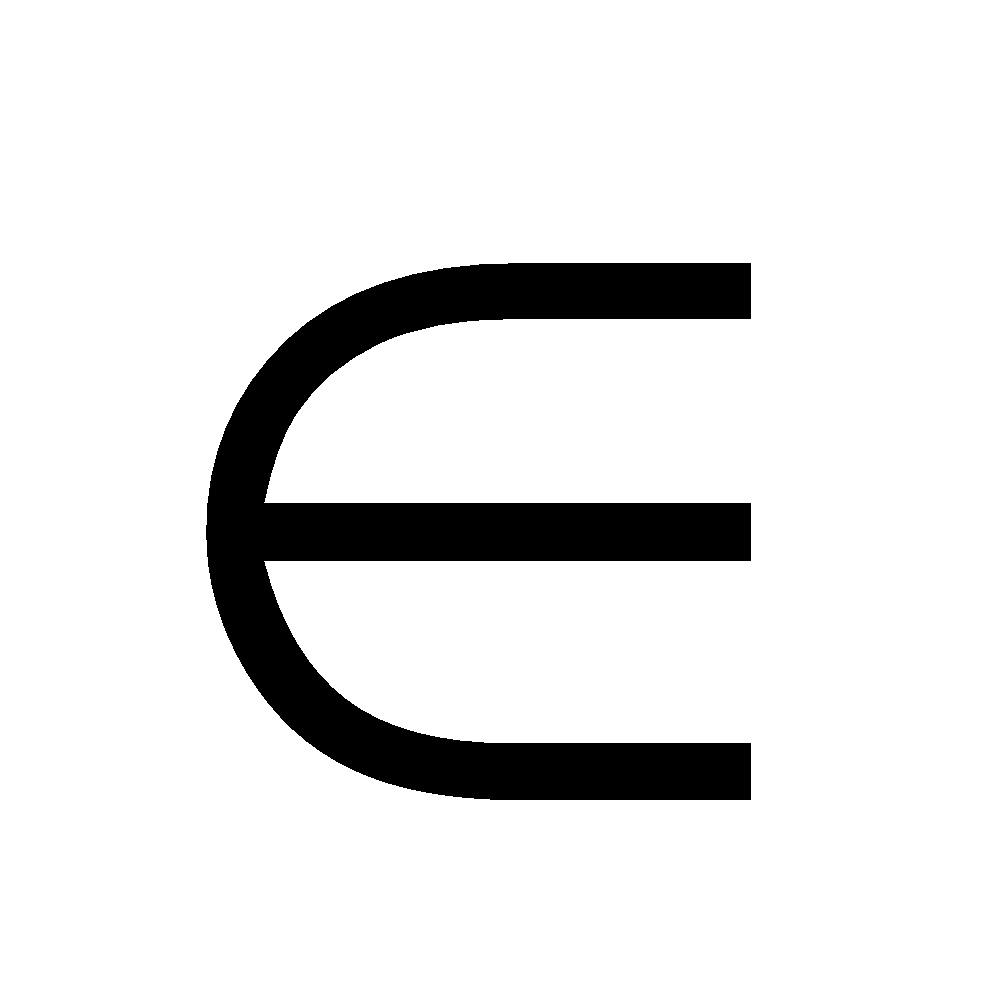
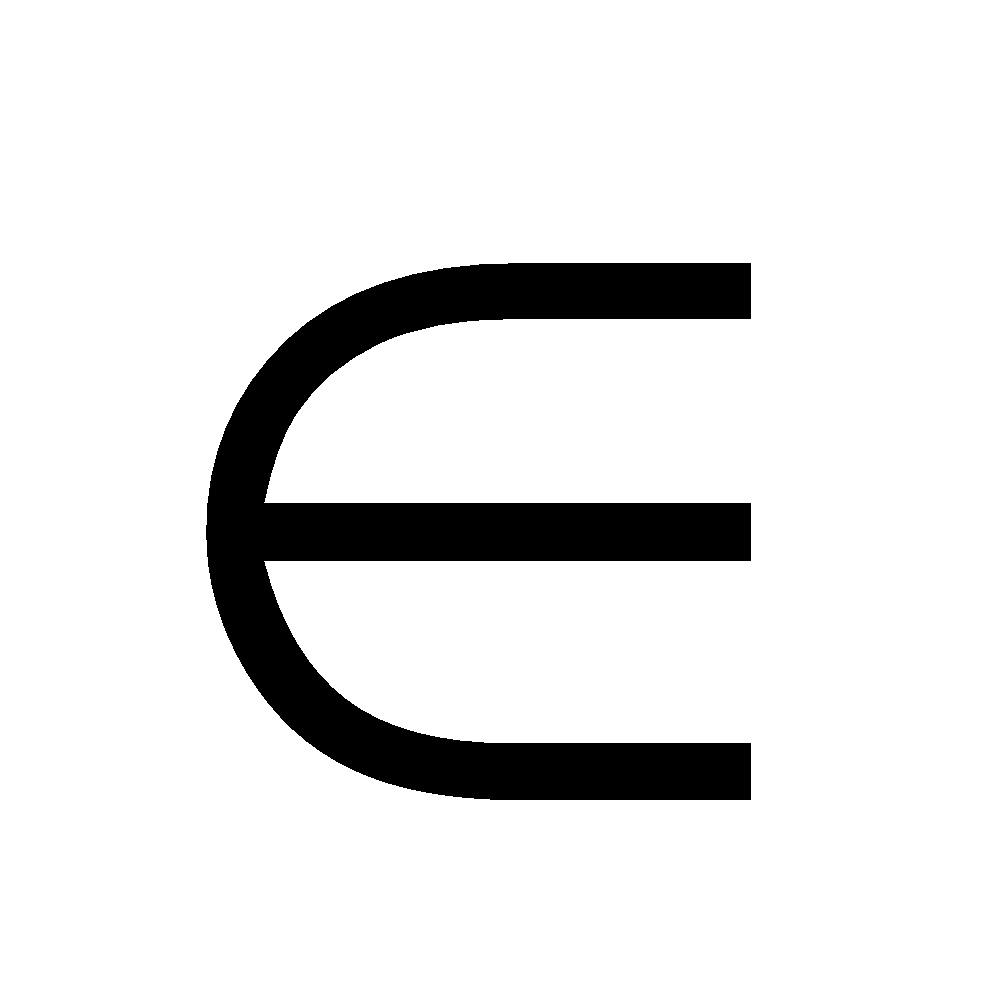
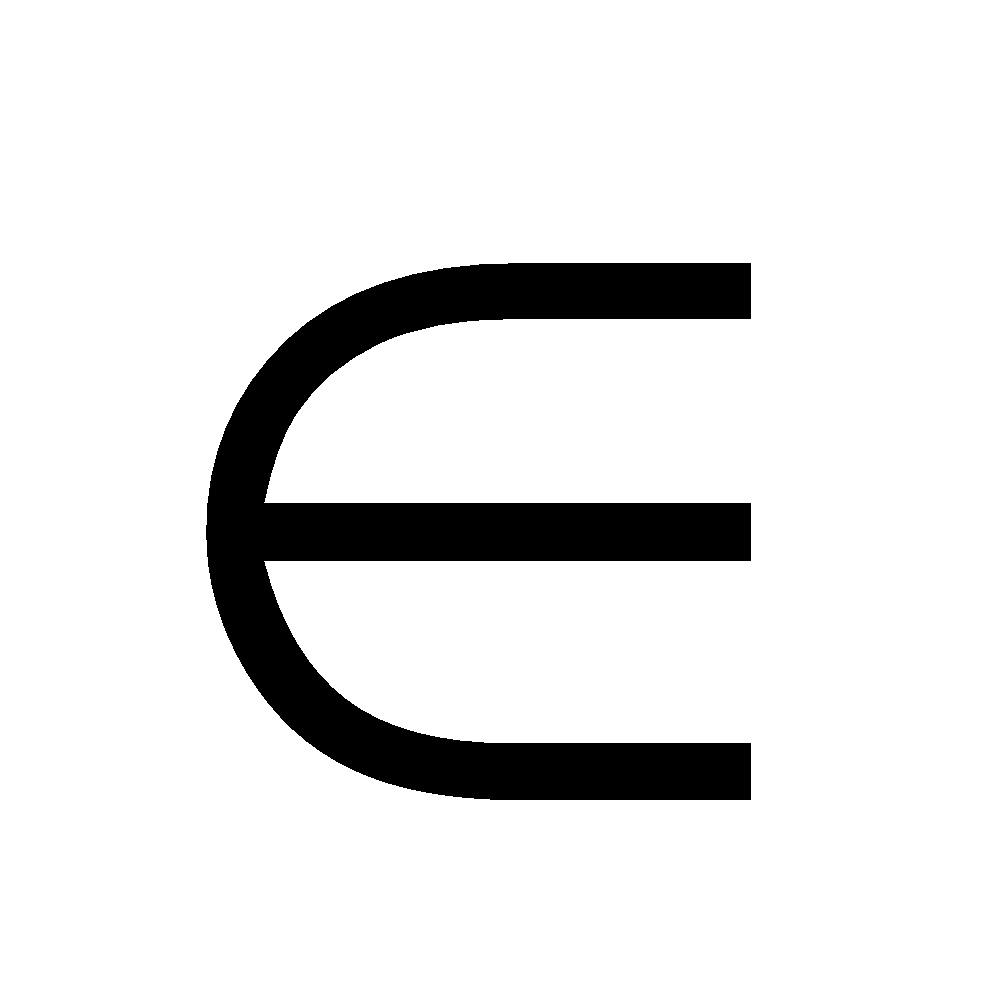
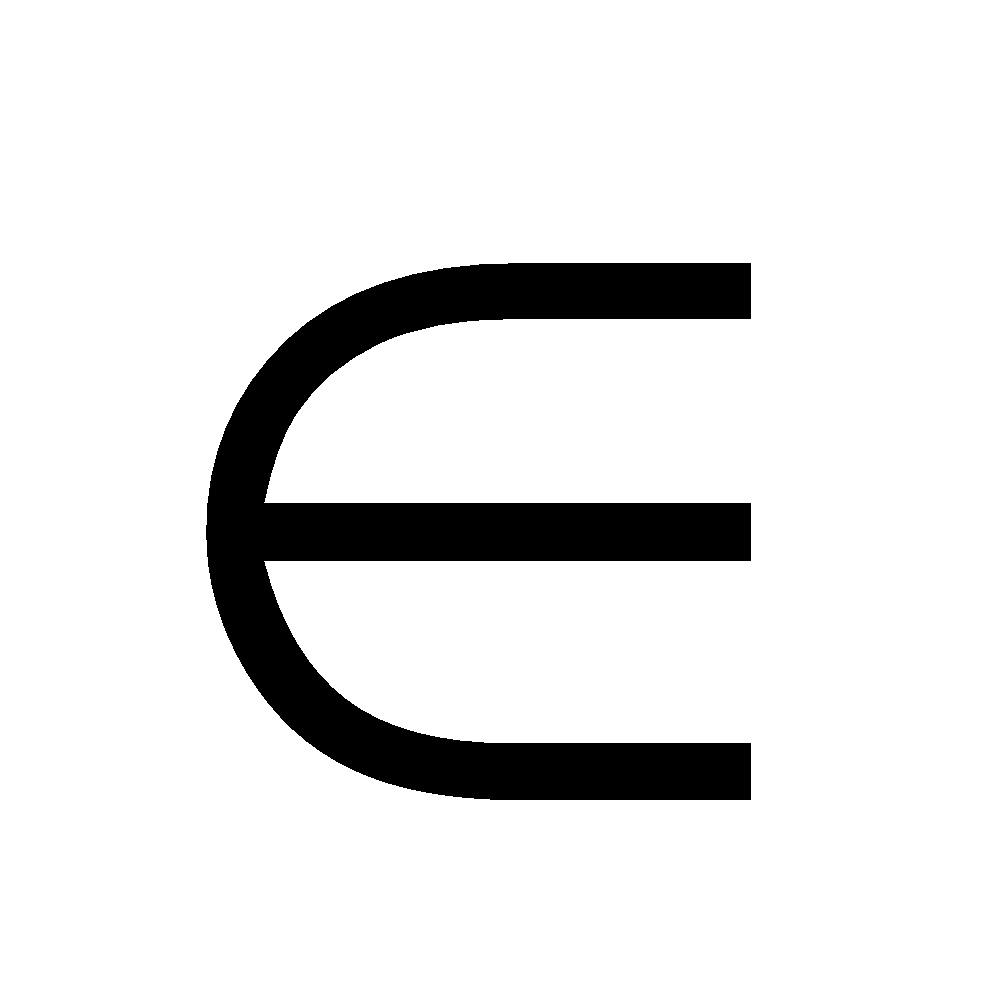
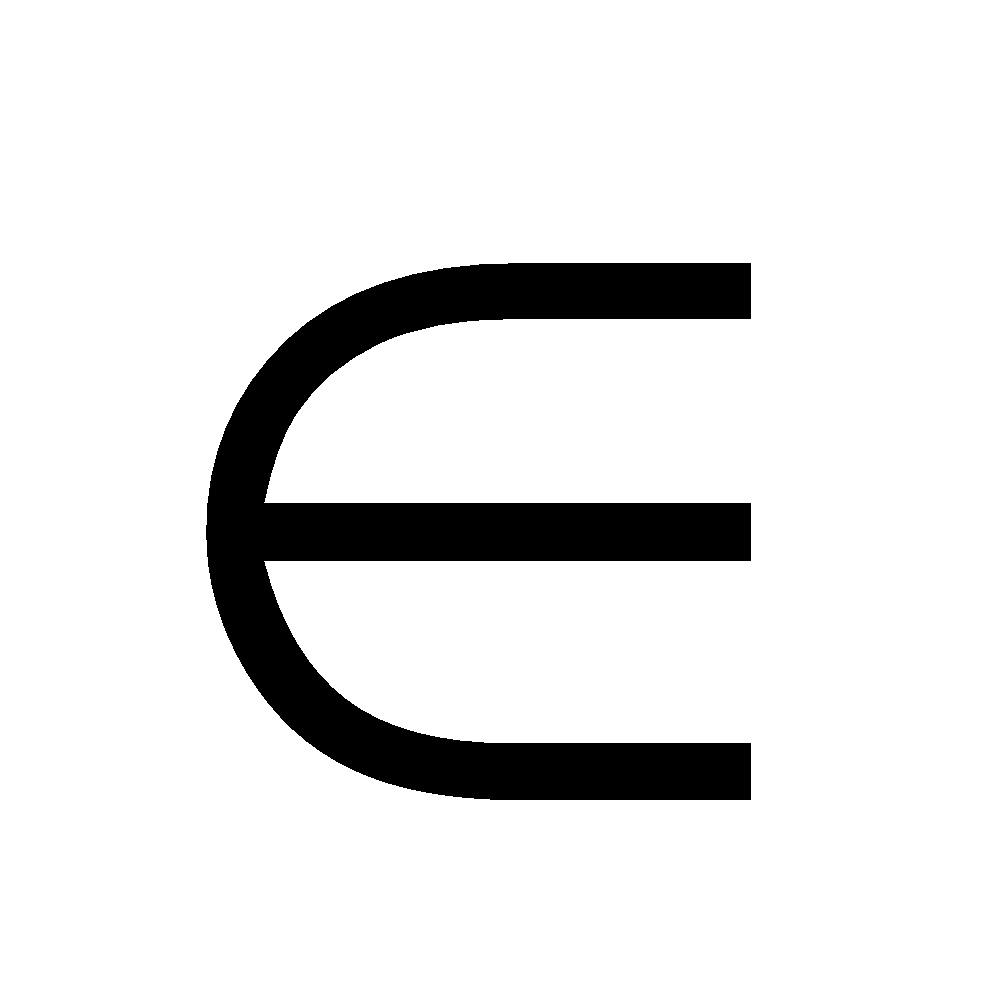
Оценка:

Дата:

Регулярные грамматики (G) и автоматы с конечным состоянием (КА) [1-3] широко применяются при решении многих задач: при определении принадлежности заданной цепочки символов к определенному языку, при разборе и генерации цепочек символов, для разработки лексических анализаторов компиляторов. текстовых редакторов, средств проверки орфографии, распознавания шаблонов с помощью регулярных выражений, проектировании комбинированных и последовательных схем, etc. Применение регулярной грамматики с операционными символами позволяет расширить возможности их применения и эффективного использования. Введем основные определения и алгоритм преобразования регулярной грамматики с операционными символами в КА. Операционными символами будем называть операции заключённые в фигурные скобки, например, операции ПРОЧИТАТЬ, РАЗДЕЛИТЬ, СОХРАНИТЬ, цепочку символов представляются в виде операционных символов: {ПРОЧИТАТЬ} {РАЗДЕЛИТЬ}, {СОХРАНИТЬ}.

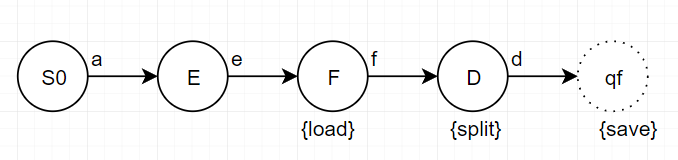
**Определение 1**. Конечный автомат (КА) с операционными символами это шестерка объектов KA = (Q, ∑, ∑op, δ, q0, F), где Q - конечное множество состояний; ∑ - конечный алфавит входных символов; ∑op - конечный алфавит операционных символов; δ - конечное множество функций (правил) переходов, задается отображением δ:Q×∑→ Q, q0 - начальное состояние автомата, q0∈Q; F ⊆ Q - множество заключительных состояний.

Преобразуем грамматику в конечный автомат (КА), так что бы языки оставались эквивалентными L(G) = L(КА).

**Определение 2.** Конфигурация КА – это пара множества (q, ω) Q×∑\*, где qQ, ω∑\*. Конфигурация (q0, ω) называется начальной, а (q,ε), где qF, - заключительной. Бинарное отношение ⎥⎯ на конфигурациях, соответствует такту работы КА. Если q'δ(q,a), то (q, aω) ⎥⎯ (q’, ω) для всех ω∑\*.

Работа КА по распознаванию цепочки aefd представлена следующим образом: (S0, aefd) ⎥⎯1 (E, efd)⎥⎯2 (F{ПРОЧИТАТЬ}, fd) ⎥⎯3 (D{РАЗДЕЛИТЬ}, d) ⎥⎯4{qf {СОХРАНИТЬ}}. Две цепочки ae2088f и d сохранены в файле.

Диаграмма



**Программная реализация**

Для реализации конечного автомата с операционными символами написал класс наследуемый от класса конечного автомата(FSAutomate).

Добавил два новых поля (множество операционных символов и словарь с правилами) перегружен конструктор, метод добавления правил и метод выполнения автомата.

**Добавление нового правила(AddRule)**:

Так как операционный символ по определению 2 пишется в поле следующего состояния автомата, то туда и запишем его после нетерминального символа.

**Выполнение автомата(Execute)**:

Для обмена данными между операцией и методом выполнения автомата придумал “интерфейс” в виде входной строки. С её помощью можем обмениваться данными между разными операциями и выполнять

их. Запись выходных данных операции производится после последнего считанного символа входной строки, который мы находим через индекс, который передаем в операцию. Остальной алгоритм идентичен классу(FSAutomate), от которого наследуемся.

**Словарь (символ, операция):**

| {save} | Операция сохранения в файл (Save) |
| --- | --- |
| {load} | Операция чтения из файла (Load) |
| {split} | Разрыв строки (Split) |

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Collections;

**using** RaGlib.Core;

**using** RaGlib.Automata;

**using** System.Text.RegularExpressions;

**using** System.IO;

**using** System.Text;

**namespace** RaGlib {

**public** **class** FSAutomateWithOpSymbols : FSAutomate

{

**private** **delegate** **string** operation(**string** inStr, **int** indxInStr);

**private** Dictionary<OPSymbol, operation> OpDictionary = new Dictionary<OPSymbol, operation>();

**public** List<OPSymbol> SigmaOP { **set**; **get**; } = **null**; *// множество алфавит операционных символов*

**public** FSAutomateWithOpSymbols(List<Symbol> Q, List<Symbol> Sigma, List<OPSymbol> SigmaOP, List<Symbol> F, Symbol q0)

{

**this**.Q = Q;

**this**.Sigma = Sigma;

**this**.SigmaOP = SigmaOP;

**this**.Q0 = q0;

**this**.F = F;

**this**.Delta = new List<DeltaQSigma>();

operation OP = new operation(Save);

OpDictionary.**Add**(new OPSymbol("{save}"), OP);

OP = new operation(Load);

OpDictionary.**Add**(new OPSymbol("{load}"), OP);

OP = new operation(Split);

OpDictionary.**Add**(new OPSymbol("{split}"), OP);

}

**private** **static** **string** Save(**string** inStr, **int** indxInStr)

{

Console.WriteLine("Введите путь и имя файла**\n**");

**string** path = Console.ReadLine();

**if** (!File.Exists(path))

{

*// Create a file to write to.*

File.WriteAllText(path, inStr.Substring(0, indxInStr + 1) + Environment.NewLine);

}

**return** inStr;

}

**private** **static** **string** Load(**string** inStr, **int** indxInStr)

{

Console.WriteLine("Введите путь и имя файла**\n**");

**string** path = Console.ReadLine();

*// Create a file to write to.*

**string** filetext = File.ReadAllText(path);

filetext = filetext.Substring(0, filetext.Length );

**string** readSubStrInStr = inStr.Substring(0, indxInStr + 1);

**string** nonreadSubStrInStr = inStr.Substring(indxInStr + 1);

inStr = readSubStrInStr + filetext + nonreadSubStrInStr;

**return** inStr;

}

**private** **static** **string** Split(**string** inStr, **int** indxInStr)

{

**string** readSubStrInStr = inStr.Substring(0, indxInStr + 1);

**string** nonreadSubStrInStr = inStr.Substring(indxInStr + 1);

inStr = readSubStrInStr + Environment.NewLine + nonreadSubStrInStr;

**return** inStr;

}

**public** **void** AddRule(**string** state, **string** term, **string** opSymbol, **string** nextState)

{

**this**.Delta.**Add**(new DeltaQSigma(state, term, new List<Symbol> { new Symbol(nextState), new OPSymbol(opSymbol) }));

}

**public** new **void** Execute(**string** chineSymbol)

{

**int** InStrLenght = chineSymbol.Length; *// для отслеживания изменения длины строки*

**var** currState = **this**.Q0;

**int** flag = 0;

**int** i = 0;

**for** (; i < chineSymbol.Length; i++)

{

flag = 0;

**foreach** (**var** d **in** **this**.Delta)

{

**if** (d.LHSQ == currState && d.LHSS == chineSymbol.Substring(i, 1))

{

currState = d.RHSQ[0].symbol; *// Для детерминированного К автомата*

**if** (d.RHSQ.Count > 1) *// говорит о том что есть оп символ*

{

operation OP = OpDictionary[new OPSymbol(d.RHSQ[1].symbol)];

chineSymbol = OP(chineSymbol, i);

*// длина строки может измениться из-за операции, ктоторая что-то туда записала*

i = i + chineSymbol.Length - InStrLenght;

InStrLenght = chineSymbol.Length;

}

flag = 1;

**break**;

}

}

**if** (flag == 0) **break**;

} *// end for*

Console.WriteLine("Length: " + chineSymbol.Length);

Console.WriteLine(" i :" + i.ToString());

Debug("curr", currState.symbol);

**if** (**this**.F.Contains(currState) && i == chineSymbol.Length)

Console.WriteLine("chineSymbol belongs to language");

**else**

Console.WriteLine("chineSymbol doesn't belong to language");

} *// end Execute*

}

}

**Пример выполнения:**

**var** Gram = new GrammarWithOpSymbol(new List<Symbol>() { "a","e", "f", "d" },

new List<OPSymbol>() { new OPSymbol("{save}"), new OPSymbol("{split}"), new OPSymbol("{load}")},

new List<Symbol>() { "S0","E", "F", "D" },

"S0");

Gram.AddRule("S0",new List<Symbol>() { "a", "E" });

Gram.AddRule("E",new List<Symbol>() { "e", "F", "{load}"});

Gram.AddRule("F",new List<Symbol>() { "f", "D", "{split}" });

Gram.AddRule("D",new List<Symbol>() { "d", "{save}"});

**var** KA = Gram.Transform();

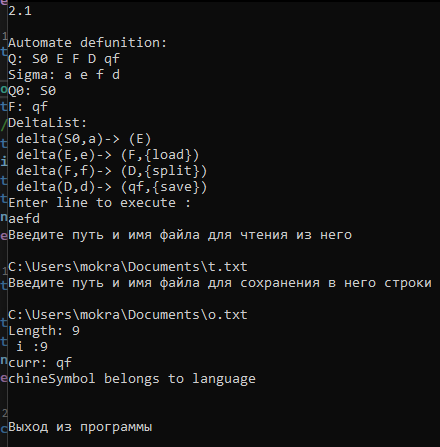
KA.DebugAuto();

Console.WriteLine("Enter line to execute :");

KA.Execute(Console.ReadLine());

Console.ReadLine();

**break**;



**Вывод:**

При выполнении работы я познакомился со словарями в C# и поработал в команде с большим проектом. Очень рад полученному опыту.