**­­­Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Системы программирования»

**Курсовая работа**

Тема: Разработка АТ-грамматики для трансляции алгоритма "устранение цепных правил" в форме конструктора множеств с оператором вывода "=>" в алгоритм с операторами типа foreach.

Студент: Деревянко Е.А.

Группа: М8О-206Б-21

Преподаватель: Семёнов А.С.

Дата: 07.06.2023

Оценка:

Москва, 2023

**Алгоритм типа foreach:**

P’ =

foreach (A→αRβP)

if (α = ε, β = ε, RViA)

foreach (RViA)

if (R→YρP, YViA)

P’= P’{A→Yρ}

end

end

else

P’= P’{A→αRβ}

end

end

**Разработаем конструктор множеств для алгоритма устранения цепных правил, опираясь на определение конструктора множеств и описание алгоритма с операторами типа foreach:**

{P’ | P’ = } = {A→αRβP | α = ε, β = ε, RViA} {{RViA | R→YρP, YViA} {P’= P’{A→Yρ}}}{P’= P’{A→αRβP}}

Обозначим:

@1 := P’ =

@2 := A→αRβP

@3 = α = ε

@4:= β = ε

@5:= RViA

@6:= R→YρP

@7:= YViA

@8:= P’= P’{A→Yρ}

@9:= P’= P’{A→αRβP}

Анализ выполняемого перевода позволяет сделать вывод о том, что выходная цепочка включает в себя объекты (сущности) двух видов:

* Объекты, не изменяющиеся в процессе перевода (множество терминальных символов выходного языка {term1, …, term22}.
* Объекты, генерируемые во время перевода {foreach, \n, \t, end}.

Так как объекты, включаемые в выходную цепочку, не передаются, то можно сделать вывод, что атрибуты в данном контексте не имеют смысла.

Составим БНФ, которая описывает синтаксис заданного конструктора множеств:

1. <алгоритм> ::= {<множество>|<терминал>} = {<терминал> | <задание вида правил>} {<добавление новых нецепных правил>}{<добавление изначально нецепных правил>}  
   (2)<задание вида правил> ::= <терминал>, <терминал>, <терминал>  
   (3)<добавление новых нецепных правил> ::= <правая нецепная часть правил> <добавление новых правил в множество>  
   (4)<правая нецепная часть правил> ::= {<терминал> | <терминал>, <терминал>}

(5)<добавление новых правил в множество> ::= {<терминал>}

(6) <добавление изначально нецепных правил> ::= <терминал>

Сначала составим КС-грамматику.

Введём обозначения: для алгоритма 1: S - <алгоритм>, А - <задание множества цепных терминалов>, В - <множество итераций цепных нетерминалов>, С - <условия цепных правил>, D - <добавление в множество цепных нетерминалов>.

Для алгоритма 2: S - <алгоритм>, А - <задание вида правил>, В - <добавление новых нецепных правил>, С - <добавление изначально нецепных правил>, D - <правая нецепная часть правил>, E - <добавление новых правил в множество>.

Тогда правила вывода КС-грамматики, описывающей синтаксис входного языка, имеют вид:

1. S → {P’ | @1} = {@2 | A} {B}{C}
2. A → @3, @4, @5
3. B → {D} {E}
4. D → @5 | @6, @7
5. E → @8
6. C → @9

Из каждой Т-грамматики можно получить две обычных грамматики, одна из которых позволяет строить входные цепочки, а другая - выходные. Правила построения таких грамматик можно сформулировать следующим образом:

Если из правил транслирующей грамматики GT удалить выходные символы, то получится входная грамматика GTвх для заданной грамматики. Если из правил заданной транслирующей грамматики удалить входные символы, то получится выходная грамматика GTвых для заданной транслирующей грамматики GT. Язык, порождаемый грамматикой GTвх, называется входным языком заданной транслирующей грамматики, а язык порождаемый GTвых называется выходным языком заданной транслирующей грамматики GT.

**Построим СУ-схему для заданного перевода:**

|  |  |  |
| --- | --- | --- |
|  | Правило входной грамматики | Правило выходной грамматики |
| (1) | S→{P’|@1}={@2|A}{B}{C} | S→@1\nforeach(@2)\n\tA\n\t\tB\n\t\tC\nend |
| (2) | A → @3, @4, @5 | A → if(@3,@4,@5) |
| (3) | B → {D} {E} | B → D\n\t\t\t\tE |
| (4) | D → @5 | @6, @7 | D → foreach(@5)\n\t\t\tif(@6,@7) |
| (5) | E → @8 | E → @8\n\t\t\tend\n\t\tend\n\telse |
| (6) | C → @9 | C → @9\n\tend |

**Построим Т-Грамматику, опираясь на данную СУ-схему:**

В начале по заданной СУ-схеме создается транслирующая грамматика Г. Это всегда можно сделать, поскольку заданная СУ-схема должна быть простой. Если входная грамматика заданной СУ-схемы относится к классу LL(1) -грамматик, то и входная грамматика транслирующей грамматики также будет относиться к этому классу.

Контекстно-свободная грамматика, не содержащая аннулирующих правил, называется разделенной или простой, если выполняются следующие два условия:

1. Правая часть каждого правила начинается терминалом.
2. Если два правила имеют одинаковые левые части, то правые части этих правил должны начинаться различными терминальными символами.

Входная КС-грамматика удовлетворяет этим условиям, значит, она является LL(1) грамматикой, следовательно, Т-грамматика также.

Зададим транслирующую грамматику GT по заданной СУ-схеме (в квадратных скобках входная часть, в фигурных выходная):

1. S → [{P’ | @1} = {@2] <@1\nforeach(@2)\n\t>[|] A[} {]<\n\t\t>B[}{]<\n\t\t>C[}] <\nend>
2. A → [@3, @4, @5]<if(@3,@4,@5)>
3. B → [{]D[} {]<\n\t\t\t\t>E[}]
4. D → [@5 | @6, @7]<foreach(@5)\n\t\t\tif(@6,@7)>
5. E → [@8]<@8\n\t\t\tend\n\t\tend\n\telse>
6. C → [@9]<@9\n\tend>

Обозначим:

a := [{P’ | @1} = {@2];

<a> := <@1\nforeach(@2)\n\t>;

b := [|];

c:= [} {]

<b> := <\n\t\t>;

d := [}{];

<c> := <\nend>;

e := [}];

<d> := <if(@3,@4,@5)>;

f := [@3, @4, @5];

<e> := <\n\t\t\t\t>;

g := [{];

<f> := <foreach(@5)\n\t\t\tif(@6,@7)>;

h := [@5 | @6, @7];

<g> := <@8\n\t\t\tend\n\t\tend\n\telse>;

i := [@8];

<h> = <@9\n\tend>;

j := [@9];

С учетом вышеперечисленных обозначений Т-грамматика будет выглядеть так:

1. S → a<a>bAc<b>Bd<b>Ce<c>
2. A → f<d>
3. B → gDc<e>Ee
4. D → h<f>
5. E → i<g>
6. C → j<h>

Рассмотрим построение выходной цепочки для входа, представляющего алгоритм устранения левой рекурсии в виде конструктора множеств:

{P’ | @1} = {@2 | @3, @4, @5} {{@5| @6, @7} {@8}}{@9}

С учетом обозначений выше эта цепочка будет выглядеть так:

abfcghciedje

**Преобразователем с магазинной памятью (МП) называется совокупность восьми объектов:**

МП = {P, S, s0, H, h0, F, W, ϕ},

где P - входной алфавит, состоящий из символов, записываемых на входную ленту; W - выходной алфавит, содержащий символы, записываемые на выходную ленту; H - магазинный алфавит, содержащий символы, записываемые и считываемые из магазина;

h0 - маркер дна магазина, h0 H;

S - множество состояний преобразователя; s0- начальное состояние из множества S;

F - множество конечных состояний, представляющих собой подмножество S;

ϕ - функция переходов преобразователя, которая задает отображение,

ϕ: S x {P {ε} x H S x H\* x W\*.

Она может быть записана в функциональном виде:

ϕ(s, p, h) = (s', β, γ),

где h ∈ H, p ∈ P, β ∈ H\*, γ ∈ W\* и s, s' ∈ S.

Поскольку преобразователь должен в процессе формирования выхода осуществлять распознавание входной цепочки, он строится на основе транслирующей грамматики с использованием правил построения распознавателя. Преобразователь должен выполнять левый вывод входной цепочки в магазине и удалять терминальные символы, находящиеся в вершине, при совпадении их с очередным символом на входной ленте. Поскольку в магазине будут находиться и выходные символы, содержащиеся в правилах T–грамматики дополним правила построения преобразователя следующим правилом: при появлении выходного символа в вершине магазина он передается на выход независимо от символа, находящегося под входной головкой.

Определим компоненты МП следующим образом:

S = {s0}, P = VTвх, H = VTвх VA {h0} VTвых, F = {s0}, W = VTвых

Определим функции переходов МП:

(1) Для всех правил вида A → bα, где b ∈ VTвх и α ∈ (VTвх VTвых  VA)\*, строятся команды:

ϕ(s0, b, A) = (s0, α', $),

где α' - зеркальное отображение цепочки α.

(2) Для всех правил вида A → Bα, где B ∈ VA и α ∈ (VTвх VTвых  VA)\*, строятся команды:

ϕ\*(s0, u, A) = (s0, α'B, $),

где u ∈ ВЫБОР(A → Bαвх) и αвх - цепочка, полученная из α путем удаления из нее всех выходных символов.

(3) Для всех правил вида A → $ строятся команды:

ϕ\*(s0, u, A) = (s0, $, $),

где u ∈ ВЫБОР(A → $).

(4) Для всех символов b, принадлежащих VTвх, стоящих не только на первом месте в правой части правил транслирующей грамматики, т.е. символов, заносимых в магазин, строятся команды:

ϕ(s0, b, b) = (s0, $, $).

(5) Для всех выходных символов <u>, таких что u ∈ VTвых, строятся команды:

ϕ\*(s0, z, <u>) = (s0, $, u),

где z ∈ VTвх {ε}

Т.е. команды строятся для сочетаний <u>z, таких, что z может следовать за <u> в цепочках, выводимых в заданной грамматике.

(6) Заключительная команда имеет вид:

ϕ\*(s0, ε, h0) = (s0, $, $).

Используя правило построения команд преобразователя (1) для правил грамматики, начинающихся входным терминальным символом, получаем команды преобразователя МП1:

1. ϕ(s0, a, S) = (s0, <c>eC<b>dB<b>cAb<a>, $)

2. ϕ(s0, f, A) = (s0, <d>, $)

3. ϕ(s0, g, B) = (s0, eE<e>cD, $)

4. ϕ(s0, j, C) = (s0, <h>, $)

5. ϕ(s0, h, D) = (s0, <f>, $)

6. ϕ(s0, i, E) = (s0, <g>, $)

Правила построения команд вида (2),(3) к заданной грамматике неприменимы, поэтому с помощью правил (4) и (5) построим команды, обеспечивающие передачу выходных символов на выход. Эти команды имеют вид:

11. ϕ\*(s0, b, <a>) = (s0, $, <a>)

12. ϕ\*(s0, g, <b>) = (s0, $, <b>)

13. ϕ\*(s0, j, <b>) = (s0, $, <b>)

14. ϕ\*(s0, c, <d>) = (s0, $, <d>)

15. ϕ\*(s0, i, <e>) = (s0, $, <e>)

16. ϕ\*(s0, c, <f>) = (s0, $, <f>)

17. ϕ\*(s0, e, <g>) = (s0, $, <g>)

18. ϕ\*(s0, e, <h>) = (s0, $, <h>)

19. ϕ\*(s0, b, b) = (s0, $, $)

20. ϕ\*(s0, c, c) = (s0, $, $)

21. ϕ\*(s0, d, d) = (s0, $, $)

22. ϕ\*(s0, e, e) = (s0, $, $)

23. ϕ\*(s0, ε, <c>) = (s0, $, <c>)

Для перехода в заключительное состояние s1 в соответствии с правилом (6) построим команду:

24. ϕ\*(s0, ε, h0) = (s0, $, $)

Последовательность конфигураций, получаемых с помощью команд преобразователя, имеет вид:

(s0, abfcghciedjeε, h0S, $) 1

(s0, bfcghciedjeε, h0<c>eC<b>dB<b>cAb<a>, $) 11

(s0,bfcghciedjeε, h0<c>eC<b>dB<b>cAb, <a>) 19

(s0, fcghciedjeε, h0<c>eC<b>dB<b>cA, <a>) 2

(s0, cghciedjeε, h0<c>eC<b>dB<b>c<d>, <a>) 14

(s0, cghciedjeε, h0<c>eC<b>dB<b>c, <a><d>) 20

(s0, ghciedjeε, h0<c>eC<b>dB<b>, <a><d>) 12

(s0, ghciedjeε, h0<c>eC<b>dB, <a><d><b>) 3

(s0, hciedjeε, h0<c>eC<b>deE<e>cD, <a><d><b>) 5

(s0, ciedjeε, h0<c>eC<b>deE<e>c<f>, <a><d><b>) 16

(s0, ciedjeε, h0<c>eC<b>deE<e>c, <a><d><b><f>) 20

(s0, iedjeε, h0<c>eC<b>deE<e>, <a><d><b><f>) 15

(s0, iedjeε, h0<c>eC<b>deE, <a><d><b><f><e>) 6

(s0, edjeε, h0<c>eC<b>de<g>, <a><d><b><f><e>) 17

(s0, edjeε, h0<c>eC<b>de, <a><d><b><f><e><g>) 22

(s0, djeε, h0<c>eC<b>d, <a><d><b><f><e><g>) 21

(s0, jeε, h0<c>eC<b>, <a><d><b><f><e><g>) 13

(s0, jeε, h0<c>eC, <a><d><b><f><e><g><b>) 4

(s0, eε, h0<c>e<h>, <a><d><b><f><e><g><b>) 18

(s0, eε, h0<c>e, <a><d><b><f><e><g><b><h>) 22

(s0, ε, h0<c>, <a><d><b><f><e><g><b><h>) 23

(s0, ε, h0, <a><d><b><f><e><g><b><h><c>) 24

(s1, $, h0$, <a><d><b><f><e><g><b><h><c>)

Код программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278 | using System;  using System.Collections.Generic;  using System.Collections;  using RaGlib.Core;  // КП Павлов, М8О-207Б-21  namespace RaGlib {  public class TSymbol: Symbol  {  public bool outSymbol = false;  public TSymbol() : base() {}  public TSymbol(string s, int production, int symbolPosition): base(s, production, symbolPosition) { }  public TSymbol(string s, List<Symbol> attr): base(s, attr) { }  public TSymbol(string value): base(value) { }  public static implicit operator TSymbol(string str) => new TSymbol(str);  public override string ToString() => this != Epsilon ? this.symbol : "e";  }  public class TProduction  {  public TSymbol LHS { set; get; } = null;  public List<TSymbol> RHS { set; get; }  public static int Count = 0;  public int Id;  public TProduction(TSymbol LHS, List<TSymbol> RHS)  {  Count++;  Id = Count;  this.LHS = LHS;  this.RHS = RHS;  }  }    public class TGrammar: AGrammar  {  public List<TSymbol> Tout = null;  public TGrammar() {}  public List<TSymbol> T = null;  public List<TSymbol> V = null;  public List<TProduction> P = null;  public TSymbol S0 = null;  public TGrammar(List<TSymbol> Tin, List<TSymbol> Tout, List<TSymbol> V, string S0)  {  for (int i = 0; i < Tout.Count; i++)  {  Tout[i].outSymbol = true;  }  this.T = Tin;  this.Tout = Tout;  this.V = V;  this.S0 = new TSymbol(S0);  this.P = new List<TProduction>();  }    public override void Inference() {}    public bool wrongCount(List<TSymbol> RHSin, List<TSymbol> RHSout)  {  return RHSin.Count == RHSout.Count;  }    public bool isSymbol(TSymbol LHS, List<TSymbol> RHSin, List<TSymbol> RHSout)  {  if (!V.Contains(LHS)) return false;  for (int i = 0; i < RHSin.Count; i++)  {  if (!V.Contains(RHSin[i]) && !T.Contains(RHSin[i]))  {  return false;  }  if (!V.Contains(RHSout[i]) && !Tout.Contains(RHSout[i]))  {  return false;  }  }  return true;  }  public void AddRule(TSymbol LHS, List<TSymbol> RHSin, List<TSymbol> RHSout)  {  if (!wrongCount(RHSin, RHSout))  {  Console.WriteLine("Wrong count of terminsls");  return;  }  if (!isSymbol(LHS, RHSin, RHSout))  {  Console.WriteLine("Wrong symbol");  return;  }  List<TSymbol> RHS = new List<TSymbol>();  for (int i = 0; i < RHSin.Count; i++)  {  if (V.Contains(RHSin[i]))  {  RHS.Add(RHSin[i]);  }  else  {  RHS.Add(RHSin[i]);  RHSout[i].outSymbol = true;  RHS.Add(RHSout[i]);  }  }  P.Add(new TProduction(LHS, RHS));  }  public void printProductions()  {  Console.WriteLine("Translation grammar:");  for (int i = 0; i < P.Count; i++)  {  Console.Write(i + 1 + ": " + P[i].LHS + " -> ");  for (int j = 0; j < P[i].RHS.Count; j++)  {  if (V.Contains(P[i].RHS[j]))  {  Console.Write(P[i].RHS[j] + " ");  }  else  {  if (!P[i].RHS[j].outSymbol)  {  Console.Write("{" + P[i].RHS[j] + "} ");  }  else  {  Console.Write("<" + P[i].RHS[j] + "> ");    }  }  }  Console.Write("\n");  }  }  }  public class Rule  {  public int type = 0;  public List<TSymbol> LHS = null;  public List<List<TSymbol>> RHS = null;  public Rule(int type, List<TSymbol> LHS, List<List<TSymbol>> RHS)  {  this.type = type;  this.LHS = LHS;  this.RHS = RHS;  }  public void printRule()  {  if (type == 1)  {  Console.Write("ф(");  for (int i = 0; i < LHS.Count; i++)  {  Console.Write(LHS[i].ToString() + " ");  }  Console.Write(") = (");  for (int i = 0; i < RHS.Count;i++)  {  Console.Write("[");  for (int j = 0; j < RHS[i].Count;j++)  {  if (RHS[i][j].outSymbol)  {  Console.Write("<" + RHS[i][j] + "> ");  }  else  {  Console.Write(RHS[i][j] + " ");  }  }  Console.Write("] ");  }  Console.Write(")\n");  }  else  {  Console.Write("ф\*(");  for (int i = 0; i < LHS.Count; i++)  {  if (!LHS[i].outSymbol)  {  Console.Write(LHS[i].ToString() + " ");  }  else  {  Console.Write("<" + LHS[i].ToString() + "> ");  }  }  Console.Write(") = (");  for (int i = 0; i < RHS.Count;i++)  {  Console.Write("[");  for (int j = 0; j < RHS[i].Count;j++)  {  if (RHS[i][j].outSymbol)  {  Console.Write("<" + RHS[i][j] + "> ");  }  else  {  Console.Write(RHS[i][j] + " ");  }  }  Console.Write("] ");  }  Console.Write(")\n");  }  }  }  public class Configuration  {  public List<Rule> rules = new List<Rule>();  public TSymbol state = "s0";  public List<TSymbol> input = null;  public List<TSymbol> stack = null;  public List<TSymbol> output = null;    public Configuration(TSymbol S0, List<TSymbol> input)  {  this.input = input;  stack = new List<TSymbol>() {"h0", S0};  output = new List<TSymbol>();  }  public void addRule(int type, List<TSymbol> LHS, List<List<TSymbol>> RHS)  {  rules.Add(new Rule(type, LHS, RHS));  }    public void printRules()  {  Console.WriteLine("Retail converter configuration rules:");  for (int i = 0; i < rules.Count; i++)  {  Console.Write(i + 1 + ": ");  rules[i].printRule();  }  }  public void execute(List<TSymbol> inputStr, TGrammar grammar)  {  List<TSymbol> inputStack = inputStr;  inputStack.Add("h0");  inputStack.Reverse();  TSymbol curSymbol = inputStack[inputStack.Count - 1];  inputStack.RemoveAt(inputStack.Count - 1);  Console.WriteLine("Trigger timing:");  int iter = 1;  while (curSymbol != "h0")  {  if (curSymbol == "ε")  {  for (int i = 0; i < rules.Count; i++)  {  if (rules[i].LHS[2] == stack[stack.Count - 1] && rules[i].LHS[1] == curSymbol)  {  output.Add(stack[stack.Count - 1]);  stack.RemoveAt(stack.Count - 1);  break;  }  }  }  Console.Write(iter + ": (s0, {");  for (int i = inputStack.Count - 1; i > 0; i--)  {  Console.Write(inputStack[i] + " ");  }  Console.Write("}, {");  for (int j = 0; j < stack.Count; j++)  {  if (stack[j].outSymbol)  {  Console.Write("<" + stack[j] + "> ");  }  else  {  Console.Write(stack[j] + " ");  }  }  Console.Write("}, {");  for (int i = 0; i < output.Count; i++)  {  if (output[i].outSymbol)  {  Console.Write("<" + output[i] + "> ");  }  else  {  Console.Write(output[i] + " ");  }  }  Console.Write("})\n");  if (curSymbol == "ε")  {  curSymbol = "h0";  }  if (grammar.V.Contains(stack[stack.Count - 1]))  {  for (int i = 0; i < rules.Count; i++)  {  if (rules[i].LHS[2] == stack[stack.Count - 1])  {  stack.RemoveAt(stack.Count - 1);  for (int j = 0; j < rules[i].RHS[1].Count; j++)  {  //Console.WriteLine(rules[i].RHS[1][j]);  stack.Add(rules[i].RHS[1][j]);  }  curSymbol = inputStack[inputStack.Count - 1];  inputStack.RemoveAt(inputStack.Count - 1);  break;  }  }  }  else if (stack[stack.Count - 1].outSymbol)  {  for (int i = 0; i < rules.Count; i++)  {  if (rules[i].LHS[2] == stack[stack.Count - 1] && rules[i].LHS[1] == curSymbol)  {  output.Add(stack[stack.Count - 1]);  stack.RemoveAt(stack.Count - 1);  break;  }  }  }  else  {  for (int i = 0; i < rules.Count; i++)  {  if (rules[i].LHS[2] == curSymbol && rules[i].LHS[1] == curSymbol)  {  stack.RemoveAt(stack.Count - 1);  curSymbol = inputStack[inputStack.Count - 1];  inputStack.RemoveAt(inputStack.Count - 1);  break;  }  }  }  iter++;  }  Console.WriteLine("Output:");  for (int i = 0; i < output.Count; i++)  {  Console.Write(output[i]);  }  Console.Write("\n");  }  }  public class KP  {  List<TSymbol> inputStr = null;  TGrammar grammar = null;  Configuration configs = null;  public KP() {  inputStr = new List<TSymbol>();  inputStr.Add(new TSymbol("{P’| P' = ∅} = {A → αRβ ∈ P"));  inputStr.Add(new TSymbol("|"));  inputStr.Add(new TSymbol("α = ε, β = ε, R ∈ Vi"));  inputStr.Add(new TSymbol("} => {"));  inputStr.Add(new TSymbol("{"));  inputStr.Add(new TSymbol("R ∈ Vi| R → Yp ∈ P, Y ∉ Vi"));  inputStr.Add(new TSymbol("} => {"));  inputStr.Add(new TSymbol("P’ = P’ ∪ {A → Yp}"));  inputStr.Add(new TSymbol("}"));  inputStr.Add(new TSymbol("}{"));  inputStr.Add(new TSymbol("P’ = P’ ∪ {A → αRβ ∈ P}"));  inputStr.Add(new TSymbol("}"));    Console.WriteLine("Input:");  for (int i = 0; i < inputStr.Count; i++)  {  Console.Write(inputStr[i]);  }  Console.WriteLine("\n");  grammar = new TGrammar(  new List<TSymbol>()  {  "{P’| P' = ∅} = {A → αRβ ∈ P",  "|",  "α = ε, β = ε, R ∈ Vi",  "} => {",  "{",  "R ∈ Vi| R → Yp ∈ P, Y ∉ Vi",  "} => {",  "P’ = P’ ∪ {A → Yp}",  "}",  "}{",  "P’ = P’ ∪ {A → αRβ ∈ P}",  "}",  },  new List<TSymbol>()  {  "P' = ∅\nforeach(A → αRβ ∈ P)\n\t",  "\n\t\t",  "\nend",  "if(α = ε, β = ε, R ∈ Vi)",  "\n\t\t\t\t",  "foreach(R ∈ Vi)\n\t\t\tif(R → Yp ∈ P, Y ∉ Vi)",  "P’ = P’ ∪ {A → Yp}\n\t\t\tend\n\t\tend\n\telse",  "P’ = P’ ∪ {A → αRβ ∈ P}\n\tend",  },  new List<TSymbol>() {"S", "A", "B", "C", "D", "E"},  "S"  );  grammar.AddRule(  "S",  new List<TSymbol>() { "{P’| P' = ∅} = {A → αRβ ∈ P", "|", "A", "} => {", "B", "}{", "C", "}" },  new List<TSymbol>() { "P' = ∅\nforeach(A → αRβ ∈ P)\n\t", "A", "\n\t\t", "B", "\n\t\t", "C", "\nend" });  grammar.AddRule(  "A",  new List<TSymbol>() { "α = ε, β = ε, R ∈ Vi" },  new List<TSymbol>() { "if(α = ε, β = ε, R ∈ Vi)" });  grammar.AddRule(  "B",  new List<TSymbol>() {"{", "D", "} => {", "E", "}" },  new List<TSymbol>() {"D", "\n\t\t\t\t", "E"});  grammar.AddRule(  "D",  new List<TSymbol>() { "R ∈ Vi| R → Yp ∈ P, Y ∉ Vi" },  new List<TSymbol>() { "foreach(R ∈ Vi)\n\t\t\tif(R → Yp ∈ P, Y ∉ Vi)" });  grammar.AddRule(  "E",  new List<TSymbol>() { "P’ = P’ ∪ {A → Yp}" },  new List<TSymbol>() { "P’ = P’ ∪ {A → Yp}\n\t\t\tend\n\t\tend\n\telse" });  grammar.AddRule(  "C",  new List<TSymbol>() { "P’ = P’ ∪ {A → αRβ ∈ P}" },  new List<TSymbol>() { "P’ = P’ ∪ {A → αRβ ∈ P}\n\tend" });  grammar.printProductions();    // здесь можно сделать лексер с вводом конструктора множеств, вместо статического поля  //for (int i = 0; i < input.Length; i++)  //{  // //Console.WriteLine(input[i]);  // inputStr.Add(new TSymbol(input[i].ToString()));  //}  //inputStr.Add(new TSymbol("ε"));  configs = new Configuration(grammar.S0, inputStr);  for (int i = 0; i < grammar.P.Count; i++)  {  if (grammar.T.Contains(grammar.P[i].RHS[0]) && !grammar.P[i].RHS[0].outSymbol)  {  List<TSymbol> configLHS = new List<TSymbol>() {"s0", grammar.P[i].RHS[0], grammar.P[i].LHS};  List<TSymbol> configRHS1 = new List<TSymbol>() {"s0"};  List<TSymbol> configRHS2 = new List<TSymbol>();  for (int j = grammar.P[i].RHS.Count - 1; j > 0; j--)  {  configRHS2.Add(grammar.P[i].RHS[j]);  }  List<TSymbol> configRHS3 = new List<TSymbol>() { "$" };  List<List<TSymbol>> configRHS = new List<List<TSymbol>>() {configRHS1, configRHS2, configRHS3};  configs.addRule(1, configLHS, configRHS);  }  }  for (int i = 0; i < grammar.P.Count; i++)  {  for (int j = 0; j < grammar.P[i].RHS.Count; j++)  {  if (grammar.P[i].RHS[j].outSymbol)  {  TSymbol configLHS2 = findSymbolIn1("ε", grammar.P[i].RHS, j);  if (configLHS2 == "ε")  {  configLHS2 = findSymbolIn2("ε", grammar.P[i].LHS);  }  List<TSymbol> configLHS = new List<TSymbol>() { "s0", configLHS2, grammar.P[i].RHS[j]};  List<TSymbol> configRHS1 = new List<TSymbol>() { "s0" };  List<TSymbol> configRHS2 = new List<TSymbol>() { "$" };  List<TSymbol> configRHS3 = new List<TSymbol>() { grammar.P[i].RHS[j] };  List<List<TSymbol>> configRHS = new List<List<TSymbol>> { configRHS1, configRHS2, configRHS3 };  configs.addRule(5, configLHS, configRHS);  }  }  }  for (int i = 0; i < grammar.T.Count; i++)  {  List<TSymbol> configLHS = new List<TSymbol>() { "s0", grammar.T[i], grammar.T[i] };  List<TSymbol> configRHS1 = new List<TSymbol>() { "s0" };  List<TSymbol> configRHS2 = new List<TSymbol>() { "$" };  List<List<TSymbol>> configRHS = new List<List<TSymbol>> { configRHS1, configRHS2, configRHS2 };  configs.addRule(4, configLHS, configRHS);  }  List<TSymbol> c6LHS = new List<TSymbol>() { "s0", "ε", "h0"};  List<List<TSymbol>> c6RHS = new List<List<TSymbol>>()  {  new List<TSymbol>() { "s1" },  new List<TSymbol>() { "$" },  new List<TSymbol>() { "$" }  };  configs.addRule(6, c6LHS, c6RHS);  configs.printRules();  }  public TSymbol findSymbolIn1(TSymbol symbol, List<TSymbol> RHS, int pos)  {  if (symbol != "ε")  {  return symbol;  }  for (int i = pos; i < RHS.Count; i++)  {  if (grammar.V.Contains(RHS[i]))  {  for (int j = 0; j < grammar.P.Count; j++)  {  if (grammar.P[j].LHS == RHS[i])  {  symbol = findSymbolIn1(symbol, grammar.P[j].RHS, 0);  }  }  if (symbol != "ε")  {  break;  }  }  else if (grammar.T.Contains(RHS[i]) && !RHS[i].outSymbol)  {  symbol = RHS[i];  }  }  return symbol;  }  public TSymbol findSymbolIn2(TSymbol symbol, TSymbol LHS)  {  if (symbol != "ε")  {  return symbol;  }  for (int i = 0; i < grammar.P.Count; i++)  {  for (int j = 0; j < grammar.P[i].RHS.Count; j++)  {  if (grammar.P[i].RHS[j] == LHS)  {  return findSymbolIn1("ε", grammar.P[i].RHS, j + 1);  }  }  }  return symbol;  }  public void execute()  {  configs.execute(inputStr, grammar);  }  }  } |