Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

Кафедра ПМиК

Курсовая работа по дисциплине

Теория языков программирования и методы трансляции

Вариант №13

Выполнил: студент 4 курса

Ф. ИВТ, группа: ИП-711

Мартасов И. О.

Проверил: доцент кафедры ПМиК

Бах Ольга Анатольевна

Новосибирск, 2020

**Содержание**

1. Текст задания……………………………………………………………….3
2. Описание интерфейса……………………………………………………...3
3. Описание алгоритма решения……………………………………………..5

3.1. Этап 1…………………………………………………………………...5

3.2. Этап 2…………………………………………………………………...7

3.3. Этап 3………………………………………………………………...…8

1. Пример работы программы …………………………………..……….....10
2. Описание основных блоков и кода программы ………………………..30
3. Результаты тестирования…………………………………………………69
4. Приложение. Полный текст программы………………………………...80

**Текст задания**

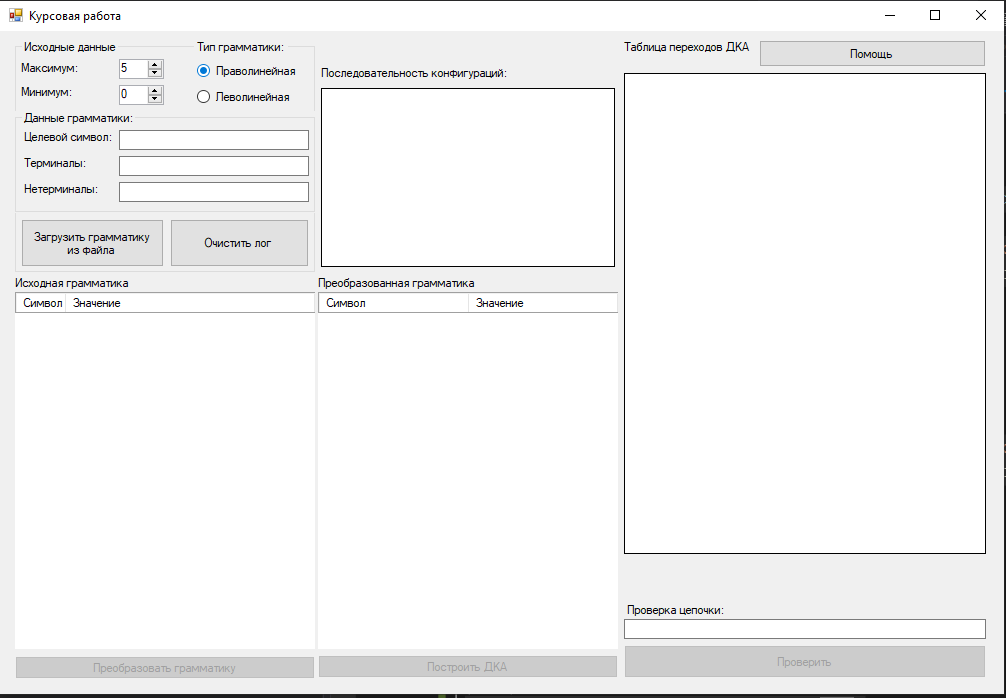
Написать программу, которая по заданной регулярной грамматике (грамматика может быть НЕ автоматного вида!, ЛЛ или ПЛ) построит эквивалентный ДКА (представление функции переходов в виде таблицы). Программа должна сгенерировать по исходной грамматике несколько цепочек в заданном диапазоне длин и проверить их допустимость построенным автоматом. Процессы построения цепочек и проверки их выводимости отображать на экране (по требованию). Предусмотреть возможность проверки цепочки, введённой пользователем.

**Описание интерфейса**

Приложение реализовано на языке C# с помощью WindowForms

Программа принимает на вход регулярную грамматику – целевой символ, алфавит и список нетерминалов, порождающих цепочки. Также выбирается диапазон длин, в котором будут генерироваться цепочки по исходной грамматике и её тип – праволинейная или леволинейная.

Интерфейс приложения имеет следующий вид:



Все процессы генерации цепочек, преобразования грамматики, построения по ней ДКА и дальнейшее распознавание цепочек записываются в лог-файл, который создается в директории C:\CursWorkLogFile\log.txt. По желанию пользователя можно удалить этот файл.

В качестве нетерминалов используются заглавные буквы английского алфавита.

Исходная грамматика может задаваться вручную или из файла.

Формат входных данных файла должен иметь следующий вид:

В первой строке должно быть написано описание грамматикики в виде:

G=({“алфавит”},{“список нетерминалов”},P,”целевой символ”)

где:

“алфавит” – список символов алфавита

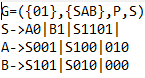
“список нетерминалов” - список нетерминалов, используемых в исходной грамматике

В последующих k строках (k – число нетерминалов) для каждого k-ого нетерминала описываются правила, имеющие вид:

“нетерминал”->”правило 1”|”правило 2”|…

Для разделения правил используется символ |. Все правила для нетерминала должны описываться в одной строке. Для обозначения пустого правила после символа-разделителя ничего не пишется. Алфавит и список нетерминалов пишутся слитно, без разделителей.

Пример правильно составленных входных данных:



**Описание алгоритма решения**

Алгоритм построения ДКА по регулярной грамматике состоит из трех основных этапов:

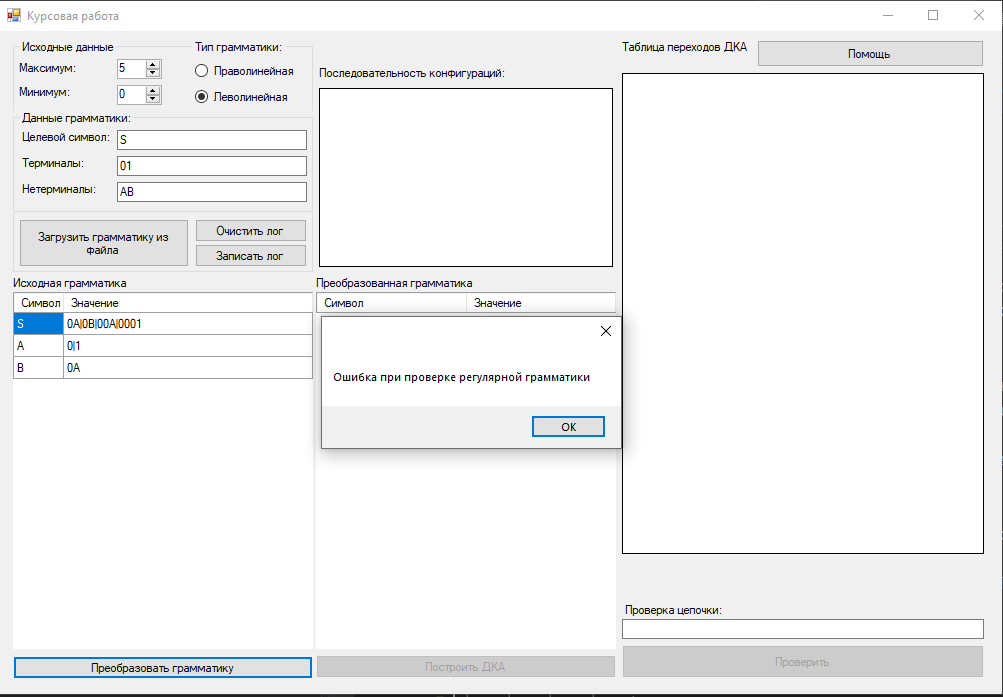
1. Проверка грамматики на корректность, генерация цепочек по исходной грамматике

2. Преобразование правил исходной грамматики к автоматному виду (если нужно)

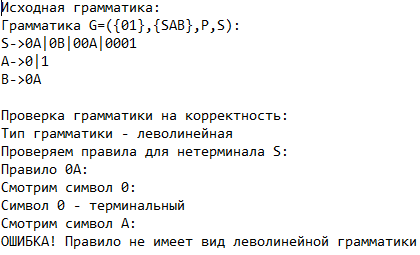
3. Построение ДКА по автоматной грамматике.

**Этап 1:**

По нажатии на клавишу “Преобразовать грамматику” идет проверка соответствия её заданному типу – праволинейному или леволинейному. Это делается путем проверки каждого правила для каждого нетерминала – в частности проверяется количество нетерминалов и их расположение в правиле (либо стоит самым первым, либо последним – в зависимости от типа). В случае ошибки на экран выводится сответствующее сообщение для пользователя, а в лог-файл записывается причина, по которой произошла ошибка.



Пример некорректной обработки в лог-файле:



После проверки на корректность происходит генерация цепочек. Делается это следующим образом:

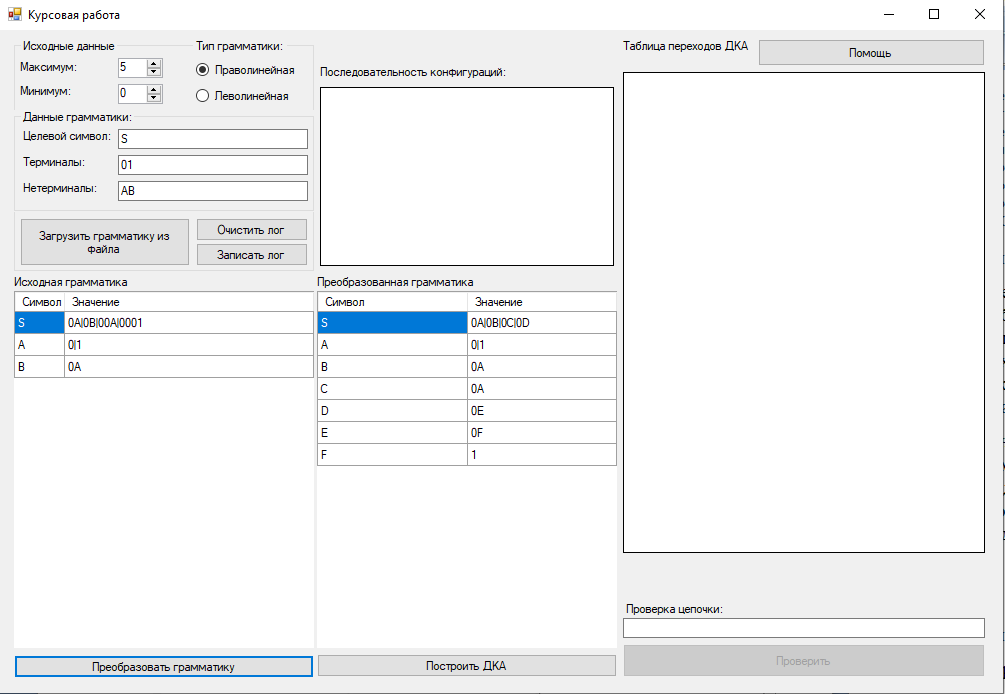
1. Сначала смотрятся правила для целевого символа. Если в правилах присутствуют только терминалы или есть пустое правило – то цепочка сразу заносится в список сгенерированных цепочек, иначе – в список несгенерированных цепочек.
2. Если список несгенерированных цепочек пуст – процесс генерации закончен.
3. Иначе – каждая цепочка, присутствующая в списке несгенерированных, просматривается на наличие нетерминалов в ней, и в случае нахождения нетерминала, для нее применяют поочередно все правила, которые этот нетерминал порождает.
4. Если получившаяся новая цепочка не содержит нетерминалов и если она попадает под диапазон длин, заданный пользователем – её заносят в список сгенерированных, иначе – в список несгенерированных. После того, как для цепочки будут рассмотрены все правила – она удаляется из списка.
5. Генерация заканчивается либо когда в списке сгенерированных цепочек присутствует 5 или более цепочек (включая пустую), (необходимый минимум для проверки по автомату), либо по окончании времени, отведенным под генерацию (на случай бесконечной рекурсии).

В программе процессы проверки грамматики и генерации цепочек не отображаются – они описаны в лог-файле.

**Этап 2:**

Далее идет преобразование исходной грамматики к автоматному виду. Идея алгоритма преобразования следующая – просматриваем поочередно все правила грамматики и при наличии в правой части правила цепочки терминальных символов длины n (n>1) вводится n–1 дополнительный нетерминальный символ. Для каждого из них записываются соответствующие правила, т.е. исходная терминальная цепочка разбивается на символы и разносится по соответсувующим нетерминалам.

В качестве новых нетерминалов выбираются символы, не используемые в грамматике ранее. После преобразования грамматики описание правил каждого нетерминала выводится в отдельном окне. Если преобразование не понадобилось (исходная грамматика принимает автоматный вид) – то в окне просто выведется копия исходной грамматики.



**Этап 3:**

Построение ДКА происходит по следующему алгоритму:

Каждому нетерминальному символу из автоматной грамматики соответствует идентичное состояние в ДКА. Также вводится дополнительное состояние # - оно может использоваться как конечное для праволинейной грамматики, так и как начальное для леволинейной грамматики (при наличии правил вида A->a).

Для праволинейной грамматики целевой символ является начальным состоянием, заключительными являются состояния # (если использовалось) и каждое состояние, у которого в автоматной грамматике имеется правило вида A->lambda.

Для леволинейной грамматики наоборот - заключительным состоянием будет являться целевой символ, а начальными будут состояния # (если использовалось) и каждое состояние, у которого в автоматной грамматике имеется правило вида A->lambda.

Входным алфавитом автомата будет являться алфавит исходной грамматики.

При праволинейной грамматике для правил вида A->aB в функцию перехода δ(A, a) будет добавлено состояние B, для правил вида A->a, δ(A, a) – состояние #.

При леволинейной грамматике для правил вида A->Ba в функцию перехода δ(B, a) будет добавлено состояние A, для правил вида A->a, δ(“целевой символ”, a) – состояние A.

При построении ДКА может возникнуть ситуация, когда автомат на выходе получился недетерминированным. Это определяется следующим образом:

- если в функции переходов есть состояния длиной более 1 символа (можно перейти в несколько состояний);

- если при построении ДКА по леволинейной грамматике, в исходной грамматике оказывается несколько начальных состояний.

В этом случае применяется алгоритм преобразования НКА в ДКА. Его суть состоит в следующем:

1. В исходной функции переходов происходит поиск состояний, которые в ней не присутствуют, и создается новая функции переходов, состоящая из исходной функции и новых состояний, полученных в результате поиска.
2. Для новых состояний, полученных при обходе исходной функции переходов, добавляются переходы, получившиеся в результате объединения всех прежних переходов из каждого состояния qi, из которых состоит новое состояние.
3. Далее происходит поиск новых состояний среди тех, которые появились на шаге 1. Аналогично добавляются переходы для этих состояний.
4. При построении автомата для праволинейной грамматики начальное состояние нового автомата остается тем же (целевой символ), а множество конечных состояний нового автомата будет состоять из всех сочетаний, в которых присутствовало # и состояний, у которых в автоматной грамматике имеется правило вида A->lambda.
5. При построении автомата для леволинейной грамматики начальным состоянием нового автомата будет являться сочетание исходных начальных состояний, а множество конечных состояний нового автомата будет состоять из всех сочетаний, в которых присутствовал целевой символ – конечное состояние исходного автомата.

После построения ДКА из него удаляются все недостижимые состояния. Делается это путем прохода по всем состояниям автомата, начиная с начального (которое всегда является достижимым), и помечанием каждого, в которое можно попасть. Обход выполняется до тех пор, пока количество новых найденных достижимых состояний на текущем обходе не будет равно количеству новых найденных достижимых состояний на предыдущем обходе. После чего в окно таблицы переходов ДКА выводятся все получившиеся достижимые состояния.

**Пример работы программы**

Рассмотрим работу алгоритма для обоих типов грамматик.

**Праволинейная грамматика**:

Пусть G=({01},{SAB},P,S)

P:

S->0A|1B|1101S|

A->001S|100S|010

B->101S|010S|000

**Этап 1:**

Для начала идет проверка правил на корректность – в данном случае все правила соответствуют виду праволинейной грамматики. Подробный разбор правил на корректность будет описан в лог-файле.

Генерация цепочек:

Диапазон длин – от 0 до 5.

Смотрим правила для целевого символа:

S->0A – присутствует нетерминал A – заносим в список несгенерированных.

S->1B – присутствует нетерминал B – заносим в список несгенерированных.

S->1101S - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” – нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

Список несгенерированных:

0A  
1B  
1101S

Список несгенерированных не пуст – просматриваем цепочки, присутствующие в нем:

0A:

A->001S -> 0001S - присутствует нетерминал S – заносим в список несгенерированных.

A->100S -> 0100S - присутствует нетерминал S – заносим в список несгенерированных.

A->010 -> 0010 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

1B:

B->010S -> 1010S - присутствует нетерминал S – заносим в список несгенерированных.

B->101S -> 1101S - присутствует нетерминал S – заносим в список несгенерированных.

B->000 -> 1000 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных

1101S:

S->0A -> 11010A - присутствует нетерминал A – заносим в список несгенерированных.

S->1B -> 11011B - присутствует нетерминал B – заносим в список несгенерированных.

S->1101S – 11011101S - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” – 1101 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

0010

1000

1101

Список несгенерированных:

0001S

0100S

1010S

1101S

11010A

11011B

11011101S

Список несгенерированных не пуст – просматриваем цепочки, присутствующие в нем:

0001S:

S->0A -> 00010A - присутствует нетерминал A – заносим в список несгенерированных.

S->1B -> 00011B - присутствует нетерминал B – заносим в список несгенерированных.

S->1101S –> 00011101S - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” –> 0001 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

0010

1000

1101

0001

В списке сгенерированных цепочек присутствует 5 цепочек – процесс генерации заканчивается.

**Этап 2:**

Просматриваем поочерёдно все правила грамматики.

Для нетерминала S существует 4 правила вывода. В правиле 1101S присутствует цепочка терминальных символов длиной 4. Следовательно, в алфавите новой регулярной грамматики добавится 3 новых нетерминальных символа. Преобразованные правила в итоге примут вид:

S->1101S:

S->1C

C->1D

D->0E

E->1S

Для нетерминала A существует 3 правила вывода. В них присутствуют цепочки терминальных символов длиной 3 каждая, причем правило 010 не содержит нетерминалов. Следовательно, в алфавите новой регулярной грамматики добавятся по 2 новых нетерминальных символа для каждого правила. Преобразованные правила в итоге примут вид:

A->001S: A->100S: A->010

A->0F A->1H A->0J

F->0G H->0I J->1K

G->1S I->0S K->0

Для нетерминала B существует 3 правила вывода. В них присутствуют цепочки терминальных символов длиной 3 каждая, причем правило 000 не содержит нетерминалов. Следовательно, в алфавите новой регулярной грамматики добавятся по 2 новых нетерминальных символа для каждого правила. Преобразованные правила в итоге примут вид:

B->101S: B->010S: B->000

B->1L B->0N B->0P

L->0M N->1O P->0Q

M->1S O->0S Q->0

Таким образом, проделав данные действия над всеми правилами получим правила новой автоматной грамматики:

G’=({012}, {SABCDEFGHIJKLMNOPQ }, P, S)

P’:

S->0A|1B|1C|

A->0F|1H|0J

B->1L|0N|0P

C->1D

D->0E

E->1S

F->0G

G->1S

H->0I

I->0S

J->1K

K->0

L->0M

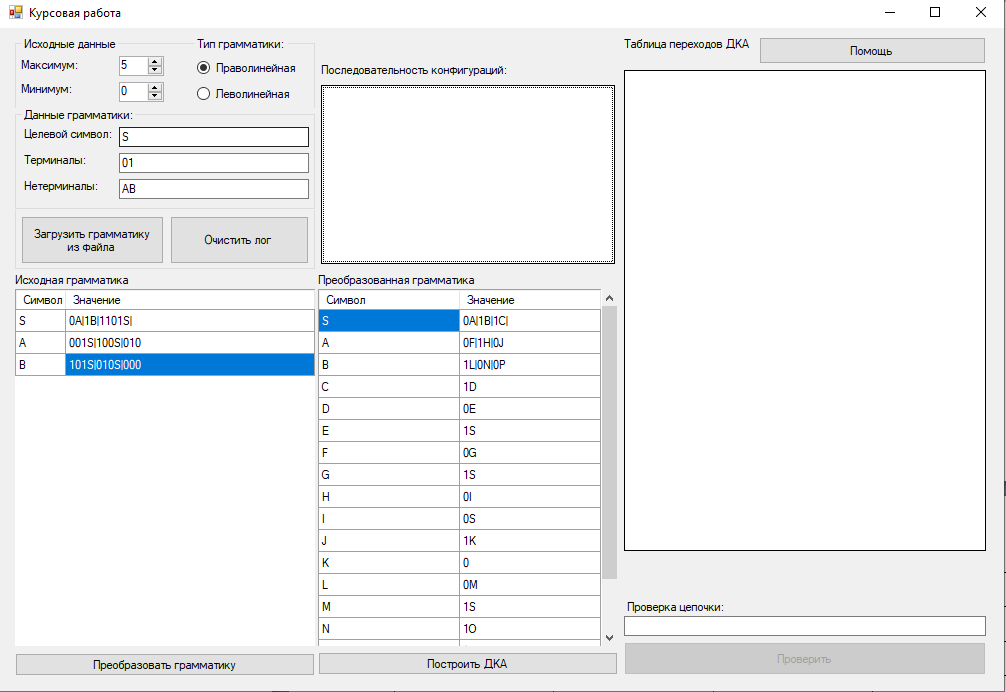
M->1S

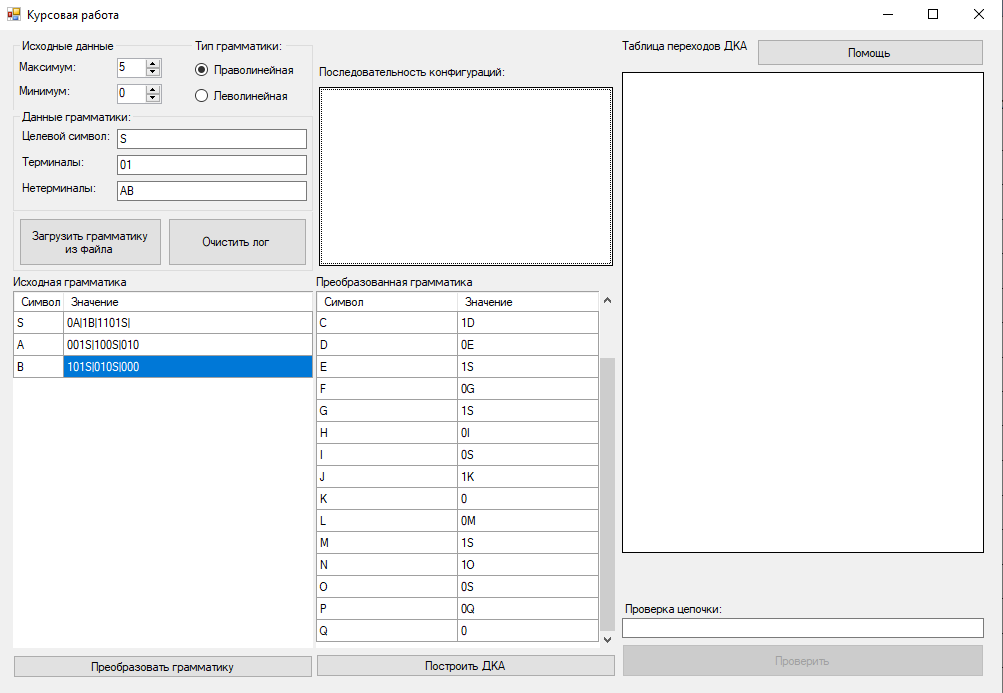
N->1O

O->0S

P->0Q

Q->0





**Этап 3:**

Представим исходную функцию переходов в табличном виде:

|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | A | BC |
| A | FJ | H |
| B | NP | L |
| C |  | D |
| D | E |  |
| E |  | S |
| F | G |  |
| G |  | S |
| H | I |  |
| I | S |  |
| J |  | K |
| K | # |  |
| L | M |  |
| M |  | S |
| N |  | O |
| O | S |  |
| P | Q |  |
| Q | # |  |
| # |  |  |

В функции переходов присутствуют состояния длиной более одного символа (BC, FJ, NP), следовательно автомат недетерминированный, и нужно его преобразовать в ДКА. Создаем новую функцию переходов, в которую переносим исходные состояния и добавляем новые, приписывая им соответствующие переходы. Также смотрим среди новых состояний состояния, которых нет в функции переходов, и аналогично приписываем им переходы:

|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | A | BC |
| A | FJ | H |
| B | NP | L |
| C |  | D |
| D | E |  |
| E |  | S |
| F | G |  |
| G |  | S |
| H | I |  |
| I | S |  |
| J |  | K |
| K | # |  |
| L | M |  |
| M |  | S |
| N |  | O |
| O | S |  |
| P | Q |  |
| Q | # |  |
| # |  |  |
| BC | NP | DL |
| FJ | G | K |
| NP | Q | O |
| DL | EM |  |
| EM |  | S |

После удаления недостижимых состояний таблица переходов примет вид:

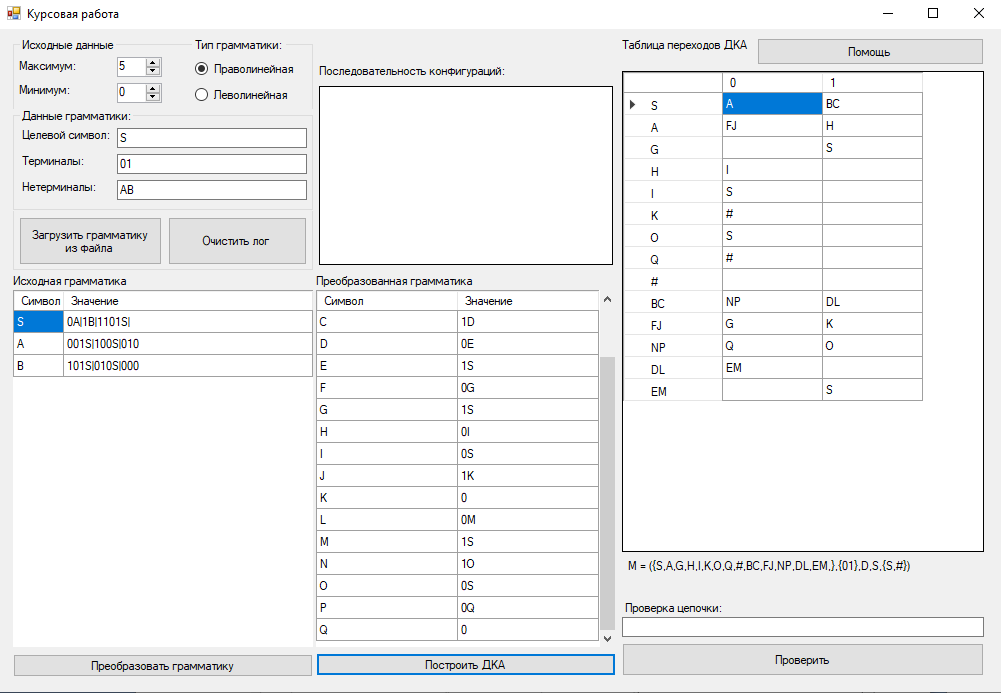
|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | A | BC |
| A | FJ | H |
| G |  | S |
| H | I |  |
| I | S |  |
| K | # |  |
| O | S |  |
| Q | # |  |
| # |  |  |
| BC | NP | DL |
| FJ | G | K |
| NP | Q | O |
| DL | EM |  |
| EM |  | S |

Начальным состоянием является S

Множество конечных состояний – {S, #}.

В итоге полученный детерминированный автомат, эквивалентный исходному НКА, будет иметь вид:

M=({S,A,G,H,I,K,O,Q,#,BC,FJ,NP,DL,EM,},{01},D,S,{S,#})

****

**Леволинейная грамматика**:

Пусть G=({01},{SAB},P,S)

P:

S->A0|B1|S1101|

A->S001|S100|010

B->S101|S010|000

**Этап 1:**

Для начала идет проверка правил на корректность – в данном случае все правила соответствуют виду праволинейной грамматики. Подробный разбор правил на корректность будет описан в лог-файле.

Генерация цепочек:

Диапазон длин – от 0 до 5.

Смотрим правила для целевого символа:

S->A0 – присутствует нетерминал A – заносим в список несгенерированных.

S->B1 – присутствует нетерминал B – заносим в список несгенерированных.

S->S1101 - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” – нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

Список несгенерированных:

A0  
B1  
S1101

Список несгенерированных не пуст – просматриваем цепочки, присутствующие в нем:

A0:

A->S001 -> S0010 - присутствует нетерминал S – заносим в список несгенерированных.

A->S100 -> S1000 - присутствует нетерминал S – заносим в список несгенерированных.

A->010 -> 0100 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

B1:

B->S010 -> S1011 - присутствует нетерминал S – заносим в список несгенерированных.

B->S101 -> S0101 - присутствует нетерминал S – заносим в список несгенерированных.

B->000 -> 0001 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных

S1101:

S->A0 -> A01101 - присутствует нетерминал A – заносим в список несгенерированных.

S->B1 -> B11101 - присутствует нетерминал B – заносим в список несгенерированных.

S->S1101 – S11011101 - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” – 1101 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

0100

0001

1101

Список несгенерированных:

S0010

S1000

S1011

S0101

A01101

B11101

S11011101

Список несгенерированных не пуст – просматриваем цепочки, присутствующие в нем:

S0010:

S->A0 -> A00010 - присутствует нетерминал A – заносим в список несгенерированных.

S->B1 -> B10010 - присутствует нетерминал B – заносим в список несгенерированных.

S->S1101 –> S11010010 - присутствует нетерминал S – заносим в список несгенерированных.

S->” ” –> 0010 - нетерминалы отсутствуют, попадает под диапазон длин – заносим в список сгенерированных.

Список сгенерированных:

0100

0001

1101

0010

В списке сгенерированных цепочек присутствует 5 цепочек – процесс генерации заканчивается.

**Этап 2:**

Просматриваем поочерёдно все правила грамматики.

Для нетерминала S существует 4 правила вывода. В правиле S1101 присутствует цепочка терминальных символов длиной 4. Следовательно, в алфавите новой регулярной грамматики добавится 3 новых нетерминальных символа. Преобразованные правила в итоге примут вид:

S->S1101:

S->C1

C->D0

D->E1

E->S1

Для нетерминала A существует 3 правила вывода. В них присутствуют цепочки терминальных символов длиной 3 каждая, причем правило 010 не содержит нетерминалов. Следовательно, в алфавите новой регулярной грамматики добавятся по 2 новых нетерминальных символа для каждого правила. Преобразованные правила в итоге примут вид:

A->S001: A->S100: A->010

A->F1 A->H0 A->J0

F->G0 H->I0 J->K1

G->S0 I->S1 K->0

Для нетерминала B существует 3 правила вывода. В них присутствуют цепочки терминальных символов длиной 3 каждая, причем правило 000 не содержит нетерминалов. Следовательно, в алфавите новой регулярной грамматики добавятся по 2 новых нетерминальных символа для каждого правила. Преобразованные правила в итоге примут вид:

B->S101: B->010S: B->000

B->L1 B->N0 B->P0

L->M0 N->O1 P->Q0

M->S1 O->S0 Q->0

Таким образом, проделав данные действия над всеми правилами получим правила новой автоматной грамматики:

G’=({012}, {SABCDEFGHIJKLMNOPQ }, P, S)

P’:

S->A0|B1|C1

A->F1|H0|J0

B->L1|N0|P0

C->D0

D->E1

E->S1

F->G0

G->S0

H->I0

I->S1

J->K1

K->0

L->M0

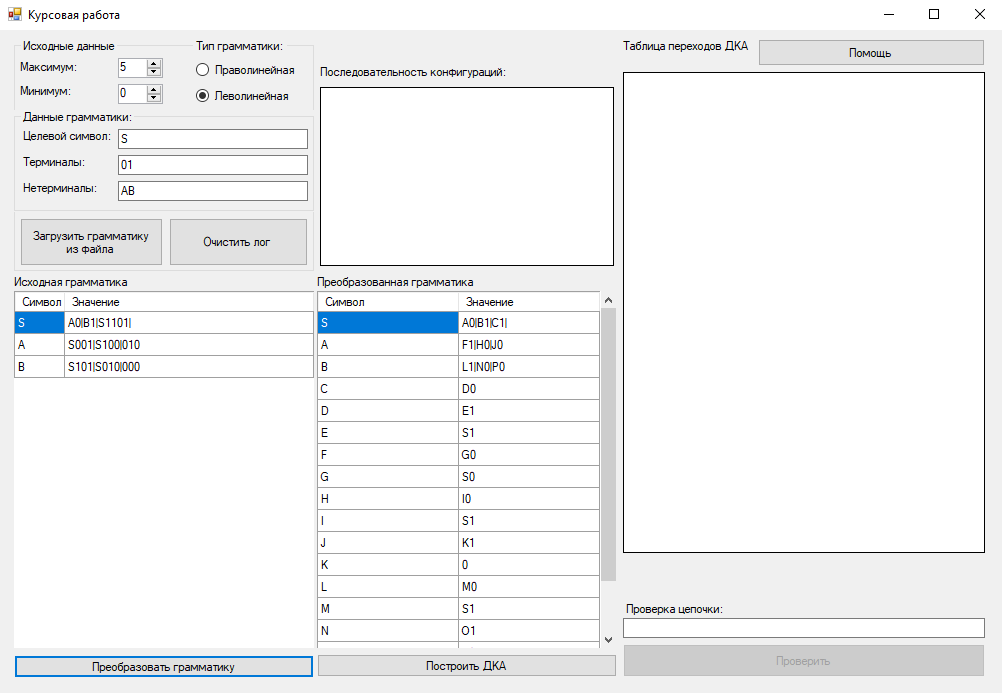
M->S1

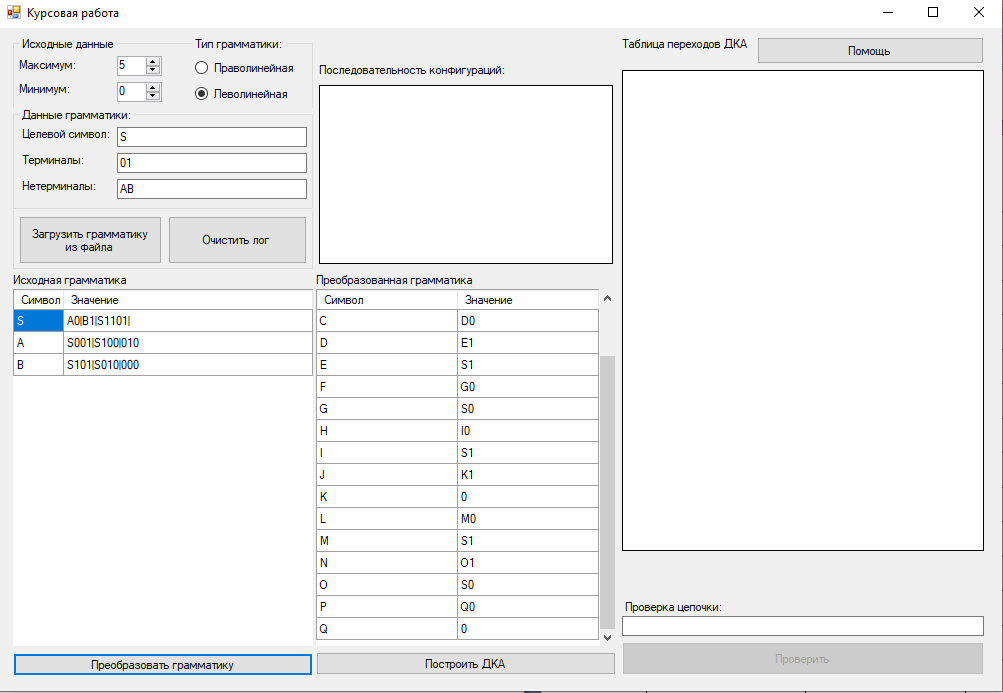
N->O1

O->S0

P->Q0

Q->0





**Этап 3:**

Представим исходную функцию переходов в табличном виде:

|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | GO | EIM |
| A | S |  |
| B |  | S |
| C |  | S |
| D | C |  |
| E |  | D |
| F |  | A |
| G | F |  |
| H | A |  |
| I | H |  |
| J | A |  |
| K |  | J |
| L |  | B |
| M | L |  |
| N | B |  |
| O |  | N |
| P | B |  |
| Q | P |  |
| # | KQ |  |

В функции переходов присутствуют состояния длиной более одного символа (GO, EIM, KQ), следовательно автомат недетерминированный, и нужно его преобразовать в ДКА. Создаем новую функцию переходов, в которую переносим исходные состояния и добавляем новые, приписывая им соответствующие переходы. Также смотрим среди новых состояний состояния, которых нет в функции переходов, и аналогично приписываем им переходы, вдобавок добавляем новое начальное состояние S# (т.к существует правило S->lambda и правила K->0 и Q->0):

|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | GO | EIM |
| A | S |  |
| B |  | S |
| C |  | S |
| D | C |  |
| E |  | D |
| F |  | A |
| G | F |  |
| H | A |  |
| I | H |  |
| J | A |  |
| K |  | J |
| L |  | B |
| M | L |  |
| N | B |  |
| O |  | N |
| P | B |  |
| Q | P |  |
| # | KQ |  |
| GO | F | N |
| EIM | HL | D |
| KQ | P | J |
| S# | GKOQ | EIM |
| HL | A | B |
| GKOQ | FP | JN |
| FP | B | A |
| JN | AB |  |
| AB | S | S |

После удаления недостижимых состояний таблица переходов примет вид:

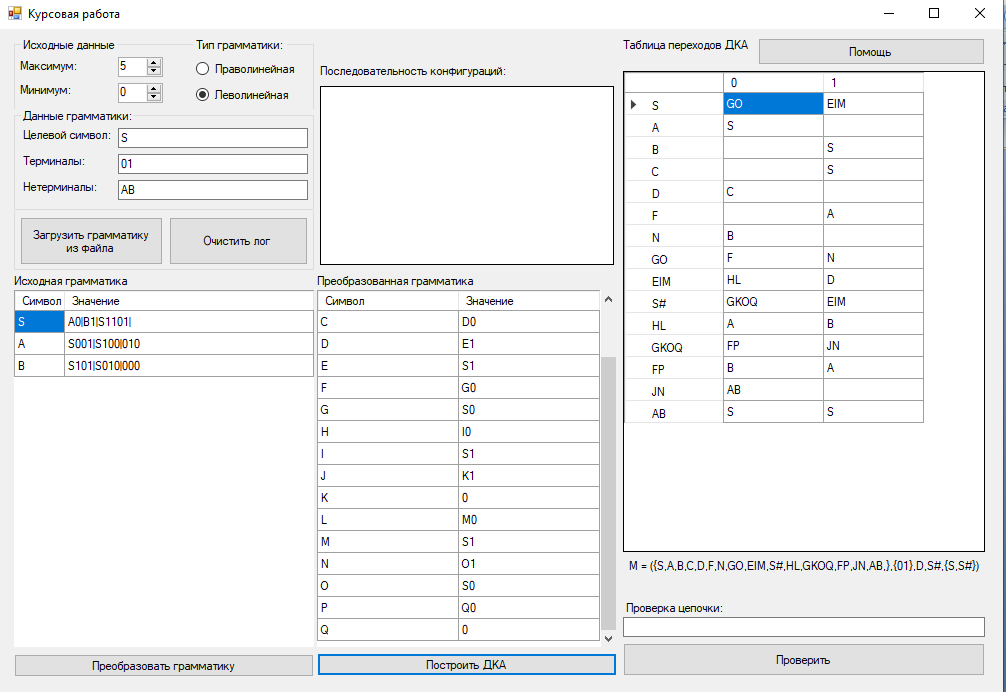
|  |  |  |
| --- | --- | --- |
| Состояние/вход | 0 | 1 |
| S | GO | EIM |
| A | S |  |
| B |  | S |
| C |  | S |
| D | C |  |
| F |  | A |
| G | F |  |
| N | B |  |
| GO | F | N |
| EIM | HL | D |
| S# | GKOQ | EIM |
| HL | A | B |
| EIM | HL | D |
| GKOQ | FP | JN |
| FP | B | A |
| JN | AB |  |
| AB | S | S |

Начальным состоянием является S#

Множество конечных состояний – {S, S#}.

В итоге полученный детерминированный автомат, эквивалентный исходному НКА, будет иметь вид:

M=({S,A,B,C,D,F,N,GO,EIM,S#,HL,GKOQ,FP,JN,AB,},{01},D,S,{S,S#})

****

**Описание основных блоков и кода программы**

**Генерация цепочек –** по нажатию на кнопку “Преобразовать грамматику” сперва генерируются цепочки по исходной грамматике.

Функция GenerateChains генерирует цепочки по исходной грамматике. Сгенерированные цепочки хранятся в списке chains\_list, несгенерированные – в non\_term\_chains. Список sub\_non\_term\_chains хранит новые несгенерированные цепочки, которые по окончании просмотра старого списка несгенерированных цепочек заносятся туда.

Описание правил для нетерминалов находятся в словаре rules\_dictionary.

Функция Is\_valid смотрит на количетсво содержащихся в ней терминалов и нетерминалов, и если вторые отсутствуют – то цепочка заносится в список сгенерированных. Иначе – в список несгенерированных.

public void GenerateChains()

{

string res;

int recursion = 0;

string chain\_prefix;

List<string> non\_term\_chains = new List<string>();

log\_buffer += "Генерируем цепочки:\n";

log\_buffer += "Смотрим правила для целевого символа:\n";

foreach (string rule\_right\_part in rules\_dictionary[start\_non\_term])

{

if (0 == Is\_valid(rule\_right\_part) && rule\_right\_part.Length >= min\_chain\_length)

{

log\_buffer += "Сгенерирована цепочка " + rule\_right\_part + "\n";

chains\_list.Add(rule\_right\_part == "" ? "" : rule\_right\_part);

}

else if (-2 == Is\_valid(rule\_right\_part))

{

log\_buffer += "Цепочка с нетериналами " + rule\_right\_part + " добалвена \n";

non\_term\_chains.Add(rule\_right\_part);

}

}

if (non\_term\_chains.Count != 0)

{

while (recursion <= MAX\_RECURSION)

{

recursion++;

List<string> sub\_non\_term\_chains = new List<string>();

foreach (string non\_term\_chain in non\_term\_chains)

{

log\_buffer += "Смотрим цепочку " + non\_term\_chain + ":\n";

chain\_prefix = "";

for (int i = 0; i < non\_term\_chain.Length; i++)

{

if (!rules\_dictionary.ContainsKey(non\_term\_chain[i].ToString()))

{

chain\_prefix += non\_term\_chain[i];

}

else

{

log\_buffer += "Смотрим правила для нетерминала " + non\_term\_chain[i].ToString() + ":\n";

foreach (string rule\_right\_part in rules\_dictionary[non\_term\_chain[i].ToString()])

{

log\_buffer += "Правило " + rule\_right\_part + ":\n";

res = chain\_prefix + rule\_right\_part + non\_term\_chain.Substring(i + 1);

log\_buffer += "Получившаяся цепочка - " + res + ":\n";

if (0 == Is\_valid(res))

{

if (chains\_list.Contains(res == "" ? "" : res) || res.Length < min\_chain\_length)

{

break;

}

else

{

chains\_list.Add(res == "" ? "" : res);

log\_buffer += "Сгенерирована цепочка " + res + "\n";

if (chains\_list.Count >= 5)

{

break;

}

}

}

else if (-2 == Is\_valid(res))

{

log\_buffer += "Заносим " + res + " в список несгенерированных:\n";

sub\_non\_term\_chains.Add(res);

}

if (chains\_list.Count >= 5)

{

break;

}

}

break;

}

}

if (chains\_list.Count >= 5)

{

break;

}

}

if (chains\_list.Count >= 5)

{

break;

}

non\_term\_chains.Clear();

non\_term\_chains = sub\_non\_term\_chains.Distinct().ToList();

}

log\_buffer += "Процесс генерации закончен. Список сгенерированных цепочек:\n";

foreach (string chain in chains\_list)

{

log\_buffer += chain + "\n";

}

}

else

{

log\_buffer += "Процесс генерации закончен. Список сгенерированных цепочек:\n";

foreach (string chain in chains\_list)

{

log\_buffer += chain + "\n";

}

}

Write\_to\_log();

}

private int Is\_valid(string line)

{

int term\_sym = 0;

int non\_term\_sym = 0;

foreach (char ch in line)

{

if (!rules\_dictionary.ContainsKey(ch.ToString()))

{

term\_sym++;

}

else

{

non\_term\_sym++;

}

}

if (term\_sym > max\_chain\_length)

{

return -1;

}

if ((term\_sym + non\_term\_sym - 5) > max\_chain\_length)

{

return -1;

}

return (non\_term\_sym > 0) ? -2 : 0;

}

**Проверка грамматики на корректность** – далее идет проверка корректности грамматики

Функция Check\_regular\_grammar проходит по каждому правилу исходной грамматики, лежащему в словаре rules\_dictionary, и проверяет корректность тех или иных правил в зависимости от типа грамматики.

Функция Check\_non\_term проверяет принадлежность символа списку нетерминалов.

private bool Check\_non\_term(string key)

{

if (key == start\_non\_term\_text\_box.Text[0].ToString())

{

return true;

}

for (int j = 0; j < non\_term\_text\_box.Text.Length; j++)

{

if (key == non\_term\_text\_box.Text[j].ToString())

{

return true;

}

}

return false;

}

private bool Check\_regular\_grammar()

{

string key;

bool is\_non\_term;

bool is\_first\_term;

string type\_grammar = regular\_side == 0 ? "праволинейная" : "леволинейная";

generic.log\_buffer += "Проверка грамматики на корректность:\nТип грамматики - " + type\_grammar + "\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Проверяем правила для нетерминала " + key + ":\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

if (is\_key.Length == 1)

{

if(Check\_non\_term(is\_key[0].ToString()))

{

generic.log\_buffer += "ОШИБКА! Правило состоит из одного нетерминала\n\n";

return false;

}

}

if (is\_key.Length >= 2)

{

generic.log\_buffer += "Смотрим символ " + is\_key[0].ToString() + ":\n";

if (Check\_non\_term(is\_key[0].ToString()))

{

generic.log\_buffer += "Символ " + is\_key[0].ToString() + " - нетерминальный\n";

is\_non\_term = true;

is\_first\_term = false;

if (regular\_side == 0)

{

generic.log\_buffer += "ОШИБКА! Правило не имеет вид праволинейной грамматики\n\n";

return false;

}

}

else

{

generic.log\_buffer += "Символ " + is\_key[0].ToString() + " - терминальный\n";

is\_non\_term = false;

is\_first\_term = true;

}

for (int j = 1; j < is\_key.Length; j++)

{

string help = is\_key[j].ToString();

generic.log\_buffer += "Смотрим символ " + help + ":\n";

if (Check\_non\_term(help))

{

if(is\_first\_term == true && regular\_side == 1)

{

generic.log\_buffer += "ОШИБКА! Правило не имеет вид леволинейной грамматики\n\n";

return false;

}

if (is\_non\_term == true)

{

generic.log\_buffer += "ОШИБКА! В правиле содержится больше одного нетерминала\n\n";

return false;

}

is\_non\_term = true;

generic.log\_buffer += "Символ " + help + " - нетерминальный\n";

}

else

{

if (is\_non\_term == true && is\_first\_term == true)

{

generic.log\_buffer += "ОШИБКА! Правило имеет некорректный вид\n\n";

return false;

}

generic.log\_buffer += "Символ " + help + " - терминальный\n";

}

}

}

is\_non\_term = false;

}

}

generic.log\_buffer += "Проверка завершена успешно!\n\n";

return true;

}

**Преобразование грамматик** – в случае, если исходная грамматика соответствует виду регулярной грамматики, то наступает этап преобразования грамматики.

Функция Transform\_grammar преобразует праволинейную грамматику, Transform\_left\_grammar – леволинейную.

В словарь transform\_dictionary заносится преобразованная грамматика, а словарь buf\_transform\_dictionary хранит в себе промежуточные преобразования.

Функция Add\_new\_key добавляет новый нетерминал в грамматику.

Функция Get\_grammar\_description выводит описание получившейся грамматики.

private string Add\_new\_key()

{

string key = "";

for (int i = 0; i < 26; i++)

{

if (!generic.rules\_dictionary.ContainsKey(non\_terms[i].ToString())

&& !buf\_transform\_dictionary.ContainsKey(non\_terms[i].ToString()))

{

key = non\_terms[i].ToString();

break;

}

}

return key;

}

private bool Transform\_grammar()

{

int length;

int new\_nonterms;

string key;

string buf\_rule;

string result\_rules = "";

bool is\_last\_term = false;

DataGridViewRow row;

final\_states = "";

transform\_dictionary.Clear();

buf\_transform\_dictionary.Clear();

fix\_grammar\_data\_grid\_view.Rows.Clear();

generic.log\_buffer += "Преобразование грамматики:\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Преобразуем правила нетерминала " + key + ":\n\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

length = is\_key.Length - 1;

if (is\_key == "")

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

final\_states += key + ",";

result\_rules += "|";

continue;

}

if (is\_key.Length == 2 && (Check\_non\_term(is\_key[length].ToString())))

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += is\_key + "|";

continue;

}

if (is\_key.Length >= 2)

{

if (Check\_non\_term(is\_key[length].ToString()))

{

new\_nonterms = length - 1;

is\_last\_term = true;

}

else

{

new\_nonterms = length;

}

if (!Check\_non\_term(is\_key[length].ToString()) && is\_key.Length == 2)

{

new\_nonterms = 1;

}

generic.log\_buffer += "Для преобразования нужно " + new\_nonterms + " новых нетерминалов\n\n";

generic.log\_buffer += "Преобразуем:\n";

string new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + new\_key + "\n";

result\_rules += is\_key[0].ToString() + new\_key + "|";

generic.log\_buffer += "Новое правило для нетерминала " + key + " - " + is\_key[0].ToString() + new\_key + "\n";

buf\_rule = is\_key.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

while (new\_nonterms >= 0)

{

string buf\_new\_key;

if (is\_last\_term == true)

{

while (buf\_rule.Length > 2)

{

string new\_rule = buf\_rule[0].ToString() + new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_rule[0].ToString() + buf\_new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

else

{

while (buf\_rule.Length > 1)

{

string new\_rule = buf\_rule[0].ToString() + new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_rule[0].ToString() + buf\_new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

}

is\_last\_term = false;

}

else

{

result\_rules += is\_key + "|";

}

}

result\_rules = result\_rules.Remove(result\_rules.Length - 1);

generic.log\_buffer += "Преобразованные правила: " + result\_rules + "\n\n";

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, key, result\_rules);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[key] = result\_rules.Split('|');

result\_rules = "";

}

for (int i = 0; i < buf\_transform\_dictionary.Keys.Count; i++)

{

string buffer\_key = buf\_transform\_dictionary.Keys.ElementAt(i);

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, buffer\_key, buf\_transform\_dictionary[buffer\_key]);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[buffer\_key] = buf\_transform\_dictionary[buffer\_key].Split('|'); ;

}

generic.log\_buffer += "Преобразованная грамматика:\n";

Get\_grammar\_description(transform\_dictionary);

generic.Write\_to\_log();

return true;

}

private bool Transform\_left\_grammar()

{

int length;

int new\_nonterms;

string key;

string buf\_rule;

string result\_rules = "";

bool is\_last\_term = false;

DataGridViewRow row;

final\_states = "";

transform\_dictionary.Clear();

buf\_transform\_dictionary.Clear();

fix\_grammar\_data\_grid\_view.Rows.Clear();

generic.log\_buffer += "Преобразование грамматики:\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Преобразуем правила нетерминала " + key + ":\n\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

length = is\_key.Length - 1;

if (is\_key == "")

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += "|";

begin\_left\_state += key;

continue;

}

if (is\_key.Length == 2 && (Check\_non\_term(is\_key[0].ToString())))

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += is\_key + "|";

continue;

}

if (is\_key.Length >= 2)

{

if (Check\_non\_term(is\_key[0].ToString()))

{

new\_nonterms = length - 1;

is\_last\_term = true;

}

else

{

new\_nonterms = length + 1;

}

if (!Check\_non\_term(is\_key[0].ToString()) && is\_key.Length == 2)

{

new\_nonterms = 2;

}

generic.log\_buffer += "Для преобразования нужно " + new\_nonterms + " новых нетерминалов\n\n";

generic.log\_buffer += "Преобразуем:\n";

string new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + new\_key + "\n";

result\_rules += new\_key + is\_key[length].ToString() + "|";

generic.log\_buffer += "Новое правило для нетерминала " + key + " - " + new\_key + is\_key[length].ToString() + "\n";

buf\_rule = is\_key.Remove(length, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

while (new\_nonterms >= 0)

{

string buf\_new\_key;

if (is\_last\_term == true)

{

while (buf\_rule.Length > 2)

{

string new\_rule = new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(buf\_rule.Length - 1, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

else

{

while (buf\_rule.Length > 1)

{

string new\_rule = new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(buf\_rule.Length - 1, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

}

is\_last\_term = false;

}

else

{

result\_rules += is\_key + "|";

}

}

result\_rules = result\_rules.Remove(result\_rules.Length - 1);

generic.log\_buffer += "Преобразованные правила: " + result\_rules + "\n\n";

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, key, result\_rules);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[key] = result\_rules.Split('|');

result\_rules = "";

}

for (int i = 0; i < buf\_transform\_dictionary.Keys.Count; i++)

{

string buffer\_key = buf\_transform\_dictionary.Keys.ElementAt(i);

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, buffer\_key, buf\_transform\_dictionary[buffer\_key]);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[buffer\_key] = buf\_transform\_dictionary[buffer\_key].Split('|'); ;

}

generic.log\_buffer += "Преобразованная грамматика:\n";

Get\_grammar\_description(transform\_dictionary);

generic.Write\_to\_log();

return true;

}

private void Get\_grammar\_description(Dictionary<string, string[]> dict)

{

string key;

string rules = "";

string non\_terms = "";

string alphabet = "";

for(int i = 0; i < term\_text\_box.Text.Length; i++)

{

alphabet += term\_text\_box.Text[i].ToString();

}

for (int i = 0; i < dict.Keys.Count; i++)

{

key = dict.Keys.ElementAt(i);

non\_terms += key;

rules += key + "->";

foreach (string is\_key in dict[key])

{

rules += is\_key + "|";

}

rules = rules.Remove(rules.Length - 1);

rules += "\n";

}

generic.log\_buffer += "G=({" + alphabet + "},{" + non\_terms + "},P," + start\_non\_term\_text\_box.Text[0].ToString() + "):\n";

generic.log\_buffer += rules + "\n";

}

**Построение ДКА** – по нажатии на кнопку “Построить ДКА” запускается построение автомата.

В словарь states\_dictionary заносятся состояния автомата, являющееся ключом, и индекс строки в таблице переходов, являющееся значением.

Словарь symbols\_dictionary хранит символы алфавита (ключ) и индексы столбцов в таблице переходов (значение).

Двумерный массив transitions – это функция переходов исходного автомата размерности m\*n, где:

m – исходное количество состояний (включая #).

n – количество символов алфавита

В случае, когда требуется преобразовать исходный автомат создается новая таблица – new\_transitions, размерности 10000\*n где:

n – количество символов алфавита

В зависимости от типа грамматики вызывается соответствующая функция, которая строит первоначальный автомат:

Build\_right\_side\_grammar\_to\_DKA – для праволинейной грамматики

Build\_left\_side\_grammar\_to\_DKA – для леволинейной грамматики

Функция Rearrange\_in\_order преобразует значения состояний в таблице переходов и ключи в словаре states\_dictionary в отсортированные по алфавиту строки. При этом соблюдается условие, чтобы целевой символ начинал эту последовательность, а # - заканчивал.

Если при построении автомата получился НКА, то функция Transform\_to\_DKA преобраузет получившийся НКА в ДКА. Создается новая таблица переходов, добавляются новые состояния (функция Add\_new\_states) для автомата и исключаются недостижимые состояния (функция Except\_non\_entry\_states). То же самое делается в случае, когда в леволинейной грамматике присутствуют несколько начальных состояний.

Функция Get\_DKA\_description выводит описание получившегося автомата

private void Build\_left\_side\_grammar\_to\_DKA()

{

generic.log\_buffer += "Добавляем целевой символ в конечное состояние\n";

final\_states += start\_non\_term\_text\_box.Text[0].ToString() + ",";

for (int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

foreach (string is\_key in transform\_dictionary[key])

{

if (is\_key.Length == 1 && !Check\_non\_term(is\_key) && is\_key != "")

{

//log\_buffer += "Добавляем переход для дополнительного состояния #:\n";

begin\_left\_state += "#";

transitions[transform\_dictionary.Keys.Count][symbols\_dictionary[is\_key[0].ToString()]] += key;

}

else if (is\_key != "")

{

//log\_buffer += "Добавляем переход для состояния " + is\_key[0].ToString() + "\n";

transitions[states\_dictionary[is\_key[0].ToString()]][symbols\_dictionary[is\_key[1].ToString()]] += key;

}

}

}

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

for (int j = 0; j < transitions[i].Length; j++)

{

if (transitions[i][j] != "")

{

transitions[i][j] = Rearrange\_in\_order(transitions[i][j]);

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

else

{

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

}

}

}

private void Build\_right\_side\_grammar\_to\_DKA()

{

string key;

string res = "";

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

if (i != transform\_dictionary.Keys.Count)

{

key = transform\_dictionary.Keys.ElementAt(i);

foreach (string is\_key in transform\_dictionary[key])

{

if(is\_key != "")

{

if (is\_key[0].ToString() == term\_text\_box.Text[j].ToString() && is\_key.Length == 2)

{

res += is\_key[1].ToString();

}

else if (is\_key[0].ToString() == term\_text\_box.Text[j].ToString() && is\_key.Length == 1)

{

res += "#";

final\_states += "#" + ",";

}

}

}

transitions[i][j] = res;

transitions[i][j] = Rearrange\_in\_order(transitions[i][j]);

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

res = "";

}

else

{

transitions[i][j] = "";

generic.log\_buffer += "Новый переход: " + "#" + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

}

}

}

private void Except\_non\_entry\_states(string[][] trans)

{

int current\_count;

int is\_new\_state = 1;

Dictionary<string, int> buf\_states\_dictionary = new Dictionary<string, int>();

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

buf\_states\_dictionary[key] = 0;

}

generic.log\_buffer += "Начинаем смотреть с состояния ";

if (regular\_side == 1)

{

generic.log\_buffer += begin\_left\_state + "\n";

buf\_states\_dictionary[begin\_left\_state] = 1;

}

else

{

generic.log\_buffer += start\_non\_term\_text\_box.Text[0].ToString() + "\n";

buf\_states\_dictionary[start\_non\_term\_text\_box.Text[0].ToString()] = 1;

}

generic.Write\_to\_log();

current\_count = is\_new\_state;

generic.log\_buffer += "Осуществляем проход по состояниям:\n";

for (; ; )

{

generic.log\_buffer += "Следующая итерация:\n";

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

if (buf\_states\_dictionary[key] == 0)

{

generic.log\_buffer += "Состояние " + key + " пока не является достижимым\n";

continue;

}

for (int j = 0; j < term\_text\_box.Text.Length; j++)

{

is\_new\_state++;

buf\_states\_dictionary[trans[i][j]] = 1;

if (trans[i][j] != "")

{

generic.log\_buffer += "Состояние " + trans[i][j] + " - достижимое\n";

}

}

}

generic.log\_buffer += "Обнаружено " + is\_new\_state + " достижимых состояний\n";

if (current\_count == is\_new\_state)

{

break;

}

current\_count = is\_new\_state;

is\_new\_state = 0;

generic.Write\_to\_log();

}

generic.log\_buffer += "Список достижимых состояний:\n";

for (int i = 0; i < buf\_states\_dictionary.Keys.Count; i++)

{

string key = buf\_states\_dictionary.Keys.ElementAt(i);

if (buf\_states\_dictionary[key] == 1)

{

result\_dictionary[key] = i;

generic.log\_buffer += key + "\n";

}

}

generic.Write\_to\_log();

}

private void Add\_new\_final\_states(Dictionary<string, int> result\_dictionary)

{

string[] buf = final\_states.Split(',');

for (int i = 0; i < result\_dictionary.Keys.Count; i++)

{

string key = result\_dictionary.Keys.ElementAt(i);

if (key != "")

{

for (int j = 0; j < buf.Length; j++)

{

for (int k = 0; k < key.Length; k++)

{

if (buf[j].ToString() == key[k].ToString())

{

final\_states += key + ",";

}

}

}

}

}

generic.log\_buffer += "Список конечных состояний - " + final\_states + "\n";

}

private void Dka\_button\_Click(object sender, EventArgs e)

{

need\_to\_transform = false;

Init\_DKA\_dictionary();

generic.log\_buffer += "Построение ДКА:\n\nДобавляем исходные состояния, " +

"соответствующие нетерминалам грамматики а также дополнительное состояние №:\n";

for (int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += key + " " + i + "\n";

states\_dictionary[key] = i;

}

states\_dictionary["#"] = transform\_dictionary.Keys.Count;

generic.log\_buffer += "#" + " " + transform\_dictionary.Keys.Count + "\n";

generic.Write\_to\_log();

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

symbols\_dictionary[term\_text\_box.Text[i].ToString()] = i;

}

transitions = new string[transform\_dictionary.Keys.Count + 1][];

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for(int j = 0; j < transitions[i].Length; j++)

{

transitions[i][j] = "";

}

}

if (regular\_side == 1)

{

generic.log\_buffer += "Создаем автомат по леволинейной грамматике:\n";

Build\_left\_side\_grammar\_to\_DKA();

}

else

{

generic.log\_buffer += "Создаем автомат по праволинейной грамматике:\n";

Build\_right\_side\_grammar\_to\_DKA();

}

generic.log\_buffer += "Построение первоначального автомата закончено\n";

generic.Write\_to\_log();

generic.log\_buffer += "Первоначальный вид автомата:\n";

Get\_DKA\_description(states\_dictionary, transitions);

generic.Write\_to\_log();

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

if (need\_to\_transform == true)

{

break;

}

for (int j = 0; j < transitions[i].Length; j++)

{

if (need\_to\_transform == true)

{

break;

}

if (transitions[i][j].Length >= 2)

{

generic.log\_buffer += "Автомат недетерминированный - нужно преобразовать\n";

generic.Write\_to\_log();

need\_to\_transform = true;

Transform\_to\_DKA();

break;

}

}

}

if(!need\_to\_transform)

{

generic.log\_buffer += "Автомат детерминированный, но...\n";

if (regular\_side == 1)

{

generic.log\_buffer += "Преобразуем множество начальных состояний для ДКА по леволинейной грамматике\n";

new\_transitions = new string[10000][];

for (int i = 0; i < 10000; i++)

{

new\_transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

new\_transitions[i][j] = transitions[i][j];

}

}

if (!Add\_new\_states())

{

return;

}

need\_to\_transform = true;

generic.log\_buffer += "Исключаем недостижимые состояния в ДКА\n";

Except\_non\_entry\_states(new\_transitions);

generic.log\_buffer += "Приводим в порядок список конечных состояний\n";

Add\_new\_final\_states(result\_dictionary);

Print\_states\_to\_user(new\_transitions);

generic.Write\_to\_log();

}

else

{

generic.log\_buffer += "Исключаем недостижимые состояния в новом ДКА\n";

Except\_non\_entry\_states(transitions);

generic.log\_buffer += "Приводим в порядок список конечных состояний\n";

Add\_new\_final\_states(result\_dictionary);

Print\_states\_to\_user(transitions);

generic.Write\_to\_log();

}

generic.log\_buffer += "Итоговый автомат:\n";

if (need\_to\_transform == true)

{

Get\_DKA\_description(states\_dictionary, new\_transitions);

}

else

{

Get\_DKA\_description(states\_dictionary, transitions);

}

Print\_ask\_window();

generic.Write\_to\_log();

check\_button\_text\_box.Enabled = true;

}

}

private bool Transform\_to\_DKA()

{

new\_transitions = new string[10000][];

for (int i = 0; i < 10000; i++)

{

new\_transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

new\_transitions[i][j] = transitions[i][j];

}

}

generic.log\_buffer += "Добавляем новые состояния в автомат:\n";

if (!Add\_new\_states())

{

return false;

}

generic.Write\_to\_log();

generic.log\_buffer += "Исключаем недостижимые состояния:\n";

Except\_non\_entry\_states(new\_transitions);

generic.Write\_to\_log();

generic.log\_buffer += "Приводим в порядок конечные состояния:\n";

Add\_new\_final\_states(result\_dictionary);

generic.Write\_to\_log();

Print\_states\_to\_user(new\_transitions);

generic.log\_buffer += "\nПреобразованный автомат:\n";

Get\_DKA\_description(result\_dictionary, new\_transitions);

generic.Write\_to\_log();

Print\_ask\_window();

check\_button\_text\_box.Enabled = true;

return true;

}

private bool Add\_new\_states()

{

int counter = 1;

string old\_state;

string result = "";

generic.log\_buffer += "Ищем новые состояния среди изначальных:\n";

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < new\_transitions[i].Length; j++)

{

if (!states\_dictionary.ContainsKey(new\_transitions[i][j]) && new\_transitions[i][j] != "")

{

old\_state = new\_transitions[i][j];

generic.log\_buffer += "Новое состояние - " + old\_state + "\n";

states\_dictionary[old\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < new\_transitions[i].Length; h++)

{

for (int k = 0; k < old\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[old\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[old\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + old\_state + "->" + new\_transitions[states\_dictionary[old\_state]][h] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

result = "";

}

counter++;

generic.Write\_to\_log();

}

}

}

int is\_new\_state;

int old\_states\_crt = counter - 1;

bool is\_entered\_in\_previous = false;

generic.log\_buffer += "Было добавлено " + old\_states\_crt + "новых состояний\n";

int clue\_number = states\_dictionary.Keys.Count - old\_states\_crt;

generic.Write\_to\_log();

if (regular\_side == 1)

{

generic.log\_buffer += "Смотрим множество начальных состояний:\n";

if (begin\_left\_state == "")

{

dka\_label.Text = "";

generic.log\_buffer += "ОШИБКА! В исходной грамматике не обнаружено правил выхода из рекурсии!\n";

MessageBox.Show("Ошибка построения автомата");

dka\_data\_grid\_view.Rows.Clear();

dka\_data\_grid\_view.Columns.Clear();

check\_button\_text\_box.Enabled = false;

generic.Write\_to\_log();

return false;

}

begin\_left\_state = Rearrange\_in\_order(begin\_left\_state);

generic.log\_buffer += "Начальное состояние для ДКА - " + begin\_left\_state + "\n";

for(int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

if (key == begin\_left\_state)

{

is\_entered\_in\_previous = true;

break;

}

}

if (is\_entered\_in\_previous == false && begin\_left\_state != "#")

{

generic.log\_buffer += "Начальное состояние отсутствует в списке исходных\n" +

"Добавляем новое начальное состояние " + begin\_left\_state + "\n";

states\_dictionary[begin\_left\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < term\_text\_box.Text.Length; h++)

{

for (int k = 0; k < begin\_left\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[begin\_left\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[begin\_left\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + begin\_left\_state + "->" + new\_transitions[states\_dictionary[begin\_left\_state]][h] + "(" + term\_text\_box.Text[h].ToString() + ")\n";

result = "";

}

counter++;

is\_new\_state = old\_states\_crt + 1;

}

else

{

is\_new\_state = old\_states\_crt;

}

generic.Write\_to\_log();

}

else

{

if (final\_states == "")

{

dka\_label.Text = "";

generic.log\_buffer += "ОШИБКА! В исходной грамматике не обнаружено правил выхода из рекурсии!\n";

MessageBox.Show("Ошибка построения автомата");

dka\_data\_grid\_view.Rows.Clear();

dka\_data\_grid\_view.Columns.Clear();

check\_button\_text\_box.Enabled = false;

generic.Write\_to\_log();

return false;

}

is\_new\_state = old\_states\_crt;

}

generic.log\_buffer += "Просматриваем новые состояния и ищем новые среди них:\n";

while (is\_new\_state != 0)

{

generic.log\_buffer += "Осталось посмотреть " + is\_new\_state + " состояний:\n";

string key = states\_dictionary.Keys.ElementAt(clue\_number);

for (int j = 0; j < term\_text\_box.Text.Length; j++)

{

if (!states\_dictionary.ContainsKey(new\_transitions[states\_dictionary[key]][j])

&& new\_transitions[states\_dictionary[key]][j] != "")

{

is\_new\_state++;

old\_state = new\_transitions[states\_dictionary[key]][j];

generic.log\_buffer += "Новое состояние - " + old\_state + "\n";

states\_dictionary[old\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < term\_text\_box.Text.Length; h++)

{

for (int k = 0; k < old\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[old\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[old\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + old\_state + "->" + new\_transitions[states\_dictionary[old\_state]][h] + "(" + term\_text\_box.Text[h].ToString() + ")\n";

result = "";

}

counter++;

}

}

clue\_number++;

is\_new\_state--;

generic.Write\_to\_log();

}

generic.log\_buffer += "Добавление состояний окончено\n";

generic.Write\_to\_log();

return true;

}

private string Rearrange\_in\_order(string string\_to\_order)

{

string\_to\_order = new string(string\_to\_order.ToCharArray().Distinct().ToArray());

string\_to\_order = string.Concat(string\_to\_order.OrderBy(x => x).ToArray());

if (string\_to\_order.Contains("#"))

{

string\_to\_order = string\_to\_order.Replace("#", "");

string\_to\_order += "#";

}

if (string\_to\_order.Contains(start\_non\_term\_text\_box.Text[0].ToString()))

{

string\_to\_order = string\_to\_order.Replace(start\_non\_term\_text\_box.Text[0].ToString(), "");

string\_to\_order = start\_non\_term\_text\_box.Text[0].ToString() + string\_to\_order;

}

return string\_to\_order;

}

private void Get\_DKA\_description(Dictionary<string, int> dict, string[][] trans)

{

string label;

string states = "";

string alphabet = "";

string buf\_final\_states = final\_states;

string begin\_state = regular\_side == 0 ? start\_non\_term\_text\_box.Text[0].ToString() : begin\_left\_state;

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

alphabet += term\_text\_box.Text[i].ToString();

}

for (int i = 0; i < dict.Keys.Count; i++)

{

states += dict.Keys.ElementAt(i) + ",";

}

if (states != "" && states[states.Length - 1].ToString() == ",")

{

states = states.Remove(states.Length - 1, 1);

}

if(buf\_final\_states != "" && buf\_final\_states[buf\_final\_states.Length - 1].ToString() == ",")

{

buf\_final\_states = buf\_final\_states.Remove(buf\_final\_states.Length - 1, 1);

}

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

Add\_spaces(15);

generic.log\_buffer += term\_text\_box.Text[i].ToString();

}

generic.log\_buffer += "\n";

for (int i = 0; i < dict.Keys.Count; i++)

{

string key = dict.Keys.ElementAt(i);

if(key != "")

{

generic.log\_buffer += key;

Add\_spaces(15 - key.Length);

for (int j = 0; j < trans[i].Length; j++)

{

generic.log\_buffer += trans[dict[key]][j];

Add\_spaces(15 - trans[dict[key]][j].Length);

}

generic.log\_buffer += "\n";

}

}

generic.log\_buffer += "M=({" + states + "},{" + alphabet + "},D," + start\_non\_term\_text\_box.Text[0].ToString() + "," + "{" + buf\_final\_states + "}):\n";

dka\_label.Text = "";

label = "M = ({" + states + "},{" + alphabet + "},D," + begin\_state + "," + "{" + buf\_final\_states + "})";

if(label.Length > 80)

{

dka\_label.Text = "Слишком длинное описание автомата. Смотрите лог";

}

else

{

dka\_label.Text = label;

}

}

private void Add\_spaces(int count)

{

for (int j = 0; j < count; j++)

{

generic.log\_buffer += " ";

}

}

**Проверка цепочек по автомату** – по нажатии на кнопку “Проверить” начинается процесс распознавания цепочки

Функция check\_dka проводит распознавание. Работа автомата продолжается до тех пор, пока на его вход поступают символы входной цепочки, автомат не придёт в конфигурацию, переход из которой не определён, или пока на вход не попадется неизвестный символ. Если по окончании распознавания автомат не находится в одном из конечных состояний – выводится оповещение.

Функция Transict\_function получает новое состояние, в которое переходит автомат при распознавании. Если переход в другое состояние не определен – выводится оповещение и процесс распознавания прекращается.

private void Check\_dka(string checkable)

{

is\_stop = false;

bool is\_final = false;

string new\_state;

string current\_state;

dka\_history\_data\_grid\_view.Rows.Clear();

List<string> dka\_history = new List<string>();

if (regular\_side == 0)

{

current\_state = start\_non\_term\_text\_box.Text.ToString();

}

else

{

current\_state = begin\_left\_state;

}

generic.log\_buffer += "Проверяем цепочку " + checkable + ":\n";

generic.log\_buffer += "Начальное состояние - " + current\_state + "\n";

dka\_history.Add("(" + current\_state + "," + checkable + ")");

generic.log\_buffer += "(" + current\_state + "," + checkable + ")\n";

for (int i = 0; i < checkable.Length; i++)

{

if (!symbols\_dictionary.ContainsKey(checkable[i].ToString()))

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "В цепочке указано неизвестное значение: '" + checkable[i].ToString() + "'!";

generic.log\_buffer += "В цепочке указано неизвестное значение: '" + checkable[i].ToString() + "'!\n\n";

is\_stop = true;

break;

}

new\_state = Transict\_function(checkable[i].ToString(), current\_state);

if (is\_stop == true)

{

break;

}

dka\_history.Add("(" + new\_state + "," + checkable.Substring(i + 1) + ")");

generic.log\_buffer += "(" + new\_state + "," + checkable.Substring(i + 1) + ")\n";

current\_state = new\_state;

}

if(is\_stop == false)

{

string[] final = final\_states.Split(',');

for (int i = 0; i < final.Length; i++)

{

if (current\_state == final[i].ToString())

{

is\_final = true;

info\_label.ForeColor = Color.Green;

info\_label.Text = "Разпознавание прошло успешно!";

generic.log\_buffer += "Разпознавание прошло успешно!\n\n";

}

}

if (is\_final == false)

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "Автомат не перешел в конечное состояние!";

generic.log\_buffer += "Автомат не перешел в конечное состояние!\n\n";

}

}

foreach (string str in dka\_history)

{

DataGridViewRow row = new DataGridViewRow();

row.CreateCells(dka\_history\_data\_grid\_view, str);

dka\_history\_data\_grid\_view.Rows.Add(row);

}

}

private void Check\_button\_text\_box\_Click(object sender, EventArgs e)

{

Check\_dka(check\_text\_box.Text.ToString());

generic.Write\_to\_log();

}

private string Transict\_function(string symbol, string current\_state)

{

string new\_state;

if (need\_to\_transform == true)

{

new\_state = new\_transitions[states\_dictionary[current\_state]][symbols\_dictionary[symbol]];

generic.log\_buffer += "Новое состояние - " + new\_state + "\n";

}

else

{

new\_state = transitions[states\_dictionary[current\_state]][symbols\_dictionary[symbol]];

generic.log\_buffer += "Новое состояние - " + new\_state + "\n";

}

if (new\_state == "")

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "Нет состояния перехода для: '" + symbol + "' и исходного состояния: '" + current\_state + "'!";

generic.log\_buffer += "Нет состояния перехода для: '" + symbol + "' и исходного состояния: '" + current\_state + "'!\n\n";

is\_stop = true;

}

return new\_state;

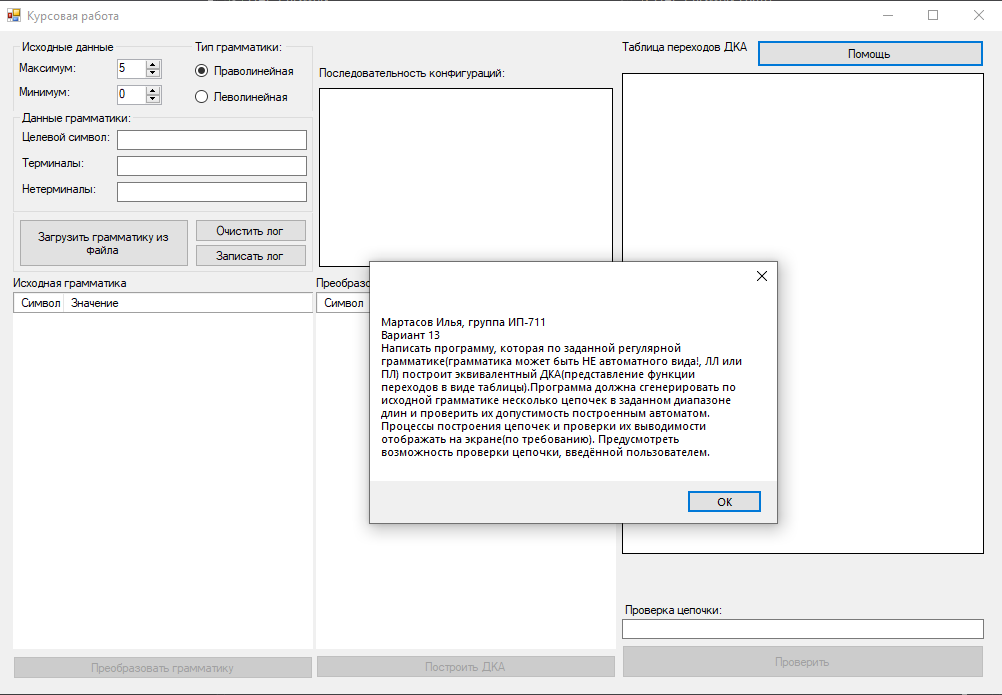
}

**Вспомогательные кнопки:**

**Загрузка грамматики из файла/запись или удаление лога/Помощь** – по нажатию на кнопку “Загрузить грамматику из файла” пользователь загружает грамматику, записанную в выбранном файле.

Кнопка “Очистить лог” удаляет лог-файл.

По нажатии на кнопку Помощь выводится имя автора, вариант и текст задания.



private void Button1\_Click(object sender, EventArgs e)

{

MessageBox.Show("Мартасов Илья, группа ИП-711\nВариант 13\nНаписать программу, которая по заданной регулярной грамматике(грамматика может быть НЕ автоматного вида!, ЛЛ или ПЛ) построит эквивалентный ДКА(представление функции переходов в виде таблицы).Программа должна сгенерировать по исходной грамматике несколько цепочек в заданном диапазоне длин и проверить их допустимость построенным автоматом. Процессы построения цепочек и проверки их выводимости отображать на экране(по требованию). Предусмотреть возможность проверки цепочки, введённой пользователем.");

}

private void Button3\_Click(object sender, EventArgs e)

{

generic.log\_buffer = "";

File.Delete(@"C:\CursWorkLogFile\log.txt");

MessageBox.Show("Лог-файл удален");

}

private void Button4\_Click(object sender, EventArgs e)

{

int counter = 4;

string path = "";

int line\_counter = 0;

char ch;

string key;

string values;

string states = "";

bool is\_fail = false;

string alphabet = "";

string start\_non\_term;

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.InitialDirectory = "c:\\";

openFileDialog.Filter = "txt files (\*.txt)|\*.txt";

openFileDialog.FilterIndex = 2;

openFileDialog.RestoreDirectory = true;

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

path = openFileDialog.FileName;

if(path == @"C:\CursWorkLogFile\log.txt")

{

generic.log\_buffer += "Нельзя загружать грамматику из лога\n";

generic.Write\_to\_log();

return;

}

}

else

{

return;

}

}

generic.log\_buffer += "Загрузка грамматики из файла:\n";

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = sr.ReadLine()) != null)

{

switch (line\_counter)

{

case 0:

if (!line.Contains("G=({") || !line.Contains("},{") || !line.Contains("},P,"))

{

generic.log\_buffer += "НЕКОРРЕКТНОЕ НАЧАЛО!\n";

generic.Write\_to\_log();

is\_fail = true;

return;

}

do

{

ch = line[counter];

if (ch == ' ')

{

continue;

}

alphabet += ch;

counter++;

} while (ch != '}');

counter += 2;

do

{

ch = line[counter];

if (ch == ' ')

{

continue;

}

states += ch;

counter++;

} while (ch != '}');

counter += 3;

start\_non\_term = line[counter].ToString();

states = states.Remove(states.Length - 1, 1);

alphabet = alphabet.Remove(alphabet.Length - 1, 1);

for(int i = 0; i < states.Length; i++)

{

if(states[i].ToString() == start\_non\_term)

{

continue;

}

}

for(int i = 0; i < states.Length; i++)

{

for(int j = 0; j < alphabet.Length; j++)

{

if(states[i] == alphabet[j])

{

generic.log\_buffer += "СОВПАДЕНИЕ!\n";

generic.Write\_to\_log();

is\_fail = true;

return;

}

}

}

term\_text\_box.Text = alphabet;

start\_non\_term\_text\_box.Text = start\_non\_term;

non\_term\_text\_box.Text = states.Replace(start\_non\_term, "");

break;

default:

for (int i = 0; i < states.Length; i++)

{

if (states[i].ToString() == line[0].ToString())

{

break;

}

}

values = line.Substring(3, line.Length - 3);

key = line[0].ToString();

Set\_grammar\_from\_file(key, values);

break;

}

counter = 0;

line\_counter++;

}

}

if(!is\_fail)

{

generic.log\_buffer += "Грамматика успешно загружена\n";

generic.Write\_to\_log();

MessageBox.Show("Грамматика успешно загружена");

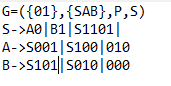
}

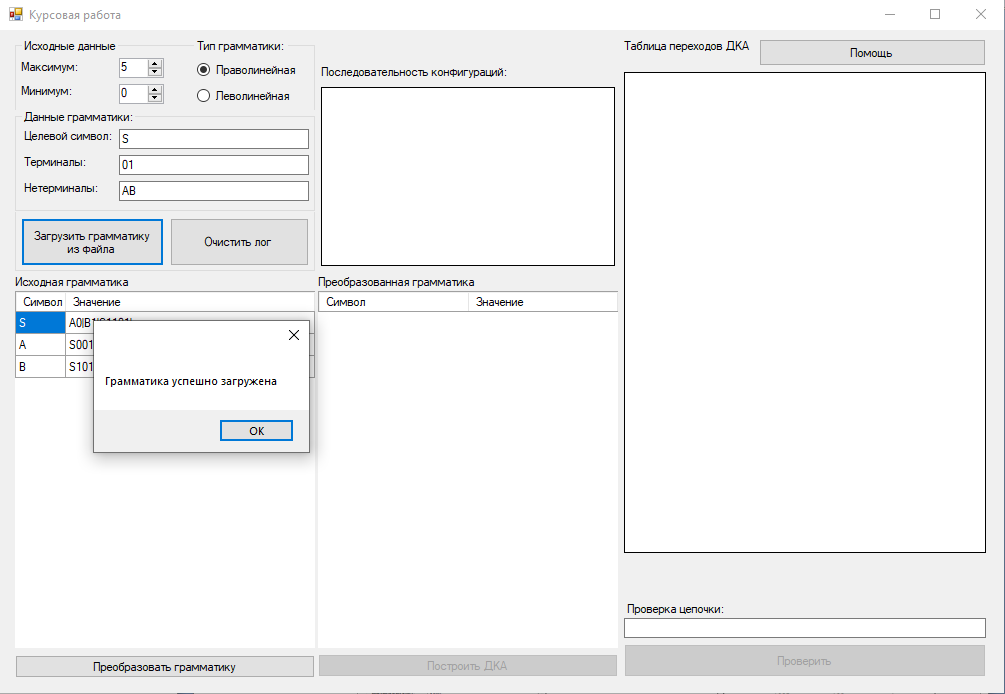
}

Результаты работы

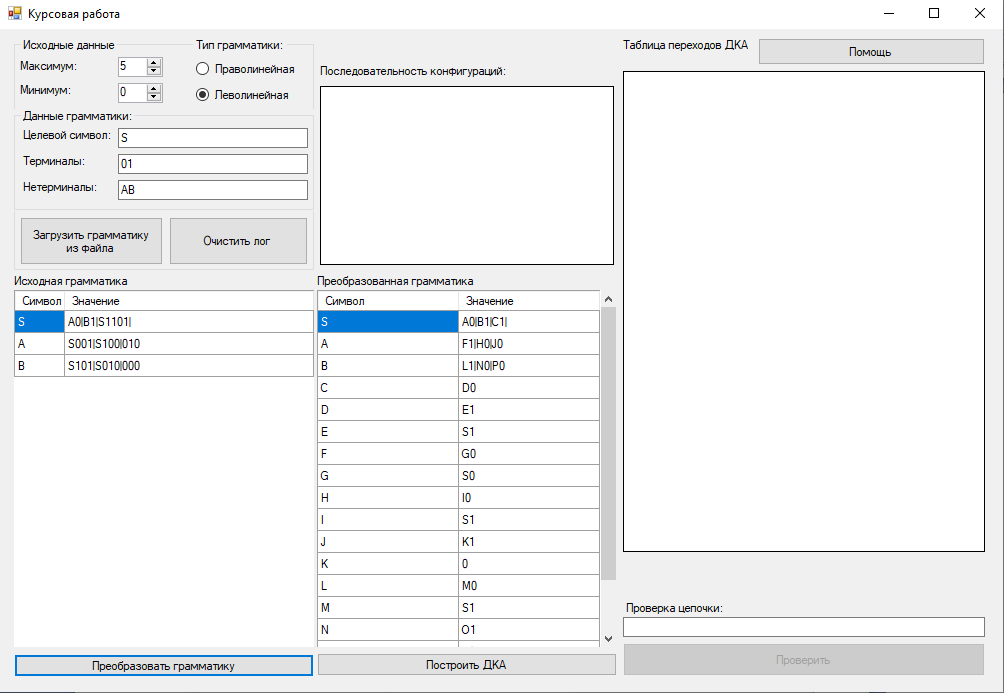
Процесс работы программы:

1. Загрузка грамматики из файла:

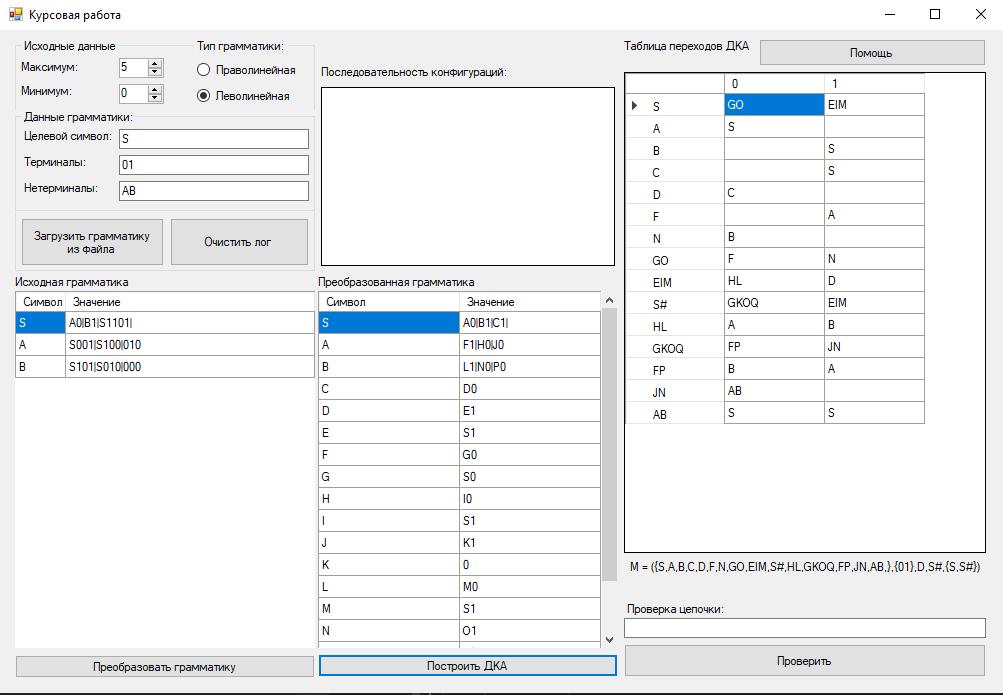




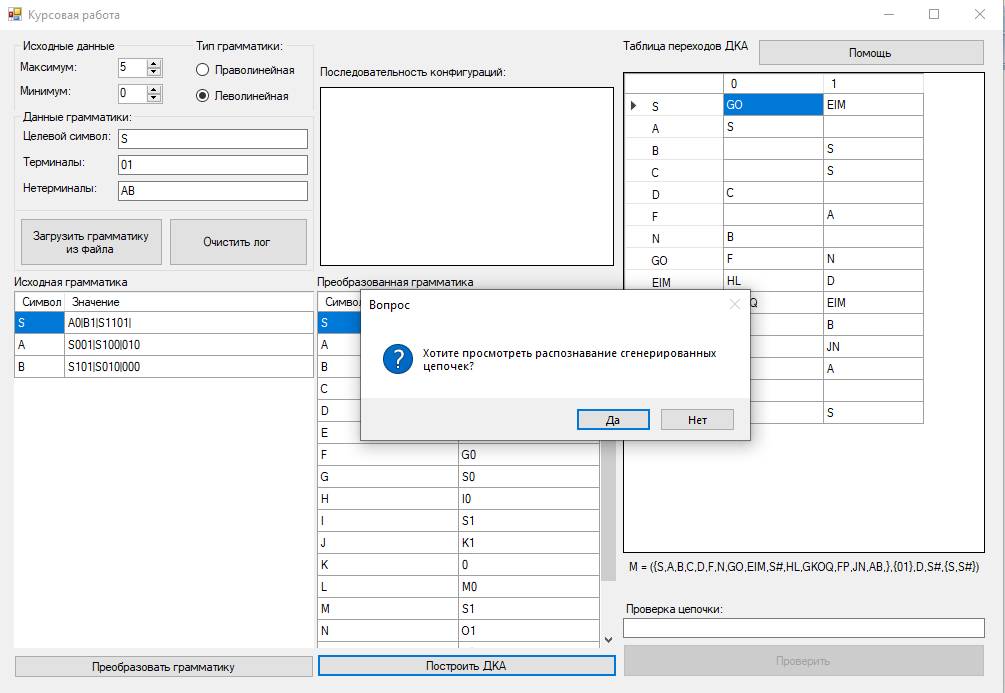
1. Преобразование грамматики



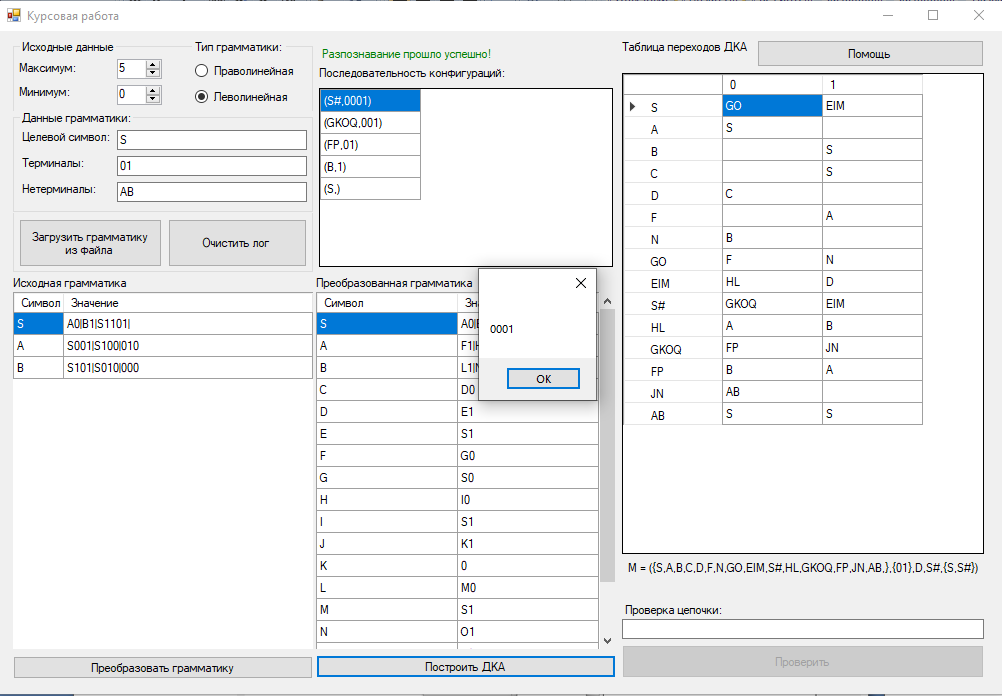
1. Построение ДКА



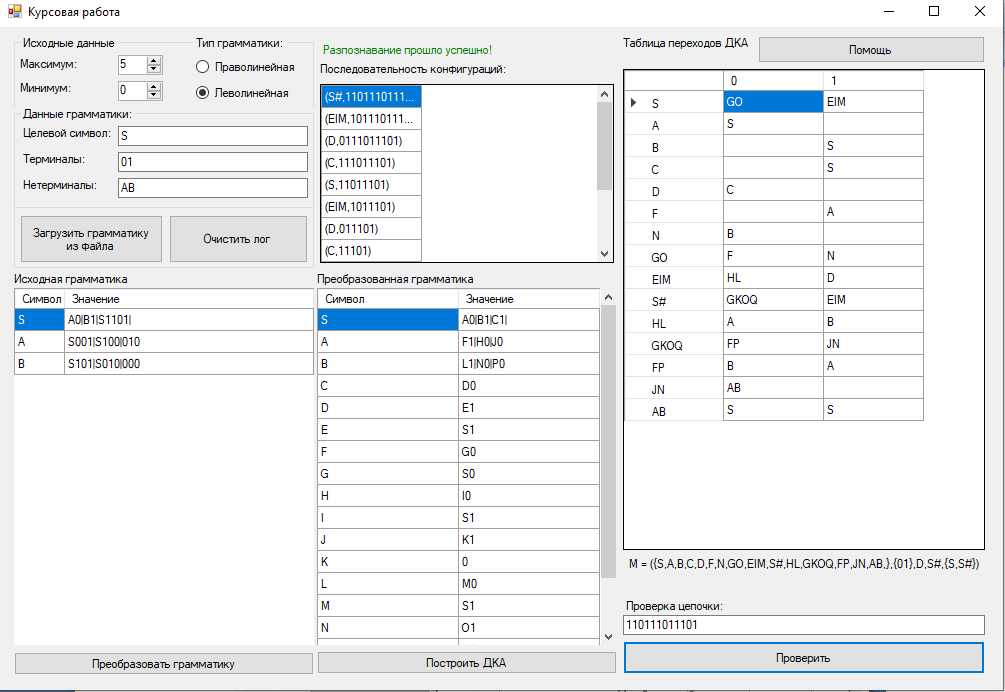
1. Вопрос о распознавании сгенерированных цепочек



1. Распознавание сгенерированной цепочки



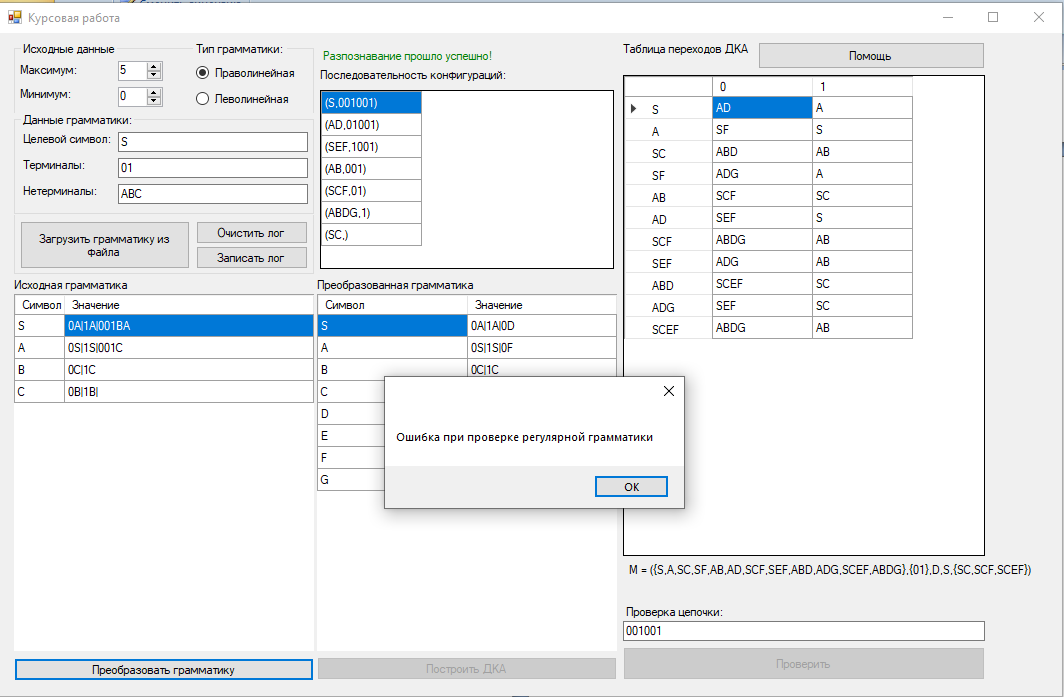
1. Распознавание пользовательской цепочки:

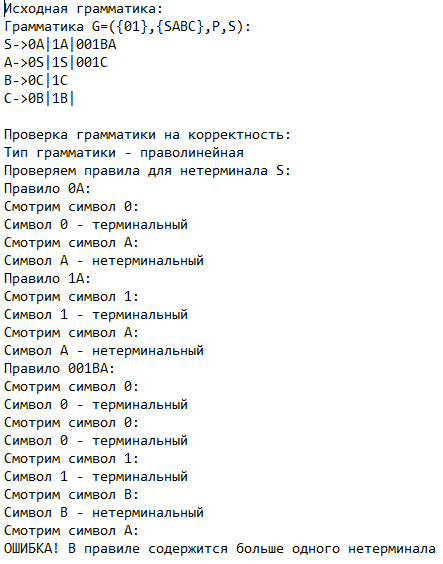


Обработка ошибок

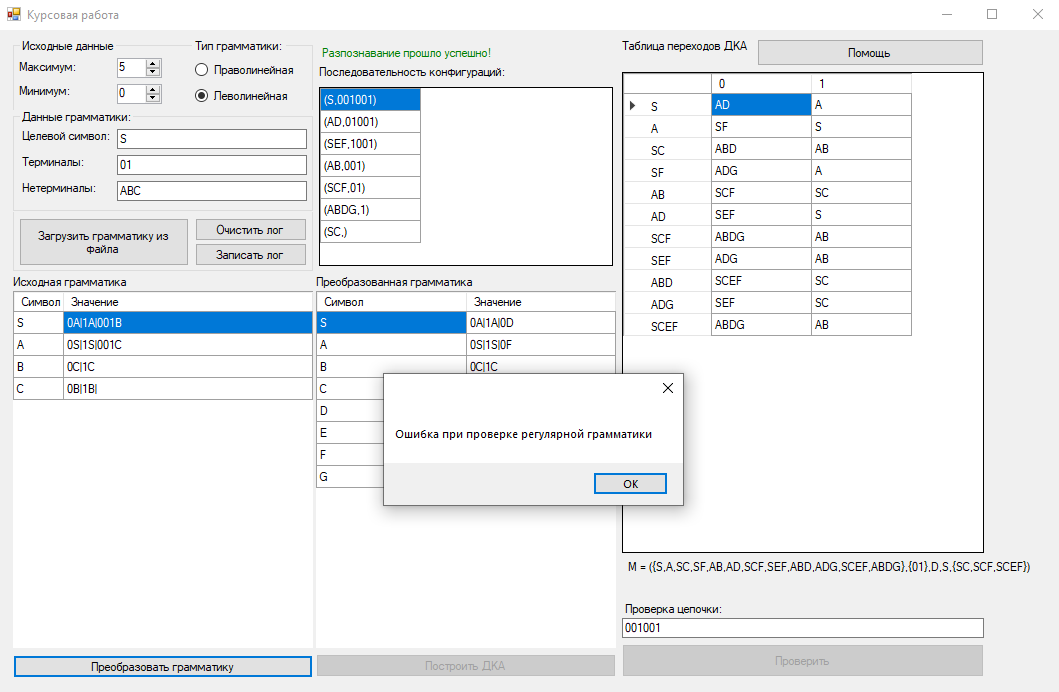
1.Ошибки при проверке регулярной грамматики:

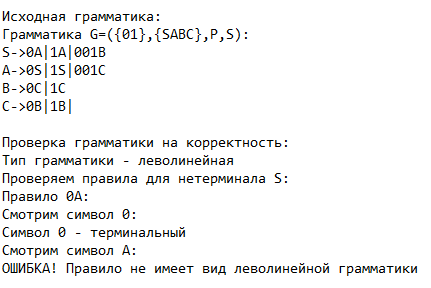
Более одного нетерминала в правиле:



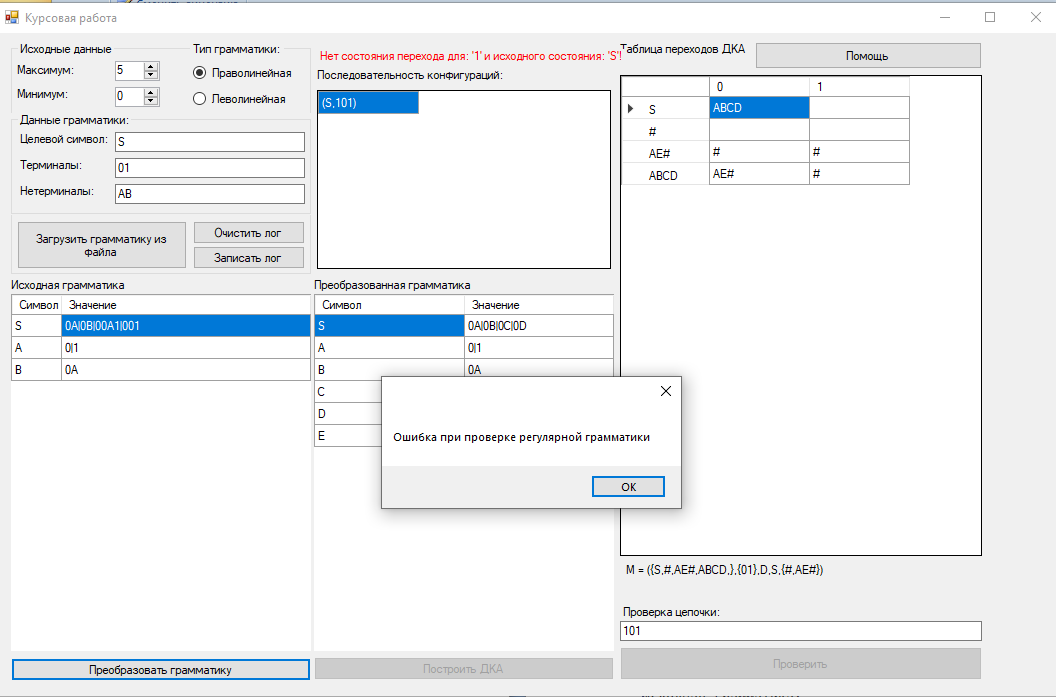


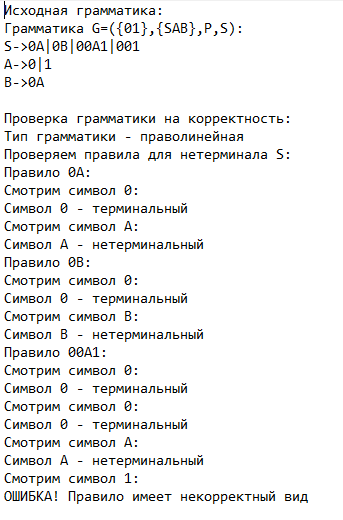
Некорректно выбранный тип грамматики



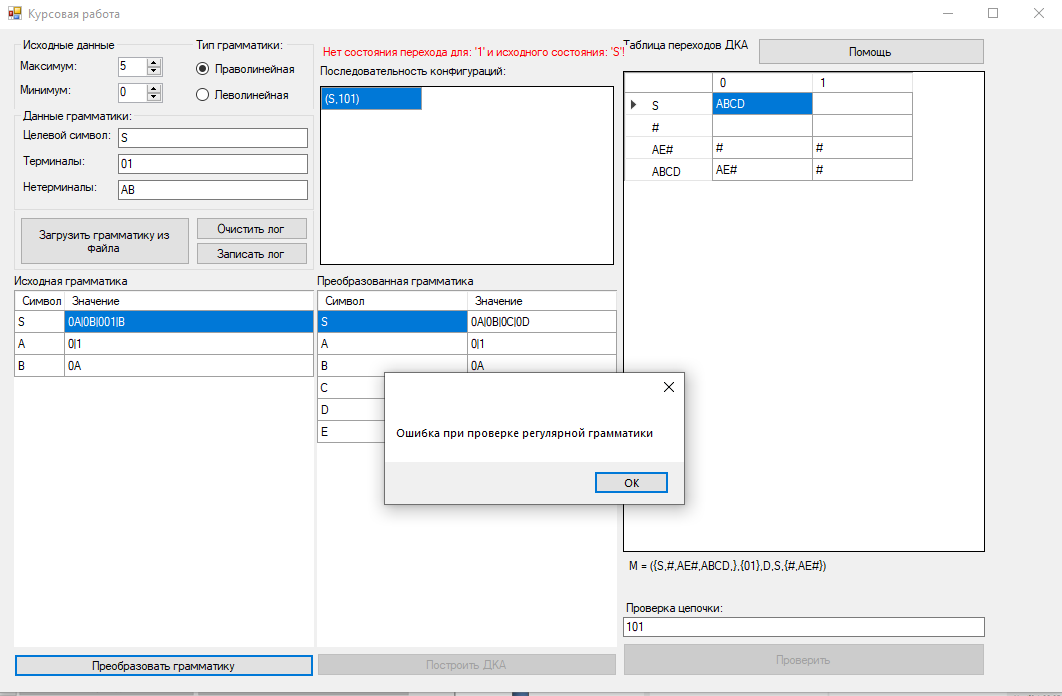


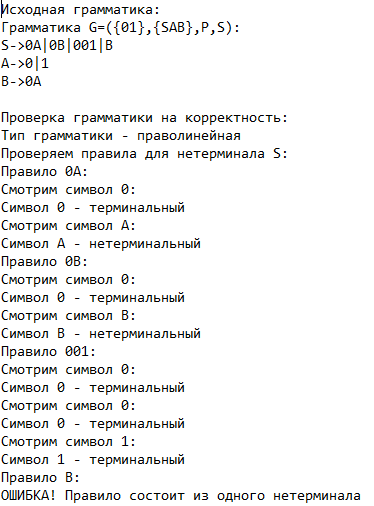
Некорректный вид правила (КС)





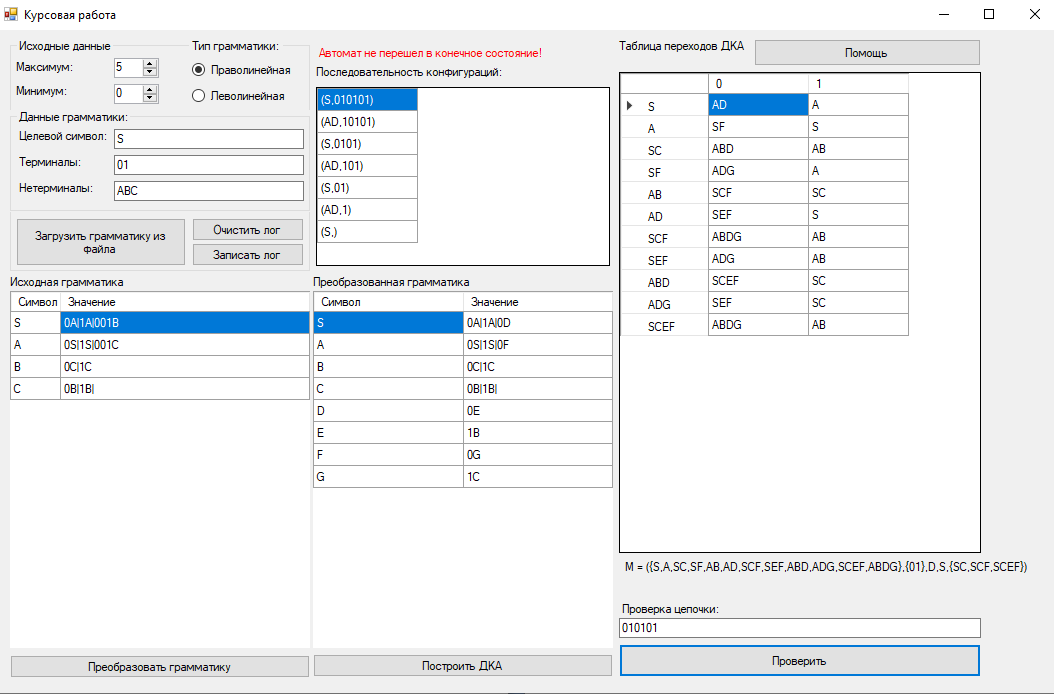
Некорректный вид правила (один нетерминал)

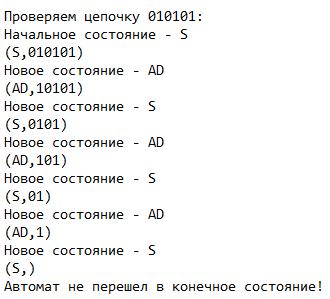




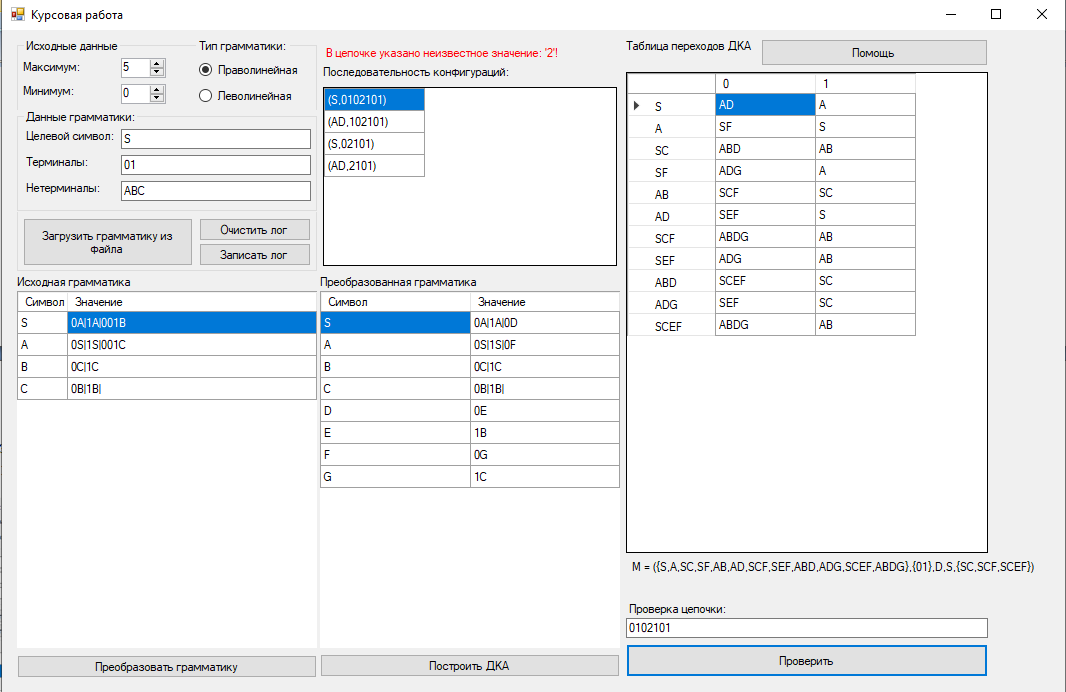
2.Ошибки при распознавании

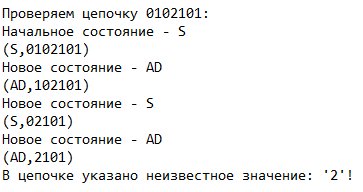
Автомат не перешел в конечное состояние



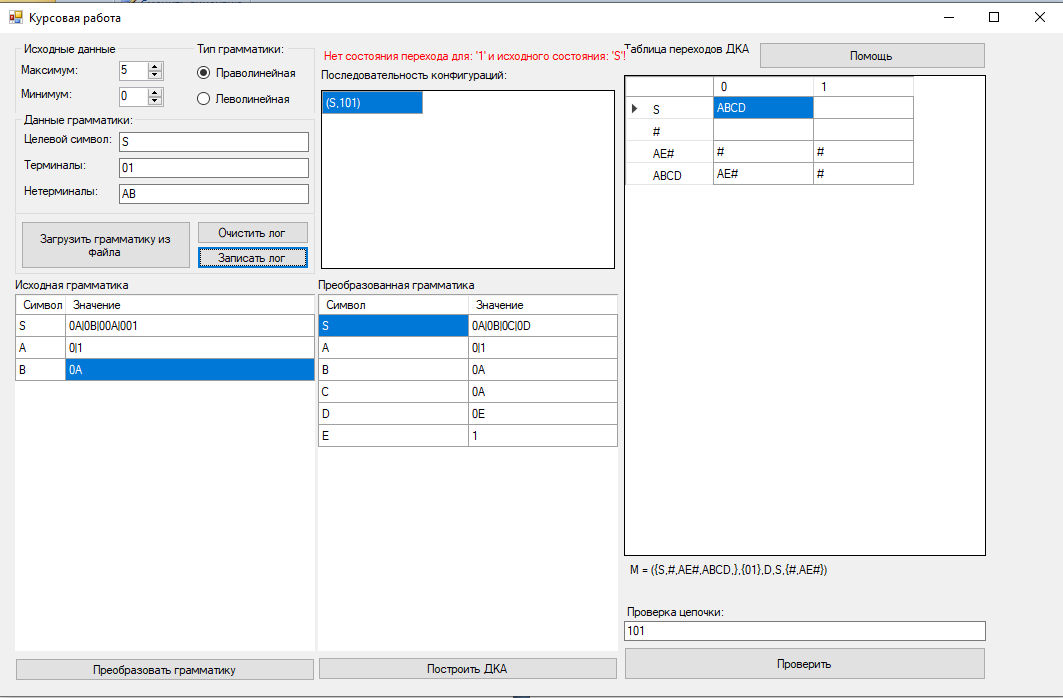


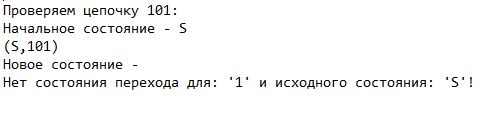
Указано неизвестное значение





Неизвестный переход





**Приложение. Полный текст программы**

**Form1.cs**

using System;

using System.IO;

using System.Linq;

using System.Data;

using System.Drawing;

using System.Windows.Forms;

using System.Collections.Generic;

namespace TYAP\_CursWork

{

public partial class Form1 : Form

{

public int regular\_side;

public string begin\_left\_state;

public bool is\_first\_launch = false;

public bool is\_changed\_start = false;

public string old\_text\_box = "";

public char is\_start;

public string res;

public int global\_keys;

public int global\_counter = 0;

readonly char[] non\_terms = new char[] {'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' };

public bool is\_stop = false;

private readonly ChainsGeneration generic;

private readonly Dictionary<string, DataGridViewRow> non\_term\_dictionary;

public string[][] transitions;

public string[][] new\_transitions;

public string final\_states;

public string buf\_final\_states;

public bool need\_to\_transform = false;

public Dictionary<string, string[]> transform\_dictionary;

public Dictionary<string, string> buf\_transform\_dictionary;

public Dictionary<string, int> states\_dictionary;

public Dictionary<string, int> symbols\_dictionary;

public Dictionary<string, int> result\_dictionary;

public Form1()

{

InitializeComponent();

generic = new ChainsGeneration();

non\_term\_dictionary = new Dictionary<string, DataGridViewRow>();

term\_text\_box.KeyPress += Term\_text\_box\_KeyPressed;

term\_text\_box.TextChanged += Term\_text\_box\_TextChanged;

non\_term\_text\_box.KeyPress += Non\_term\_text\_box\_KeyPressed;

non\_term\_text\_box.TextChanged += Non\_term\_text\_box\_TextChanged;

start\_non\_term\_text\_box.KeyPress += Start\_non\_term\_text\_box\_KeyPressed;

start\_non\_term\_text\_box.TextChanged += Start\_non\_term\_text\_box\_TextChanged;

main\_grammar\_data\_grid\_view.Columns.Add("name", "Символ");

main\_grammar\_data\_grid\_view.Columns.Add("content", "Значение");

main\_grammar\_data\_grid\_view.Columns[0].ReadOnly = true;

main\_grammar\_data\_grid\_view.Columns[0].Width = 50;

main\_grammar\_data\_grid\_view.EditingControlShowing += Rules\_grid\_view\_EditingControlShowing;

fix\_grammar\_data\_grid\_view.Columns.Add("name", "Символ");

fix\_grammar\_data\_grid\_view.Columns.Add("content", "Значение");

fix\_grammar\_data\_grid\_view.Columns[0].ReadOnly = true;

fix\_grammar\_data\_grid\_view.Columns[1].ReadOnly = true;

dka\_history\_data\_grid\_view.ColumnCount = 1;

generic.Init();

dka\_button.Enabled = false;

transform\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

transform\_button.Click += Execute\_button\_Click;

dka\_button.Click += Dka\_button\_Click;

check\_button\_text\_box.Click += Check\_button\_text\_box\_Click;

}

public void Init\_transform\_dictionary()

{

transform\_dictionary = new Dictionary<string, string[]>();

buf\_transform\_dictionary = new Dictionary<string, string>();

}

public void Init\_DKA\_dictionary()

{

result\_dictionary = new Dictionary<string, int>();

states\_dictionary = new Dictionary<string, int>();

symbols\_dictionary = new Dictionary<string, int>();

}

private void Update\_rules(bool is\_changed, int is\_text)

{

string key;

DataGridViewRow row;

if (is\_text == 1)

{

string total\_res;

string total\_count = "";

for (int i = 0; i < non\_term\_dictionary.Keys.Count; i++)

{

key = non\_term\_dictionary.Keys.ElementAt(i);

total\_res = non\_term\_dictionary[key].Cells[1].Value.ToString();

for (int j = 0; j < total\_res.Length; j++)

{

if (total\_res[j].ToString() == res)

{

total\_count += is\_start;

}

else if (total\_res[j] == is\_start)

{

total\_count += res;

}

else

{

total\_count += total\_res[j];

}

}

non\_term\_dictionary[key].Cells[1].Value = total\_count;

total\_count = "";

}

}

if (is\_changed)

{

if (!non\_term\_dictionary.ContainsKey(res))

{

//MessageBox.Show("CHANGED");

string oldValue = non\_term\_dictionary[is\_start.ToString()].Cells[1].Value.ToString();

main\_grammar\_data\_grid\_view.Rows.Remove(non\_term\_dictionary[is\_start.ToString()]);

non\_term\_dictionary.Remove(is\_start.ToString());

DataGridViewRow new\_row = new DataGridViewRow();

new\_row.CreateCells(main\_grammar\_data\_grid\_view, res, oldValue);

main\_grammar\_data\_grid\_view.Rows.Add(new\_row);

non\_term\_dictionary[res] = new\_row;

}

if (is\_changed && is\_changed\_start == false)

{

//MessageBox.Show("CHANGED 2");

string total\_res;

string total\_count = "";

for (int i = 0; i < non\_term\_dictionary.Keys.Count; i++)

{

key = non\_term\_dictionary.Keys.ElementAt(i);

total\_res = non\_term\_dictionary[key].Cells[1].Value.ToString();

for (int j = 0; j < total\_res.Length; j++)

{

if (total\_res[j].ToString() == res)

{

total\_count += is\_start;

}

else if (total\_res[j] == is\_start)

{

total\_count += res;

}

else

{

total\_count += total\_res[j];

}

}

non\_term\_dictionary[key].Cells[1].Value = total\_count;

total\_count = "";

}

}

/\*else

{

temp = non\_term\_dictionary[is\_start.ToString()].Cells[1].Value.ToString();

non\_term\_dictionary[is\_start.ToString()].Cells[1].Value = non\_term\_dictionary[res].Cells[1].Value.ToString();

non\_term\_dictionary[res].Cells[1].Value = temp;

}\*/

}

for (int i = 0; i < non\_term\_dictionary.Keys.Count; i++)

{

key = non\_term\_dictionary.Keys.ElementAt(i);

if (non\_term\_text\_box.Text.IndexOf(key) == -1 && start\_non\_term\_text\_box.Text.IndexOf(key) == -1)

{

main\_grammar\_data\_grid\_view.Rows.Remove(non\_term\_dictionary[key]);

non\_term\_dictionary.Remove(key);

}

}

for (int i = 0; i < non\_term\_text\_box.Text.Length; i++)

{

key = non\_term\_text\_box.Text[i].ToString();

if (!non\_term\_dictionary.ContainsKey(key))

{

row = new DataGridViewRow();

row.CreateCells(main\_grammar\_data\_grid\_view, key, "");

main\_grammar\_data\_grid\_view.Rows.Add(row);

non\_term\_dictionary[key] = row;

}

}

for (int i = 0; i < start\_non\_term\_text\_box.Text.Length; i++)

{

key = start\_non\_term\_text\_box.Text[i].ToString();

if (!non\_term\_dictionary.ContainsKey(key))

{

row = new DataGridViewRow();

row.CreateCells(main\_grammar\_data\_grid\_view, key, "");

main\_grammar\_data\_grid\_view.Rows.Add(row);

non\_term\_dictionary[key] = row;

}

}

}

private void Term\_text\_box\_TextChanged(object sender, EventArgs e)

{

Is\_button\_valid();

}

private void Is\_button\_valid()

{

transform\_button.Enabled = term\_text\_box.Text.Length != 0 && start\_non\_term\_text\_box.Text.Length == 1;

}

private void Add\_spaces(int count)

{

for (int j = 0; j < count; j++)

{

generic.log\_buffer += " ";

}

}

private void Get\_grammar\_description(Dictionary<string, string[]> dict)

{

string key;

string rules = "";

string non\_terms = "";

string alphabet = "";

for(int i = 0; i < term\_text\_box.Text.Length; i++)

{

alphabet += term\_text\_box.Text[i].ToString();

}

for (int i = 0; i < dict.Keys.Count; i++)

{

key = dict.Keys.ElementAt(i);

non\_terms += key;

rules += key + "->";

foreach (string is\_key in dict[key])

{

rules += is\_key + "|";

}

rules = rules.Remove(rules.Length - 1);

rules += "\n";

}

generic.log\_buffer += "G=({" + alphabet + "},{" + non\_terms + "},P," + start\_non\_term\_text\_box.Text[0].ToString() + "):\n";

generic.log\_buffer += rules + "\n";

}

private void Get\_DKA\_description(Dictionary<string, int> dict, string[][] trans)

{

string label;

string states = "";

string alphabet = "";

string buf\_final\_states = final\_states;

string begin\_state = regular\_side == 0 ? start\_non\_term\_text\_box.Text[0].ToString() : begin\_left\_state;

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

alphabet += term\_text\_box.Text[i].ToString();

}

for (int i = 0; i < dict.Keys.Count; i++)

{

states += dict.Keys.ElementAt(i) + ",";

}

if (states != "" && states[states.Length - 1].ToString() == ",")

{

states = states.Remove(states.Length - 1, 1);

}

if(buf\_final\_states != "" && buf\_final\_states[buf\_final\_states.Length - 1].ToString() == ",")

{

buf\_final\_states = buf\_final\_states.Remove(buf\_final\_states.Length - 1, 1);

}

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

Add\_spaces(15);

generic.log\_buffer += term\_text\_box.Text[i].ToString();

}

generic.log\_buffer += "\n";

for (int i = 0; i < dict.Keys.Count; i++)

{

string key = dict.Keys.ElementAt(i);

if(key != "")

{

generic.log\_buffer += key;

Add\_spaces(15 - key.Length);

for (int j = 0; j < trans[i].Length; j++)

{

generic.log\_buffer += trans[dict[key]][j];

Add\_spaces(15 - trans[dict[key]][j].Length);

}

generic.log\_buffer += "\n";

}

}

generic.log\_buffer += "M=({" + states + "},{" + alphabet + "},D," + start\_non\_term\_text\_box.Text[0].ToString() + "," + "{" + buf\_final\_states + "}):\n";

dka\_label.Text = "";

label = "M = ({" + states + "},{" + alphabet + "},D," + begin\_state + "," + "{" + buf\_final\_states + "})";

if(label.Length > 80)

{

dka\_label.Text = "Слишком длинное описание автомата. Смотрите лог";

}

else

{

dka\_label.Text = label;

}

}

private void Execute\_button\_Click(object sender, EventArgs e)

{

string key;

bool is\_term;

bool is\_grammar;

bool is\_correct = true;

string fail\_message = "";

final\_states = "";

begin\_left\_state = "";

generic.RemoveAllRules();

generic.term = term\_text\_box.Text;

generic.non\_term = non\_term\_text\_box.Text;

generic.max\_chain\_length = (uint)max\_numeric\_up\_down.Value;

generic.min\_chain\_length = (uint)min\_numeric\_up\_down.Value;

generic.start\_non\_term = start\_non\_term\_text\_box.Text[0].ToString();

regular\_side = radioButton2.Checked == true ? 0 : 1;

for (int i = 0; i < non\_term\_dictionary.Keys.Count; i++)

{

string value = "";

key = non\_term\_dictionary.Keys.ElementAt(i);

if (non\_term\_dictionary[key].Cells[1].Value != null)

{

value = non\_term\_dictionary[key].Cells[1].Value.ToString();

}

for (int er = 0; er < value.Length; er++)

{

is\_term = false;

for (int k = 0; k < term\_text\_box.Text.Length; k++)

{

if (value[er] == term\_text\_box.Text[k])

{

is\_term = true;

}

}

if (is\_term == false)

{

if ((value[er] != 'A' && value[er] != 'B'

&& value[er] != 'C' && value[er] != 'D' && value[er] != 'E' && value[er] != 'F' && value[er] != 'G'

&& value[er] != 'H' && value[er] != 'I' && value[er] != 'J' && value[er] != 'K' && value[er] != 'L'

&& value[er] != 'M' && value[er] != 'N' && value[er] != 'O' && value[er] != 'P' && value[er] != 'Q'

&& value[er] != 'R' && value[er] != 'S' && value[er] != 'T' && value[er] != 'U' && value[er] != 'V'

&& value[er] != 'W' && value[er] != 'X' && value[er] != 'Y' && value[er] != 'Z' && value[er] != '|' && value[er] != '#'))

{

is\_correct = false;

fail\_message = "ОШИБКА!\nТЕРМИНАЛЬНОГО СИМВОЛА " + value[er] + " НЕТ В АЛФАВИТЕ";

break;

}

}

}

for (int j = 0; j < value.Length; j++)

{

if (!non\_term\_dictionary.ContainsKey(value[j].ToString()) && (value[j] == 'A' || value[j] == 'B'

|| value[j] == 'C' || value[j] == 'D' || value[j] == 'E' || value[j] == 'F' || value[j] == 'G'

|| value[j] == 'H' || value[j] == 'I' || value[j] == 'J' || value[j] == 'K' || value[j] == 'L'

|| value[j] == 'M' || value[j] == 'N' || value[j] == 'O' || value[j] == 'P' || value[j] == 'Q'

|| value[j] == 'R' || value[j] == 'S' || value[j] == 'T' || value[j] == 'U' || value[j] == 'V'

|| value[j] == 'W' || value[j] == 'X' || value[j] == 'Y' || value[j] == 'Z'))

{

is\_correct = false;

fail\_message = "ОШИБКА!\nНЕТЕРМИНАЛЬНОГО СИМВОЛА " + value[j] + " НЕТ В СПИСКЕ НЕТЕРМИНАЛОВ";

break;

}

}

if (non\_term\_dictionary[key].Cells[1].Value != null)

{

generic.AddRuleDescription(key, non\_term\_dictionary[key].Cells[1].Value.ToString());

}

else

{

generic.AddRuleDescription(key, "");

}

}

generic.log\_buffer += "Исходная грамматика:\n";

Get\_grammar\_description(generic.rules\_dictionary);

is\_grammar = Check\_regular\_grammar();

if (is\_correct == true && is\_grammar == true)

{

generic.GenerateChains();

Init\_transform\_dictionary();

if(regular\_side == 0)

{

Transform\_grammar();

}

else

{

Transform\_left\_grammar();

}

dka\_button.Enabled = true;

}

else

{

if(fail\_message == "")

{

fail\_message = "Ошибка при проверке регулярной грамматики";

}

MessageBox.Show(fail\_message);

}

}

private void Check\_dka(string checkable)

{

is\_stop = false;

bool is\_final = false;

string new\_state;

string current\_state;

dka\_history\_data\_grid\_view.Rows.Clear();

List<string> dka\_history = new List<string>();

if (regular\_side == 0)

{

current\_state = start\_non\_term\_text\_box.Text.ToString();

}

else

{

current\_state = begin\_left\_state;

}

generic.log\_buffer += "Проверяем цепочку " + checkable + ":\n";

generic.log\_buffer += "Начальное состояние - " + current\_state + "\n";

dka\_history.Add("(" + current\_state + "," + checkable + ")");

generic.log\_buffer += "(" + current\_state + "," + checkable + ")\n";

for (int i = 0; i < checkable.Length; i++)

{

if (!symbols\_dictionary.ContainsKey(checkable[i].ToString()))

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "В цепочке указано неизвестное значение: '" + checkable[i].ToString() + "'!";

generic.log\_buffer += "В цепочке указано неизвестное значение: '" + checkable[i].ToString() + "'!\n\n";

is\_stop = true;

break;

}

new\_state = Transict\_function(checkable[i].ToString(), current\_state);

if (is\_stop == true)

{

break;

}

dka\_history.Add("(" + new\_state + "," + checkable.Substring(i + 1) + ")");

generic.log\_buffer += "(" + new\_state + "," + checkable.Substring(i + 1) + ")\n";

current\_state = new\_state;

}

if(is\_stop == false)

{

string[] final = final\_states.Split(',');

for (int i = 0; i < final.Length; i++)

{

if (current\_state == final[i].ToString())

{

is\_final = true;

info\_label.ForeColor = Color.Green;

info\_label.Text = "Разпознавание прошло успешно!";

generic.log\_buffer += "Разпознавание прошло успешно!\n\n";

}

}

if (is\_final == false)

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "Автомат не перешел в конечное состояние!";

generic.log\_buffer += "Автомат не перешел в конечное состояние!\n\n";

}

}

foreach (string str in dka\_history)

{

DataGridViewRow row = new DataGridViewRow();

row.CreateCells(dka\_history\_data\_grid\_view, str);

dka\_history\_data\_grid\_view.Rows.Add(row);

}

}

private void Check\_button\_text\_box\_Click(object sender, EventArgs e)

{

Check\_dka(check\_text\_box.Text.ToString());

generic.Write\_to\_log();

}

private string Transict\_function(string symbol, string current\_state)

{

string new\_state;

if (need\_to\_transform == true)

{

new\_state = new\_transitions[states\_dictionary[current\_state]][symbols\_dictionary[symbol]];

generic.log\_buffer += "Новое состояние - " + new\_state + "\n";

}

else

{

new\_state = transitions[states\_dictionary[current\_state]][symbols\_dictionary[symbol]];

generic.log\_buffer += "Новое состояние - " + new\_state + "\n";

}

if (new\_state == "")

{

info\_label.ForeColor = Color.Red;

info\_label.Text = "Нет состояния перехода для: '" + symbol + "' и исходного состояния: '" + current\_state + "'!";

generic.log\_buffer += "Нет состояния перехода для: '" + symbol + "' и исходного состояния: '" + current\_state + "'!\n\n";

is\_stop = true;

}

return new\_state;

}

private void Build\_left\_side\_grammar\_to\_DKA()

{

generic.log\_buffer += "Добавляем целевой символ в конечное состояние\n";

final\_states += start\_non\_term\_text\_box.Text[0].ToString() + ",";

for (int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

foreach (string is\_key in transform\_dictionary[key])

{

if (is\_key.Length == 1 && !Check\_non\_term(is\_key) && is\_key != "")

{

//log\_buffer += "Добавляем переход для дополнительного состояния #:\n";

begin\_left\_state += "#";

transitions[transform\_dictionary.Keys.Count][symbols\_dictionary[is\_key[0].ToString()]] += key;

}

else if (is\_key != "")

{

//log\_buffer += "Добавляем переход для состояния " + is\_key[0].ToString() + "\n";

transitions[states\_dictionary[is\_key[0].ToString()]][symbols\_dictionary[is\_key[1].ToString()]] += key;

}

}

}

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

for (int j = 0; j < transitions[i].Length; j++)

{

if (transitions[i][j] != "")

{

transitions[i][j] = Rearrange\_in\_order(transitions[i][j]);

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

else

{

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

}

}

}

private void Build\_right\_side\_grammar\_to\_DKA()

{

string key;

string res = "";

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

if (i != transform\_dictionary.Keys.Count)

{

key = transform\_dictionary.Keys.ElementAt(i);

foreach (string is\_key in transform\_dictionary[key])

{

if(is\_key != "")

{

if (is\_key[0].ToString() == term\_text\_box.Text[j].ToString() && is\_key.Length == 2)

{

res += is\_key[1].ToString();

}

else if (is\_key[0].ToString() == term\_text\_box.Text[j].ToString() && is\_key.Length == 1)

{

res += "#";

final\_states += "#" + ",";

}

}

}

transitions[i][j] = res;

transitions[i][j] = Rearrange\_in\_order(transitions[i][j]);

generic.log\_buffer += "Новый переход: " + key + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

res = "";

}

else

{

transitions[i][j] = "";

generic.log\_buffer += "Новый переход: " + "#" + "->" + transitions[i][j] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

}

}

}

}

private void Except\_non\_entry\_states(string[][] trans)

{

int current\_count;

int is\_new\_state = 1;

Dictionary<string, int> buf\_states\_dictionary = new Dictionary<string, int>();

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

buf\_states\_dictionary[key] = 0;

}

generic.log\_buffer += "Начинаем смотреть с состояния ";

if (regular\_side == 1)

{

generic.log\_buffer += begin\_left\_state + "\n";

buf\_states\_dictionary[begin\_left\_state] = 1;

}

else

{

generic.log\_buffer += start\_non\_term\_text\_box.Text[0].ToString() + "\n";

buf\_states\_dictionary[start\_non\_term\_text\_box.Text[0].ToString()] = 1;

}

generic.Write\_to\_log();

current\_count = is\_new\_state;

generic.log\_buffer += "Осуществляем проход по состояниям:\n";

for (; ; )

{

generic.log\_buffer += "Следующая итерация:\n";

for (int i = 0; i < states\_dictionary.Keys.Count; i++)

{

string key = states\_dictionary.Keys.ElementAt(i);

if (buf\_states\_dictionary[key] == 0)

{

generic.log\_buffer += "Состояние " + key + " пока не является достижимым\n";

continue;

}

for (int j = 0; j < term\_text\_box.Text.Length; j++)

{

is\_new\_state++;

buf\_states\_dictionary[trans[i][j]] = 1;

if (trans[i][j] != "")

{

generic.log\_buffer += "Состояние " + trans[i][j] + " - достижимое\n";

}

}

}

generic.log\_buffer += "Обнаружено " + is\_new\_state + " достижимых состояний\n";

if (current\_count == is\_new\_state)

{

break;

}

current\_count = is\_new\_state;

is\_new\_state = 0;

generic.Write\_to\_log();

}

generic.log\_buffer += "Список достижимых состояний:\n";

for (int i = 0; i < buf\_states\_dictionary.Keys.Count; i++)

{

string key = buf\_states\_dictionary.Keys.ElementAt(i);

if (buf\_states\_dictionary[key] == 1)

{

result\_dictionary[key] = i;

generic.log\_buffer += key + "\n";

}

}

generic.Write\_to\_log();

}

private void Add\_new\_final\_states(Dictionary<string, int> result\_dictionary)

{

string[] buf = final\_states.Split(',');

for (int i = 0; i < result\_dictionary.Keys.Count; i++)

{

string key = result\_dictionary.Keys.ElementAt(i);

if (key != "")

{

for (int j = 0; j < buf.Length; j++)

{

for (int k = 0; k < key.Length; k++)

{

if (buf[j].ToString() == key[k].ToString())

{

final\_states += key + ",";

}

}

}

}

}

generic.log\_buffer += "Список конечных состояний - " + final\_states + "\n";

}

private void Dka\_button\_Click(object sender, EventArgs e)

{

need\_to\_transform = false;

Init\_DKA\_dictionary();

generic.log\_buffer += "Построение ДКА:\n\nДобавляем исходные состояния, " +

"соответствующие нетерминалам грамматики а также дополнительное состояние №:\n";

for (int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += key + " " + i + "\n";

states\_dictionary[key] = i;

}

states\_dictionary["#"] = transform\_dictionary.Keys.Count;

generic.log\_buffer += "#" + " " + transform\_dictionary.Keys.Count + "\n";

generic.Write\_to\_log();

for (int i = 0; i < term\_text\_box.Text.Length; i++)

{

symbols\_dictionary[term\_text\_box.Text[i].ToString()] = i;

}

transitions = new string[transform\_dictionary.Keys.Count + 1][];

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for(int j = 0; j < transitions[i].Length; j++)

{

transitions[i][j] = "";

}

}

if (regular\_side == 1)

{

generic.log\_buffer += "Создаем автомат по леволинейной грамматике:\n";

Build\_left\_side\_grammar\_to\_DKA();

}

else

{

generic.log\_buffer += "Создаем автомат по праволинейной грамматике:\n";

Build\_right\_side\_grammar\_to\_DKA();

}

generic.log\_buffer += "Построение первоначального автомата закончено\n";

generic.Write\_to\_log();

generic.log\_buffer += "Первоначальный вид автомата:\n";

Get\_DKA\_description(states\_dictionary, transitions);

generic.Write\_to\_log();

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

if (need\_to\_transform == true)

{

break;

}

for (int j = 0; j < transitions[i].Length; j++)

{

if (need\_to\_transform == true)

{

break;

}

if (transitions[i][j].Length >= 2)

{

generic.log\_buffer += "Автомат недетерминированный - нужно преобразовать\n";

generic.Write\_to\_log();

need\_to\_transform = true;

Transform\_to\_DKA();

break;

}

}

}

if(!need\_to\_transform)

{

generic.log\_buffer += "Автомат детерминированный, но...\n";

if (regular\_side == 1)

{

generic.log\_buffer += "Преобразуем множество начальных состояний для ДКА по леволинейной грамматике\n";

new\_transitions = new string[10000][];

for (int i = 0; i < 10000; i++)

{

new\_transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

new\_transitions[i][j] = transitions[i][j];

}

}

if (!Add\_new\_states())

{

return;

}

need\_to\_transform = true;

generic.log\_buffer += "Исключаем недостижимые состояния в ДКА\n";

Except\_non\_entry\_states(new\_transitions);

generic.log\_buffer += "Приводим в порядок список конечных состояний\n";

Add\_new\_final\_states(result\_dictionary);

Print\_states\_to\_user(new\_transitions);

generic.Write\_to\_log();

}

else

{

generic.log\_buffer += "Исключаем недостижимые состояния в новом ДКА\n";

Except\_non\_entry\_states(transitions);

generic.log\_buffer += "Приводим в порядок список конечных состояний\n";

Add\_new\_final\_states(result\_dictionary);

Print\_states\_to\_user(transitions);

generic.Write\_to\_log();

}

generic.log\_buffer += "Итоговый автомат:\n";

if (need\_to\_transform == true)

{

Get\_DKA\_description(states\_dictionary, new\_transitions);

}

else

{

Get\_DKA\_description(states\_dictionary, transitions);

}

Print\_ask\_window();

generic.Write\_to\_log();

check\_button\_text\_box.Enabled = true;

}

}

private bool Transform\_to\_DKA()

{

new\_transitions = new string[10000][];

for (int i = 0; i < 10000; i++)

{

new\_transitions[i] = new string[term\_text\_box.Text.Length];

}

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < transitions[i].Length; j++)

{

new\_transitions[i][j] = transitions[i][j];

}

}

generic.log\_buffer += "Добавляем новые состояния в автомат:\n";

if (!Add\_new\_states())

{

return false;

}

generic.Write\_to\_log();

generic.log\_buffer += "Исключаем недостижимые состояния:\n";

Except\_non\_entry\_states(new\_transitions);

generic.Write\_to\_log();

generic.log\_buffer += "Приводим в порядок конечные состояния:\n";

Add\_new\_final\_states(result\_dictionary);

generic.Write\_to\_log();

Print\_states\_to\_user(new\_transitions);

generic.log\_buffer += "\nПреобразованный автомат:\n";

Get\_DKA\_description(result\_dictionary, new\_transitions);

generic.Write\_to\_log();

Print\_ask\_window();

check\_button\_text\_box.Enabled = true;

return true;

}

private bool Add\_new\_states()

{

int counter = 1;

string old\_state;

string result = "";

generic.log\_buffer += "Ищем новые состояния среди изначальных:\n";

for (int i = 0; i < transform\_dictionary.Keys.Count + 1; i++)

{

for (int j = 0; j < new\_transitions[i].Length; j++)

{

if (!states\_dictionary.ContainsKey(new\_transitions[i][j]) && new\_transitions[i][j] != "")

{

old\_state = new\_transitions[i][j];

generic.log\_buffer += "Новое состояние - " + old\_state + "\n";

states\_dictionary[old\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < new\_transitions[i].Length; h++)

{

for (int k = 0; k < old\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[old\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[old\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + old\_state + "->" + new\_transitions[states\_dictionary[old\_state]][h] + "(" + term\_text\_box.Text[j].ToString() + ")\n";

result = "";

}

counter++;

generic.Write\_to\_log();

}

}

}

int is\_new\_state;

int old\_states\_crt = counter - 1;

bool is\_entered\_in\_previous = false;

generic.log\_buffer += "Было добавлено " + old\_states\_crt + "новых состояний\n";

int clue\_number = states\_dictionary.Keys.Count - old\_states\_crt;

generic.Write\_to\_log();

if (regular\_side == 1)

{

generic.log\_buffer += "Смотрим множество начальных состояний:\n";

if (begin\_left\_state == "")

{

dka\_label.Text = "";

generic.log\_buffer += "ОШИБКА! В исходной грамматике не обнаружено правил выхода из рекурсии!\n";

MessageBox.Show("Ошибка построения автомата");

dka\_data\_grid\_view.Rows.Clear();

dka\_data\_grid\_view.Columns.Clear();

check\_button\_text\_box.Enabled = false;

generic.Write\_to\_log();

return false;

}

begin\_left\_state = Rearrange\_in\_order(begin\_left\_state);

generic.log\_buffer += "Начальное состояние для ДКА - " + begin\_left\_state + "\n";

for(int i = 0; i < transform\_dictionary.Keys.Count; i++)

{

string key = transform\_dictionary.Keys.ElementAt(i);

if (key == begin\_left\_state)

{

is\_entered\_in\_previous = true;

break;

}

}

if (is\_entered\_in\_previous == false && begin\_left\_state != "#")

{

generic.log\_buffer += "Начальное состояние отсутствует в списке исходных\n" +

"Добавляем новое начальное состояние " + begin\_left\_state + "\n";

states\_dictionary[begin\_left\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < term\_text\_box.Text.Length; h++)

{

for (int k = 0; k < begin\_left\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[begin\_left\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[begin\_left\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + begin\_left\_state + "->" + new\_transitions[states\_dictionary[begin\_left\_state]][h] + "(" + term\_text\_box.Text[h].ToString() + ")\n";

result = "";

}

counter++;

is\_new\_state = old\_states\_crt + 1;

}

else

{

is\_new\_state = old\_states\_crt;

}

generic.Write\_to\_log();

}

else

{

if (final\_states == "")

{

dka\_label.Text = "";

generic.log\_buffer += "ОШИБКА! В исходной грамматике не обнаружено правил выхода из рекурсии!\n";

MessageBox.Show("Ошибка построения автомата");

dka\_data\_grid\_view.Rows.Clear();

dka\_data\_grid\_view.Columns.Clear();

check\_button\_text\_box.Enabled = false;

generic.Write\_to\_log();

return false;

}

is\_new\_state = old\_states\_crt;

}

generic.log\_buffer += "Просматриваем новые состояния и ищем новые среди них:\n";

while (is\_new\_state != 0)

{

generic.log\_buffer += "Осталось посмотреть " + is\_new\_state + " состояний:\n";

string key = states\_dictionary.Keys.ElementAt(clue\_number);

for (int j = 0; j < term\_text\_box.Text.Length; j++)

{

if (!states\_dictionary.ContainsKey(new\_transitions[states\_dictionary[key]][j])

&& new\_transitions[states\_dictionary[key]][j] != "")

{

is\_new\_state++;

old\_state = new\_transitions[states\_dictionary[key]][j];

generic.log\_buffer += "Новое состояние - " + old\_state + "\n";

states\_dictionary[old\_state] = transform\_dictionary.Keys.Count + counter;

generic.log\_buffer += "Добавляем для нового состояния переходы:\n";

for (int h = 0; h < term\_text\_box.Text.Length; h++)

{

for (int k = 0; k < old\_state.Length; k++)

{

result += new\_transitions[states\_dictionary[old\_state[k].ToString()]][h];

}

result = Rearrange\_in\_order(result);

new\_transitions[states\_dictionary[old\_state]][h] = result;

generic.log\_buffer += "Новый переход: " + old\_state + "->" + new\_transitions[states\_dictionary[old\_state]][h] + "(" + term\_text\_box.Text[h].ToString() + ")\n";

result = "";

}

counter++;

}

}

clue\_number++;

is\_new\_state--;

generic.Write\_to\_log();

}

generic.log\_buffer += "Добавление состояний окончено\n";

generic.Write\_to\_log();

return true;

}

private string Rearrange\_in\_order(string string\_to\_order)

{

string\_to\_order = new string(string\_to\_order.ToCharArray().Distinct().ToArray());

string\_to\_order = string.Concat(string\_to\_order.OrderBy(x => x).ToArray());

if (string\_to\_order.Contains("#"))

{

string\_to\_order = string\_to\_order.Replace("#", "");

string\_to\_order += "#";

}

if (string\_to\_order.Contains(start\_non\_term\_text\_box.Text[0].ToString()))

{

string\_to\_order = string\_to\_order.Replace(start\_non\_term\_text\_box.Text[0].ToString(), "");

string\_to\_order = start\_non\_term\_text\_box.Text[0].ToString() + string\_to\_order;

}

return string\_to\_order;

}

private void Print\_states\_to\_user(string [][]trans)

{

DataGridViewRow viewRow;

dka\_data\_grid\_view.Rows.Clear();

dka\_data\_grid\_view.Columns.Clear();

for (int j = 0; j < term\_text\_box.Text.Length; j++)

{

dka\_data\_grid\_view.Columns.Add(term\_text\_box.Text[j].ToString(), term\_text\_box.Text[j].ToString());

}

for (int i = 0; i < result\_dictionary.Keys.Count; i++)

{

string key = result\_dictionary.Keys.ElementAt(i);

if (key != "")

{

viewRow = new DataGridViewRow();

viewRow.HeaderCell.Value = key;

dka\_data\_grid\_view.Rows.Add(viewRow);

for (int j = 0; j < trans[i].Length; j++)

{

dka\_data\_grid\_view.Rows[i].Cells[j].Value = trans[result\_dictionary[key]][j];

}

}

}

final\_states = final\_states.Remove(final\_states.Length - 1, 1);

string[] final = final\_states.Split(',');

final = final.Distinct().ToArray();

final\_states = "";

for (int i = 0; i < final.Length; i++)

{

if (result\_dictionary.ContainsKey(final[i].ToString()))

{

final\_states += final[i].ToString() + ",";

}

};

for (int i = 0; i < result\_dictionary.Keys.Count; i++)

{

string key = result\_dictionary.Keys.ElementAt(i);

if (key != "")

{

for (int j = 0; j < trans[i].Length; j++)

{

dka\_data\_grid\_view.Rows[i].Cells[j].Value = trans[result\_dictionary[key]][j];

}

}

}

}

private void Print\_ask\_window()

{

var result = MessageBox.Show("Хотите просмотреть распознавание сгенерированных цепочек?",

"Вопрос",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question);

if (result == DialogResult.Yes)

{

generic.log\_buffer += "\n\n\nПроверяем сгенерированные цепочки:\n";

foreach (string chain in generic.chains\_list)

{

Check\_dka(chain);

MessageBox.Show(chain);

}

}

generic.Write\_to\_log();

}

private bool Check\_non\_term(string key)

{

if (key == start\_non\_term\_text\_box.Text[0].ToString())

{

return true;

}

for (int j = 0; j < non\_term\_text\_box.Text.Length; j++)

{

if (key == non\_term\_text\_box.Text[j].ToString())

{

return true;

}

}

return false;

}

private bool Check\_regular\_grammar()

{

string key;

bool is\_non\_term;

bool is\_first\_term;

string type\_grammar = regular\_side == 0 ? "праволинейная" : "леволинейная";

generic.log\_buffer += "Проверка грамматики на корректность:\nТип грамматики - " + type\_grammar + "\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Проверяем правила для нетерминала " + key + ":\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

if (is\_key.Length == 1)

{

if(Check\_non\_term(is\_key[0].ToString()))

{

generic.log\_buffer += "ОШИБКА! Правило состоит из одного нетерминала\n\n";

return false;

}

}

if (is\_key.Length >= 2)

{

generic.log\_buffer += "Смотрим символ " + is\_key[0].ToString() + ":\n";

if (Check\_non\_term(is\_key[0].ToString()))

{

generic.log\_buffer += "Символ " + is\_key[0].ToString() + " - нетерминальный\n";

is\_non\_term = true;

is\_first\_term = false;

if (regular\_side == 0)

{

generic.log\_buffer += "ОШИБКА! Правило не имеет вид праволинейной грамматики\n\n";

return false;

}

}

else

{

generic.log\_buffer += "Символ " + is\_key[0].ToString() + " - терминальный\n";

is\_non\_term = false;

is\_first\_term = true;

}

for (int j = 1; j < is\_key.Length; j++)

{

string help = is\_key[j].ToString();

generic.log\_buffer += "Смотрим символ " + help + ":\n";

if (Check\_non\_term(help))

{

if(is\_first\_term == true && regular\_side == 1)

{

generic.log\_buffer += "ОШИБКА! Правило не имеет вид леволинейной грамматики\n\n";

return false;

}

if (is\_non\_term == true)

{

generic.log\_buffer += "ОШИБКА! В правиле содержится больше одного нетерминала\n\n";

return false;

}

is\_non\_term = true;

generic.log\_buffer += "Символ " + help + " - нетерминальный\n";

}

else

{

if (is\_non\_term == true && is\_first\_term == true)

{

generic.log\_buffer += "ОШИБКА! Правило имеет некорректный вид\n\n";

return false;

}

generic.log\_buffer += "Символ " + help + " - терминальный\n";

}

}

}

is\_non\_term = false;

}

}

generic.log\_buffer += "Проверка завершена успешно!\n\n";

return true;

}

private string Add\_new\_key()

{

string key = "";

for (int i = 0; i < 26; i++)

{

if (!generic.rules\_dictionary.ContainsKey(non\_terms[i].ToString())

&& !buf\_transform\_dictionary.ContainsKey(non\_terms[i].ToString()))

{

key = non\_terms[i].ToString();

break;

}

}

return key;

}

private bool Transform\_grammar()

{

int length;

int new\_nonterms;

string key;

string buf\_rule;

string result\_rules = "";

bool is\_last\_term = false;

DataGridViewRow row;

final\_states = "";

transform\_dictionary.Clear();

buf\_transform\_dictionary.Clear();

fix\_grammar\_data\_grid\_view.Rows.Clear();

generic.log\_buffer += "Преобразование грамматики:\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Преобразуем правила нетерминала " + key + ":\n\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

length = is\_key.Length - 1;

if (is\_key == "")

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

final\_states += key + ",";

result\_rules += "|";

continue;

}

if (is\_key.Length == 2 && (Check\_non\_term(is\_key[length].ToString())))

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += is\_key + "|";

continue;

}

if (is\_key.Length >= 2)

{

if (Check\_non\_term(is\_key[length].ToString()))

{

new\_nonterms = length - 1;

is\_last\_term = true;

}

else

{

new\_nonterms = length;

}

if (!Check\_non\_term(is\_key[length].ToString()) && is\_key.Length == 2)

{

new\_nonterms = 1;

}

generic.log\_buffer += "Для преобразования нужно " + new\_nonterms + " новых нетерминалов\n\n";

generic.log\_buffer += "Преобразуем:\n";

string new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + new\_key + "\n";

result\_rules += is\_key[0].ToString() + new\_key + "|";

generic.log\_buffer += "Новое правило для нетерминала " + key + " - " + is\_key[0].ToString() + new\_key + "\n";

buf\_rule = is\_key.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

while (new\_nonterms >= 0)

{

string buf\_new\_key;

if (is\_last\_term == true)

{

while (buf\_rule.Length > 2)

{

string new\_rule = buf\_rule[0].ToString() + new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_rule[0].ToString() + buf\_new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

else

{

while (buf\_rule.Length > 1)

{

string new\_rule = buf\_rule[0].ToString() + new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_rule[0].ToString() + buf\_new\_key;

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(0, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

}

is\_last\_term = false;

}

else

{

result\_rules += is\_key + "|";

}

}

result\_rules = result\_rules.Remove(result\_rules.Length - 1);

generic.log\_buffer += "Преобразованные правила: " + result\_rules + "\n\n";

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, key, result\_rules);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[key] = result\_rules.Split('|');

result\_rules = "";

}

for (int i = 0; i < buf\_transform\_dictionary.Keys.Count; i++)

{

string buffer\_key = buf\_transform\_dictionary.Keys.ElementAt(i);

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, buffer\_key, buf\_transform\_dictionary[buffer\_key]);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[buffer\_key] = buf\_transform\_dictionary[buffer\_key].Split('|'); ;

}

generic.log\_buffer += "Преобразованная грамматика:\n";

Get\_grammar\_description(transform\_dictionary);

generic.Write\_to\_log();

return true;

}

private bool Transform\_left\_grammar()

{

int length;

int new\_nonterms;

string key;

string buf\_rule;

string result\_rules = "";

bool is\_last\_term = false;

DataGridViewRow row;

final\_states = "";

transform\_dictionary.Clear();

buf\_transform\_dictionary.Clear();

fix\_grammar\_data\_grid\_view.Rows.Clear();

generic.log\_buffer += "Преобразование грамматики:\n";

for (int i = 0; i < generic.rules\_dictionary.Keys.Count; i++)

{

key = generic.rules\_dictionary.Keys.ElementAt(i);

generic.log\_buffer += "Преобразуем правила нетерминала " + key + ":\n\n";

foreach (string is\_key in generic.rules\_dictionary[key])

{

generic.log\_buffer += "Правило " + is\_key + ":\n";

length = is\_key.Length - 1;

if (is\_key == "")

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += "|";

begin\_left\_state += key;

continue;

}

if (is\_key.Length == 2 && (Check\_non\_term(is\_key[0].ToString())))

{

generic.log\_buffer += "Правило не нуждается в преобразовании\n\n";

result\_rules += is\_key + "|";

continue;

}

if (is\_key.Length >= 2)

{

if (Check\_non\_term(is\_key[0].ToString()))

{

new\_nonterms = length - 1;

is\_last\_term = true;

}

else

{

new\_nonterms = length + 1;

}

if (!Check\_non\_term(is\_key[0].ToString()) && is\_key.Length == 2)

{

new\_nonterms = 2;

}

generic.log\_buffer += "Для преобразования нужно " + new\_nonterms + " новых нетерминалов\n\n";

generic.log\_buffer += "Преобразуем:\n";

string new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + new\_key + "\n";

result\_rules += new\_key + is\_key[length].ToString() + "|";

generic.log\_buffer += "Новое правило для нетерминала " + key + " - " + new\_key + is\_key[length].ToString() + "\n";

buf\_rule = is\_key.Remove(length, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

while (new\_nonterms >= 0)

{

string buf\_new\_key;

if (is\_last\_term == true)

{

while (buf\_rule.Length > 2)

{

string new\_rule = new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(buf\_rule.Length - 1, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

else

{

while (buf\_rule.Length > 1)

{

string new\_rule = new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n";

buf\_new\_key = Add\_new\_key();

generic.log\_buffer += "Добавляем новый нетерминал " + buf\_new\_key + "\n";

buf\_transform\_dictionary.Remove(new\_key);

new\_rule = buf\_new\_key + buf\_rule[buf\_rule.Length - 1].ToString();

buf\_transform\_dictionary[new\_key] = new\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + new\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

new\_key = buf\_new\_key;

buf\_rule = buf\_rule.Remove(buf\_rule.Length - 1, 1);

generic.log\_buffer += "Остаток исходного правила - " + buf\_rule + "\n";

}

buf\_transform\_dictionary[new\_key] = buf\_rule;

generic.log\_buffer += "Новое правило:\n" + new\_key + "->" + buf\_rule + "\n\n";

generic.log\_buffer += "Осталось добавить " + new\_nonterms + " нетерминалов\n";

new\_nonterms--;

}

}

is\_last\_term = false;

}

else

{

result\_rules += is\_key + "|";

}

}

result\_rules = result\_rules.Remove(result\_rules.Length - 1);

generic.log\_buffer += "Преобразованные правила: " + result\_rules + "\n\n";

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, key, result\_rules);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[key] = result\_rules.Split('|');

result\_rules = "";

}

for (int i = 0; i < buf\_transform\_dictionary.Keys.Count; i++)

{

string buffer\_key = buf\_transform\_dictionary.Keys.ElementAt(i);

row = new DataGridViewRow();

row.CreateCells(fix\_grammar\_data\_grid\_view, buffer\_key, buf\_transform\_dictionary[buffer\_key]);

fix\_grammar\_data\_grid\_view.Rows.Add(row);

transform\_dictionary[buffer\_key] = buf\_transform\_dictionary[buffer\_key].Split('|'); ;

}

generic.log\_buffer += "Преобразованная грамматика:\n";

Get\_grammar\_description(transform\_dictionary);

generic.Write\_to\_log();

return true;

}

private void Rules\_grid\_view\_EditingControlShowing(object sender, DataGridViewEditingControlShowingEventArgs e)

{

e.Control.KeyPress += CellKeyPressed;

e.Control.Leave += Control\_Leave;

((TextBox)e.Control).ShortcutsEnabled = false;

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Control\_Leave(object sender, EventArgs e)

{

((Control)sender).KeyPress -= CellKeyPressed;

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Start\_non\_term\_text\_box\_TextChanged(object sender, EventArgs e)

{

if (start\_non\_term\_text\_box.Text.Length == 0)

{

Update\_rules(false, 0);

Is\_button\_valid();

is\_first\_launch = false;

}

else

{

bool is\_changed = false;

string total\_res = "";

for (int i = 0; i < non\_term\_text\_box.Text.Length; i++)

{

if (start\_non\_term\_text\_box.Text[0].Equals(non\_term\_text\_box.Text[i]))

{

is\_changed\_start = true;

is\_changed = true;

res = non\_term\_text\_box.Text[i].ToString().Replace(non\_term\_text\_box.Text[i], is\_start);

total\_res += res;

res = non\_term\_text\_box.Text[i].ToString();

}

else

{

total\_res += non\_term\_text\_box.Text[i];

}

}

non\_term\_text\_box.Text = total\_res;

old\_text\_box = total\_res;

if (is\_changed == false)

{

res = start\_non\_term\_text\_box.Text[0].ToString();

if (!non\_term\_dictionary.ContainsKey(res) && is\_first\_launch == false)

{

is\_first\_launch = true;

}

else if (!non\_term\_dictionary.ContainsKey(res) && is\_first\_launch == true)

{

is\_changed = true;

}

}

Update\_rules(is\_changed, 0);

is\_start = start\_non\_term\_text\_box.Text[0];

Is\_button\_valid();

is\_changed\_start = false;

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Non\_term\_text\_box\_TextChanged(object sender, EventArgs e)

{

bool buf\_changed = is\_changed\_start;

string new\_text\_box = non\_term\_text\_box.Text;

if (new\_text\_box.Length == old\_text\_box.Length && is\_changed\_start == false)

{

for (int i = 0; i < old\_text\_box.Length; i++)

{

if (old\_text\_box[i] != new\_text\_box[i])

{

is\_start = old\_text\_box[i];

res = new\_text\_box[i].ToString();

break;

}

}

is\_changed\_start = true;

Update\_rules(true, 1);

is\_changed\_start = buf\_changed;

is\_start = start\_non\_term\_text\_box.Text[0];

}

else

{

Update\_rules(false, 0);

Is\_button\_valid();

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Start\_non\_term\_text\_box\_KeyPressed(object sender, KeyPressEventArgs e)

{

start\_non\_term\_text\_box.SelectionLength = 0;

if ((start\_non\_term\_text\_box.Text.IndexOf(e.KeyChar) != -1 || e.KeyChar < 'A' || e.KeyChar > 'Z') && e.KeyChar != (char)Keys.Back)

{

e.Handled = true;

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Non\_term\_text\_box\_KeyPressed(object sender, KeyPressEventArgs e)

{

non\_term\_text\_box.SelectionLength = 0;

old\_text\_box = non\_term\_text\_box.Text;

if ((start\_non\_term\_text\_box.Text.IndexOf(e.KeyChar) != -1 || non\_term\_text\_box.Text.IndexOf(e.KeyChar) != -1 || e.KeyChar < 'A' || e.KeyChar > 'Z') && e.KeyChar != (char)Keys.Back)

{

e.Handled = true;

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Term\_text\_box\_KeyPressed(object sender, KeyPressEventArgs e)

{

term\_text\_box.SelectionLength = 0;

if (term\_text\_box.Text.IndexOf(e.KeyChar) != -1 || (e.KeyChar >= 'A' && e.KeyChar <= 'Z') || e.KeyChar == '|')

{

e.Handled = true;

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void CellKeyPressed(object sender, KeyPressEventArgs e)

{

if (start\_non\_term\_text\_box.Text.IndexOf(e.KeyChar) == -1 && non\_term\_text\_box.Text.IndexOf(e.KeyChar) == -1 && term\_text\_box.Text.IndexOf(e.KeyChar) == -1 && e.KeyChar != '|' && e.KeyChar != (char)Keys.Back)

{

e.Handled = true;

}

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void MaxChainResultLenSpinner\_ValueChanged(object sender, EventArgs e)

{

min\_numeric\_up\_down.Maximum = max\_numeric\_up\_down.Value;

dka\_button.Enabled = false;

check\_button\_text\_box.Enabled = false;

}

private void Button1\_Click(object sender, EventArgs e)

{

MessageBox.Show("Мартасов Илья, группа ИП-711\nВариант 13\nНаписать программу, которая по заданной регулярной грамматике(грамматика может быть НЕ автоматного вида!, ЛЛ или ПЛ) построит эквивалентный ДКА(представление функции переходов в виде таблицы).Программа должна сгенерировать по исходной грамматике несколько цепочек в заданном диапазоне длин и проверить их допустимость построенным автоматом. Процессы построения цепочек и проверки их выводимости отображать на экране(по требованию). Предусмотреть возможность проверки цепочки, введённой пользователем.");

}

private void Button3\_Click(object sender, EventArgs e)

{

generic.log\_buffer = "";

File.Delete(@"C:\CursWorkLogFile\log.txt");

MessageBox.Show("Лог-файл удален");

}

private void Button4\_Click(object sender, EventArgs e)

{

int counter = 4;

string path = "";

int line\_counter = 0;

char ch;

string key;

string values;

string states = "";

bool is\_fail = false;

string alphabet = "";

string start\_non\_term;

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.InitialDirectory = "c:\\";

openFileDialog.Filter = "txt files (\*.txt)|\*.txt";

openFileDialog.FilterIndex = 2;

openFileDialog.RestoreDirectory = true;

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

path = openFileDialog.FileName;

if(path == @"C:\CursWorkLogFile\log.txt")

{

generic.log\_buffer += "Нельзя загружать грамматику из лога\n";

generic.Write\_to\_log();

return;

}

}

else

{

return;

}

}

generic.log\_buffer += "Загрузка грамматики из файла:\n";

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = sr.ReadLine()) != null)

{

switch (line\_counter)

{

case 0:

if (!line.Contains("G=({") || !line.Contains("},{") || !line.Contains("},P,"))

{

generic.log\_buffer += "НЕКОРРЕКТНОЕ НАЧАЛО!\n";

generic.Write\_to\_log();

is\_fail = true;

return;

}

do

{

ch = line[counter];

if (ch == ' ')

{

continue;

}

alphabet += ch;

counter++;

} while (ch != '}');

counter += 2;

do

{

ch = line[counter];

if (ch == ' ')

{

continue;

}

states += ch;

counter++;

} while (ch != '}');

counter += 3;

start\_non\_term = line[counter].ToString();

states = states.Remove(states.Length - 1, 1);

alphabet = alphabet.Remove(alphabet.Length - 1, 1);

for(int i = 0; i < states.Length; i++)

{

if(states[i].ToString() == start\_non\_term)

{

continue;

}

}

for(int i = 0; i < states.Length; i++)

{

for(int j = 0; j < alphabet.Length; j++)

{

if(states[i] == alphabet[j])

{

generic.log\_buffer += "СОВПАДЕНИЕ!\n";

generic.Write\_to\_log();

is\_fail = true;

return;

}

}

}

term\_text\_box.Text = alphabet;

start\_non\_term\_text\_box.Text = start\_non\_term;

non\_term\_text\_box.Text = states.Replace(start\_non\_term, "");

break;

default:

for (int i = 0; i < states.Length; i++)

{

if (states[i].ToString() == line[0].ToString())

{

break;

}

}

values = line.Substring(3, line.Length - 3);

key = line[0].ToString();

Set\_grammar\_from\_file(key, values);

break;

}

counter = 0;

line\_counter++;

}

}

if(!is\_fail)

{

generic.log\_buffer += "Грамматика успешно загружена\n";

generic.Write\_to\_log();

MessageBox.Show("Грамматика успешно загружена");

}

}

private void Set\_grammar\_from\_file(string key, string value)

{

non\_term\_dictionary[key].Cells[1].Value = value;

Update\_rules(false, 0);

Is\_button\_valid();

}

}

}

ChainsGeneration.cs

using System.Linq;

using System.Collections.Generic;

using System.IO;

using System;

using System.Windows.Forms;

namespace TYAP\_CursWork

{

class ChainsGeneration

{

public Form1 former = null;

public string log\_buffer = "";

public List<string> chains\_list;

public Dictionary<string, string[]> rules\_dictionary;

public uint max\_chain\_length = 0;

public uint min\_chain\_length = 0;

public string term = "";

public string non\_term = "";

public string start\_non\_term = "";

public const string EMPTY\_STR = "λ";

private const int MAX\_RECURSION = 10000;

public void Init()

{

chains\_list = new List<string>();

rules\_dictionary = new Dictionary<string, string[]>();

}

public void AddRuleDescription(string key, string description)

{

rules\_dictionary[key] = description.Split('|');

}

public void RemoveAllRules()

{

chains\_list.Clear();

rules\_dictionary.Clear();

}

public void GenerateChains()

{

string res;

int recursion = 0;

string chain\_prefix;

List<string> non\_term\_chains = new List<string>();

log\_buffer += "Генерируем цепочки:\n";

log\_buffer += "Смотрим правила для целевого символа:\n";

foreach (string rule\_right\_part in rules\_dictionary[start\_non\_term])

{

if (0 == Is\_valid(rule\_right\_part) && rule\_right\_part.Length >= min\_chain\_length)

{

log\_buffer += "Сгенерирована цепочка " + rule\_right\_part + "\n";

chains\_list.Add(rule\_right\_part == "" ? "" : rule\_right\_part);

}

else if (-2 == Is\_valid(rule\_right\_part))

{

log\_buffer += "Цепочка с нетериналами " + rule\_right\_part + " добалвена \n";

non\_term\_chains.Add(rule\_right\_part);

}

}

if (non\_term\_chains.Count != 0)

{

while (recursion <= MAX\_RECURSION)

{

recursion++;

List<string> sub\_non\_term\_chains = new List<string>();

foreach (string non\_term\_chain in non\_term\_chains)

{

log\_buffer += "Смотрим цепочку " + non\_term\_chain + ":\n";

chain\_prefix = "";

for (int i = 0; i < non\_term\_chain.Length; i++)

{

if (!rules\_dictionary.ContainsKey(non\_term\_chain[i].ToString()))

{

chain\_prefix += non\_term\_chain[i];

}

else

{

log\_buffer += "Смотрим правила для нетерминала " + non\_term\_chain[i].ToString() + ":\n";

foreach (string rule\_right\_part in rules\_dictionary[non\_term\_chain[i].ToString()])

{

log\_buffer += "Правило " + rule\_right\_part + ":\n";

res = chain\_prefix + rule\_right\_part + non\_term\_chain.Substring(i + 1);

log\_buffer += "Получившаяся цепочка - " + res + ":\n";

if (0 == Is\_valid(res))

{

if (chains\_list.Contains(res == "" ? "" : res) || res.Length < min\_chain\_length)

{

break;

}

else

{

chains\_list.Add(res == "" ? "" : res);

log\_buffer += "Сгенерирована цепочка " + res + "\n";

if (chains\_list.Count >= 5)

{

break;

}

}

}

else if (-2 == Is\_valid(res))

{

log\_buffer += "Заносим " + res + " в список несгенерированных:\n";

sub\_non\_term\_chains.Add(res);

}

if (chains\_list.Count >= 5)

{

break;

}

}

break;

}

}

if (chains\_list.Count >= 5)

{

break;

}

}

if (chains\_list.Count >= 5)

{

break;

}

non\_term\_chains.Clear();

non\_term\_chains = sub\_non\_term\_chains.Distinct().ToList();

}

log\_buffer += "Процесс генерации закончен. Список сгенерированных цепочек:\n";

foreach (string chain in chains\_list)

{

log\_buffer += chain + "\n";

}

}

else

{

log\_buffer += "Процесс генерации закончен. Список сгенерированных цепочек:\n";

foreach (string chain in chains\_list)

{

log\_buffer += chain + "\n";

}

}

Write\_to\_log();

}

private int Is\_valid(string line)

{

int term\_sym = 0;

int non\_term\_sym = 0;

foreach (char ch in line)

{

if (!rules\_dictionary.ContainsKey(ch.ToString()))

{

term\_sym++;

}

else

{

non\_term\_sym++;

}

}

if (term\_sym > max\_chain\_length)

{

return -1;

}

if ((term\_sym + non\_term\_sym - 5) > max\_chain\_length)

{

return -1;

}

return (non\_term\_sym > 0) ? -2 : 0;

}

public void Write\_to\_log()

{

string dir\_path = @"C:\CursWorkLogFile";

string path = @"C:\CursWorkLogFile\log.txt";

try

{

if (!Directory.Exists(dir\_path))

{

DirectoryInfo di = Directory.CreateDirectory(dir\_path);

}

}

catch (Exception e)

{

MessageBox.Show("The process failed: {0}");

}

finally { }

using (StreamWriter sw = new StreamWriter(path, false, System.Text.Encoding.Default))

{

sw.WriteLine(log\_buffer);

}

}

}

}