Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Институт информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"

профиль "Программное обеспечение средств вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

**Курсовая работа по дисциплине  
 Теория языков программирования и методы трансляции**

Вариант 13

Выполнил:

Cтудент гр. ИП-813 \_\_\_\_\_\_\_\_\_\_\_\_\_/Пещеров В. А./

ФИО студента

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Проверил:

Ассистент кафедры ПМиК \_\_\_\_\_\_\_\_\_\_\_\_\_/Павлова У. В./

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2021 г.

**Содержание**

Оглавление

[**Текст задания** 3](#_Toc89732403)

[**Описание алгоритма решения** 4](#_Toc89732404)

[**Пример работы программы** 6](#_Toc89732405)

[**Описание основных блоков программы** 15](#_Toc89732406)

[**Полный текст программы** 18](#_Toc89732407)

[**Результаты тестирования** 29](#_Toc89732408)

# **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 алгоритма решения**

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

1. Проверка грамматики на правильность ввода и генерация цепочек по грамматике

2. Преобразовываем правила грамматики к автоматному виду (если нужно)

3. Построение ДКА.

**Пункт 1:**

При нажатии на кнопку “Преобразовать грамматику” программа проверяет соответствие грамматики на принадлежность выбранному типу (ПП или ЛЛ). Программа просто проверяет расположение нетерминалов в правиле. Затем наша программа генерирует цепочки. Для этого в начале она смотрит на целевой символ, и если за ним есть нетерминальный символ, цепочка продолжает генерироваться (в данной программе пустой символ означает λ). Затем программа идёт по нетерминальным символам и каждый раз проверяет наличие в цепочки нетерминальных символов, если их нет , то цепочка сгенерирована полностью.

**Пункт 2:**

Далее преобразуем грамматику к автоматному виду. Программа поочерёдно просматривает правила грамматики и при наличии в цепочке более одного терминального символа, программа создаёт дополнительный нетерминальный символ и для каждого из них записываются правила. Исходное правило просто разбивается на символы и разносятся по терминалам.

**Пункт 3:**

Далее строится сам ДКА. Программа присваивает каждому нетерминалу идентичное состояния в виде индекса, также создаётся дополнительное состояние # - оно может использоваться как конечное для ПП или как начальное для ЛЛ. При построении ДКА может возникнуть проблема, что автомат будет недетерминированным. Программа определяет это следующим образом, если в переходов есть состояния больше длинны 1 и если при построении ДКА по ЛЛ, в грамматике оказывается несколько состояний, тогда трансформируем НКА в ДКА. Программа в переходах ищет состояния, которые в ней не присутствуют, и создаётся новая функция переходов, состоящая из исходной функции и новых состояний, полученных в результате поиска. Затем для новых состояний, полученных при обходе, добавляются переходы, получившиеся в результате объединения всех прежних переходов из каждого состояния, из которых состоит новое состояние. Далее происходит поиск новых состояний среди тех, которые появились на шаге 1. Аналогично добавляются переходы для этих состояний. 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. 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 ПП.

**Праволинейная грамматика**:

Пусть G=({01},{SAB},P,S)

P:

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

A->001S|100S|010

B->101S|010S|000

**Пункт 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.

**Пункт 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

**Пункт 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,#})

# **Описание основных блоков 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.

**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.

**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\_non\_term добавляет новый нетерминал в грамматику.

Функция Get\_grammar\_description выводит описание получившейся грамматики.

**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

**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.

# **Полный текст программы**

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;

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

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();

}

}

}

# **Результаты тестирования**

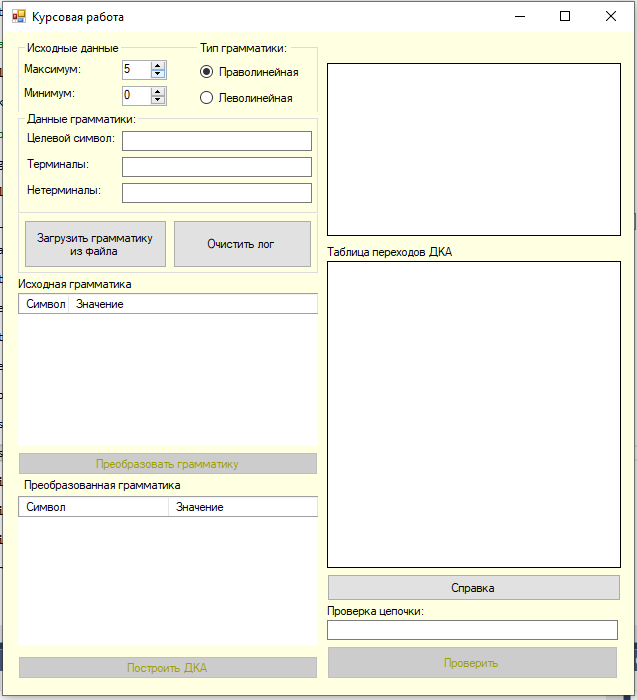
****

Рисунок 1. Интерфейс программы

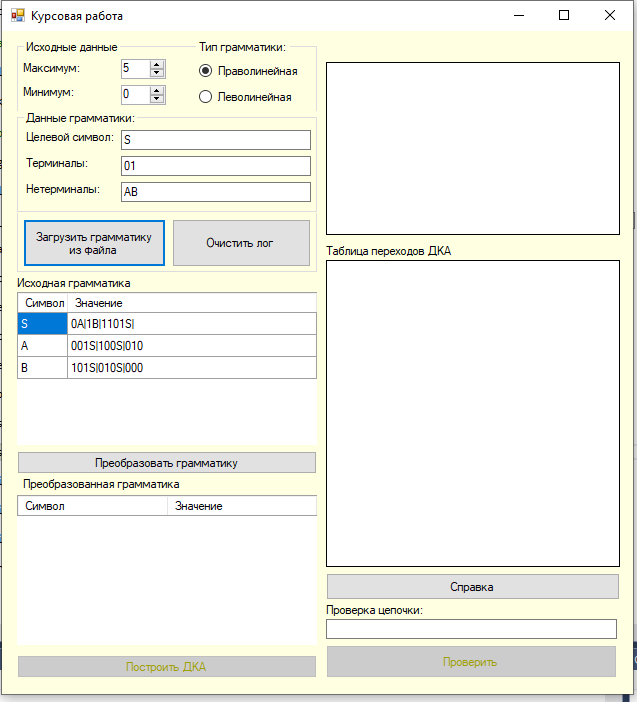


Рисунок 2. Ввод грамматики из фаила

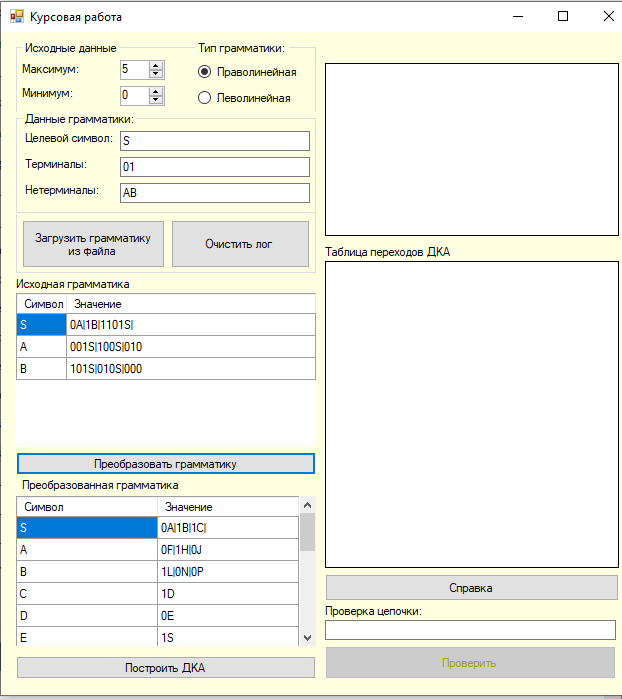


Рисунок 3. Преобразование грамматики

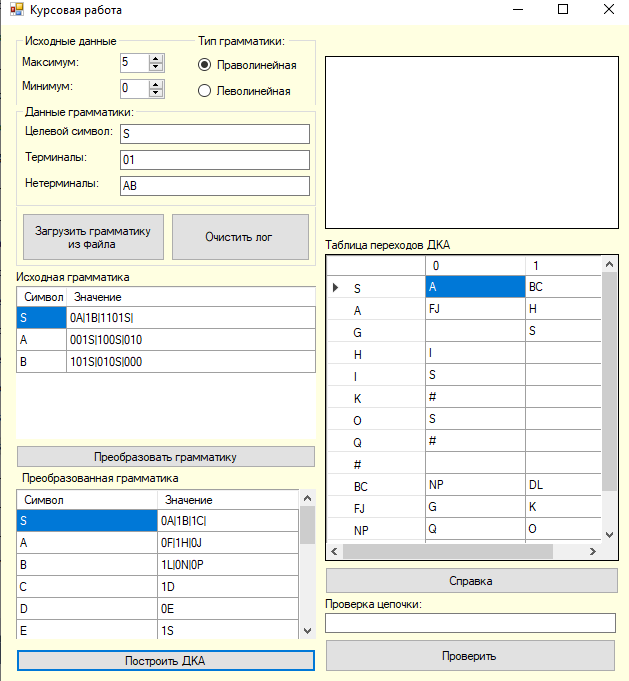


Рисунок 4. Построение ДКА

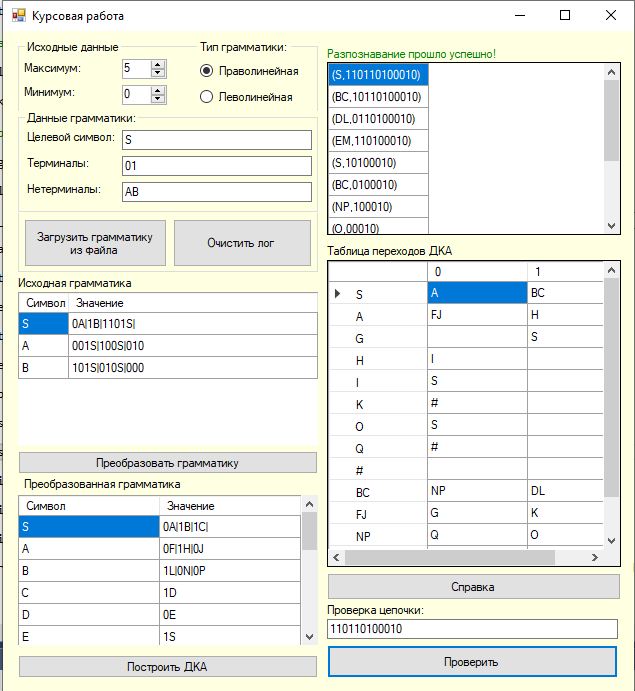


Рисунок 5. Проверка цепочки