Einführung

in

Automatentheorie

und

Formale Sprachen

Version 2020

Autor: Dieter Lindenberg

Inhalt

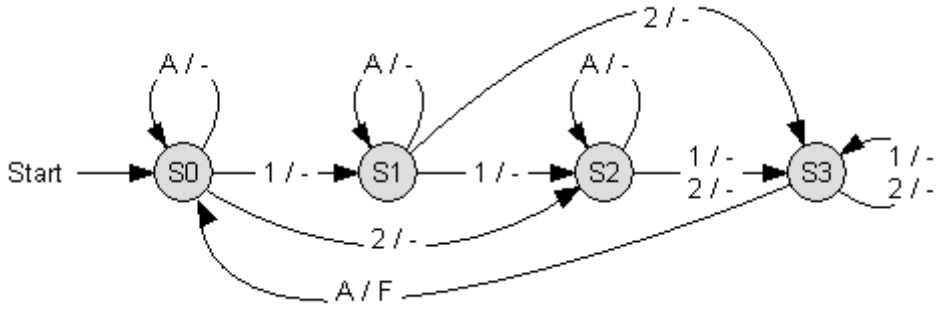
Transduktor

Ein Fahrscheinautomat verkauft Fahrkarten im Wert von 3€. Der Automat sei sehr einfach konstruiert. Es muss exakt der Wert von 3€ eingeworfen werden, denn zu viel eingeworfenes Geld wird einbehalten. Man kann nur 1€- oder 2€-Münzen einwerfen. Nach Drücken des Ausgabeknopfes erhält man den Fahrschein.

Eingabealphabet E = {1, 2, A} entsprechend den Münzen bzw. dem Ausgabeknopf.
Ausgabealphabet A = {F, -} entspricht der Fahrkarte bzw. der leeren Ausgabe. Die leere Ausgabe wird oft auch mit dem Leerstring ‘‘ oder mit dem Buchstaben n *(nothing)* gekennzeichnet.

Es gibt vier innere ***Zustände*** S_0, S_1, S_2, S_3 des Automaten:
S_0: ***Anfangszustand***,
S_1: 1€ wurde bereits eingeworfen,
S_2: 2€ wurden bereits eingeworfen,
S_3: 3€ oder mehr wurden eingeworfen. Betätigen der Ausgabetaste wird erwartet.

Zustandsmenge S = {S_0, S_1, S_2, S_3}



Jeder Zustand wird benannt, durch einen Kreis symbolisiert und bildet so einen Knoten des ***Zustandsdiagramms***. Der Anfangszustand wird durch einen zusätzlichen Eingangspfeil gekennzeichnet.

Von jedem Zustand aus führen Pfeile (die sog. Kanten) zu den möglichen Folgezuständen. Diese Kanten werden mit den Eingabezeichen beschriftet, die den entsprechenden Übergang auslösen. Falls mehrere Eingabezeichen das System in denselben Folgezustand überführen, werden diese Zeichen oft mit dem Oder-Zeichen $\vee$ verknüpft.

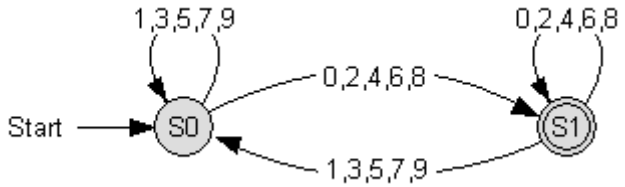
Produziert der Automat auch Ausgabezeichen, so müssen diese ebenfalls an der Kante, durch „/“ von den Eingabezeichen getrennt, angezeigt werden.

Zur Eingabefolge 1A2A1 gehören die Zustandsfolge $S_0S_1S_1S_3S_0S_1$ und die Ausgabefolge nnnFn

Weil dieser Automat Zeichen aus einem Eingabealphabet in Zeichen aus einem Ausgabealphabet überführt, nennt man ihn auch ***Transduktor***. Ein Transduktor wird auch ***endlicher Automat mit Ausgabe*** genannt.

Akzeptor

Der folgende Automat besitzt keine Ausgaben, dafür aber einen besonders gekennzeichneten sog. ***Endzustand***.



Wenn man hier eine gerade, natürliche Dezimalzahl eingibt, landet man im Endzustand S1. Das Problem der führenden Nullen ist hier unwichtig.

Endzustände erkennt man an einem Doppelkreis.

Die Eingaben 1434 und 20010 enden beide mit dem Endzustand. Man sagt, diese Zahlen werden vom Automaten **akzeptiert** und den zugehörigen Automaten nennt man ***Akzeptor***.

Die Menge aller Eingabefolgen, die von einem Akzeptor akzeptiert werden, nennt man die ***Sprache L(A)*** des Automaten. Dabei steht ***L*** für *language* und ***A*** für Akzeptor.

Automaten-Definitionen

Offensichtlich gibt es unterschiedliche Automaten. Die obigen beiden sind nur zwei Beispiele dafür. Leider werden die Automaten in der Fachliteratur nicht einheitlich definiert. Dieses Skript orientiert sich an den Vorgaben für das Zentral-Abitur in NRW.

- Ein Automat heißt ***endlich***, wenn das Eingabealphabet E, das Ausgabe-alphabet A und die Zustandsmenge S (nichtleer und) endlich sind.

- Ein Automat heißt ***deterministisch***, wenn es für jede Kombination von Eingabezeichen und Zustand nur eine mögliche Kombination von Ausgabezeichen und Folgezustand gibt.

- Ein ***Mealy-Automat*** ist ein Transduktor, dessen Ausgabe von seinem Zustand **und** seiner Eingabe abhängt. Anschaulich bedeutet das, dass jeder Kante im Zustandsdiagramm ein Ausgabewert zugeordnet wird. Der Name geht auf *George H. Mealy (1927 – 2010)* zurück.

- Ein ***Moore-Automat*** ist ein Transduktor, dessen Ausgabe nur von seinem Zustand abhängt. Anschaulich bedeutet das, dass im Zustandsdiagramm an den Kanten keine Ausgabewerte stehen. Der Name geht auf *Edward F. Moore (1925-2003)* zurück.

Unter einem ***Mealy-Automaten*** versteht man ein 6-Tupel
$M = (E, A, S, s_0, u, g)$, welches beschrieben wird durch:

$E = \{e_1, e_2, \ldots, e_r\}$	das endliche Eingabealphabet
$A = \{a_1, a_2, \ldots, a_m\}$	das endliche Ausgabealphabet
$S = \{s_1, s_2, \ldots, s_n\}$	die endliche Zustandsmenge
$s_0 \in S$	den Anfangszustand
$u: E \times S \to S$	die Übergangsfunktion
$g: E \times S \to A^*$	die Ausgabefunktion

Dabei versteht man unter A^* die Menge aller Worte, die mit dem Ausgangsalphabet A erzeugt werden können, einschließlich des leeren Wortes.

Alle diese Angaben sind schon im Zustandsdiagramm enthalten und müssen eigentlich nicht mehr extra herausgeschrieben werden. Wir wollen deshalb diese Formalitäten nicht allzu wichtig nehmen. Allerdings sollten sie z.B. für das Zentralabitur bekannt sein.

Unter einem ***Moore-Automaten*** versteht man ein 6-Tupel
$M = (E, A, S, s_0, u, g)$, welches beschrieben wird durch:

$E = \{e_1, e_2, \ldots, e_r\}$	das endliche Eingabealphabet
$A = \{a_1, a_2, \ldots, a_m\}$	das endliche Ausgabealphabet
$S = \{s_1, s_2, \ldots, s_n\}$	die endliche Zustandsmenge
$s_0 \in S$	den Anfangszustand
$u: E \times S \rightarrow S$	die Übergangsfunktion
$g: S \rightarrow A^*$	die Ausgabefunktion

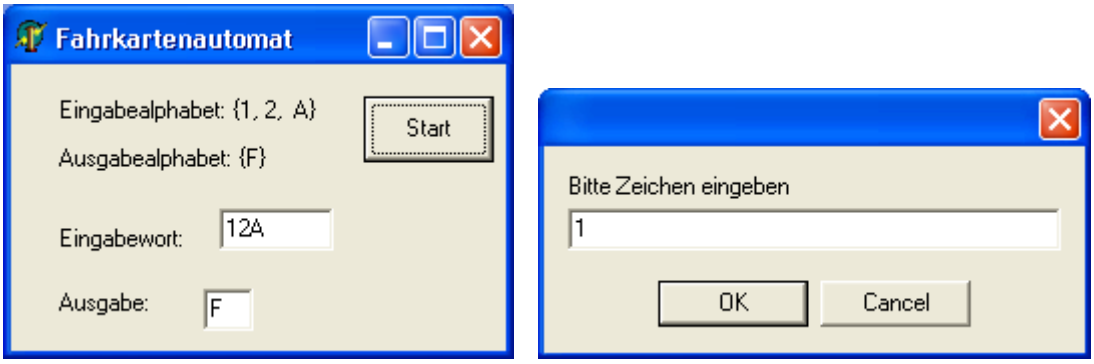
Unter einem ***Akzeptor-Automaten*** versteht man ein 5-Tupel
$M = (E, S, s_0, u, F)$, welches beschrieben wird durch:

$E = \{e_1, e_2, \ldots, e_r\}$	das endliche Eingabealphabet
$S = \{s_1, s_2, \ldots, s_n\}$	die endliche Zustandsmenge
$s_0 \in S$	den Anfangszustand
$u: E \times S \rightarrow S$	die Übergangsfunktion
$F \subset S$	eine Menge von Endzuständen

Bemerkung: die einzelnen Bezeichnerbuchstaben sind in der Literatur leider auch nicht einheitlich. Selbst im Zentral-Abitur NRW können die Bezeichner von Jahr zu Jahr unterschiedlich sein.

Programmierung von Automaten in DELPHI

Im Folgenden werden verschiedene Möglichkeiten gezeigt, den anfangs vorgestellten Fahrkarten-Automaten zu programmieren (Mealy-Automat).




```
unit MMain;    {Mealy-Automat, Version mit GOTO und Label}
……..
var  Main: TMain;
     ch: CHAR;
     Eingabealphabet: SET of CHAR;


Implementation ….


Function TMain.LiesZeichen: CHAR;
VAR wort : STRING;
BEGIN
 Repeat
  wort := InputBox('', 'Bitte Zeichen eingeben', '');
        // Parameter: Überschrift, Aufforderungstext, voreingestellter Wert
  IF wort = '' THEN wort := 'X'; // Dummy
  IF NOT (wort[1] in Eingabealphabet)
  THEN Showmessage('Zeichen war nicht im Eingabealphabet')
 UNTIL wort[1] in Eingabealphabet;
 EEingabe.text := EEingabe.text + wort[1];
 RESULT := wort[1]
END;
```

```
procedure TMain.BtStartClick(Sender: TObject);
Label s0, s1, s2, s3;
begin
  EEingabe.Text := '';
  EAusgabe.Text := '';
  Eingabealphabet := ['1', '2', 'A'];
  s0: ch := LiesZeichen;
      EAusgabe.Text := '';
      CASE ch of
        '1': GOTO s1;
        '2': GOTO s2;
        'A': GOTO s0
      END;
  s1: ch := LiesZeichen;
      CASE ch of
        '1': GOTO s2;
        '2': GOTO s3;
        'A': GOTO s1
      END;
  s2: ch := LiesZeichen;
      CASE ch of
        '1': GOTO s3;
        '2': GOTO s3;
        'A': GOTO s2
      END;
  s3: ch := LiesZeichen;
      CASE ch of
        '1': GOTO s3;
        '2': GOTO s3;
        'A': BEGIN EAusgabe.Text := 'F'; GOTO s0 END
      END;
 END; {of procedure}
end.
```

```
unit MMain; {Mealy-Automat, Version mit Übergangs- und Ausgabefunktion}
interface…….
Uses……..
CONST Eingabealphabet = ['1','2','A'];

type
  TZustand = (s0, s1, s2, s3);
  TMain = class(TForm)
    ……….
    function g(s: TZustand; c: CHAR): STRING;
    function u(s: TZustand; c: CHAR): TZustand;
  end;

var  Main: TMain;
     ch: CHAR;
     s:  TZustand;
implementation……..

function TMain.u(s: TZustand; c: CHAR): TZustand;
// Übergangsfunktion
BEGIN
 CASE s of
   s0: Case c of
         '1': u := s1;
         '2': u := s2;
         'A': u := s0
       End;
   s1: Case c of
         '1': u := s2;
         '2': u := s3;
         'A': u := s1
       End;
   s2: Case c of
         '1': u := s3;
         '2': u := s3;
         'A': u := s2
       End;
   s3: Case c of
         '1': u := s3;
         '2': u := s3;
         'A': u := s0
       End
  END  {of Case s}
END;
```

```
Function TMain.g(s: TZustand; c: CHAR): STRING;
// Ausgabefunktion
BEGIN
 g := '';
 IF (s = s3) AND (c = 'A') THEN g := 'F'
END;

procedure TMain.BtStartClick(Sender: TObject);
begin
  EEingabe.Text := '';
  EAusgabe.Text := '';
  s := s0;
  REPEAT
    ch := LiesZeichen;
    EAusgabe.Text := g(s,ch);
    s := u(s,ch)
  UNTIL 0=1
  {Man könnte auch einen definierten Schlusszustand einführen und dann
 ....UNTIL s = Schlusszustand}
END; {of procedure}

end.
```

Die Übergangsfunktion lässt sich sehr schön tabellarisch darstellen:

	1	2	A
S_0	S_1	S_2	S_0
S_1	S_2	S_3	S_1
S_2	S_3	S_3	S_2
S_3	S_3	S_3	S_0

Genauso kann man natürlich die Ausgabefunktion darstellen:

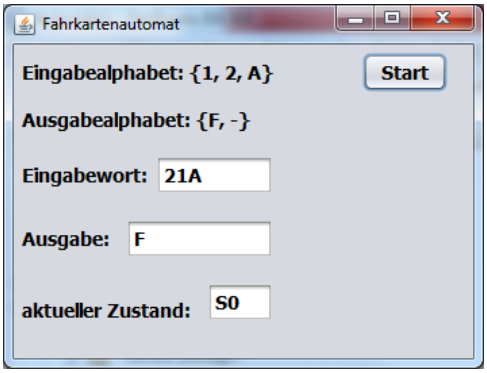
	1	2	A
S_0	n	n	n
S_1	n	n	n
S_2	n	n	n
S_3	n	n	F

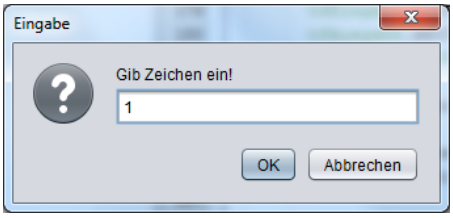
Man kann auch beide Darstellungen zur sog. ***Zustandstabelle*** kombinieren:

	1	2	A
S_0	S_1, n	S_2, n	S_0, n
S_1	S_2, n	S_3, n	S_1, n
S_2	S_3, n	S_3, n	S_2, n
S_3	S_3, n	S_3, n	S_0, F

Programmierung von Automaten in Java

Im Folgenden wird eine Möglichkeit gezeigt, den anfangs vorgestellten Fahrkarten-Automaten zu programmieren (Mealy-Automat).





Hinweis: Das Formular für die Eingabeaufforderung erscheint üblicherweise örtlich direkt auf dem eigentlichen Hauptformularblatt.

```
import javax.swing.*;    // für Dialogformulare
public class Mealy extends javax.swing.JFrame {

 enum Zustand {S0, S1, S2, S3};
 Zustand s = Zustand.S0;
 String Eingabealphabet  = "12A";

 public Mealy() {
   initComponents();
   tfEingabe.setText("");
   tfAusgabe.setText("");
   tfZustand.setText("S0");
 }


 private Zustand u(Zustand s, char c)  {
 // Übergangsfunktion
   Zustand sHilf = Zustand.S0;
   switch (s)  {
     case S0: switch (c)  {
                case '1': sHilf = Zustand.S1; break;
                case '2': sHilf = Zustand.S2; break;
                case 'A': sHilf = Zustand.S0;
           }; break;
```

```
    case S1: switch (c)  {
               case '1': sHilf = Zustand.S2; break;
               case '2': sHilf = Zustand.S3; break;
               case 'A': sHilf = Zustand.S1;
             }; break;

   case S2: switch (c)  {
              case '1': sHilf = Zustand.S3; break;
              case '2': sHilf = Zustand.S3; break;
              case 'A': sHilf = Zustand.S2;
            }; break;

   case S3: switch (c)  {
              case '1': sHilf = Zustand.S3; break;
              case '2': sHilf = Zustand.S3; break;
              case 'A': sHilf = Zustand.S0;
            };
  }

  return sHilf;
 }

private String g(Zustand s, char c)  {
 // Ausgabefunktion
  String ergebnis = "";
  if (s == Zustand.S3 && c == 'A') ergebnis = "F";
  return ergebnis;
}


private char liesZeichen()   {
  String wort;
  int n;
  char ch0;
  do {
      wort = JOptionPane.showInputDialog(this, "Gib
                                        Zeichen ein!");
      if (wort == "") wort = "X"; // Dummy
      ch0 = wort.charAt(0);
      n = Eingabealphabet.indexOf(ch0);
      if (n < 0) JOptionPane.showMessageDialog(this,
            "Zeichen war nicht im Eingabealphabet");
  }
  while (n < 0);
```

```
    tfEingabe.setText(tfEingabe.getText() + ch0);
    return ch0;
  }



  private void btStartMouseClicked(java…) {
    char ch;
    tfEingabe.setText("");
    tfAusgabe.setText("");
    s = Zustand.S0;
    do {
      ch = liesZeichen();
      tfAusgabe.setText(g(s,ch));
      s = u(s, ch);

      switch (s)  {
         case S0: tfZustand.setText("S0"); break;
         case S1: tfZustand.setText("S1"); break;
         case S2: tfZustand.setText("S2"); break;
         case S3: tfZustand.setText("S3");
     }
    }
    while (0 != 1);
  }


}
```

Die Übergangsfunktion lässt sich sehr schön tabellarisch darstellen:

	1	2	A
S_0	S_1	S_2	S_0
S_1	S_2	S_3	S_1
S_2	S_3	S_3	S_2
S_3	S_3	S_3	S_0

Genauso kann man natürlich die Ausgabefunktion darstellen:

	1	2	A
S_0	n	n	n
S_1	n	n	n
S_2	n	n	n
S_3	n	n	F

Man kann auch beide Darstellungen zur sog. ***Zustandstabelle*** kombinieren:

	1	2	A
S_0	S_1, n	S_2, n	S_0, n
S_1	S_2, n	S_3, n	S_1, n
S_2	S_3, n	S_3, n	S_2, n
S_3	S_3, n	S_3, n	S_0, F

Aufgaben für Transduktoren

1. Der Fahrscheinautomat soll nun durch einen zusätzlichen sog. ***Fehlerzustand f*** erweitert werden. In diesen gelangt man, wenn zu viel Geld eingeworfen wurde. Aus diesem Fehlerzustand gelangt man nicht mehr heraus. Zeichne den Zustandsgraphen dieses Mealy-Automaten und gib die Übergangsfunktion in Form einer Tabelle an!

2. Was bewirkt ein Mealy-Automat mit folgender Übergangstabelle? Vergleiche zu den folgenden Eingaben 010111, 111000111 und 1100110011 jeweils die zugehörigen Ausgaben! Zeichne den Zustands-graphen!

	0	1
S_0	S_2,0	S_1,0
S_1	S_2,1	S_1,1
S_2	S_2,0	S_1,0

3. Entwickle einen Mealy-Automaten mit dem Eingabealphabet E = {a, b, c, d}, der bei der Ausgabe die eingegebenen Zeichen zyklisch vertauscht (Cäsar-Verschlüsselung). Beispiel: badcadcc → cbadbadd

4. Entwickle einen Mealy-Automaten mit vier Zuständen, dem Eingabe-alphabet E = {0, 1, 2} und dem Ausgabealphabet A = {0, 1, 2, 3, 4, n}, der zwei Ziffern addiert und die Summe ausgibt. Beispiele: 01→1, 22→4. Danach soll sich der Automat wieder im Startzustand befinden.

5. In einigen heute üblichen Textverarbeitungsprogrammen ist ein sog. Smiley-Erkennungs-Automat eingebaut.

Zeichenfolge	Smiley
:-)	☺
:)	☺
:-(	☹
:(	☹

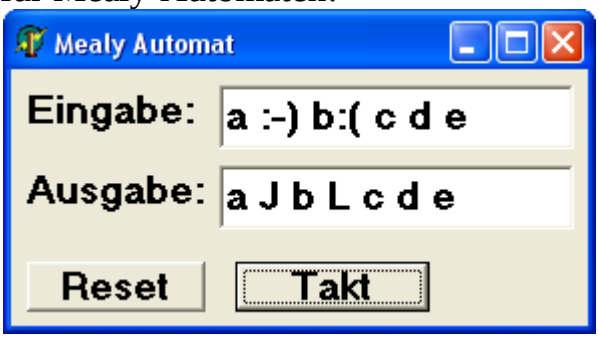
Dieser Automat soll also einen Eingabezeichenstrom lesen und – bis auf die Smileys – unverändert ausgeben.
Um unseren Automaten nicht zu kompliziert werden zu lassen, wollen wir als Eingabezeichen nur die in diesen vier Smileys vorkommenden Zeichen und zusätzlich das **x** zulassen, das für alle anderen Zeichen steht: E = {:, -,), (, x}
Das Ausgabealphabet wäre dann A = {:, -,), (, x, ☺, ☹, n}

Hinweis: Wähle für das Ausgabefeld die Schriftart *Wingdings*. Dort haben die Smiley-Gesichter die Codenummern 74 (lachendes Smiley) und 76 (weinendes Smiley).
Der Befehl **`tfAusgabe.setText(""+ (char) 74)`** liefert dann das lachende Smiley.
Problem: *Wingdings* kennt keine normalen Zeichen.

Programmiere diesen Smiley-Erkennungsautomaten (Typ Mealy)!

6. Erweitere den Smiley-Erkenner durch die Zeichenfolge :-| = 😐
Hinweis: das neutrale Smiley-Gesicht hat in *Windings* die Codenummer 75.

7. Erstelle ein neues Programm-Layout ***GUI*** (*graphic user interface*) für Mealy-Automaten!

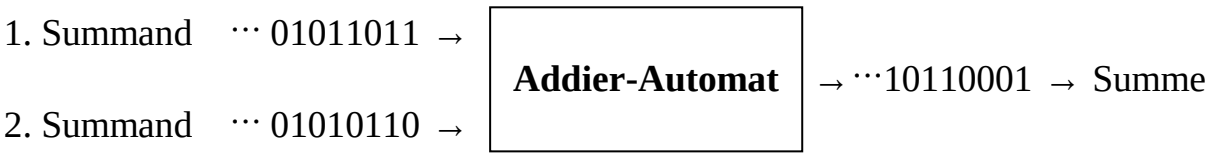


Der Benutzer gibt eine Zeichenfolge in das Eingabetextfeld ein.
Jedes Mal, wenn der Button *Takt* gedrückt wird, wird ein Eingabezeichen verarbeitet.

8. Entwirf einen Mealy-Automaten zur Autokorrektur, der alle eingegebenen Zeichen wieder ausgibt, allerdings **kontextunabhängig**. Zusätzlich soll er
a) nur die Zeichenfolge „daß“ in „dass“ ändern. Hinweis: Man kann auch „ss“ als ein einziges Ausgabezeichen definieren.
b) die Zeichenfolge (bzw. Buchstabendreher) „dre“ in „der“ ändern.
Bemerkung: Damit würden (leider) auch der Name *Andreas* und das Wort *drei* geändert werden.

Hinweis: Um die Zustandsgraphen bzw. insbesondere deren Kantenzahl klein zu halten, benutze das Zeichen x, welches stellvertretend für „alle anderen“ Zeichen steht.

9. Serienaddier-Automat



Da ein Automat immer nur ein einziges Zeichen interpretiert, der Serienaddierer aber zwei Binärzahlen am Eingang vorfindet, müssen wir jeweils zwei Binärziffern mithilfe eines neuen Zeichens kodieren. Wir wählen willkürlich:
00 → N (**N**ull), 01 → E (**E**ins), 10 → A (**A**uch Eins), 11 → Z (**Z**wei).
Also das Eingabealphabet ist {N, E, A, Z}. Das Ausgabealphabet besteht einfach aus den beiden Dualziffern {0, 1}.
Die Eingabe geschieht von rechts nach links. Im obigen Beispiel würde also zuerst ein A eingegeben, danach ein Z, dann ein E, dann ein A, dann ein Z, dann ein N usw.
Bei jedem Schritt kann also der durch die Summation entstehende Übertrag mit einbezogen werden.

Der Automat soll unendlich lange Summanden addieren. Es braucht also keine Ende-Bedingung berücksichtigt zu werden. Daher handelt es sich um einen Mealy-Automaten.

Zeichne das Zustandsdiagramm und schreibe das entsprechende Programm!

10. Ampelschaltung

Eine Ampel hat üblicherweise den Zyklus rot, rot-gelb, grün, gelb. Im Normalmodus sind alle Phasen gleich lang (beispielsweise 20 Sekunden). Als Eingangssignal gibt es das Taktzeichen ***T***, welches automatisch erzeugt und nach jeweils exakt 20 Sekunden in den Automaten eingegeben wird. Es bewirkt einen Wechsel zur nächsten Phase. Außerdem gibt es noch die Eingabezeichen ***R*** (ab jetzt doppelt so lange Rotphase) und das Zeichen ***N*** (Wechsel zum Normalmodus). Die Eingaben *R* und *N* bewirken keinen Phasenwechsel, allerdings durchaus einen Zustandswechsel.
Mehrere Eingaben von *R* direkt nacheinander bewirken dasselbe wie die Eingabe eines einzigen *R*. Analoges gilt für *N*.
Die Eingaben *R* und *N* können zu beliebigen Zeiten erfolgen. Die einzelnen Farbphasen werden aber nur durch die Eingabe von *T* gewechselt.

Jeder Zustand ist mit allen drei Lampen der Ampel durch Kabel verbunden. Das bedeutet, dass die Ampel entsprechend leuchtet, wenn sich der Automat in einem bestimmten Zustand befindet.
Zeichne das Zustandsdiagramm dieses Moore-Automaten (die Ausgabe braucht nicht eingezeichnet zu werden) und schreibe das entsprechende Programm!

11. Prüfbitgenerator

Entwickle einen Prüfbitgenerator. Da die meisten Zeichensätze weniger als 128 Zeichen enthalten, können sie als siebenstellige Dualzahlen codiert werden. Rechner arbeiten aber mit Bytes. Das eigentlich überflüssige achte Bit kann als ein Prüfbit benutzt werden, das bei der Datenübertragung zur Fehlerermittlung benutzt wird. Eine Möglichkeit, Prüfbits zu erzeugen, ist die, an das zu übertragende Zeichen jeweils eine 1 oder 0 so anzuhängen, dass das gesendete Byte immer eine gerade Anzahl von Einsen enthält. Beschreibe einen solchen Prüfbitgenerator als Mealy-Automaten, der nacheinander 7 Bits liest und dann nur das achte Prüfbit ausgibt. Anschließend geht er wieder in den Anfangszustand.

12. Zählschaltung

Entwickle eine Zählschaltung modulo 3 als Moore-Automat (also einen Zähler, der bis zwei zählen kann: 0-1-2-0-1-2-…, die Zahlen entsprechen den „Zählerstand-Ausgaben“. Es gibt also nur 3 Zählerstand-Ausgaben (allerdings hat der zu konstruierende Automat in Teilaufgabe b mehr als 3 Zustände). Zusätzlich soll der Zähler durch die Eingabe des *Resetsignals* ***R*** in den Anfangszustand und damit in die Zählerstand-Ausgabe 0 zurückgesetzt werden.

Der Zähler ändert seinen Zustand, wenn
a) ein Taktsignal *T* eintrifft.
b) die Eingabe von 0 auf 1 wechselt.
Gib für beide Teilaufgaben jeweils das Eingabealphabet und das Zustandsdiagramm des Moore-Automaten an!

13. J-K-Flip-Flop

Ein sog. J-K-Flip-Flop ist ein 1-Bit-Speicher für 0 oder 1. Es kann in folgender Weise angesprochen werden:

Eingabezeichen: S: eine 1 speichern (**S**et)
R: eine 0 speichern (**R**eset)
W: Zustand wechseln - von 0 auf 1 oder von 1 auf 0
B: Bewahren (Zustand beibehalten)
Ausgabezeichen: 0 oder 1 entsprechend der gespeicherten Ziffer.
Gib das Eingabealphabet und das Zustandsdiagramm des Moore-Automaten an!

14. Seriell-Parallel-Konverter

Es gibt prinzipiell zwei Arten der Datenübertragung etwa zwischen Computer und Drucker: entweder werden die Daten nacheinander, also seriell, auf eine einzige Datenleitung übertragen, oder es gibt mehrere parallele Datenleitungen zwischen den Geräten, so dass alle Datenbits gleichzeitig übertragen werden. Entwickle einen *3-Bit-Seriell-Parallel-Konverter*, der **immer wieder** eine 3-Bit-Zahl einliest (das höchstwertige Bit MSB zuerst) und danach im entsprechenden Endzustand die zugehörige Dezimalzahl ausgibt. Gib sowohl einen Moore-Automaten als auch einen Mealy-Automaten dafür an!

15. Einerkomplement

Entwickle einen Automaten, der das 1-Komplement von Binärzahlen ausgibt. Das Ausgabealphabet sei {0, 1, n}.
Beispiel: Eingabe 1001 Ausgabe 0110
Gib sowohl einen Moore-Automaten als auch einen Mealy-Automaten dafür an!

16. Entwickle folgenden Mealy-Automaten

δ	a	b	c
q_0	q_1	q_0	q_2
q_1	q_2	q_1	q_3
q_2	q_0	q_0	q_2
q_3	q_2	q_2	q_3

λ	a	b	c
q_0	1	2	1
q_1	1	4	2
q_2	3	1	2
q_3	2	2	3

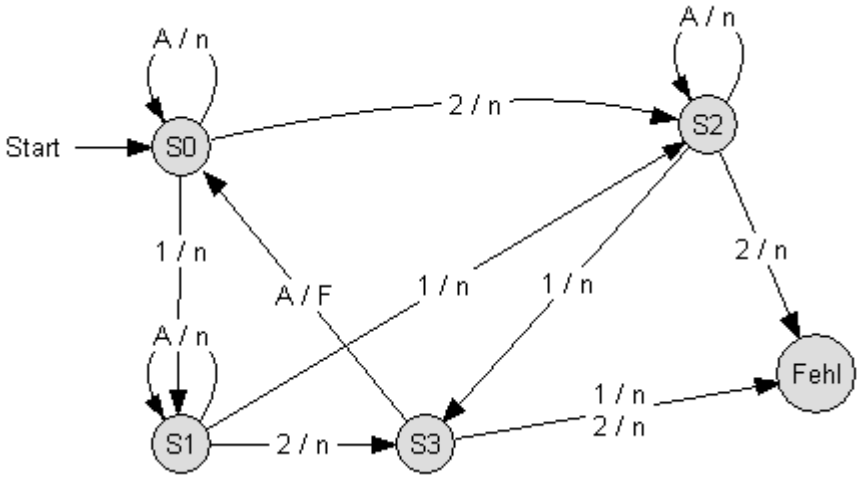
Hinweis: Der praktische Verwendungszweck der beschriebenen Maschine steht nicht im Vordergrund.

Bemerkung:
Die Anzahl der Zustände eines Automaten kann manchmal verkleinert, also optimiert werden. Zum Beispiel sind in einem Mealy-Automaten (und auch in Akzeptor-Automaten, welche später besprochen werden) zwei Zustände identisch, wenn alle ihre entsprechenden Ausgänge zu denselben Folgezuständen führen (allerdings darf dabei – im Falle von sog. Akzeptoren – nicht einer ein Endzustand und der andere kein Endzustand sein).

Dies ist schon ein spezielles Beispiel. Automaten lassen sich manchmal auch dann noch verkleinern, wenn die entsprechenden Ausgänge nicht direkt in denselben Folgezustand führen.

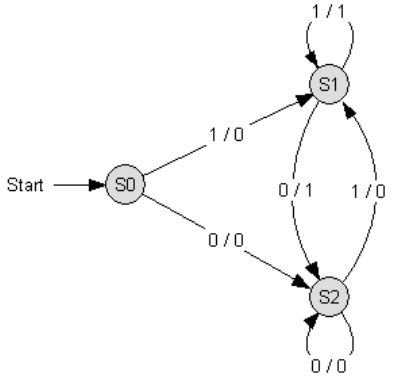
Lösungen

1.

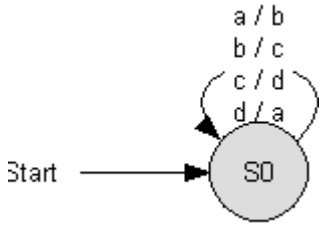


	1	2	A
S_0	S_1	S_2	S_0
S_1	S_2	S_3	S_1
S_2	S_3	f	S_2
S_3	f	f	S_0
f	f	f	f

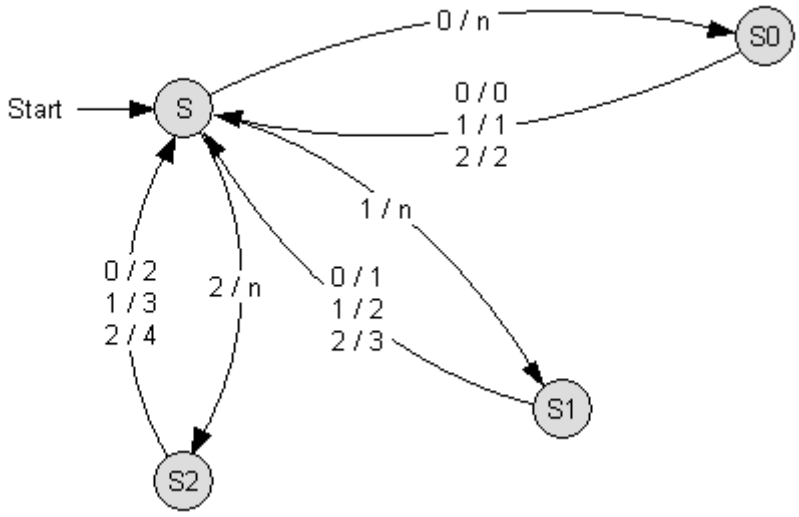
2. 010111 ↦ 001011,
111000 ↦ 011100
1100110011 ↦ 0110011001
Der Automat bewirkt eine
Shift Right Operation
bzw. eine Division durch 2.



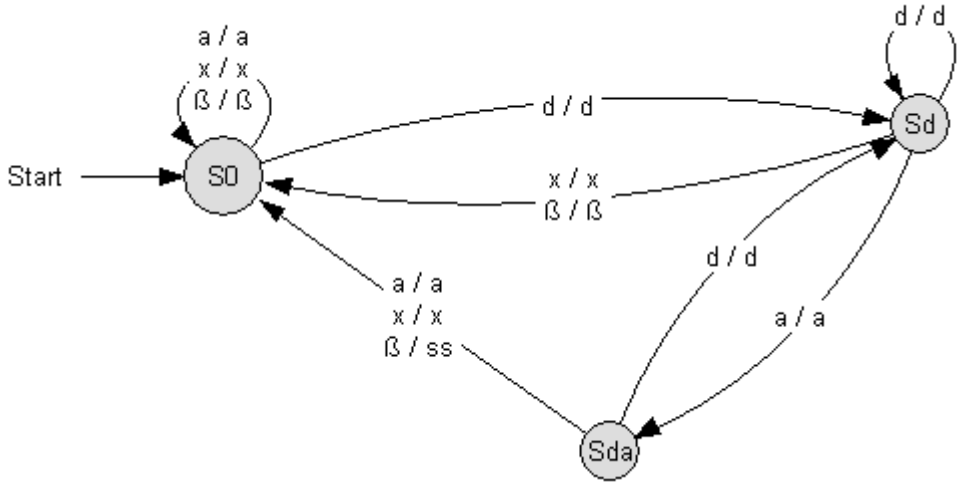
3.



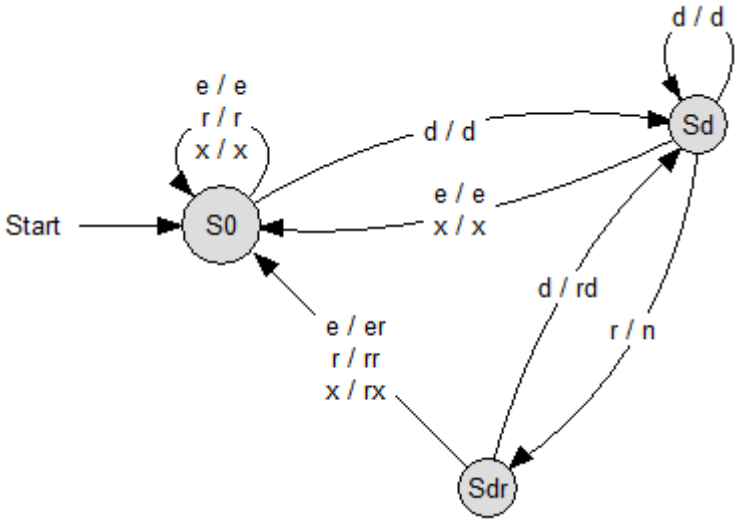
4.



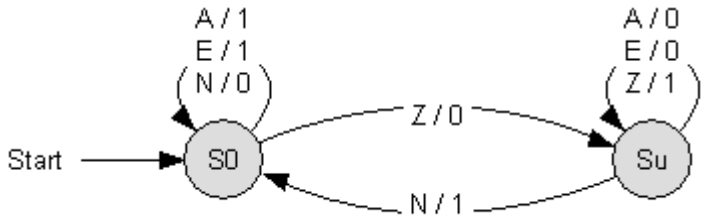
8.a) Das Zeichen x steht für alle anderen Buchstaben außer a, d, ß.



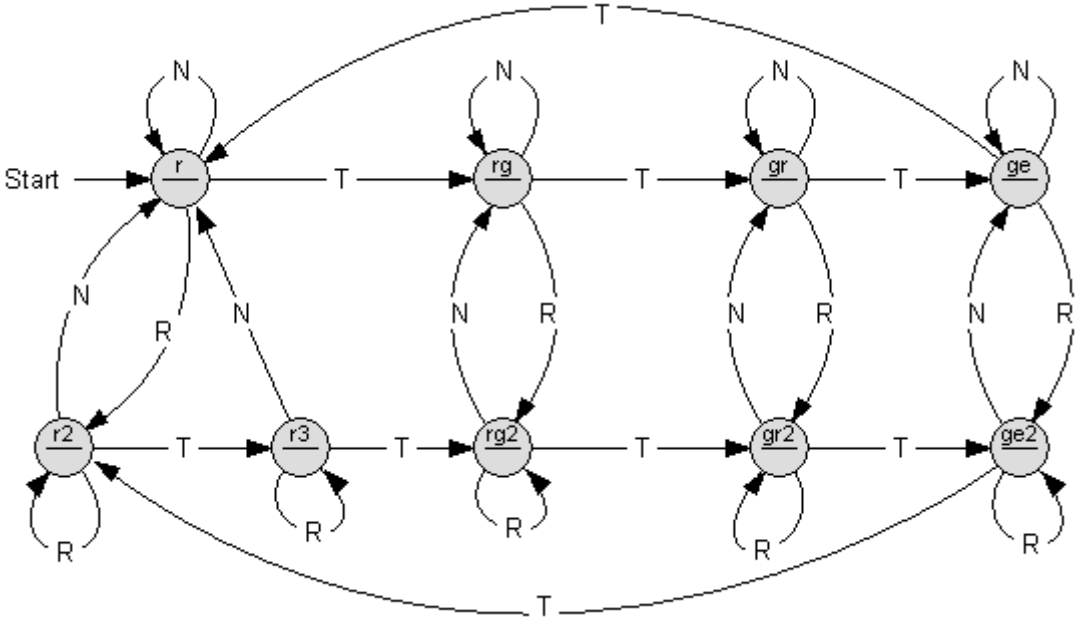
8.b) Das Zeichen x steht für alle anderen Buchstaben außer d, e, r



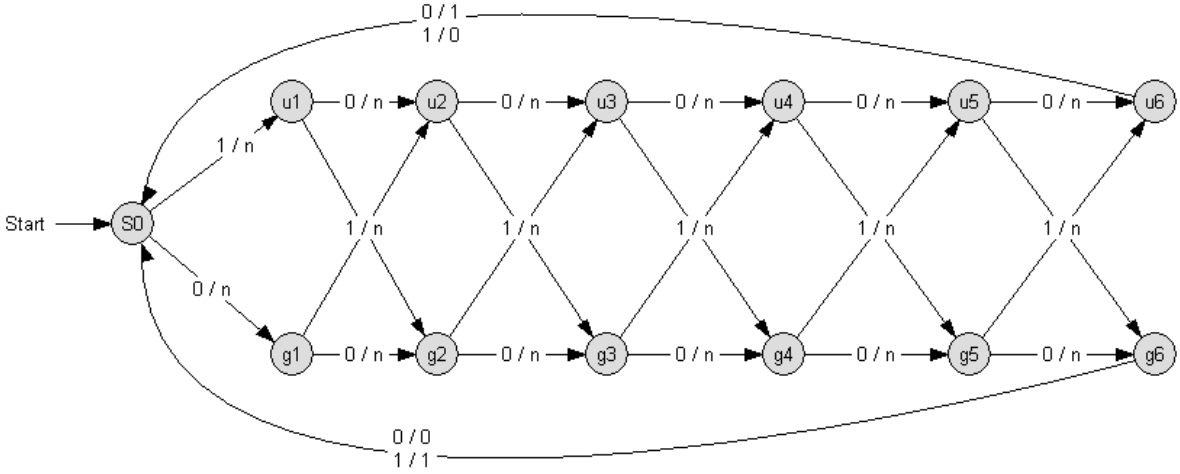
9.



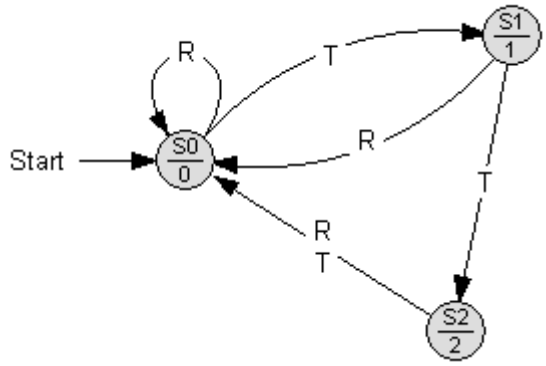
10. (Die Ausgabe wird in dieser Lösung nicht angegeben)



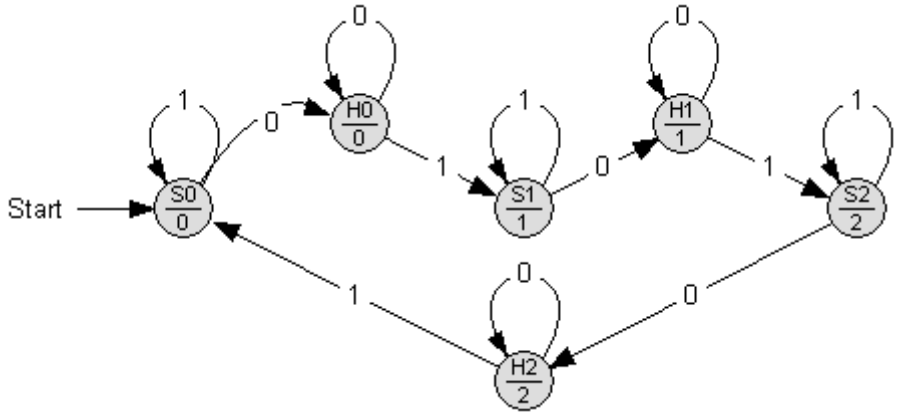
11.



12.a)

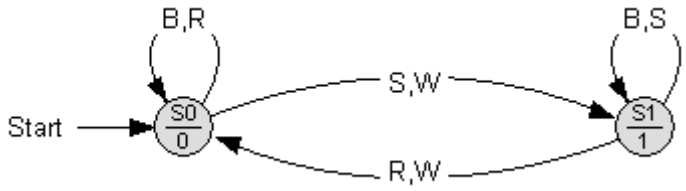


12.b)

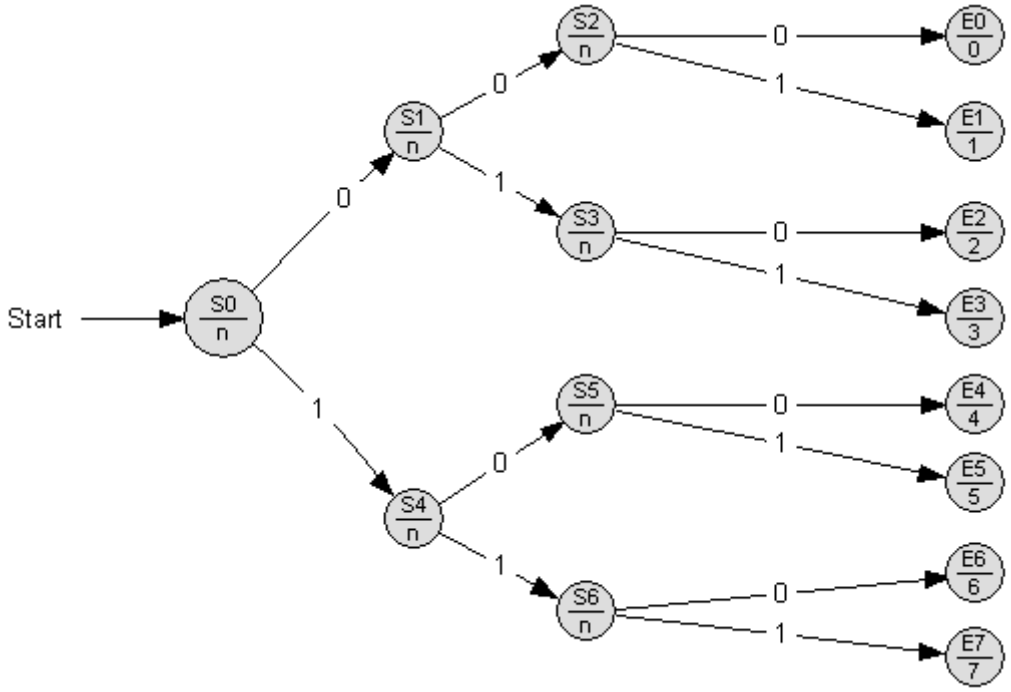


Die Reset-Übergänge wurden nicht eingezeichnet.

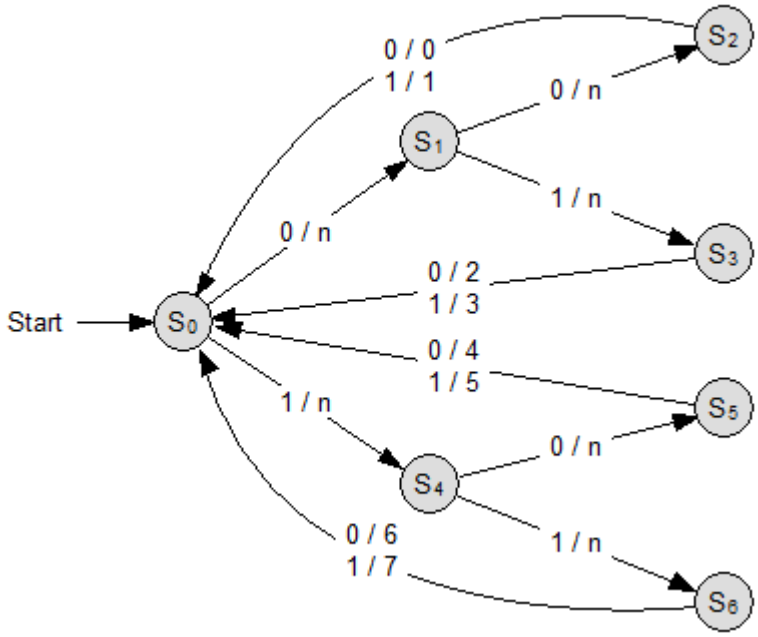
13. Eingabealphabet = {B, R, S, W}



14. a) Moore-Automat: Für jeden einzelnen der Zustände E0 bis E7 fehlen (nur aus Übersichtsgründen) im Zustandsgraphen noch die beiden Übergänge mit 0 in den Zustand S1 und mit 1 in den Zustand S4.

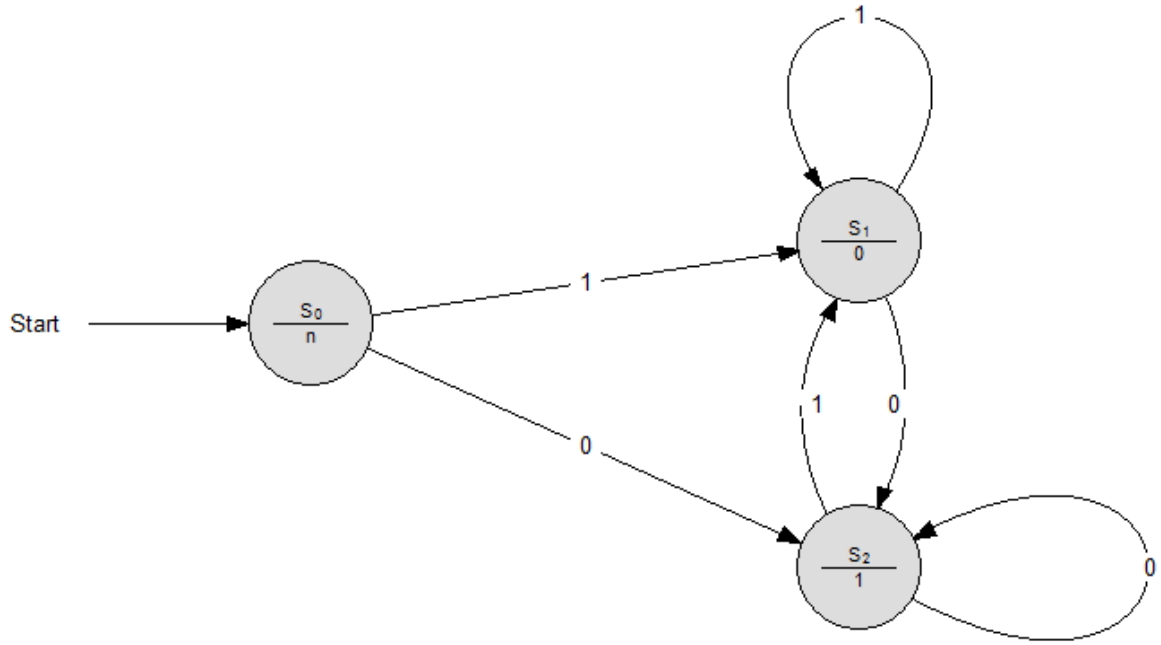


14. b) Mealy-Automat:

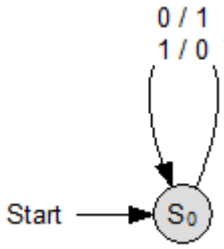


Beachte, dass hier der Mealy-Automat deutlich kleiner ist als der Moore-Automat!

15. a) Moore-Automat:



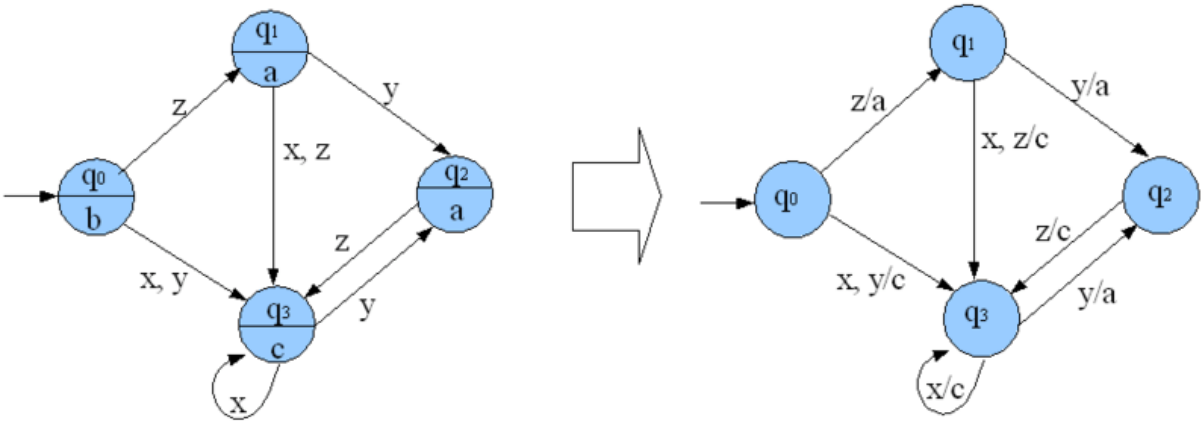
15. b) Mealy-Automat:



Beachte, dass auch hier der Mealy-Automat kleiner ist als der Moore-Automat!

Überführung eines Moore-in einen Mealy-Automaten

Jeder Moore-Automat lässt sich sehr leicht in einen äquivalenten Mealy-Automaten überführen. Dazu muss lediglich das Ausgabesymbol des Zielzustandes mit auf den Zustandsübergang des Mealy-Automaten geschrieben werden. Betrachte dazu das folgende Beispiel:



Problematisch ist hier nur, dass der Moore-Automat schon im Startzustand eine Ausgabe hat. Das ließe sich jedoch folgendermaßen lösen: „gib b aus und starte anschließend den Mealy-Automaten!“

Die beiden obigen Automaten sind sich also sehr ähnlich: Sie haben dieselbe Anzahl von Zuständen und dieselbe Anzahl von Kanten, an denen sogar dieselben Eingangselemente stehen. Lediglich die Ausgangselemente stehen beim Moore-Automaten in den Zuständen, beim Mealy-Automaten hingegen an den Knoten, die zum Zustand hinführen.

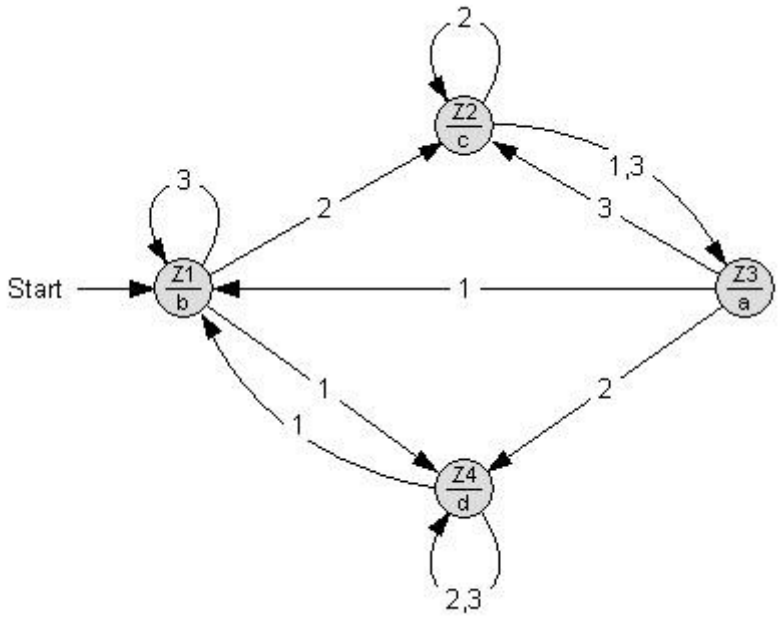
Interessante Folgerung: Angenommen, man sucht für eine bestimmte Aufgabe einen minimalen Automaten, in dem Sinne, dass die Anzahl der Zustände minimal ist. Dann kann es nicht sein, dass der minimale Moore-Automat kleiner ist als der minimale Mealy-Automat.

Aufgabe:
Gegeben sei ein Moore-Automat durch die Zustandsmenge $\{Z_1, Z_2, Z_3, Z_4\}$, das Eingangsalphabet {1, 2, 3}, das Ausgangsalphabet {a, b, c, d} und folgende Übergangstabelle:

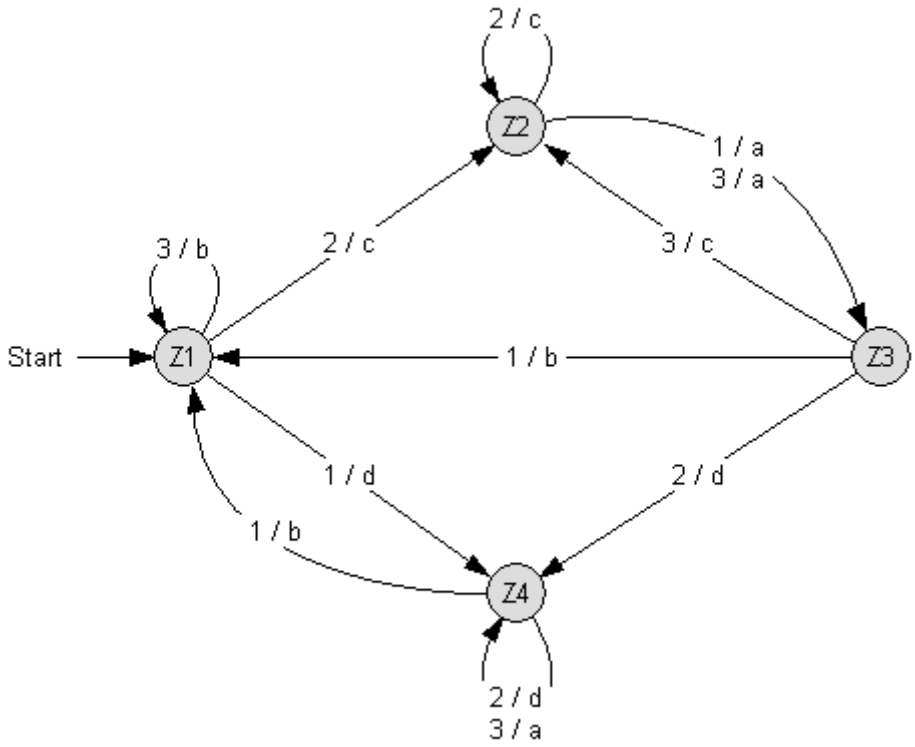
		1	2	3
Z_1	b	Z_4	Z_2	Z_1
Z_2	c	Z_3	Z_2	Z_3
Z_3	a	Z_1	Z_4	Z_2
Z_4	d	Z_1	Z_4	Z_4

Führe diesen Moore-Automaten in einen äquivalenten Mealy-Automaten über und zeichne beide Zustandsgraphen!

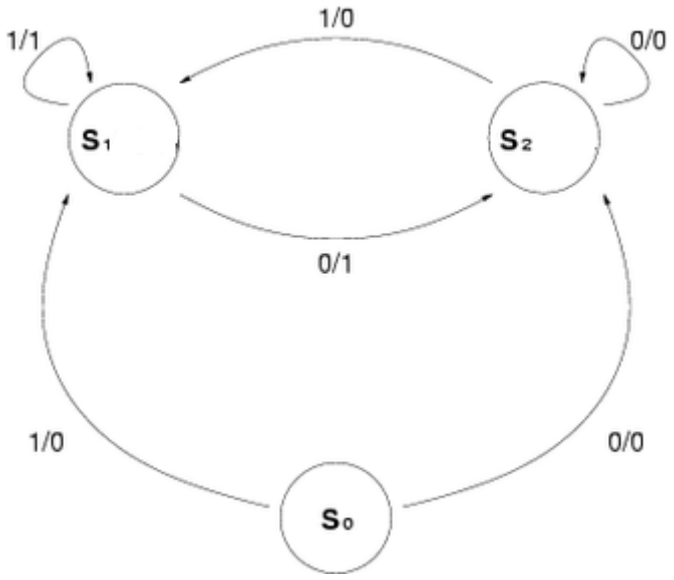
Lösung:
gegebener Moore-Automat:



äquivalenter Mealy-Automat:



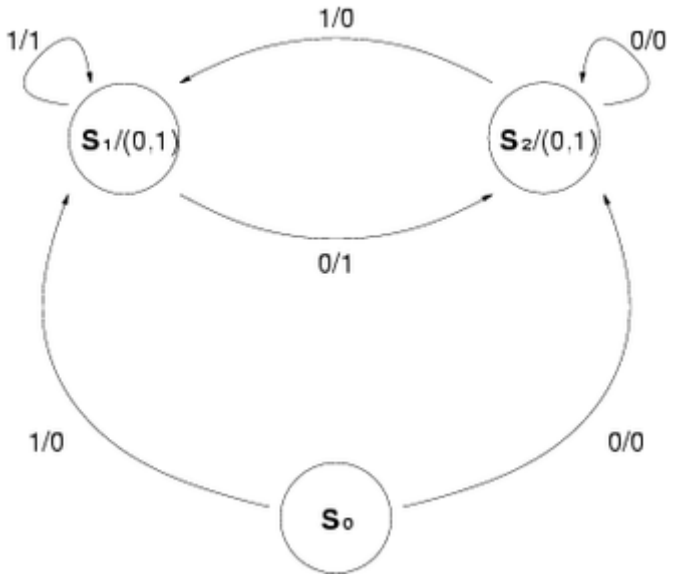
Überführung eines Mealy-in einen Moore-Automaten



Will man z.B. obigen Mealy-Automaten in einen Moore-Automaten umwandeln, kann man in folgenden drei Schritten vorgehen:

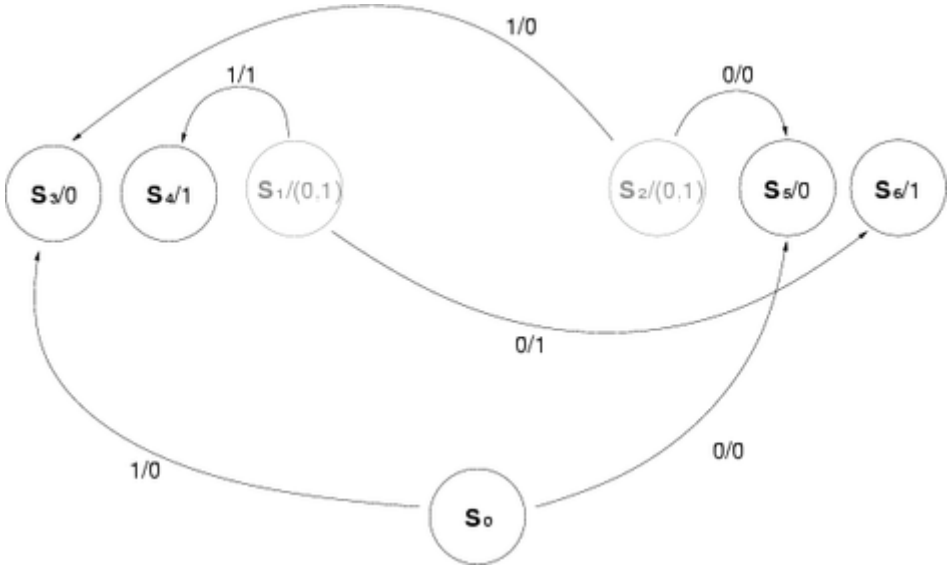
Schritt 1: Ausgabe in die Knoten schreiben

Für jede Kante wird die Ausgabe in den Zustand übertragen, auf dem die Kante endet. Hierbei stehen dann mindestens ein Ausgabewert, in der Regel sogar mehrere, verschiedene Ausgabewerte in einem Zustandsknoten.



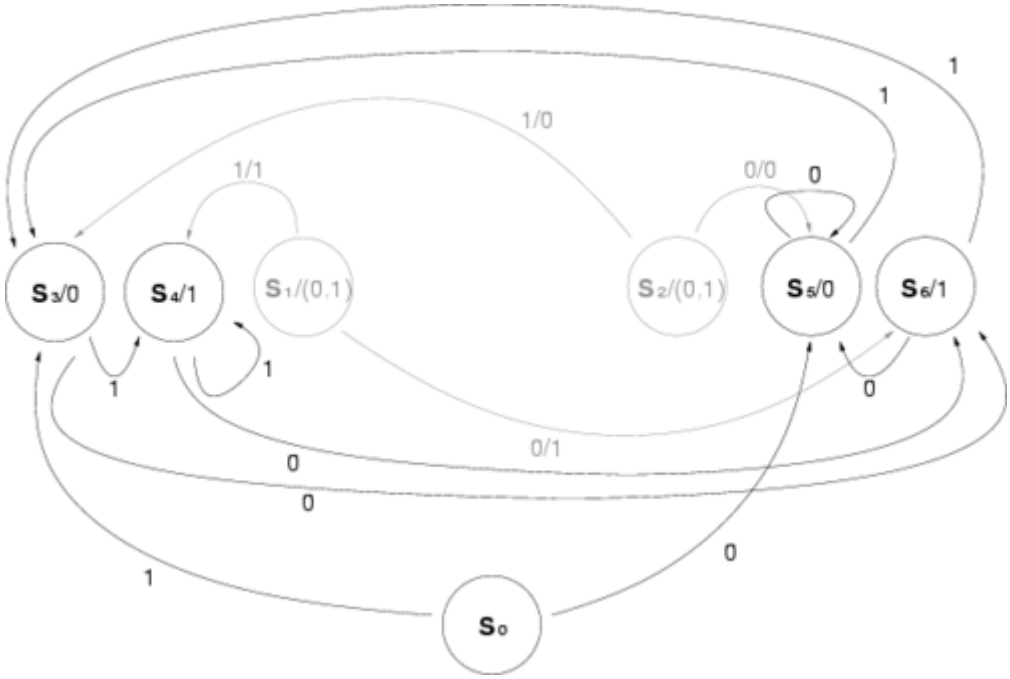
Schritt 2: Knoten aufspalten und eingehende Kanten umhängen

Die Zustände werden vervielfacht, so dass jedem Zustand nur noch höchstens ein Ausgabewert zugeordnet ist; anschließend hängt man eingehende Kanten entsprechend der Ausgabewerte auf die neuen Zustände um.

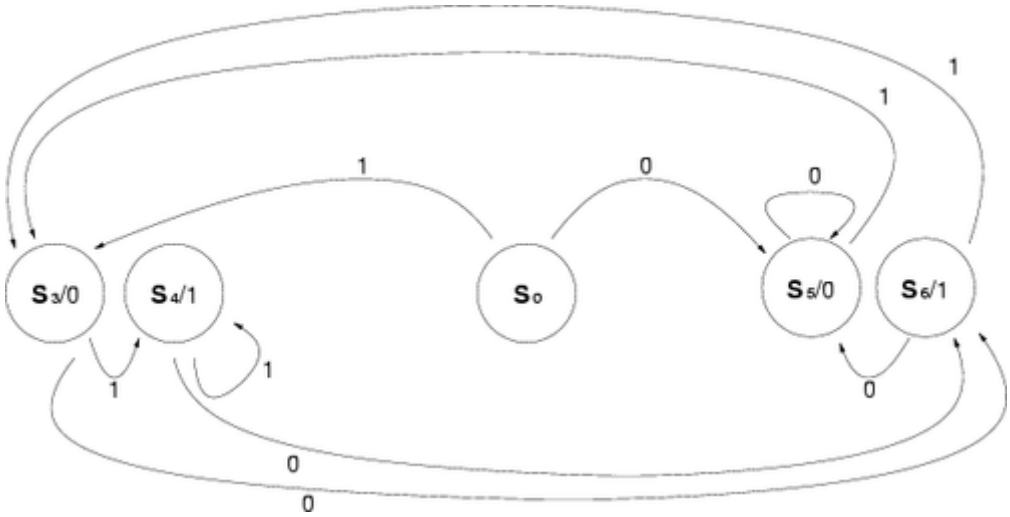


Schritt 3: Ausgehende Kanten vervielfachen

Zuletzt muss man alle ausgehenden Kanten der ursprünglichen Zustände kopieren und an die Zustände aus Schritt 2 anhängen.



Die Ausgabe des so konstruierten Moore-Automaten ist äquivalent zu der des ursprünglichen Mealy-Automaten.



Bemerkung: Oft besitzt dann auch der Startzustand des Moore-Automaten eine Ausgabe. Will man dies vermeiden, so könnte man einen zusätzlichen *Pre-Start-Zustand* einführen, welcher eine leere Ausgabe besitzt.

Aufgabe 1
Gegeben sei ein Mealy-Automat durch das Eingangsalphabet {0, 1}, das Ausgangsalphabet {a, b, c, d, e, f}, die Zustandsmenge $\{Z_1, Z_2, Z_3\}$ und folgende Übergangstabelle:

	0	1
Z_1	Z_2/a	Z_3/b
Z_2	Z_3/c	Z_3/d
Z_3	Z_3/a	Z_1/c

Führe diesen Mealy-Automaten in einen äquivalenten Moore-Automaten über und zeichne beide Zustandsgraphen!

Dein erhaltener Moore-Automat wird ein (bzgl. der Anzahl der Zustände) größerer Automat sein.
Andererseits kann man bekanntlich jeden Moore-Automaten in einen gleich großen Mealy-Automaten überführen.
Überführe also deinen Moore-Ergebnisautomaten, wie im letzten Kapitel beschrieben, wieder zurück in einen gleichgroßen Mealy-Automaten (d.h. genau so groß wie dein Ergebnis-Moore-Automat).

Ganz offensichtlich muss sich dieser nun entstandene Mealy-Automat bzgl. der Anzahl seiner Zustände optimieren lassen; denn Ausgangspunkt all deiner Umwandlungen war ja ein Mealy-Automat mit nur 3 Zuständen.
Woran kann man die Optimierungsmöglichkeit im Zustandsgraphen erkennen?

Aufgabe 2
Gegeben sei ein Mealy-Automat durch das Eingangsalphabet {1, 2, 3}, das Ausgangsalphabet {a, b, c}, die Zustandsmenge {Z_1, Z_2, Z_3} und folgende Übergangstabelle:

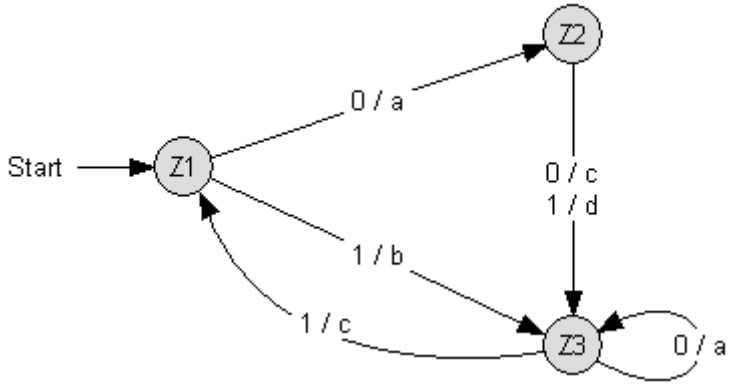
	1	2	3
Z_1	Z_1/a	Z_2/a	Z_3/c
Z_2	Z_1/a	Z_2/b	Z_3/c
Z_3	Z_1/c	Z_2/b	Z_3/c

Mache dasselbe wie in Aufgabe 1 !

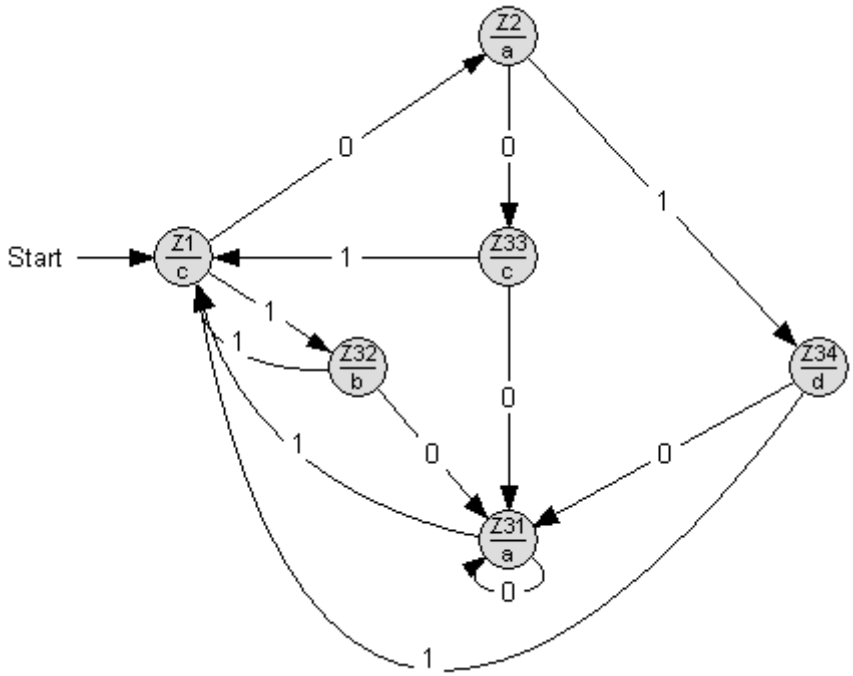
Lösungen

Aufgabe 1

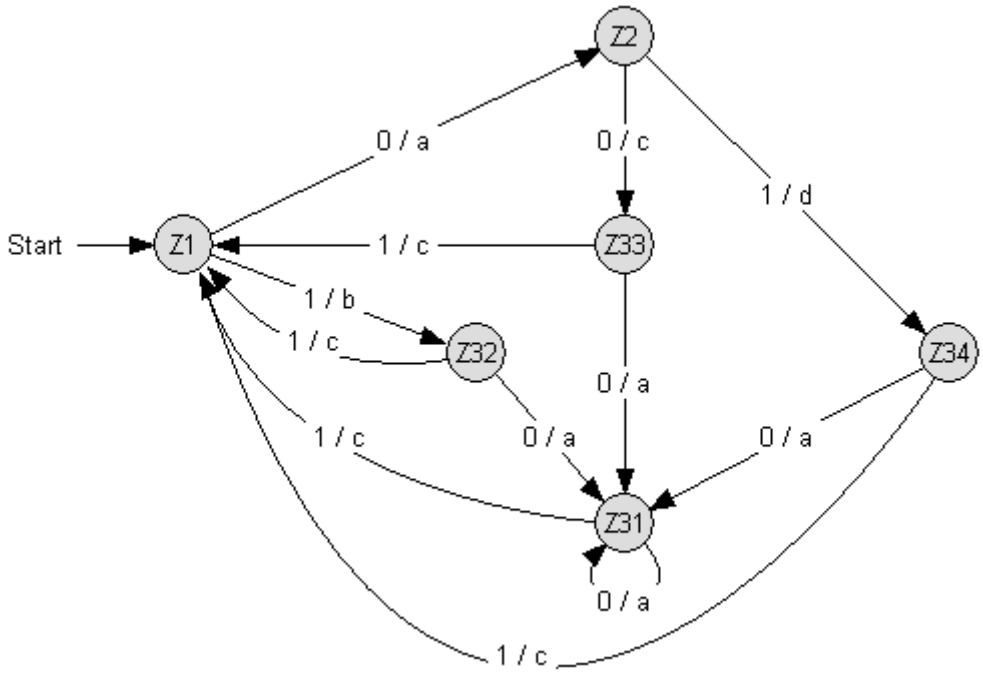
gegebener Mealy-Automat:



äquivalenter Moore-Automat:



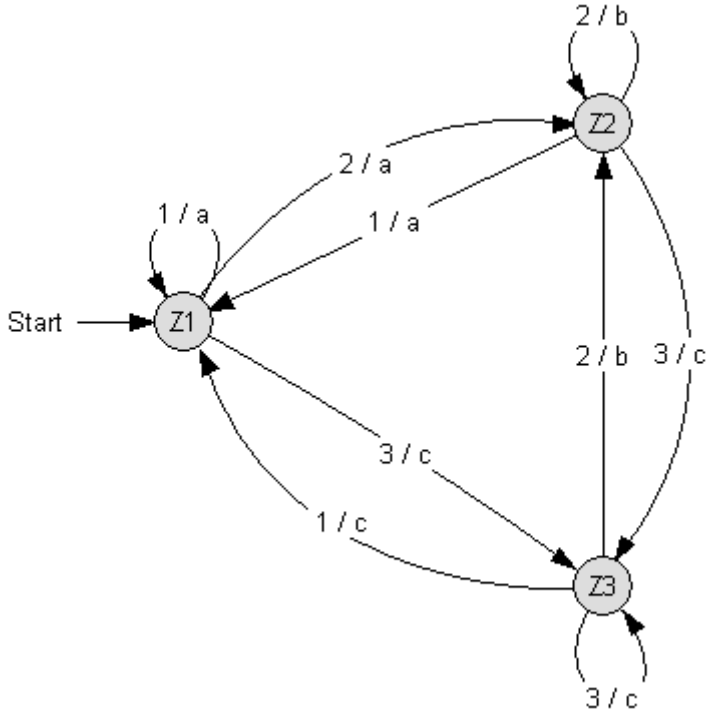
Äquivalenter Mealy-Automat:



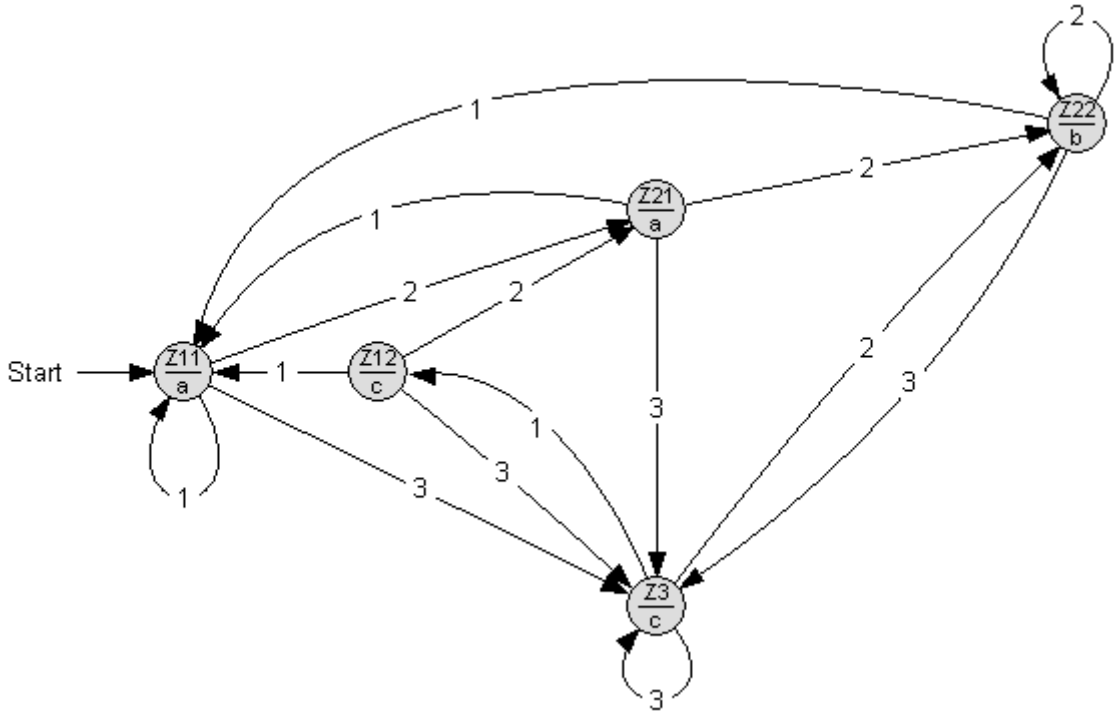
Man erkennt leicht, dass in diesem letzten Mealy-Automaten die vier Zustände Z_{31}, Z_{32}, Z_{33} und Z_{34} praktisch identisch sind und somit durch einen einzigen Zustand ersetzt werden können. Denn aus jedem dieser vier Zustände folgt eine Kante 1/c in den Zustand Z_1 und eine Kante 0/a in den Zustand Z_{31}.

Aufgabe 2

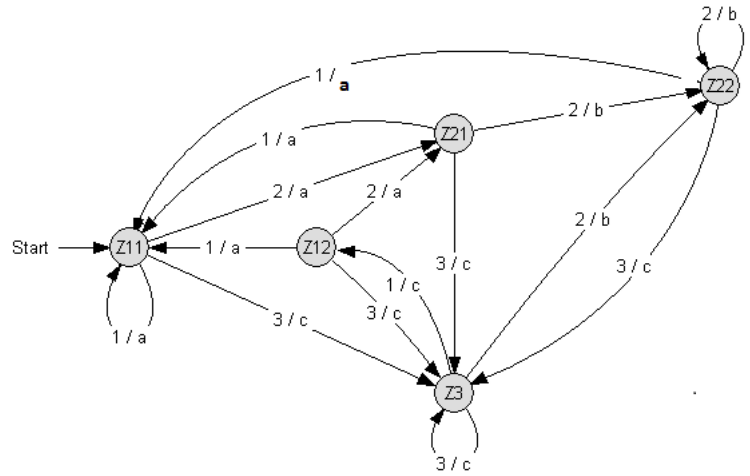
gegebener Mealy-Automat:



äquivalenter Moore-Automat:



Äquivalenter Mealy-Automat:



Man erkennt leicht, dass in diesem letzten Mealy-Automaten die Zustände Z_{11} und Z_{12} praktisch identisch sind und somit durch einen einzigen Zustand ersetzt werden können. Denn die Kanten, welche aus diesen beiden Zuständen herausführen, haben bei gleichem Eingangselement dasselbe Ausgangselement und denselben Folgezustand. Dasselbe gilt für die beiden Zustände Z_{21} und Z_{22} .

Erkennende Automaten

Erkennende Automaten sind Automaten, die keine Ausgabe haben. Ihre Aufgabe liegt darin, festzustellen, ob die gelesene Eingabefolge „korrekt“ ist. Man sagt dann, der Automat akzeptiert das Eingabewort. Erkennende Automaten nennt man deshalb auch ***Akzeptoren***. Die Anzeige, ob ein Wort akzeptiert wurde oder nicht, erfolgt über die Zustände des Akzeptors. Zum Beispiel könnte man die Endzustände dieses Automaten physikalisch mit einer entsprechenden Signallampe verbinden.

In diesem Sinne ist ein erkennender Automat nur ein 5-Tupel
$M = \{E, S, s_0, \sum, u\}$

Aufgaben

1. Entwickle einen Akzeptor, der nur Worte erkennt, welche das Teilwort „DELPHI“ enthalten. Benutze x für „alle anderen Zeichen“

2. Entwickle einen Akzeptor mit dem Eingabealphabet E = {0, 1}, der nur Worte akzeptiert, die mit 101 enden.

3. Entwickle einen Akzeptor mit dem Eingabealphabet E = {0, 1}, der nur Worte akzeptiert, die
a) irgendwo die Zeichenfolge 101 enthalten.
b) nirgendwo die Zeichenfolge 101 enthalten.

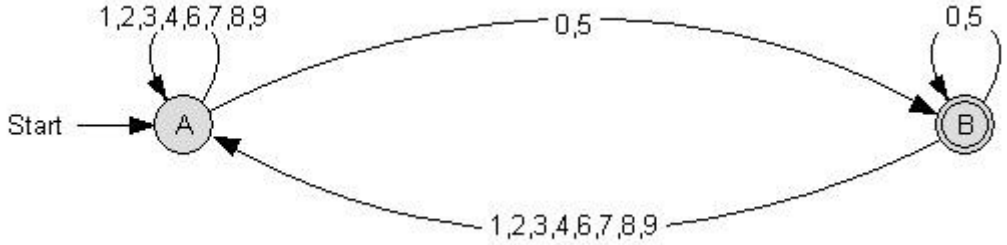
4. Entwickle einen Akzeptor mit zwei Zuständen und dem Eingabealphabet E = {0, 1}, der nur Worte akzeptiert, die eine gerade Anzahl von Einsen enthalten.

5. Entwickle einen Akzeptor (mit Fehlerzustand) mit dem Eingabealphabet E = {0, 1, 2, #}, der nur Zahlen akzeptiert, deren Ziffernsumme 5 beträgt. Eine Eingabe wird mit dem Doppelkreuz als Abschlusszeichen beendet. Beispiele: 01121#, 221#, 202010#

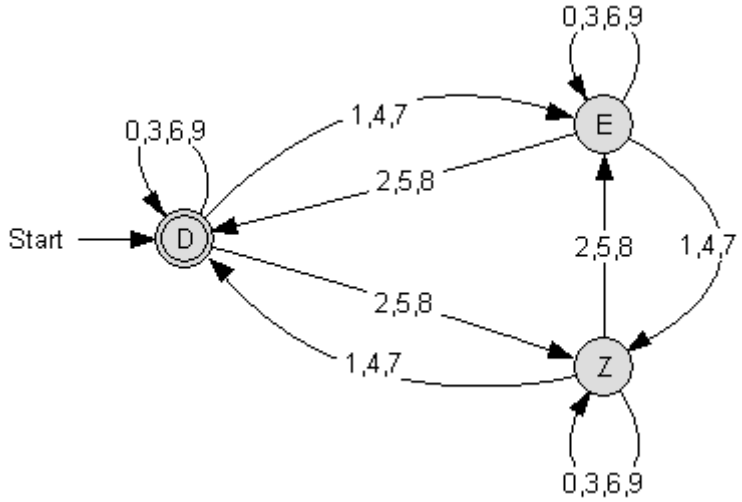
6. Entwickle einen Akzeptor mit dem Eingabealphabet E = {0, 1}, der nur Worte akzeptiert, die eine gerade Anzahl von Einsen und eine gerade Anzahl von Nullen haben.

7. Gesucht ist jeweils ein erkennender Automat, der nur Dualzahlen erkennt, die
a) eine ungerade Anzahl von Einsen enthalten.
b) mindestens entweder zwei aufeinander folgende Einsen oder zwei aufeinander folgende Nullen enthalten.
c) durch 2 (mindestens zweistellig und durch 4; mindestens dreistellig und durch 8) teilbar sind.
d) auf 0100 enden.

8. Was bewirkt ein Akzeptor mit folgendem Zustandsgraphen?



9. Was bewirkt ein Akzeptor mit folgendem Zustandsgraphen?



10. Erstelle einen Mealy-Automaten (für Teilaufgabe a) bzw. einen Akzeptor (für Teilaufgabe b), der nur BCD-Zahlen (**B**inary **C**oded **D**ecimal) erkennt. Jede einzelne Ziffer einer Dezimalzahl wird (wie üblich) 4-stellig binär codiert. Beispielsweise entspricht die Eingabe 1001 0100 0011 dem BCD-Code für die Dezimalzahl 943.
Es werden beliebig viele BCD-Zahlen eingegeben.
Nach jeder erkannten Dezimalziffer geht der Automat in den Anfangszustand zurück. Die Binärziffernfolge 1111 bedeutet das Ende der Eingabe.
Beim Übergang zum Endzustand wird das Zeichen ***e*** (Ende) ausgegeben.
Es existiert ein Fehlerzustand S_f, aus dem man nicht mehr herauskommt.
Beim Auftreten eines Fehlers wird (beim Mealy-Automaten) das Zeichen **a** (für Alarm) ausgegeben. Eine fehlerhafte Eingabe wäre z.B. 1010

Erstelle diesen Automaten in 2 Versionen:
a) Beim Rücksprung in den Startzustand (also nach dem Erkennen einer BCD-Zahl) wird die entsprechende Dezimalziffer ausgegeben.
b) Als Akzeptor, d.h. es gibt grundsätzlich keine Ausgaben.

11. Entwickle einen Automaten, der im Dezimalsystem nur natürliche Zahlen einschließlich Null akzeptiert, welche durch
a) 2
b) 100
c) 4 teilbar sind.

12. Schiffe in Seenot senden die Notruffolge SOS (**s**ave **o**ur **s**ouls) aus, die im Morsealphabet aus drei „Punkten“, drei „Strichen“ und wieder drei Punkten besteht. Damit ein Schiffskoch, der bei seiner Reederei Curry**sos**se bestellt, nicht versehentlich Alarm auslöst, wollen wir verlangen, daß SOS-Rufe von Worttrennzeichen **W**(Leertaste, Satzzeichen usw.) eingerahmt werden.
Notruffolge ist also W…---…W
Entwickle einen erkennenden Automaten, der nur durch eine Notruffolge in den Endzustand gesetzt wird. Eingangsalphabet = {W, . , - }

13. Entwickle einen erkennenden Automaten für Zahlen in mehreren Schritten! Die Problematik von führenden Nullen wird in dieser Aufgabe ignoriert.
a) natürliche Zahlen, etwa 17583
b) ganze Zahlen mit und ohne Vorzeichen, etwa -5289, 22891, +17
c) Dezimalzahlen mit Dezimalpunkt, etwa 0.52, -17.4, +66.341, 233
d) Dezimalzahlen in Exponentendarstellung, etwa -2.5E17, 4.8E-6, +24358

14. Entwickle einen Akzeptor für Variablennamen. Diese beginnen mit einem Buchstaben und können außer weiteren Buchstaben noch Ziffern und den Unterstrich enthalten. Wähle als Eingabealphabet E = {B, Z, _}

15. Entwickle einen Automaten, der nur Wörter akzeptiert, die mit „man“ enden. Akzeptiert werden also Herman, Müsliman und Mangaman, nicht aber command oder manamana. Benutze das Zeichen x, welches für alle anderen Zeichen außer a, m, n stehen soll!

16. Entwickle einen Akzeptor für korrekte mathematische Terme, die nur aus vorzeichenlosen Zahlen und Operatoren bestehen (keine Klammern und keine Variablen)! Beispiel: 582+33/11-5*2 wird akzeptiert, 23+*4 wird nicht akzeptiert.

17. Entwickle einen Akzeptor mit dem Eingabealphabet E = {0, 1}, der nur Worte akzeptiert, die an drittletzter Stelle eine 0 haben!

18. Entwickle einen Akzeptor mit dem Eingabealphabet E = {a}, der nur Worte akzeptiert, deren Länge durch 3 oder durch 4 teilbar ist!

19. Gib das Zustandsdiagramm zu folgendem Akzeptor-Automaten an:
$M = (\{0, 1\}, \{Q_1, Q_2, Q_3, Q_4, Q_5\}, Q_1, \{Q_3\}, u)$, wobei die Übergangsfunktion u durch folgende Tabelle gegeben ist:

	0	**1**
Q_1	**Q_1**	**Q_2**
Q_2	**Q_1**	**Q_3**
Q_3	**Q_2**	**Q_4**
Q_4	**Q_3**	**Q_5**
Q_5	**Q_4**	**Q_5**

20. Entwickle einen endlichen Automaten, der eine gültige Uhrzeit im Format hh:mm akzeptiert.

Beispiele: 00:23 wird akzeptiert,
45:12 wird nicht akzeptiert,
:1234 wird nicht akzeptiert.

Hinweis: der Doppelpunkt ist auch ein Zeichen des Eingabealphabets.

21. Ein Logistikunternehmen verwendet intern für Pakete die Angaben für Länge, Breite und Höhe in der Form 195x100x80 (ohne Maßeinheit). Ist ein Paket auf mindestens einer Seite sehr lang, muss es mit einem Extra-Fahrzeug transportiert werden. Ein Aufkleber auf jedem Paket wird von einem Automaten eingescannt. Der Automat bekommt eine Eingabe in der oben angegebenen Form. Entwickle einen endlichen Automaten so, dass nur die Pakete akzeptiert werden, bei denen alle Kanten **echt kleiner** als 200 sind.

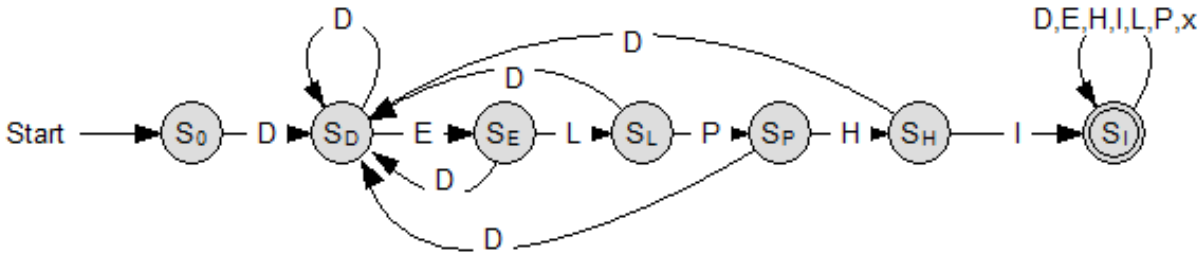
Beispiele: 50x120x80 wird akzeptiert,
200x90x90 wird nicht akzeptiert,
10x190x3 wird akzeptiert.

Hinweise: Beachte, dass auch das „x“ ein Zeichen des Eingabealphabets ist. Es wird vorausgesetzt, dass es keine führenden Nullen gibt.

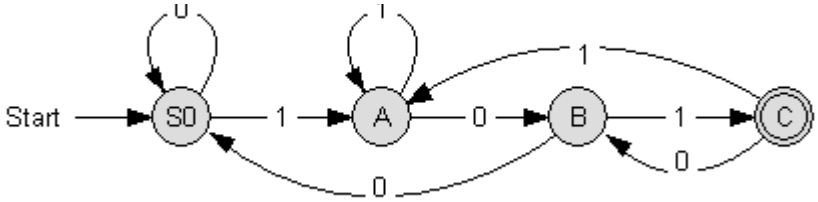
Lösungen

Bemerkung: Falls nicht anders erwähnt, führen bei allen Lösungen alle nicht gezeichneten Übergänge in einen Fehlerzustand.

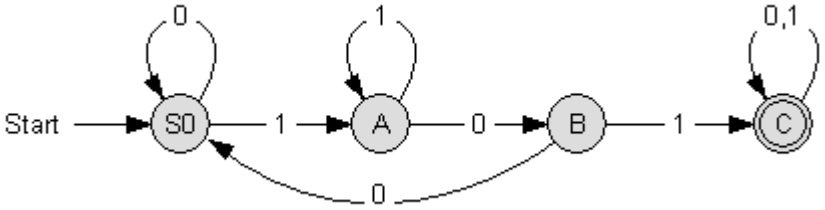
Aufgabe 1:
Die nicht eingezeichneten Übergänge führen alle in den Anfangszustand.



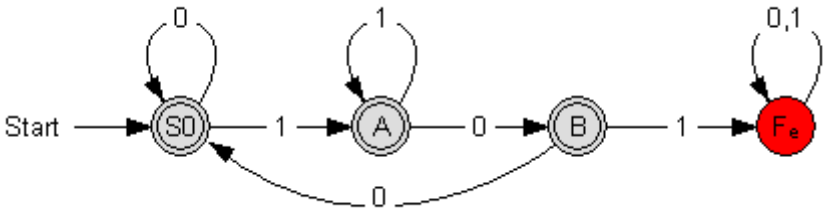
Aufgabe 2:



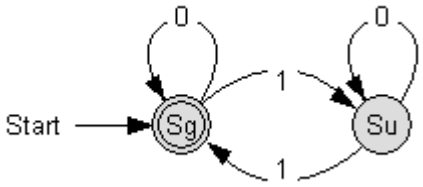
Aufgabe 3a):



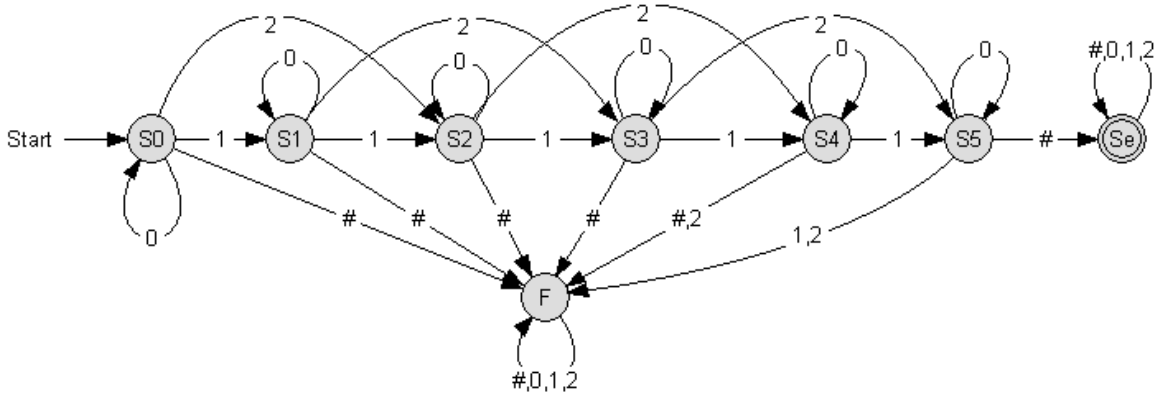
Aufgabe 3b):



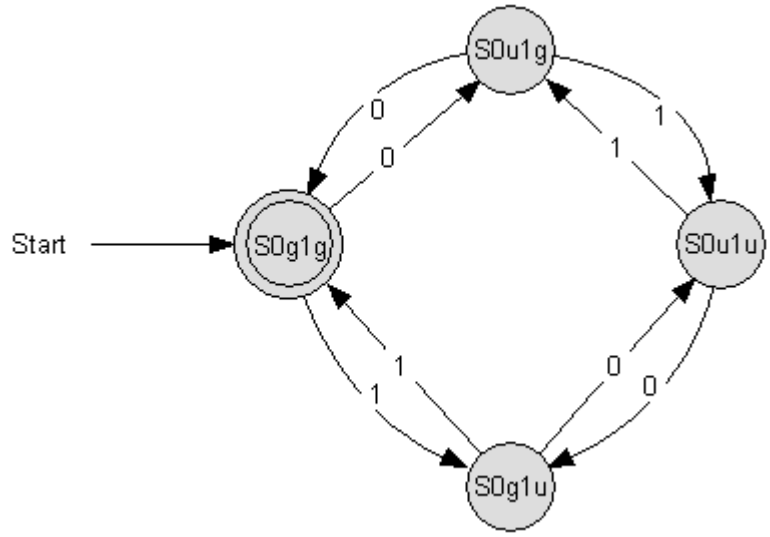
Aufgabe 4:



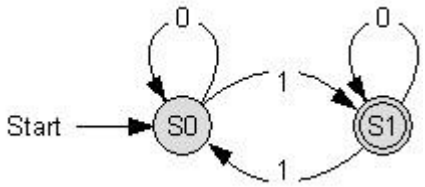
Aufgabe 5:



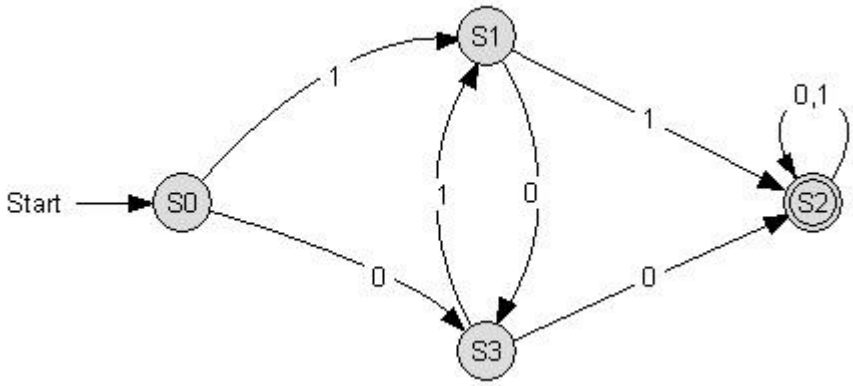
Aufgabe 6:



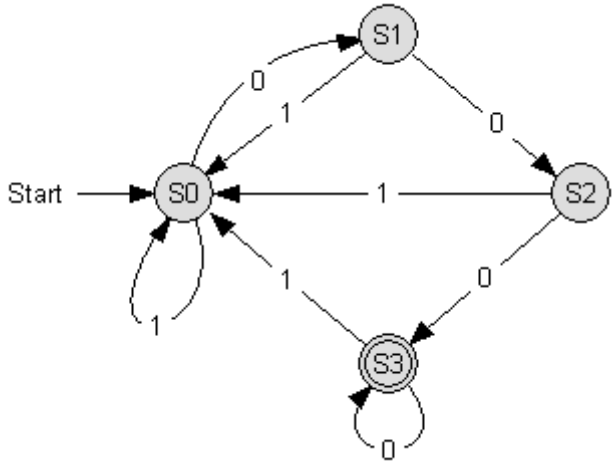
Aufgabe 7. a)



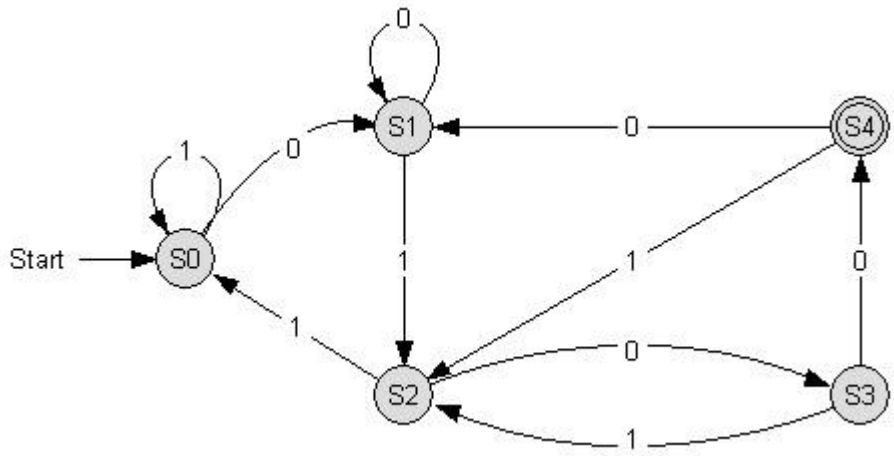
b)



c) Der folgende Automat akzeptiert mindestens dreistellige Dualzahlen, die durch 8 teilbar sind.



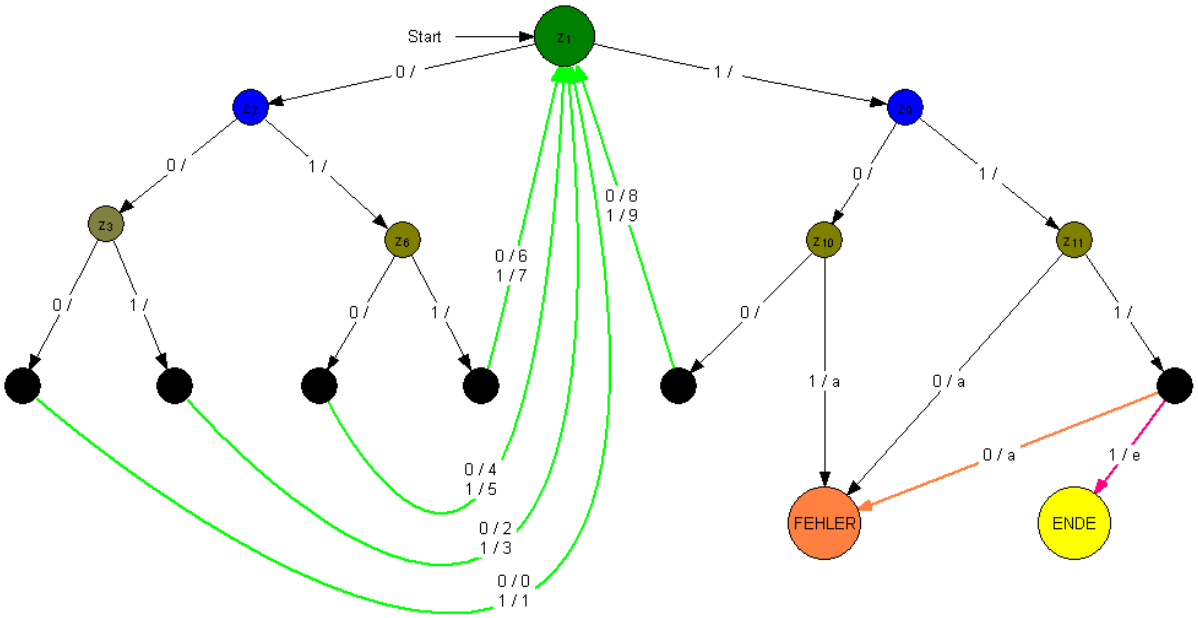
d)



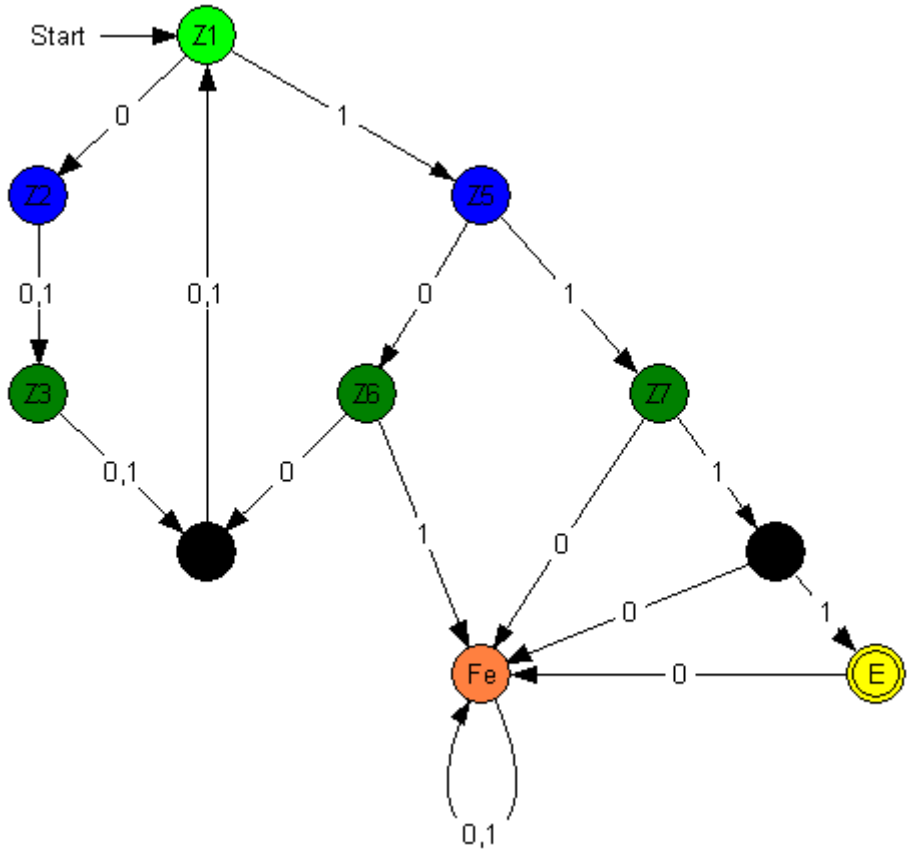
Aufgabe 8: Der Akzeptor erkennt Dezimalzahlen, die durch 5 teilbar sind.

Aufgabe 9: Der Akzeptor erkennt Dezimalzahlen, die durch 3 teilbar sind.

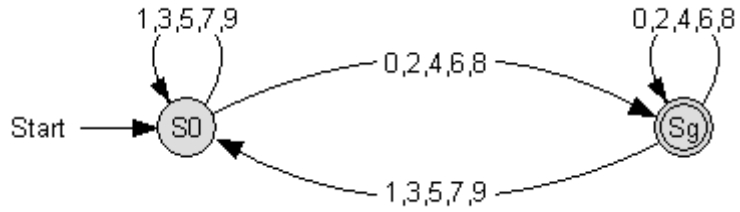
Aufgabe 10a:



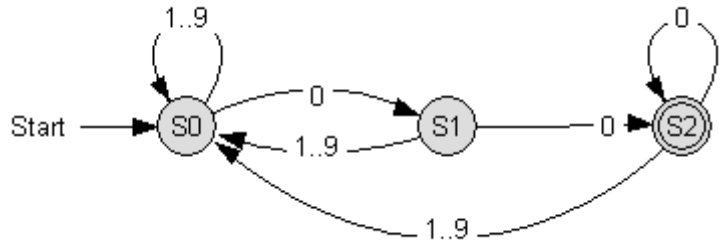
Aufgabe 10b:



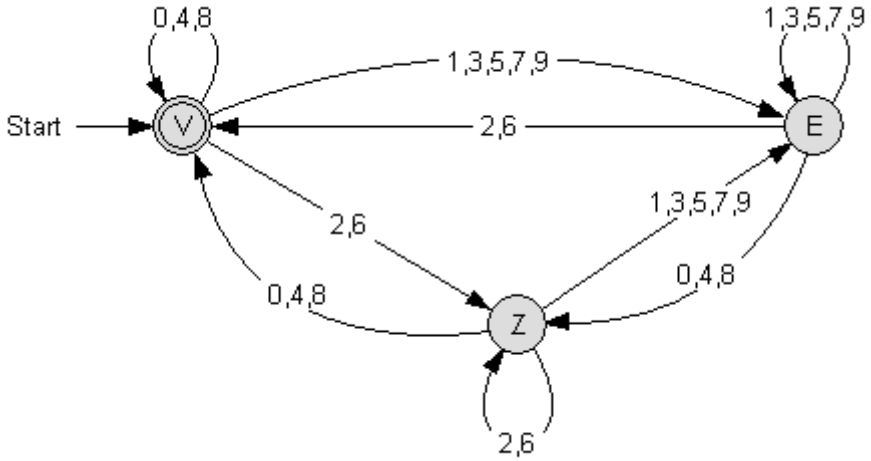
Aufgabe 11 a)



b)

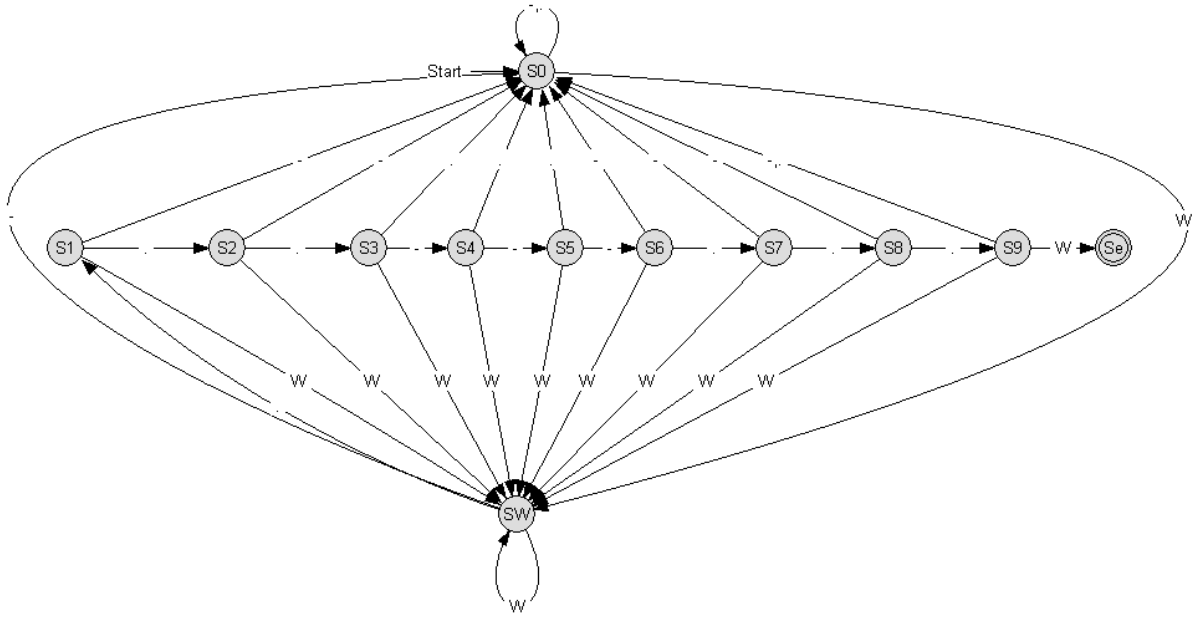


c)



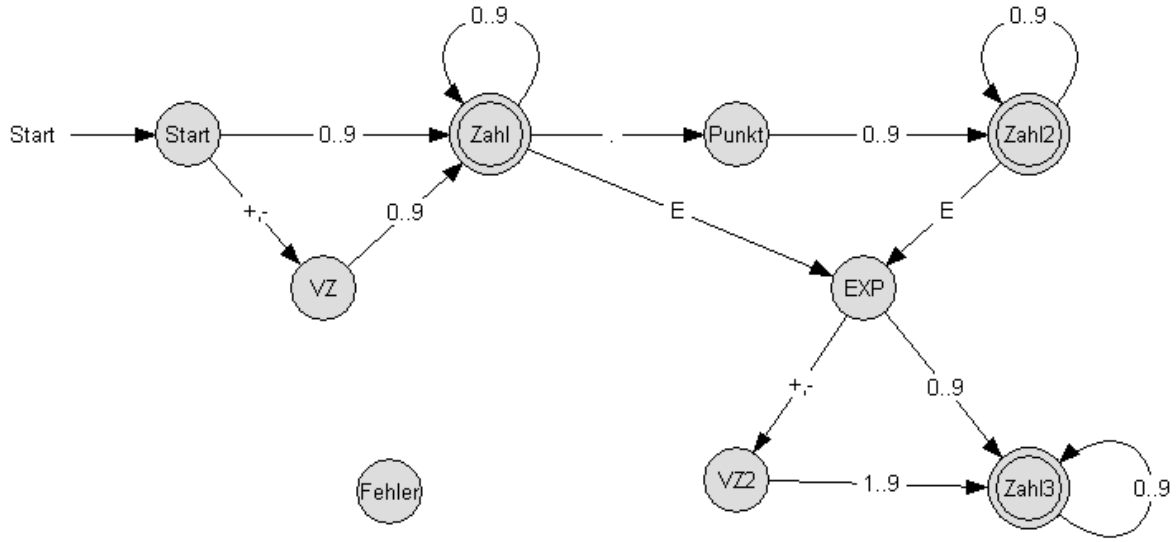
Bemerkung: die obigen Zustände V und Z sind nicht identisch, obwohl alle ihre entsprechenden Ausgänge zu denselben Folgezuständen führen. Vergleiche dazu auch die Bemerkung auf Seite 14.

Aufgabe 12:

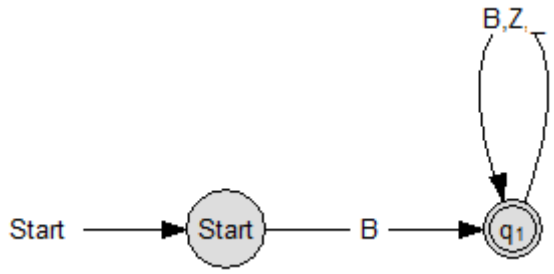


Aufgabe 13:

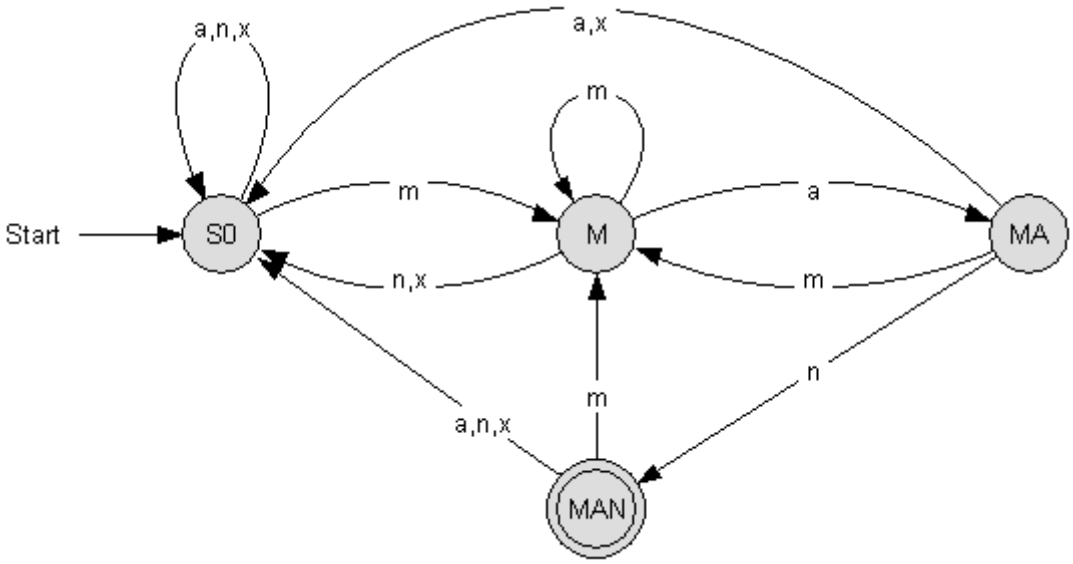
Alle nicht eingezeichneten Übergänge führen in den Fehlerzustand.



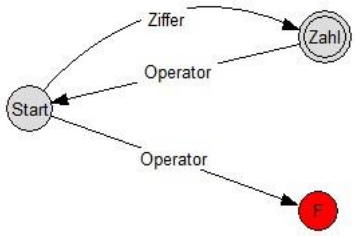
Aufgabe 14:



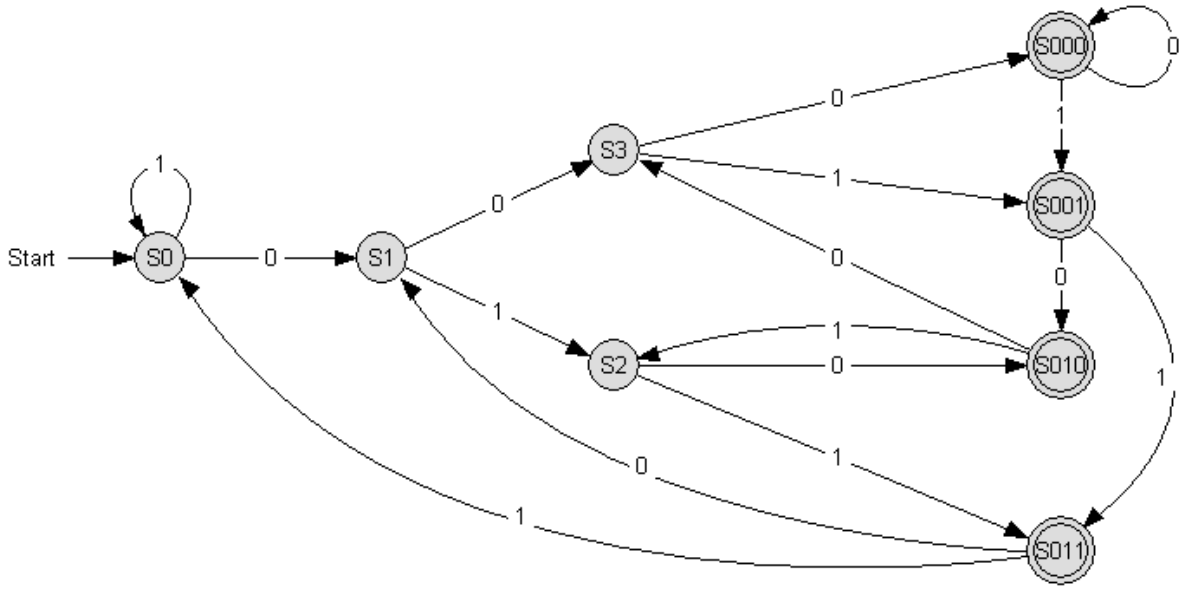
Aufgabe 15:
Das Zeichen x steht für alle anderen Zeichen außer a, m, n



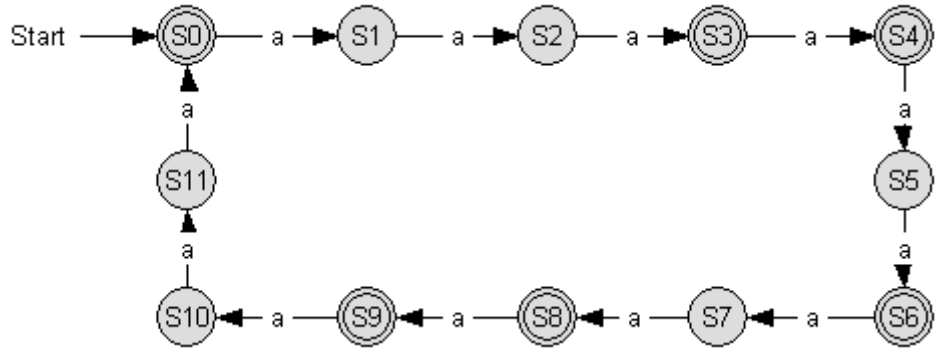
Aufgabe 16:



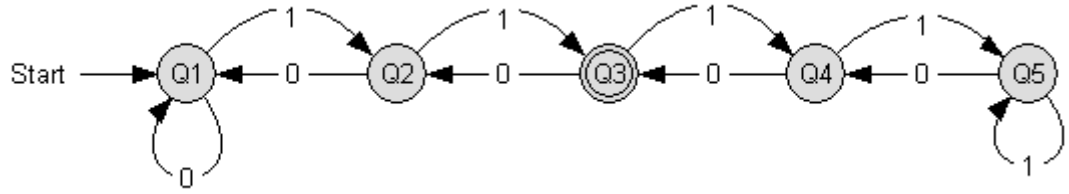
Aufgabe 17:



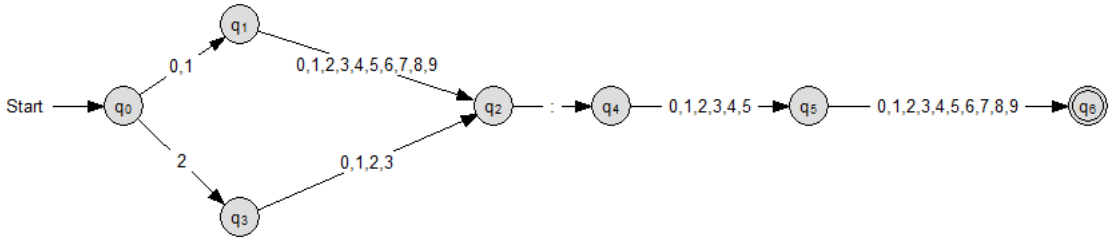
Aufgabe 18:



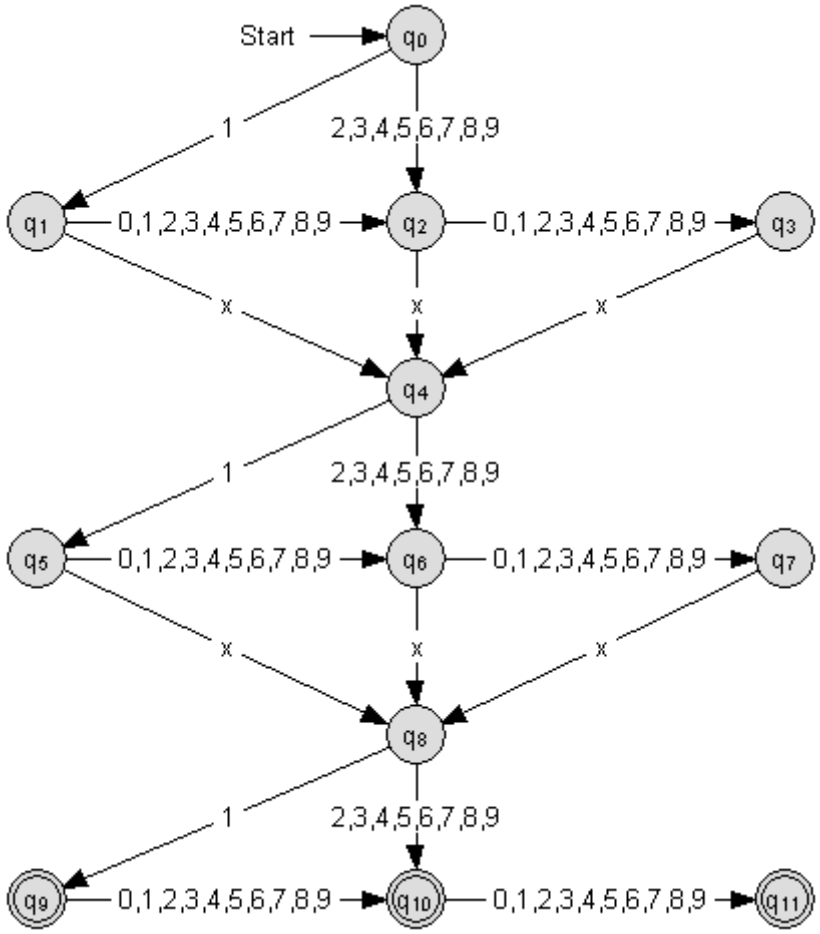
Aufgabe 19:



Aufgabe 20:



Aufgabe 21:


Start
q0
1
2,3,4,5,6,7,8,9
q1
0,1,2,3,4,5,6,7,8,9
q2
0,1,2,3,4,5,6,7,8,9
q3
x
x
x
q4
1
2,3,4,5,6,7,8,9
q5
0,1,2,3,4,5,6,7,8,9
q6
0,1,2,3,4,5,6,7,8,9
q7
x
x
x
q8
1
2,3,4,5,6,7,8,9
q9
0,1,2,3,4,5,6,7,8,9
q10
0,1,2,3,4,5,6,7,8,9
q11

Suche nach mehreren Begriffen gleichzeitig

Oft benötigt man eine Suchmaschine, welche nach mehreren Begriffen gleichzeitig sucht. Generell gibt es zwei mögliche Fälle:

Fall 1: Zwei Suchworte können sich **teilweise** überlappen, wie z.B. ER, SIE, ES. Zum Beispiel würde man in dem Eingabewort „RASIERMESSER“ zweimal ER, einmal SIE und einmal ES finden.

Fall 2: Zwei Suchworte können sich **vollständig** überlappen, wie z.B. „Delphi“ und „Delphin“ oder etwa „AFFE“ und „GIRAFFE“.

Die Wahl des Automatentyps (Moore-, Mealy-Automat oder Akzeptor) hängt von der konkreten Aufgabenstellung ab:

- Soll der Automat in einem Endzustand bleiben, sobald er einen von mehreren Begriffen gefunden hat? Allerdings würde dieser dann von den beiden Wörtern DELPHI und DELPHIN niemals das zweite Wort finden.
- Soll der Automat, nachdem er einen Begriff gefunden hat, diesen durch ein entsprechendes Signal bekannt geben und sofort weiter suchen? In diesem Fall bietet sich ein Moore-Automat an.

Aufgaben

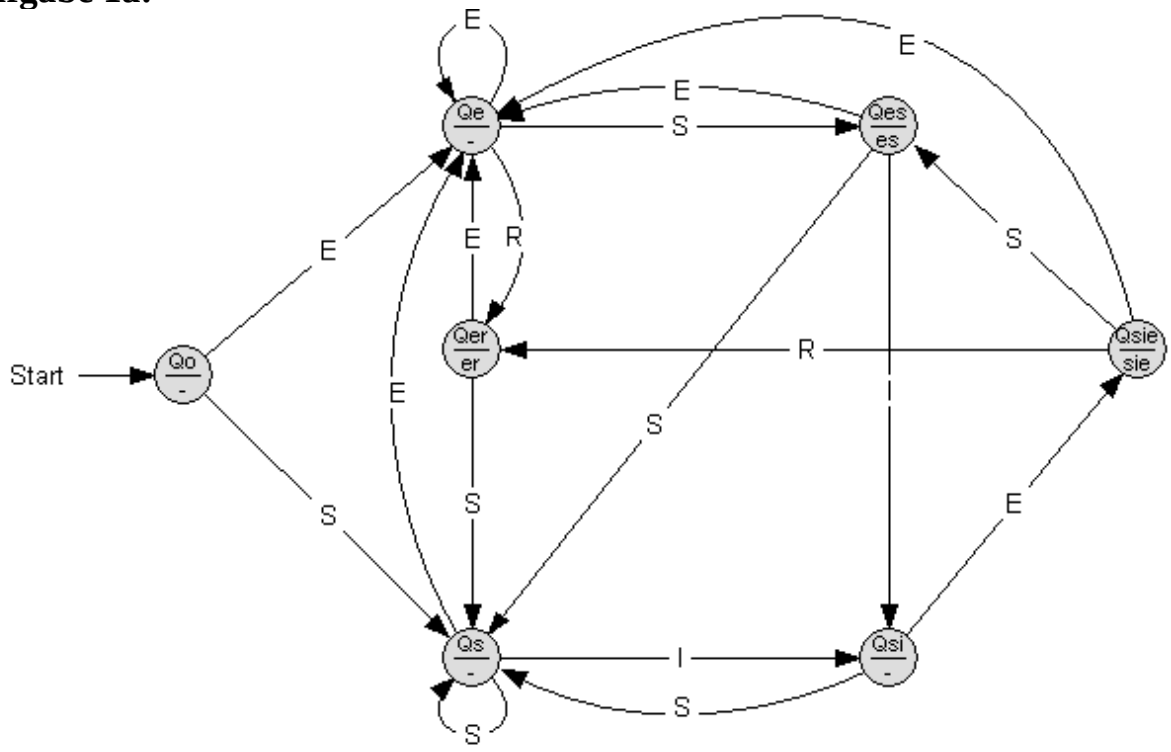
Entwickle erkennende Automaten, die (egal, in welchem Kontext) gleichzeitig nach verschiedenen Begriffen suchen! Beschreibe auch, was deine Automaten konkret machen!
Zeichne den Zustandsgraphen der jeweiligen Automaten nur mit den für die Übergänge wichtigen Eingabezeichen. Dies hat den Vorteil der Übersichtlichkeit, allerdings den Nachteil, dass man diese Automaten nicht mehr mit dem Hilfsprogramm AtoCC simulieren kann.

1. a) ER, SIE, ES
b) ELIA, ELISA, JESAJA.

2. a) AFFE, GIRAFFE
b) HE, SHE, HERS

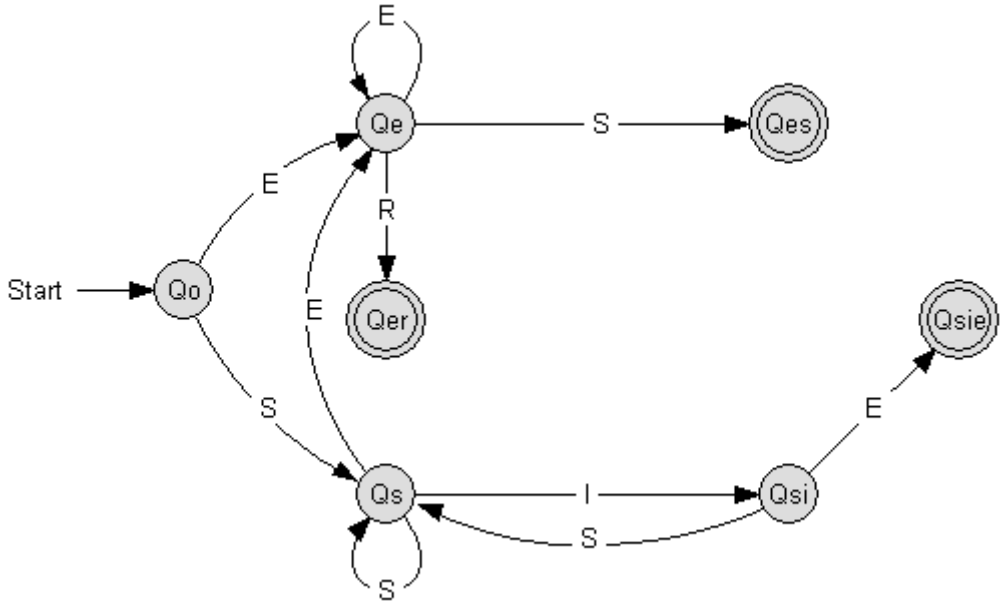
Lösungen

Aufgabe 1a:



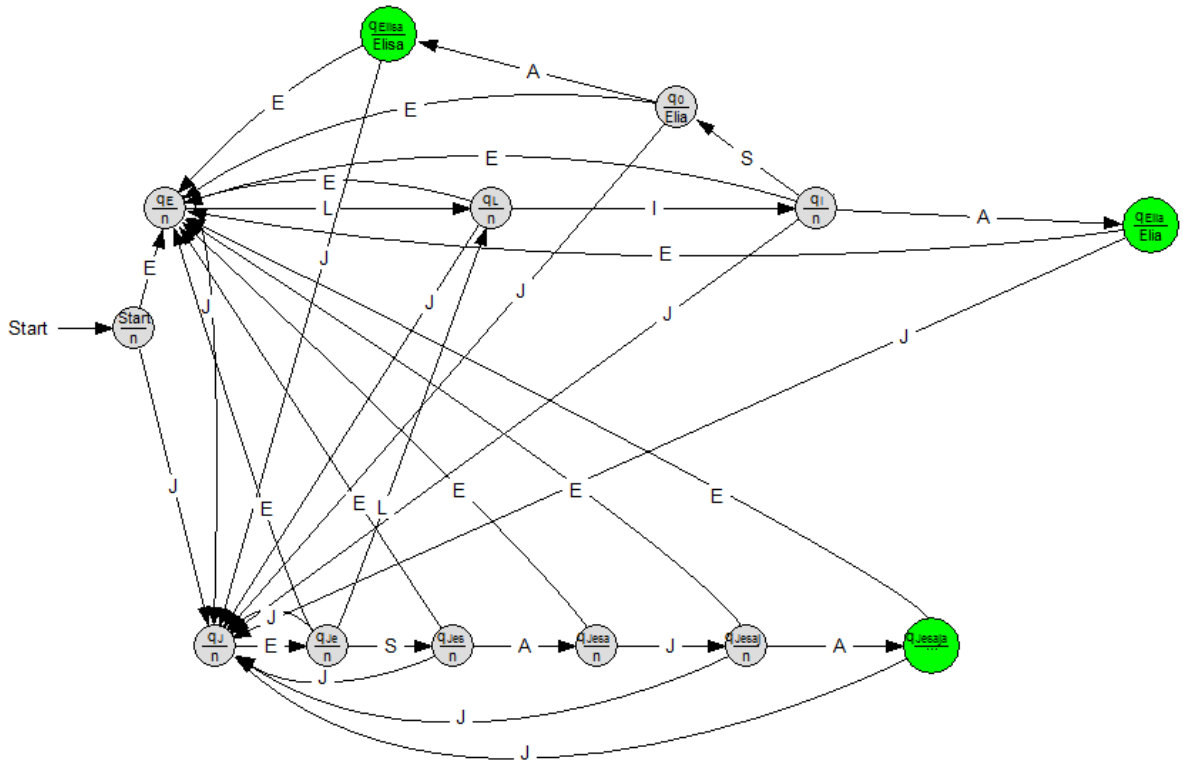
Alle nicht eingezeichneten Übergänge führen in den Anfangszustand.

Nun eine Lösung als Akzeptor:



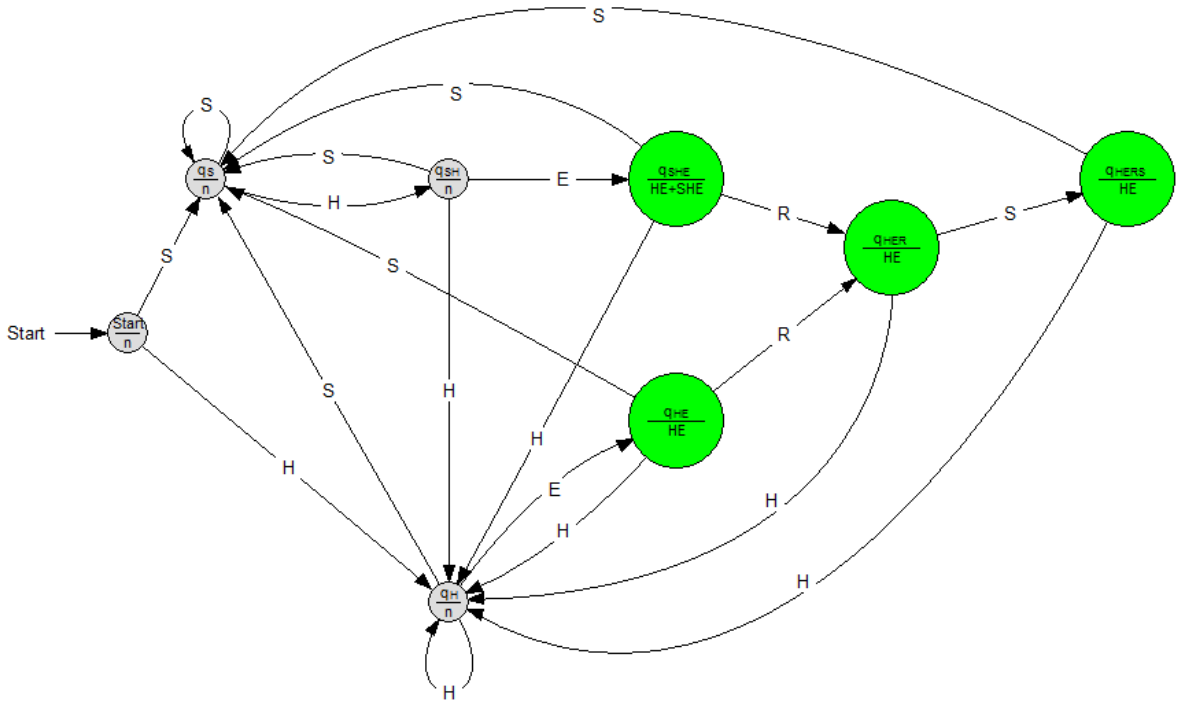
Aus dem Endzustand kommt man nicht mehr heraus. Alle anderen nicht eingezeichneten Übergänge führen in den Anfangszustand.

Aufgabe 1b:



Alle nicht eingezeichneten Übergänge führen in den Startzustand.

Aufgabe 2b:



Alle nicht eingezeichneten Übergänge führen in den Startzustand.

Formale Sprachen

Um Informationen austauschen zu können, bedient man sich der Sprache. Dabei müssen Sätze nach bestimmten Regeln aufgestellt werden, damit ihre Bedeutung verstanden wird. Diese Regeln werden Grammatik genannt. Natürliche Sprachen haben einen hohen Wortschatzumfang und komplexe Grammatiken. Somit kann man komplizierte Sachverhalte sprachlich ausdrücken.

In technischen oder theoretischen Systemen reicht jedoch meistens eine geringe Wortanzahl, um Informationen auszutauschen. Daher ist es nicht sinnvoll, sich an natürlichen Sprachen zu orientieren. Vielmehr versucht man, die Sprachen den Bedürfnissen der Systeme anzupassen. Dadurch entstehen formale Sprachen wie die mathematische Formelsprache und die Computersprachen.

Formale Sprachen sind nicht nur für den Informationsaustausch wichtig, sondern auch für die Grundlagenforschung. So werden etwa Methoden gewonnen zur Sprach- und Zeichenerkennung bei natürlichen Sprachen.

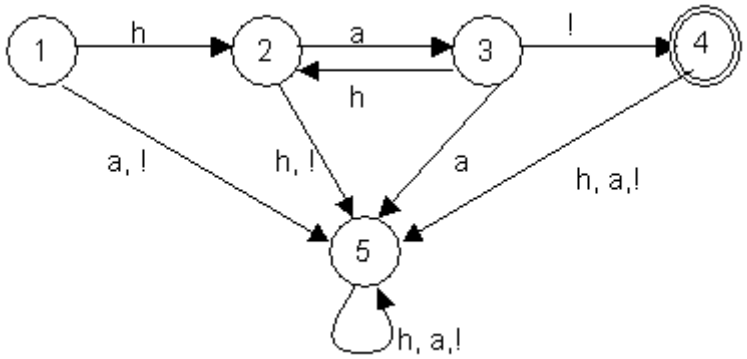
Die grammatikalischen Regeln einer Sprache bestimmen ihre Syntax. Bei Formalen Sprachen reichen häufig wenige Regeln und ein stark eingeschränkter Wortschatz aus. Wichtig ist, dass die grammatischen Regeln eindeutig sind.

Der Begriff der *Formalen Sprache* soll zeigen, dass es sich bei den betrachteten Sprachen nicht um natürliche oder Programmier-Sprachen handelt. In Formalen Sprachen spielt die Semantik (= Bedeutung der Sätze) keine, die Syntax (= Lehre vom Satzbau; Grammatik) hingegen die zentrale Rolle.

Wir richten unser Augenmerk nun nur auf die Syntax, also auf die formale Richtigkeit der Satzbildung. Deshalb spricht man von *formalen Sprachen*. Anzumerken ist, dass formal richtig gebildete Sätze immer noch völlig unsinnig sein können: „Der Hund fliegt kluges Wasser“. Die Bedeutung der Sätze, die sog. *Semantik*, wird von uns nicht untersucht.

Einführendes Beispiel: Lachautomaten-Akzeptor

Aufgabe: Es ist ein Akzeptor zu konstruieren, der folgendes erkennt: ha!, haha!, hahaha! usw.
Lösung als Zustandsgraph:



Dieser Automat akzeptiert (nur) alle Wörter der geforderten Gestalt. Er ist also in der Lage, eine – wenn auch beschränkte – Sprache zu erkennen.

Ein ***Wort w*** ist die Hintereinanderreihung endlich vieler Zeichen aus einem vorgegebenen Alphabet ***T***. Dieses Alphabet wird auch als die Menge ***T*** der ***Terminalzeichen*** bezeichnet. Die Menge aller Wörter wird mit T^* bezeichnet. Das leere Wort gehört zu T^*, besteht aus keinem Alphabetzeichen und wird mit **ε** bezeichnet.

Beispiel für den Lachautomaten:
$T = \{a, h, !\}$
$T^* = \{\varepsilon, a, h, !, aa, ah, a!, ha, hh, h!, !a, !h, !!, aaa, aah, \ldots\}$

Eine ***formale Sprache L*** über einem Alphabet ***T*** ist eine Teilmenge aus T^*, also $L \subseteq T^*$. Die Elemente von ***L*** sind auch Wörter. Normalerweise gehört aber nicht jedes mögliche Wort aus T^* auch zur Sprache ***L*** . Wir werden in Kürze ein Beispiel kennenlernen, in dem sogar ein einzelnes, bestimmtes Terminalzeichen nicht zur Sprache ***L*** gehört. Das hängt natürlich von den Regeln ab, wie eine Sprache gebildet werden soll.

Beispiel für den Lachautomaten: ***L*** soll die Lachsprache sein:
$L = \{ha!, haha!, hahaha!, \ldots\} = \{(ha)^n! \mid n > 0\}$

Oft ist es Auffassungssache, welches die Terminalzeichen und welches die aus diesen Terminalzeichen gebildeten ***Worte*** oder ***Sätze*** sind.
Beispiel: Bilden in der Morsesprache Punkt, Strich und Pause die Menge der Terminalzeichen oder besteht die Menge der Terminalzeichen aus den codierten Buchstaben?

Unter einem ***Wort*** versteht man eine (eventuell auch leere) Folge von Terminalzeichen.
Es ist möglich, aus den Terminalzeichen der deutschen Sprache (dazu gehören das Alphabet und die Satzzeichen) Worte zu bilden, die nicht zur deutschen Sprache gehören, z.B. „Xryzs". Ebenso kann man aus richtigen deutschen Worten völlig falsche Sätze bilden, z.B. „Hund und baut alle sind".

Wir werden deshalb aus allen möglichen Terminalzeichenfolgen diejenigen herausgreifen, die nach „gewissen Regeln", den sog. ***Produktionsregeln,*** aufgebaut sind und damit die Sprache ***L*** bilden.

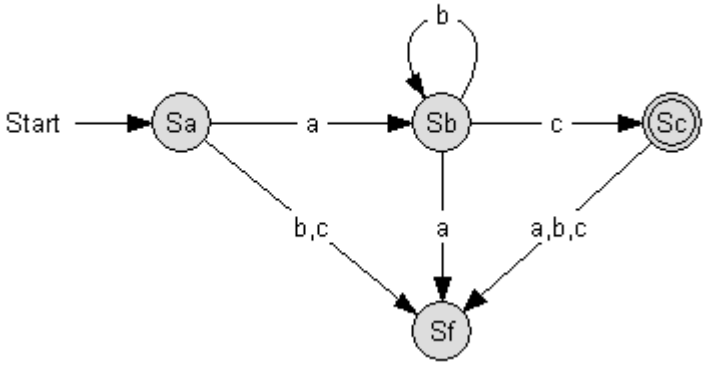
Aufgaben

1. Man gebe einen Automaten an, der genau die Sprache $L = \{a\, b^n\, c \mid n \geq 0\}$ akzeptiert. Dabei versteht man unter b^0 das leere Wort ε.

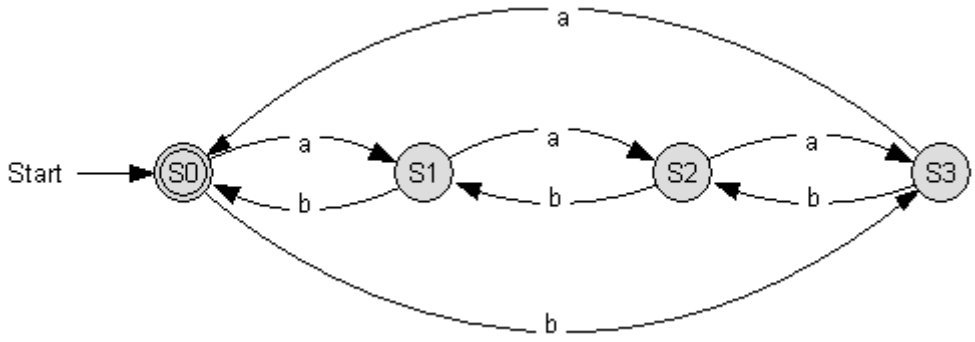
2. Man gebe einen Automaten an, der ein Wort einer Sprache über dem Alphabet {a, b} genau dann akzeptiert, wenn die Differenz der Anzahl der Elemente a und der Anzahl der Elemente b in diesem Wort durch vier teilbar ist.

Lösungen

1.



2. Die Zustände entsprechen den Werten (Anzahl a - Anzahl b) MOD 4



Im Folgenden wird nun versucht, formale Sprachen ohne Zuhilfenahme von Automaten zu beschreiben.

Beispiel 1: Umgangssprachliche Sätze

Es werden folgende Sätze als kleiner Ausschnitt der deutschen Sprache untersucht:

"Die Katze jagt",
"Das Buch frisst das Heu",
"Susanne liest das Buch".

Grammatikalisch gesehen haben diese Sätze die Nominalphrase und die Verbalphrase gemeinsam. Die Nominalphrase ist daran erkennbar, dass sie ein Nomen oder ein Pronomen enthält. Die Verbalphrase ergänzt die Nominalphrase zu einem vollständigen Satz. In Form einer Grammatikregel wird der Sachverhalt folgendermaßen ausgedrückt:

(1) <Satz> → <Nominalphrase> <Verbalphrase>

Der Pfeil wird als *besteht aus* gelesen. Die Nominalphrasen dieser Sätze bestehen aus Eigennamen oder einem Substantiv mit zugehörigem Artikel; dies formalisiert folgende Grammatikregel:

(2.1) <Nominalphrase> → <Eigenname>

(2.2) <Nominalphrase> → <Artikel> <Substantiv>

Die Verbalphrase besteht entweder aus einem Verb oder aus einem Verb mit Akkusativobjekt; letzteres ist wieder eine Nominalphrase:

(3.1) <Verbalphrase> → <Verb>

(3.2) <Verbalphrase> → <Verb><Nominalphrase>

Außerdem ergibt sich ein kleines *Lexikon*:

<Eigenname> → *Susanne*

<Substantiv> → *Katze* | *Pferd* | *Heu* | *Buch*

<Artikel> → *der* | *die* | *das*

<Verb> → *jagt* | *frißt* | *liest*

Die Gesamtheit der Ersetzungsregeln (1) – (3) heißt ***Grammatik.***

Anwendung: Es soll der Satz *„Susanne liest“* hergeleitet werden!

Die Herleitung eines Satzes geschieht dadurch, dass jeweils der linke Teil einer Regel durch deren rechten Teil ersetzt wird, wobei stets mit <Satz> zu beginnen ist:

<Satz> → <Nominalphrase> <Verbalphrase> nach Regel (1)

→ <Eigenname> <Verbalphrase> nach Regel (2.1)

→ *Susanne* <Verbalphrase> Lexikon

→ *Susanne liest* Lexikon

Den Pfeil liest man als *„wird ersetzt durch“*. Der zweite Beispielsatz oben macht deutlich, dass sich auch semantisch sinnlose Sätze herleiten lassen. Möchte man sicherstellen, dass eine Grammatik nur sinnvolle Sätze erzeugt, muss man zusätzlich semantische Regeln festlegen.

Beispiel 2: Wohlgeformte Klammerterme

Wenn man in einem Rechenausdruck sämtliche Buchstaben und Zahlen weglässt, so erhält man Klammerterme der Form (); (()()) oder ((((())()))). Ein solcher Term muss gleich viele öffnende wie schließende Klammern besitzen. Die Grammatikregeln zur Erzeugung wohlgeformter Klammerterme lauten:

(1) S → ( ) (2) S → (S) (3) S → SS

Mit (3) kann man Klammerterme aneinanderreihen, mit (2) eine rekursive Verschachtelung vornehmen und mit (1) schließlich den Erzeugungsvorgang abbrechen.
Vereinfacht lassen sich die drei Regeln auch so darstellen
S → ( ) | (S) | SS

Die Klammerfolge (( )) (( ) ( )) kann so abgeleitet werden:

S → SS → (S) S → (( )) (S) → (( )) (S S) → (( )) (( ) S) → (( )) (( ) ( ))

In den Grammatikregeln der beiden Beispiele kommen zwei Arten von Zeichen vor:

Terminalzeichen : endgültige Zeichen; das sind Zeichen, die zum Alphabet gehören; in Beispiel 2 sind es die Klammern.

Nicht-Terminalzeichen : vorläufige Zeichen; diese treten innerhalb einer Herleitung, aber nicht in einer endgültig abgeleiteten Zeichenfolge auf; im Beispiel 2 ist es das Zeichen S.

Im Jahre 1959 stellte Noam Chomsky ein Beschreibungssystem für formale Sprachen vor, mit dem wir uns in Zukunft beschäftigen werden.

Man geht von einem einzelnen Startsymbol ***S*** aus. Dieses ist kein Terminalzeichen. Es muss ersetzt werden und gehört damit zur Menge ***N*** der ***Nichtterminalsymbole***. Die gängigen Notationen von formalen Sprachen verwenden für die Nichtterminalsymbole in der Regel Großbuchstaben.
Das Startsymbol wird durch eine Symbolfolge ersetzt, die aus Terminalsymbolen sowie weiteren Nichtterminalsymbolen bestehen kann, die wiederum zu ersetzen sind. Solange das neue Wort also Nichtterminalsymbole enthält, ist es noch nicht „fertig".

Die Ersetzungen erfolgen nach Regeln, die insgesamt ein Regelsystem ***R*** bilden. Regeln können in der Form *Nichtterminalsymbol* → *Symbolfolge* aufgeschrieben werden.
Die Grammatik einer formalen Sprache wird damit durch ein 4-Tupel festgelegt:
G = (T, N, S, R)

Bemerkung: Das Regelsystem R wird oft auch mit dem Buchstaben P (Produktionsregeln) abgekürzt.
Die Reihenfolge in dem 4-Tupel wird in der Literatur unterschiedlich gewählt.

Formale Sprache:

Wenn T ein Alphabet, also eine endliche Menge von Zeichen ist, dann ist T^* die Menge aller endlichen Zeichenfolgen über T (einschließlich der leeren Zeichenfolge). Jede Teilmenge L von T^* heißt Formale Sprache.

Wenn G eine Grammatik mit T als Menge der Terminalzeichen ist, so ist L(G) die Menge aller Zeichenfolgen über T, welche mit Hilfe der Ersetzungsregeln aus G erstellt werden können. L(G) ist also die von G erzeugte formale Sprache.

Eine formale Grammatik erlaubt es, zu unterscheiden, ob ein Wort dieser Grammatik folgt, also ob es „gültig“ ist. Die Menge aller gültigen Worte nennt man die Sprache dieser Grammatik L(G). Umgekehrt erlaubt es eine Grammatik auch, alle Wörter ihrer Sprache zu erzeugen.

Beispiel 3: Ausgewogene Bitmuster

Über den Terminalzeichen der Menge {0;1} wird eine Sprache betrachtet, bei der jedes Wort gleich viele Einsen wie Nullen enthalten muss. Die Worte 01, 10, 001011 oder 110010 gehören dazu. Als Nicht-Terminalzeichen dienen S, A und B, wobei S das Startzeichen ist. Dabei sollen aus A alle Wörter herleitbar sein, die genau eine Eins mehr als Nullen enthalten und aus B soll man alle Wörter erhalten, die genau eine 0 mehr als Einsen besitzen. Das führt zu folgenden Grammatikregeln **R** :
(1) S → 0A | 1B (2) A → 1| 1S| 0AA (3) B → 0| 0S| 1BB

Das Wort 110010 kann dann wie folgt abgeleitet werden:
S → 1B →11BB → 11B0 → 110S0→1100A0 →110010
Alternativ könnte man auch folgende Ableitung benutzen:
S → 1B →11BB → 110B → 1100S→11001B →110010

Es kann also vorkommen, dass bzgl. einer formalen Grammatik für ein Wort mehrere unterschiedliche Ableitungen existieren.

Definitionen:
Eine Grammatik heißt ***eindeutig***, wenn es zu jedem Wort aus der erzeugten Sprache nur eine Ableitung gibt.
Eine Ableitung heißt ***Linksableitung***, wenn in jedem Schritt nur das am weitesten links stehende Nichtterminalsymbol ersetzt wird.

Man kann durch kurze Überlegung einsehen, dass jedes mit diesen Regeln erzeugte Wort wirklich gleich viele Einsen und Nullen besitzt. Es fehlt hier allerdings der Beweis, dass sich auch jedes Wort mit gleich vielen Einsen und Nullen mit diesen Regeln erzeugen lässt.

Bemerkung: Die Sprache der ausgewogenen Bitmuster lässt sich auch mit den folgenden Produktionsregeln **P** herleiten (ohne Beweis):
$S \rightarrow 01 \mid 10 \mid SS \mid 0S1 \mid 1S0$

Beispiel: $S \rightarrow 1S0 \rightarrow 1\ SS\ 0 \rightarrow 1S\ 01\ 0 \rightarrow 1\ 10\ 010$ oder alternativ:
$S \rightarrow SS \rightarrow 1S0\ S \rightarrow 1\ 10\ 0S \rightarrow 1100\ 10$

Folgerung: Es gibt also Sprachen, die man mit völlig unterschiedlichen Regeln, und damit mit unterschiedlichen Grammatiken erzeugen kann. In diesem Zusammenhang stellt sich gleich die Frage: Wie findet man für eine vorgegebene Sprache eine minimale Grammatik?

Definition:
Zwei Grammatiken heißen **äquivalent,** wenn sie die gleiche formale Sprache erzeugen.

Bemerkung: Man kann eine Ableitungsregel der Form $S \rightarrow \varepsilon$ zulassen, wenn auch das leere Wort ε zur Sprache gehören soll. Hierbei ist S das Startsymbol. Eine Ableitungsregel der Form $A \rightarrow \varepsilon$ (wobei A nicht das Startsymbol sein soll) wird in der Literatur einerseits oft benutzt, andererseits aber auch genauso oft abgelehnt.

Aufgabe: Leite für die Sprache der ausgewogenen Bitmuster jeweils die beiden Worte 000111 und 111000 ab, einmal mit den Grammatikregeln R und einmal mit den Grammatikregeln P !

Als Beispiel betrachten wir die Sprache der natürlichen Zahlen.

T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
N = {S}, weitere Nichtterminalsymbole benötigen wir hier nicht.
R = {S→1, S→2, S→3, S→4, S→5, S→6, S→7, S→8, S→9}

oder kürzer geschrieben: {S→1| 2| 3| 4| 5| 6| 7| 8| 9}

Für größere Zahlen benötigen wir noch weitere Regeln:

{S→S0| S1| S2| S3| S4| S5| S6| S7| S8| S9}

Interessant ist, dass das Startsymbol auch auf der rechten Seite der Ersetzungsregeln auftaucht. Man erhält rekursive Regeln.

Die Sprache der natürlichen Zahlen lässt sich damit durch folgende Grammatik beschreiben:
$G_{\text{natürliche Zahlen}}$ = (T, N, S, R) mit
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
N = {S}
R = {S→1| 2| 3| 4| 5| 6| 7| 8| 9|S0| S1| S2| S3| S4| S5| S6| S7| S8| S9}

Die Zahl 123 z.B. entsteht durch folgende Ableitung:
S→S3→S23→123

Interessant: Ein Terminalsymbol (im obigen Beispiel die Null) muss nicht unbedingt zur Sprache gehören! Mach dir klar, dass sich die Null nicht aus dem Startsymbol ableiten lässt!

Eine Sprache lässt sich durchaus durch unterschiedliche Grammatiken beschreiben. Die gerade beschriebene Grammatik für die Sprache der natürlichen Zahlen ist eine sog. linksreguläre Sprache. Einige Seiten weiter werden wir eine rechtsreguläre Sprache für die natürlichen Zahlen kennenlernen.

Ein wichtiges Thema in der theoretischen Informatik ist das sog. **Wortproblem:** ***Gibt es einen Algorithmus, der feststellt, ob ein beliebiges Wort zu einer bestimmten Sprache gehört – oder nicht?***

Die Antwort auf diese Frage hängt davon ab, wie schwierig die betreffende formale Sprache ist bzw. welche Produktionsregeln es für diese Sprache gibt.

Sprachklassen

Die formalen Sprachen lassen sich anhand des Aufbaus der Produktionsregeln in unterschiedliche Sprachklassen einordnen, von denen wir hier drei aufführen:

In ***kontextsensitiven*** Sprachen dürfen Ersetzungen nur vorgenommen werden, wenn das Nichtterminalsymbol in einem bestimmten Kontext auftaucht. Formal läßt sich das so schreiben: $\alpha A \beta \rightarrow \alpha \gamma \beta$ wobei A ein Nichtterminalsymbol ist und α, β und γ Wörter bestehend aus Terminalen und Nichtterminalen sind. Dabei bilden α und β den Kontext, in dem die Ersetzung vorgenommen werden darf. Dabei können α und β auch das leere Wort ε darstellen. Daher ist die Klasse der kontextsensitiven Sprachen eine Oberklasse der kontextfreien Sprachen.

In ***kontextfreien*** Sprachen dürfen Ersetzungen ohne Berücksichtigung des Umfeldes vorgenommen werden, also z.B. $A \rightarrow \alpha$ wobei A ein Nichtterminalsymbol ist und α ein aus Terminalen und Nichtterminalen bestehendes Wort ist. Für die Symbolfolge α gelten keine weiteren Einschränkungen. Ein Nichtterminalsymbol kann also insbesondere durch eine beliebig lange Folge von Nichtterminal- und Terminalsymbolen ersetzt werden.

In ***regulären*** Sprachen darf das Nichtterminalsymbol entweder durch das leere Wort $\boldsymbol{\varepsilon}$ oder durch ein **einziges** Terminalsymbol oder durch eine Folge aus **einem** Nichtterminal- und **einem** Terminalsymbol ersetzt werden.
Steht das Nichtterminalsymbol in dieser Folge immer links, so spricht man von einer ***linksregulären*** Sprache.

Alle Regeln einer linksregulären Sprache sind also von einer der folgenden Formen:
$A \rightarrow \boldsymbol{\varepsilon}$ oder $A \rightarrow a$ oder $A \rightarrow Ba$

In der Literatur ist die Definition einer regulären Sprache nicht ganz eindeutig. Oft fehlt auch in der Definition die Regel $A \rightarrow \boldsymbol{\varepsilon}$. Das ist für die Sprachklasse aber unwichtig. Natürlich entscheidet das Vorhandensein der Regel $\mathbf{S} \rightarrow \varepsilon$, ob das leere Wort zur Sprache gehört oder nicht. Ansonsten beeinflusst das Vorhandensein oder Fehlen dieser Regel nur die Anzahl der Produktionsregeln.

Satz: **Jede reguläre Sprache lässt sich auch als linksreguläre Sprache darstellen.**
Jede reguläre Sprache lässt sich auch als rechtsreguläre Sprache darstellen. *(ohne Beweis)*

Sowohl die Sprache der *wohlgeformten Klammerausdrücke* als auch die Sprache der *ausgewogenen Bitmuster* sind nicht regulär. (Da z.B. die Anzahl der öffnenden Klammern beliebig groß sein kann, müsste ein entsprechender Akzeptor beliebig weit zählen können). Wenn eine Programmiersprache beliebig viele Klammern zulässt, kann sie nicht regulär sein.

Satz: **Die zu deterministischen endlichen Akzeptoren gehörenden Sprachen sind genau die regulären Sprachen.**

Wir werden diesen Satz in Kürze dadurch beweisen, indem wir einerseits für einen beliebigen endlichen Akzeptor eine zugehörige reguläre Grammatik ermitteln und andererseits für jede beliebige reguläre Sprache einen zugehörigen endlichen Akzeptor ermitteln.

Reguläre Ausdrücke, also Worte einer regulären Sprache, lassen sich leicht beschreiben, wenn man folgende Kurzschreibweise benutzt:

ab	das Zeichen a gefolgt vom Zeichen b
a|b	das Zeichen a oder das Zeichen b
a^*	das Zeichen a beliebig oft (auch keinmal)
$(ab)^*$	die Zeichenfolge ab beliebig oft
aa^*	beliebig viele aber mindestens ein a
$(a^*b^*)^*$	beliebig viele a und b in beliebiger Reihenfolge

Aufgabe: Gib zu folgenden regulären Sprachen über dem Alphabet {0, 1} passende reguläre Ausdrücke in obiger Kurzschreibweise an:

a) {w| w beginnt mit 1 und endet mit 0}
b) {w| w enthält mindestens 3 Einsen}
c) {w| w enthält den Teilstring 0101 }

Lösung

a) $1(0^*1^*)^*0$

b) $0^*10^*10^*1(0^*1^*)^*$

c) $(0^*1^*)^*0101(0^*1^*)^*$

Eine Sprache L sei gegeben durch T = {a, b}, N = {S} und durch die Produktionen P = {S → ε | ab | SS}
Es ist leicht zu sehen, dass L = {ε, ab, abab, ababab, ...}

Eine äquivalente reguläre Grammatik für diese Sprache L wäre gegeben durch T = {a, b}, N = {S, A} und durch die Produktionen
R = {S → aA | ε; A → bS}

Diese Sprache L ist also rechtsregulär.

Bemerkung: Es ist also in manchen Fällen möglich, zu einer nicht-regulären Grammatik eine äquivalente reguläre Grammatik zu finden.

Aufgaben

1. Finde eine Grammatik für die Menge aller Bezeichner einer Quellsprache. Bezeichner müssen mit einem Buchstaben beginnen.
Es sei T = {a, b, …., z, 0, 1, …, 9}
Gib eine Ableitung für den Bezeichner „gk132“ an!

Aufgaben zu den kontextfreien Sprachen

2. Konstruiere eine nicht-reguläre Grammatik, welche die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ erzeugt.

3. Eine Satzgliederungsgrammatik hat folgende Regeln (Nichtterminalzeichen sind durch spitze Klammern gekennzeichnet):

<Satz>	→	<Nominalphrase> <Verbalphrase>
<Nominalphrase>	→	<Eigenname> | <Artikel> <Substantiv>
<Verbalphrase>	→	<Verb> | <Verb> <Nominalphrase>
<Eigenname>	→	Anne | Bernd | Carla
<Substantiv>	→	Katze | Pferd | Heu | Buch | Bild | Zimmer
<Artikel>	→	der | die | das
<Verb>	→	jagt | frisst | liest | betritt

Leite die Sätze „Anne liest“ und „das Buch jagt die Katze“ ab!

4. Konstruiere eine möglichst einfache Grammatik, welche alle Terme mit Klammern, einstelligen Dezimalzahlen und den vier Grundrechenarten erzeugt. Beispiel: 7+(3+4)*(5-(2+(3+4)))*(6-(8:4))

5. Finde mit der Grammatik aus Aufgabe 4 zwei echt verschiedene Ableitungen für den Term 3+4*5
Zeichne auch die zugehörigen Syntaxbäume!

6. Ergänze die Grammatik aus Aufgabe 3 durch folgende Zusatzregeln:

<Nominalphrase> → <Artikel> <Substantiv> <Präpositionalphrase>
<Verbalphrase> → <Verb> <Nominalphrase> <Präpositionalphrase>
<Präpositionalphrase> → <Präposition> <Nominalphrase>
<Präposition> → mit | in | auf | unter
<Artikel> → dem | den

Leite nun den Satz „Anne betritt das Zimmer mit dem Bild“ mit zwei verschiedenen Syntaxbäumen ab! Was bedeuten die beiden Varianten?

7. Gegeben ist die Grammatik G = {T, N, S, P}. Dabei gilt:
T = {a, b, p, q, IF, THEN, ELSE}, wobei a und b Anweisungen und p, q Bedingungen darstellen sollen.
N = {S, B}, wobei S stellvertretend für Anweisung und B stellvertretend für Bedingung steht.
P = {S→a| b| IF B THEN S| IF B THEN S ELSE S; B→p| q}

Zeige, dass die Anweisung IF p THEN IF q THEN a ELSE b zwei verschiedene Syntaxbäume besitzt! Wie sind diese zu interpretieren?

8. Gib zu den Beispielen aus den Aufgaben 4, 6 und 7 jeweils **eindeutige** Grammatiken an!

9. Betrachte folgende Grammatik G: ({a, b}, {S, T, X}, S, P), wobei
P = { S→aTb| bTa; T→XTX| X| ε; X→a| b}

a) Gib 3 Wörter aus L(G) an!
b) Gib 3 Wörter aus $\{a, b\}^*\backslash L(G)$ an!

10. Die nachfolgende Grammatik G: ({a, +, *, (,)}, {E, T, F}, E, R), mit
R = {E→E + T| T; T→T*F| F; F→(E)| a} beschreibt eine einfache Form arithmetischer Ausdrücke. Das Terminalsymbol a steht dabei für eine beliebige Konstante, E für Expression (Ausdruck), T für Term und F für Faktor.
Gib für a*a+a+(a) eine Linksableitung und den Ableitungsbaum an!

Lösungen

1. G = (T, N, S, R)
T = {a, b, …., z, 0, 1, …, 9}
N = {S, A}
R = {A → ε | 0A | 1A | ... | 9A | aA | bA | ... | zA ; S → aA | bA | ... | zA}

„gk132“: S → gA → gkA → gk1A → gk13A → gk132A → gk132

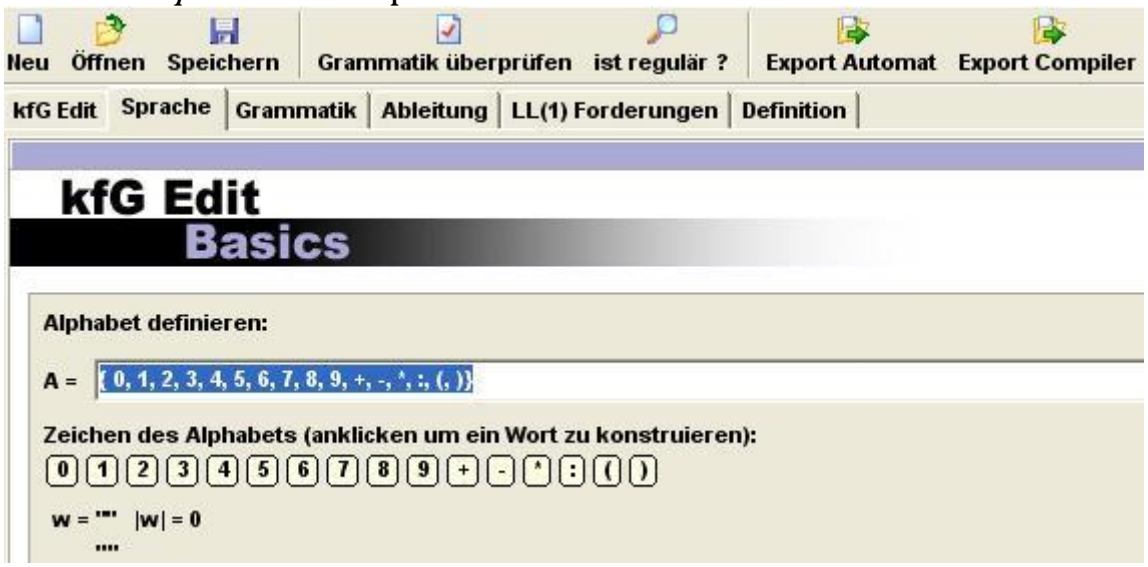
2. G = (T, N, S, R)
N = {S}
R = {S → ab | aSb}

3. „Anne liest“:
<Satz> → <Nominalform> <Verbalphrase>
→ <Eigenname> <Verbalphrase>
→ Anne <Verbalphrase>
→ Anne <Verb>
→ Anne liest

„das Buch jagt die Katze“:
<Satz> → <Nominalform> <Verbalphrase>
→ <Artikel> <Substantiv> <Verbalphrase>
→ das Buch <Verb> <Nominalphrase>
→ das Buch jagt <Artikel> <Substantiv>
→ das Buch jagt die Katze

4. G = (T, N, S, R)
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, :, (,)}
N = {S, Z}
R = {S → (S) | Z | S+S | S-S | S*S | S:S ; Z → 0| 1| 2| ... | 9}

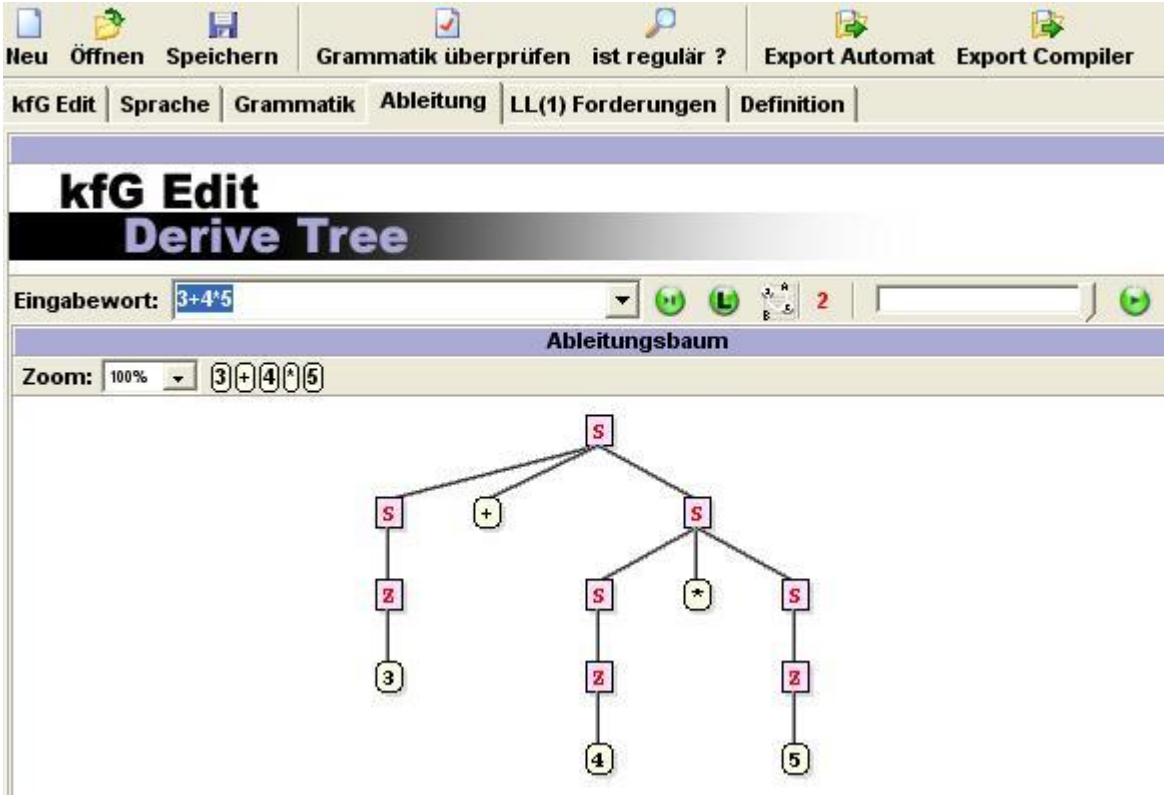
5. Zunächst legt man im kfG-Editor (kfG = kontextfreie Grammatik) im Menü *Sprache* das Alphabet fest:



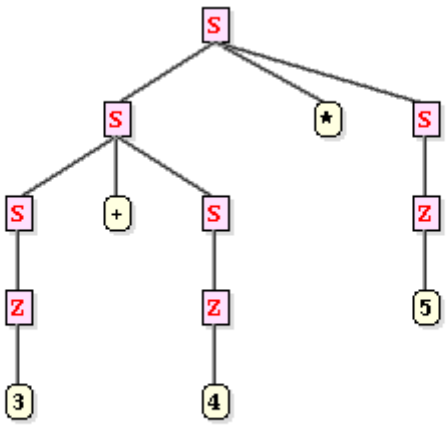
Danach gibt man im Menü *Grammatik* die Produktionsregeln ein:

Der Pfeil wird einfach durch ein Minuszeichen, gefolgt von einem Größerzeichen, dargestellt. Terminalsymbole werden in einfachen Hochkommata geschrieben. Sie erscheinen schwarz.

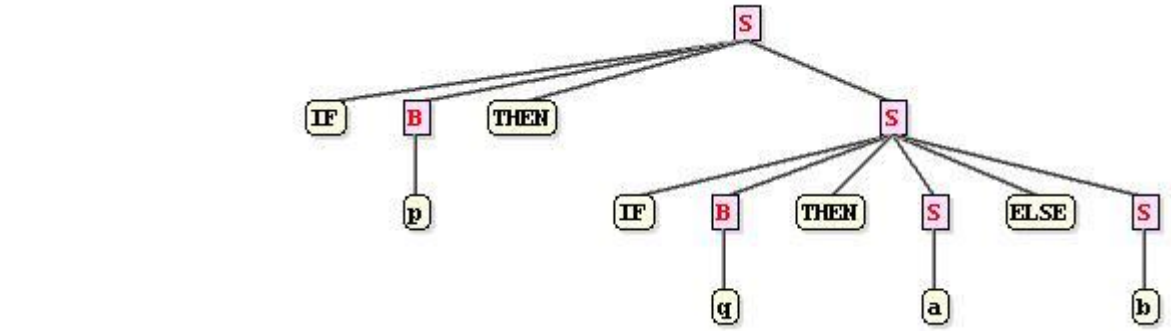
Anschließend kann man im Menue *Ableitung* ein Wort eingeben. Es wird in derselben Zeile angezeigt, wie viele unterschiedliche Ableitungen es gibt (im Beispiel signalisiert das die rote 2). Unter diesen kann man auswählen.

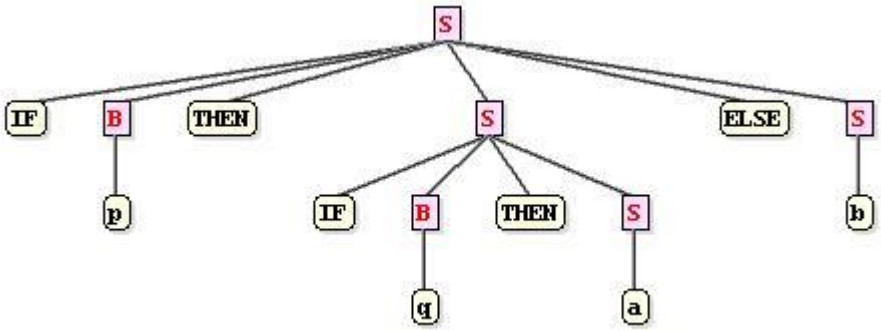


Der andere Syntaxbaum sieht so aus:



7.





8. Eindeutige Lösungen zu Aufgabe 4:

P = {T → S| S + S| S − S;
S → F| F * F| F : F| (T);
F → 0|1|2|3|4||5|6|7|8|9}

Dabei ist T das Startsymbol. T bedeutet Term, S Summand und F Faktor

Alternative Möglichkeit:

P = {T → T_1| T_2;
T_1 → Z| (T)| T_1 R_1 T_1;
T_2 → Z| (T_2)| T R_2 T;
R_1 → *| : ;
R_2 → +| - }

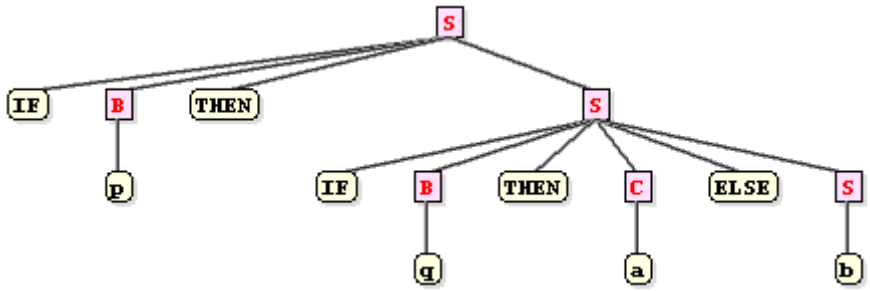
Eindeutige Lösungen zu Aufgabe 7:
kontextsensitive Lösung:

```
P = {S → a|b|IF B THEN S;
     THEN S → THEN S ELSE S;
     B → p|q}
```

Damit folgt die eindeutige Linksableitung:

S → IF B THEN S
→ IF p THEN S
→ IF p THEN IF B THEN S
→ IF p THEN IF q THEN S ELSE S
→ IF p THEN IF q THEN a ELSE S
→ IF p THEN IF q THEN a ELSE b

kontextfreie Lösung:
P = {S → a| b| IF B THEN S| IF B THEN C ELSE S;
B → p| q;
C → a| b| IF B THEN C ELSE S }
Damit folgt die eindeutige Ableitung als Syntaxbaum:



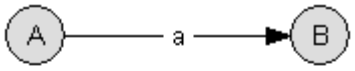
9. Bei allen Worten aus L(G) sind Anfangs- und Endbuchstabe unterschiedlich. Außerdem gehört das leere Wort **ε** nicht zu L(G).

Ermittlung der Sprache eines Akzeptors

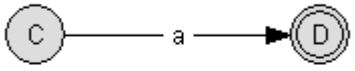
Folgende Größen bei endlichen Automaten und regulären Grammatiken entsprechen sich gegenseitig:

Endlicher Automat	Reguläre Grammatik
Eingabealphabet	Terminalsymbole
Zustandsmenge	Nichtterminalsymbole
Anfangszustand	Startsymbol
Überführungsfunktion	Produktionsregeln

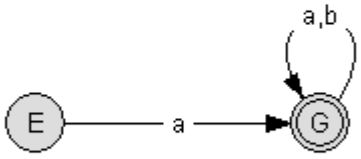
Mit nachfolgenden Regeln lässt sich aus einem endlichen Automaten eine rechtsreguläre Grammatik erzeugen. Aus Vereinfachungsgründen werden hier für die Elemente der Zustandsmenge dieselben Namen benutzt wie für die Elemente der Nichtterminalsymbole.



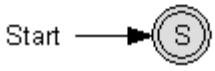
{A → aB}



{C → a | aD}
alternativ: {C→aD; D → ε}



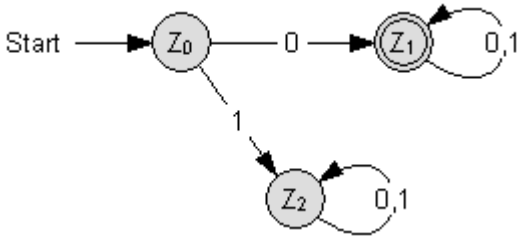
{E → a | aG; G → a | b | aG | bG}
alternativ: {E → aG; G → ε | aG | bG}



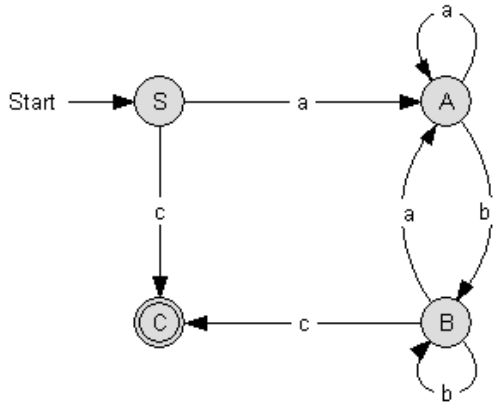
{S → ε}

Aus Übergängen in einen Fehlerzustand folgen natürlich keine Produktions-regeln!

Aufgabe:
Ermittle eine zum nebenstehenden Automaten passende Grammatik!



Der folgende Akzeptor erkennt offensichtlich reguläre Ausdrücke vom Typ $(aa^*bb^*)^*c$. Die zugehörige Sprache heiße L_{abc} .



Alle nicht eingezeichneten Übergänge enden im Fehlerzustand F.

Nun soll eine zu diesem Automaten passende rechtsreguläre Grammatik G = (T, N, S, R) hergeleitet werden.

Das Startsymbol S entspricht dem Startzustand S.
Die Menge der Terminalsymbole sind die Übergangszeichen, also das Eingangsalphabet des Automaten: T = {a, b, c}
Die Menge der Nichtterminalsymbole sind die Zustandsbezeichnungen des Automaten, also N = {S, A, B, C}

Die Produktionsregeln geben an, wie man von einem Zustand zum nächsten kommt. Sie entsprechen also der Übergangsfunktion des Automaten. Übergänge in einen möglichen Fehlerzustand liefern dabei natürlich keine Produktionsregeln für die Sprache des Akzeptors.
Die Produktionsregeln kann man auf zwei Arten formulieren: mit Übergängen zum leeren Wort ε oder ohne Übergänge zum leeren Wort ε.
Für das obige Beispiel gilt:

Regeln mit Übergängen zum leeren Wort ε:
R = {S→aA| cC; A→aA| bB; B→bB| aA| cC; C→ε }
Für Endzustände gibt es immer die zusätzliche Regel: Endzustand→ε

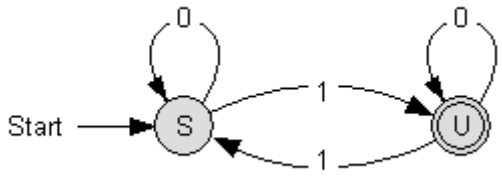
Regeln ohne Übergänge zum leeren Wort ε:
P = {S→aA| cC| c; A→aA| bB; B→bB| aA| cC| c}
In diesem Fall gibt es bei jedem Übergang in einen Endzustand noch die zusätzliche Regel mit einem Übergang zu einem Terminalsymbol.
Weil es hier (in dieser Aufgabe) keinen erlaubten Übergang mehr aus dem Endzustand heraus gibt, sind (bei dieser Regel ohne Übergang zum leeren Wort ε) auch die Regeln S→cC und B→cC überflüssig.
Es genügt also: P = {S→aA| c; A→aA| bB; B→bB| aA| c}

Wenn man untersuchen will, ob das Wort *abbbabc* von obiger Grammatik erzeugt werden kann, so gibt man die Folge der zugehörigen Produktionen an. Dabei wird immer ein Zustand durch eine Kombination aus Zeichen und Zustand ersetzt.
S→aA→abB→abbB→abbbB→abbbaA→abbbabB→abbbabcC→abbbabc

Obige Folge nennt man auch eine ***Ableitung*** oder ***Ableitungsfolge*** für das Wort *abbbabc*.

Gehört ein Wort zur Sprache des (deterministischen) Automaten, so gibt es natürlich nur einen einzigen Weg durch den Zustandsgraphen, um dieses Wort zu erzeugen. Wenn man also, wie oben beschrieben, aus einem endlichen Automaten eine rechtsreguläre Grammatik erzeugt, so erhält man immer eine eindeutige Grammatik!

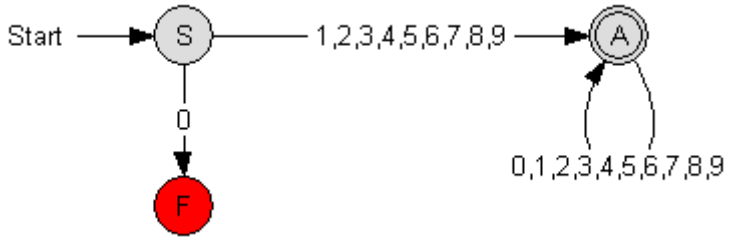
Beispiel: Es sei L die Sprache aller Binärausdrücke mit ungerader Parität, d.h. die Anzahl der Einsen ist ungerade.



Produktionsregeln mit Übergang zum leeren Wort ε:
R = {S→0S| 1U; U→1S| 0U| ε}

Produktionsregeln ohne Übergang zum leeren Wort ε:
P = {S→0S| 1U| 1; U→1S| 0U| 0}

Im Folgenden wird eine rechtsreguläre Grammatik G = (T, N, S, R) für die Menge der natürlichen Zahlen $\mathbb{N} = \{1, 2, 3, \ldots\}$bestimmt, indem man zunächst einen zugehörigen endlichen Automaten erstellt.



Das Startsymbol S entspricht dem Startzustand S.
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
N = {S, A}
R = {S→1A|2A|3A|4A|5A|6A|7A|8A|9A,
A→ε|0A|1A|2A|3A|4A|5A|6A|7A|8A|9A}

bzw. ohne ε –Übergänge:
R = {S→1| 2| 3| 4| 5| 6| 7| 8| 9| 1A|2A|3A|4A|5A|6A|7A|8A|9A,
A→0|1| 2| 3| 4| 5| 6| 7| 8| 9|0A|1A|2A|3A|4A|5A|6A|7A|8A|9A}

Beachte, dass in diesem Beispiel aus dem Endzustand heraus durchaus weitere, sinnvolle Übergänge erfolgen können! Das wird bei den Produktionsregeln berücksichtigt.

Eine Ableitung für die Beispielszahl 203 wäre: S→2A→20A→203A→203

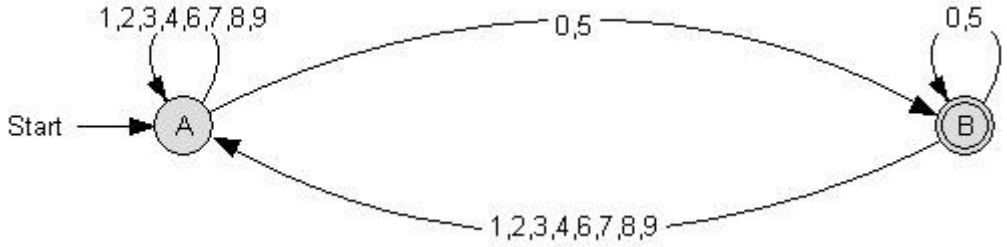
Zustände und Übergänge, die auf keinem Weg zum Endzustand vorkommen, brauchen natürlich auch für die Produktionsregeln nicht berücksichtigt zu werden.

Bemerkung: auf der Seite 56 wurde eine linksreguläre Grammatik für die Menge der natürlichen Zahlen angegeben.

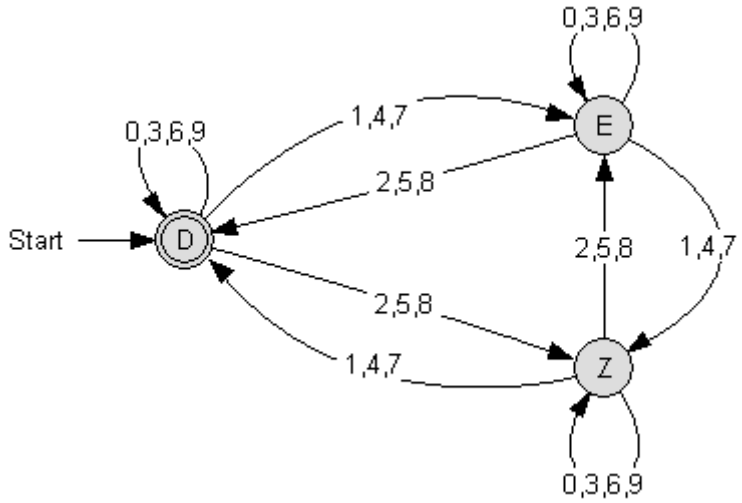
Aufgaben

1. Untersuche, ob es für die auf Seite 68 dargestellte Sprache L_{abc} Ableitungen gibt für die Worte
   a) *abbac*
   b) *aaabbbc*
   c) *abc*
   d) *bbabc*

2. Gib eine rechtsreguläre Grammatik G zu folgendem Automaten an. Prüfe durch Angabe der Ableitungen, ob die Worte *23540* und *123456* von der Grammatik erzeugt werden können.



3. Gib eine rechtsreguläre Grammatik G zu folgendem Automaten an. Prüfe durch Angabe der Ableitungen, ob die Worte *23540* und *123456* von der Grammatik erzeugt werden können.



4. Ein Computerpasswort soll aus mindestens 3 Zeichen bestehen, bei denen mindestens eine Ziffer (0 oder 1) und ein Buchstabe (a oder b) vorkommen soll. Entwickle den zugehörigen Akzeptor mit dem Eingabealphabet E = {0, 1, a, b}und einer dazu passenden rechtsregulären Grammatik.

5. Zeichne das Zustandsdiagramm eines Automaten, der ganze Zahlen akzeptiert, also z.B. -23, 13, +45, 0. Führende Nullen sollen nicht akzeptiert werden.
Gib anschließend eine zugehörige rechtsreguläre Grammatik an!

6. a) Ein Blumenautomat soll 5€ in 1€- oder 2€-Stücken akzeptieren. Mögliche Eingaben wären also z.B. 1121 oder 212. Entwirf einen Automaten für diese Sprache (nur Zustandsgraph)!
b) Entwickle aus diesem Akzeptor eine reguläre Grammatik G für die akzeptierte Sprache!
c) Wenn der Automat auch 10-Cent-Stücke akzeptiert, wird die Sprache durch die vielen möglichen Reihenfolgen der Eingaben deutlich komplexer. Entwirf eine kontextsensitive, aber nicht kontextfreie Grammatik für diese Sprache.
d) Begründe, ob die in Teilaufgabe c beschriebene Sprache auch kontextfrei oder sogar regulär ist!

7. Gib für die Sprache $L = \{w \in \{0, 1\}^* \mid w \text{ enthält mindestens 3 Einsen}\}$ eine Grammatik an!

8. Es soll eine reguläre Sprache entworfen werden, die aus Bitfolgen besteht, welche mindestens 2 Bit lang sind und sowohl mit einer 1 beginnen als auch mit einer 1 enden.
a) Entwirf den Zustandsgraphen zu dem entsprechenden Automaten!
b) Erläutere die Bedeutung der Zustände dieses Automaten!
c) Gib die zugehörige Grammatik an!
d) Leite aus dem Startsymbol dieser Grammatik die Nachricht 1101 ab!

9. Es sei $L=\{w \in \{a,b,c,\ldots,z\}^* \mid w=das \text{ oder } w=was\}$
Gib den Zustandsgraphen eines passenden endlichen Akzeptors an und leite damit auch eine zugehörige Grammatik her!

10. Schreibe eine Grammatik, die nur gerade Binärzahlen produziert!

Lösungen

1.a) für das Wort abbac gibt es keine Ableitung. Begründung:
S→aA→abB→abbB→abbaA weiteres Ableiten ist nicht möglich.

b) Ableitung für aaabbbc:
S→aA→aaA→aaaA→aaabB→aaabbB→aaabbbB→aaabbbcC→aaabbbc

c) Ableitung für abc:
S→aA→abB→abcC→abc

d) es gibt keine Ableitung für das Wort bbabc

2. G = (T, N, S, R) mit
S = A
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
N = {A, B}
R = {A→0B|5B|1A|2A|3A|4A|6A|7A|8A|9A;
B→ε|0B|5B|1A|2A|3A|4A|6A|7A|8A|9A}

Ableitung für 23540: A→2A→23A→235B→2354A→23540B→23540
Es gibt keine Ableitung für 123456:
A→1A→12A→123A→1234A→12345B→123456A

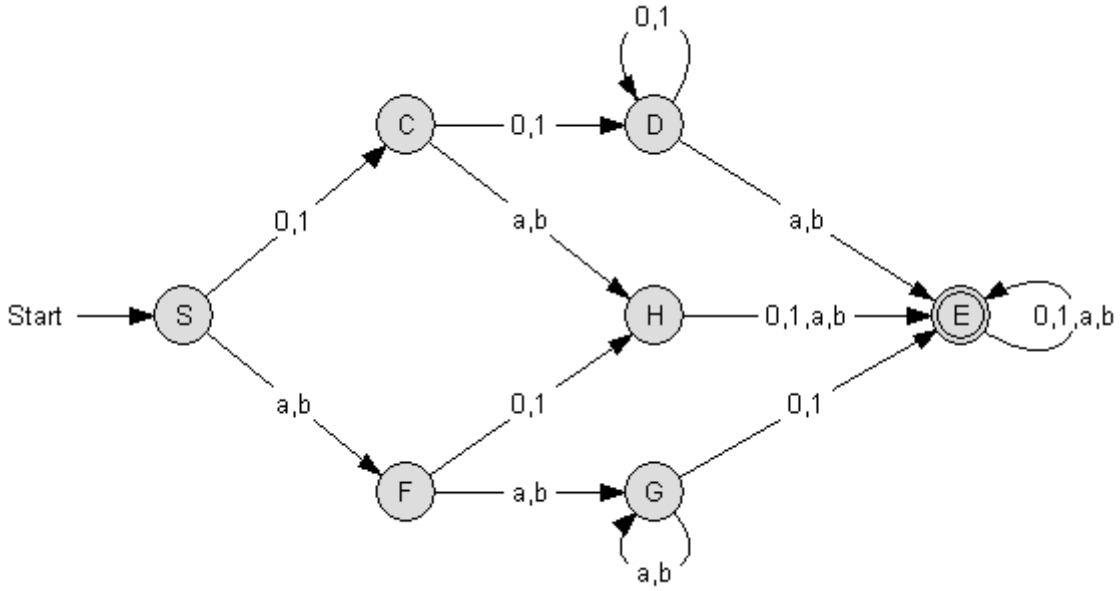
Bemerkung: Die Terminalsymbole 1,2,3,4,6,7,8,9 gehören nicht zur Sprache.

3. G = (T, N, S, R) mit
S = D
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
N = {D, E, Z}
R = {D→ε|0D|3D|6D|9D|1E|4E|7E|2Z|5Z|8Z;
E→0E|3E|6E|9E|1Z|4Z|7Z|2D|5D|8D;
Z→0Z|3Z|6Z|9Z|1D|4D|7D|2E|5E|8E}

Es gibt keine Ableitung für 23540: D→2Z→23Z→235E→2354Z→23540Z
Ableitung für 123456:
D→1E→12D→123D→1234E→12345D→123456D→123456

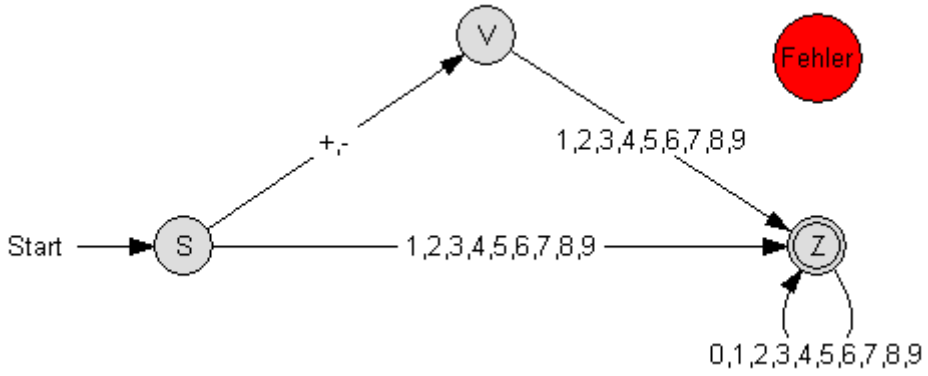
Bemerkung: Das leere Wort gehört zur Sprache.

4.



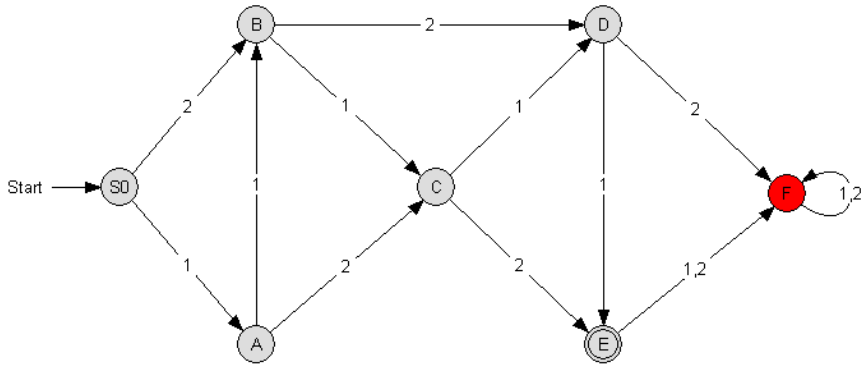
G = (T, N, S, R) mit
S = S
T = {0, 1, a, b}
N = {S, C, D, E, F, G, H}
R = {S→0C|1C|aF|bF, C→0D|1D|aH|bH, D→0D|1D|aE|bE,
F→0H|1H|aG|bG, G→0E|1E|aG|bG, H→0E|1E|aE|bE,
E→ε|0E|1E|aE|bE}

5.



Alle nicht eingezeichneten Übergänge führen in den Fehlerzustand.
G = (T, N, S, R) mit
S = S
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -}
N = {S, V, Z}
R = {S→+V|-V|1|2|3|4|5|6|7|8|9|1Z|2Z|3Z|4Z|5Z|6Z|7Z|8Z|9Z,
V→1|2|3|4|5|6|7|8|9|1Z|2Z|3Z|4Z|5Z|6Z|7Z|8Z|9Z ,
Z→ε|0Z|1Z|2Z|3Z|4Z|5Z|6Z|7Z|8Z|9Z }

6. a)

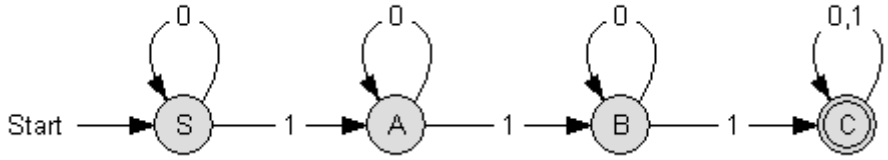


b)
G = { N, T, S, R}
T = {1, 2, c}
N = {S, A, B, C, D, E}
R = {S→1A| 2B;
A→1B|2C;
B→1C| 2D;
C→1D| 2;
D→1}

c)
G = {N, T, S, R}
T = {1, 2, c}
N = {S, E, Z, V}
R = {S → EV| VE;
V → EZE| ZZ;
Z → EE| 2;
E → 1|ccccccccc;
cE → Ec
cZ → Zc }

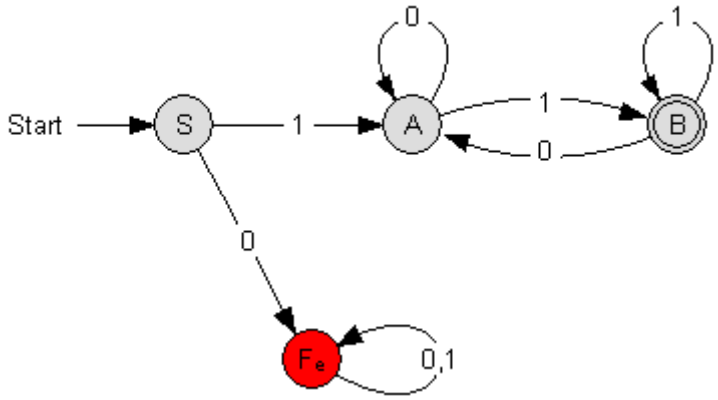
d) Für die in Teilaufgabe c beschriebene Sprache könnte man einen endlichen deterministischen Akzeptor entwickeln. Dieser müsste für jeden möglichen, eingezahlten Geldbetrag einen Zustand besitzen. Das wären 52 Zustände (entsprechend 0, 10, 20, 30,…490, 500 Cent und noch ein Fehlerzustand für zu viel eingezahltes Geld). Die zugehörige Grammatik besäße entsprechend viele Produktionsregeln, aber sie wäre kontextfrei, sogar rechtsregulär. Die zugehörige Sprache ist deshalb auch regulär.

7.



G = {N, T, S, R}
T = {0, 1}
N = {S, A, B, C}
R = {S → 0S| 1A;
A → 0A| 1B;
B → 0B| 1C;
C → 0C| 1C| ε }

8. a)



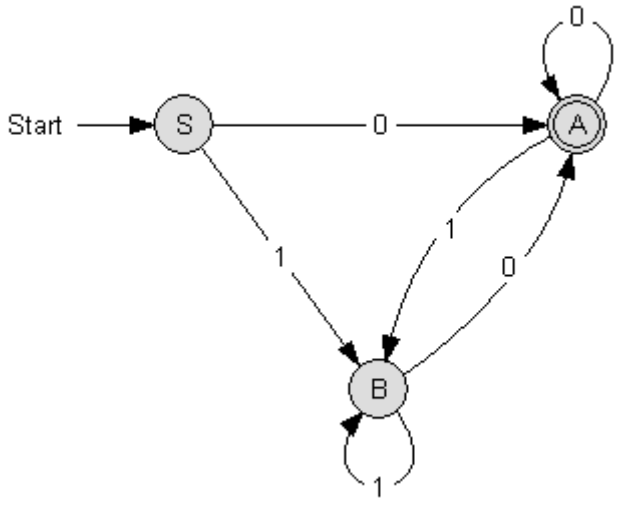
c) G = {N, T, S, R}
T = {0, 1}
N = {S, A, B}
R = {S → 1A;
A → 0A| 1B;
B → 0A| 1B;| ε }

d) Ableitung des Wortes 1101:
S → 1A → 11B →110A → 1101B → 1101

9. Alle nicht eingezeichneten Übergänge führen in den Fehlerzustand.

G = {N, T, S, R}
T = {a..z}
N = {S, A, B, C}
R = {S → dA| wA;
A → aB;
B → sC;
C → ε }

10.



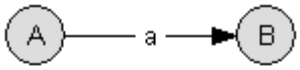
G = {N, T, S, R}
T = {0, 1}
N = {S, A, B}
R = {S → 0A| 1B;
A → 0A| 1B| ε;
B → 0A| 1B; }

Ermittlung des Akzeptors für eine Sprache

Aus den Produktionsregeln einer rechtsregulären Sprache lässt sich umgekehrt auch sehr einfach das Zustandsdiagramm eines zu dieser Sprache gehörenden Automaten ermitteln.

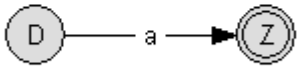
Aus Vereinfachungsgründen werden hier für die Elemente der Zustandsmenge dieselben Namen benutzt wie für die Elemente der Nichtterminalsymbole.

{A → aB}



{C → a | aC}

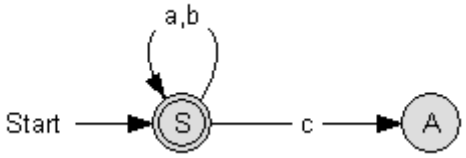
{D → a}



{E → ε}

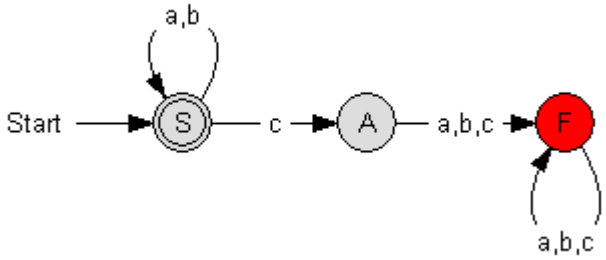
Beispiel:
Die folgende Sprache enthält alle Worte, die nur aus den Zeichen a und b gebildet sind, aber kein c enthalten. Außerdem gehört das leere Wort ebenfalls zur Sprache.
T = {a, b, c}
N = {S, A, F}
R = {S→ε|aS|bS|cA}

Nach einiger Überlegung findet man recht schnell den folgenden Automaten:

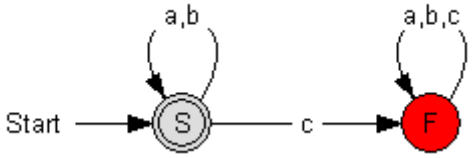


Schon hier kann man deutlich erkennen, dass die Regel S→cA für die Sprache irrelevant ist. Das hätte man auch schon am Aufbau der Produktionsregeln bemerken können.

Alle anderen möglichen Übergänge, die nicht aus den Produktionsregeln folgen, führen in einen Fehlerzustand:



Wenn man an die zu erzeugende Sprache denkt, so kann man diesen Akzeptor offensichtlich noch optimieren:



Damit könnte man rückwirkend aber auch die Produktionsregeln der zugehörigen Grammatik optimieren, d.h. minimalisieren.

Aufgaben

1. Die Grammatik G_4 mit dem Startsymbol S sei folgendermaßen definiert:
T = {a, b, c, d}
N = {S, A, B, C, D}
R = {S→aA|bS|cS|dS, A→aA|bB|cS|dS, B→aA|bS|cC|dS,
C→aA|bS|cS|dD, D→ε|aD|bD|cD|dD}

Prüfe durch Ableiten, ob die Wörter *abdabcdba*, *bcdabdab*, *cabcdbcab* von der Grammatik G_4 produziert werden können.
Entwickle den zugehörigen Akzeptor und teste die Wörter. Erkennst du, was die Grammatik leistet? Kannst du einen regulären Ausdruck dafür angeben?

2. Gegeben ist die Grammatik G = (T, N, S, R) mit
N = {S, O}, T = {0, 1, 2, …9, +, -} und
R = {S→0O|1O|…|9O, O→ε|+S|-S}.
Konstruiere einen äquivalenten Akzeptor. Beschreibe die Sprache dieser Grammatik!

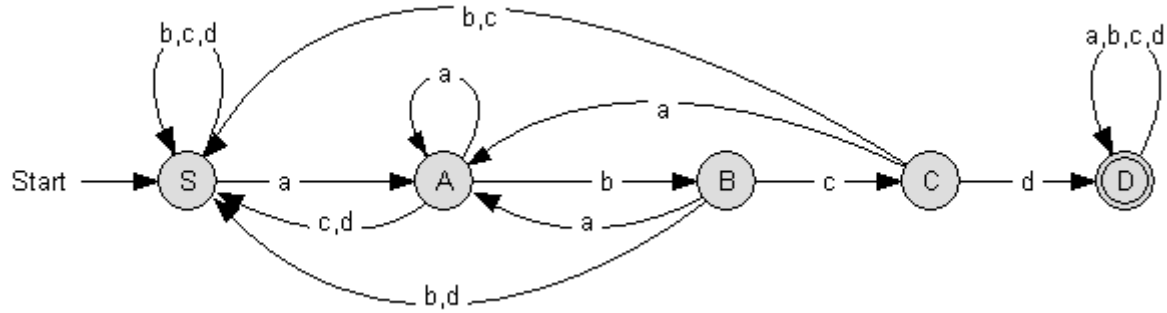
3. Gib zu jeder der folgenden regulären Sprachen über dem Alphabet {0, 1} das Zustandsdiagramm eines deterministischen, endlichen Akzeptors an!
a) {w| w beginnt mit 1 und endet mit 0}
b) {w| w ist mindestens 3 Zeichen lang und das 3. Zeichen ist 0}
c) {w| w beginnt mit 0 und hat eine ungerade Länge oder beginnt mit 1 und hat eine gerade Länge}
d) {ε, 0}
e) die leere Menge
f) alle Wörter außer dem leeren Wort

Lösungen

1. Eine Ableitung für abdabcdba:
S→aA→abB→abdS→abdaA→abdabB→abdabcC→abdabcdD→abdabcdbD→
abdabcdbaD→abdabcdba

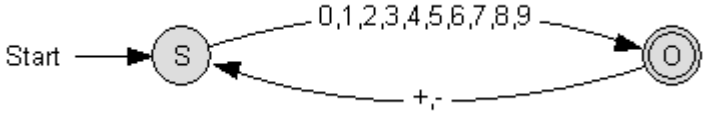
Eine Ableitung für bcdabdab gibt es nicht:
S→bS→bcS→bcdS→bcdaA→bcdabB→bcdabdS→bcdabdaA→bcdabdabB

Eine Ableitung für cabcdbcab:
S→cS→caA→cabB→cabcC→cabcdD→cabcdbD→cabcdbcD→cabcdbcaD→
cabcdbcabD→cabcdbcab

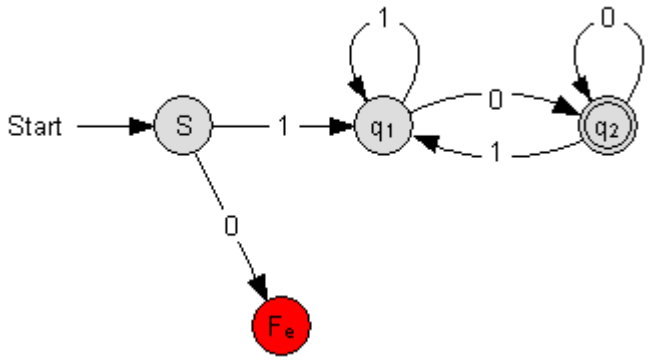


Diese Sprache akzeptiert alle Wörter, welche die Zeichenfolge „abcd“ enthalten.
Regulärer Ausdruck: $(a^*b^*c^*d^*)^*abcd(a^*b^*c^*d^*)^*$

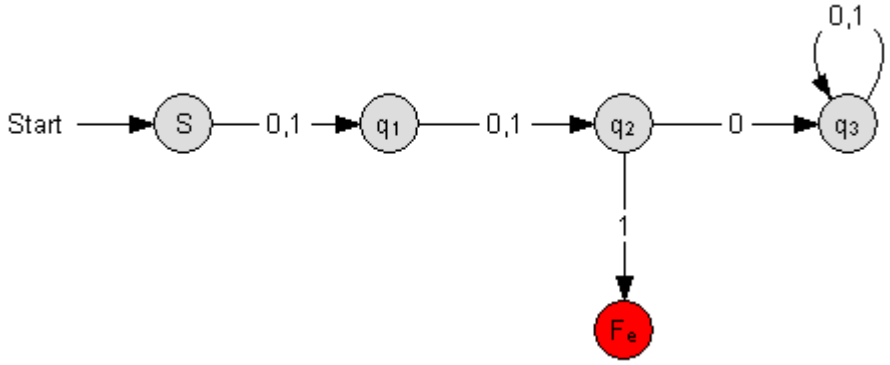
2. Der Automat akzeptiert mathematische Terme mit einstelligen Zahlen (keine Klammern, nur Addition und Subtraktion).



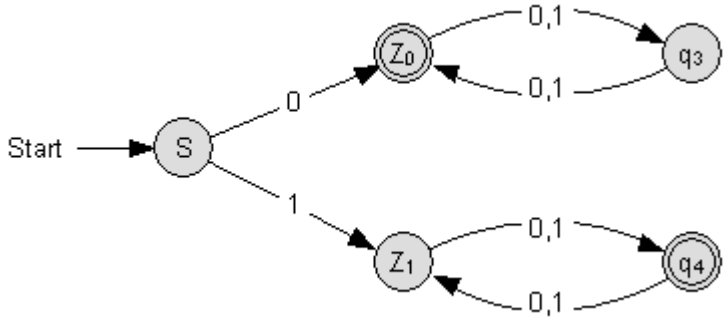
3.a)



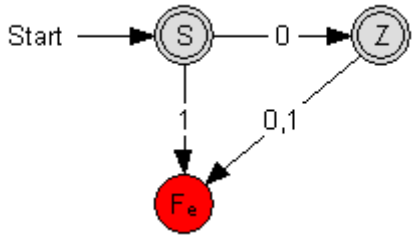
3b)



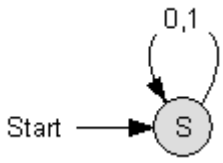
3c)



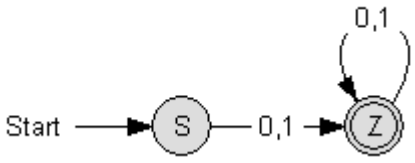
3d)



3e)

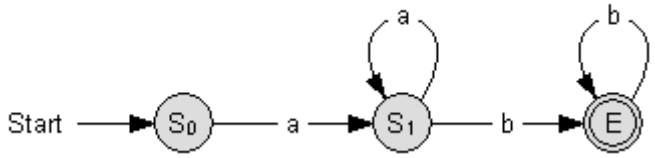


3f)

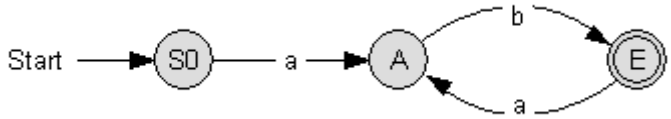


Ausblick

Die Sprache $L = \{ a^m b^n \mid a, b \in \sum \text{ und } m, n \in N \}$ ist leicht mit einem endlichen Automaten darstellbar (nicht eingezeichnete Übergänge führen in einen Fehlerzustand):



Auch die Sprache $L_5 = \{ (ab)^n \mid a, b \in \sum \text{ und } n \in N \}$ ist leicht mit einem endlichen Automaten darstellbar (nicht eingezeichnete Übergänge führen in einen Fehlerzustand):



Die Sprache $L_2 = \{ a^n b^n \mid a, b \in \sum \text{ und } n \in N \}$ ist hingegen nicht regulär. Dazu müsste sich der zugehörige Automat nämlich merken können, wie viele Zeichen a vorhanden sind (und dafür benötigt er unendlich viele Zustände), um dies mit der Anzahl der Zeichen b vergleichen zu können.

Etwas bekannter ist dieser Sachverhalt als folgender Satz:
Endliche Automaten können nicht richtig (d.h. beliebig weit) zählen.

Die üblichen Programmiersprachen akzeptieren Terme mit beliebig vielen Klammern, wobei die Anzahl der öffnenden und schließenden Klammern natürlich gleich groß sein muss. Hieraus folgt, dass diese Programmiersprachen nicht regulär sein können.

In der theoretischen Informatik gibt es weitere Automatentypen, mit denen man komplexere Sprachen beschreiben kann. Der sog. ***deterministische Keller-automat*** besitzt zusätzlich einen einzigen, eigenen Stack, mit dessen Hilfe er etwa die gerade erwähnte Sprache L_2 erkennt. Allerdings kann dieser Keller-automat nicht die Sprache $L_3 = \{ a^n b^n c^n \mid a, b, c \in \sum \text{ und } n \in N \}$ erkennen. Dafür benötigte man kompliziertere Automatentypen.