Automatentheorie und Formale Sprachen

– LV 4110 –

Reguläre Sprachen und Mengen

- Kennenlernen der Begriffe: Reguläre Sprachen und reguläre Ausdrücke
- Definition der Operationen, die – angewandt auf reguläre Sprachen – wieder reguläre Sprachen erzeugen
- Definition einer Operations-Hierarchie
- Elementarautomaten für die Verkettung, die Potenz und für die Iteration
- Kennenlernen der Vorgehensweise bei der Zusammenführung von Elementarautomaten
- Entwicklung und Anwendung von Suchalgorithmen und Texterkennungsprozeduren: Skelettautomaten, goto- und failure-Funktion

II. Reguläre Sprachen und Mengen

II. Reguläre Sprachen und Mengen

Die Menge aller Sprachen, die von einem endlichen Automaten akzeptiert werden, nennt man auch die Familie der regulären Sprachen.

Fragestellungen:

1. Welche Operationen auf reguläre Sprachen erzeugen wieder reguläre Sprachen?
2. Wie findet man zu einer regulären Sprache den „einfachsten“ deterministischen Automaten?
3. Welche Probleme sind für reguläre Sprachen algorithmisch lösbar?

II. Reguläre Sprachen und Mengen

Definition:

Ein *regulärer Ausdruck* besteht aus Zeichen eines Alphabets und/ oder anderen regulären Ausdrücken, die durch die Operationen *Iteration* (*), *Verkettung* (...) oder *Wahlmöglichkeit* (|) miteinander verbunden sind. Jedem regulären Ausdruck α entspricht eine Wortmenge $L(\alpha)$ aus Σ^*, die als *reguläre Menge oder reguläre Sprache* bezeichnet wird.

Interpretation:

Ein regulärer Ausdruck kann also verstanden werden als *Formel*, die beschreibt, wie die Wörter einer Sprache, d. h. einer gewissen Untermenge von Σ^* aus den Zeichen des Alphabets Σ, anderen Formeln und den genannten Operationen zu bilden sind.

II. Reguläre Sprachen und Mengen

Operations-Hierarchie:

Ähnlich wie bei algebraischen Formeln gibt es eine Operationen-Hierarchie:

1. Iteration (*)
2. Verkettung (...)
3. Wahlmöglichkeiten (|)

Den Operationen Iteration, Verkettung und Auswahl zur Verknüpfung von regulären Ausdrücken entsprechen im Bereich der zugehörigen regulären Mengen aus Σ^* die *Mengenoperationen:*

Iteration, *Mengenprodukt* und *Vereinigung*.

Regulärer Ausdruck α vs. Endlicher Automat **A**:

$\Sigma = \{a, b\}$

Regulärer Ausdruck: $\alpha = a^*ba$

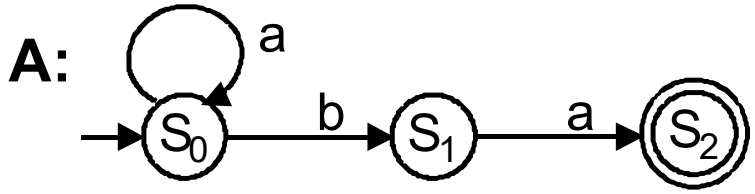
Anmerkung: Der Stern bedeutet in diesem Zusammenhang die Hintereinanderreihung von a.

Wortmenge L(α):

$L(\alpha) = \{ba, aba, aaba, aaaba, aaaaba, ...\}$

$= T(\mathbf{A})$

Automatenmodell **A**:



DFA **A** := ($\Sigma = \{a, b\}$, $S = \{S_0, S_1, S_2\}$, S_0, δ, $F = \{S_2\}$)

Definitionen:

Seien L_1, L_2 Mengen von Wörtern über dem Alphabet Σ.

1. Verkettung oder Mengenprodukt

Man definiert als Mengenprodukt von L_1 und L_2 die Menge $L_1 L_2$ durch:

$$L_1 L_2 = \{ w_1 w_2 \in \Sigma^* \mid w_1 \in L_1 \text{ und } w_2 \in L_2 \}$$

Definitionen:

Sei L eine Menge von Wörtern über dem Alphabet Σ.
Ferner sei ε das leere Eingabewort.

2. Potenz

Man definiert die Potenz $L^{(i)}$ von L für $i \geq 0$ durch:

$$L^{(0)} = \{ \varepsilon \} ;$$

$$L^{(1)} = L ;$$

$$L^{(i+1)} = L^{(i)} L .$$

Definitionen:

Sei L eine Menge von Wörtern über dem Alphabet Σ.
Ferner sei ε das leere Eingabewort.

3. Iteration $\cup$ = Vereinigungsmenge

Man definiert die Iteration L* von L als:

$$L^* = \{ \varepsilon \} \cup \{ w_1 w_2 \ldots w_n \} \mid w_i \in L$$

für $i = 1, 2, \ldots, n$; $n = 1, 2, \ldots, \infty$

$$= \{ \varepsilon \} \cup L \cup LL \cup LLL \cup \ldots$$

$$= \{ \varepsilon \} \cup L^{(1)} \cup L^{(2)} \cup L^{(3)} \cup \ldots = \bigcup_{i=0}^{\infty} L^{(i)}$$

Definition:

Nun können wir die Begriffe „regulärer Ausdruck“ und „reguläre Sprache“ **induktiv** wie folgt definieren. Es sei dabei Σ ein endliches Alphabet; dann gilt vereinbarungsgemäß:

(1) ε ist ein regulärer Ausdruck über Σ mit der Sprache $L(\varepsilon) = \{\varepsilon\}$.

(2) Jedes $a \in \Sigma$ ist ein regulärer Ausdruck über Σ mit der Sprache $L(a) = \{a\}$.

(3) Sind α und ß reguläre Ausdrücke über Σ, so sind auch α^*, $\alpha ß$ und $\alpha|ß$ reguläre Ausdrücke und die zugehörigen Sprachen sind:

$$L(\alpha^*) = (L(\alpha))^* \; ; \; L(\alpha ß) = L(\alpha)L(ß) \text{ und } L(\alpha|ß) = L(\alpha) \cup L(ß).$$

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie

2. Endliche Automaten und reguläre Sprachen
   - 2.1 Elementarautomaten
   - 2.2 Zusammenführung von Teilautomaten

3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

Satz:

Zu jedem regulären Ausdruck α gibt es einen Automaten A (mit genau einem Anfangs- und einem Endzustand), dessen Sprache **T**(A) identisch ist mit der regulären Sprache **L**(α), d. h.

$$\mathbf{T}(A) = \mathbf{L}(\alpha)$$

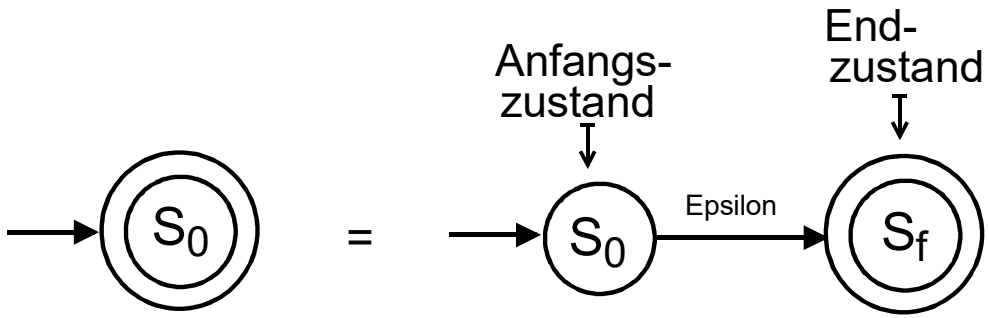
und dessen Zustandszahl von der Länge des Ausdrucks α abhängt.

Beweis:

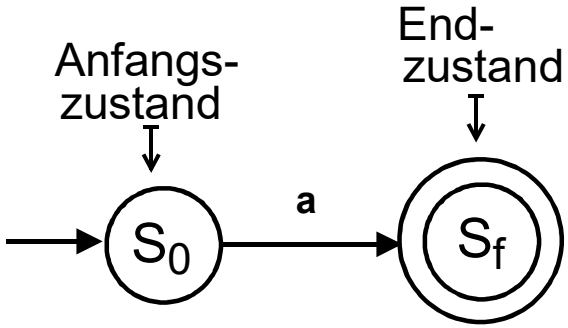
Der Beweis erfolgt durch Angabe von Elementarautomaten für (1) und (2) in der Definition und die Konstruktion entsprechend zusammengesetzter Automaten gemäß Regel (3):

II. Reguläre Sprachen und Mengen

Elementarautomat für (1) $\rightarrow$ Aε:

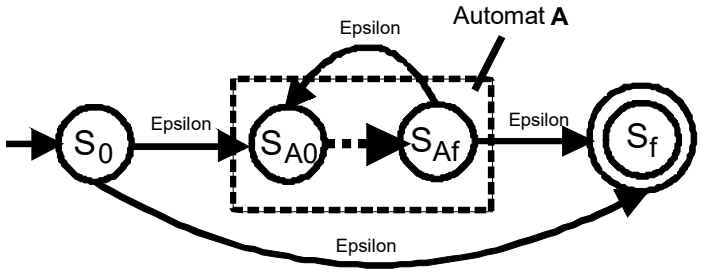


Elementarautomat für (2) $\rightarrow$ Aa:

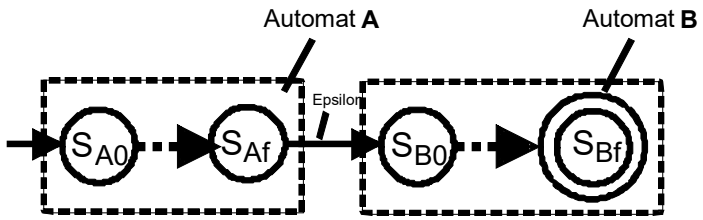


Elementarautomat für (3) mit L(α) = T(A) sowie L(β) = T(B):

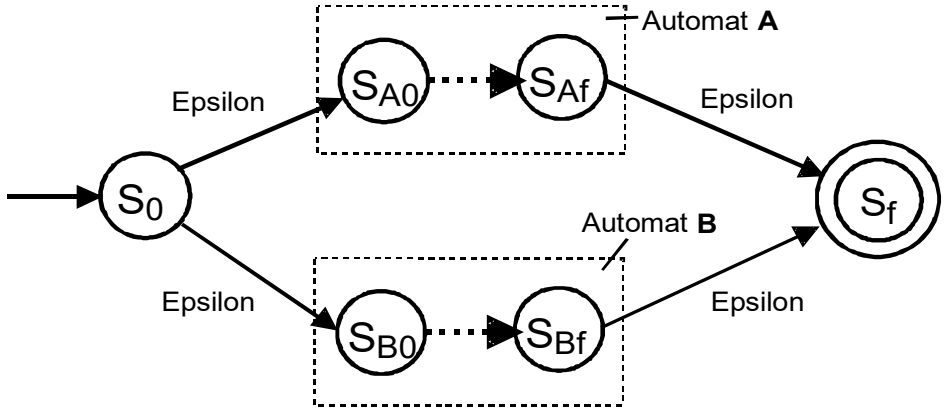
i) für α*:



ii) für αß:



iii) für α|ß:



II. Reguläre Sprachen und Mengen

Vorgehensweise:

1. Mit Hilfe der Elementarautomaten für α^*, $\alpha\ \beta$ und $\alpha \mid \beta$ lassen sich durch sukzessive Anwendung Zustandsautomaten mit **spontanen ε-Übergängen** rekonstruieren.
2. Aus diesen sog. **ε-Automaten** können wir dann **nicht-deterministische** Automaten **ohne** ε-Übergänge ableiten, die dieselben Mengen von Worten akzeptieren.
3. Schließlich lassen sich die NFA in **deterministische** Automaten (DFA) überführen (**Teilmengenverfahren**) und letztere ggf. noch optimieren (Zusammenlegen äquivalenter Zustände → **Minimal-automat**).

Aufgabe:

Gesucht ist der Zustandsautomat für folgenden regulären Ausdruck:

$$\alpha = a\ (\ a \mid bb\ )^*$$

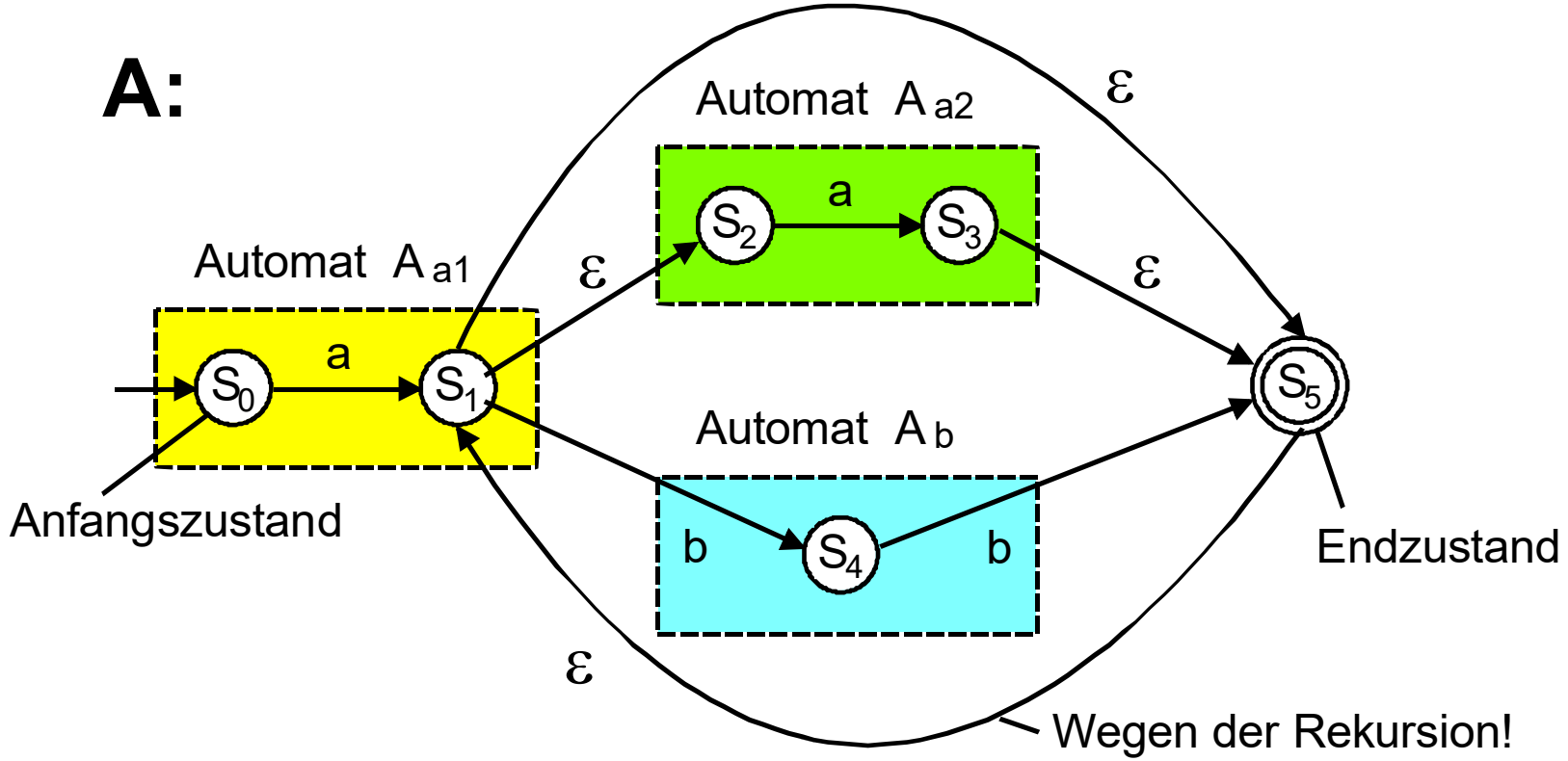
Interpretation:

Alle Wörter, die mit a beginnen, gefolgt von Teilwörtern, die nur aus Zeichen der Form bb oder a bestehen.

Lösungsidee:

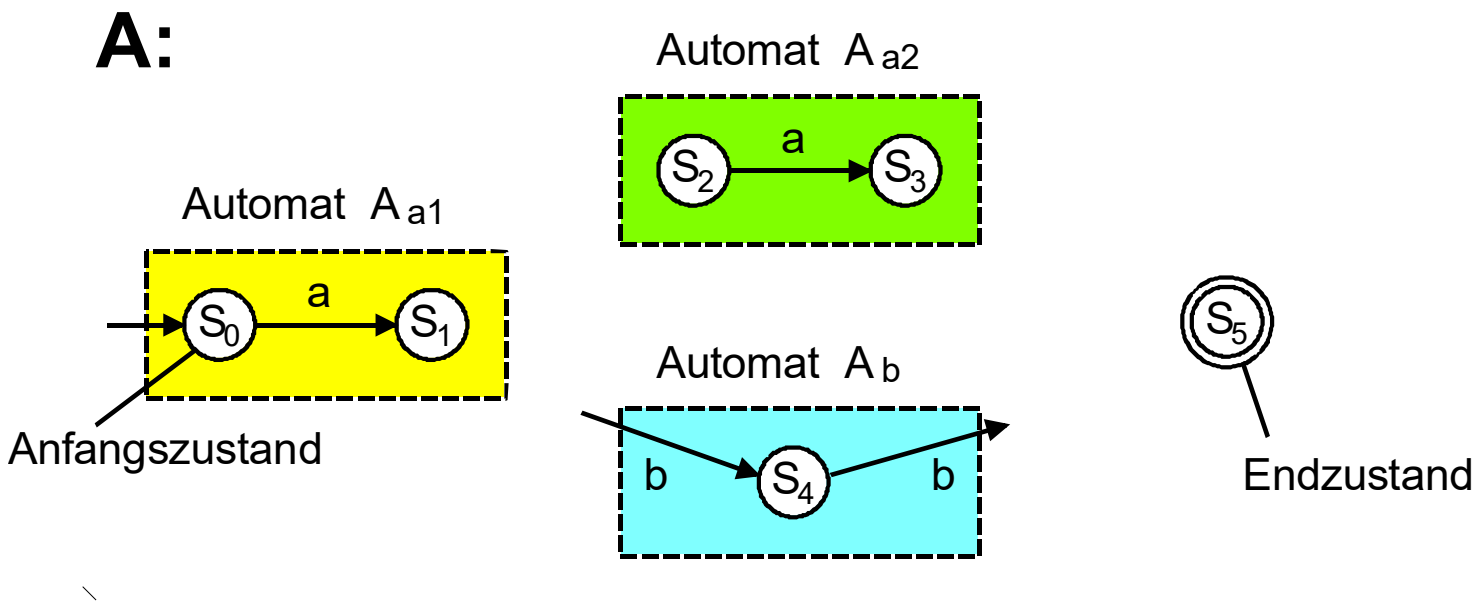
Konstruktion des gesuchten Automaten A aus **drei** Teilautomaten A_{a1}, A_{a2} und A_b in Verbindung mit spontanen ε-Übergängen.

Komposition:



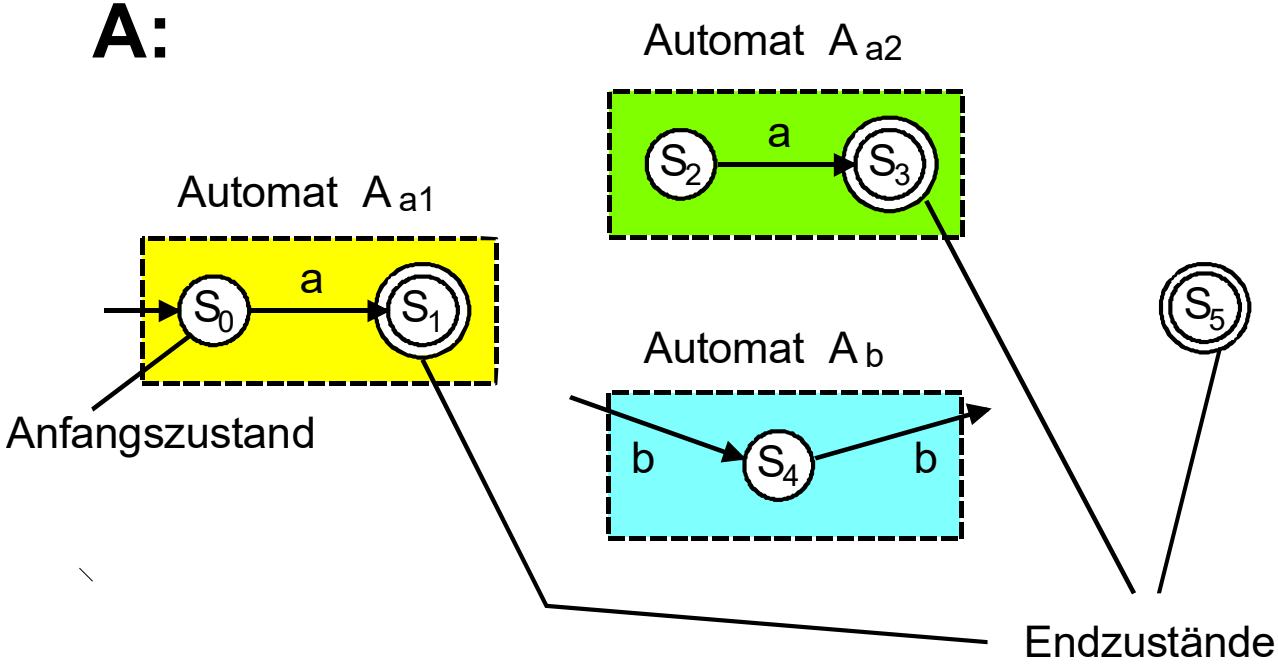
Umwandlung in Automaten ohne ε-Übergänge:

1. Schritt: Übertragen der Zustände

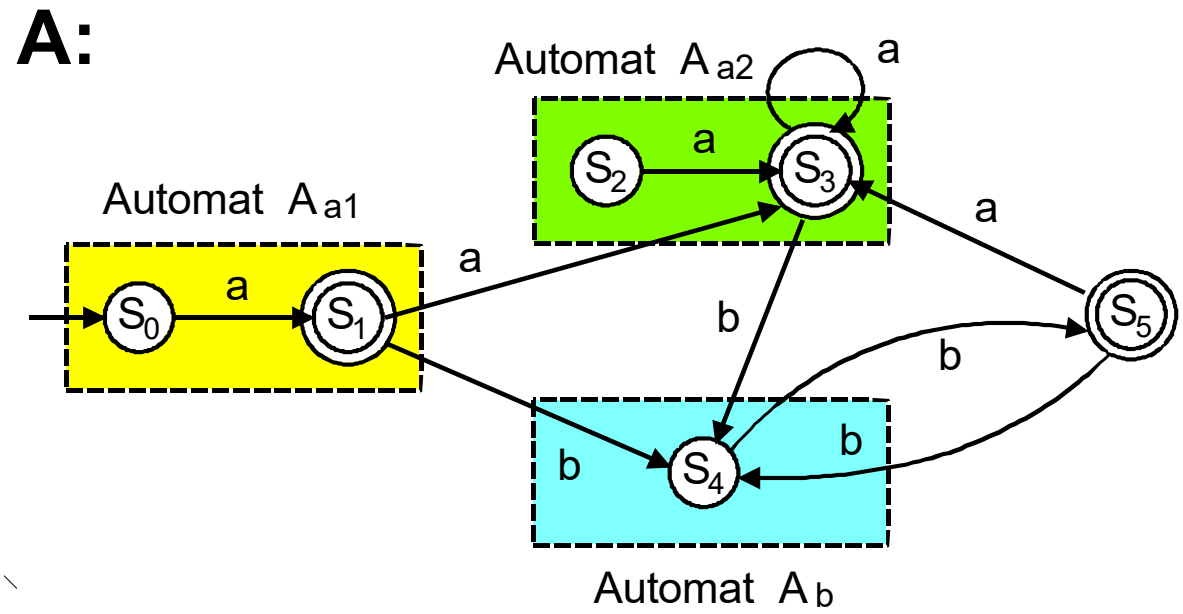


2. Schritt: S_1 und S_3 zu einem Endzustand machen, da man von S_1 bzw. S_3 durch ε-Übergänge in den Endzustand des ε-Automaten kommt.

A:

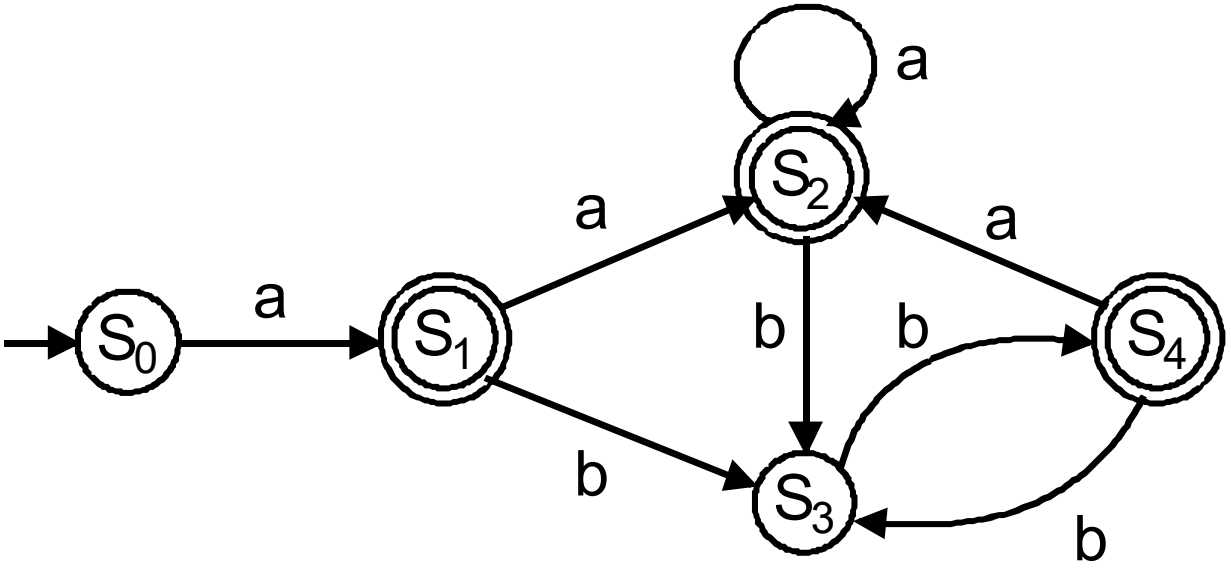


3. Schritt: Ersetzen der ε-Übergänge durch Nicht-ε-Übergänge.



4. Schritt: Entfernen des Zustands S_2, weil dieser nicht erreichbar.

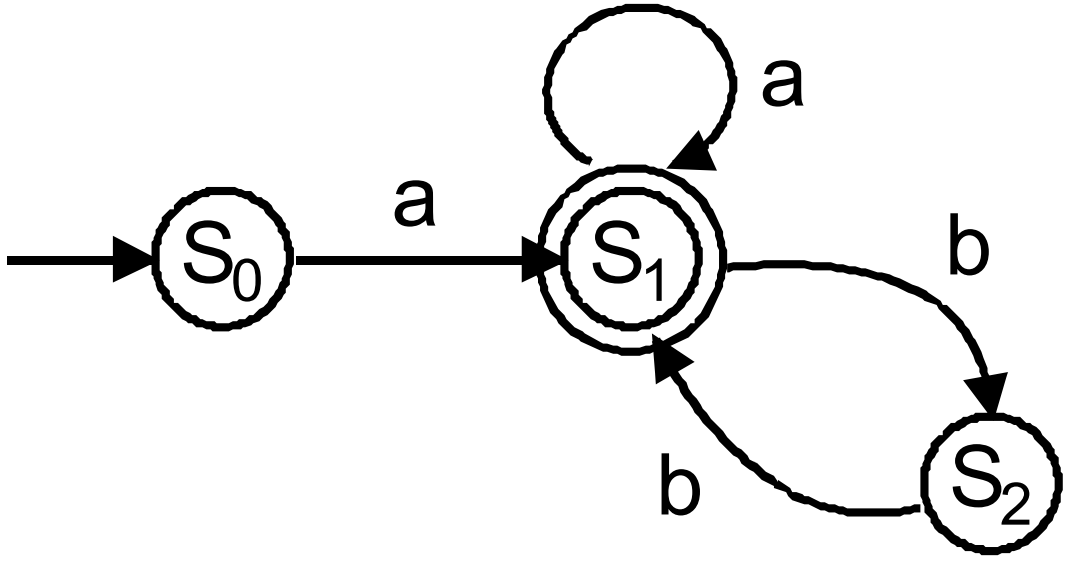
Ergebnis: **A:**



$\rightarrow \Sigma = \{a, b\}$, $\mathbf{S} = \{S_0, S_1, S_2, S_3, S_4\}$ und $\mathbf{F} = \{S_1, S_2, S_4\}$

5. Schritt: Reduzierung zum Minimalautomaten A'.

Endergebnis: **A':**



$\rightarrow \Sigma = \{a, b\}$, $\mathbf{S} = \{S_0, S_1, S_2\}$ und $\mathbf{F} = \{S_1\}$

II. Reguläre Sprachen und Mengen

Ziel:

Umsetzung eines deterministischen endlichen Automaten (DFA) in ein Programm.

Idee:

Zustände des Automaten als Sprungmarken ansehen.

- {neue Sprungmarke mit neuem Zeichen}
  = g(aktuelle Sprungmarke mit aktuellem Zeichen).
- Delimiterzeichen zur Kennzeichnung des Wortendes.
- Unterfunktion FAIL(.), die zur Anwendung kommt, wenn eingelesenes Zeichen im aktuellen Zustand nicht erlaubt ist.

```
                                procedure numbertest(.);
DIGIT := { ´0´, ... ,´9´}
SIGN := {´+´,´-´}
…
Label: 0,1,2,3,4,5,6,7;
0: a := NEXTCHAR(.);
  if a in SIGN then goto 1
  else if a in DIGIT then goto 2 else FAIL(.);
1: a := NEXTCHAR(.);
  if a in DIGIT then goto 2 else FAIL(.);
2: a := NEXTCHAR(.);
...
7: a := NEXTCHAR(.);
  if a in DIGIT then goto 7
  else if a in DELIMITER then write („Zahldarstellung o.k.“)
         else FAIL(.)
```

Es seien:

a = Eingabezeichen

Zustand_neu = g(Zustand_aktuell, a)

```
                                    procedure numbertest(.);
...                                      << Wenn Eingabezeichen im aktuellen >>
Zustand:=0;                              << Zustand nicht erlaubt ist, liefert g    >>
repeat                                   << den Wert FAIL                           >>
   a := NEXTCHAR(.)
   if a not in DELIMITER then Zustand := g(Zustand,a)
until (Zustand = FAIL) or (a in DELIMITER)
if a in DELIMITER then write („Zahldarstellung o.k.“)
else ...
```

II. Reguläre Sprachen und Mengen

Pattern-matching Problem

Es seien gegeben: ein Text **X** = $x_1x_2...x_n$

eine Zeichenkette **Y** = $y_1y_2...y_m$

mit n >> m .

Ziel: Jedes Vorkommen von Y in X soll festgestellt werden.

Erster Lösungsansatz:

i = Laufindex über den Text **X**: ***i*** = 0 ... n-m

j = Laufindex über die Zeichenkette **Y**: ***j*** = 1 ... m

found = logische Variable := ***true***, wenn **Y** in **X** gefunden

false, sonst (nicht gefunden)

```
i:= 0;
repeat                                   << über gesamten Text X >>
  found := TRUE;
  j:= 1;
  while (j <= m) and found do            << über ges. Zeichenkette Y >>
     if x(i+j) <> y(j) then found = FALSE;
     j:= j +1;
  endwhile
  if found then write (‚Y kommt vor in X an der Position‘, i+1)
  i:= i+1;
until i > n-m
```

Anzahl der Such-Schritte: Suchschritte $\approx (n - m)\, q$ mit $1 < q < m$.

q = durchschnittliche Anzahl der Durchläufe der while-Schleife (abhängig von m und der Art der Zeichenkette)

Anmerkung: Überraschenderweise kann man jedoch einen Algorithmus angeben, der die Frage, ob **Y** Teilwort von **X** ist, in

$\approx n + m \approx n$; da $m << n$

Schritten beantwortet.

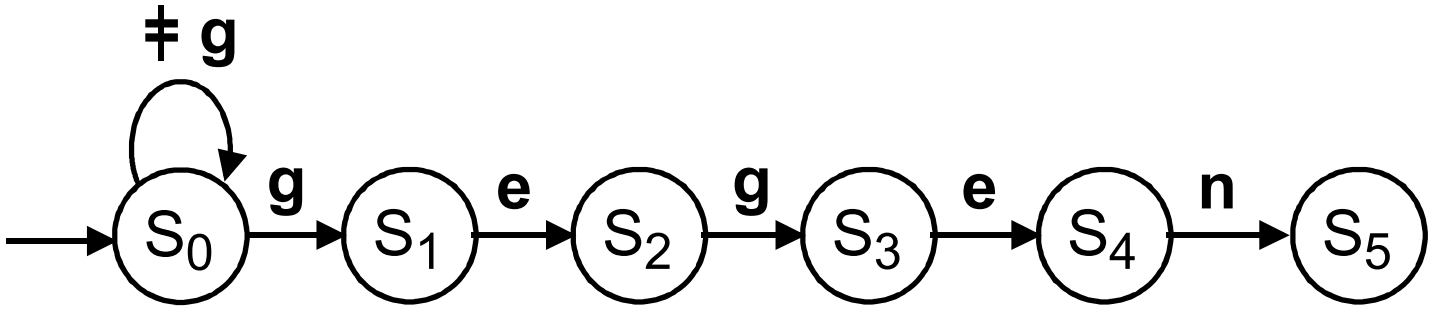
Idee:

Wir konstruieren zur Zeichenkette **Y** einen deterministischen endlichen Automaten, der auf die Eingabe des Textes **X** genau dann in einen Endzustand übergeht, wenn **Y** in **X** entdeckt wurde.

Ein anderer Lösungsweg besteht darin, den deterministischen „Skelettautomaten“ zu verwenden, dessen goto-Funktion **g** eindeutig ist.

Übergangsgraph des Skelettautomaten:

Es sei: $X \in \Sigma^*$ = { gesamte Alphabet }
Y = **y1y2…y5** = “gegen“ (Schlüsselwort)



⇒ deterministischer Automat ohne Endzustand

Eigenschaften:

- Der Skelettautomat ist im Zustand S_j ($1 \leq j \leq 5$) genau dann, wenn die letzen j gelesenen Buchstaben des Wortes **X** mit den ersten j Buchstaben von **Y** (hier: **Y** = $y_1y_2\ldots y_5$ = "gegen") übereinstimmen.
- Beim Lesen des nächsten Zeichens von **X** sind also zwei Fälle möglich:
  1. Das nächste Zeichen von **X** entspricht dem nächsten Zeichen von Y $\Rightarrow$ Automat geht in den Zustand S_{j+1} über
  2. Oder das nächste Zeichen **a** ist verschieden $\Rightarrow$ Automat geht in den Zustand S_k mit $k \leq j$ zurück, wobei k = größte Zahl derart, dass $y_1y_2\ldots y_k$ Endstück von $y_1y_2\ldots y_j$**a** ist

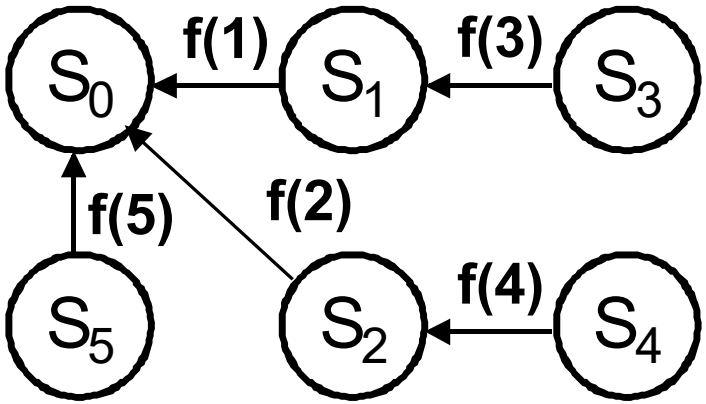
Goto-Funktion g:

Mit der aus dem Zustandsgraphen ablesbaren goto-Funktion **g** erhält man als Texterkennungsprozedur:

```
Zustand := 0;                    << Anfangszustand >>
i := 0;
while i < n do                   << n = Länge von X >>
  i := i+1;
  Zustand := g(Zustand, xi);     << xi = Eingabezeichen des Textes X >>
  if Zustand = ERKENNUNGSZUSTAND then write('Y kommt vor in
                                                X an der Position', i)
endwhile
```

Failure-Funktion f:

Um das „Zurückgehen“ im 2. Fall zu bewerkstelligen, definiert man die sog. *failure*-Funktion f, die für unser Beispiel wie folgt dargestellt werden kann:



d. h. die ***failure*-Funktion f** bildet Zustände auf Zustände ab. Der failure-Funktion muss man immer dann folgen, wenn die ***goto*-Funktion g** den Wert FAIL liefert.

Anmerkung zur Failure-Funktion f:

Die ***failure*-Funktion f: S → S** gibt an, in welchen Zustand man zurückzugehen hat, wenn nach dem Einlesen eines Zeichens x_j die ***goto*-Funktion g** des Skelettautomaten nicht definiert ist. Dabei wird der Failure-Zustand so gewählt, dass ein möglichst großes Ende des bis dahin eingelesenen **X**-Textes (außer x_j!) wieder mit dem Anfang von **Y** übereinstimmt. Paßt auch da das eingelesene Zeichen x_j nicht, geht man gemäß der failure-Funktion weiter zurück. Spätestens im Anfangszustand endet der Prozeß, weil dort die *goto*-Funktion **g** für alle Zeichen definiert ist.

Algorithmus:

Mit Hilfe der *goto*- und der *failure*-Funktion lautet der Algorithmus:

```
Zustand := 0; i := 0;
while i < n do                          << über alle Zeichen des Textes X >>
  i := i+1;
  while g(Zustand, xi) = FAIL do        << Wenn die goto-Funktion g fehl- >>
     Zustand := f(Zustand);             << schlägt, dann berechne neuen  >>
  endwhile;                             << Zustand mit Hilfe der failure-    >>
  Zustand := g(Zustand, xi);            << Funktion f                        >>
  if Zustand = ERKENNUNGSZUSTAND then write(...)
endwhile
```

Algorithmus:

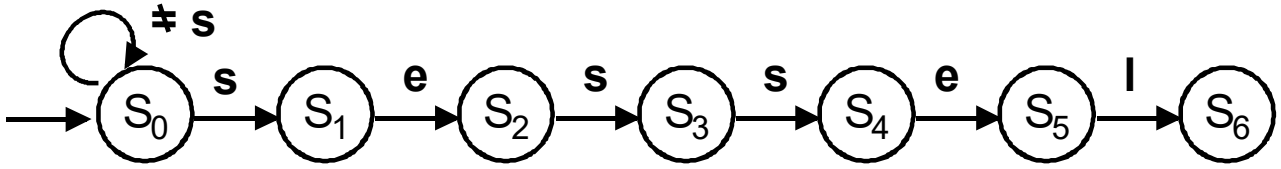
Als Algorithmus für die Funktion **f** ergibt sich also:

```
f(1) := 0;
For s = 2,3,.. ,m do
    t := s-1;
    repeat
        t := f(t);
    until g(t, ys) ≠ FAIL
    f(s) := g(t, ys);
enddo;
```

Der Rechenaufwand hierfür hat die Größenordnung m.

Beispiel:

Die Berechnung der **failure-Funktion f(t)** liefert für eine Suche nach dem Wort **„s e s s e l“** gemäß vorstehenden **Algorithmus** folgendes Ergebnis:



f: S → S

f(1) = 0 (per Def.)
f(2) = g(0, e) = 0
f(3) = g(0, s) = 1
f(4) = g(0, s) = 1
f(5) = g(1, e) = 2
f(6) = g(0, l) = 0