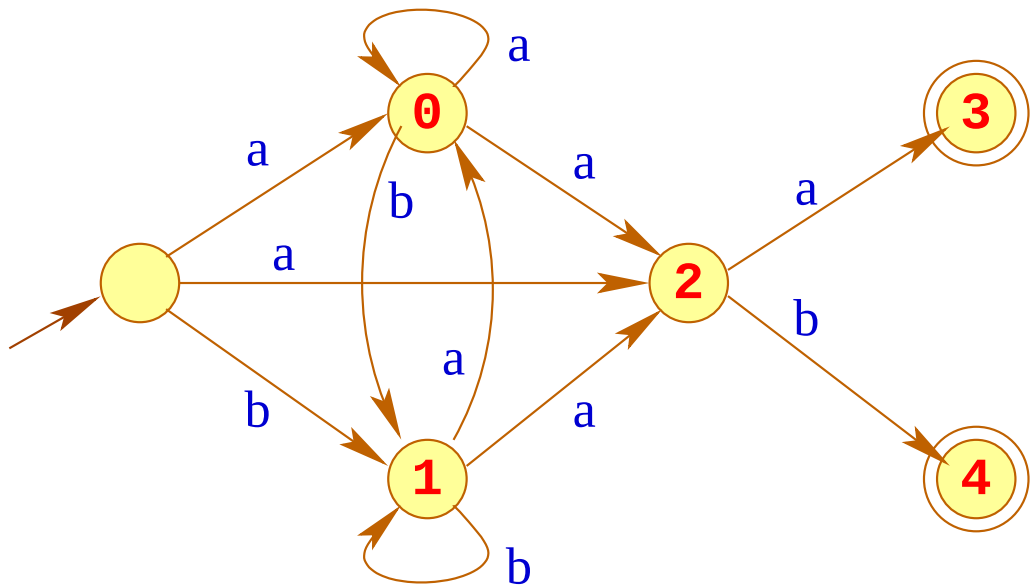
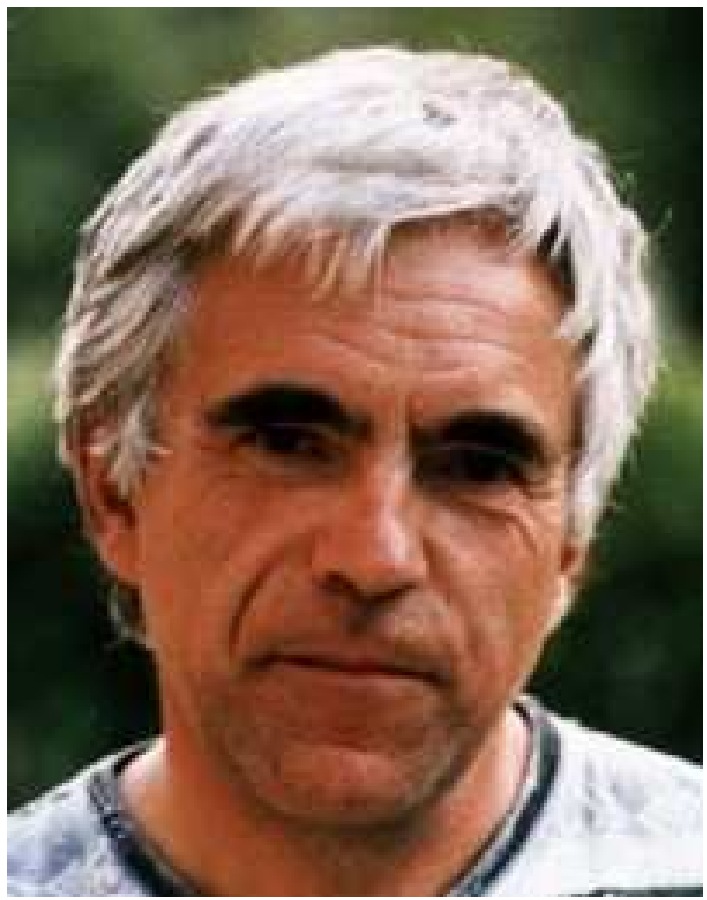
... im Beispiel:

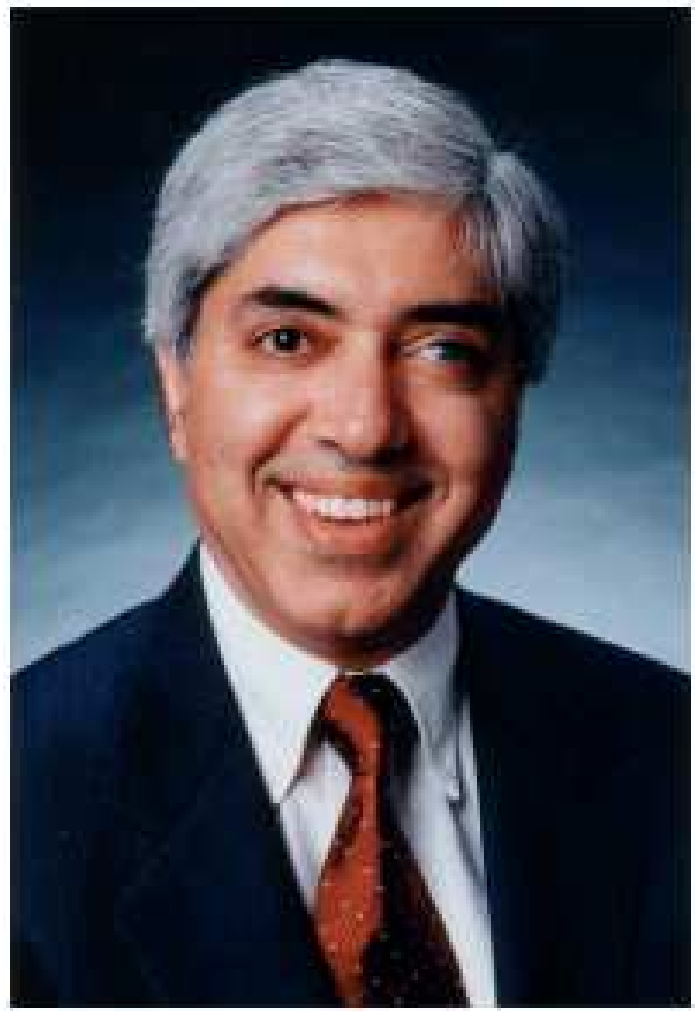


Bemerkung:

- Die Konstruktion heißt auch Berry-Sethi- oder Glushkow-Konstruktion.
- Sie wird in XML zur Definition von Content Models benutzt ;-)
- Das Ergebnis ist vielleicht nicht, was wir erwartet haben ...

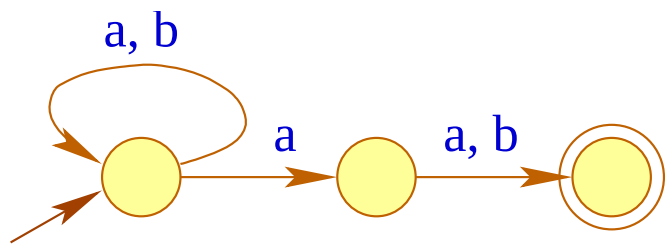


Gerard Berry, Esterel Technologies



Ravi Sethi, Research VR, Lucent Technologies

Der erwartete Automat:

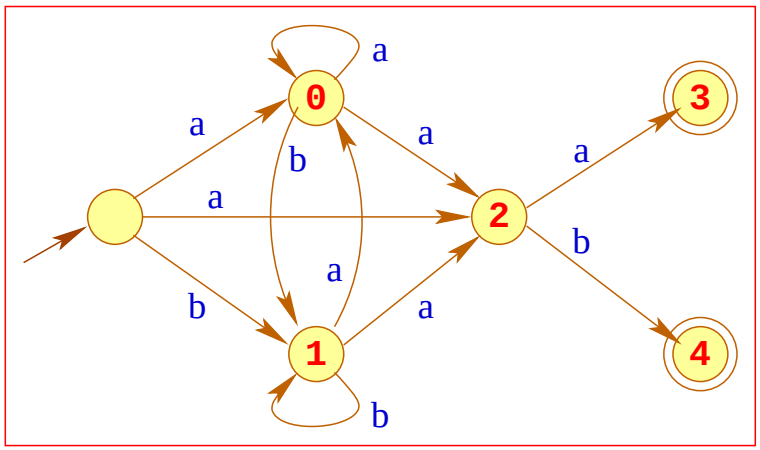


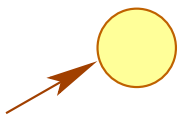
Bemerkung:

- in einen Zustand eingehende Kanten haben hier nicht unbedingt die gleiche Beschriftung :-)
- Dafür ist die Berry-Sethi-Konstruktion direkter ;-)
- In Wirklichkeit benötigen wir aber deterministische Automaten

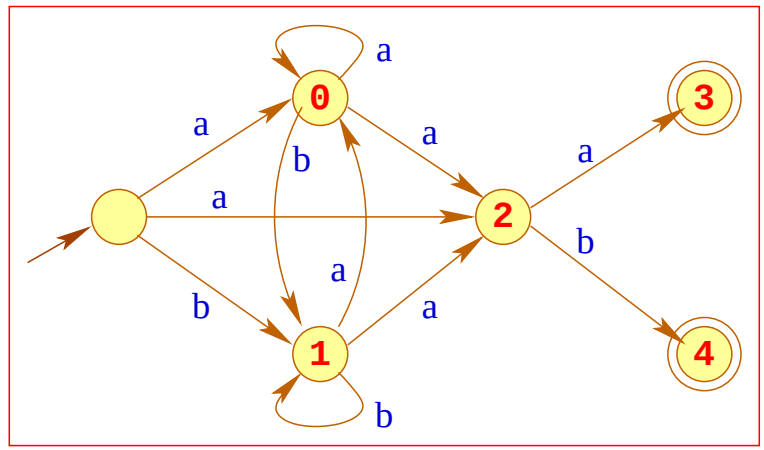
$\Longrightarrow$ Teilmengen-Konstruktion

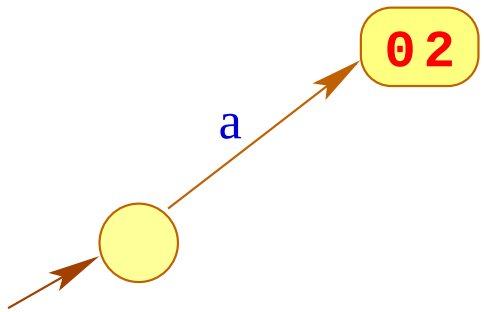
... im Beispiel:



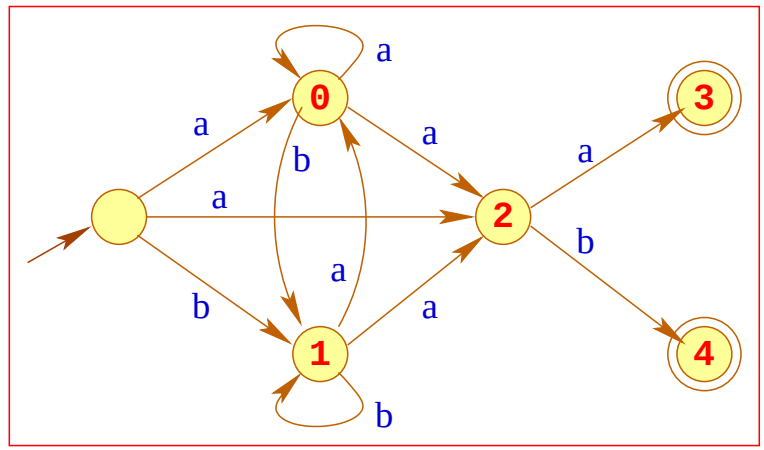


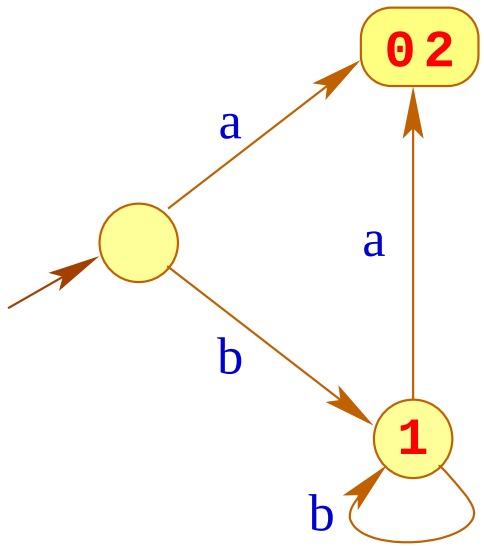
... im Beispiel:



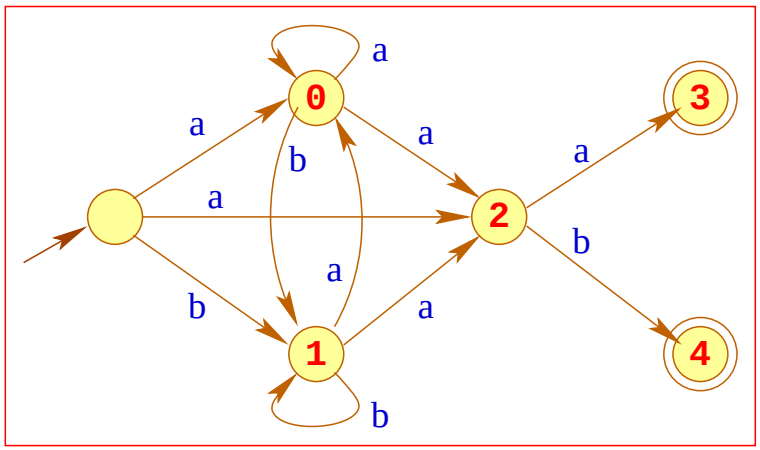


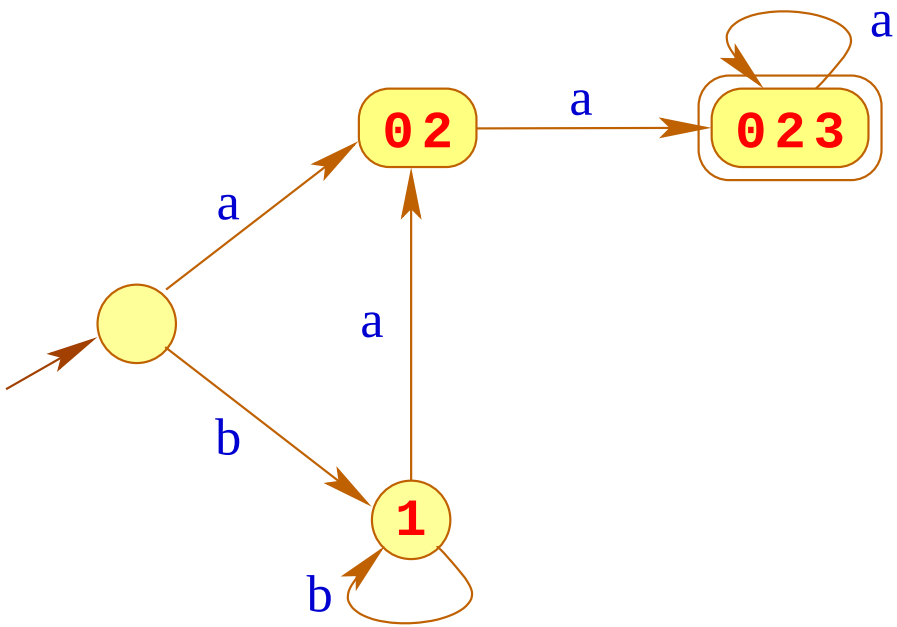
... im Beispiel:



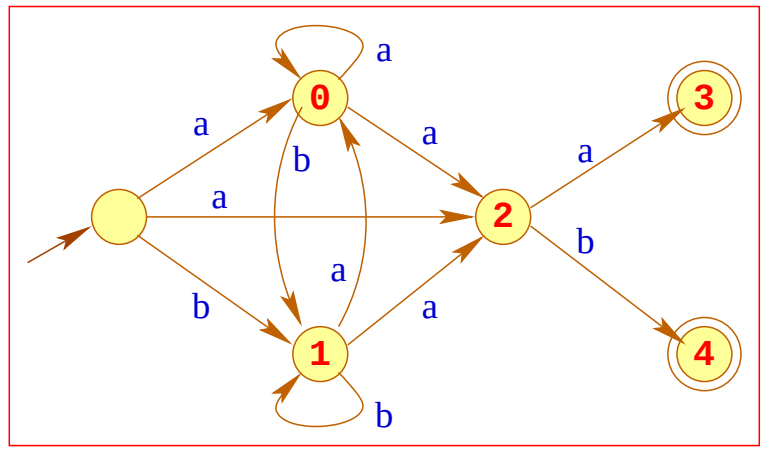


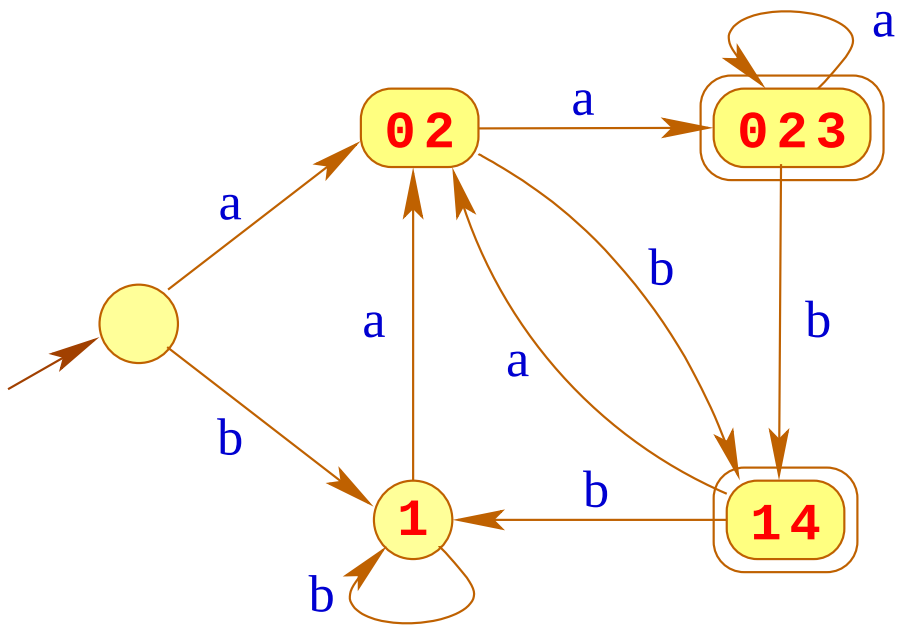
... im Beispiel:





... im Beispiel:





Satz:

Zu jedem nichtdeterministischen Automaten $A = (Q, \Sigma, \delta, I, F)$ kann ein deterministischer Automat $\mathcal{P}(A)$ konstruiert werden mit

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

Satz:

Zu jedem nichtdeterministischen Automaten $A = (Q, \Sigma, \delta, I, F)$ kann ein deterministischer Automat $\mathcal{P}(A)$ konstruiert werden mit

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

Konstruktion:

Zustände: Teilmengen von Q;

Anfangszustände: $\{I\}$;

Endzustände: $\{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$;

Übergangsfunktion: $\delta_{\mathcal{P}}(Q', a) = \{q \in Q \mid \exists\, p \in Q' : \ (p, a, q) \in \delta\}$.

Achtung:

- Leider gibt es exponentiell viele Teilmengen von Q :-(
- Um nur nützliche Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur nach Bedarf hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
- Trotz dieser Optimierung kann der Ergebnisautomat riesig sein :-((

  ... was aber in der Praxis (so gut wie) nie auftritt :-))

Achtung:

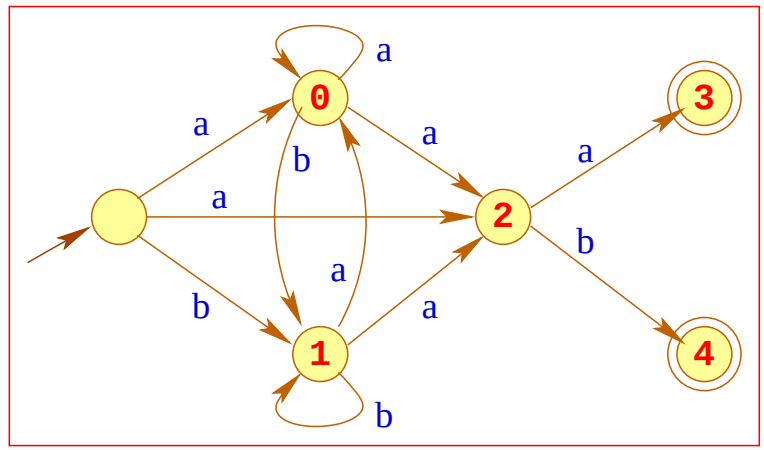
- Leider gibt es exponentiell viele Teilmengen von Q :-(
- Um nur nützliche Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur nach Bedarf hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
- Trotz dieser Optimierung kann der Ergebnisautomat riesig sein :-((

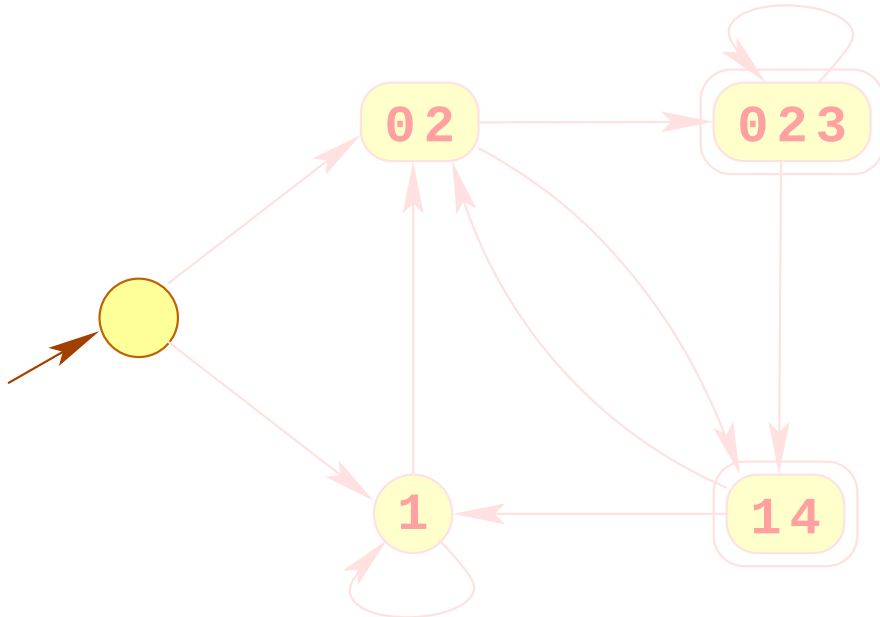
  ... was aber in der Praxis (so gut wie) nie auftritt :-))

- In Tools wie `grep` wird deshalb zu der DFA zu einem regulären Ausdruck nicht aufgebaut !!!
- Stattdessen werden während der Abbarbeitung der Eingabe genau die Mengen konstruiert, die für die Eingabe notwendig sind ...

... im Beispiel:

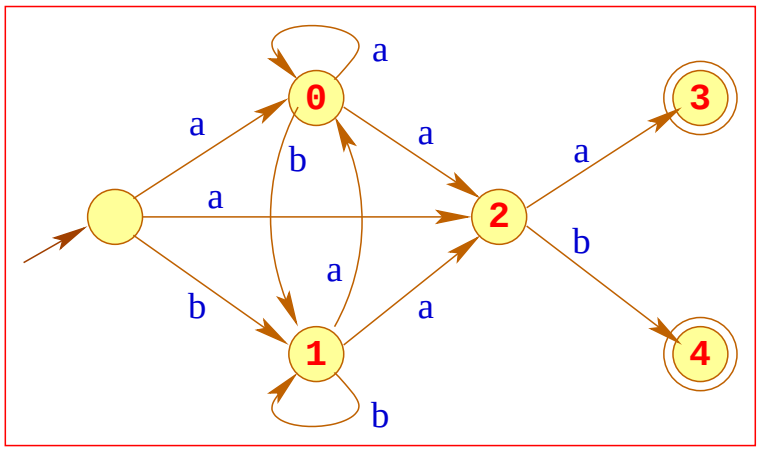
a	b	a	b

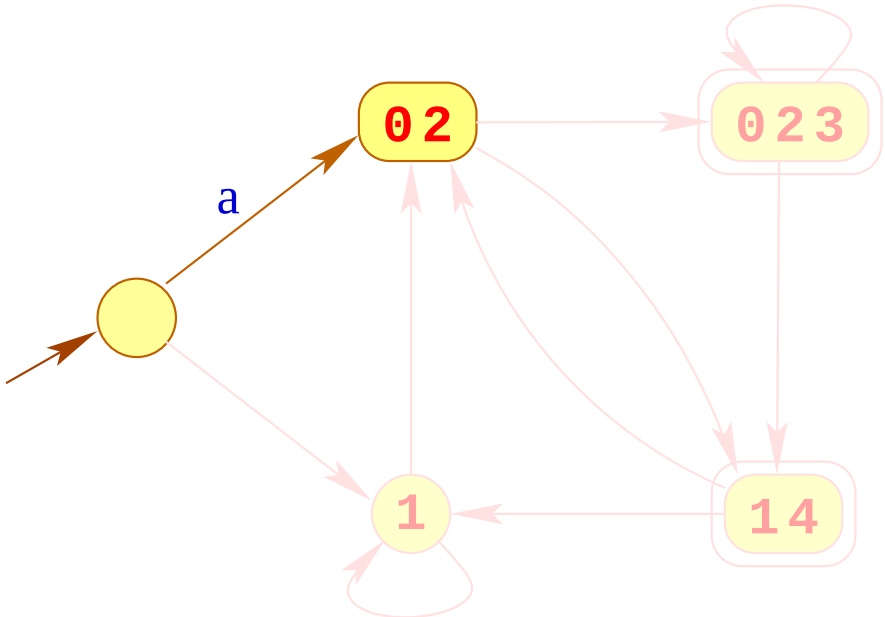




... im Beispiel:

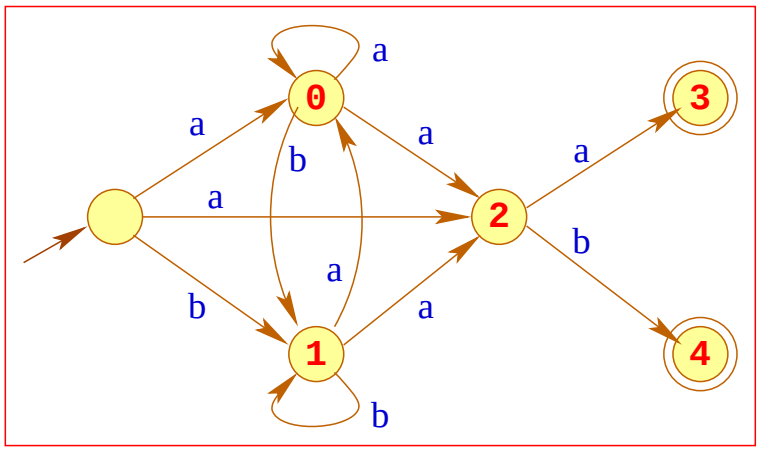
a	b	a	b

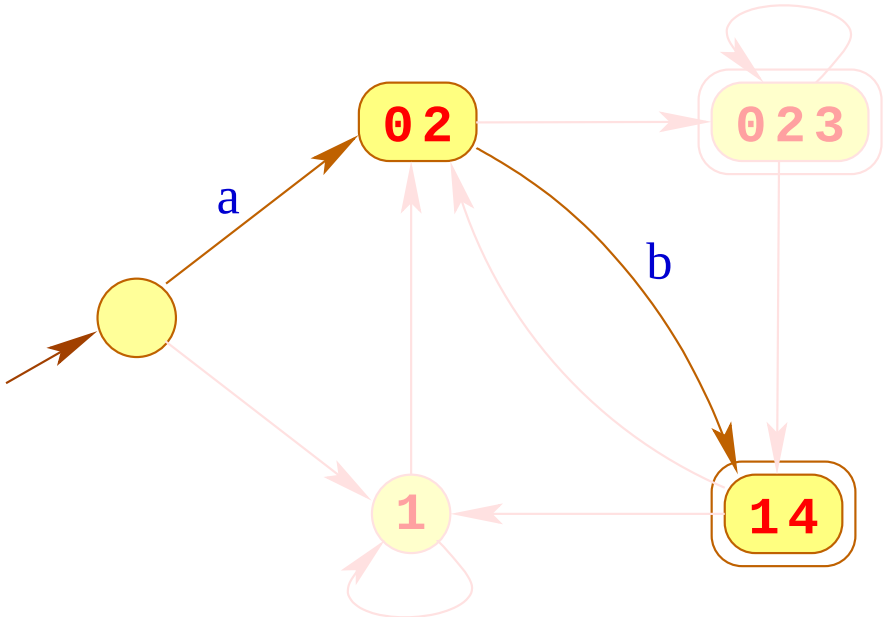




... im Beispiel:

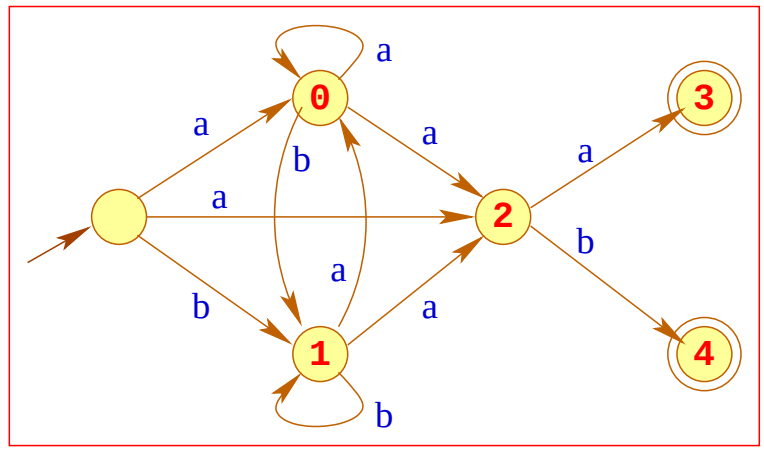
a	b	a	b

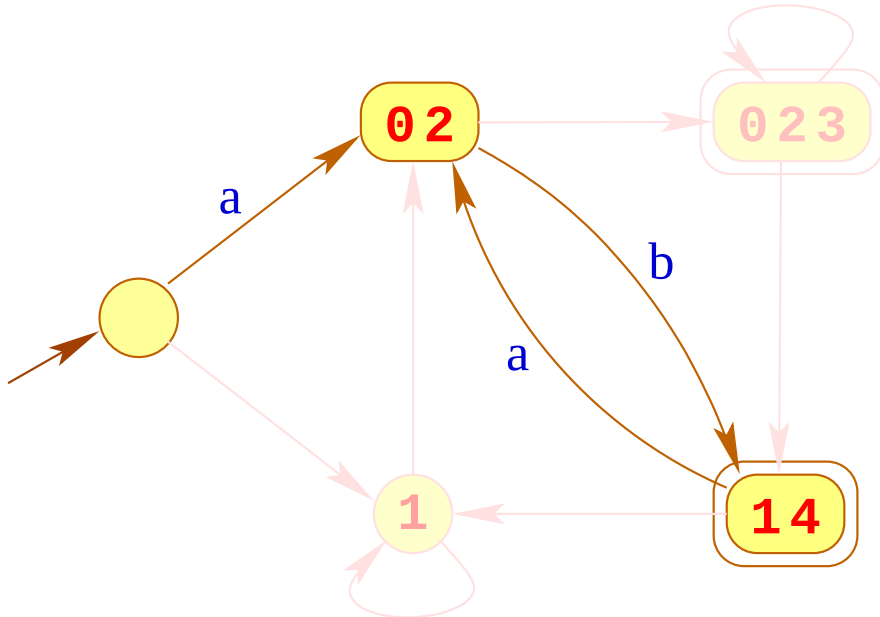




... im Beispiel:

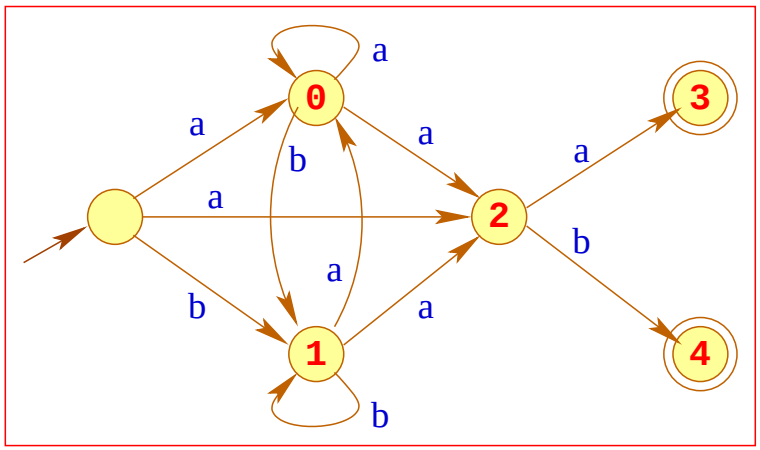
a	b	a	b

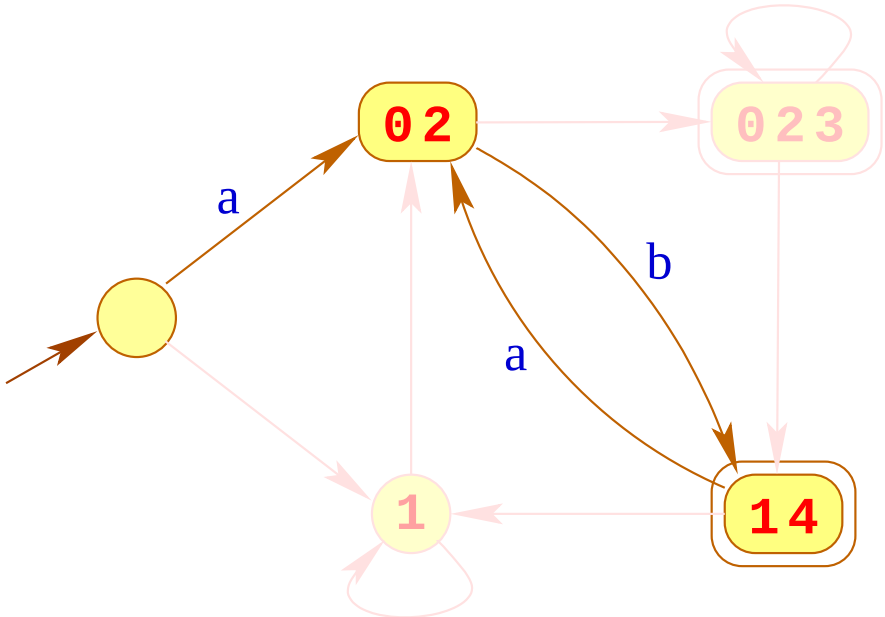




... im Beispiel:

a	b	a	b





Bemerkungen:

- Bei einem Eingabewort der Länge n werden maximal $\mathcal{O}(n)$ Mengen konstruiert :-)
- Ist eine Menge bzw. eine Kante des DFA einmal konstuiert, heben wir sie in einer Hash-Tabelle auf.
- Bevor wir einen neuen Übergang konstruieren, sehen wir erst nach, ob wir diesen nicht schon haben :-)

Bemerkungen:

- Bei einem Eingabewort der Länge n werden maximal $\mathcal{O}(n)$ Mengen konstruiert :-)
- Ist eine Menge bzw. eine Kante des DFA einmal konstuiert, heben wir sie in einer Hash-Tabelle auf.
- Bevor wir einen neuen Übergang konstruieren, sehen wir erst nach, ob wir diesen nicht schon haben :-)

Zusammen fassend finden wir:

Satz

Zu jedem regulären Ausdruck e kann ein deterministischer Automat $A = \mathcal{P}(A_e)$ konstruiert werden mit

$$\mathcal{L}(A) = [\![e]\!]$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

$$
\begin{array}{ll}
e_1 & \{\ \texttt{action}_1\ \} \\
e_2 & \{\ \texttt{action}_2\ \} \\
\ldots & \\
e_k & \{\ \texttt{action}_k\ \}
\end{array}
$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

$$
\begin{array}{ll}
e_1 & \{\ \texttt{action}_1\ \} \\
e_2 & \{\ \texttt{action}_2\ \} \\
\quad \ldots & \\
e_k & \{\ \texttt{action}_k\ \}
\end{array}
$$

Ausgabe: ein Programm, das

... von der Eingabe ein maximales Präfix w liest, das $e_1 \mid \ldots \mid e_k$ erfüllt;

... das minimale i ermittelt, so dass $w \in [\![e_i]\!]$;

... für w $\texttt{action}_i$ ausführt.

Implementierung:

Idee:

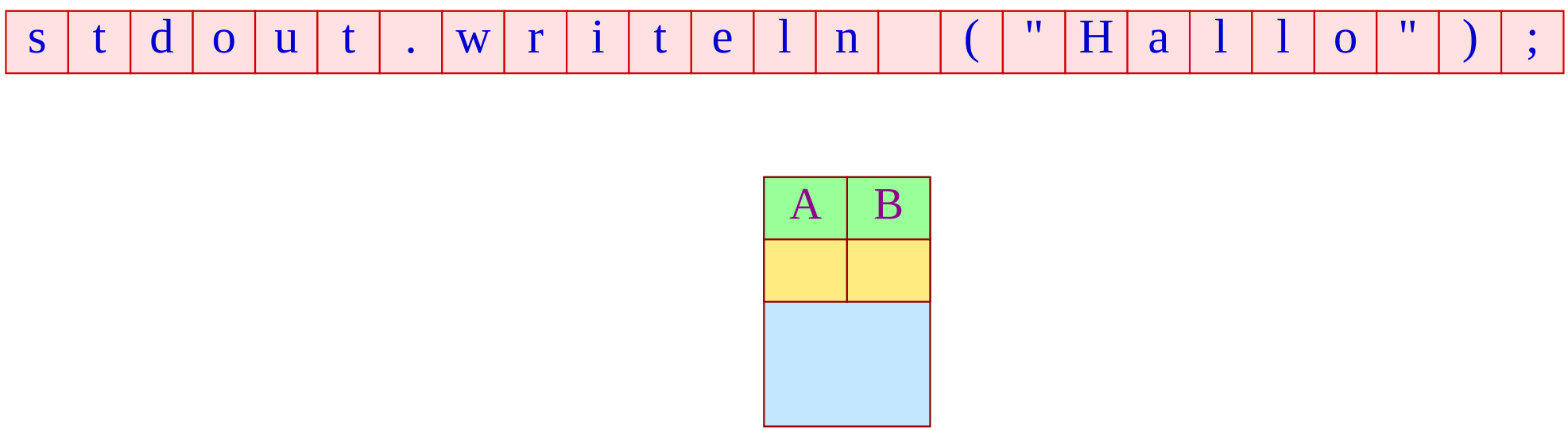
- Konstruiere den DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, \{q_0\}, F)$ zu dem Ausdruck $e = (e_1 \mid \ldots \mid e_k)$;
- Definiere die Mengen:

$$
\begin{array}{lcl}
F_1 & = & \{q \in F \mid q \cap \mathsf{last}[e_1] \neq \emptyset\} \\
F_2 & = & \{q \in (F \backslash F_1) \mid q \cap \mathsf{last}[e_2] \neq \emptyset\} \\
 & \ldots & \\
F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \mathsf{last}[e_k] \neq \emptyset\}
\end{array}
$$

- Für Eingabe w gilt: $\delta^*(q_0, w) \in F_i$ genau dann wenn der Scanner für w $\texttt{action}_i$ ausführen soll :-)

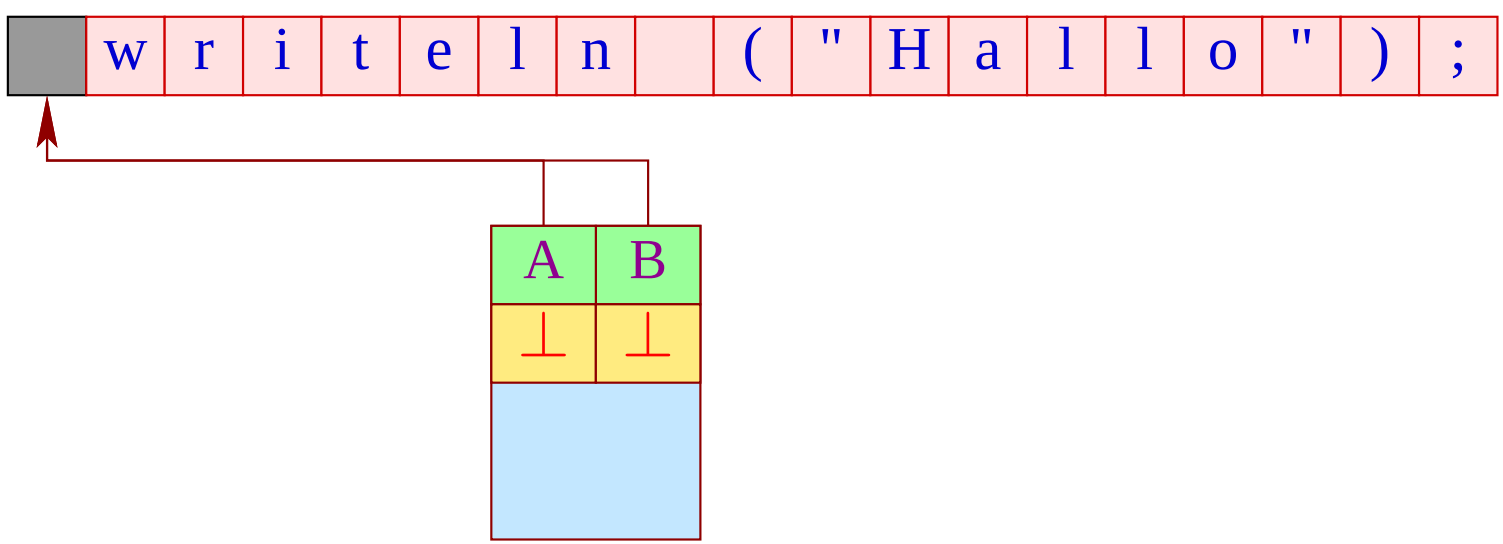
Idee (Fortsetzung):

- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.



Idee (Fortsetzung):

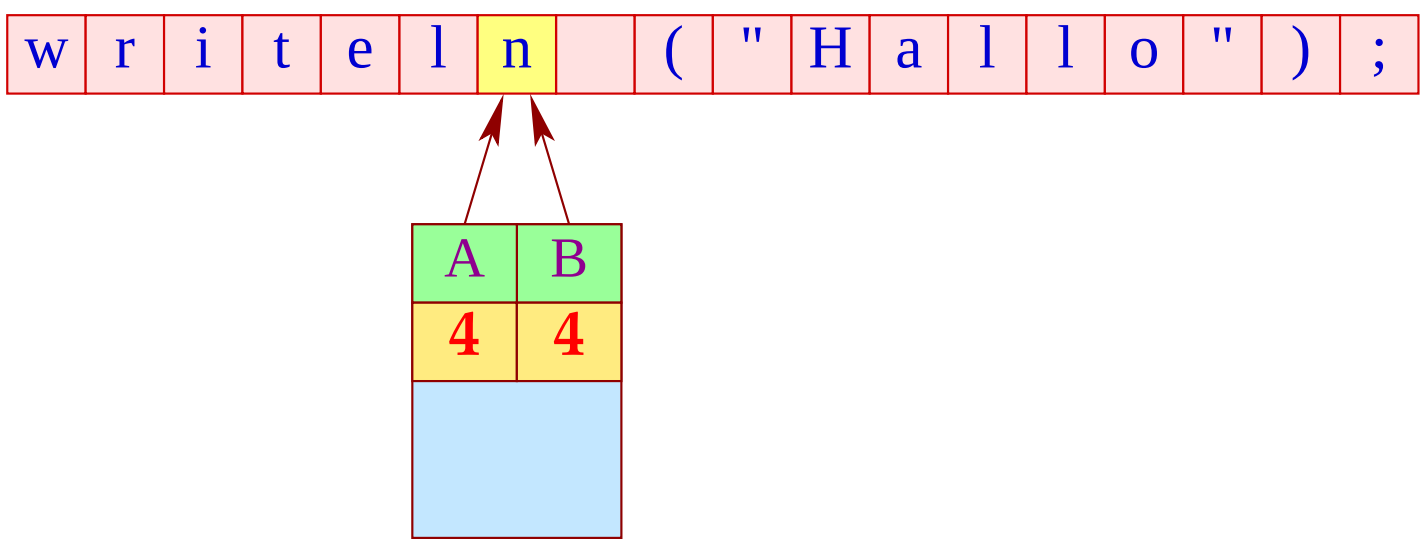
- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

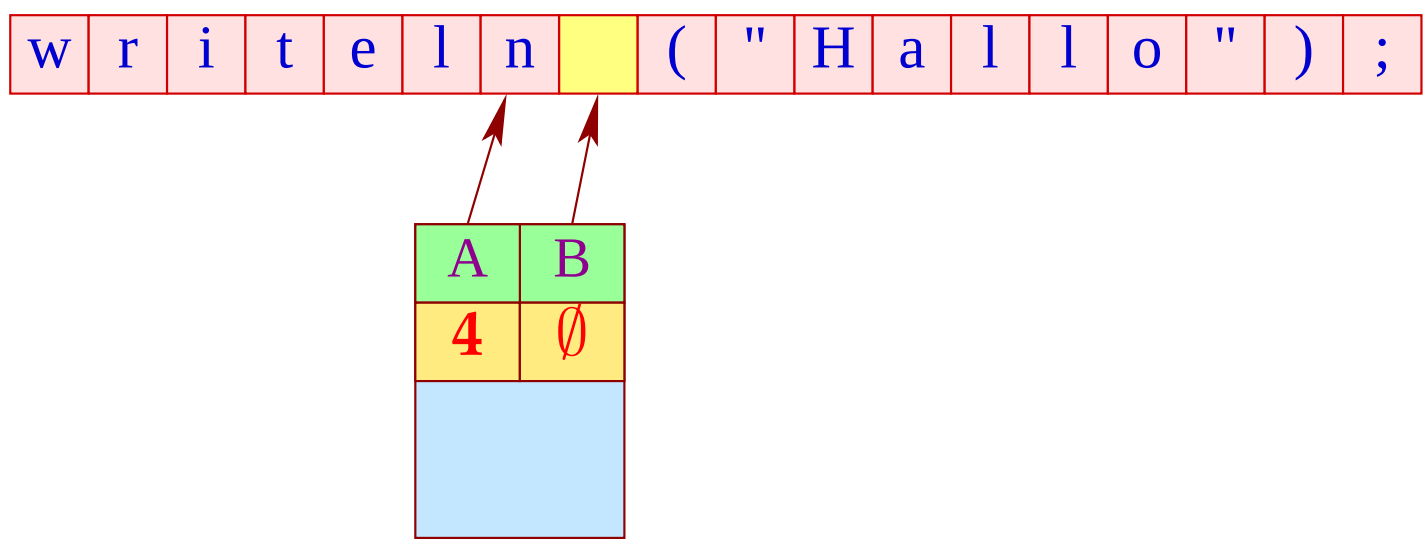
$$B := A; \quad A := \perp;$$
$$q_B := q_0; \quad q_A := \perp$$



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

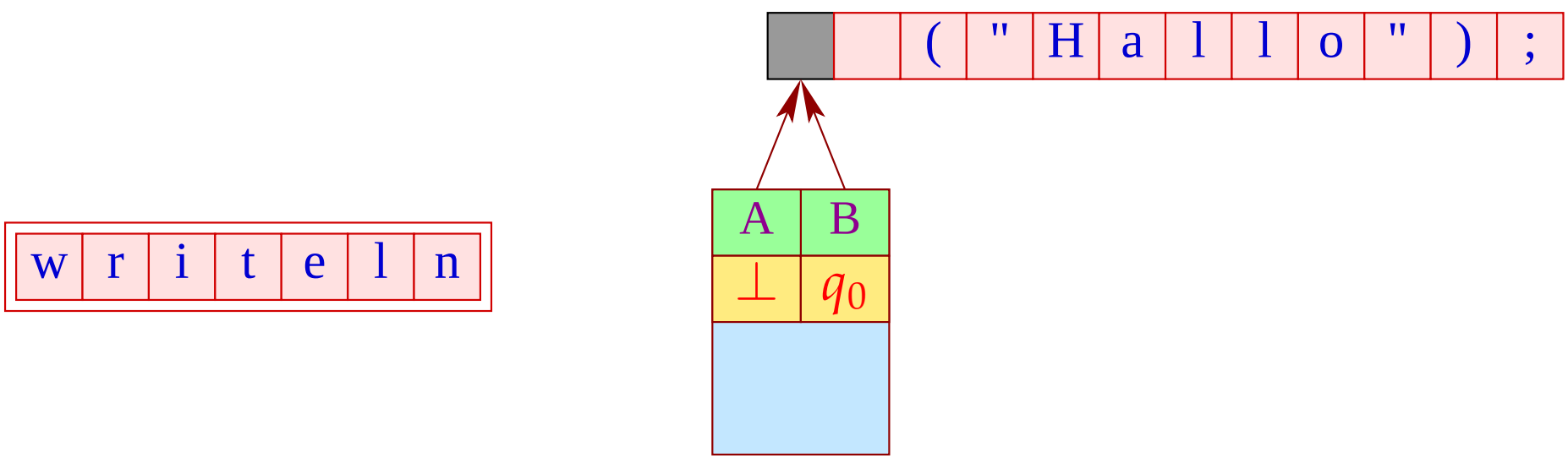
$$B := A; \quad A := \perp;$$
$$q_B := q_0; \quad q_A := \perp$$



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

$$B := A; \quad A := \bot;$$
$$q_B := q_0; \quad q_A := \bot$$



Erweiterung: Zustände

- Gelegentlich ist es nützlich, unterschiedliche Scanner-Zustände zu unterscheiden.
- In unterschiedlichen Zuständen sollen verschiedene Tokenklassen erkannt werden können.
- In Abhängigkeit der gelesenen Tokens kann der Scanner-Zustand geändert werden ;-)

Beispiel: Kommentare

Innerhalb eines Kommentars werden Identifier, Konstanten, Kommentare, ... nicht erkannt ;-)

Eingabe (verallgemeinert): eine Menge von Regeln:

$$
\begin{array}{llll}
\langle \texttt{state} \rangle & \{ & e_1 & \{\ \texttt{action}_1 \quad \texttt{yybegin}(\texttt{state}_1);\ \} \\
 & & e_2 & \{\ \texttt{action}_2 \quad \texttt{yybegin}(\texttt{state}_2);\ \} \\
 & & \ldots & \\
 & & e_k & \{\ \texttt{action}_k \quad \texttt{yybegin}(\texttt{state}_k);\ \} \\
 & \} & &
\end{array}
$$

- Der Aufruf `yybegin (state`$_i$`);` setzt den Zustand auf `state`$_i$.
- Der Startzustand ist (z.B. bei JFlex) `YYINITIAL`.

... im Beispiel:

$$
\begin{array}{llll}
\langle \texttt{YYINITIAL} \rangle & & \texttt{"/*"} & \{\ \texttt{yybegin(COMMENT);}\ \} \\
\langle \texttt{COMMENT} \rangle & \{ & \texttt{"*/"} & \{\ \texttt{yybegin(YYINITIAL);}\ \} \\
 & & \texttt{. | \backslash n} & \{\ \ \} \\
 & \} & &
\end{array}
$$

Bemerkungen:

- "`.`" matcht alle Zeichen ungleich "`\n`".
- Für jeden Zustand generieren wir den entsprechenden Scanner.
- Die Methode `yybegin (STATE);` schaltet zwischen den verschiedenen Scannern um.
- Kommentare könnte man auch direkt mithilfe einer geeigneten Token-Klasse implementieren. Deren Beschreibung ist aber ungleich komplizierter :-)
- Scanner-Zustände sind insbesondere nützlich bei der Implementierung von Präprozessoren, die in einen Text eingestreute Spezifikationen expandieren sollen.

1.4 Implementierung von DFAs

Aufgaben:

- Implementiere die Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- Implementiere eine Klassifizierung $r : Q \to \mathbb{N}$

1.4 Implementierung von DFAs

Aufgaben:

- Implementiere die Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- Implementiere eine Klassifizierung $r : Q \to \mathbb{N}$

Probleme:

- Die Anzahl der Zustände kann sehr groß sein :-(
- Das Alphabet kann sehr groß sein: z.B. Unicode :-((

Reduzierung der Anzahl der Zustände

Idee: Minimierung

- Identifiziere Zustände, die sich im Hinblick auf eine Klassifizierung r gleich verhalten :-)
- Sei $A = (Q, \Sigma, \delta, \{q_0\}, r)$ ein DFA mit Klassifizierung. Wir definieren auf den Zuständen eine Äquivalenzrelation durch:

$$p \equiv_r q \quad \text{gdw.} \quad \forall\, w \in \Sigma^* \;:\; r(\delta(p, w)) = r(\delta(q, w))$$

- Die neuen Zustände sind Äquivalenzklassen der alten Zustände :-)

Zustände	$[q]_r, q \in Q$
Anfangszustand	$[q_0]_r$
Klassifizierung	$r([q]_r) = r(q)$
Übergangsfunktion	$\delta([p]_r, a) = [\delta(p, a)]_r$

Problem: Wie berechnet man $\equiv_r$?

Idee:

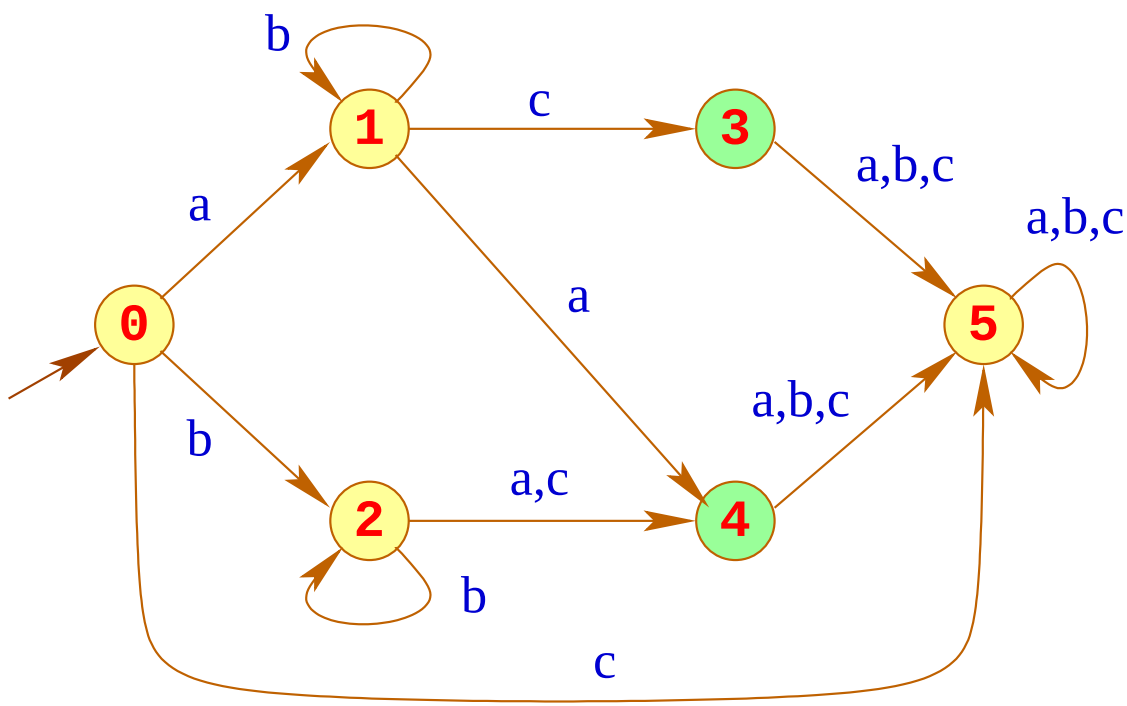
- Wir nehmen an, maximal viel sei äquivalent :-)

  Wir starten mit der Partition:

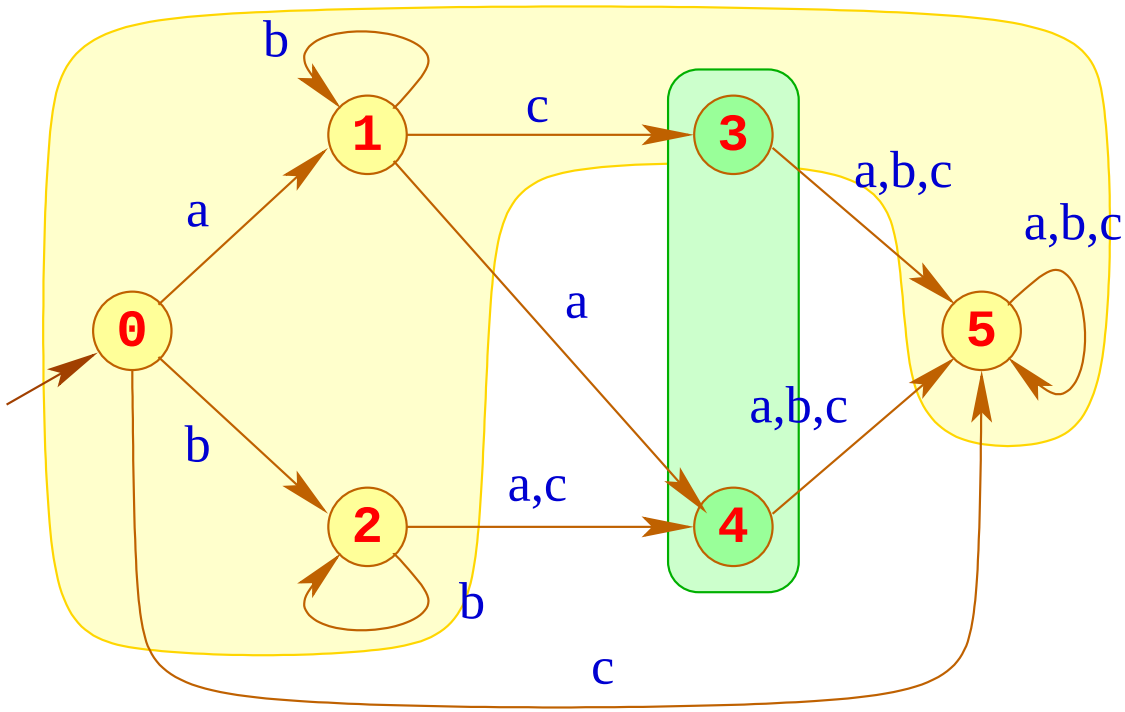
$$\overline{Q} = \{r^{-1}(i) \neq \emptyset \mid i \in \mathbb{N}\}$$

- Finden wir in $\bar{q} \in \overline{Q}$ Zustände p_1, p_2 sodass $\delta(p_1, a)$ und $\delta(p_2, a)$ in verschiedenen Äquivalenzklassen liegen (für irgend ein a), müssen wir $\bar{q}$ aufteilen ...

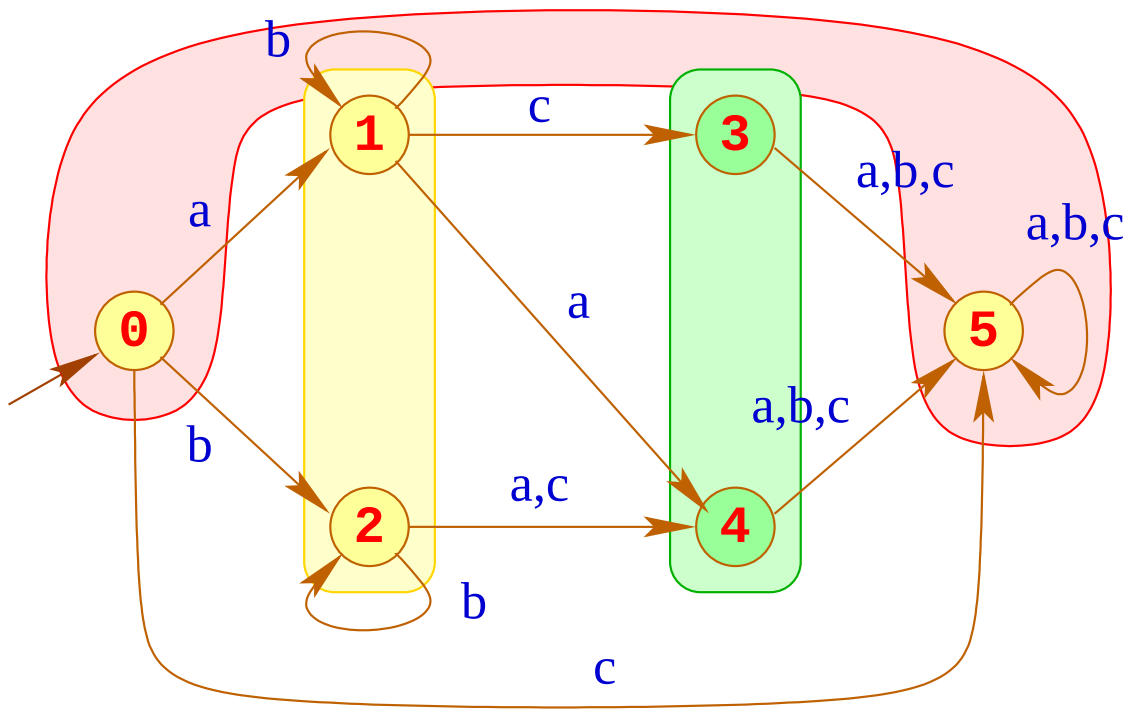
Beispiel:



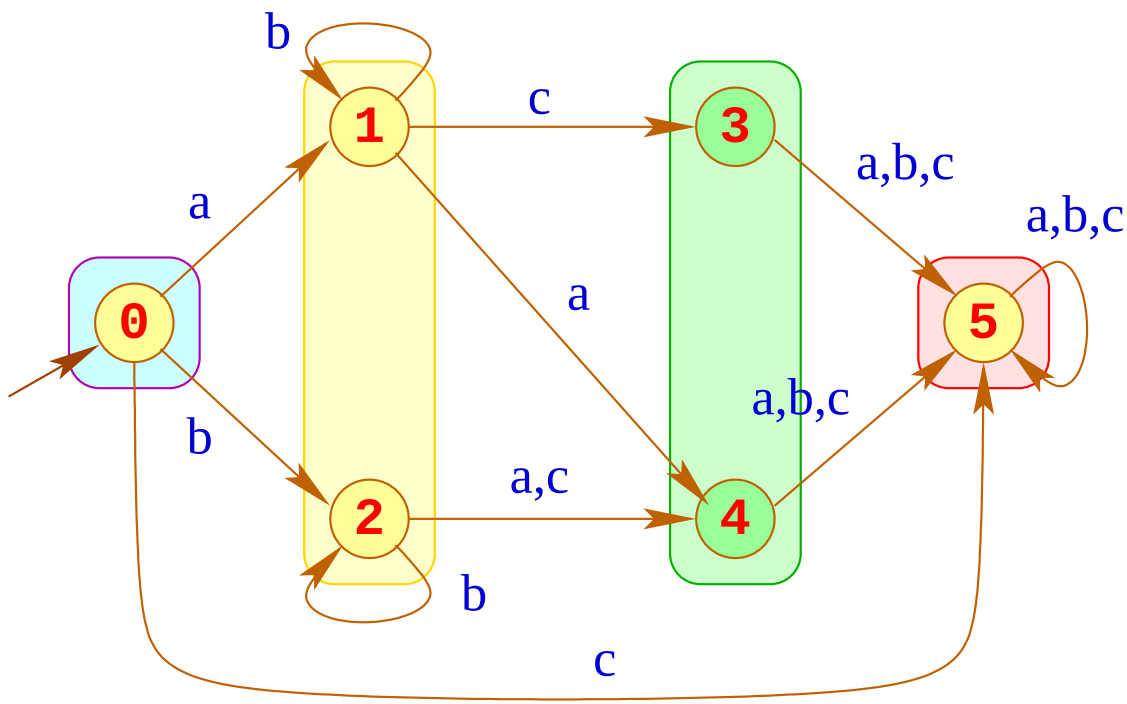
Beispiel:



Beispiel:



Beispiel:



Bemerkungen:

- Das Verfahren liefert die gröbste Partition $\overline{Q}$, die mit r und δ verträglich ist, d.h. für $\bar{q} \in \overline{Q}$,

  (1) $p_1, p_2 \in \bar{q} \implies r(p_1) = r(p_2)$

  (2) $p_1, p_2 \in \bar{q} \implies \delta(p_1, a), \delta(p_2, a)$ gehören zur gleichen Klasse

- Der Ergebnis-Automat ist der eindeutig bestimmte minimale Automat für $\mathcal{L}(A)$;-)
- Eine naive Implementierung erfordert Laufzeit $\mathcal{O}(n^2)$.

  Eine raffinierte Verwaltung der Partition liefert ein Verfahren mit Laufzeit $\mathcal{O}(n \cdot \log(n))$.

Anil Nerode , Cornell University, Ittaca

John E. Hopcroft, Cornell University, Ittaca

Reduzierung der Tabellengröße

Problem:

- Die Tabelle für δ wird mit Paaren (q, a) indiziert.
- Sie enthält eine Spalte für jedes $a \in \Sigma$.
- Das Alphabet Σ umfasst i.a. ASCII, evt. aber ganz Unicode :-(

1. Idee:

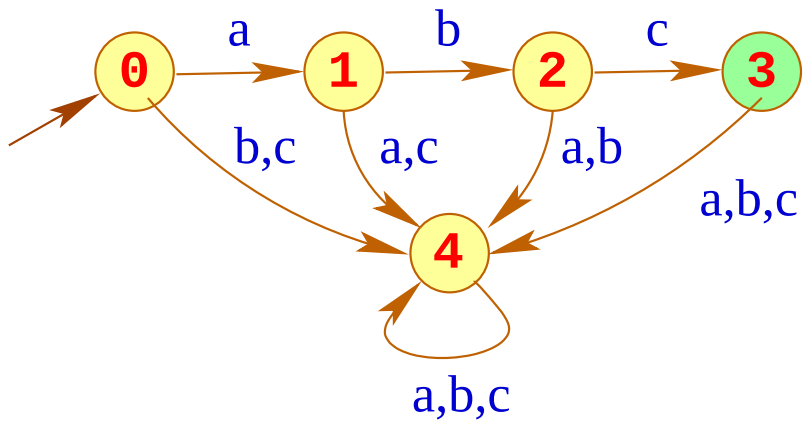
- Bei großen Alphabeten wird man in der Spezifikation i.a. nicht einzelne Zeichen auflisten, sondern Zeichenklassen benutzen :-)
- Lege Spalten nicht für einzelne Zeichen sondern für Klassen von Zeichen an, die sich gleich verhalten.

Beispiel:

```
le = [a-zA-Z_\$]
ledi = [a-zA-Z_\$0-9]
Id = {le} {ledi}*
```

- Der Automat soll deterministisch sein.
- Sind die Klassen der Spezifikation nicht disjunkt, teilt man sie darum in Unterklassen auf, hier in die Klassen `[a-zA-Z_\$]` und `[0-9]` :-)

2. Idee:

- Finden wir, dass mehrere (Unter-) Klassen der Spezifikation in der Spalte übereinstimmen, können wir sie nachträglich wieder vereinigen :-)
- Wir können weitere Methoden der Tabellen-Komprimierung anwenden, z.B. Zeilenverschiebung (Row Displacement) ...

Beispiel: