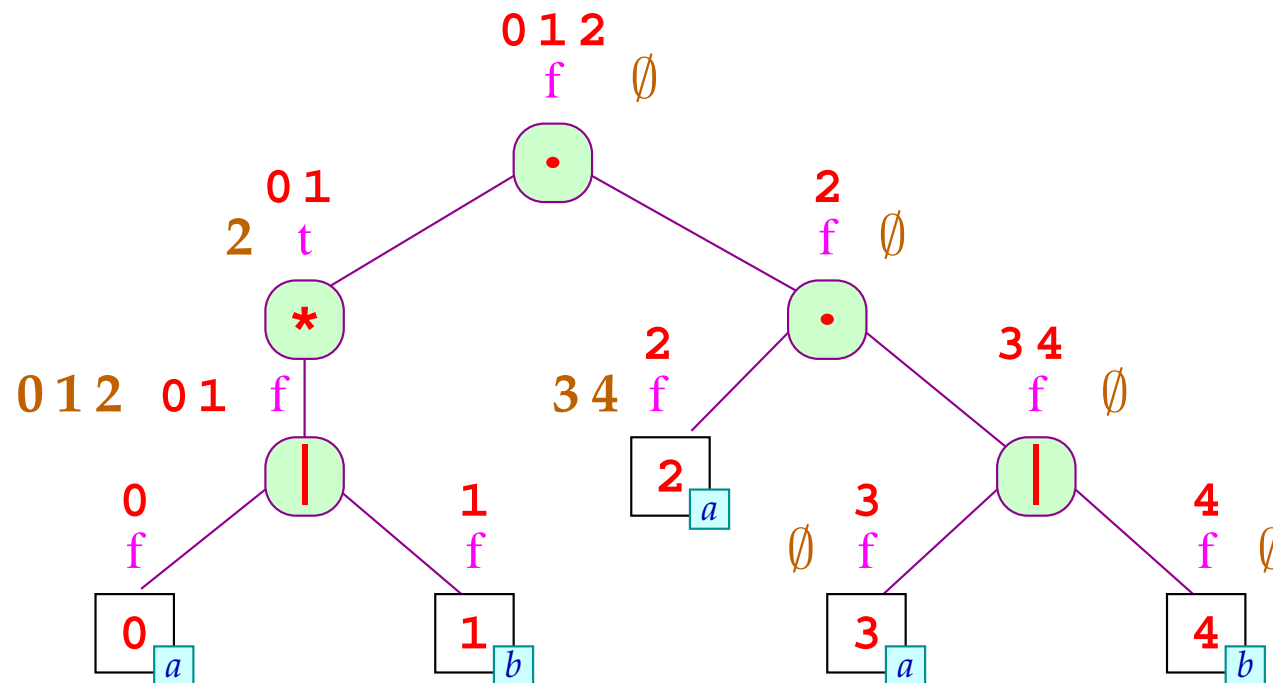


3. Schritt:

Die Menge **nächster** Bätter:

$$\text{next}[r] = \{i \mid (r \bullet, \epsilon, \bullet \begin{array}{|c|c|} \hline i & x \\ \hline \end{array}) \in \delta^*\}$$

... im Beispiel:

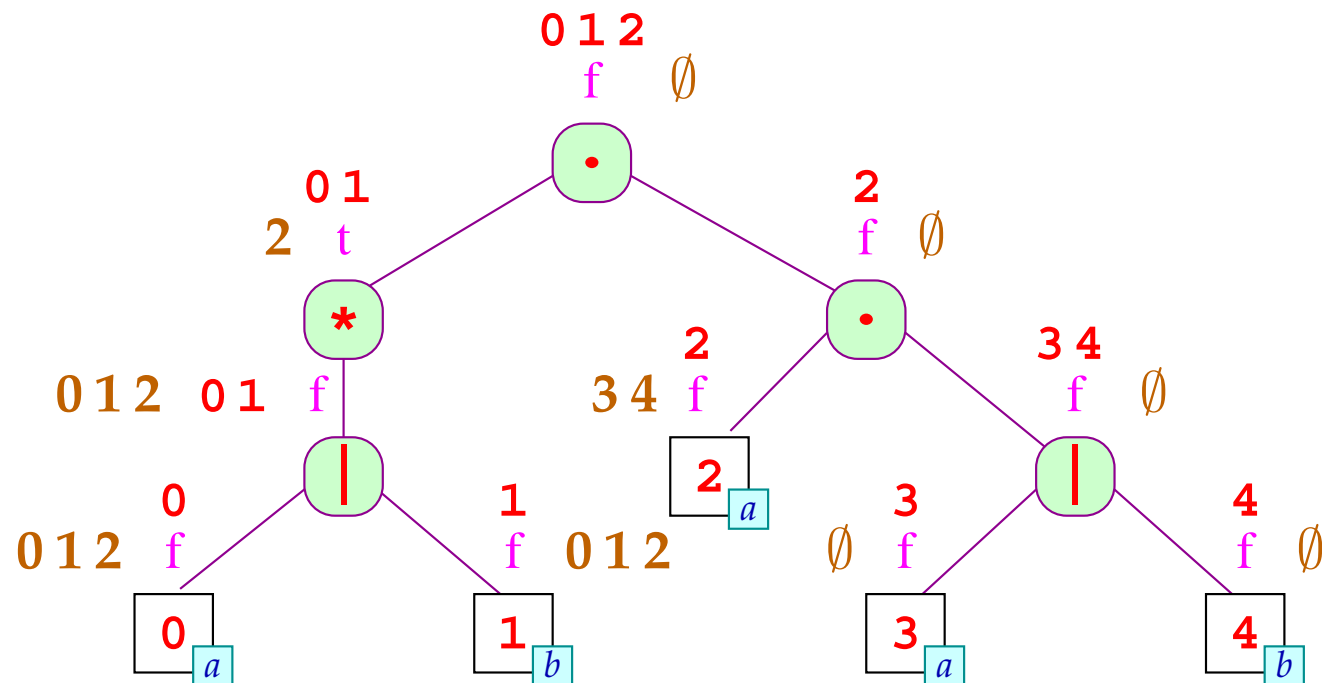


3. Schritt:

Die Menge **nächster** Bätter:

$$\text{next}[r] = \{i \mid (r \bullet, \epsilon, \bullet \begin{array}{|c|c|} \hline i & x \\ \hline \end{array}) \in \delta^*\}$$

... im Beispiel:



Implementierung: DFS pre-order Traversierung ;-)

Für die Wurzel haben wir:

$$\text{next}[e] = \emptyset$$

Ansonsten machen wir eine Fallunterscheidung über den Kontext:

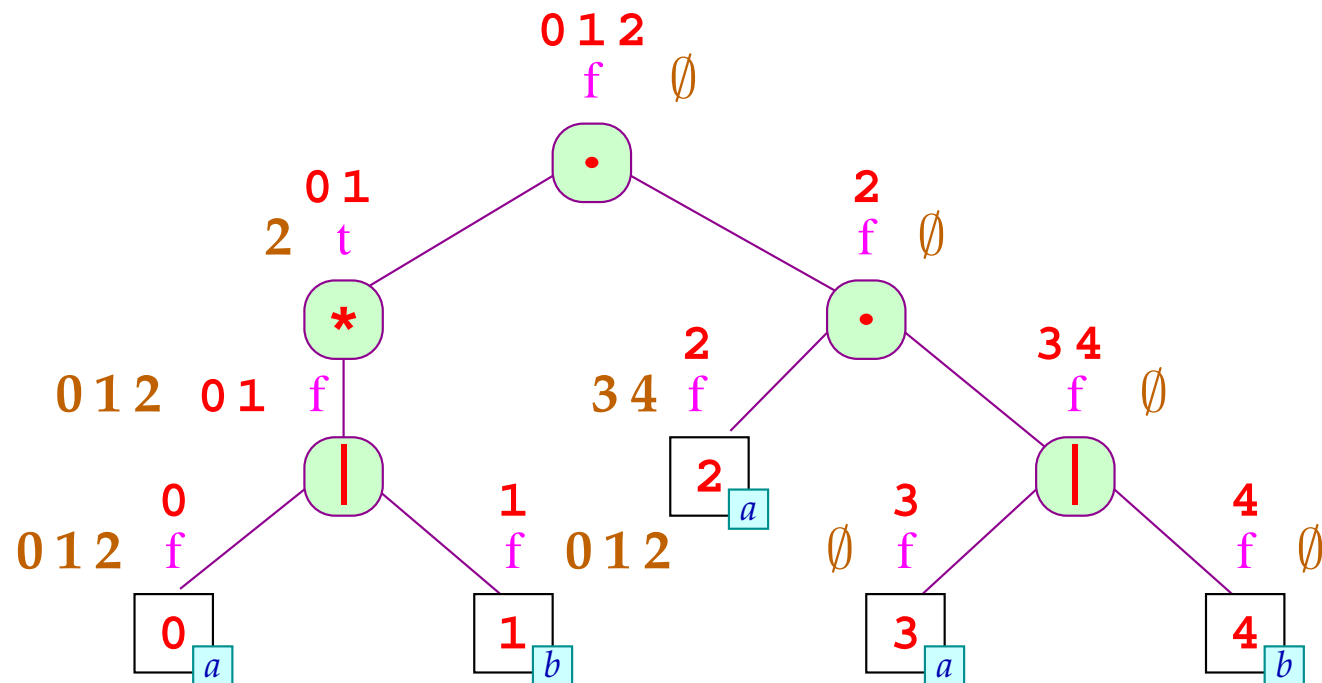
r	Regeln
$r_1 \mid r_2$	$\text{next}[r_1] = \text{next}[r]$ $\text{next}[r_2] = \text{next}[r]$
$r_1 \cdot r_2$	$\text{next}[r_1] = \begin{cases} \text{first}[r_2] \cup \text{next}[r] & \text{falls } \text{empty}[r_2] = t \\ \text{first}[r_2] & \text{falls } \text{empty}[r_2] = f \end{cases}$ $\text{next}[r_2] = \text{next}[r]$
r_1^*	$\text{next}[r_1] = \text{first}[r_1] \cup \text{next}[r]$
$r_1?$	$\text{next}[r_1] = \text{next}[r]$

4. Schritt:

Die Menge **letzter** Bätter:

$$\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x} \bullet, \epsilon, r \bullet) \in \delta^*\}$$

... im Beispiel:

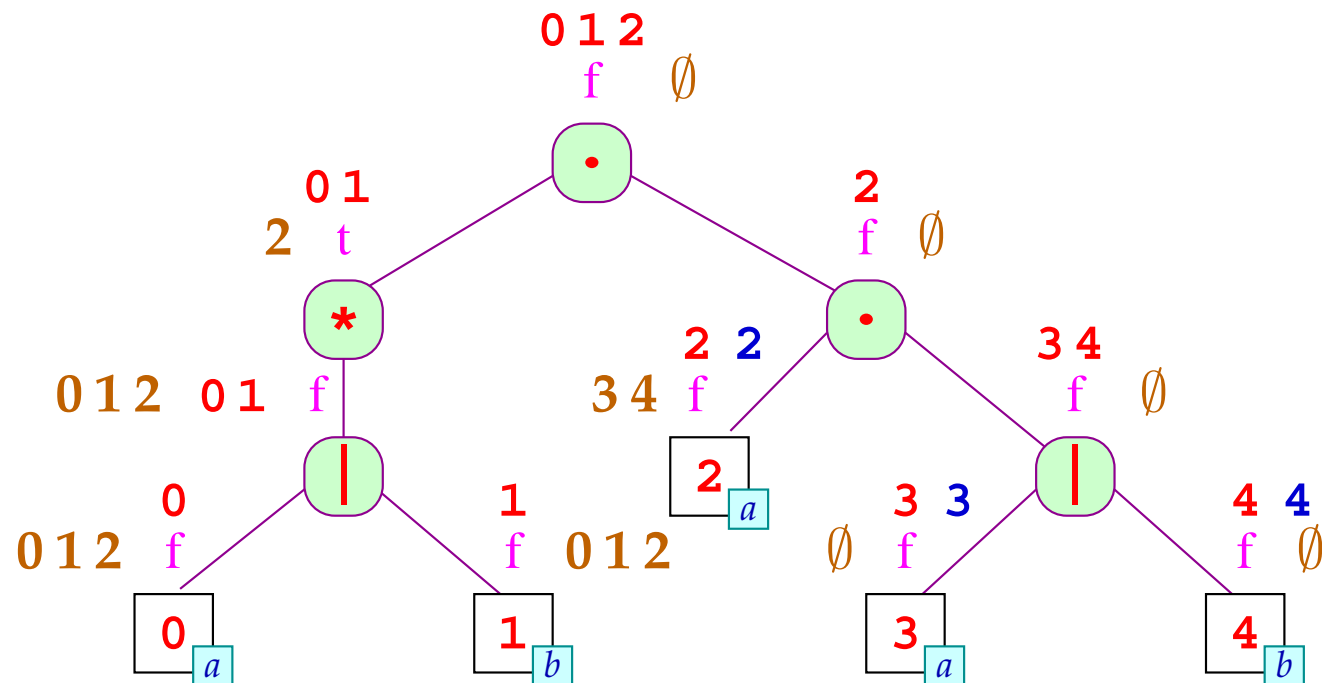


4. Schritt:

Die Menge **letzter** Bätter:

$$\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x} \bullet, \epsilon, r \bullet) \in \delta^*\}$$

... im Beispiel:

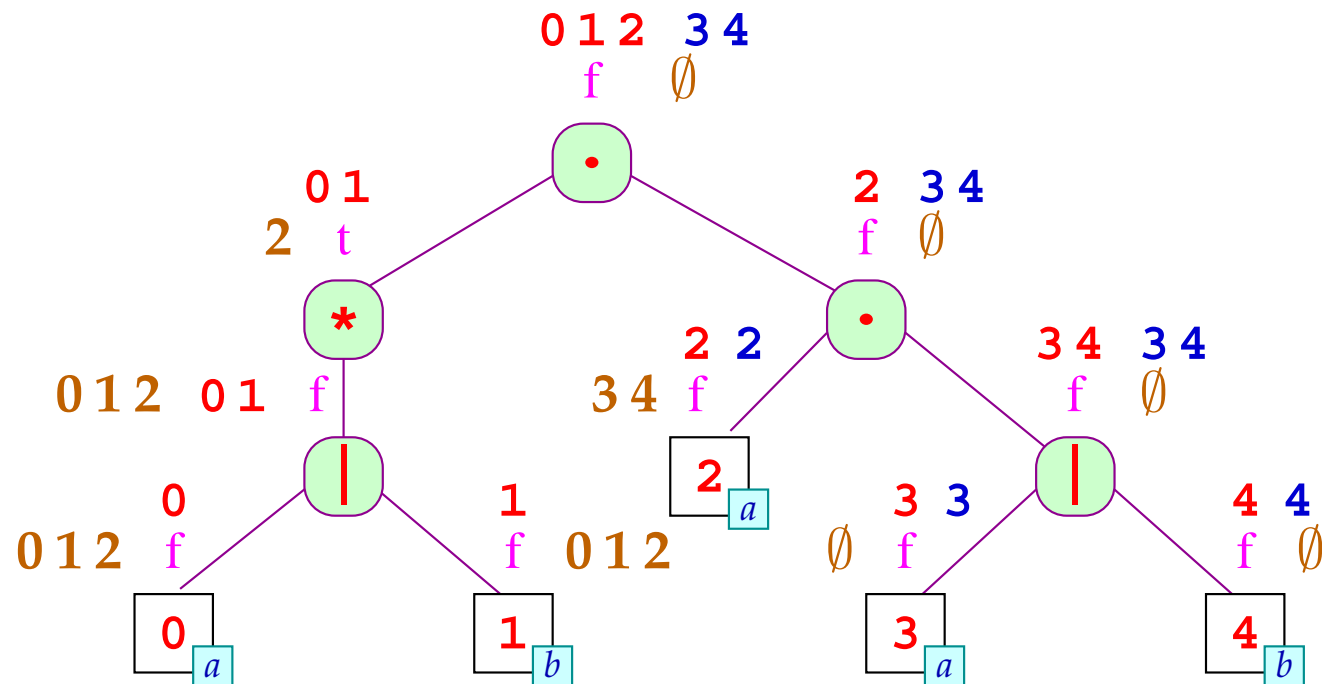


4. Schritt:

Die Menge **letzter** Bätter:

$$\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x} \bullet, \epsilon, r \bullet) \in \delta^*, x \neq \epsilon\}$$

... im Beispiel:



Implementierung: DFS post-order Traversierung :-)

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\text{last}[r] = \{i \mid x \neq \epsilon\}$.

Andernfalls:

$$\begin{aligned}\text{last}[r_1 \mid r_2] &= \text{last}[r_1] \cup \text{last}[r_2] \\ \text{last}[r_1 \cdot r_2] &= \begin{cases} \text{last}[r_1] \cup \text{last}[r_2] & \text{falls } \text{empty}[r_2] = t \\ \text{last}[r_2] & \text{falls } \text{empty}[r_2] = f \end{cases} \\ \text{last}[r_1^*] &= \text{last}[r_1] \\ \text{last}[r_1?] &= \text{last}[r_1]\end{aligned}$$

Integration:

Zustände: $\{\bullet e\} \cup \{i\bullet \mid i \text{ Blatt}\}$

Startzustand: $\bullet e$

Endzustände:

Falls $\text{empty}[e] = f$, dann $\text{last}[e]$. Andernfalls: $\{\bullet e\} \cup \text{last}[e]$.

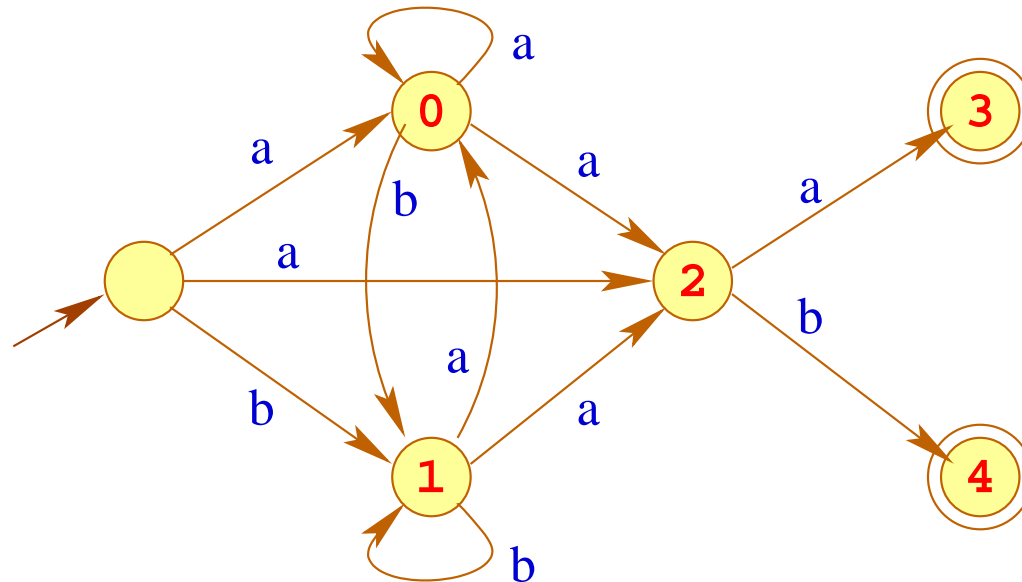
Übergänge:

$(\bullet e, a, i\bullet)$ falls $i \in \text{first}[e]$ und i mit a beschriftet ist;

$(i\bullet, a, i'\bullet)$ falls $i' \in \text{next}[i]$ und i' mit a beschriftet ist.

Den resultierenden Automaten bezeichnen wir mit A_e .

... im Beispiel:

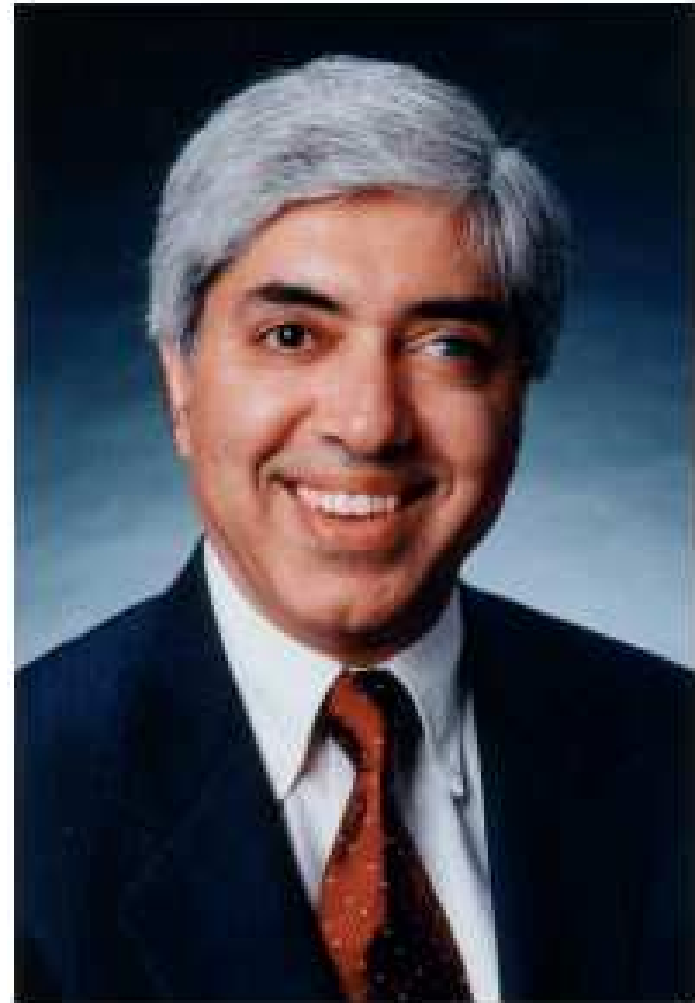


Bemerkung:

- 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 ...

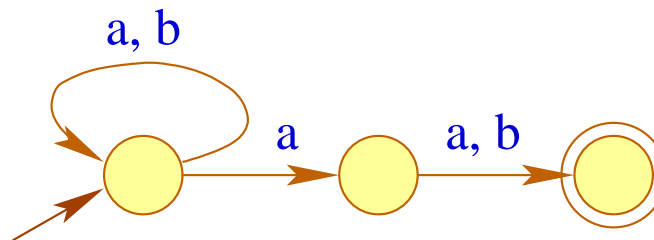


Gerard Berry, Esterel Technologies



Ravi Sethi, Research VR, Lucent
Technologies

Der erwartete Automat:

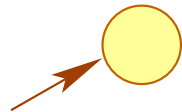
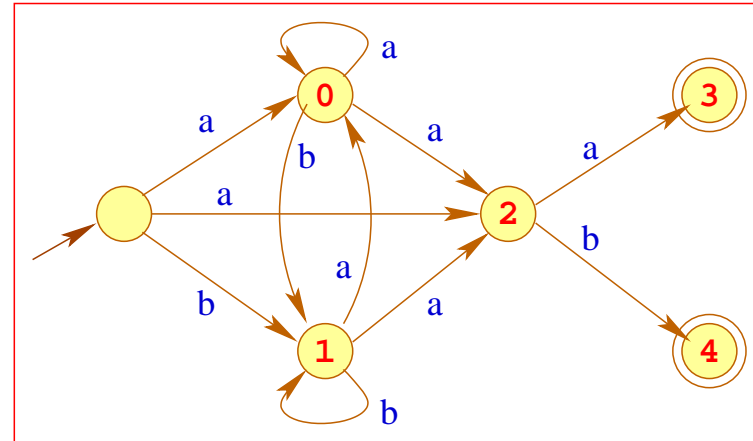


Bemerkung:

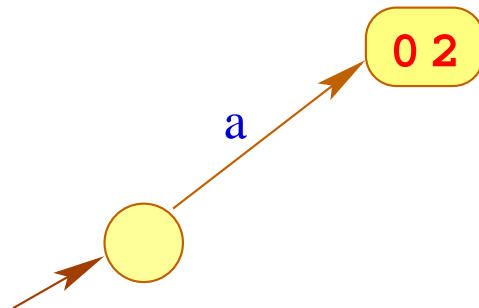
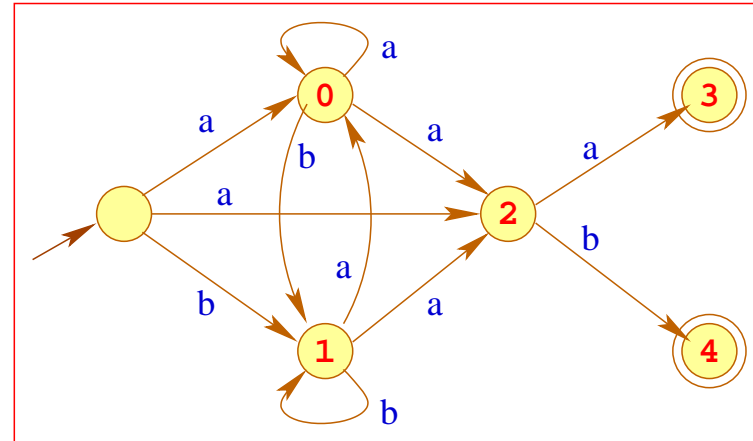
- in einen Zustand eingehende Kanten haben hier nicht unbedingt die gleiche Beschriftung :-)
- Dafür ist die Berry-Sethi-Konstruktion direkter ;-)
- In Wirklichkeit benötigen wir aber **deterministische** Automaten

⇒ Teilmengen-Konstruktion

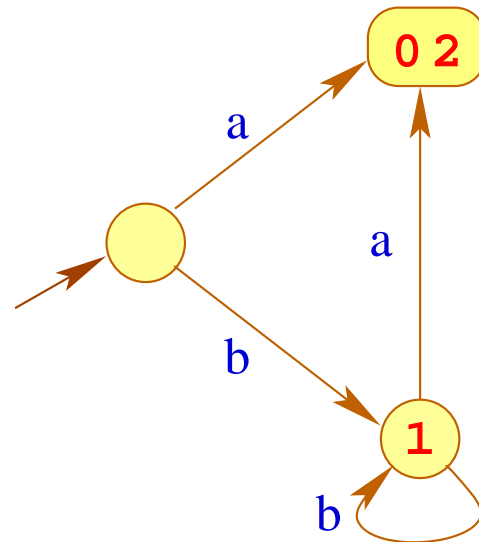
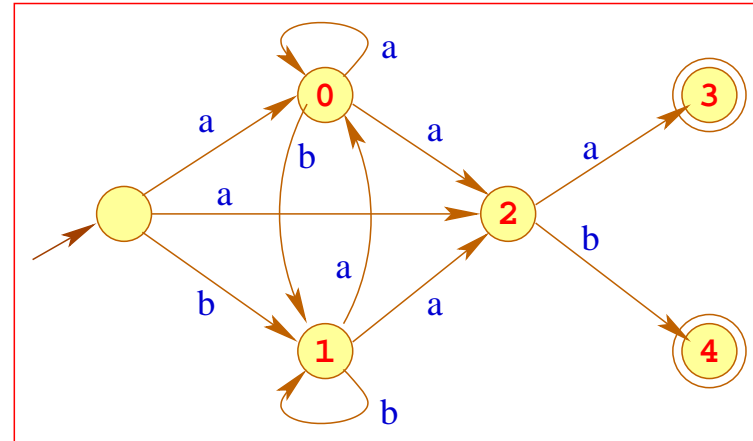
... im Beispiel:



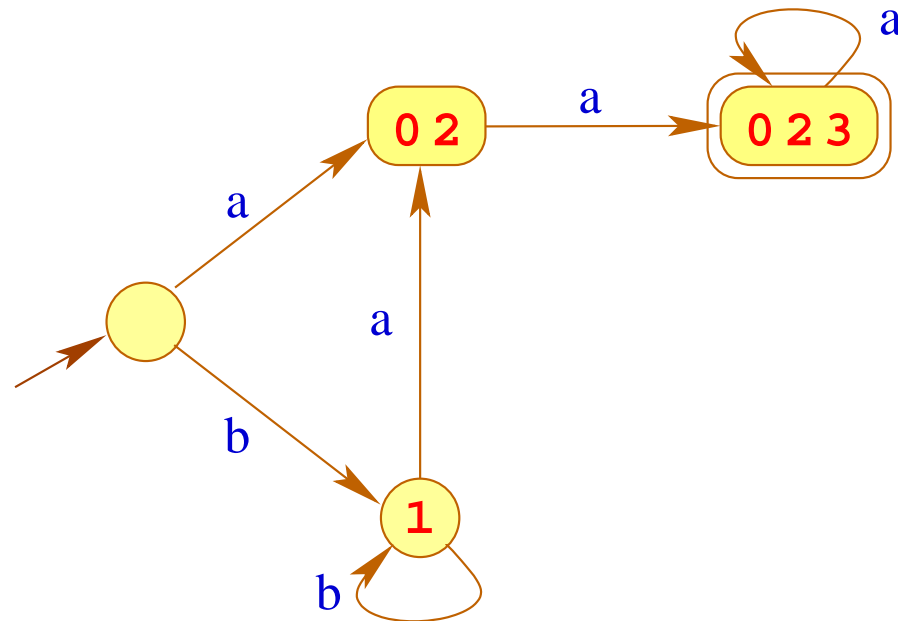
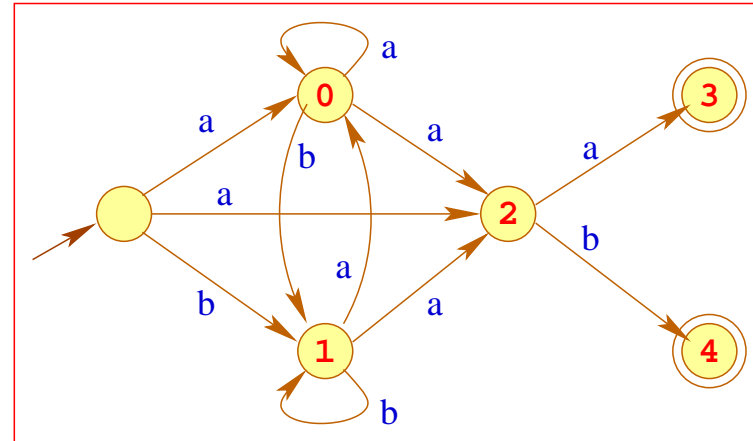
... im Beispiel:



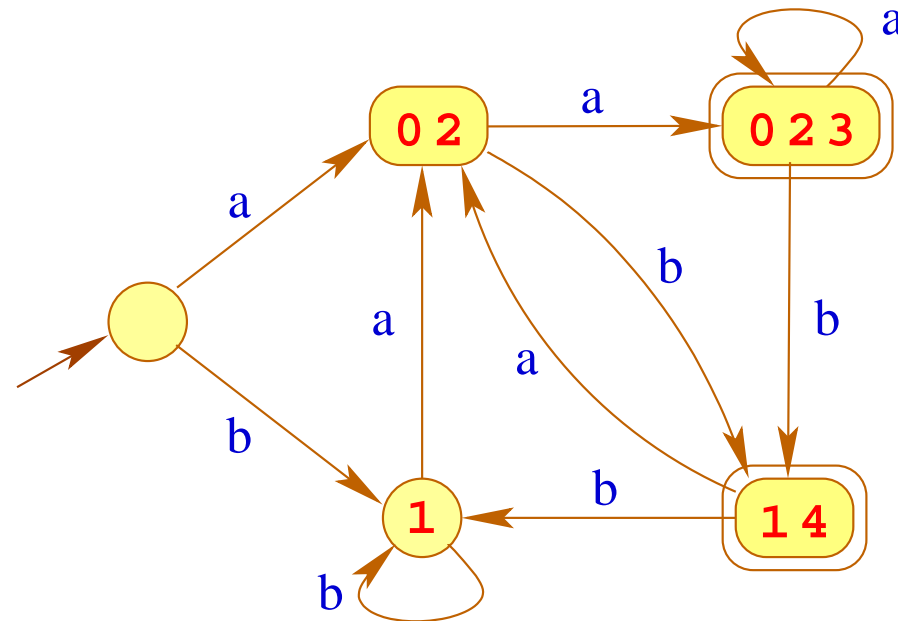
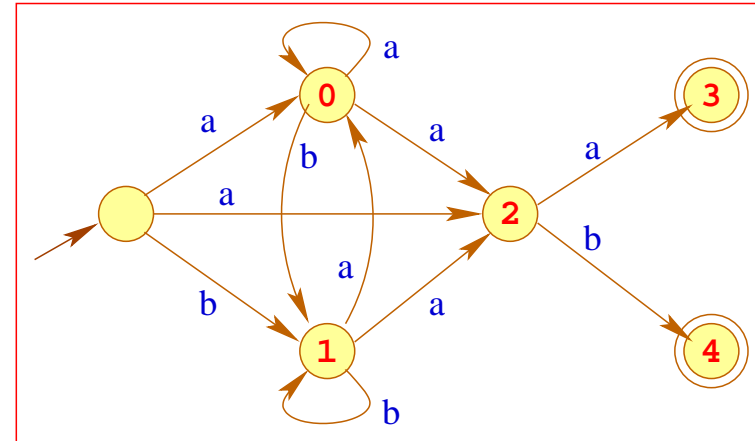
... im Beispiel:



... 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:

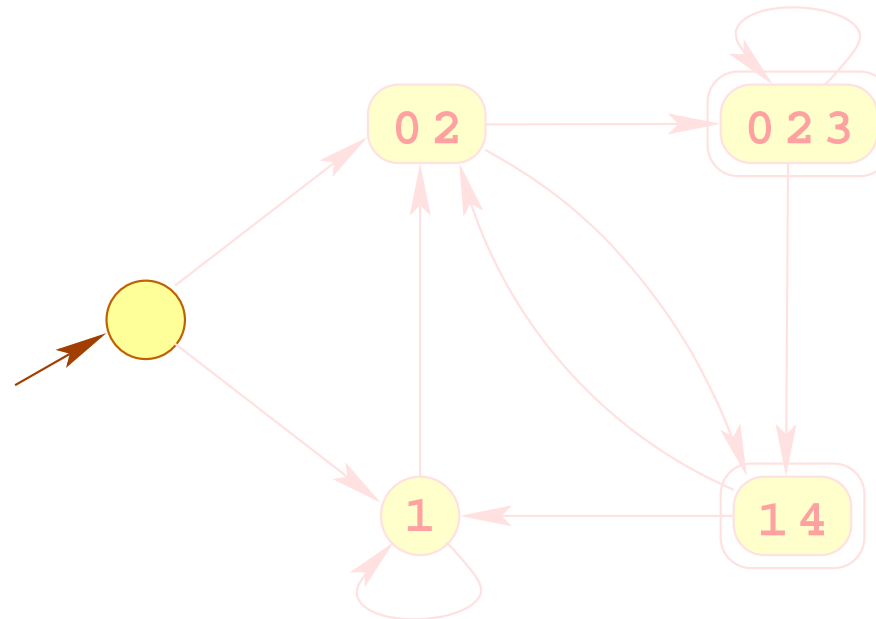
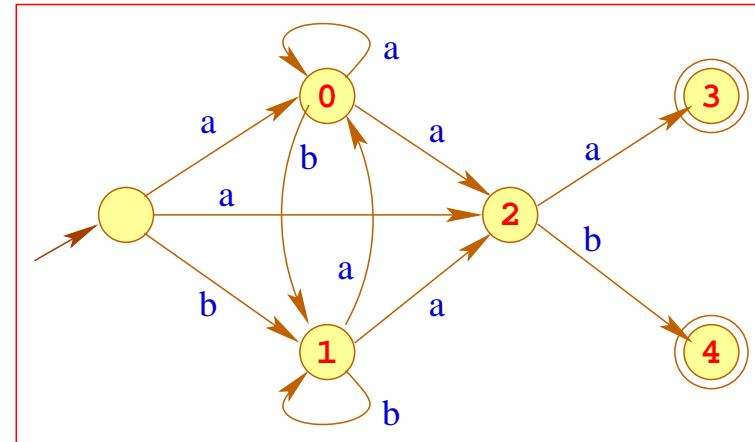
- Leider gibt es exponentiell viele Teilmengen von Q :-)
- Um nur **nützliche** Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur **nach Bedarf** hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
-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

Achtung:

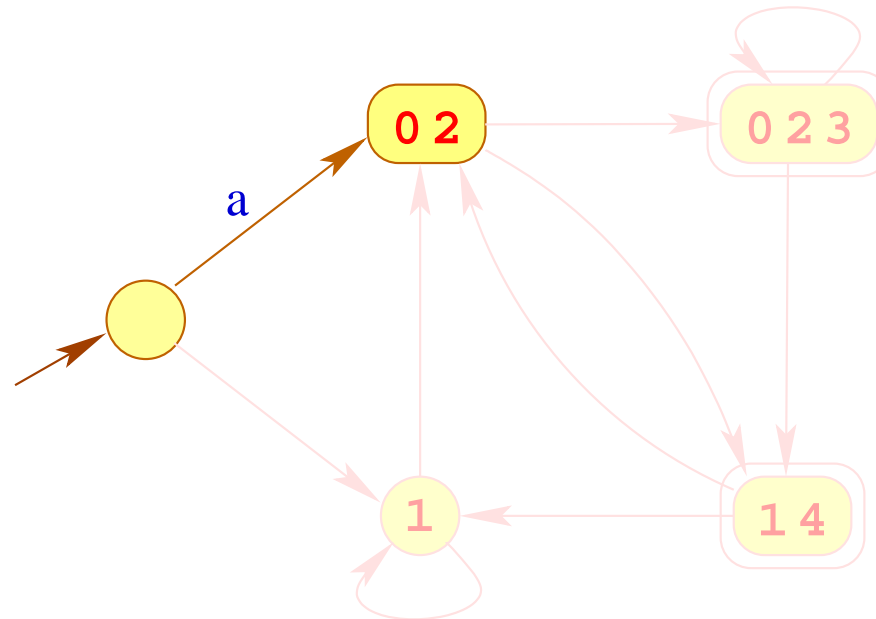
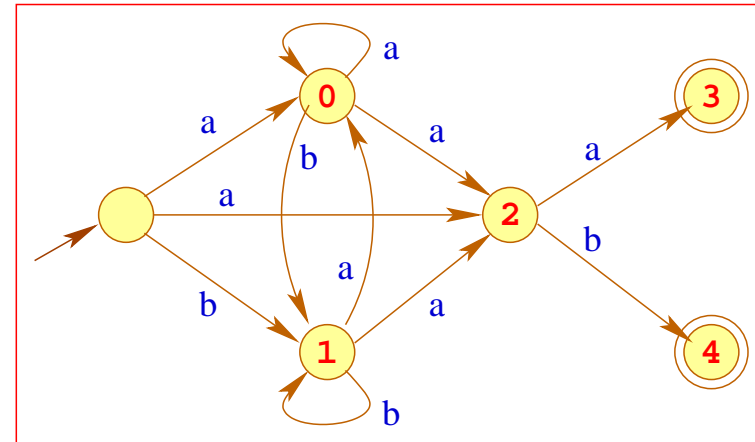
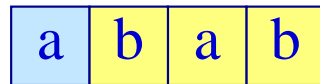
- Leider gibt es exponentiell viele Teilmengen von Q :-)
- Um nur **nützliche** Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur **nach Bedarf** hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
- 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 Abarbeitung der Eingabe** genau die Mengen konstruiert, die für die Eingabe notwendig sind ...

... im Beispiel:

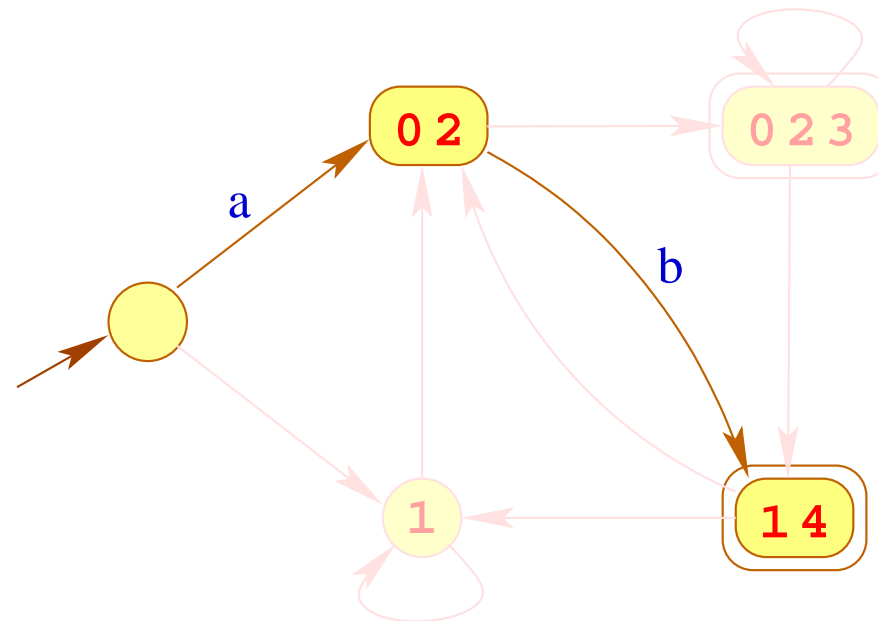
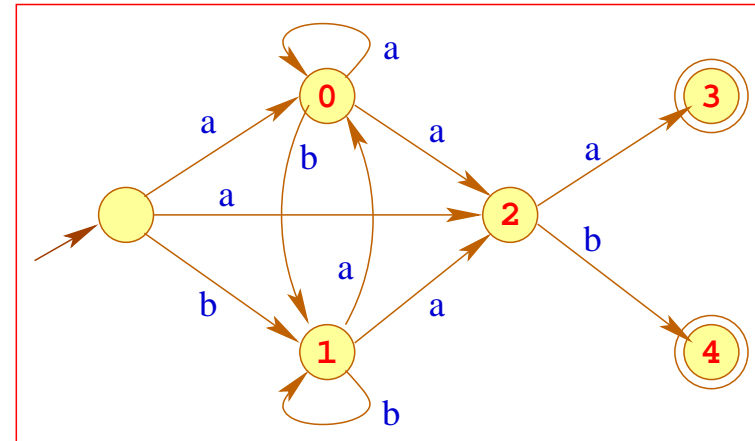
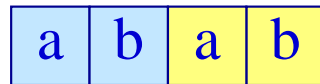
a	b	a	b
---	---	---	---



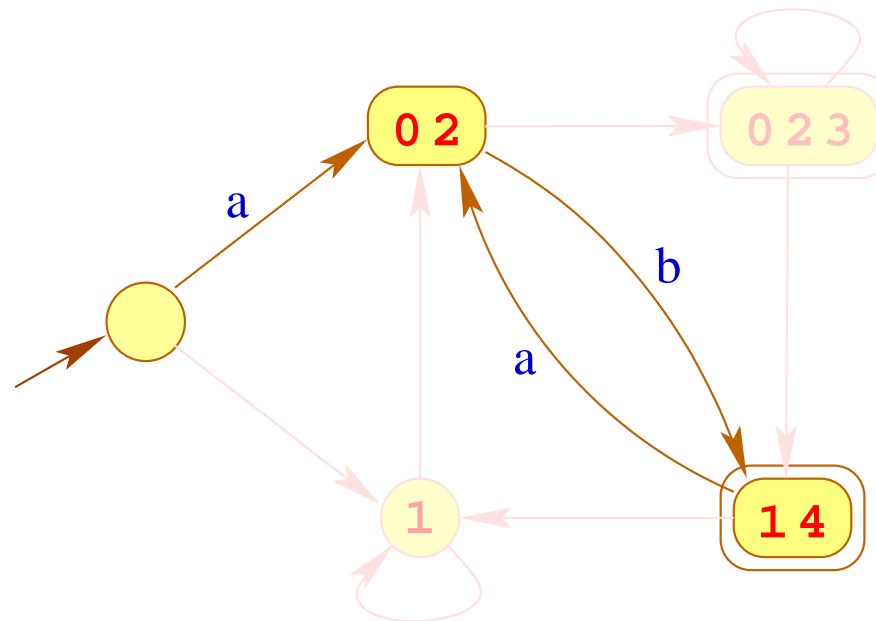
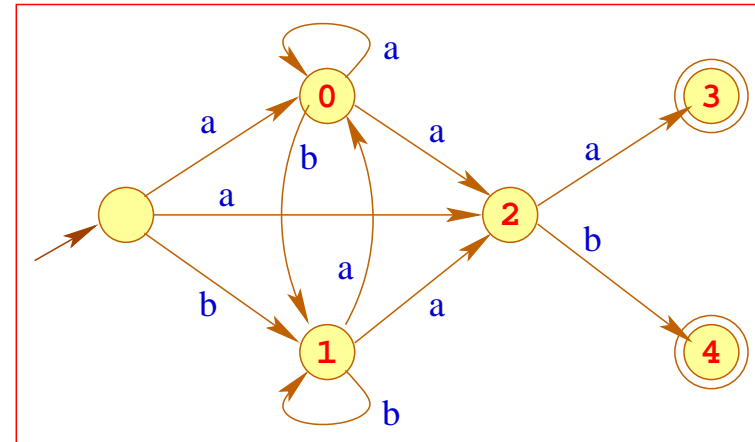
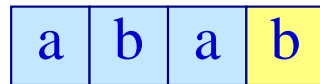
... im Beispiel:



... im Beispiel:

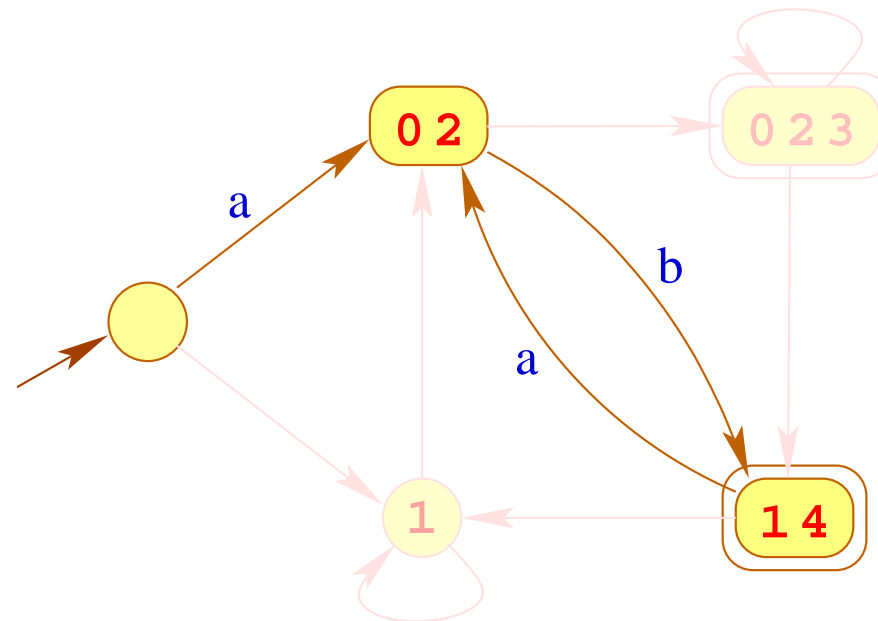
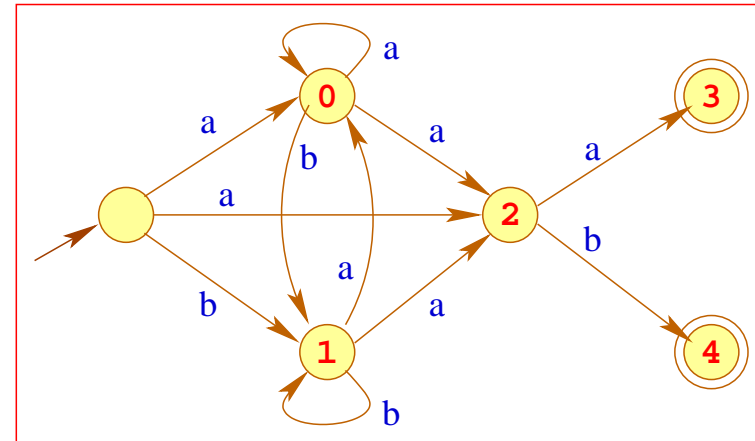


... im Beispiel:



... 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 konstruiert, 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 konstruiert, 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

$$\mathcal{L}(A) = \llbracket e \rrbracket$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

e_1 $\{ \text{action}_1 \}$

e_2 $\{ \text{action}_2 \}$

...

e_k $\{ \text{action}_k \}$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

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

Ausgabe: ein Programm, das

- ... von der Eingabe ein maximales Präfix w liest, das $e_1 \mid \dots \mid e_k$ erfüllt;
- ... das minimale i ermittelt, so dass $w \in \llbracket e_i \rrbracket$;
- ... für w action_i ausführt.

Implementierung:

Idee:

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

$$F_1 = \{q \in F \mid q \cap \text{last}[e_1] \neq \emptyset\}$$

$$F_2 = \{q \in (F \setminus F_1) \mid q \cap \text{last}[e_2] \neq \emptyset\}$$

...

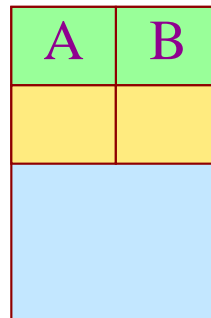
$$F_k = \{q \in (F \setminus (F_1 \cup \dots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset\}$$

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

Idee (Fortsetzung):

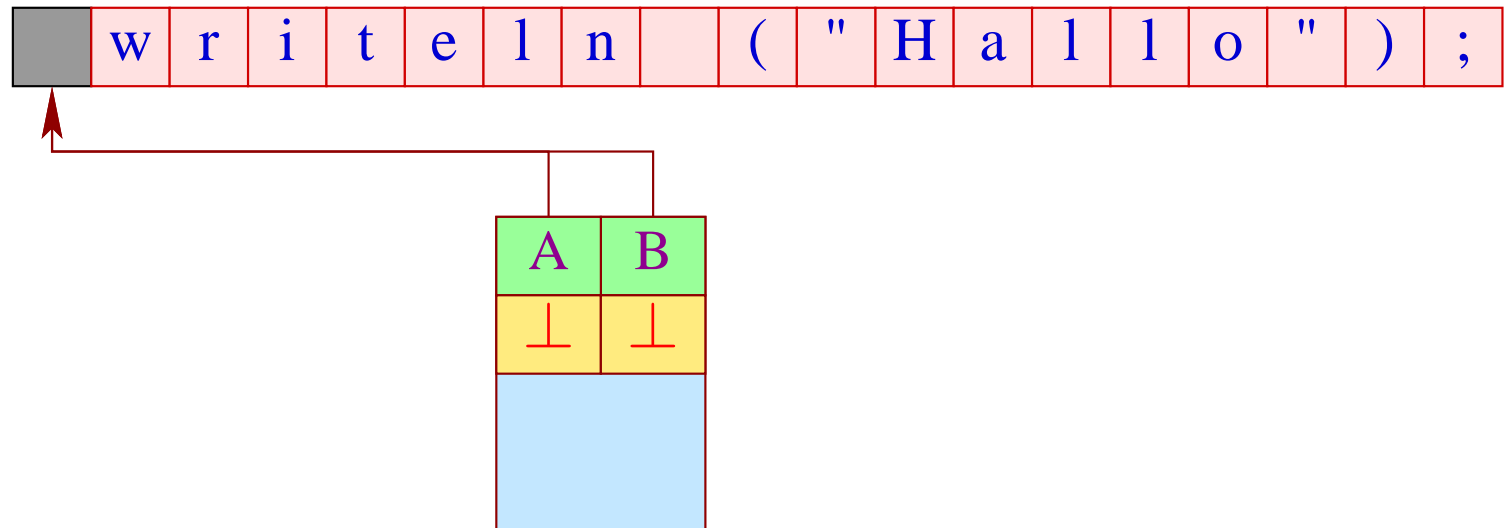
- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle \dots$
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.

s	t	d	o	u	t	.	w	r	i	t	e	l	n		("	H	a	l	l	o	")	;
---	---	---	---	---	---	---	---	---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---



Idee (Fortsetzung):

- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle \dots$
- 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.

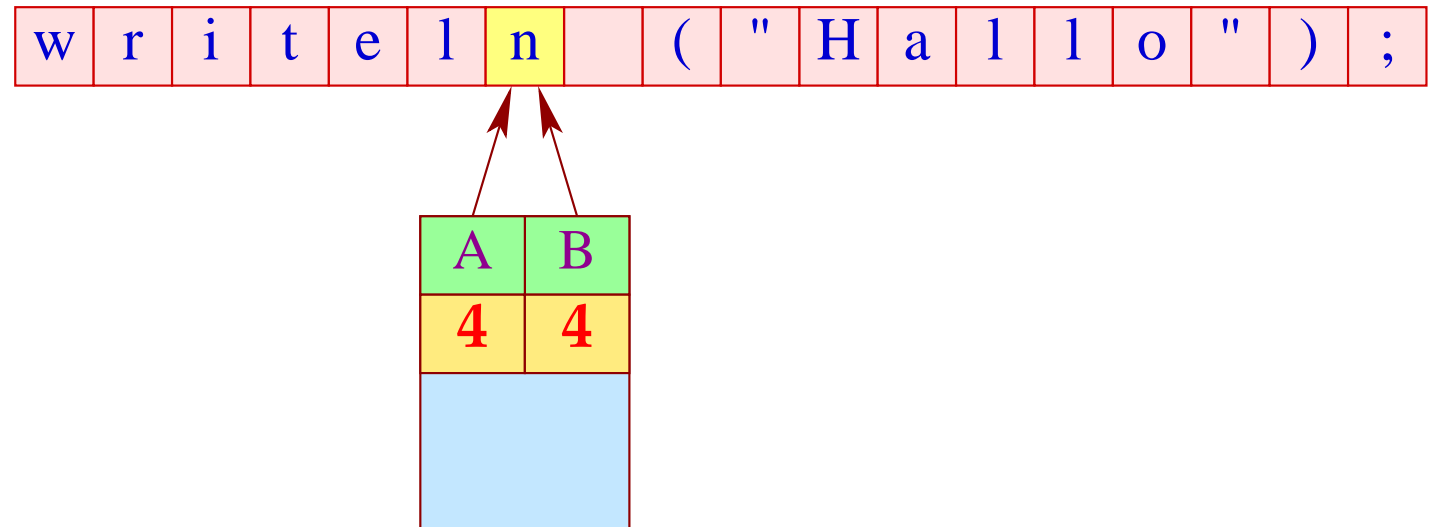


Idee (Fortsetzung):

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

$B := A; \quad A := \perp;$

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

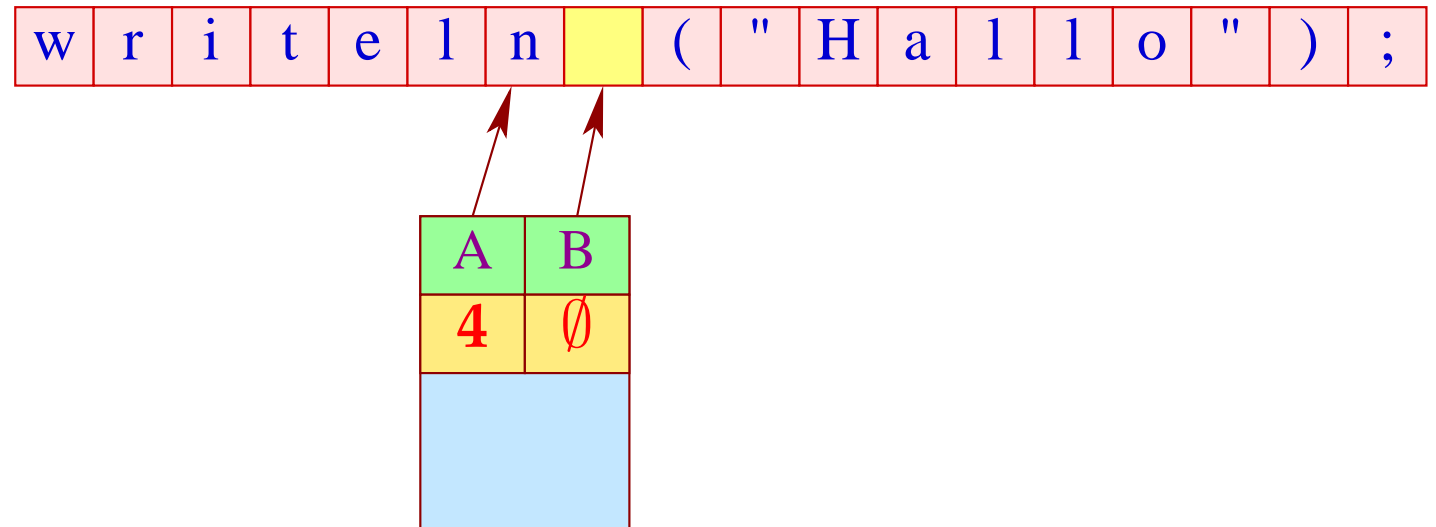


Idee (Fortsetzung):

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

$B := A; \quad A := \perp;$

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



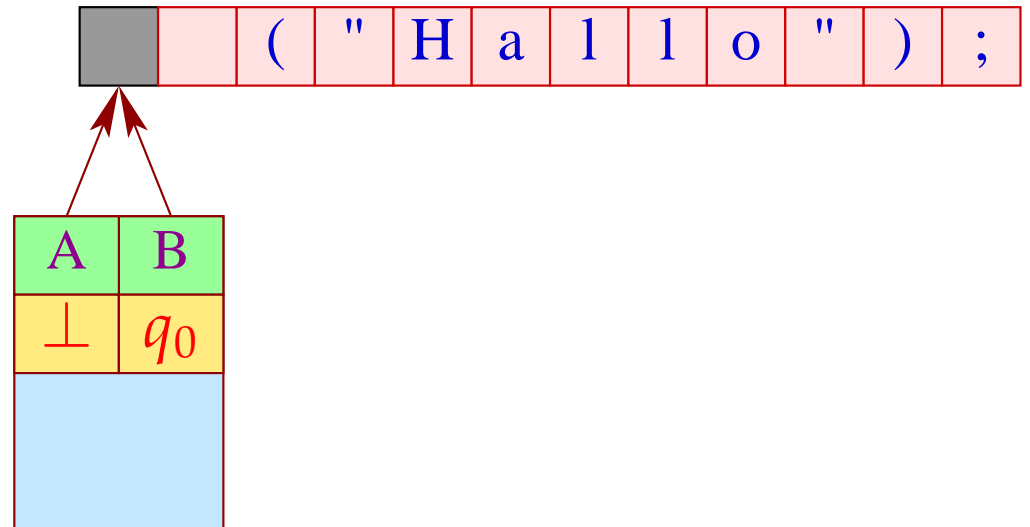
Idee (Fortsetzung):

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

$B := A; \quad A := \perp;$

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

w	r	i	t	e	l	n
---	---	---	---	---	---	---



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:

```
⟨state⟩ {  e1      { action1 yybegin(state1); }
           e2      { action2 yybegin(state2); }
           ...
           ek      { actionk yybegin(statek); }
        }
```

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

... im Beispiel:

```
⟨YYINITIAL⟩    "/" * " { yybegin(COMMENT); }
⟨COMMENT⟩     {  " * "/" { yybegin(YYINITIAL); }
               . | \n {   }
               }
```

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

1.4 Implementierung von DFAs

Aufgaben:

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

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:

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

- 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:

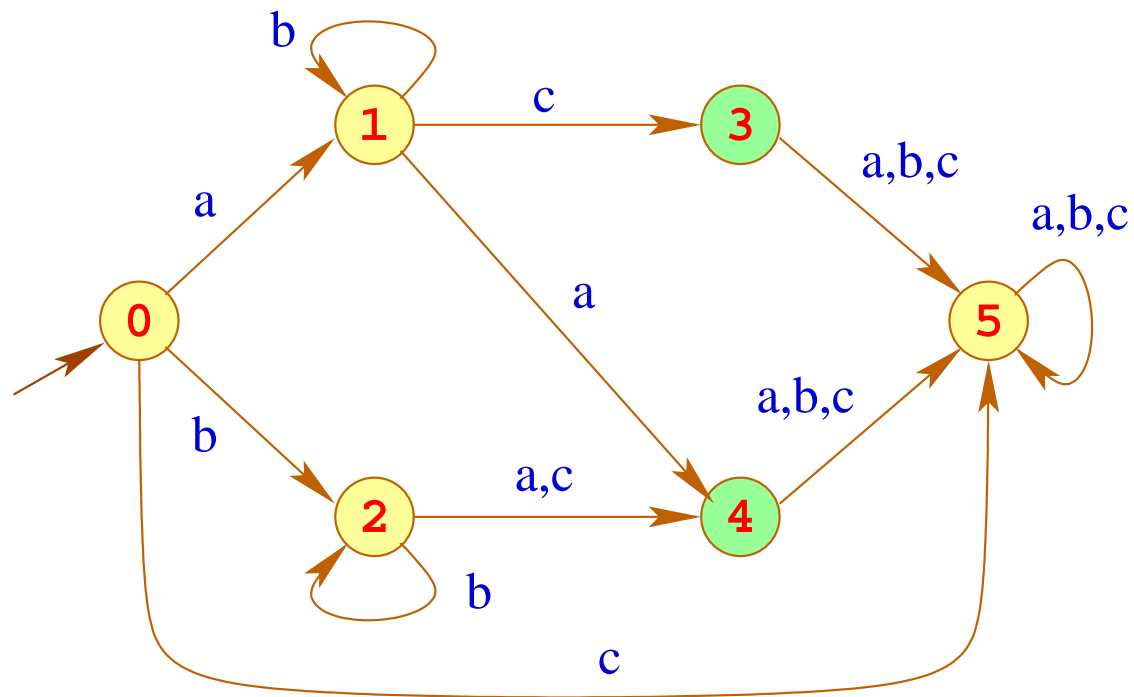
- Wir nehmen an, **maximal viel** sei äquivalent :-)

Wir starten mit der Partition:

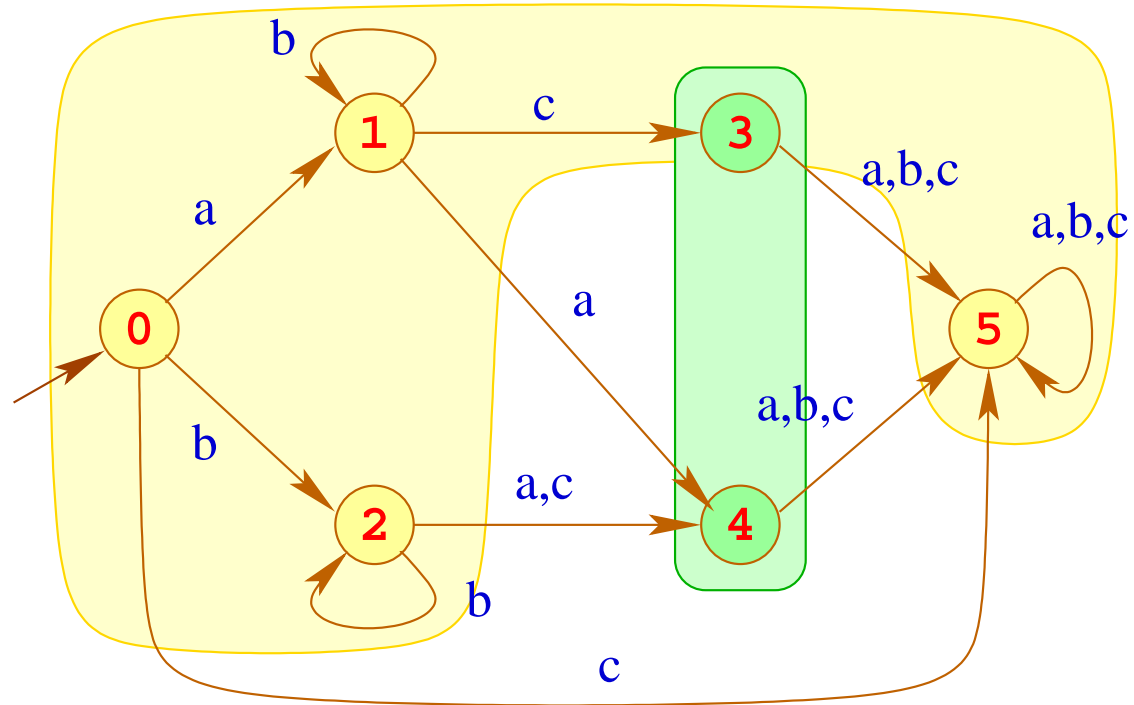
$$\overline{Q} = \{r^{-1}(i) \neq \emptyset \mid i \in \mathbb{N}\}$$

- 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 ...

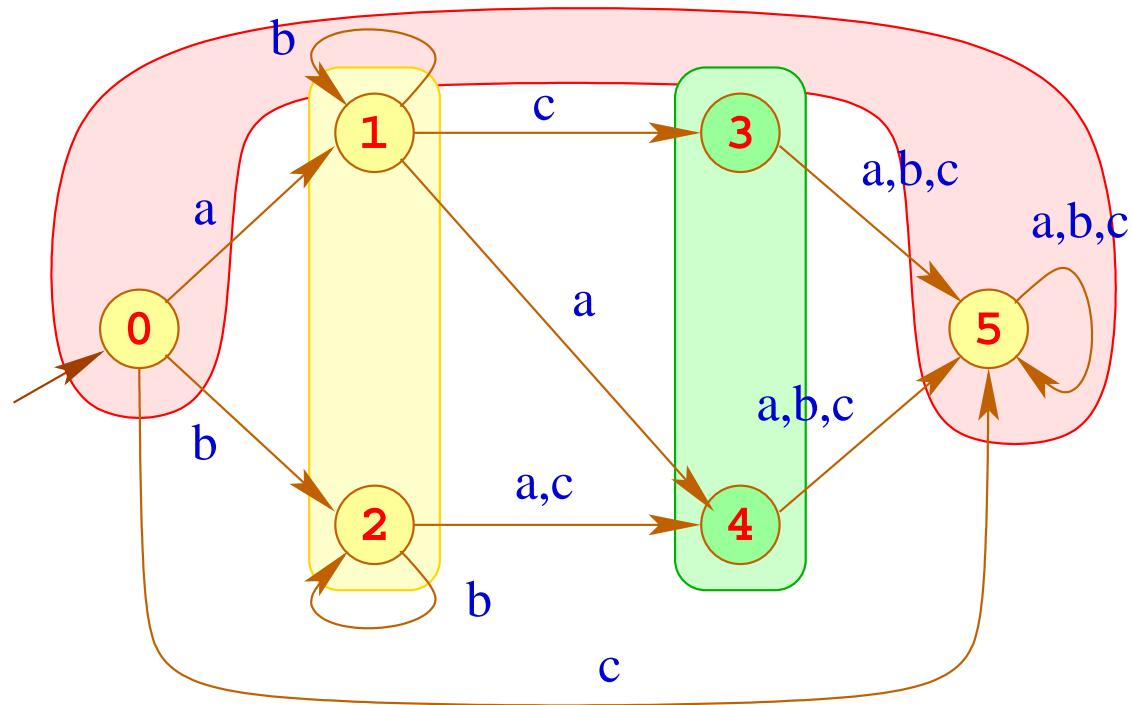
Beispiel:



Beispiel:



Beispiel:



Beispiel:

