1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

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

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

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

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):

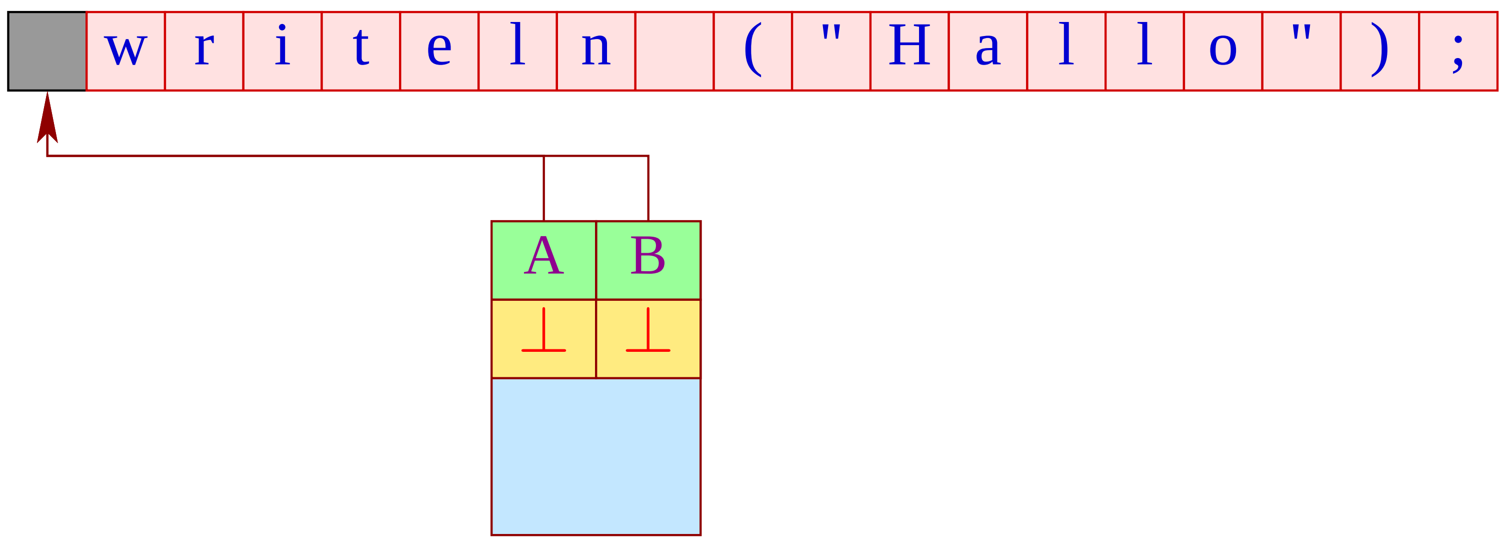
- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.

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

A	B

Idee (Fortsetzung):

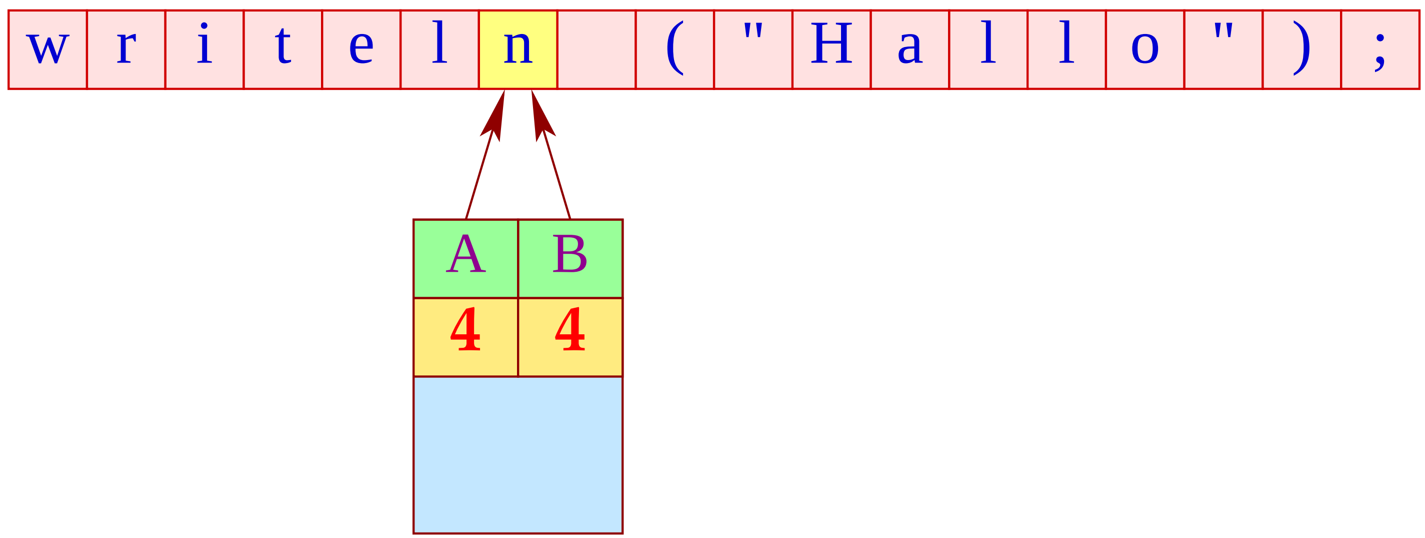
- Der Scanner verwaltet zwei Zeiger $\langle A, B\rangle$ und die zugehörigen Zustände $\langle q_A, q_B\rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.



Idee (Fortsetzung):

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

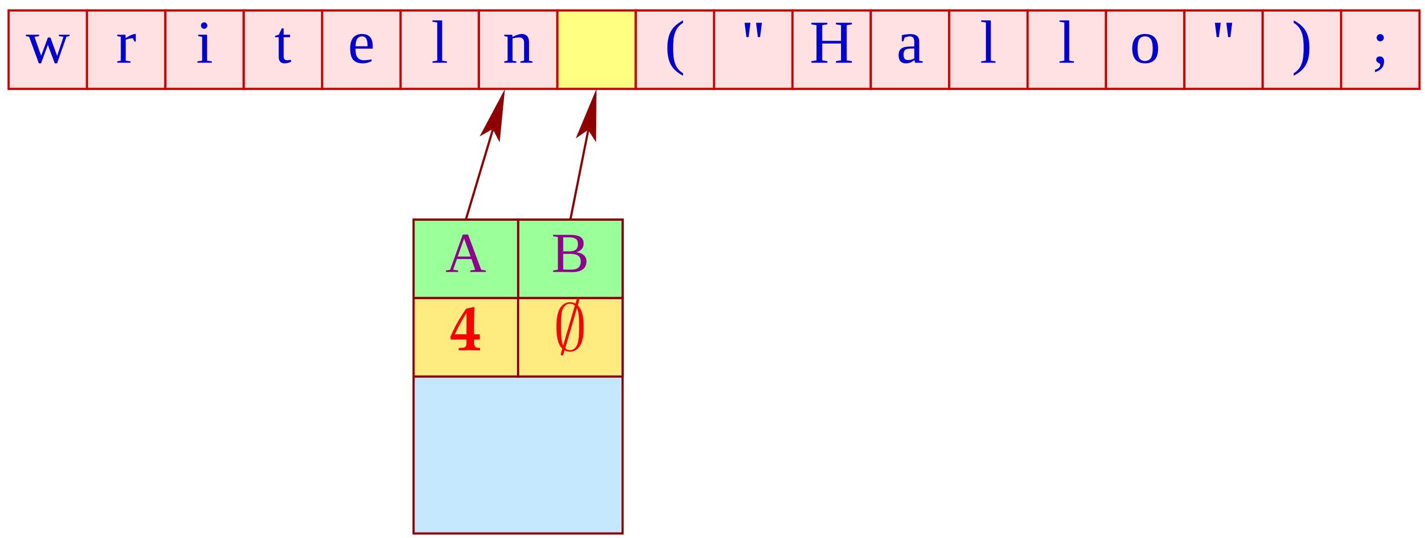
$$\begin{array}{lllllll} B & := & A; & \quad & A & := & \perp; \\ q_B & := & q_0; & \quad & q_A & := & \perp \end{array}$$



Idee (Fortsetzung):

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

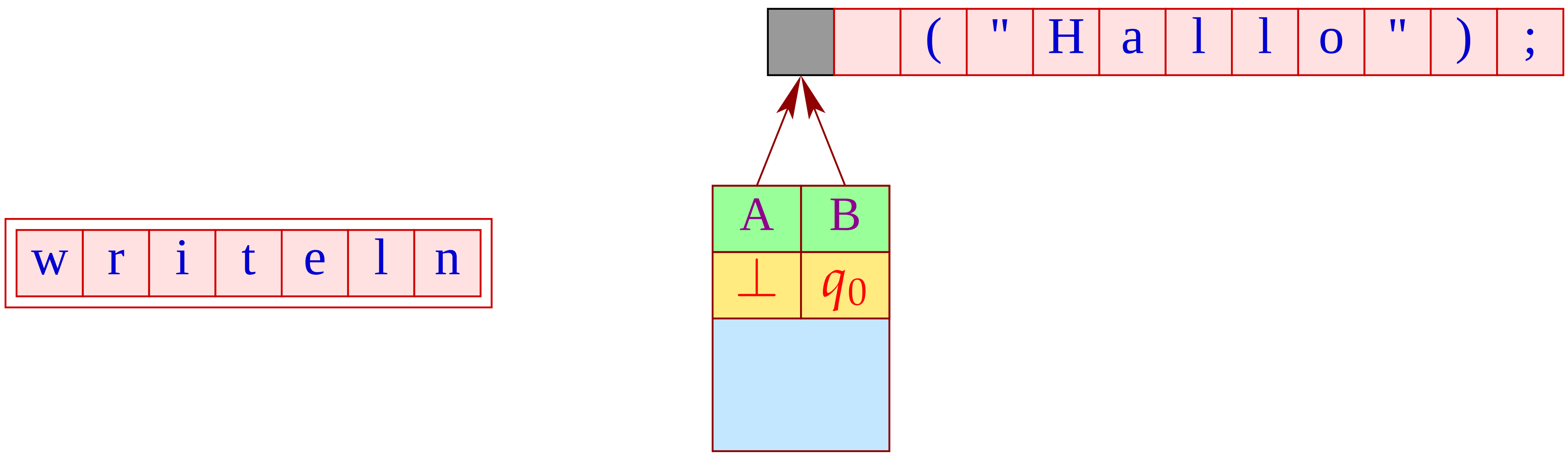
$$\begin{array}{llllll} B & := & A; & A & := & \bot; \\ q_B & := & q_0; & q_A & := & \bot \end{array}$$



Idee (Fortsetzung):

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

$$\begin{array}{lllllll} B & := & A; & \quad & A & := & \perp; \\ q_B & := & q_0; & \quad & q_A & := & \perp \end{array}$$



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:

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

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

... im Beispiel:

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

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:

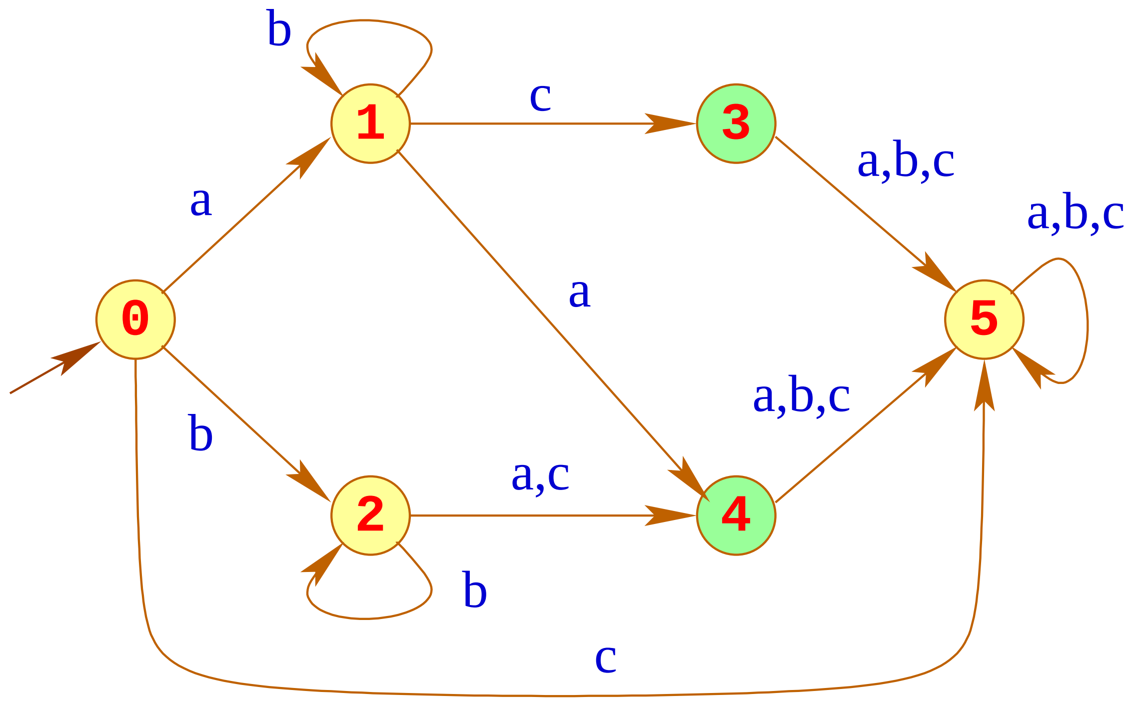
- Wir nehmen an, maximal viel sei äquivalent :-)

  Wir starten mit der Partition:

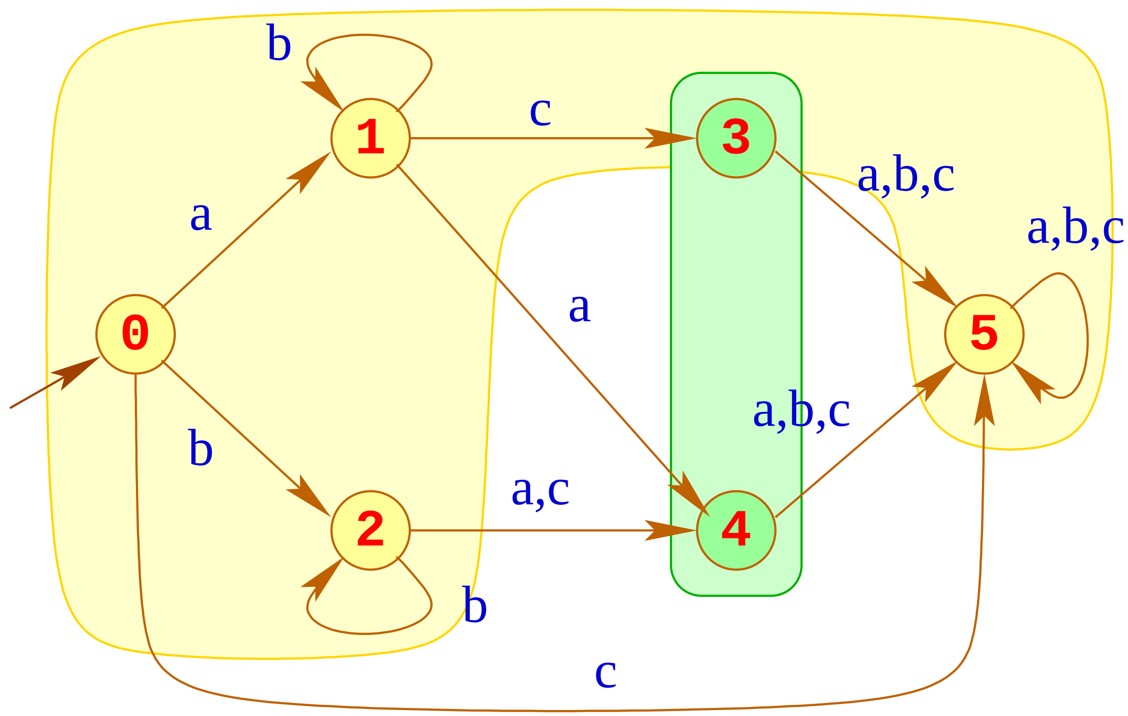
$$\overline{Q} = \{r^{-1}(i) \neq \emptyset \mid i \in \mathbb{N}\}$$

- 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 ...

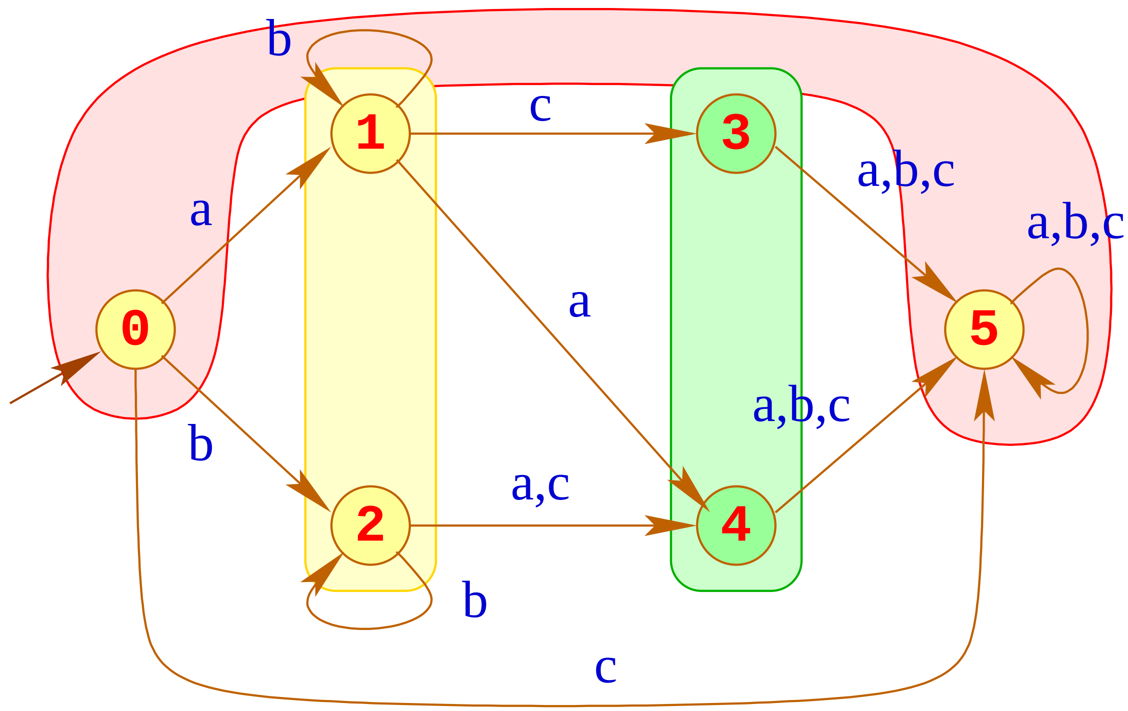
Beispiel:



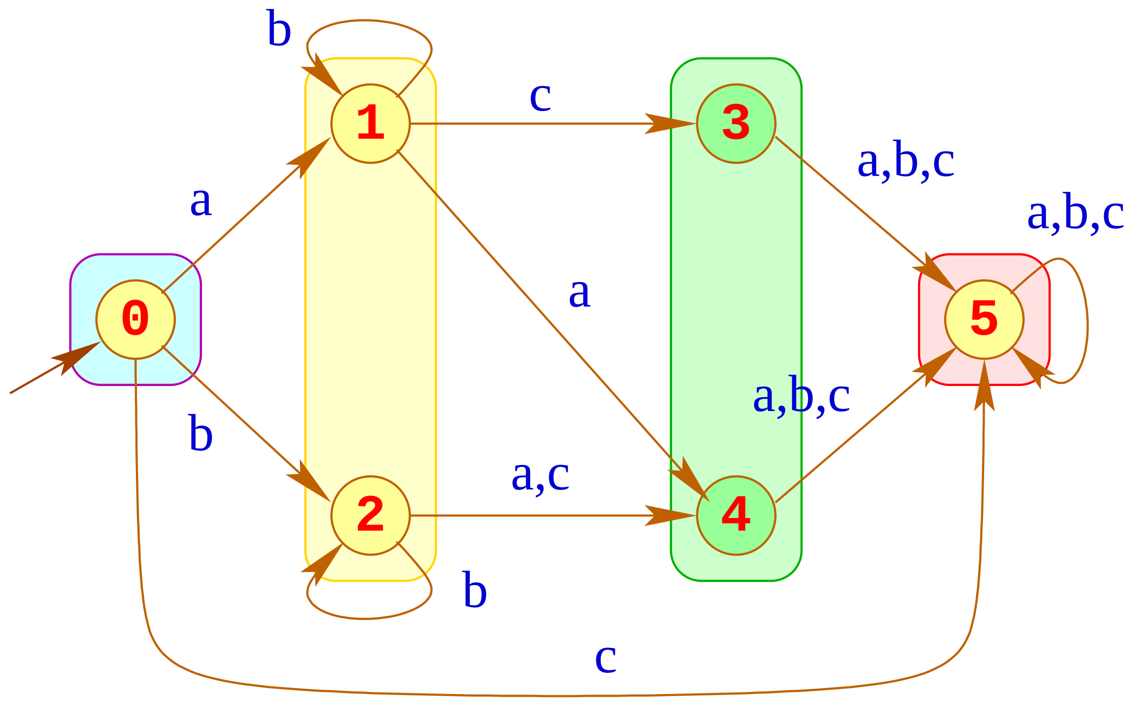
Beispiel:



Beispiel:



Beispiel:



Bemerkungen:

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

$$
\begin{array}{llll}
(1) & p_1, p_2 \in \bar{q} & \Longrightarrow & r(p_1) = r(p_2) \\
(2) & p_1, p_2 \in \bar{q} & \Longrightarrow & \delta(p_1, a), \delta(p_2, a) \quad \text{gehören zur gleichen Klasse}
\end{array}
$$

- 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:

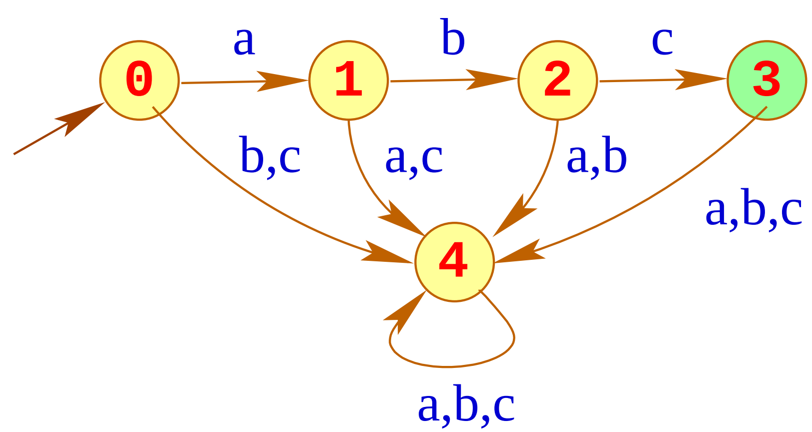
```
le = [a-zA-Z_\$]
ledi = [a-zA-Z_\$0-9]
Id = {le} {ledi}*
```

- 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:



... die zugehörige Tabelle (transponiert):

	0	1	2	3	4
a	1	4	4	4	4
b	4	2	4	4	4
c	4	4	3	4	4

Beobachtung:

- Viele Einträge in der Tabelle sind gleich einem Wert Default (hier: `4`)
- Diesen Wert brauchen wir nicht zu repräsentieren :-)

... die zugehörige Tabelle (transponiert):

	0	1	2	3	4
a	1				
b		2			
c			3		

Beobachtung:

- Viele Einträge in der Tabelle sind gleich einem Wert Default (hier: 4)
- Diesen Wert brauchen wir nicht zu repräsentieren :-)
- Dann legen wir einfach mehrere (transponierte) Spalten übereinander :-))

... im Beispiel:

	0	1	2
A	1	2	3
valid	a	b	c

- Feld valid teilt mit, für welches Element aus Σ der Eintrag gilt :-)
- Achtung: I.a. werden die Spalten nicht so perfekt übereinander passen!

  Dann verschieben wir sie so lange, bis die jeweils nächste in die bisherigen Löcher hineinpasst.
- Darum müssen wir ein zusätzliches Feld displacement verwalten, in dem wir uns die Verschiebung merken ;-)

Ein Feld-Zugriff $\delta(j, a)$ wird dann so realisiert:

$$\begin{array}{lll} \delta(j, a) & = & \textsf{let} \quad d \quad = \quad \textsf{displacement}[a] \\ & & \textsf{in} \;\; \textsf{if} \;\; (\textsf{valid}[d + j] \equiv a) \\ & & \qquad\quad \textsf{then} \;\; \textsf{A}[d + j] \\ & & \qquad\quad \textsf{else} \;\; \textsf{Default} \\ & & \textsf{end} \end{array}$$

Ein Feld-Zugriff $\delta(j, a)$ wird dann so realisiert:

$$
\begin{array}{lll}
\delta(j,a) & = & \mathsf{let}\quad d \quad = \quad \mathsf{displacement}[a] \\
 & & \mathsf{in}\ \ \mathsf{if}\ \ (\mathsf{valid}[d+j] \equiv a) \\
 & & \qquad\quad \mathsf{then}\ \ \mathsf{A}[d+j] \\
 & & \qquad\quad \mathsf{else}\ \ \mathsf{Default} \\
 & & \mathsf{end}
\end{array}
$$

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.