“Theorie der Informatik” (CS206) Sprache, Grammatik, deterministische und nicht-determ. finite Automaten

13. März 2013

Proff Malte Helmert und Christian Tschudin
Departement Mathematik und Informatik, Universität Basel

Wiederholung/Einstieg

1. Was ist die Menge SAT?
2. Was ist eine Sprache?
3. Was ist eine Grammatik?

Grammatik – Definition

Definition: Eine Grammatik ist ein Vier-Tupel (Σ, V, P, S), wobei

- Σ ein endliches Alphabet.
- V eine endliche Menge von Variablen-Symbolen, $\Sigma \cap V = \emptyset$ (auch Nicht-Terminale genannt)
- P eine endliche Menge von Produktionsregeln ("rewriting rules") der Form $(\Sigma \cup V)^* \rightarrow (\Sigma \cup V)^*$
  wobei der linke Teil mind. ein Nichtterminal enthalten muss.
  Beispiel: `T --> F * F`
- Startsymbol (Startvariable) $S \in V$

Ableitung $\Rightarrow$

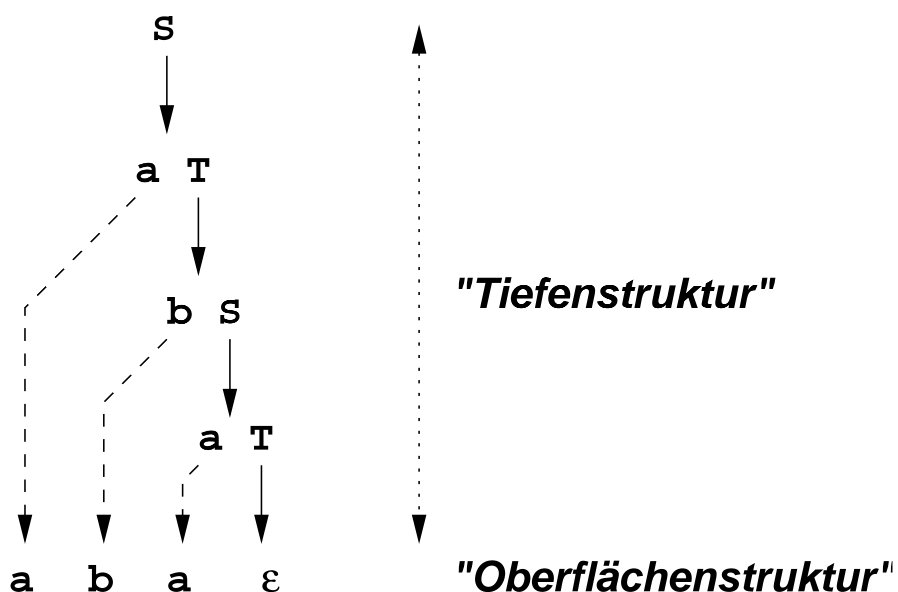
- Ableitungsschritt = Anwendung einer Produktionsregel
- Linke Seite muss "passen", i.A.
  Wortumwandlung $xyz \Rightarrow xy'z$, wobei $w, y, z \in (V \cup \Sigma)^*$
  und $y \rightarrow y'$ eine Regel der Grammatik
- Ausgehend vom Startsymbol lassen sich neue Wörter erzeugen (generative Grammatik):
  $S \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \; \ldots \Rightarrow w_n$
  $S \Rightarrow^* w_n$ (Kleen-Stern)
- Wenn mehrere Ableitungen möglich sind, bleibt die Wahl unbestimmt. Es wird nur gesagt, dass w_n abgeleitet werden kann.

Grammatik – Beispiel

```
S --> a T
T --> b S
T --> ε
```

Mögliche Ableitungen: `a, aba, ababa, ...`

Grammatik – Ableitungsbaum (Beispiel)

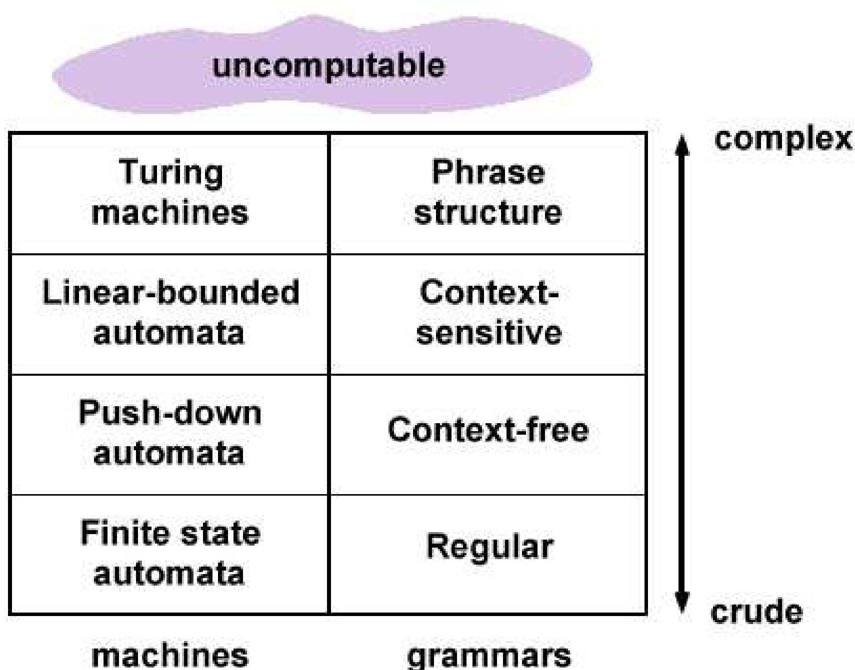


Chomsky–Hierarchie

In Abhängigkeit der Form der Regeln können die durch Gramatiken definierte Sprachen in einfache Klassen eingeteilt werden:

- Sprachen vom Typ 0 – keinerlei Einschränkungen
- Sprachen vom Typ 1 – “**kontextsensitiv**”, falls:
  Für alle Regeln $w_1 \to w_2$ gilt $|w_1| \leq |w_2|$
- Sprachen vom Typ 2 – “**kontextfrei**”, falls:
  wie Typ 1, wobei zusätzlich für jede Regel gilt: $w_1 \in V$
- Sprachen vom Typ 3 – “**regulär**” falls:
  wie Typ 2, wobei zusätzlich gilt $w_2 \in \ \Sigma \cup \Sigma V$

Chomsky-Hierarchie, Tabelle 1



Chomsky-Hierarchie, Tabelle 2

Grammatikregeln haben die Form $w_1 \rightarrow w_2$

Typ	Auflage Regel	Auflage linke Seite	Auflage rechte Seite
0	-	-	-
1	$\lvert w_1 \rvert \leq \lvert w_2 \rvert$	-	-
2	$\lvert w_1 \rvert \leq \lvert w_2 \rvert$	$w_1 \in V$	-
3	$\lvert w_1 \rvert \leq \lvert w_2 \rvert$	$w_1 \in V$	$w_2 \in \Sigma \cup \Sigma V$

Backus-Naur-Form (BNF)

Kompakte Schreibweise (mehrere Regeln auf einer Zeile)

- $A \rightarrow B_1 \,|B_2\,| \ldots$ steht für:
  $A \rightarrow B_1$
  $A \rightarrow B_2$
  ...
- $A \rightarrow x\,[y]\,z$ steht für:
  $A \rightarrow x\,z$
  $A \rightarrow x\,y\,z$
  ("Option")

Backus-Naur-Form (Forts.)

- $A \rightarrow x\,\{y\}\,z$ steht für
  $A \rightarrow xBz$
  $B \rightarrow \epsilon$
  $B \rightarrow yB$
  ("Wiederholung", inklusive 0 mal)

Präzisierung: wenn `[ ]` und $\{\}$ eingesetzt werden, ist die Grammatik in "erweiterter BNF"-Form dargestellt (EBNF)

Varianten:
"::=" statt "$\rightarrow$", "`<name>?`" statt "`[<name>]`"

Verwendung der EBNF in der Informatik

Die Grammatik von Programmiersprachen wird heute in EBFN angegeben. Auszug für Java:

```
...
<method declaration> ::= <method header> <method body>
<method header> ::= <method modifiers>? <result type> <method declarator>
<result type> ::= <type> | void
<method modifiers> ::= <method modifier> | <method modifiers> <method mod
<method modifier> ::= public | protected | private | static | abstract | :
<method declarator> ::= <identifier> ( <formal parameter list>? )
...
```

Beachte: `<result type>` ist ein Nicht-Terminal
`public` ist ein Symbol (des Alphabets)
| ist ein Metasymbol der Grammatikdefinition

Grammatik-Entwurf – Beispiel

Aendere die Beispiel-Grammatik

```
S --> a T
T --> b S
T --> ϵ
```

so ab, dass jedes Wort mit `b` aufhört.

(d.h. ϵ, `ab`, `abab`, `...`)

Wie sieht diese neue Grammatik aus?

Grammatik-Entwurf – Antwort

Aendere die Beispiel-Grammatik

```
S --> a T
T --> b S
T --> ϵ
```

so ab, dass jedes Wort mit `b` aufhört.

(d.h. ϵ, `ab`, `abab`, `...`

```
S --> a T | ϵ
T --> b S
```

oder kompakter: `S --> { a b }`

Wortproblem: $w \in \Sigma^*$ oder $w \notin \Sigma^*$?

- **Einscheidungsproblem**
  d.h. gebe ein *allgemeines* Verfahren/Algorithmus an, um einen Entscheid zu treffen. Input: Grammatik!
- Auf Sprachen angewandt:
  Kann für jede Sprache entschieden werden, ob ein Wort ein Element der Sprache ist oder nicht? (i.A. nein)
- Sprachklassen:
  Kann für jede Sprache *vom Typ N* entschieden werden, ob das Wort in der Sprache ist oder nicht?
  (dies stimmt für $N > 0$)

Entscheidbares Wortproblem

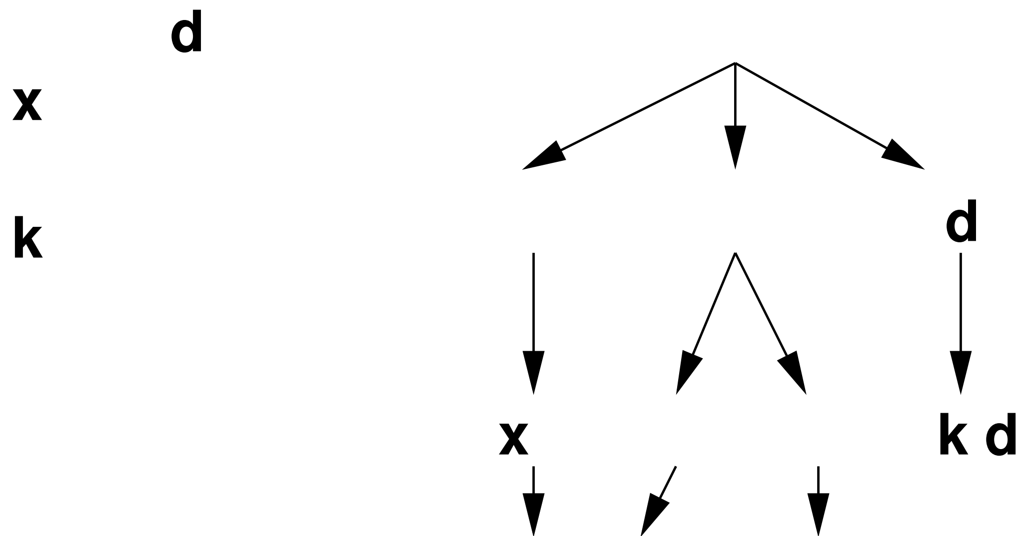
Für Sprachen mit Grammatik G vom Typ 1 (und damit auch 2 und 3) ist entscheidbar, ob für ein gegebenes Wort $w \in \Sigma^*$ gilt
$w \in L(G)$ oder $w \notin L(G)$

Beweisidee: vollständige Aufzählung (Enumerierung)

- Generiere alle Ableitung von S bis zur Länge $|w|$ und vergleiche jeweils mit w
- Warum ist dieses Verfahren gültig?
  Kann immer beendet werden.

Siehe Ableitungsbaum

Ableitungsbaum



Aufbau der Menge L(G)

Menge T_m enthält alle Worte w mit $|w| \leq m$
Rezept: baue die Sprache auf, beginnend mit dem “Ersetzen” von S

- $T_0 = \{S\}$
- $T_{m+1} = Abl(T_m)$
- $Abl(X) := X \cup \{w \in (V \cup \Sigma)^* \; mit \; |w| \leq n, \exists w' \in X : w' \Rightarrow w\}$

Entscheidungsverfahren (Pseudo-Code)

```
BOOL wordIsInLanguage(G, x):
BEGIN
   VAR Set R, T; INT n;
   T = {S}, n = len(x);
   REPEAT
     R = T;
     T = Abl(G, T, n);
   UNTIL (x in T) OR (R == T)
   RETURN x in T;
END
```


Entscheidungsverfahren (Forts): `Abl(G, T, n)`

```
VAR Set R;
R = T;
FOREACH w in T DO
   FOREACH p in G.Prod DO
      v = p(w);   // Regel p auf w anwenden
      IF valid(v) AND (len(v) <= n) THEN
         R = R union { v };
      FI
   OD
OD
RETURN R
```

Laufzeit dieses Verfahrens für Wort der Länge n

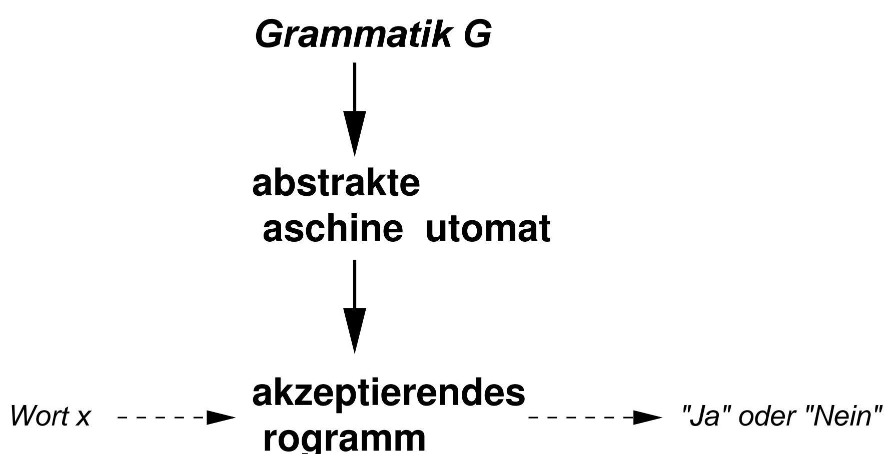
Komplexität (dieser) Worterkennung ist $O(|\Sigma|^n)$

d.h. exponentiell –
doppelt so langes Wort gibt quadratische Laufzeit!

Good News:

- Reguläre Ausdrücke (Chomsky Typ 3)
  entscheidbar in $O(n)$
  d.h. Wortproblem in linearer Zeit entscheidbar!

Schema für einen “Akzeptor” einer Sprache



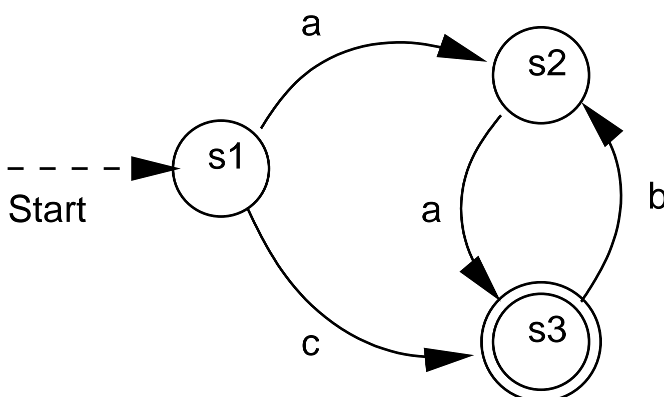
Behauptungen:
– Typ 3-Sprachen werden durch “endliche Automaten” erkannt.
– “Endliche Automaten” erkennen genau die Typ 3-Sprachen.

Definition Endlicher Automat

DFA = deterministischer finiter Automat $(\Sigma, Z, z_0, E, \delta)$

Σ	endliches Alphabet
Z	endliche Menge von Zuständen, $Z \cap \Sigma = \{\,\}$
z_0	Startzustand $\in Z$
E	Endzustandsmenge $\subseteq Z$
$\delta : Z \times \Sigma \to Z$	Transitionsfunktion

Zustandsgraph



δ	a	b	c
s1	s2	–	s3
s2	s3	–	–
s3	–	s2	–

Als Generator (Eisenbahnfahren):

a, aa, aab, aaba, ...

c, cb, cba, ...

Bezug DFA und Typ 3-Sprachen

Ansatz: Zustände = Variabelsymbole, Kanten mit Terminalen beschr.

Alle DFA-Regeln sollen die Form haben:

$v \to w$

wobei $v \in V, w \in \Sigma \cup \Sigma V$ (V: Menge der (Zustands) Variablen)

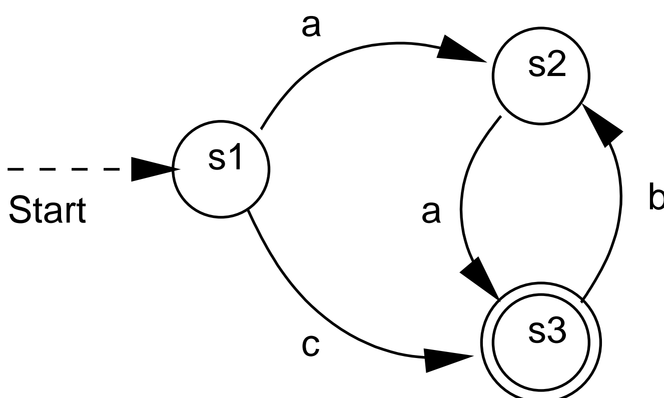
Gemäss Definition von Chomsky ist das vom Typ 3:

$v \to {}'\mathrm{a}'$, oder

$v \to {}'\mathrm{a}'\, u, u \in V$

Bei jedem Ableitungsschritt wird ein Zeichen des Wortes "erzeugt/erkannt".

Bezug DFA und Typ 3-Sprachen – Beispiel



δ	a	b	c
s1	s2	–	s3
s2	s3	–	–
s3	–	s2	–

```
S  -> S1          S2 -> a S3
S1 -> a S2        S3 -> b S2
S1 -> c S3        S3 ->
```

Bemerkung: Start-"Zustand" S, und S3 ein Endzustand.

Konstruktion: “aus DFA eine Grammatik bauen”

Behauptung:
Jede durch einen DFA M erkennbare Sprache ist regulär (Typ 3).
Ansatz: Konstruiere zu M eine Grammatik G, überprüfe die Form der Regeln, beweise die Aequivalenz.

- Für jedes $\delta(z_i, a) = z_j$ definiere eine Regel
  $Z_i \rightarrow a\, Z_j$
- Für jeden Zustand $z_k = \delta(z_l, a) \in E$ zusätzlich die Regel
  $Z_l \rightarrow a$
- Anfangszustand Z_0 heisst S (bzw: Regel $S \rightarrow Z_0$)

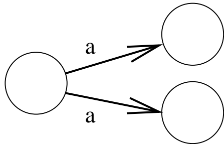
Beweis: Aequivalenz zeigen

M **akzeptiert** $x = a_1 \ldots\ a_n \in \Sigma^* \Leftrightarrow$

- $\exists$ Folge von Zuständen $z_0, z_1, \ldots$ mit
  $z_0 = S$, $z_n \in E$, $\delta(z_{i-1}, a_i) = z_i\ \ \forall i = 1 \ldots n$
  $\Leftrightarrow$
- $\exists$ Folge von Variablen $Z_0 \ldots Z_n$ mit
  $Z_0 = S, Z_0 \Rightarrow a_1 Z_1 \Rightarrow \ldots\quad a_1 \ldots a_{n-1} Z_n \Rightarrow a_1 \ldots a_n$
  $\Leftrightarrow$
- $x \in L(G)$, d.h. ist **ableitbar** ($S \Rightarrow^* w$)

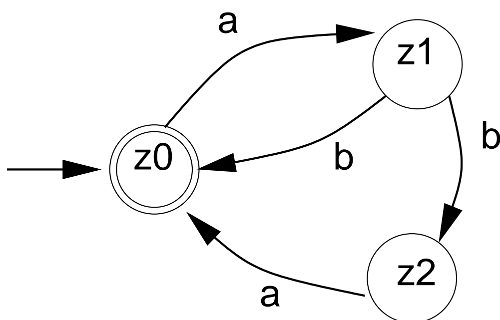
Nicht-Deterministischer Finiter Automat (NFA)

- Erweiterung von DFA:
  der Akzeptor kann, für gegebenes Zeichen, zwischen mehreren verschiedenen Kanten im Graph wählen.Graphisch dargestellt:



- Formal: **neue Transitionsfunktion** δ
  $$\delta : Z \times \Sigma \rightarrow P(Z)$$
  wobei $P(Z)$ die Potenzmenge von Z ist.
  Potenzmenge (power set) = "Menge aller Mengen über Z").
  Beispiel: $Z = \{a, b\} \Rightarrow P(Z) = \{\,\epsilon, \{a\}, \{b\}, \{a, b\}\,\}$

Beispiel NFA



	a	b
z0	{z1}	–
z1	–	{z0,z2}
z2	{z0}	–

Generierte/akzeptierte Sprache: $L = \{ab,\ aba\}^*$

NFA in einen DFA verwandeln

Grundidee: aus NFA neuen Automaten konstruieren, dessen Zustände Mengen (aus ursprünglichen Zuständen) sind.

Konstruktion:

- $\Sigma' := \Sigma$
- $Z' := P(Z)$, und $S' = \{z_0\}$
- $E' := \{Y \subseteq Z' \mid Y \cap E \neq \{\}\}$
- $\delta'(Z, a) := \bigcup \delta(z, a)$ über alle $z \in Z$

NFA in einen DFA verwandeln – Beispiel

$\Sigma = \{a, b\}$ $S = z_0$, $E = \{z_0\}$

	a	b
$\{z_0\}$	$\{z_1\}$	–
$\{z_0, z_1\}$	$\{z_1\}$	$\{z_0, z_2\}$
$\{z_0, z_1, z_2\}$	$\{z_0, z_1\}$	$\{z_0, z_2\}$
$\{z_1\}$	–	$\{z_0, z_2\}$
$\{z_1, z_2\}$	$\{z_0\}$	$\{z_0, z_2\}$
$\{z_2\}$	$\{z_0\}$	–
$\{z_0, z_2\}$	$\{z_1, z_0\}$	–

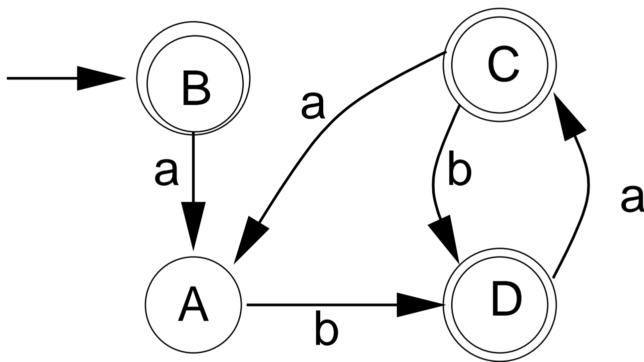
Neuer Automat:

$S' = \{z_0\}$, $E' = \{\{z_0\}\}$

Neue Zustände A-Z:

$A := \{z_1\}$
$B := \{z_0\}$
$C := \{z_0, z_1\}$
$D := \{z_0, z_2\}$

NFA in einen DFA verwandeln – Beispiel (Forts)



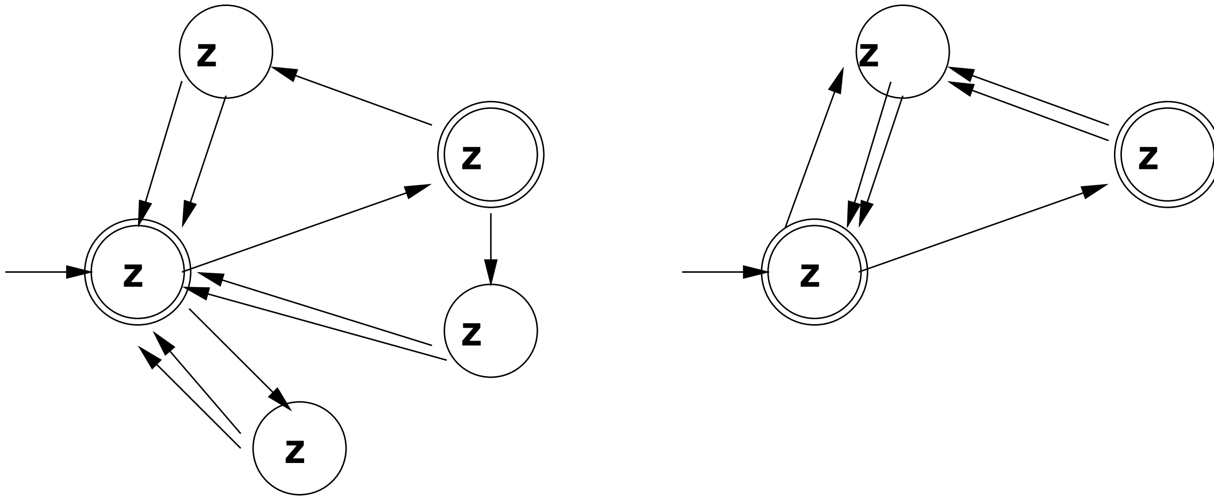
{z0} –> B
{z1} –> A
{z0,z1} –> C
{z0,z2} –> D

DFA-Minimierung

Verwandlung eines NFA in DFA mit Potenzmengen-Konstruktion:
Gewisse Zustände können weggelassen werden:

1. Sackgasse
   Zustand q ist Sackgasse, wenn $\neg\exists w \in \Sigma^* : \delta(q, w) \in E$
2. unerreichbare Zustände
   Zustand q unerreichbar, wenn $\neg\exists w \in \Sigma^* : \delta(z_0, w) = q$
3. Aequivalenzklassen
   Zustände sind äquivalent, wenn von ihnen aus die gleichen Wörter akzeptiert bzw nicht akzeptiert werden.

DFA-Minimierung: Beispiel Aequivalenzklassen



Von z1, z2 und z4 aus werden die gleichen Wörter akzeptiert.

Zusammenfassung

- Formale Sprachen:
  Alphabet, Wort, Kleen-Stern, Σ^+, Sprache
- Grammatik:
  Produktionsregeln, Startvariable, EBNF,
  Chomsky-Hierarchie, kontextfrei und -sensitiv
- Ableitung, Wortproblem
- “Akzeptor”
- Deterministischer finiter Automat (DFA)
- Nicht-deterministischer Finiter Automat (NFA),
  Aequivalenz zu einem DFA, Minimierung