“Theorie der Informatik” (CS206)
Nicht-deterministische endl. Automaten reguläre Sprachen, Pumping Lemma

18. März 2018

Proff Malte Helmert und Christian Tschudin

Departement Mathematik und Informatik, Universität Basel

Wiederholung/Einstieg

1. Wie lautet das Wortproblem?
2. Zusammenhang Typ-3 Sprachen und endliche Automaten?
3. Was ist der Unterschied zw DFA und NFA?

Definition Endlicher Automat (Repetition)

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

“Berechnung durch DFA” (=Akzeptor, determinist. Fall)

Sei M ein endlicher Automat $(\Sigma, Z, z_0, E, \delta)$ und w ein Wort über dem Alphabet Σ

- M **akzeptiert** w falls es eine Sequenz von Zuständen $r_0, \ldots r_n$ gibt in Z so dass
  - $r_0 = z_0$
  - $\delta(r_i, w[i+1]) = r_{i+1}$ für alle $i = 0, \ldots, n-1$
  - $r_n \in E$
- M **erkennt** die Sprache A falls $A = \{w | M$ akzeptiert $w\}$
- Eine Sprache ist **regulär** falls ein endlicher Automat existiert der diese Sprache erkennt.

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:

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

Neue Zustände A-Z:

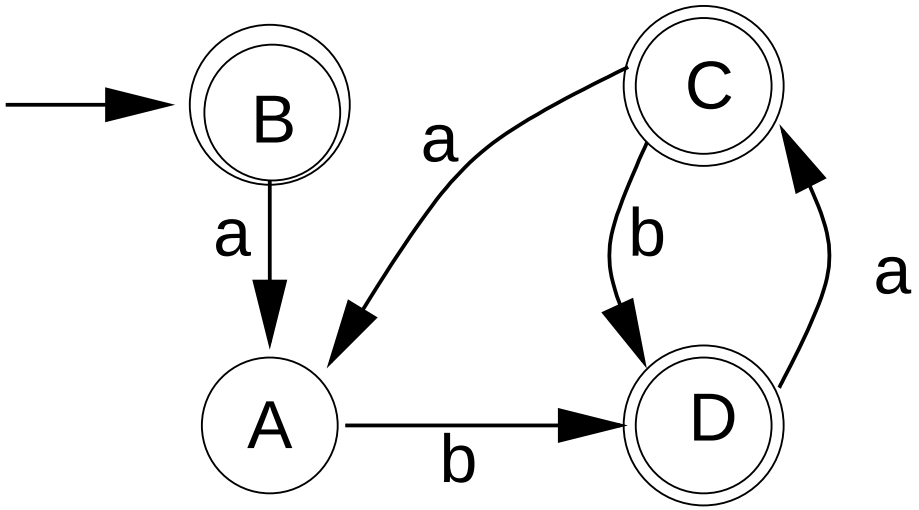
$A := \{z_1\}$

$B := \{z_0\}$

$C := \{z_0, z_1\}$

$D := \{z_0, z_2\}$

NFA in einen DFA verwandeln – Beispiel (Forts)



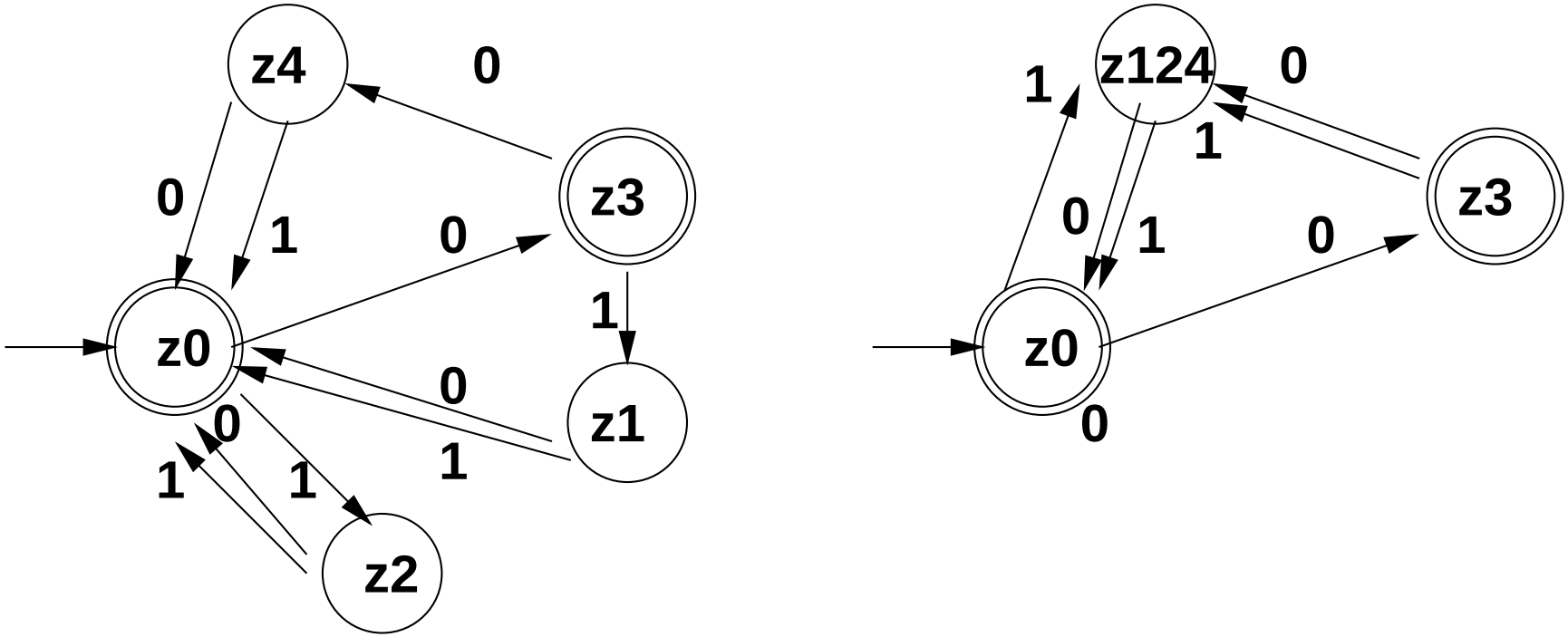
{z0} –> B
{z1} –> A
{z0,z1} –> C
{z0,z2} –> D

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.

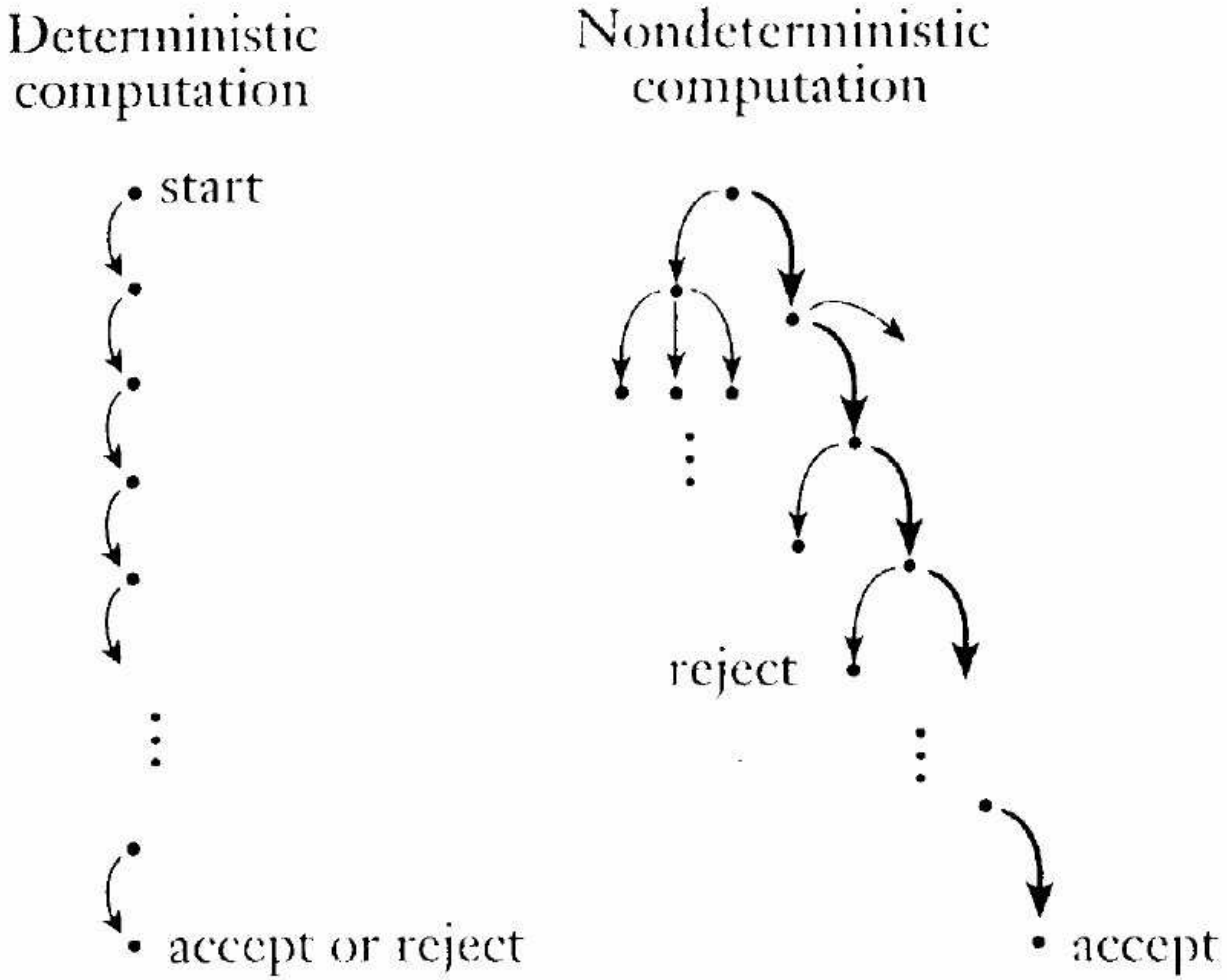
Akzeptor (nicht-deterministischer Fall)

Sei M ein nicht-deterministischer endlicher Automat $(\Sigma, Z, z_0, E, \delta)$ und w ein Wort über dem Alphabet Σ

- M **akzeptiert** w falls es eine Sequenz von Zuständen $r_0, \ldots r_n$ gibt in Z so dass
  - $r_0 = z_0$
  - $r_{i+1} \in \delta(r_i, w[i+1])$ für alle $i = 0, \ldots, n-1$
  - $r_n \in E$

Bemerkung:
Diese Definition ist bis auf die Aenderung wegen der unterschiedlichen Transitionsfunktion (die nun eine Menge statt einen Folgezustand zurückgibt) identisch mit dem Akzeptieren durch einen deterministischen Automaten.

Deterministisch vs Nicht-determ. endliche Automaten



Typ-3-Sprachen und reguläre Ausdrücke

Enger Zusammenhang zwischen **regulären Ausdrücken** und Sprachen vom Typ-3

- Reguläre Ausdrücke definieren ein “Muster” für Wörter, definieren eine Sprache.
- Regulärer Ausdruck kann in einen DFA verwandeln werden (siehe nachfolgende Slides)
- `egrep` macht dies intern, um die Mustersuche zu beschleunigen.

Reguläre Ausdrücke (RA, “regex”)

Induktive Definition eines RA

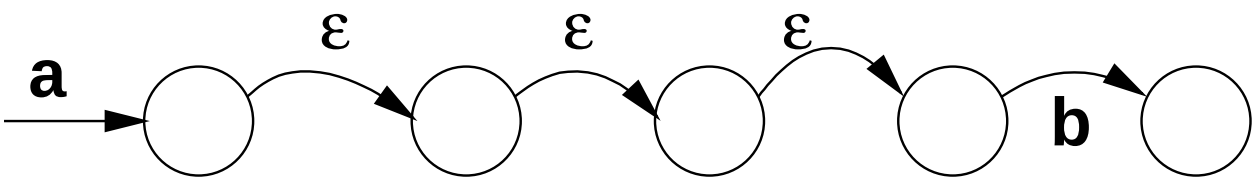
1. ϵ ist der RA für die Sprache $\{\epsilon\}$
2. für $a \in \Sigma$ ist a ein RA (für die Sprache $\{a\}$)
3. sei r, s jeweils ein RA. Dann ist

$(r)\|(s)$	$L(r) \cup L(s)$	
$(r)(s)$	$L(r)L(s)$	(Konkatenation)
$(r)^*$	$(L(r))^*$	
(r)	$L(r)$	(Klammern)

Bau eines DFA, der die Sprache eines RA akzeptiert

Hilfsmittel: ϵ-Transition (leeres Wort)

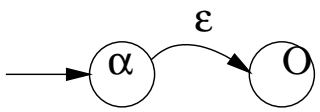
- kann überall eingesetzt werden
  $a\epsilon\epsilon\epsilon b \sim ab$
- wird auch “stumme Transition” genannt



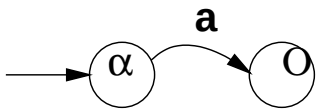
Bau eines DFA zu einem RA (Forts I)

Plan: RA $\rightarrow$ NFA $\rightarrow$ DFA

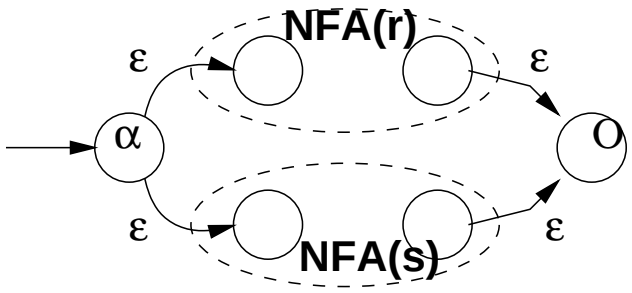
ϵ wird zu



$a \in \Sigma$ wird zu

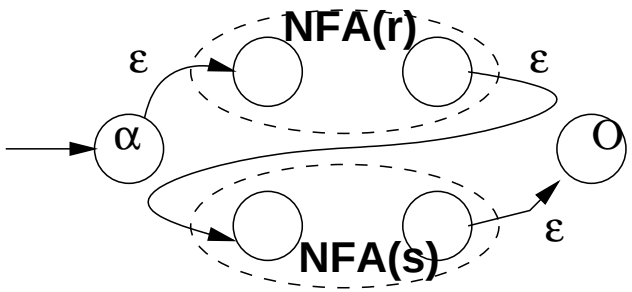


$(r)|(s)$ wird zu

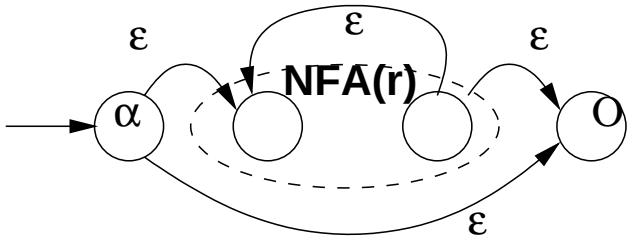


Bau eines DFA zu einem RA (Forts I)

$(r)(s)$ wird zu



$(r)^*$ wird zu



Gesamt-NFA: Bilden des "ϵ-Abschlusses", DFA, Minimierung

Elimination der ϵ

Die stillen Transitionen (ϵ) sollten “ignoriert” werden. Wie?

- Ausgehend von einem Zustand z:
  welche andere Zustände kann ich durch (reine) ϵ-Schritte von z aus erreichen?
- ϵ-“Abschluss” genannt.
- Um neues δ (für einen NFA ohne Epsilons) zu finden:
  bei jedem Zustand zuerst dessen Epsilon-Abschluss bilden, dann von dieser Menge (!) die möglichen abgehenden Transitionen zeichnen.

ϵ-Abschluss (ϵ-closure)

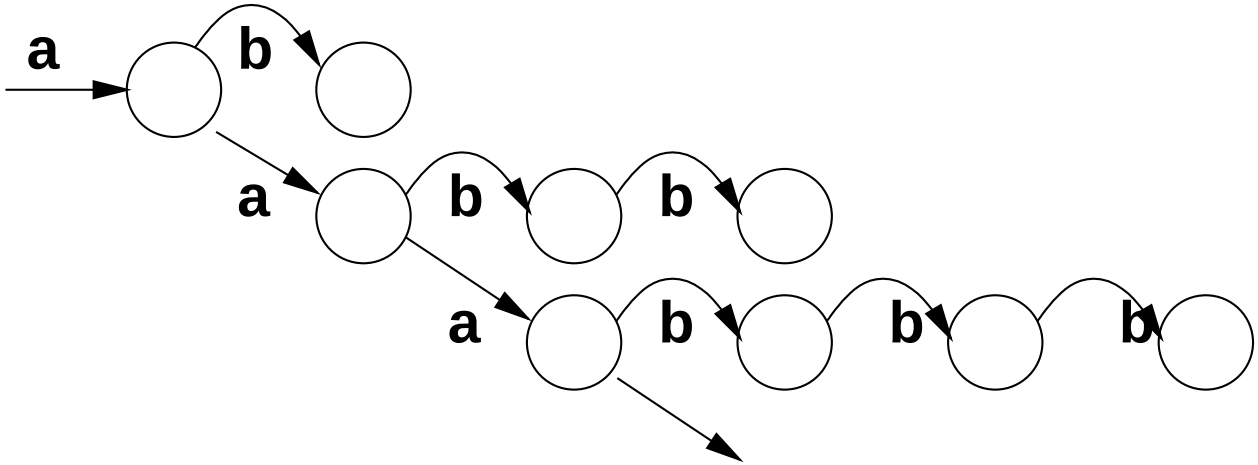
Input: Zustands(teil)menge T; Output: Closure C

```
S = T
C = T
WHILE S <> {} DO
   pick (and remove) any s from S
   FOREACH l in Alphabet plus epsilon: transition(s,l) <> {} DO
      u = transition(s,l)
      IF l == epsilon AND NOT (u IN C) THEN
         C = C + {u}
         S = S + {s}
   OD
OD
```

Kontextfreie Sprachen

Ist $\{a^n b^n | n \geq 1\}$ eine reguläre Sprache?

Versuch:



Pumping-Lemma

Schleifentheorem, Bar-Hill-Theorem

Sei L eine reguläre Sprache. Dann $\exists n \in \mathbb{N}$ mit:
alle Wörter $x \in L$ mit $|x| \geq n$ lassen sich so in die Form $x = u\,v\,w$ ($u, v, w \in \Sigma^*$) zerlegen, dass gilt:

1. $|v| \geq 1$
2. $|u\,v| \leq n$ (d.h., u ist “endlicher Prefix”)
3. $\forall i = 0, 1, \ldots$ gilt: $u\,v^i\,w \in L$ (d.h., v^i ist eine Schleife)

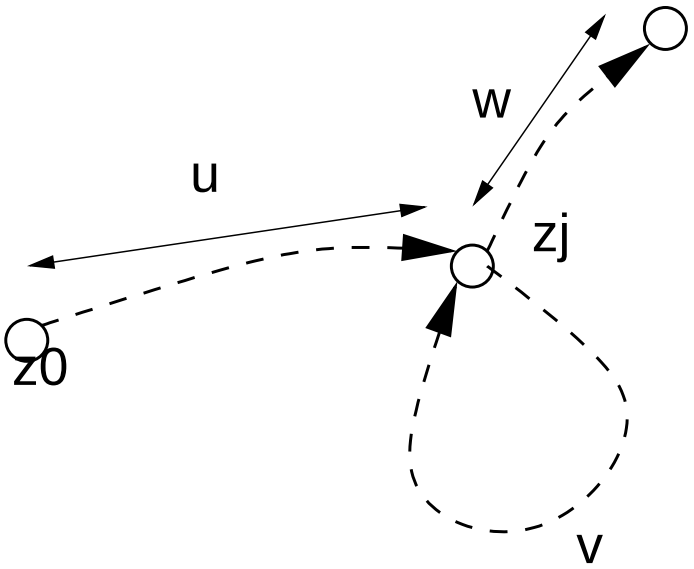
Pumping Lemma, Beweis

Gegeben: reguläre Sprache L
Gesucht: n so dass jedes $x \in L, |x| \geq n$ gewünschte Form hat.
Beweis:

- Da L regulär ist, gibt es einen DFA M, der L akzeptiert.
- Sei $n = |Z|$ die Anzahl Zustände von M,
  und x ein Wort mit $|x| \geq n$.
- Beim Akzeptieren von x durchlaufen wir $|x| + 1$ Zustände,
  d.h. mind. 1 Zustand z_j wird zweimal besucht
- Wähle u so, dass $\delta(z_0, u) = z_j$, v so, dass $\delta(z_j, v) = z_j$,
  und w so, dass $\delta(z_j, w) \in E$.

Pumping Lemma, DFA-Ueberlegung

Bei genügend langem Wort wird es einen Zustand z_j geben, den man zwei- oder mehrmals besucht:

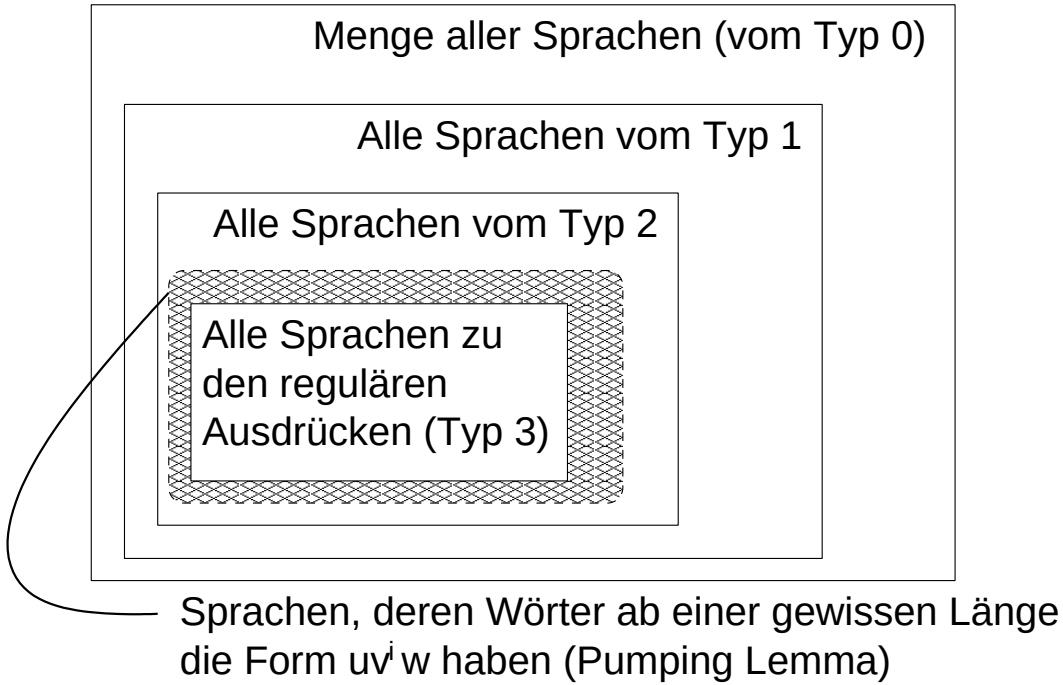


Uebrigens: Wort "uw" wird auch akzeptiert (d.h. v^0)

Pumping Lemma, Benutzung

- Einsatz: um zu zeigen, dass eine Sprache *nicht* regulär ist
- Dazu:
  – suche (konstruiere) ein $x \in L$
  – so dass es keine Zerlegung uv^iw gibt.
- Beachte:
  Es gibt nicht-reguläre Sprachen, für die das Pumping-Lemma auch gilt (d.h. mit Wortstruktur uvw für genügend grosses n).
  Deshalb folgendes Mengendiagramm:

Durch Pumping Lemma-Struktur erfasste Sprachen



Alle RA-Sprachen werden dabei erfasst, es gibt aber auch Typ 2-Sprachen mit der Pumping-Lemma-Eigenschaft.

Kontextfreie Sprachen: Ambiguität

Betrachte die Grammatik

```
Stmt -->   'if'  Expr  'then'  Stmt
         | 'if'  Expr  'then'  Stmt  'else'  Stmt
         | Other
```

und folgende zwei Programme

```
if  E1  then
   if  E2  then
      S1
   else
      S2
```

```
if  E1  then
   if  E2  then
      S1
else
   S2
```

Ambiguität, Umgang

Ambiguität kann behoben werden:

- Regeln einführen, welche Ableitung vorzuziehen ist
  (im Falle von Ambiguität)
- Linksableitung-Regel:
  Versuche, immer die am meisten links stehende Variable der rechten Produktionsseite abzuleiten.
  $\Rightarrow$ mögliches Rekursionsproblem.
- Oder Grammatik umschreiben (falls möglich)

Elimination der Links-Rekursion

- Gegeben: Grammatik mit (unmittelbarer) Links-Rekursion
  $A \to A\alpha_1 | A\alpha_2 | \ldots \ \beta_1 | \beta_2 | \ldots \ | \beta_n$
- Ersetze die A-Produktionen, führe ein A' ein, wie folgt:
  $A \to \beta_1 A' | \beta_2 A' | \ldots \ | \beta_n A'$
  $A' \to \alpha_1 A' | \alpha_2 A' | \ldots \ | \epsilon$
- Beispiel:

```
E --> E "+" T | T
T --> T "*" F | F
F --> "(" E ")" | Identifier
```

- (Weitere Schritte nötig, falls mehrstufige Link-Rekursion)

Links-Faktorisierung

- “Dangling else”-Beispiel: rechte Seiten mit gleichem Anfang:

```
Stmt -->   "if"  Expr  "then"  Stmt
         | "if"  Expr  "then"  Stmt  "else"  Stmt
         | Other
```

- Die “gleichen Anfänge” zusammenfassen, neue Verlängerungsvariable S':

```
Stmt -->   "if"  Expr  "then"  Stmt S'
         | Other
S'   -->   "else"  Expr S  | epsilon
```

- Bemerkung: Ambiguität weiterhin vorhanden, aber schon beim ¨if¨, bzw ¨else¨ vorauszusagen, welche Regel anzuwenden ist.

Parsing Problem

- Reguläre Ausdrücke: jedes eingelesene Symbol bestimmt die nächste Aktion (Wechsel des Zustands)
- Kontext-Frei: Falls die “falsche” Ableitungs-Variante gewählt wurde, muss dies rückgängig gemacht werden, Symbole müssen “zurückgeschrieben” werden.
- Wieviele Zeichen muss man max vorausschauen, um **die** korrekte Ableitungsregel zu bestimmen?
- Sprachen vom Typ **LL(1)**:
  L(inks-nach-rechts-Parsing) L(inks-Ableitung) mit
  max 1 Zeichen zurückschreiben

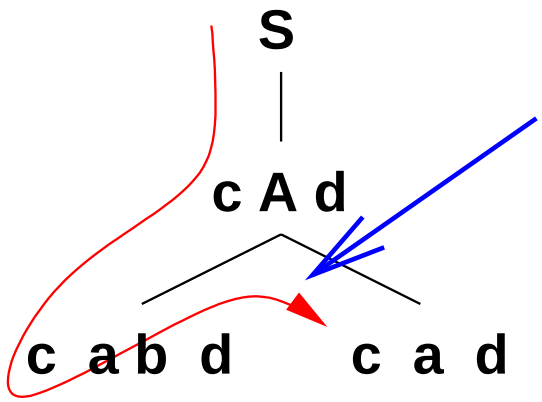
Top-Down Parsing

LL(1) erlaubt Parsing ohne Backtracking

- Beispiel *für* Bedarf an Backtracking am Wort `c a d`

```
S --> c A d
A --> a b | a
```

- Beginne mit Regel S, dann mit Links-Ableitungs-Regel parsen.



Backtracking!
Beim Lesen des nächsten Zeichens (d) wurde erkannt, dass es kein 'b' ist.

Top-Down Parsing ohne Backtrack: Predictive Parsing

Siehe Beispiel arithm. Ausdrücke (Links-Rekursion-Elimination):

```
S  --> E "#"                    T  --> ...
E  --> T  E'                    ...
E' --> "+" T E' | epsilon
```

```
PROC parse() {
  expr();
  t = getToken();
  if (t != "#")
    error();
}
```

```
PROC expr() {
  term();
  exprCont();
}
```

```
PROC exprCont() {
  s = getToken();
  if (s != "+") {
    ungetToken(s);
    return;
  }
  term(); exprCont();
} ...
```

Predictive Parsing – Bewertung

Predictive Parser auch “recursive-descent parser” genannt.

- Wie unterscheidet sich ein solcher Akzeptor
  von den einfachen endlichen Automaten? $\rightarrow$ **Speicherplatz**
- Für DFA: 1 Zustandsvariable
- Für recursive-descent Parser:
  – Rekursion möglich
  – Speicherbedarf hängt vom Eingabewort ab!
  – “Stack” muss genügend gross sein

Formalisierung: Keller/Stapel-Automat mit beliebig grossem Stapel

Kontextfreie Sprache (Wiederholung)

Def: alle Grammatikregeln haben die Form

$V \to w_2, V \in$ Nichtterminale *Kontext von V (linke Seite) ist leer!*

Beispiel von vorher: Sprache $L = \{a^n b^n | n \geq 1\}$

Endlicher Automat kann die Anzahl der Klammern nicht speichern:

```
S --> "(" T ")"
T --> S | epsilon
```

aber mit einem Stapel (Stack) könnte es gehen.

Beispiel natürliche Sprache

<SENTENCE>	→	<NOUN-PHRASE><VERB-PHRASE>
<NOUN-PHRASE>	→	<CMPLX-NOUN>|<CMPLX-NOUN><PREP-PHRASE>
<VERB-PHRASE>	→	<CMPLX-VERB>|<CMPLX-VERB><PREP-PHRASE>
<PREP-PHRASE>	→	<PREP><CMPLX-NOUN>
<CMPLX-NOUN>	→	<ARTICLE><NOUN>
<CMPLX-VERB>	→	<VERB>|<VERB><NOUN-PHRASE>
<ARTICLE>	→	a | the
<NOUN>	→	boy | girl | flower
<VERB>	→	touches | likes | sees
<PREP>	→	with

Beispiel natürliche Sprache (Forts.)

Mit der Grammatik (vorangehendes Slides) kann abgeleitet werden:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Zeichne jeweils den Ableitungsbaum . . .

Chomsky Normalform (CNF)

Definition:
Eine kontextfreie Grammatik ist in **Chomsky-Normalform**,
falls jede der Regeln eine der folgenden Formen hat:

$A \to BC$, oder
$A \to a$, oder
$S \to \epsilon$

wobei
– A, B, C, S sind Variablen (Nicht-Terminale)
– a ist ein Terminalsymbol
– S ist die Startvariable, B und C sind *nicht* die Startvariable

Chomsky Normalform – Theorem

Jede kontextfreie Sprache ist durch eine Grammatik in Chomsky-Normalform erzeugt.

Beispiel

Die Sprache zur kontextfreie Grammatik

$G = (\{A, B\}, \{0, 1, \#\}, R, A\})$

$R = \{A \to 0A1, A \to B, B \to \#\}$

wird auch durch diese Grammatik in CNF erzeugt:

$G' = (\{A, B, C, N, E\}, \{0, 1, \#\}, R, S\})$

$R = \{S \to NC, N \to 0, S \to \#, A \to NC, C \to AE, E \to 1, A \to \#\}$

Chomsky Normalform – Technik

Ansatz: Umschreiben aller Regeln die nicht in CNF sind,
allenfalls neue Variablen einführen. Vier Fälle:

1. Startvariable erscheint in rechter Seite:
   $\Rightarrow$ Führe "Ersatz-S_0" ein und eine Ableitungsregel dazu.

2. Epsilon-Regeln der Form $A \to \epsilon$
   $\Rightarrow$ Falls A in der rechten Seite der Regel auftritt,
   führe eine neue Regel ohne A auf der rechten Seite ein.

3. Einer-Regeln der Form $A \to B$
   $\Rightarrow$ ersetze B direkt durch seine eigene Produktion.

4. Lange und/oder gemischte Regeln: $A \to aBcAbA$
   $\Rightarrow$ neue (Zwischen-) Variablen und Regeln

Chomsky Normalform – Technik (II)

Beispiel für (den Beginn) der Transformationstechniken (Ansatz 1)

$$\begin{aligned} S &\to ASA \,|\, aB \\ A &\to B \,|\, S \\ B &\to b \,|\, \epsilon \end{aligned}$$

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \,|\, aB \\ A &\to B \,|\, S \\ B &\to b \,|\, \epsilon \end{aligned}$$

. . . und nun die beiden ϵ wegtransformieren

Bedeutung der CNF

- Der “Cocke-Younger-Kasami-Algorithmus” (CYK) erlaubt die Konstruktion des Parse-Trees zu einem gegebenen Wort.
- Voraussetzung for CYK: die Grammatik muss in CNF vorliegen.

Da die Grammatik jeder kontextfreien Sprache in CNF gebracht werden kann, ist das Wortproblem für kontextfreie Sprachen lösbar.

(Verallgemeinerung der CNF für kontext*sensitive* Sprachen: Kuroda-Normalform)

Keller-Automat

Ansatz: endlicher Automat, der beliebig viele Symbole
auf einem “Stack” (Stapel) zwischenlagern kann.

Theoreme: (ohne Beweis)
a) Jede kontextfreie Sprache kann von einem
nicht-deterministischen Keller-Automat akzeptiert werden.
b) Nicht-deterministische Keller-Automaten akzeptieren
gerade die kontextfreien Sprachen.

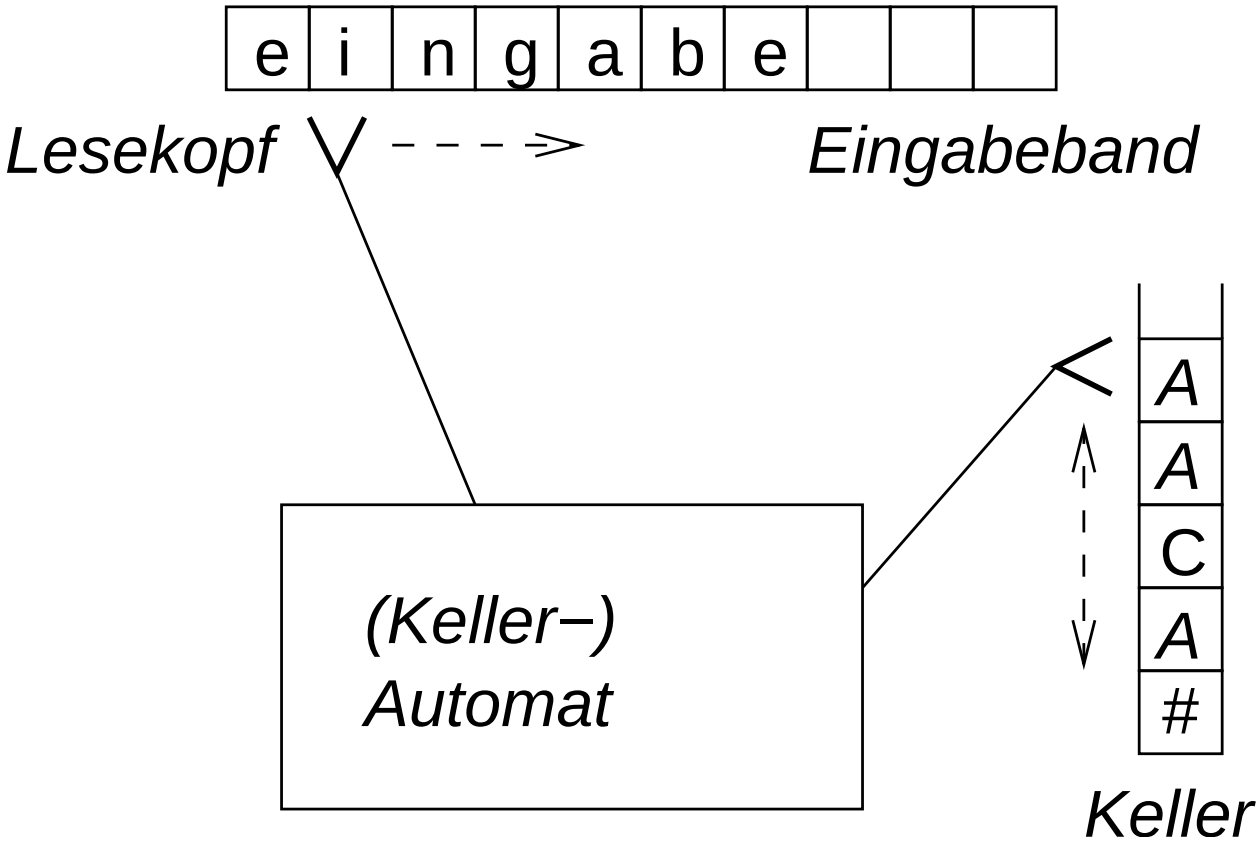
(Deterministische Keller-Automaten erkennen nur eine Teilmenge
der kontextfreien Sprachen, aber mehr als die regulären Sprachen)

Def. eines Keller-Automaten (Push Down Automaton)

PDA = 6-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$

Z	endl. Zustandsmenge
Σ	Eingabealphabet
Γ	Keller-Alphabet
δ	$Z \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow$
	Menge aller endl. Teilmengen von $Z \times \Gamma^*$
z_0	Startzustand
#	Stapel-End-Symbol

Keller-Automat – graphische Darstellung



Keller-Automat – Abarbeitung

1. $\delta(z, a, A) \to (z', B_1 \dots B_n)$
   "Push": aus Zustand z, oberstes Stapelsymbol A und Input a,
   gehe in neuen Zustand z' und lege $B_1 \dots B_n$ auf den Stapel

2. $\delta(z, a, A) \to (z'', \{\})$ "Pop": aus Zustand z, oberstes Stapelsymbol A und Input a,
   gehe in neuen Zustand z'' und entferne A vom Stapel

3. $\delta(z, \epsilon, A) \to (z''', \{\})$
   "stille Transition": aus Zustand z, oberstes Stapelsymbol A,
   gehe in neuen Zustand z''' ohne einen Input zu konsumieren.