Grundlagen der Programmierung (Vorlesung 23)

Ralf Möller, FH-Wedel

- Vorige Vorlesung
  - Endliche Automaten
- Inhalt dieser Vorlesung
  - Reguläre Ausdrücke, Eigenschaften regulärer Sprachen, Kontextfreie Sprachen, CYK-Algorithmus für Wortproblem
- Lernziele
  - Kennenlernen von abstrakten Maschinen zur Generierung von Sprachen oder zur Entscheidung des Wortproblems

Danksagung

- Die Präsentationen sind an den Inhalt des Buches "Theoretische Informatik kurzgefaßt" von Uwe Schöning angeleht und wurden aus den Unterlagen zu der Vorlesung "Informatik IV – Theoretische Informatik" an der TU München von Angelika Steger übernommen
- Die Originalunterlagen befinden sich unter: http://www14.in.tum.de/lehre/2000SS/info4/

Vorbemerkung: Potenzmenge

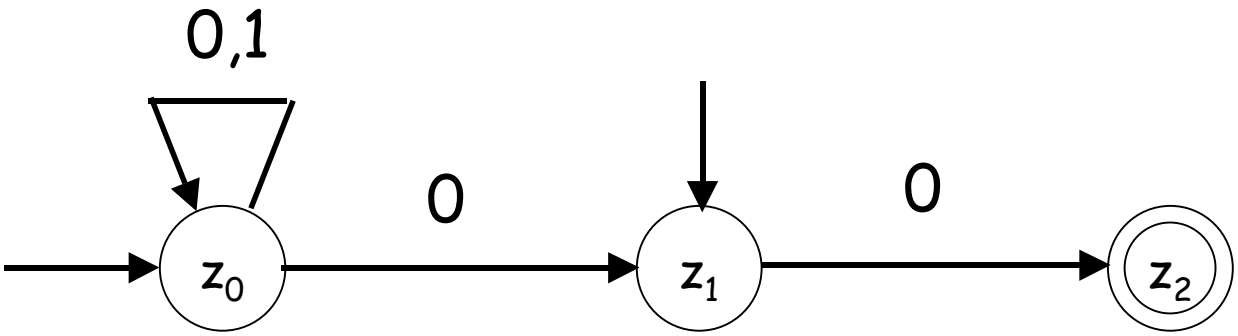
- Sei M eine (nicht-leere) Menge.
- Dann heißt $\mathcal{P}(M) := \bigcup_{M' \subseteq M} M'$ die Potenzmenge von M.

Nichtdeterministischer Endlicher Automat

Definition. Ein *nichtdeterministischer endlicher Automat* (englisch: nondeterministic finite automata, kurz NFA) wird durch ein 5-Tupel $M = (Z, \Sigma, \delta, S, E)$ beschrieben, das folgende Bedingungen erfüllt:

- Z ist eine endliche Menge von *Zuständen*.
- Σ ist eine endliche Menge, das *Eingabealphabet*, wobei $Z \cap \Sigma = \emptyset$.
- $S \subseteq Z$ ist die Menge der *Startzustände*.
- $E \subseteq Z$ ist die Menge der *Endzustände*.
- $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ heißt *Übergangsfunktion*.

Beispiel



- Dieser NFA akzeptiert genau die Wörter x über {0, 1}, die mit 00 enden, oder die Wörter x = 0.

Akzeptierte Sprache

Die von M *akzeptierte* Sprache ist

$$L(M) := \{w \in \Sigma^* \mid \hat{\delta}(S, w) \cap E \neq \emptyset\},$$

wobei $\hat{\delta} : Z \times \Sigma^* \to \mathcal{P}(Z)$ wieder induktiv definiert ist durch

$$\begin{aligned} \hat{\delta}(Z', \epsilon) &= Z' \quad \text{für alle } Z' \subseteq Z \\ \hat{\delta}(Z', ax) &= \bigcup_{z \in Z'} \hat{\delta}(\delta(z, a), x) \end{aligned}$$

Satz. Ist $G = (V, \Sigma, S, P)$ eine reguläre Grammatik, so ist $M = (V \cup \{X\}, \Sigma, \delta, S, E)$, wobei

$$E := \begin{cases} \{S, X\}, & \text{falls } S \to \epsilon \in P \\ \{X\}, & \text{sonst} \end{cases}$$

$$B \in \delta(A, a) \quad \text{falls} \quad A \to aB \in P$$

$$X \in \delta(A, e) \quad \text{falls} \quad A \to a \in P$$

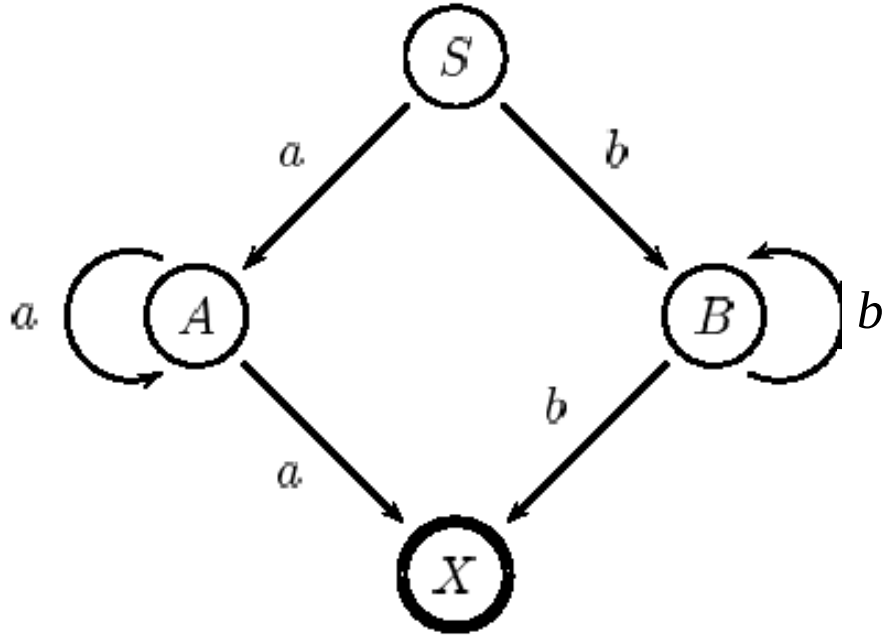
ein nicht deterministischer endlicher Automat. **und L(M) = L(G).**

Produktionen:

$$S \to aA, \quad S \to bB,$$

$$A \to a, \quad A \to aA,$$

$$B \to b, \quad B \to bB$$



NFA vs. DFA

Satz. (RABIN, SCOTT)
JEDE VON EINEM NFA AKZEPTIERBARE SPRACHE IST AUCH DURCH EINEN DFA AKZEPTIERBAR.

Beweis (1)

Beweis: Sei $M = (Z, \Sigma, \delta, S, E)$ ein gegebener NFA. Wir konstruieren einen DFA M', der ebenfalls $T(M)$ akzeptiert, dadurch dass wir in M' jede mögliche Teilmenge der Zustände von M (also die Elemente der Potenzmenge von Z) als *Einzelzustand* von M' vorsehen. Die restlichen Teile der Definition von M' ergeben sich dann mehr oder weniger zwangsläufig.

Wir setzen also

$$M' = (\mathcal{Z}, \Sigma, \delta', z_0', E')$$

wobei

$$\begin{aligned} \mathcal{Z} &= \mathcal{P}(Z) \\ \delta'(Z', a) &= \bigcup_{z \in Z'} \delta(z, a) = \hat{\delta}(Z', a), \; Z' \in \mathcal{Z} \\ z_0' &= S \\ E' &= \{Z' \subseteq Z \mid Z' \cap E \neq \emptyset\} \end{aligned}$$

Beweis (2)

Es ist klar, dass nun für alle $x = a_1 \ldots a_n \in \Sigma^*$ gilt:

$x \in T(M)$

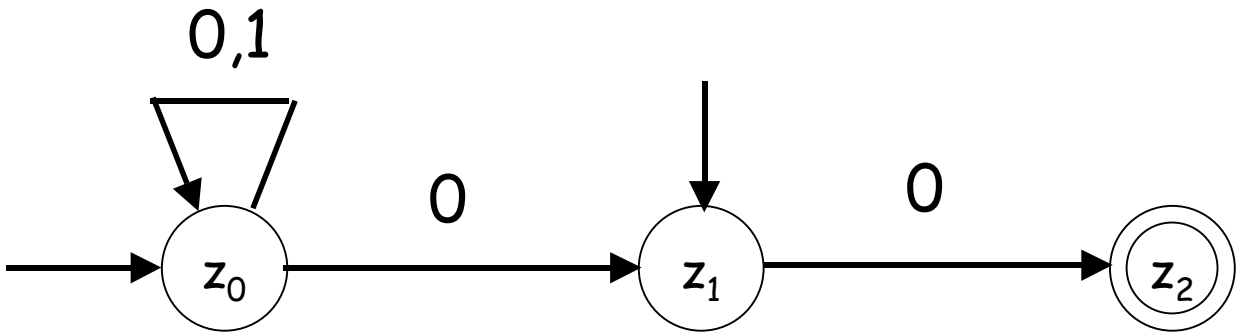
genau dann wenn $\hat{\delta}(S, x) \cap E \neq \emptyset$

genau dann wenn es gibt eine Folge von Teilmengen $Z_1, Z_2, \ldots, Z_n$ von Z mit $\delta'(S, a_1) = Z_1$, $\delta'(Z_1, a_2) = Z_2$, …, $\delta'(Z_{n-1}, a_n) = Z_n$ und $Z_n \cap E \neq \emptyset$.

genau dann wenn $\hat{\delta}'(S, x) \in E'$

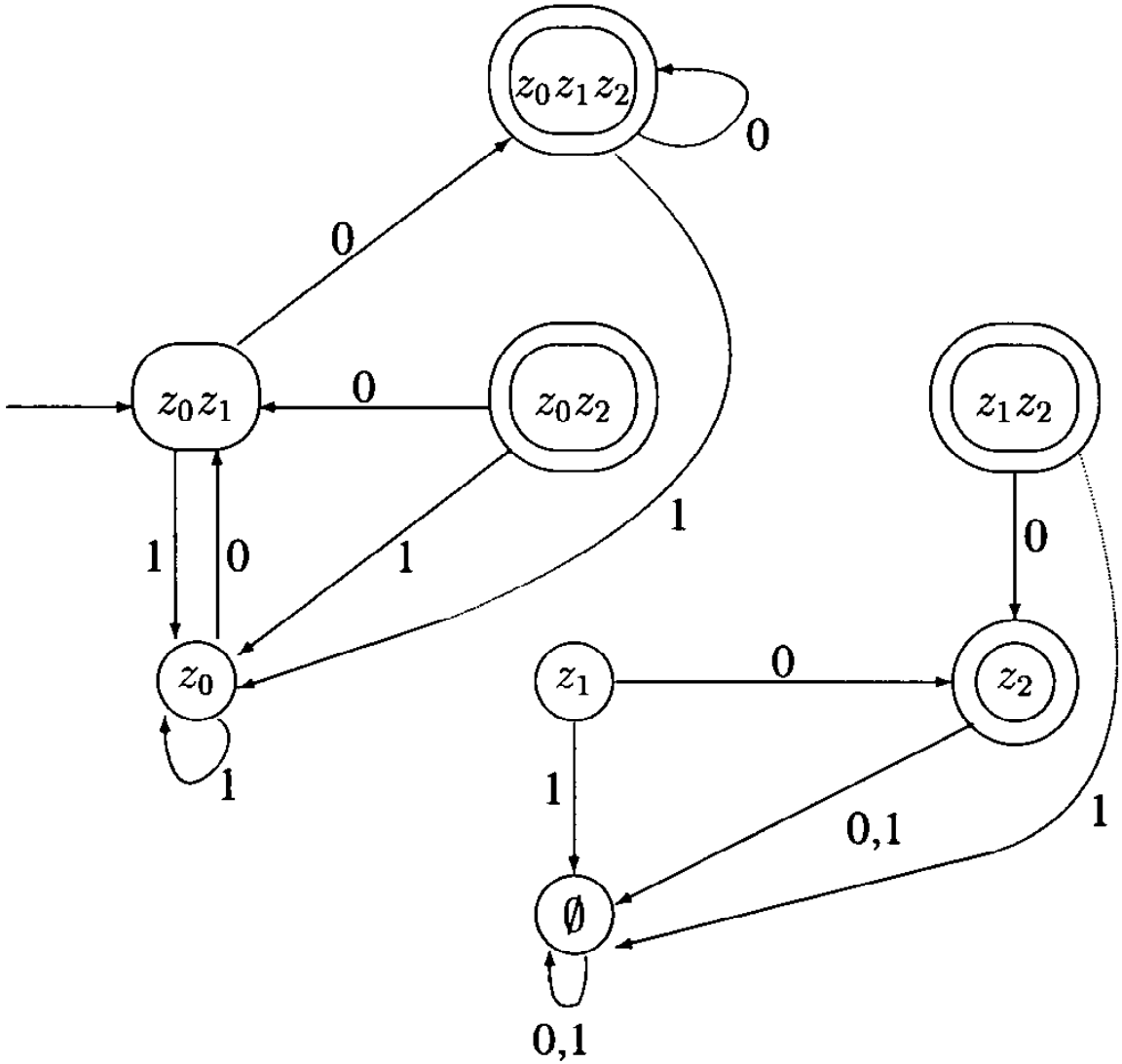
genau dann wenn $x \in T(M')$. ■

Beispiel (1)



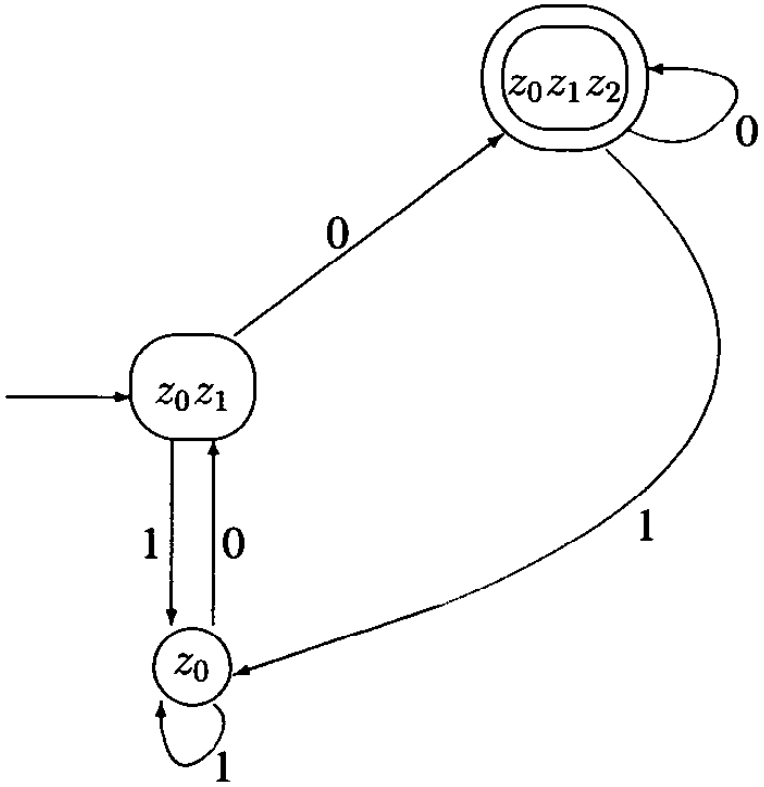
Beispiel: Für obigen nichtdeterministischen Automaten ergibt sich aus dem Beweis der folgende deterministische Automat mit den 8 Zuständen $\emptyset$, $\{z_0\}, \{z_1\}, \{z_2\}, \{z_0, z_1\}, \{z_0, z_2\}, \{z_1, z_2\}, \{z_0, z_1, z_2\}$. Der Startzustand ist $\{z_0, z_1\}$ (da z_0 und z_1 die Startzustände des NFAs sind). Die Endzustände des neuen Automaten sind alle Zustände, die mindestens einen ursprünglichen Endzustand enthalten.

Beispiel (2)



Beispiel (3)

Im Allgemeinen enthält der so entstandene deterministische Automat viele überflüssige Zustände. Wenn wir alle Zustände entfernen, die vom Startzustand (hier: $z_0 z_1$) nicht erreichbar sind, erhalten wir:



Damit ergibt sich als Zusammenfassung

- Für einen DFA existiert eine reguläre Grammatik, der die gleiche Sprache akzeptiert
- Für eine reguläre Grammatik existiert ein NFA, der die gleiche Sprache akzeptiert
- Für einen NFA existiert ein DFA, der die gleiche Sprache akzeptiert
- -> Es gibt keine regulären Sprachen, die inhärent mehrdeutig sind (man kann über die obigen Umformungen eine eindeutige Grammatik erzeugen)

Reguläre Ausdrücke

Definition. *Reguläre Ausdrücke* sind definiert durch:

- $\emptyset$ ist ein regulärer Ausdruck.
- ε ist ein regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist a ist ein regulärer Ausdruck.
- Wenn α und β reguläre Ausdrücke sind, dann sind auch $\alpha\beta$, $(\alpha|\beta)$ und $(\alpha)^*$ reguläre Ausdrücke.

Zu einem regulären Ausdruck γ ist eine zugehörige Sprache $L(\gamma)$ induktiv definiert durch:

- Falls $\gamma = \emptyset$, so gilt $L(\gamma) = \emptyset$.
- Falls $\gamma = \varepsilon$, so gilt $L(\gamma) = \{\varepsilon\}$.
- Falls $\gamma = a$, so gilt $L(\gamma) = \{a\}$.
- Falls $\gamma = \alpha\beta$, so gilt

$$L(\gamma) = L(\alpha)L(\beta) = \{uv \mid u \in L(\alpha), v \in L(\beta)\}.$$

- Falls $\gamma = (\alpha \mid \beta)$, so gilt

$$L(\gamma) = L(\alpha) \cup L(\beta) = \{u \mid u \in L(\alpha) \vee u \in L(\beta)\}.$$

- Falls $\gamma = (\alpha)^*$, so gilt

$$L(\gamma) = L(\alpha)^* = \{u_1 u_2 \dots u_n \mid n \in \mathbb{N}, u_1, \dots, u_n \in L(\alpha)\}$$

Pumping Lemma

Satz: (Pumping Lemma, uvw Theorem)

Sei L eine reguläre Sprache. Dann gibt es ein $n \in \mathbb{N}$, so daß es für jedes Wort $x \in L$ mit $|x| \geq n$ eine Zerlegung $x = uvw$ gibt, so daß

1. $|v| \geq 1$,
2. $|uv| \leq n$,
3. für alle $i \in \mathbb{N}_0$ gilt: $uv^i w \in L$.

Beweis

Beweis: Da L regulär ist, gibt es einen DFA M, der L akzeptiert.

Wir wählen für n die Zahl der Zustände von M: $n = |Z|$. Sei nun x ein beliebiges Wort der Länge $\geq n$, das der Automat akzeptiert. Beim Abarbeiten von x durchläuft der Automat $|x| + 1$ Zustände (den Startzustand mitgezählt). Da $|x| \geq n$ können diese Zustände nicht alle verschieden sein (Schubfachschluss). Mit anderen Worten, der DFA muss beim Abarbeiten von x eine Schleife durchlaufen haben. Wir wählen die Zerlegung $x = uvw$ so, dass der Zustand nach Lesen von u und von uv derselbe ist. Es ist klar, dass die Zerlegung so gewählt werden kann, dass die Bedingungen 1 und 2 erfüllt sind. Da die Zustände gleich sind, erreicht der Automat bei Eingabe von uw denselben Endzustand wie bei Lesen von $x = uvw$. Das heißt $uw = uv^0w \in L$. Dasselbe gilt für $uvvw = uv^2w$, $uvvvw = uv^3w$, usw. Daher ist auch die Bedingung 3 erfüllt. ■

Verwendung des Pumping Lemmas

Man beachte, dass durch das Pumping Lemma nicht eine äquivalente Charakterisierung der regulären Sprachen erreicht wird. Es stellt lediglich eine einseitige Implikation dar. Die logische Struktur ist

$$(L \text{ regulär}) \Rightarrow (\exists n)(\forall z \in L, |z| \geq n)(\exists u, v, w)[(z = uvw) \wedge 1 \wedge 2 \wedge 3]$$

Daher ist die typische Anwendung des Pumping Lemmas der Nachweis, dass gewisse Sprachen *nicht* regulär sind. Dies muss allerdings nicht bei jeder nicht-regulären Sprache gelingen!

Ein Beispiel ...

- in dem sich das Pumping-Lemma anwenden läßt, um die Nicht-Regularität zu zeigen:
- Wir betrachten $L = \{ a^n b^n \mid n \geq 1 \}$
- Annahme: L ist regulär
- Es gibt also ein Zahl n, so daß sich alle Wörter x aus L der Länge größer n wie im Pumping-Lemma beschrieben zerlegen lassen
- Wie betrachten das Wort $a^n b^n$ der Länge 2n
- Zerlegung: v nicht leer, v nur aus a's bestehend
- Also: auch $a^{n-|v|} b^n$ in L (Widerspruch)
- Also muß die Annahme falsch sein, d.h. L ist nicht regulär

Ein Beispiel, wo es nicht gelingt, ...

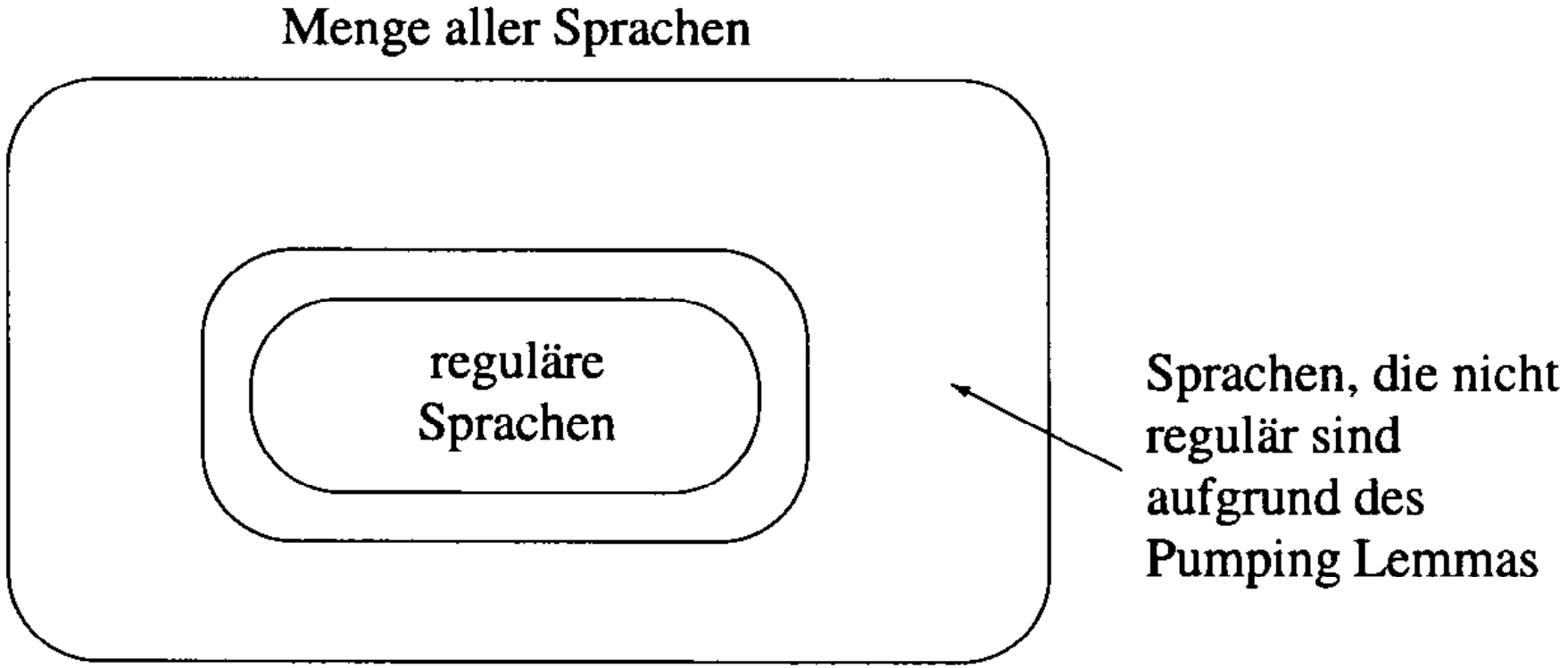
- das Pumping-Lemma anzuwenden, um die Nicht-Regularität zu zeigen:

$$L = \{c^m a^n b^n \mid m, n \geq 0\} \cup \{a, b\}^*$$

denn sie erfüllt die Behauptung des Pumping-Lemmas: Dies ist klar für Wörter aus $\{a, b\}^*$. Sei $k \geq 1$ die Pumping Lemma Zahl zu L und sei nun $x = c^m a^n b^n$ ein Wort aus L der Länge $\geq k$. Dann ist zum Beispiel $u = \varepsilon$, $v = c$, $w = c^{m-1} a^n b^n$ eine Zerlegung, die die Eigenschaften 1,2,3 besitzt. (Man beachte, dass dies auch für den Fall $m = 1$ gilt).

Es gibt also Sprachen, die nicht regulär sind

Graphische Darstellung:



Nicht-reguläre Sprachen

- Geklammerte Sprachkonstrukte sind *nicht* regulär, denn schon

$$L = \{a^n b^n : n \in \mathbb{N}\}$$

ist *nicht* regulär (Beweis durch Anwendung des Pumping Lemmas)...

Ausprägungen:

- Arithmetische Ausdrücke

$$E \to T|E+T, \quad T \to F|T*F, \quad F \to a|(E)$$

- Klammerpaare in Programmiersprachen: $\{\ldots\}$, `begin...end`, `repeat...until`,`then...end`

Kontextfreie Sprachen

$L = \{a^n b^n : n \in \mathbb{N}\}$ ist kontextfrei:

$$S \to ab \mid aSb$$

- Die Menge der regulären Sprachen ist also eine *echte* Untermenge der Menge der kontextfreien Sprachen.

Entscheidung des Wortproblems für Typ-2-Sprachen

Beispiel: Die Sprache

$$L = \{a^n b^n c^m \mid n, m \geq 1\}$$

ist kontextfrei:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow ab \mid aAb \\ B &\rightarrow c \mid cB \end{aligned}$$

Umformen in CNF ergibt:

$$\begin{aligned} S &\rightarrow AB & \qquad D &\rightarrow b \\ A &\rightarrow CD \mid CF & E &\rightarrow c \\ B &\rightarrow c \mid EB & F &\rightarrow AD \\ C &\rightarrow a \end{aligned}$$

Anwendung des Algorithmus für Typ-1-Sprachen?

- Generierung aller Worte bis zu einer Länge n bedeutet exponentiellen Aufwand
- Verwendung eines nichtdeterministischen Kellerautomaten?
  - Nicht besser, da Nichtdeterminismus durch Suche in einem Algorithmus ausgedrückt werden muß.
  - Besser im Worst-Case?
- Idee: Bei der Suche Zwischenergebnisse aufbewahren
- Warum beim Startsymbol anfangen?

Bottom-Up-Vorgehen

Sei $x = aaabbbcc$. Dann erzeugt der Algorithmus die folgende Tabelle:

$x =$	a	a	a	b	b	b	c	c
$j \downarrow$ / $i \rightarrow$								
1	C	C	C	D	D	D	E, B	E, B
2			A				B	
3			F					
4		A						
5		F						
6	A							
7	S							
8	S							

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow CD \mid CF \\
B &\rightarrow c \mid EB \\
C &\rightarrow a
\end{aligned}
\qquad
\begin{aligned}
D &\rightarrow b \\
E &\rightarrow c \\
F &\rightarrow AD
\end{aligned}
$$

Da S im untersten Kästchen vorkommt, liegt x in der Sprache.

CYK-Algorithmus

Eingabe: $x = a_1 a_2 \ldots a_n$

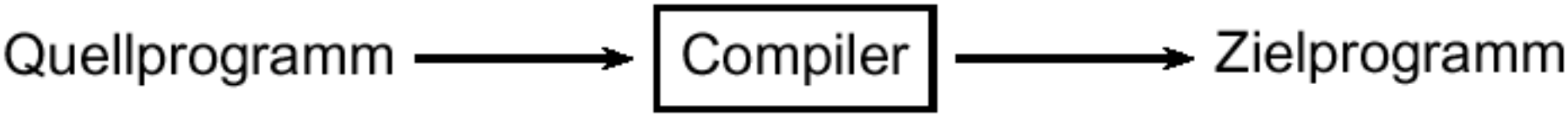
FOR $i := 1$ TO n DO $(*$ Fall $j = 1 *)$
  $T[i,1] := \{A \in V \mid A \to a_i \in P\}$
END;
FOR $j := 2$ TO n DO $(*$ Fall $j > 1 *)$
  FOR $i := 1$ TO $n + 1 - j$ DO
    $T[i,j] := \emptyset$;
    FOR $k := 1$ TO $j - 1$ DO
      $T[i,j] := T[i,j] \cup \{A \in V \mid A \to BC \in P$
        $\wedge B \in T[i,k] \wedge C \in T[i+k, j-k]\}$
    END;
  END;
END;
IF $S \in T[1,n]$ THEN
  WriteString('x liegt in L(G)')
ELSE
  WriteString('x liegt nicht in L(G)')
END

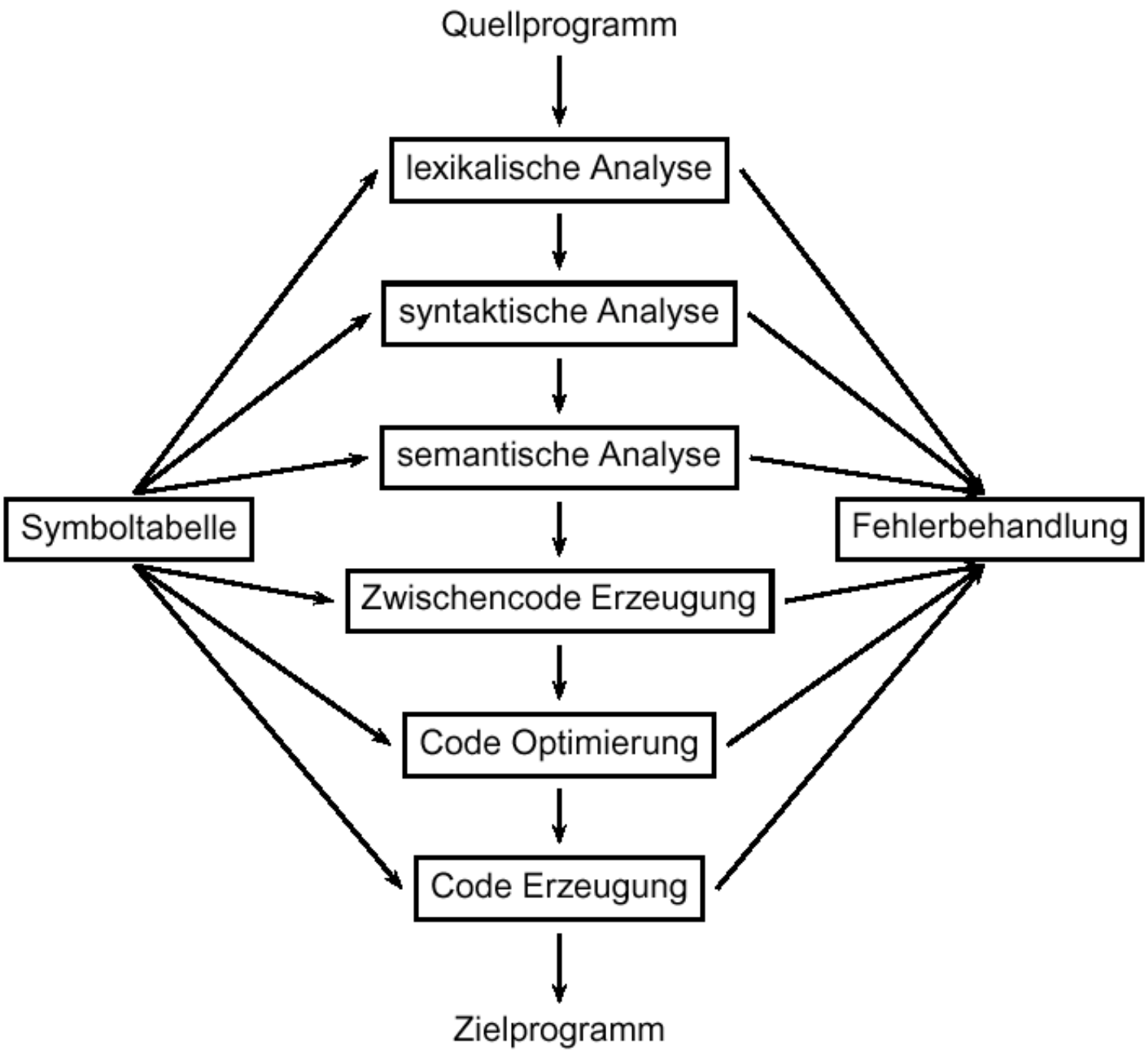
Der Algorithmus ist benannt nach den Entwicklern: Cocke, Younger, Kasami

Analyse des Algorithmus

Es ist offensichtlich, dass dieser Algorithmus die Komplexität $O(n^3)$ hat, denn er besteht aus 3 ineinander verschachtelten FOR-Schleifen, die jeweils $O(n)$ viele Elemente durchlaufen.

Anwendungen: Compilerbau




Quellprogramm
lexikalische Analyse
syntaktische Analyse
semantische Analyse
Symboltabelle
Fehlerbehandlung
Zwischencode Erzeugung
Code Optimierung
Code Erzeugung
Zielprogramm

Lexikalische Analyse: Scanner

Aufgabe: Extrahiere aus dem Eingabestring die nächste "Einheit", z.B. Namen einer Variablen, eine Zahl, reserviertes Wort (wie z.B. if, while, etc.), "+"-Zeichen, usw.

BEISPIEL: **identifier** := letter(letter|digit)*

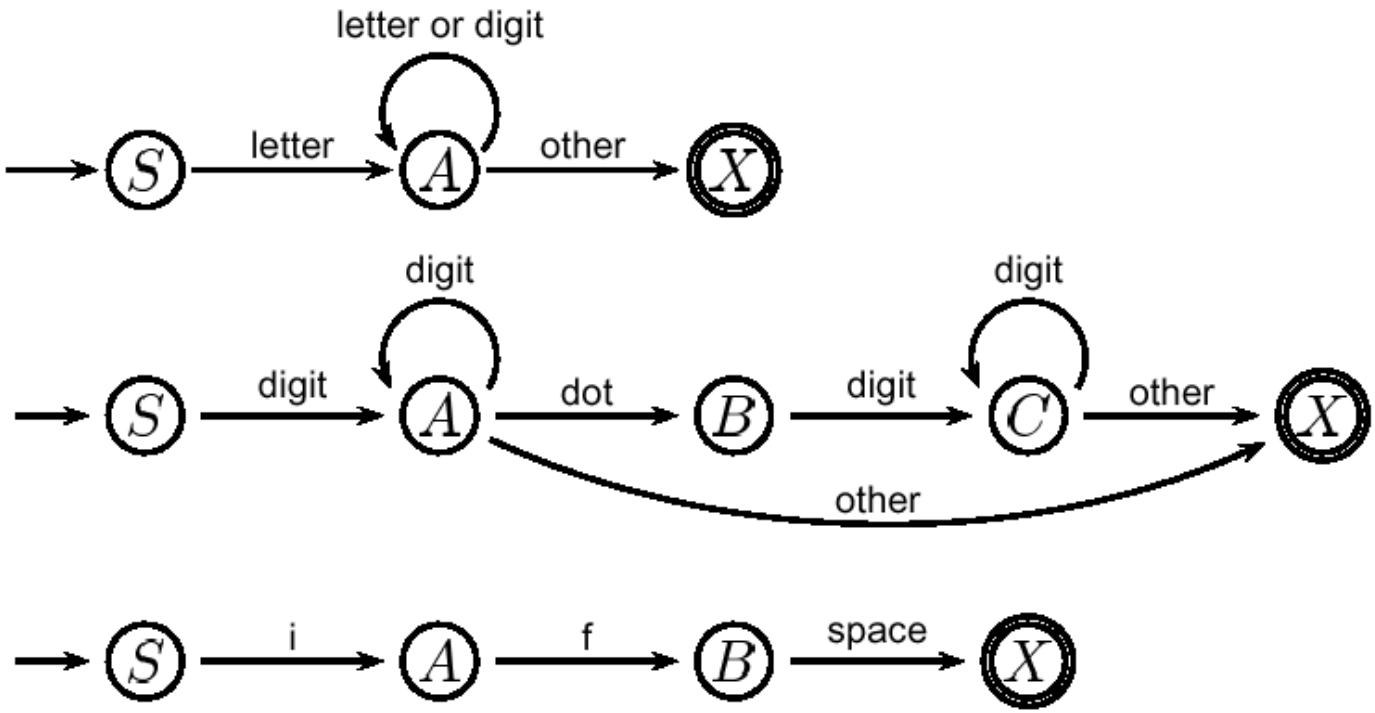
number := digit+ | digit+.(digit)+

if := if

Anwendung der Erkenntnisse der Vorlesung

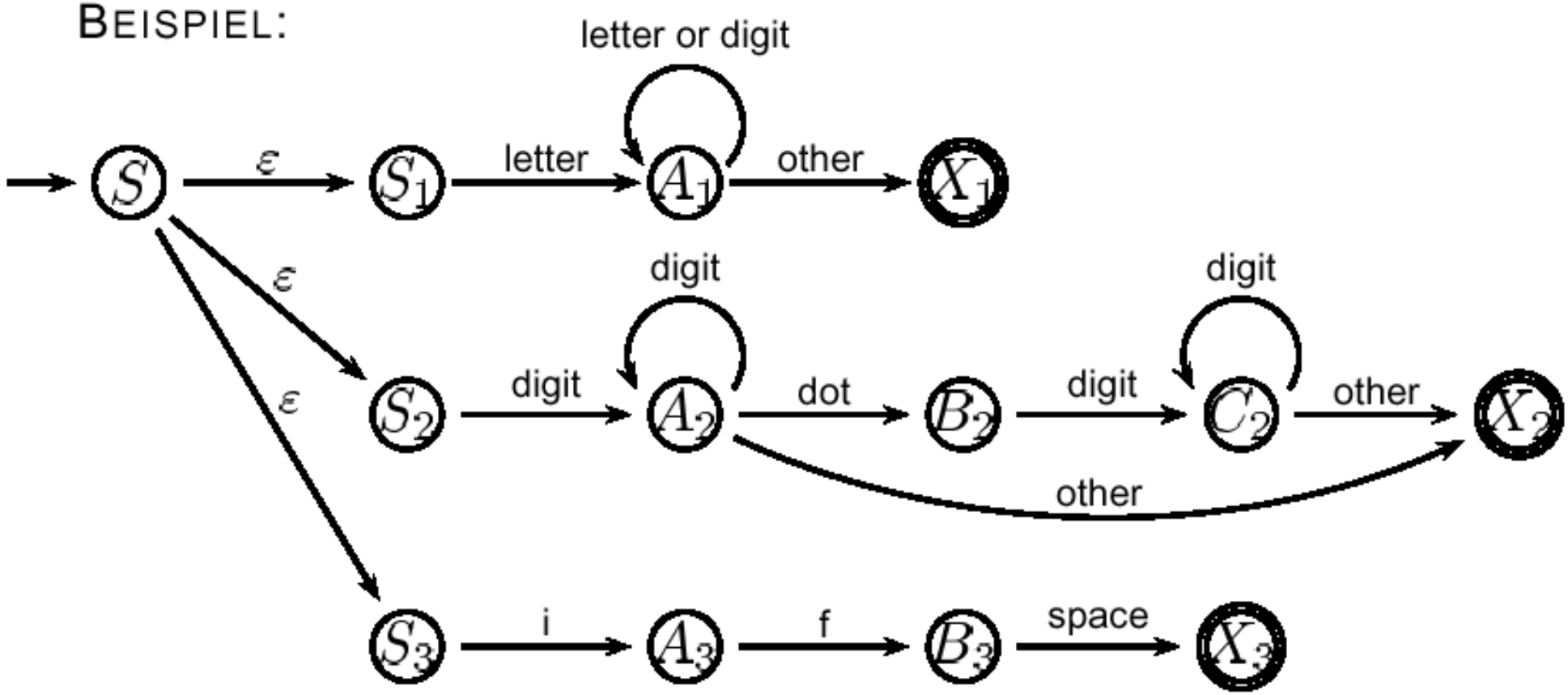
Zu jedem regulären Ausdruck gibt es einen endlichen Automaten, der genau die Wörter aus dieser Sprache erkennt.

BEISPIEL:



Für die lexikalische Ananlyse verbindet man diese endlichen Automaten zu einem einzigen *nichtdeterministischen* Automaten.

BEISPIEL:



Mit Hilfe der *Potenzmengenkonstruktion* kann man daraus wieder einen deterministischen endlichen Automaten bauen ... und aus diesem kann man dann relativ einfach ein C oder Java Programm erzeugen.

Tools: Es gibt fertige Programme, die aus regulären Ausdrücken den zugehörigen Parser erzeugen.

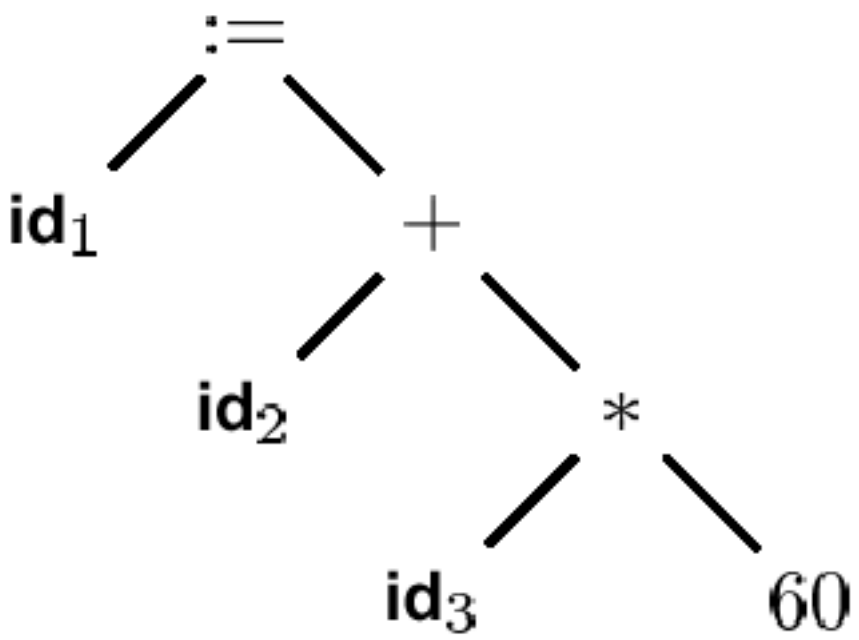
Klassisch: **lex** (A Lexical Analyzer Generator)

Gnu-Version: **flex**

Syntaktische Analyse: Parser

Aufgabe: Extrahiere aus der vom Scanner bereitgestellten Eingabe die logische Struktur.

BEISPIEL: Aus $\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60$ soll werden:



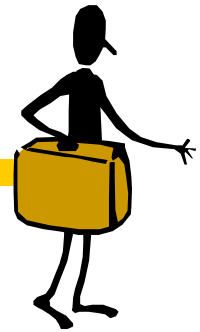
Parsing von Programmiersprachen

Ansatz: Die zulässige Syntax eines Programms wird durch eine (möglichst deterministische) kontextfreie Grammatik beschrieben.

Anwendung der Erkenntnisse der Vorlesung:

- Mit Hilfe des CYK-Algorithmus kann man in $\mathcal{O}(n^3)$ Zeit aus einem Wort die zugehörige Ableitung rekonstruieren.
- Ist die Grammatik deterministisch kontext-frei geht dies sogar in linearer Zeit.

Zusammenfassung, Kernpunkte



- Pumping-Lemma für reguläre Sprachen
  - "Nicht-Regularität" einiger Sprachen

Anwendungen Kontextfreier Sprachen

- CYK-Algorithmus

Was kommt beim nächsten Mal?

- Kellerautomaten
- Automaten für Typ-1- und Typ-0-Sprachen