

# Komplexität und Formale Sprachen

Karsten Wolf

# Worum geht es?

## *Grenzen der Informatik:*

- Was kann man prinzipiell berechnen, und was nicht?  
→ *Berechenbarkeit*
- Was kann man unter Einsatz vernünftiger Ressourcen berechnen, und was nur mit unvernünftigem Aufwand?  
→ *Komplexität*
- Wie komplex muss ein Regelwerk/eine Maschine sein, um ein Problem (als Sprache) zu lösen  
→ *Formale Sprachen*

# Wozu ist das gut?

1. Kenntnis eigener Grenzen gehört zur *Kultur* des Fachs

Beispiele von anderswo: Unmöglichkeit von ...

- Physik: Perpetuum Mobile, Maxwellschem Dämon,...
- Chemie: Quecksilber zu Gold
- Mathematik: Axiomatisierung der Zahlentheorie

2. Viele *relevante* Probleme liegen jenseits der Grenzen:

- mit an Sicherheit grenzender Wahrscheinlichkeit nicht effizient lösbar  
z.B:
  - Optimale Auslastung, optimale Routen, ...
  - Erfüllbarkeit in der Aussagenlogik

3. Unlösbarkeit/schwere Lösbarkeit kann *gewollt* sein

- Kryptologie

4. Technologie für tatsächliche Lösung von Problemen

# Fertigkeiten

„Diagnose“:

- Ein Gefühl für die Schwere eines Problems
- Methodik zum Nachweis der schweren Lösbarkeit

„Therapie“:

- Kenntnis der Annahmen, unter denen die Resultate zustandekommen
- Methoden zum Umgang mit unlösbaren bzw. schwer lösbaren Problemen

# Inhalt

## *Teil I: Komplexität*

1. Der Aufwand eines Problems
2. Die Klassen P und NP
3. NP-Vollständigkeit
4. Die Beweismethode  
Polynomialzeitreduktion;  
Einige NP-vollständige Probleme
5. Umgang mit schwer lösbaren  
Problemen

## *Teil I: Formale Sprachen*

1. Sprachen
2. Automaten
3. Reguläre Sprachen/endliche  
Automaten
4. Kontextfreie Sprachen/  
Kellerautomaten
5. Jenseits von Sprachen/Jenseits von  
Automaten

# Teil 1: Komplexität

# Setting

- Betrachten ab jetzt nur totale Funktionen
- Aufwand
  - einer Berechnung (eines Programmlaufs):
    - Zeit = Anzahl der Schritte bis Terminierung
    - Platz = Anzahl der benutzten Zellen
  - eines Programms (einer Maschine)
    - *worst case*: Funktion, die jedem  $n$  das Maximum der Aufwände von Berechnungen bei Eingaben der Länge  $n$  zuordnet
    - *average case*: ... den Durchschnitt ...
    - *best case*: ... das Minimum
  - eines Problems = Komplexität
    - = Aufwand des *besten* Programms zur Lösung des Problems

# Schritte, Zellen: Auf welcher Maschine???

- Variante 1: Turing-Maschine
- Problem: Varianten, Alphabetgröße etc. beeinflussen das Resultat
  - Wie weit?
  - Satz: Zu jeder TM  $M$ , die eine Funktion  $f$  mit Platzbedarf  $S_M: \mathbb{N} \rightarrow \mathbb{N}$  berechnet und jedem  $\varepsilon$  ex. eine Maschine  $M'$ , für deren Platzbedarf  $S_{M'}$  für fast alle  $n$  gilt:  $S_{M'}(n) \leq \lceil \varepsilon S_M(n) \rceil$ 
    - Idee:  $X' = X^n$ , d.h. eine Zelle von  $M'$  speichert Inhalt von  $n$  Zellen von  $M$
  - Satz, Zu jeder  $k$ -Band-TM  $M$  ( $k > 1$ ), die eine Funktion  $f$  mit Zeitbedarf  $T_M: \mathbb{N} \rightarrow \mathbb{N}$  berechnet und jedem  $\varepsilon$ , ex. eine TM  $M'$ , die  $f$  mit Zeitbedarf  $T_{M'}(n) \leq n + \lceil \varepsilon T_M(n) \rceil + c$  arbeitet
    - Idee: Komprimiere Zellen (wie oben) auf 2. Band und fasse alle Taktfolgen, bei denen Kopf innerhalb einer Zellengruppe bleibt, zu einem einzigen Takt zusammen



# Schritte, Zellen: Auf welcher Maschine???

- Satz: Sei  $M$  eine TM, die  $f$  mit Zeitaufwand  $T_M$  und Platzaufwand  $S_M$  berechnet. Sei  $k \in \mathbb{N}$ . Dann gibt es eine TM  $M'$ , die  $f$  berechnet, wobei
  - $T_{M'}(n) = n$ , falls  $n \leq k$ ,  $T_{M'}(n) = T_M(n)$ , sonst
  - $S_{M'}(n) = 0$ , falls  $n \leq k$ ,  $S_{M'}(n) = S_M(n)$ , sonst
- Idee: Für Eingaben der Länge  $\leq k$  (beschränkt viele) Tabelle mit vorberechneten Werten direkt ins Programm kodieren
- Fazit: Für Komplexitätsbetrachtungen:
  - nicht wichtig: lineare Faktoren
  - nur wichtig: asymptotisches Verhalten der Aufwandsfunktionen

# O-Notation

Seien  $f, g: \mathbb{N} \rightarrow \mathbb{N}$

Def:  $f \leq_{ae} g$  falls für fast alle  $n$ :  $f(n) \leq g(n)$

Def:  $O(f) := \{g \mid \text{ex. } k \in \mathbb{N} \setminus \{0\} : g \leq_{ae} k \cdot f\}$

d.h.  $\limsup_{n \rightarrow \infty} f(n)/g(n) < \infty$

Beispiele:

$0.001n + 10^{19} \in O(n)$      $1000000000n + 10^{19} \in O(n)$      $3n^2 - 27n + 18 \in O(n^2)$

$\lg n \in O(\log_2 n)$      $\ln n \in O(\log_2 n)$      $\log_{1000} n \in O(\log_2 n) \rightarrow O(\log n)$

$O(\log n) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \dots \subset O(n^k) \subset O(n^{k+1}) \subset \dots \subset O(2^n) \subset O(3^n) \dots$

# Random-Access-Maschinen (RAM)

- Realistischeres Maschinenmodell
- abzählbar viele Speicherregister  $R_0, R_1, \dots$ , jedes speichert (beliebig große) Zahl
- 6 Arbeitsregister
  - 3 Operationsregister (X,Y,Z für 2 Operanden und 1 Ergebnis)
  - 1 Adressregister A
  - 1 Befehlsregister B
  - 1 Flagregister F
- Programm mit numerierten Befehlen

# Random-Access-Maschinen (RAM)

- Befehlssatz variiert, z.B.
- $\text{COPY}(V, V')$ :  $V := V', B := B + 1$
- $\text{READ}(V)$ :  $V := R_A; B := B + 1$
- $\text{LOAD}(V, c)$ :  $V := c; B := B + 1$
- $\text{WRITE}(V)$ :  $R_A := V; B := B + 1$
- $\text{COMP}_{=/\neq/\leq/\geq/</>}$   $F := 1$ , falls  $X =/\neq/\leq/\geq/</> Y$ , sonst 0;  $B := B + 1$
- $\text{JUMP } b$   $B := b$ , falls  $F = 1$ , sonst  $B := B + 1$
- $\text{STOP}$  Maschine hält
- $\text{ADD}$   $Z := X + Y; B := B + 1$
- $\text{SUB}$   $Z := X \ominus Y; B := B + 1$
- $\text{MULT}$   $Z := X * Y; B := B + 1$
- $\text{DIV}$   $Z := X / Y; B := B + 1$
- $\text{SHIFT}$   $Z := Z/2; B := B + 1$

# Aufwand einer RAM

- Zahl der Schritte könnte unfair gegenüber TM sein, da RAM beliebig große Zahlen mit einem Befehl verarbeiten kann

→ zwei Maße

- uniformer Aufwand: Zahl der Schritte, Zahl der benutzten Register
- logarithmischer Aufwand: Summe der binären Logarithmen der verarbeiteten/gespeicherten Werte

# RAM versus TM

- Satz: Zu jeder RAM  $R$  mit logarithmischem Zeitaufwand  $T_R$  und logarithmischem Platzaufwand  $S_R$  existiert eine TM  $M$  mit  $T_M \in O(T_R \cdot S_R)$  und  $S_M \in O(T_R \cdot S_R)$
- Also: Wechsel zwischen RAM und TM ändert Potenz, verursacht aber keinen Wechsel von Polynomialfunktion zu Exponentialfunktion
- Ähnliche Resultate existieren für andere (vernünftige) Maschinenmodelle  
→ können zur Aufwandsabschätzung auch Schritte eines C-Programms zählen

# Komplexitätsklassen

Komplexitätsklasse = über den Aufwand definierte  
Menge von *Problemen*

$P = \{f \mid \text{ex. TM } M, \text{ die } f \text{ berechnet, ex. } k \text{ mit } T_M \in O(n^k)\}$

$PSPACE = \{f \mid \text{ex. TM } M, \text{ die } f \text{ berechnet, ex. } k \text{ mit } S_M \in O(n^k)\}$

$EXP = \{f \mid \text{ex. TM } M, \text{ die } f \text{ berechnet, ex. Polynom } p(n) \text{ mit } T_M \in O(2^p)\}$

$EXPSPACE = \{f \mid \text{ex. TM } M, \text{ die } f \text{ berechnet, ex. Polynom } p(n) \text{ mit } S_M \in O(2^p)\}$

# Schlimmer geht immer

Sei  $t: \mathbb{N} \rightarrow \mathbb{N}$  berechenbar, total. Dann gibt es eine berechenbare totale Funktion  $f$ , die von keiner TM  $M$  mit  $T_M \leq_{ae} t$  berechnet werden kann.

Beweisidee (Diagonalisierung):

Bestimmen  $f(n)$  wie folgt mit Hilfe einer universellen TM  $A$ :

Verfolge für diejenigen  $x$  zwischen 0 und  $n$ , die Gödelzahl einer TM sind,  $A(x, n)$  so viele Takte, dass dies der Abarbeitung von  $t(n)$  Takten der durch  $x$  kodierten TM entspricht. Setze  $f(n)$  auf irgendeinen Wert, der verschieden ist von den Resultaten von allen dabei auftretenden *terminierenden* Berechnungen.

Annahme: Es gibt ein  $M$ , das  $f$  berechnet und  $T_M \leq_{ae} t$ . Dann gibt es ein  $n$  mit  $n > \text{gö}(M)$  und  $T_M(n) < t(n)$ . Wid. zur Wahl von  $f(n)$ !



# Effiziente Probleme

Die Probleme in P werden im allgemeinen als die effizient lösbaren, die berechenbaren Probleme außerhalb von P als die nicht effizient lösbaren („intractable“) Probleme bezeichnet.

Gründe:

- Robustheit gegenüber trivialen Beschleunigungen
- Robustheit gegenüber Maschinenmodell
- „angenehme“ Wachstumseigenschaften:
- Robustheit gegenüber Konstruktionen

n	1	2	3	4	5	10	100	1000
$n^2$	1	4	9	16	25	100	10000	$10^6$
$n^3$	1	8	27	64	125	1000	1000000	$10^9$
$n^{10}$	1	1024	59049	1048576	9765625	$10^{10}$	$10^{20}$	$10^{30}$
$2^n$	2	4	8	16	32	1024	$\approx 10^{30}$	$\approx 10^{300}$

# Effiziente Probleme

... „angenehme“ Wachstumseigenschaften:

In der gleichen Zeit, in der ich auf Rechner X eine Eingabe der Länge  $n$  verarbeite, kann ich auf dem doppelt so schnellen Rechner Y Eingabe der Länge ...verarbeiten:

Aufwand	Eingabelänge	Verbesserung
$n$	$2n$	100%
$n^2$	$1.41n$	41%
$n^3$	$1.26n$	26%
$n^{10}$	$1.07n$	7%
$2^n$	$n+1$	$\rightarrow 0\%$

# Effiziente Probleme

... „Robustheit gegenüber Konstruktionen“

Sind P, Q Programme mit polynomiellen Aufwand, so auch

- Hintereinanderausführung P ; Q
- Alternative: if bla then P else Q end
- Ersetzen eines elementaren Schrittes in P durch Unterprogrammaufruf Q

...

# Nichtdeterministische Turing-Maschine

- deterministisch:  $\delta: Z \setminus \{z_f\} \times X \rightarrow Z \times X \times \{-1, 0, 1\}$
- nichtdeterministisch:  $\delta: Z \setminus \{z_f\} \times X \rightarrow \wp(Z \times X \times \{-1, 0, 1\})$
- Lauf einer nichtdeterministischen Maschine
  - $K \rightarrow K'$  gdw. *es existiert* ein  $[z', x', b] \in \delta(z, x)$  mit ...  
→ es gibt verschiedene Läufe bei gleicher Anfangskonfiguration
- nichtdet. Turingmaschine entscheidet die Menge  $\{x: \text{es gibt einen Lauf, der mit Resultat 1 terminiert}\}$

# Aufwand einer nichtdeterministischen Turing-Maschine

- Variante 1:
  - Zeit/Platzaufwand für Eingabe  $x$ : Länge der längsten Berechnung/Speicherverbrauch der üppigsten Berechnung
- Variante 2 (bei Entscheidungsproblemen):
  - Für Eingabe  $x$  aus  $M$ : Länge der kürzesten *akzeptierenden* Berechnung / Verbrauch der sparsamsten *akzeptierenden* Berechnung
  - Motivation: Parallelverarbeitung

# Komplexitätsklassen (für Entscheidungsprobleme)

NP - polynomieller Zeitaufwand (Variante 2) einer nichtdeterministischen Maschine

NEXP – exponentieller Zeitaufwand (Variante 2) einer nichtdeterministischen Maschine

usw.

Für Platz: keine neuen Klassen erforderlich, da diese mit den entsprechenden deterministischen Klassen zusammenfallen

(Backtracking-Algorithmen zur deterministischen Durchmusterung)

# Andere Aufwandsmaße

- Schaltkreiskomplexität:
  - Platz = Zahl der Gatter
  - Zeit = max. Zahl von Gattern auf dem Weg von Input zu Output
- Nebenläufige Maße
  - Platz = Zahl der verwendeten Prozessoren
  - Zeit = Länge des kritischen Berechnungspfades
- usw.

# Einige Zusammenhänge

- Für alle  $X$  ( $P$ ,  $EXP$ , ...) :
  - $XTIME \subseteq XSPACE$  (In  $n$  Takten kann ich max.  $n$  Zellen anfassen)
  - $NXTIME \subseteq XSPACE$  (dito)
  - $XTIME \subseteq NXTIME$  (det. Maschinen SIND nichtdet. Maschinen)
- Tiefer:  $P \subset EXPTIME$ ,  $PSPACE \subset EXPSPACE$  (sog. Hierarchiesätze  $\rightarrow$  weiterführende Vorlesungen zur Komplexitätstheorie)



# Offene Fragen:

- Haben:  $P \subseteq NP \subseteq PSPACE$

Aber:

- $P = NP?$
- $NP = PSPACE?$

# $P = NP?$

- bedeutendste Fragestellung der theoretischen Informatik
- seit Jahrzehnten offen, trotz gewaltiger Manpower
- hohes Preisgeld
- viele relevante Probleme in NP
- inzwischen allgemein akzeptierte Vermutung:  $P \neq NP$
- $\rightarrow NP \setminus P$  : „die leichtesten unter den schwer lösbaren Problemen“

# Einige Entscheidungsprobleme in NP

- SAT: Zu gegebenem Ausdruck der Aussagenlogik [in konjunktiver Normalform] ist seine Erfüllbarkeit zu entscheiden
  - „Rate“ Belegung der Variablen; Prüfe deterministisch
- NPRIM: Zu gegebener nat. Zahl  $> 1$  ist zu entscheiden, ob sie echte Teiler besitzt
  - „Rate“ Teiler; Dividiere mit Rest
- SET PACKING: Gegeben: Menge  $M$ ,  $n$  Teilmengen  $M_1, \dots, M_n$  und Zahl  $k$ . Frage: Gibt es unter den gegebenen Teilmengen  $k$  paarweise disjunkte?
  - Rate sie...
- 0/1-INTEGER LINEAR PROGRAMMING: Gegeben Ungleichungen, eine Zielfunktion  $f$  in  $n$  Variablen  $x_1, \dots, x_n$  sowie eine Zahl  $k$ . Frage: Gibt es eine Zuordnung von Werten 0 bzw. 1 für die  $x_i$  derart, dass  $f(x_1, \dots, x_k) \geq k$ ?
- KNAPSACK: Gegeben Menge, für jedes Element Gewicht und Wert, und Zahlen  $G$  und  $W$ . Frage: Gibt es eine Teilmenge, deren Gewichtssumme  $\leq G$  und Wertsumme  $\geq W$  ist?

# Einige Entscheidungsprobleme in NP

- CLIQUE: Gegeben: ungerichteter Graph  $[V, E]$  und Zahl  $k$ :  
Frage: gibt es eine Knotenmenge  $V' \subseteq V$  mit  $|V'| \geq k$  und  $V' \times V' \subseteq E$ ?
- TSP: Gegeben: vollständiger Graph  $[V, E = V \times V]$  mit Kostenfunktion  $c: E \rightarrow \mathbb{N}$  sowie  $k$ . Frage: Gibt es eine Rundreise in  $[V, E]$ , die billiger als  $k$  ist?
- COLOR: Gegeben: Ungerichteter Graph  $[V, E]$ , Zahl  $k$ . Frage: Kann man Knoten mit  $k$  Farben so färben, dass benachbarte Knoten immer verschiedene Farben haben?
- VERTEX COVER: Gegeben: ungerichteter Graph  $[V, E]$ , Zahl  $k$ . Frage: Gibt es eine Teilmenge  $V'$  mit  $|V'| \leq k$  derart, dass jede Kante mit einem Knoten aus  $V'$  verbunden ist?
- HAMILTON CIRCUIT: Gegeben: ungerichteter Graph. Frage: Gibt es einen Kreis, der alle Knoten durchläuft?

# Entscheidung vs. Optimierung

- Entscheidungsvariante eines Problems:
  - Geg.: ... und Zahl  $k$ . Frage: Gibt es Lösung des Problems mit Güte  $\leq$  (bzw.  $\geq k$ )
- Optimierungsvariante 1:
  - Geg.: .... Frage: Finde das minimale (bzw. maximale)  $k$ , so dass ...
- Optimierungsvariante 2:
  - Gegeben: Geg.: .... Frage: Finde das minimale (bzw. maximale)  $k$ , so dass ... sowie einen Satz Werte für die Problemparameter, für die dieses  $k$  realisiert wird!

# Entscheidung vs. Optimierung

Komplexität der Varianten unterscheidet sich oft nicht wesentlich:

- Optimierungsvarianten lösen trivialerweise Entscheidungsvarianten; Optimierung II löst Optimierung I
- Entscheidungsvariante eingebettet in eine Binärschachtelung für k löst Optimierung I
- Optimierung I schränkt Suchraum für Optimierung II wesentlich ein

## Beispiel: CLIQUE

```
PROC CLIQUE_Optimierung_II(V,E)
  k* := CLIQUE_Optimierung_I(V,E);
  U := E;
  REPEAT
    Wähle e aus U; U := U \{e}
    k := CLIQUE_Optimierung_I(V,E\{e})
    IF k = k* THEN E := E \{e} END;
  UNTIL U = Ø;
  RETURN E;
```

# Einige Probleme in P

- Für keins der Probleme in F170-171 (**außer NPRIM**) ist ein Algorithmus in P bekannt, im Unterschied zu:
- Geg: Ausdruck H der Aussagenlogik in *alternativer (disjunktiver)* Normalform. Frage: ist H erfüllbar? Antwort: genau dann, wenn in wenigstens einer Elementarkonjunktion keine Variable sowohl negiert als auch unnegiert vorkommt.
- Geg: Graph  $[V,E]$  Frage: Gibt es einen Euler-Kreis, d.h. einen Weg, der jede Kante genau einmal benutzt und zum Ausgangspunkt zurückkehrt. Antwort: Genau dann, wenn der Graph zusammenhängend ist und jeder Knoten eine gerade Zahl an Nachbarknoten hat
- Geg: Menge von Ungleichungen in Variablen  $x_1, \dots, x_n$  und Zielfunktion  $f$ . Gesucht: *rationale* Werte für  $x_1, \dots, x_n$ , die alle Ungleichungen erfüllen und Wert der Zielfunktion maximieren
- $\text{CLIQUE}_k$ : Gegeben: Graph  $[V,E]$  Frage: Gibt es eine Teilmenge  $V'$  mit  $|V'| \geq k$  und  $V' \times V' \subseteq E$ ? Antwort: for all  $v_1$  do ... for all  $v_k$  do ...



# Polynomialzeitreduktion

- Wenn  $P \neq NP$  wäre, wie könnte man zeigen, dass ein Problem  $M$  nicht in  $P$  liegt:
  - Antwort: Nehme ein Problem  $M'$ , bekanntermaßen außerhalb  $P$
  - zeige: wenn  $M$  in  $P$  läge, läge auch  $M'$  in  $P$
  - Weg: zeige eine Funktion **in  $P$** :  $N \rightarrow N$  vor mit  $x \in M' \text{ gdw. } f(x) \in M$
- Offenbar:  $P$ -Algorithmus für  $M$  und polynomielle Laufzeit für  $f$  würden zu  $P$ -Algorithmus für  $M'$  führen.

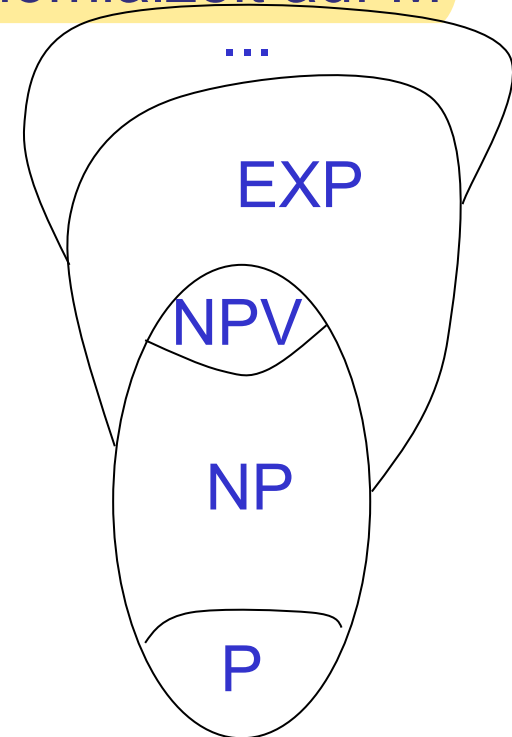
# NP-Vollständigkeit

- Wie kann man die Frage, ob  $P = NP$  ist, vereinfachen?
- M heißt **NP-vollständig**, falls
  - M ist aus NP
  - Jedes Problem aus NP lässt sich in Polynomialzeit auf M reduzieren („M ist NP-hart“)

→ Ist M aus P, ist  $P = NP$

→ Ist M nicht aus P, ist  $P \neq NP$

NP-vollständige Probleme sind unter den leichtesten nicht effizient lösbaren Problemen die schwersten



# Satz von Cook (1969): SAT ist NP-vollständig

Beweis Teil 1: SAT  $\in$  NP:

- „Rate“ Belegung; rechne Wert des Ausdrucks aus
- Braucht  $O(n)$  Schritte, wenn  $n$  die Anzahl der Symbole ist, aus denen der Ausdruck aufgebaut ist

Idee Teil 2: SAT ist NP-hart

- Gegeben: irgendein NP-Problem, also nichtdet. TM  $M$  und Polynom  $p$ , das  $T_M$  beschränkt.
- Bilden zu Eingabe  $x$  für  $M$  in Polynomialzeit aussagenlogischen Ausdruck  $\phi$ , der Arbeit von  $M$  nachbildet derart, dass  $\phi$  erfüllbar ist gdw. ein Lauf von  $M$  ex., der mit Ergebnis 1 terminiert.
- Nutzen dabei:  $M$  läuft auf  $x$  max.  $p(|x|)$  Takte;  $p$  effizient berechenbar

## Beweis des 2. Teils Satz von Cook

- Betrachten Indizes aus folgenden Indexmengen
  - $x$  aus  $\Gamma$ : das verwendete Bandalphabet der Maschine; Größe unabhängig von der Eingabelänge
  - $z$  aus  $Z$ : die verwendete Zustandsmenge der Maschine; Größe unabhängig von der Eingabelänge
  - $t$  aus TAKT:  $\{0, \dots, p(n)\}$ , polynomielle Größe in Abhängigkeit von der Eingabelänge  $n$
  - $a$  aus ADDRESS:  $\{-p(n), \dots, 0, \dots, p(n)\}$  polynomielle Größe abhängig von der Eingabelänge
- Betrachten Aussagenvariablen mit folgendem Verwendungszweck
  - $z_{tz}$ :  $[ \text{zustand}(t) = z ]$  Nach  $t$  Takten ist  $M$  in Zustand  $z$
  - $y_{ta}$ :  $[ \text{position}(t) = a ]$  Nach  $t$  Takten steht Kopf auf Zelle  $a$
  - $x_{tax}$ :  $[ \text{band}(t,a) = x ]$  Nach  $t$  Takten enthält Zelle  $a$  Wert  $x$

## Fortsetzung 2. Beweisteil

- Konstruierte Formel:  $A \wedge \ddot{U} \wedge E \wedge R$ 
  - A: Anfangsbedingung, beschreibt Startkonfiguration
  - $\ddot{U}$ : Übergangsfunktion, beschreibt Konfigurationswechsel
  - E: Endbedingung, beschreibt Endkonfiguration
  - R: Randbedingungen, schließt ungewollte Lösungen aus

Zu A (Anfangsbedingung): Für Eingabe  $x_1 \dots x_n$

$z_{00} \wedge$  ... Zustand nach 0 Takten ist  $z_0$

$y_{00} \wedge$  ... Kopfposition nach 0 Takten ist 0

$\bigwedge_{a=1 \dots n} x_{0,a-1,x_a} \wedge$  ... Eingabe nach 0 Takten auf dem Band

$\bigwedge_{-p(n) \leq a \leq 0 \text{ oder } n+1 \leq a \leq p(n)} x_{0,a-1,\square}$  ... restliches Band leer

Länge von A:  $2p(n)+3$ ; Konstruktion effizient realisierbar

## Fortsetzung 2. Beweisteil

Zu  $\ddot{U}$  (Übergangsrelation):

Für  $d(z,x)=\emptyset$  wird  $d(z,x) = \{(z,x,0)\}$  gesetzt (Terminierung auf Zeitpunkt  $p(n)$  verschoben)

$\left( \bigwedge_{0 \leq t < p(n), -p(n) \leq a \leq p(n), z, x} ((z_{tz} \wedge y_{ta} \wedge x_{tax}) \right) \dots$  Aktuelle Konfiguration

$\Rightarrow \bigvee_{(z',x',b) \in \delta(z,x)} (z_{t+1,z'} \wedge y_{t+1,a+b} \wedge x_{t+1,a,x'}) \dots$  Folgekonfiguration

$\wedge \bigwedge_{0 \leq t < p(n), -p(n) \leq a \leq p(n), x} (\neg y_{ta} \wedge x_{tax}) \Rightarrow x_{t+1,a,x}$   
 $\dots$  keine Änderung woanders

Länge von  $\ddot{U}$  etwa  $2|X||Z|p(n)^2 + 3|X||Z| + 6|X|p(n)^2$ ;  
 Konstruktion effizient realisierbar

## Fortsetzung 2. Beweisteil

Zu E (Endbedingung):

$z_{p(n),zf} \wedge$  ... Maschine im Endzustand

$\bigwedge_{-p(n) \leq a \leq p(n)} y_{p(n),a} \Rightarrow (x_{p(n),a+1,1} \wedge x_{p(n),a+2,\square} \wedge x_{p(n),a,\square})$   
... Am Kopf steht Ergebnis 1

Länge von E etwa  $8p(n)$  ;

Konstruktion effizient realisierbar

## Fortsetzung 2. Beweisteil

Zu R (Randbedingung):

$\bigwedge_{0 \leq t \leq p(n)} \bigvee_z z_{tz} \wedge \bigwedge_{z' \neq z} \neg z_{tz'}$  ... Maschine immer in genau einem Zst  
 $\wedge$

$\bigwedge_{0 \leq t \leq p(n)} \bigvee_{-p(n) \leq a \leq p(n)} y_{ta} \wedge \bigwedge_{a' \neq a} \neg y_{ta'}$  ...Kopf immer an genau 1 Pos  
 $\wedge$

$\bigwedge_{0 \leq t \leq p(n)} \bigwedge_{-p(n) \leq a \leq p(n)} \bigvee_x x_{tax} \wedge \bigwedge_{x' \neq x} \neg x_{tax'}$

...In jeder Zelle immer genau ein Zeichen

Länge von R etwa  $|Z|^2 p(n) + 4p(n)^3 + 2|X|^2 p(n)^2$

Konstruktion effizient realisierbar



# Rest des Beweises

- a) Falls ein Lauf der Maschine existiert, der mit 1 terminiert, liefert dieser Lauf auf naheliegende Weise eine erfüllende Belegung der konstruierten Formel
- b) Falls die konstruierte Formel erfüllbar ist, kann man aus der erfüllenden Belegung einen akzeptierenden Lauf der Maschine ableiten (R dabei wichtig)

Also: Ausdruck erfüllbar gdw. Maschine auf betrachteter Eingabe hält.

Außerdem: Konstruktion in  $O(p(n)^3)$  möglich

## 4. Weitere NP-vollständige Probleme

# Nochmal Polynomialzeitreduktion

Sei  $M$  (bekanntes) NP-vollständiges Entscheidungsproblem

Sei  $M'$  Entscheidungsproblem in NP und  $f$  Funktion in P mit

$x \in M$  genau dann, wenn  $f(x) \in M'$

Dann ist  $M'$  NP-vollständig.

Beweis: Sei  $M^*$  beliebiges NP-Problem. Weil  $M$  NP-vollständig, gibt es ein  $g$  aus P mit  $x \in M^*$  gdw  $g(x) \in M$ .

Zu  $M^*$  ist demnach  $f \circ g$  eine Funktion, die folgendes erfüllt:

- $f \circ g$  ist aus P
- $x \in M^*$  gdw  $f \circ g(x) = f(g(x)) \in M'$

Also ist  $M'$  NP-vollständig.

# Beispiel 1: 3SAT

- Problem: Geg: aussagenlogische Formel  $\phi$  in konjunktiver Normalform, in der jede Klausel max. 3 Literale enthält.  
Frage: Ist  $\phi$  erfüllbar?

3SAT ist aus NP, weil es Spezialfall von SAT ist und SAT in NP liegt.

Zeigen: SAT lässt sich auf 3SAT reduzieren

# SAT $\rightarrow$ 3SAT: Reduktionsidee

Voraussetzung: Negationen bis zu den Literalen nach innen getrieben (geht ohne nennenswerte Vergrößerung des Ausdrucks)

Idee: Einführung neuer Variablen, die den Wert komplexer Teilformeln repräsentieren

$$(H1 \text{ op } H2) \rightarrow x \text{ op } y \wedge (x \Leftrightarrow H1) \wedge (y \Leftrightarrow H2)$$

Das ganze rekursiv für H1, H2 fortsetzen

# Formal

- Definieren induktiv Menge der nichttrivialen Teilausdrücke eines Ausdrucks
  - $TA(\top) = TA(\perp) = TA(\pi) = \emptyset$
  - $TA(H1 \text{ op } H2) = TA(H1) \cup TA(H2) \cup \{(H1 \text{ op } H2)\}$
- Zu jedem  $H^*$  aus  $TA(H)$  sei  $x_{H^*}$  eine nicht in  $H$  vorkommende Variable; für elementare Teilausdrücke sei  $x_{H^*} = H^*$
- Dann ist zu beliebigem Ausdruck  $H$  (wo Negationen sich nur auf einzelne Variablen beziehen) der Ausdruck  $3H$  wie folgt definiert:
  - Falls  $H$  keinen Binäroperator enthält, so  $3H = H$ , sonst
  - $3H = x_H \wedge \bigwedge_{(H1 \text{ op } H2) \in TA(H)} (x_{(H1 \text{ op } H2)} \Leftrightarrow x_{H1} \text{ op } x_{H2})$
- $3H$  lässt sich in P-Zeit in CNF überführen
  - für jedes Konjunktionsglied: konstante Zeit

# Beispiel für Konstruktion

- $\neg(x \Rightarrow y) \vee (x \Leftrightarrow z)$
- Negationen nach innen:  $(x \wedge \neg y) \vee (x \Leftrightarrow z)$
- 3 neue Variablen u,o,g:

$$o \wedge u \Leftrightarrow (x \wedge \neg y) \wedge g \Leftrightarrow (x \Leftrightarrow z) \wedge o \Leftrightarrow u \vee g$$

- Weiter umformen (jeweils Ausdruck mit 3 Variablen in KNF umwandeln):

$$o \wedge$$

$$\neg u \vee x \wedge \neg u \vee \neg y \wedge u \vee \neg x \vee y \wedge$$

$$\neg g \vee x \vee \neg z \wedge \neg g \vee \neg x \vee z \wedge g \vee x \vee z \wedge g \vee \neg x \vee \neg z \wedge$$

$$\neg o \vee u \vee g \wedge o \vee \neg u \wedge o \vee \neg g$$

# Fortsetzung Beweis

- Gesamtkonstruktion offenbar in Polynomialzeit möglich
  - behaupten:  $H$  erfüllbar gdw.  $3H$  erfüllbar
1. Ist  $\beta$  erfüllende Belegung für  $H$ , so lässt sich diese zu erfüllender Belegung  $\beta^*$  für  $3H$  fortsetzen mit  $\beta^*(x_{H^*}) = \text{wert}(H^*, \beta)$ .
  2. Ist  $\beta$  erfüllende Belegung von  $3H$ , so erfüllt  $\beta$  auch  $H$ , insbesondere ist  $\beta(x_{H^*})$  gerade  $\text{wert}(H^*, \beta)$



## Als Kontrast: 2SAT liegt in P

Sei  $H$  2Sat-Ausdruck der Länge  $n$ . Also kommen max.  $n$  Variablen vor. Es gibt max.  $(2n)^2$  verschiedene Klauseln mit 2 Literalen.

Bilden zu den Klauseln in  $H$  Cut-Abschluss.

Cut-Regel, angewendet auf max. zweielementige Klauseln, liefert max. zweielementige Klausel

→ Abschlussbildung in P

## 2. Beispiel: SET COVER ist NP-vollständig

- SET COVER: Geg.: Menge  $M$ , Mengensystem  $\mathcal{M} \subseteq \wp(M)$  und Zahl  $k$ .

Frage: Gibt es eine Teilmenge  $\mathcal{M}'$  von  $\mathcal{M}$  mit max.  $k$  Elementen  
derart, dass  $\bigcup \mathcal{M}' = M$ ?

Satz: SET COVER ist NP-vollständig

Beweis: Reduzieren 3SAT auf SET COVER, d.h.

Gegeben: KNF über  $n$  Variablen mit  $m$  Klauseln

Brauchen: Mengensystem und Zahl (in Polyzeit bestimmbar)  
derart, dass diese überdeckbar ist gdw. KNF erfüllbar

# Konstruktion

- Idee 1: Menge = Menge derjenigen Klauseln, die dadurch wahr werden, dass ein bestimmtes Literal wahr wird (also  $2^n$  Mengen)
- Idee 2: sorgen dafür, dass für jede Variable genau eine der zu  $x$  bzw.  $\neg x$  gebildeten Klauselmengen enthalten sein muss.
- Ausführung:
- Grundmenge =  $\{1, \dots, m, m+1, \dots, n\}$
- Mengensystem  $\mathcal{M} = \{M_i \mid i=1\dots n\} \cup \{M'_i \mid i=1\dots n\}$  mit
  - $M_i = \{j \mid x_i \text{ kommt in Klausel } j \text{ vor}\} \cup \{m+i\}$
  - $M'_i = \{j \mid \neg x_i \text{ kommt in Klausel } j \text{ vor}\} \cup \{m+i\}$
- $k = n$
- Offenbar: Konstruktion in P

# Begründung

- Sei KNF erfüllbar. Wähle zu einer erfüllenden Belegung  $\beta$   
 $M' = \{M_i \mid \beta(x_i) = W\} \cup \{M'_i \mid \beta(x_i) = F\}$ . ( $\rightarrow$  genau  $n$  Mengen)
- Da zu jedem  $i$   $M_i$  oder  $M'_i$  enthalten, sind Elemente  $m+1 \dots n$  überdeckt
- Da jede Klausel mind. ein wahres Literal enthalten muss sind Elemente  $1 \dots m$  ebenfalls überdeckt
- Sei  $M'$  überdeckend mit max.  $n$  Elementen. Da jede Menge genau eines der Elemente  $m+1 \dots n$  enthält, hat  $M'$  genau  $n$  Elemente, und zwar für jedes  $i$  entweder  $M_i$  oder  $M'_i$ .
- Bilden Belegung  $\beta$  mit  $\beta(x_i) = W$  gdw.  $M_i \in M'$ .
- Nach Konstruktion sind bei dieser Belegung alle Klauseln erfüllt.

### 3. Beispiel: CLIQUE ist NP-vollständig

- CLIQUE: geg. ungerichteter Graph  $[V, E]$  und Zahl  $k$ .
- Frage: Gibt es eine Knotenmenge  $V'$  mit  $|V'| \geq k$  und  $V' \times V' \subseteq E$ ?
- Satz: CLIQUE ist NP-vollständig
- Beweis: Reduzieren 3SAT auf CLIQUE.
- Idee: Knoten = Vorkommen von Literalen in Klauseln
- Kanten = angrenzende Knoten könnten gleichzeitig wahr sein
- Clique = erfüllende Belegung

# Ausführung

- geg.: KNF mit  $m$  Klauseln
- oBdA: in jeder Klausel *genau* 3 Literale (ggf. eins kopieren)
- sei  $x_{ij}$  das in Klausel  $i$  an Position  $j$  stehende Literal
- $V = \{1, \dots, m\} \times \{1, 2, 3\}$
- $E = \{ [(i,j), (p,q)] \mid i \neq p \text{ und } x_{ij} \neq \neg x_{pq} \}$
- $k = m$  ... Konstruktion offensichtlich in  $P$
- sei KNF erfüllbar  $\rightarrow$  mindestens 1 Literal pro Klausel wahr  
 $\rightarrow$  diese alle paarweise verbunden  $\rightarrow$  Clique mit  $m$  Elementen
- Sei Clique der Größe mind.  $m$  gegeben  $\rightarrow$  beteiligte Knoten haben paarweise verschiedene 1. Komponenten  $\rightarrow$  Clique enthält pro Klausel genau einen Knoten, die beteiligten Literale sind verträglich  $\rightarrow$  erfüllende Belegung

## 4. Beispiel: VERTEX COVER ist NP-vollständig

- VERTEX COVER: Gegeben: ungerichteter Graph  $[V, E]$  und  $k$
- Frage: Gibt es  $V' \subseteq V$  mit max.  $k$  Elementen, dass jede Kante Quelle oder Ziel in  $V'$  hat (Knotenüberdeckung)
- Satz: VERTEX COVER ist NP-vollständig
- Idee: Reduzieren CLIQUE auf VERTEX COVER.
- Konstruktion: Bilde Graph  $[V, (V \times V) \setminus E]$  (Komplementärgraph)
- Behauptung:  $[V, E]$  hat eine Clique  $V'$  mit  $k$  Knoten gdw.  $[V, (V \times V) \setminus E]$  Knotenüberdeckung mit  $|V| - k$  Elementen hat (nämlich  $V \setminus V'$ )

## 5. Beispiel: KNAPSACK ist NP-vollständig

- KNAPSACK: Gegeben Menge, für jedes Element Gewicht und Wert, und Zahlen  $G$  und  $W$ . Frage: Gibt es eine Teilmenge, deren Gewichtssumme  $\leq G$  und Wertsumme  $\geq W$  ist?
- Satz: KNAPSACK ist NP-vollständig
- Beweis: reduzieren 3SAT auf KNAPSACK (sogar auf Spezialfall, wo Gewicht und Wert jeweils gleich sind sowie  $G$  und  $W$  gleich sind)
- Idee:
  - Summieren für jede Klausel Zahl der erfüllten Literale
  - Sorgen dafür, dass nie Literal und seine Negation summiert werden
  - Gesamtsumme kann nur erreicht werden, wenn jede Klausel mit wenigstens 1 in die Summe eingeht



# Ausführung

- Gesamtsumme: 44...411...10 kg (mx4, nx1)
- Elemente:
  - für jedes  $i = 1 \dots n$ :  $a_i = 10^i + \sum_{x_i \text{ kommt in Klausel } j \text{ k mal vor}} k 10^{n+j}$
  - für jedes  $i = 1 \dots n$ :  $b_i = 10^i + \sum_{\neg x_i \text{ kommt in Klausel } j \text{ k mal vor}} k 10^{n+j}$
  - für jedes  $j = 1 \dots m$ :  $c_j = 10^{n+j}$
  - für jedes  $j = 1 \dots m$ :  $d_j = 2 \cdot 10^{n+j}$

# Begründung

- Zu erfüllender Belegung  $\beta$  summiere die  $a_i$  für  $\beta(x_i) = W$  und die  $b_i$  für  $\beta(x_i) = F$ . Summe im vorderen Bereich hat mind. 1 in jeder Ziffer, kann durch  $c_j$  und/oder  $d_j$  zu je 4 ergänzt werden.
- Sei Auswahl an Elementen gegeben, die Summe exakt realisiert. Also für alle  $i$  genau eins der  $a_i$  bzw.  $b_i$  dabei  
→ Belegung mit  $\beta(x_i) = w$  gdw.  $a_i$  dabei.
- Eine „4“ im vorderen Bereich kann nicht allein durch die  $c_j$  und  $d_j$  erreicht werden, also müssen alle Klauseln erfüllt sein.

## 6. Beispiel: PARTITION

- PARTITION: geg: nat. Zahlen  $a_1, \dots, a_k$
- Frage: gibt es Teilmenge  $J \subseteq \{1, \dots, k\}$  derart, dass  $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$
- Satz: PARTITION ist NP-vollständig
- Reduzieren (eben betrachteten Spezialfall von) KNAPSACK auf PARTITION

# Ausführung

- Geg.: Aufgabe aus KNAPSACK, d.h. Werte (=Gewichte)  $a_1, \dots, a_k$  und Gesamtwert (=Gesamtgewicht)  $b$ .
- Sei  $m = \sum_{i=1 \dots k} a_i$ .
- Betrachten Partitionsaufgabe  $a_1, \dots, a_k, m-b+1, b+1$
- Sei  $J \subseteq \{1, \dots, k\}$  Lösung von KNAPSACK, also  $\sum_{i \in J} a_i = b$ .
- Dann ist  $\sum_{i \in J} a_i + (m-b+1) = m+1$  und  $\sum_{i \notin J} a_i + (b+1) = m+1$ , also  $J \cup \{m-b+1\}$  Lösung von PARTITION.
- Sei  $J$  Lösung von Partition. Dann muss  $J$  genau eins der Elemente  $b+1, m-b+1$  enthalten. Sei o.b.d.A  $m-b+1$  in  $J$ .
- Dann ist  $\sum_{i \in J} a_i + (m-b+1) = \sum_{i \notin J} a_i + (b+1)$ , also
- $\sum_{i \in J} a_i = b$ , d.h. Lösung der KNAPSACK-Aufgabe

# Weitere NP-vollständige Probleme

- 0/1-INTEGER LINEAR PROGRAMMING:
- TSP:
- COLOR:
- HAMILTON CIRCUIT:

Fazit: Viele Planungs-, Optimierungs-,  
Ressourcenverwaltungsprobleme sind NP-vollständig → hohe  
praktische Relevanz der Klasse NPV!

# Umgang mit NP-vollständigen Entscheidungsproblemen

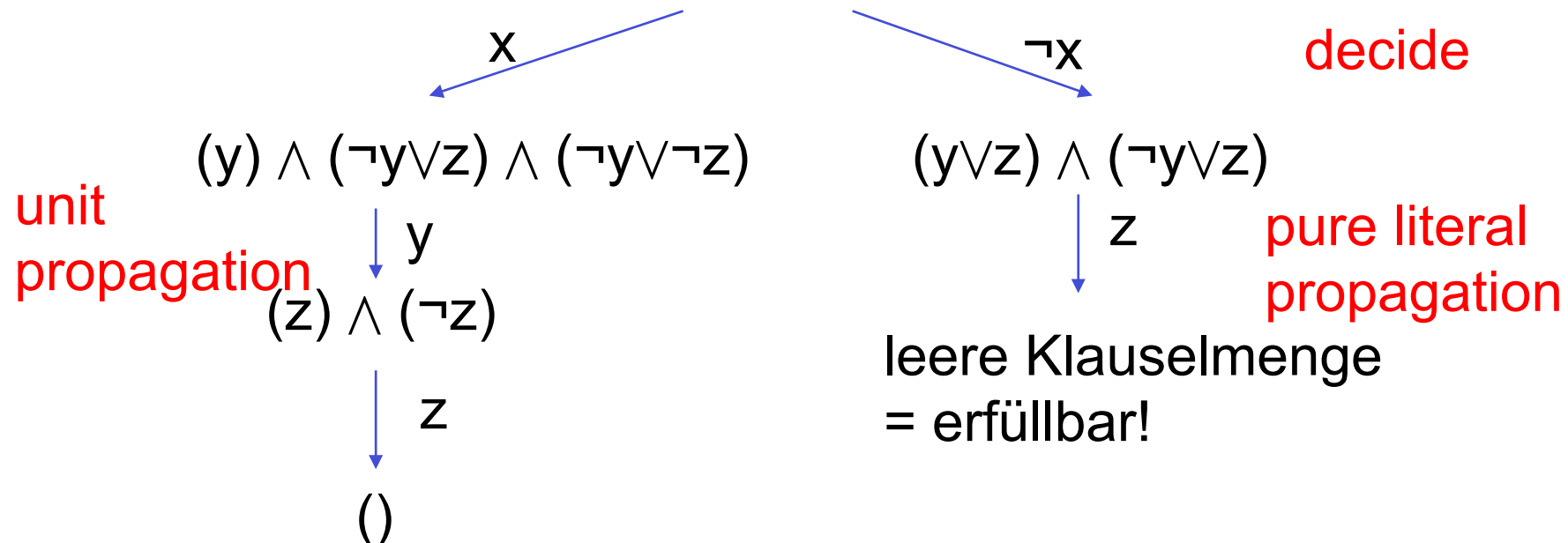
Am Beispiel von SAT

# SAT-Solver für CNF

(Suche nach erfüllender Belegung)

Ausgangspunkt: Algorithmus von Davis-Putnam aus den 60ern

$$(x \vee y \vee z) \wedge (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$



leere Klausel = Konflikt

→ **Backtracking** zur letzten offenen Entscheidung

# Davis-Putnam-Algorithmus

DP(K)            K – Klauselmenge

IF  $K = \emptyset$  THEN RETURN erfüllbar

IF  $() \in K$  THEN RETURN unerfüllbar

IF  $\kappa = ( \dots \vee 1 \vee \dots ) \in K$  THEN RETURN DP( $K \setminus \{\kappa\}$ )            Tautologie

IF  $(0 \vee \dots \vee 0 \vee [\neg]x_i) \in K$  THEN RETURN DP( $K / x_i \leftarrow 1[0]$ )            unit

IF K enthält Literal I, aber nicht seine Negation THEN

    RETURN DP( $K \setminus \{\kappa \mid I \in \kappa\}$ )            pure literal

choose i

IF DP( $K / x_i \leftarrow 1$ ) = SAT THEN RETURN SAT            decide/

RETURN DP( $K / x_i \leftarrow 0$ )            backtrack



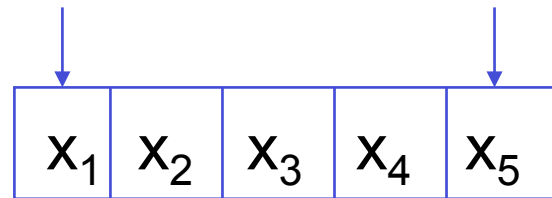
# 1. Trick: Schnelles Finden von Unit Klauseln

Unit-Klausel = Klausel mit  $n-1$  Literalen auf 0, 1 Literal auf ?

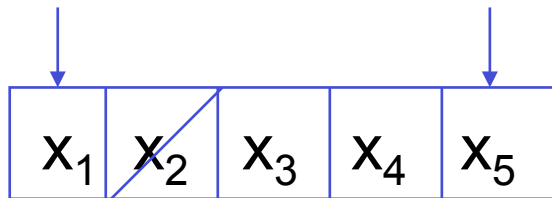
Lösung (z.B. in *chaff*): Pro Klausel 2 Beobachter, die auf ?-Literale zeigen

Solange beide Beobachter auf ?-Literale zeigen, wird Klausel nicht angerührt

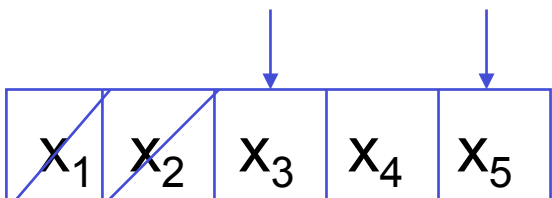
# Beispiel



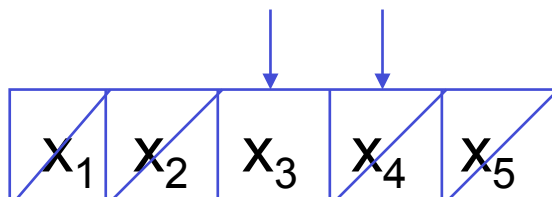
$$\underline{x_2=0} \rightarrow$$



$$\underline{x_1=0} \rightarrow$$

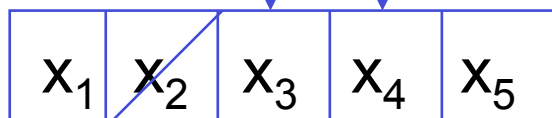


$$\underline{x_{4,5}=0} \rightarrow$$



backtrack

$$\underline{x_{4,5} := ? \quad x_1 := 1} \rightarrow$$



Beim Backtracking bleiben Beobachter, wo sie sind  
→ konst. Zeit

Beobachter “wandern” zu selten gesetzten Literalen  
→ Klausel muß seltener besucht werden  
→ “Lernstrategie”

## 2. Trick: Konfliktanalyse

Konflikt = leere Klausel = Literal, für das sowohl 0 als auch 1 propagiert wird

Idee: “Grund” für den Widerspruch wird explizit als Klausel zur Klauselbasis hinzugefügt      “Lernen”

- “denselben Fehler nicht noch mal machen”
- Suchraum einschränken

“Grund” wird durch Analyse eines “Implikationsgraphen” generiert, der Ursache/Wirkung von Wertzuweisungen dokumentiert

### 3. Trick: Nichtchronologisches Backtracking, Zufällige Restarts

Auf Grund der Konfliktanalyse nicht die letzte, sondern eine frühere Entscheidung rückgängig machen

Von Zeit zu Zeit einfach von vorn anfangen, und (randomisiert) an anderen Variablen verzweigen

→ Wissen über bisherige Suche in Form der gelernten Klauseln verfügbar

→ Möglichkeit, komplizierten Teilen des Suchraums zu entfliehen

## 4. Trick: Heuristiken für Entscheidungen

*Grasp*: Setze die Variable, die in den meisten offenen Klauseln vorkommt

*Chaff*: Setze Variable, die häufig in gelernten Klauseln vorkommt

# Der Stålmarck-Algorithmus

patentierter Algorithmus, um Tautologie/Erfüllbarkeit  
*beliebiger* Formeln zu entscheiden

weicht vom üblichen Decide/Deduce/Backtrack ab

# Datenstruktur: Triplets

1. Beseitige Negationen: de Morgan +  $\neg x \Leftrightarrow (x \Rightarrow \perp)$

nehme für jede Teilformel  $\psi$  von  $\phi$  eine frische Variable  $x_\psi$  her, substituiere jede Benutzung von  $\psi$  durch  $x_\psi$ , und nehme  $x_\psi \Leftrightarrow \psi$  in die Formelmenge auf (wie bei 3SAT!)

Beispiel:

$$p \Rightarrow (q \Rightarrow p)$$

wird zu

$$\begin{aligned} x_1 &\Leftrightarrow (q \Rightarrow p) \\ x_2 &\Leftrightarrow (p \Rightarrow x_1) \end{aligned}$$

Triplet:  $\langle \text{var} \rangle \Leftrightarrow (\langle \text{var} \rangle \langle \text{op} \rangle \langle \text{var} \rangle)$

# Start des Algorithmus

Tautologie:

Setze Top-Level-Variable auf 0, suche konsistente Belegung

SAT:

Setze Top-Level-Variable auf 1, suche konsistente Belegung



# Simple Rules

$$\frac{0 \Leftrightarrow (y \Rightarrow z)}{y/1 \quad z/0}$$

$$\frac{x \Leftrightarrow (y \Rightarrow 1)}{x/1}$$

$$\frac{x \Leftrightarrow (1 \Rightarrow z)}{x/z}$$

$$\frac{x \Leftrightarrow (0 \Rightarrow z)}{x/1}$$

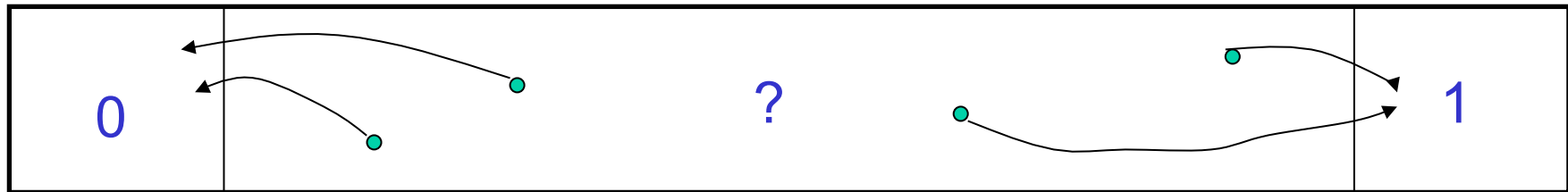
$$\frac{x \Leftrightarrow (y \Rightarrow 0)}{x/\neg y}$$

$$\frac{x \Leftrightarrow (x \Rightarrow z)}{x/1}$$

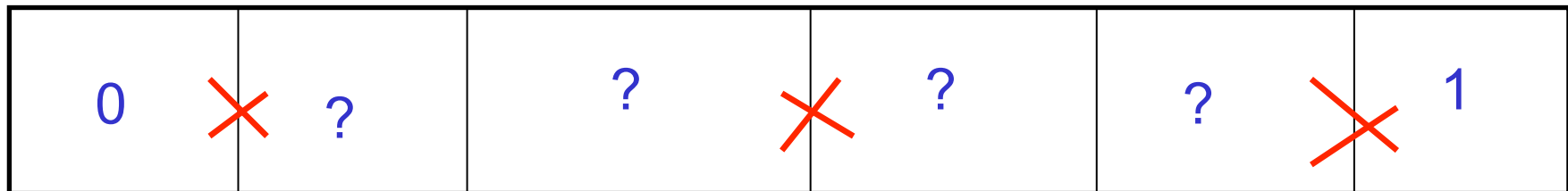
$$\frac{x \Leftrightarrow (y \Rightarrow y)}{x/1}$$


# Wirkung der simple rules

naiv:

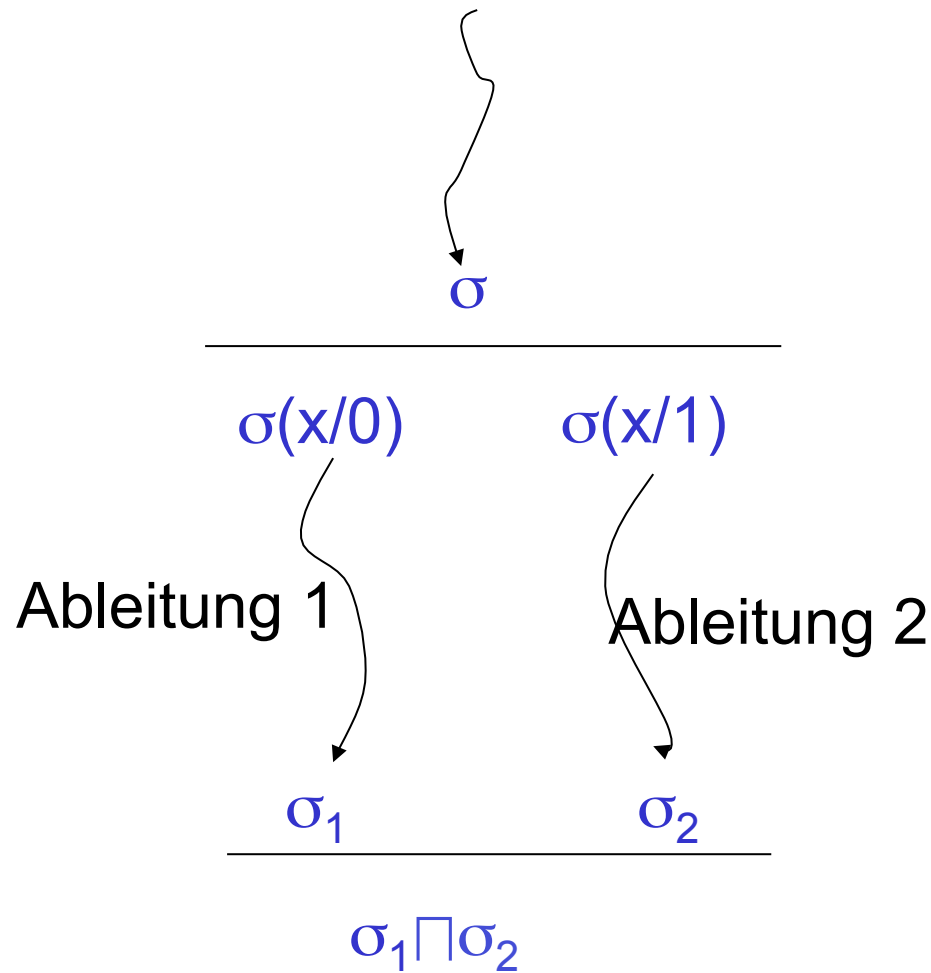


Stålmärck:



  
 $x/y, x/\neg y$

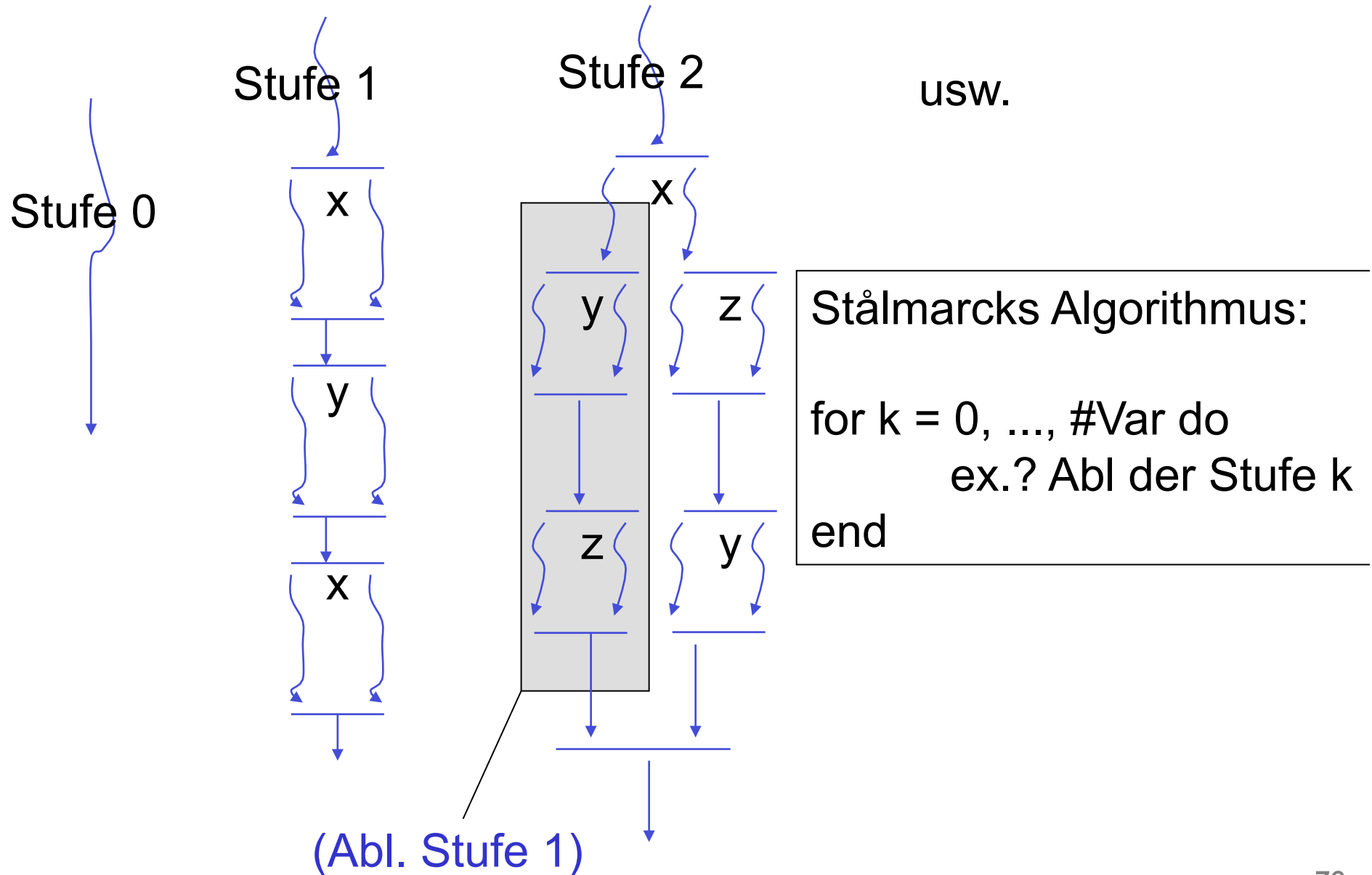
# Dilemma rule



Zusammenführen der  
Zweige  $\rightarrow$  Vermeide  
redundante Arbeit in  
verschiedenen Zweigen

= Eine der Subst., falls andere zu Konflikt führt;  
= diejenigen Subst., die in beiden Zweigen gleich sind, sonst

# Ableitungsstufen



# Härte von Formeln

$\phi$  ist k-hard, wenn Algorithmus frühestens in Runde k terminiert

$\phi$  ist k-leicht, wenn Algorithmus spätestens in Runde k terminiert

Die meisten Formeln aus industriellem Kontext sind 1-leicht

2-hard = too hard

Laufzeit von Stålmarcks Algorithmus hängt viel mehr von der Härte des Ausdrucks ab als von seiner Länge

# Zusammenfassung SAT-Checker

arbeiten hervorragend auf Problemen “mit Struktur”

heute: über 1.000.000 Variablen

derzeitig Nummer 1: randomisierte Verfahren

neben den vollständigen Checkern ex. noch viele unvollständige Verfahren

# Umgang mit NP-vollständigen Optimierungsproblemen

Am Beispiel von TSM

# Travelling Salesman

- Geg.: Menge (von Städten), Kosten für jedes Paar von Städten
- Ges.: Rundreise mit minimalen Kosten
- symmetrisch, falls  $\text{Kosten}(A \rightarrow B) = \text{Kosten}(B \rightarrow A)$



# 1. Verfahren: Branch & Bound

- Idee: Suchraum zerlegen in Teile  $T_1, \dots, T_n$ , die jeweils wenigstens eine Lösung enthalten  
→ Branch
- Für jeden Teil obere und untere Schranken für eine beste Lösung bestimmen (obere Schranke: z.B. eine bereits probierte Lösung aus diesem Teil)  
→ Bound
- Suche nur in denjenigen Teilen fortsetzen, deren untere Schranke nicht über der oberen Schranke eines anderen Teils liegt  
→ Truncate

Führt zu optimaler Lösung, schränkt Suchraum ein

# Beispiel TSM:

- Starte mit Pfad „x0“ für irgendeine Stadt
- Allgemeiner Schritt: geg.: Pfad „x1 x2 x3 ... xn“
- Branch: Probiere alle verbleibenden Möglichkeiten für  $x_{n+1}$
- Bound: Untere Schranke: Aktuelle Kosten (Kosten [x1,x2] + ... + Kosten [xn-1,xn]) plus Hälfte der Kosten der jeweils min. noch möglichen ausgehenden Kanten für noch nicht verlassene Städte plus Hälfte der Kosten der jeweils min. noch möglichen eingehenden Kante für noch nicht betretene Städte
- Obere Schranke: Erstbeste Vervollständigung der Teillösung

## 2. Verfahren: Graduelle Verbesserung

- Idee: Starte mit irgendeiner Lösung
- führe iterativ lokale Verbesserungen durch

→ führt zu Näherungslösung

- Verbesserungsschritt:
- Entferne  $k$  Städte und ergänze restliche Fragmente um optimale Anbindung dieser Städte (per systematischem Probieren ermittelt)
- mit festem  $k$ : polynomieller Aufwand

### 3. Verfahren: Greedy-Strategie

- Greedy=gierig
- Nimm iterativ die jeweils billigste noch mögliche Kante hinzu
- Kann für gewisse Klassen von Graphen auch zur global schlechtesten Lösung führen

## 4. Verfahren: Simulated Annealing

- Nachbildung eines physikalischen Effekts:
  - Metallgitter spiegeln Energielevel der beteiligten Atome wider
  - durch Erhitzen (=Energiezufuhr = randomisierte Bewegung in großen Radien)
  - und langsame Abkühlung (= schrittweise Energiereduktion = Einschränkung des Wirkungsradius randomisierter Bewegung)
  - wird eine Gitterstruktur mit nahezu minimalen Energielevel erreicht
- Ähnlich: genetische Algorithmen (Nachbildung biologischer Optimierung durch Mutation & Selektion)

# Nutzung zusätzlicher Annahmen

- Beispiel: Dreiecksungleichung
- Betrachten Graphen, wo für alle Knoten  $x, y, z$  gilt:  
 $\text{Kosten}(x, y) + \text{Kosten}(y, z) \geq \text{Kosten}(x, z)$

Es gibt einen polynomiellen Algorithmus, der für solche Graphen in Polynomialzeit eine Lösung des symmetrischen TSM berechnet, deren Kosten max. doppelt so hoch sind wie die der optimalen Lösung

# Idee: Nutzen minimal spannenden Baum

- Was ist das?
  - (ungerichteter) Graph  $[V, E]$  heißt Baum, falls er keine Kreise enthält und alle Knoten verbindet
  - $[V, E']$  heißt minimal spannender Baum eines zusammenhängenden Graphen  $[V, E]$ , falls  $[V, E']$  Baum ist und die Summe der Kosten der Kanten in  $E'$  minimal ist

# Berechnung eines minimal spannenden Baumes

- Sei  $V=\{v_1, \dots, v_n\}$
- Starte mit Mengensystem  $\{\{v_1\}, \dots, \{v_n\}\}$  und leerer Kantenmenge  $E'$
- Schritt: haben Mengensystem  $M$ .
  - Suche billigste Kante  $e$ , für die Quelle und Ziel in unterschiedlichen Elementen  $M_1, M_2$  von  $M$  enthalten ist
  - Vereinige  $M_1$  mit  $M_2$
  - Füge  $e$  zu  $E'$  hinzu
- Abbruch:  $M = \{\{v_1, \dots, v_n\}\}$



# Algorithmus berechnet minimal spannenden Baum

1.  $[V, E']$  verbindet alle Knoten: Man kann sich überlegen, dass in jedem Zwischenschritt jeweils genau alle Elemente innerhalb einer Menge in  $M$  verbunden sind
2.  $[V, E']$  enthält keine Kreise: Sonst müssten beim Einfügen der letzten Kante des Kreises bereits alle Knoten innerhalb einer Menge verbunden gewesen sein.
3. Kosten von  $[V, E']$  sind minimal: Die billigste Kante zwischen zwei disjunkten Knotenmengen ist jeweils Teil der optimalen Lösung

# Lösung des TSM mit Dreiecksungleichung

- Berechne minimal spannenden Baum
- Dieser liefert Rundreise mit Umwegen
- Generiere echte Rundreise durch Überspringen bereits besuchter Städte
- Kosten der echten Rundreise  $\leq 2$  Kosten des Baumes
- optimale Rundreise -1 Kante ist spannender Baum, also mindestens so teuer wie minimal spannender Baum

# Zufallsabhängige Berechnungen

Grundlage: Zufallsabhängige TM:

- kann in einem Zustand „Münze werfen“, d.h. eines von mehreren Zeichen auf der Basis einer festen Zufallsverteilung auf das Band schreiben

Klasse RP (randomisiert polynomiell) entscheidet Problem M

- Jeder Lauf terminiert in Polyzeit
- Kein Lauf akzeptiert ein  $x \notin M$
- $x \in M$  wird mit Wahrscheinlichkeit von mind. 0.5 akzeptiert

Klasse ZPP (zero error, probabilistic, polynomial)

- Jeder Lauf für  $x \in M$  akzeptiert
- Kein Lauf für  $x \notin M$  akzeptiert
- *Erwartungswert* für Laufzeit durch Polynom abschätzbar

# Aussagen

- $P \subseteq ZPP \subseteq RP \subseteq NP$
- Für  $M \in RP$  und beliebiges  $c < 1$  ex. ein polynomieller zufallsabhängiger Algorithmus, bei dem
  - kein  $x \notin M$  akzeptiert
  - Wahrscheinlichkeit für Akzeptanz eines  $x \in M > c$
  - Idee: Original“experiment“ hinreichend oft wiederholen

# Letztes Thema zu P-NP

- Gibt es Probleme, die in NP, aber weder in NPV noch in P liegen?
- (Wenn  $P = NP$  wäre, natürlich nicht)
- Beispiel Graphisomorphie
  - liegt in NP (siehe nächste Folie)
  - es gibt (derzeit) keinen polynomiellen Algorithmus
  - es gibt (derzeit) keinen Beweis der NP-Vollständigkeit

# Graphisomorphie

- geg.: Graphen  $[V, E]$  und  $[V', E']$
- Frage: Gibt es eine bijektive Abb.  $f: V \rightarrow V'$  derart, dass  $[v_1, v_2] \in E$  gdw.  $[f(v_1), f(v_2)] \in E'$ ?
- Liegt in NP: rate  $f$ , Überprüfung effizient realisierbar

# Jenseits von NP

## PSPACE

- enthält P und NP
- es gibt PSPACE-vollständige Probleme (jedes PSPACE-Problem ist in Polynomialzeit auf dieses reduzierbar)
- Beispiel: quantifizierte aussagenlogische Ausdrücke
  - Aussagenlogische Formeln mit folgenden Erweiterungen:
    - $\exists x H$  ( $= H[x \leftarrow 0] \vee H[x \leftarrow 1]$ )
    - $\forall x H$  ( $= H[x \leftarrow 0] \wedge H[x \leftarrow 1]$ )
  - Naheliegende Überführung nach SAT nicht in P, weil sich pro Quantor Länge der Formel verdoppelt.

# Zusammenfassung Komplexität

- Man sollte Probleme als schwer lösbar erkennen und nachweisen können
  - nicht trivial, weil manchmal doch (effiziente) Lösungen existieren
    - Eulerkreis vs. Hamiltonkreis
    - Lineare Optimierung: rationale vs. ganzzahlige Lösungen
- Man sollte mit unlösbaren/schwer lösbaren umgehen können
  - Spezialfälle, eingeschränkte Lösungsgüte, Techniken (z.B. branch & bound)



# Beweistechniken

- „Durchbruchstechniken“
  - Satz von Cook sollte man kennen (widergeben können), nicht notwendigerweise aktiv beherrschen (für neue Probleme)
- Reduktion/Polynomialzeitreduktion
  - sollte man aktiv beherrschen

# Weiterer Nutzen

- Kenntnis von Grenzen gestattet
  - bessere Einschätzung des Herausforderungsgrades von Problemen,
  - begründete Würdigung von Lösungsversuchen,
  - begründete Erwartungshaltungen an Werkzeuge
  - realistische eigene Zielsetzungen
- Reduktionstechnik schult kreatives Denken
  - Setzen Problem X ein, um Problem Y zu lösen (Denken „out of the box“) →kann man immer brauchen!
- Haben enge Beziehungen zwischen sehr verschiedenen Problemen kennengelernt
  - Recycling von bewährten Lösungsideen möglich

# In der Vorlesung nicht/kaum behandelt

- Programme, die etwas anderes tun als Werte auszurechnen (interaktive Programme, reaktive Programme, eingebettete Systeme)
- massive Parallelisierung
- Hardwarekomplexität
- Weitere Komplexitätsklassen, innerhalb und außerhalb von NP
- u.v.a.m.

# Formale Sprachen

# Worum geht es?

## 1. Verarbeitung von Sprachen

- Sprache = Formulierung von Probleminstanzen
- Sprache = Träger von Information
- Sprache = Beschreibung von Systemverhalten

## 2. Beschreibung von Systemverhalten

- jenseits von Sprachen
- Modellierung und Analyse von Systemen

# Motivation

## 1. Verarbeitung von Sprachen:

- Sprache = Grundlage für Kommunikation
  - Mensch-Mensch ... natürliche Sprachen,  
künstliche Sprachen,  
Formelsprachen, ...
  - Mensch-Maschine ... Programmiersprachen,  
Markup-Sprachen,  
Kommandosprachen,  
Ausgabeformate, ...
  - Maschine-Maschine ... Dateiformate,  
Protokolle, ...

# Motivation

## 1. Verarbeitung von Sprachen

Anwendung:

- Compilerbau
- Internet
- Datenbanken
- Nutzerschnittstellen
- Dateien einlesen

C, C++, JAVA, ...

HTML

SQL, XML

SHELL

XML

# Motivation

## 2. Beschreibung von Systemverhalten

- formale Modelle für Systeme, formale Semantik
- Beschreibung & Analyse von Phänomenen
- Studium von Operationen auf Systemen  
(Komposition, Verfeinerung, ...)



# Motivation

## 2. Beschreibung von Systemverhalten

Anwendung:

- Model Checking (Fehlersuche jenseits von Testen)
- Statische Programmanalyse, -optimierung
- Design leistungsfähiger Operationen
- Korrektheit per Konstruktion

...

# Plan

## Kapitel 1: Sprachen

*Träger von Information, Beschreibung von Abläufen*

## Kapitel 2: Automaten

*Verarbeiter von Sprachen, Systembeschreibung*

## Kapitel 3: Reguläre Sprachen und endliche Automaten

*Suchmuster, Lexik*

## Kapitel 4: Kontextfreie Sprachen und Kellerautomaten

*Formel-, Programmier-, Markup-Sprachen*

## Kapitel 5: Jenseits von Sprachen, jenseits von Automaten

*Beschreibung interagierender, verteilter Systeme*

# 1.0 Sprache: Grundlegendes

<i>Alphabet</i>	$X$	endliche Menge $X = \{x_1, \dots, x_n\}$
<i>Wort</i>	$w$	endliche Folge von Buchstaben $w : \{0, \dots, n-1\} \rightarrow X$
<i>Wortlänge</i>	$ w $	Anzahl der Buchstaben (n)
<i>Wortmenge</i>	$X^*$	die Menge <i>aller</i> Wörter über Alphabet $X$
<i>Sprache</i>	$L$	eine Menge von Wörtern über $X$ $L \subseteq X^*$
<i>Satz von L</i>		ein Wort $w \in L$

# 1.1 Sprache: Aspekte

## *Lexik*

Welchen Wortschatz (hier: Buchstaben) hat eine Sprache?

→ Compilerbau, Kapitel 3

## *Syntax*

Welche Wörter gehören zur Sprache?

Wie sind Sätze der Sprache aufgebaut?

→ Compilerbau, Kapitel 4

## *Semantik*

Welche Information enthält ein Satz?

→ Semantik von Programmiersprachen, Compilerbau

## *Pragmatik*

Wie / zu welchem Zweck wird Sprache genutzt?

→ In den jeweiligen Anwendungsgebieten

# Syntax: Wozu Struktur?

## *1. Sprechen*

Satz der Sprache konstruieren

*Synthese*

## *2. Hören*

Satz der Sprache dekonstruieren

*Analyse*

## *3. Verstehen*

Einem Satz seine Semantik zuordnen

# Beispiel

Subjekt

Prädikat

Objekt

Fruit flies like banana

Subjekt Prädikat Adverbialkonstruktion

Fruit flies like banana

→ Schlüssel zur Sprache: Grammatik

## 1.2 Grammatik

= Lehre/Beschreibung von der Struktur einer Sprache  
(z.B. Deutsche Grammatik im Duden)

Für natürliche Sprachen: schwierig

Für formale Sprachen: einfach

# Formale Grammatiken: Geschichte

*Noam Chomsky* (\*1928)

1957: Buch *Syntactic Structures*

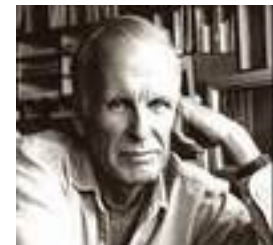
Ziel: Formale Beschreibung der  
Syntax natürlicher Sprachen



*John Backus* (1924-2007), *Peter Naur* (\*1928)

1960: Programmiersprache ALGOL 60

Ziel: Formale Beschreibung der Syntax  
einer Programmiersprache





# Formale Grammatik: Idee

## Unterscheidung

- Symbole der Sprache → Terminalsymbole (= X)
- Metasprachliche Einheiten → Hilfssymbole

Transformationsregeln zur Bildung von Sätzen aus der  
allgemeinsten metasprachlichen Einheit *Satz*  
(Satzsymbol)

# Beispiel: deutsch

X = {das, kleine, mädchen, pflückt, fröhlich,  
einen, blumenstrauß,...}

H = {S, SBJ, ART, PRR, NOM, ADJ, SBT, PRÄ, OBJ,  
VRB, ADV, ...}

R = S → SBJ PRR	ART → das
SBJ → ART NOM	ART → einen
NOM → ADJ SBT	ADJ → kleine
PRR → PRÄ OBJ	SBT → Mädchen
PRÄ → VRB ADV	SBT → Blumenstrauß
OBJ → ART SBT	VRB → pflückt
	ADV → fröhlich, ...

# Beispiel: deutsch

R = S → SBJ PRR	ART → das
SBJ → ART NOM	ART → einen
NOM → ADJ SBT	ADJ → kleine
PRR → PRÄ OBJ	SBT → Mädchen
PRÄ → VRB ADV	SBT → Blumenstrauß
OBJ → ART SBT	VRB → pflückt
	ADV → fröhlich, ...

S → SBJ PRR → ART NOM PRR → das NOM PRR →  
das ADJ SBT PRR → das kleine mädchen PRR → ...  
→ Das kleine mädchen pflückt fröhlich einen blumenstrauß

Ableitung

# Grammatik formal

$G = [X, H, R, s]$  mit

- $X, H$  disjunkte Alphabete (*Terminalsymbole, Hilfssymbole*)
- $R \subseteq ((X \cup H)^* \setminus X^*) \times (X \cup H)^*$  (*Regeln/Produktionen*), endlich!
- $s \in H$  (*Startsymbol*)

Bemerkung zu  $R$ : linke Seite  $\neq$  leer, mindestens ein Hilfssymbol

# Ableitbarkeit formal

*Ableitbarkeit in einem Schritt („ $\rightarrow$ “):*

$w \rightarrow w'$  gdw. ex.  $w_1, w_2, u, v$  mit

- $w = w_1 u w_2$
- $w' = w_1 v w_2$
- $[u, v] \in R$

*Ableitbarkeit (beliebig,  $\rightarrow^*$ ) = reflexiv/transitive Hülle von  $\rightarrow$*

( $w$  kann durch endlich häufige Anwendung (0 mal eingeschlossen) mittels  $\rightarrow$  in  $w'$  überführt werden)

*Die von  $G$  erzeugte Sprache,  $L_G = \{w \mid s \rightarrow^* w, w \in X^*\}$*

# Beispiel

$G = [\{0, \dots, 9\}, \{S, \text{Null}, \dots, \text{Neun}\}, \{S \rightarrow \text{Null Eins} \dots \text{Neun},$   
     $\text{Null Eins} \rightarrow \text{Eins Null},$   
     $\text{Null Zwei} \rightarrow \text{Zwei Null},$   
     $\dots$   
     $\text{Eins Null} \rightarrow \text{Null Eins},$   
     $\dots$   
     $\text{Neun Null} \rightarrow \text{Null Neun},$   
     $\text{Null} \rightarrow 0, \dots, \text{Neun} \rightarrow 9\}, S]$

$L_G = \{0123456789, 0123456798, \dots, 9876543210\}.$

# Grammatik-Sprachen sind aufzählbar

Erinnerung:  $L_G$  *aufzählbar* = es gibt ein *berechenbares*  
 $f: \mathbb{N} \rightarrow X^*$  mit  $L_G = \{f(0), f(1), \dots\}$ .

Berechnungsidee:

$f(k)$ : für aufsteigendes  $i$ , Liste aller in max.  $i$   
Ableitungsschritten herstellbaren Zeichenreihen  
erzeugen, bis mind.  $k$  Elemente aus  $X^*$  dabei sind.  
 $k$ -tes Element aus  $X^*$  ausgeben.

# Aufzählbare Sprachen haben Grammatik

Beweis*idee*: Arbeit einer Turingmaschine als Ableitungsprozess einer Grammatik nachbilden

Konkret:  $M = [Z, X, \Gamma, \delta, z_0, z_f]$   
 $\rightarrow G = [X, Z \cup \{s, s', L, R\}, R, s]$  mit

$$R = R_1 \cup R_2 \cup R_3$$

(eine beliebige natürliche Zahl in Binärkodierung aufs Band ): z.B.  $L11001z_0R$   
 $R_1 = \{s \rightarrow L1s', s' \rightarrow 1s', s' \rightarrow 0s', s' \rightarrow z_0R, s \rightarrow L0z_0R\}$

(M rechnen lassen)

$$R_2 = \{azc \rightarrow bcz' \mid z \in Z, a, b, c \in \Gamma, \delta(z, a) = [z', b, +1]\} \cup \\ \{az \rightarrow z'b \mid z \in Z, a, b \in \Gamma, \delta(z, a) = [z', b, -1]\} \cup \\ \{az \rightarrow bz' \mid z \in Z, a, b, c \in \Gamma, \delta(z, a) = [z', b, 0]\} \cup \{z_f \rightarrow \varepsilon\}$$

(Kopfbewegung über linken/rechten Rand hinweg beachten)

$$R_3 = \{Lz \rightarrow L\Box z \mid z \in Z\} \cup \{zR \rightarrow z\Box R \mid z \in Z\} \cup \{La \rightarrow a, aR \rightarrow a \mid a \in \Gamma\} \cup \{L\Box \rightarrow L, \\ \Box R \rightarrow R\}$$



## 1.3 Kontextsensitive Grammatiken

Kritik beliebiger Grammatiken: Ersetzungsregeln spiegeln nicht die Idee „syntaktischer Einheiten“ wider

→ Idee: nur ein einziges Symbol wird ersetzt

$G = [X, H, R, s]$  heißt *kontextsensitiv*, falls alle Regeln die Form  $u h w \rightarrow u v w$  mit  $h \in H$ ,  $u, w \in (X \cup H)^*$ ,  $v \in (X \cup H)^+$  haben.

Ausnahme:  $s \rightarrow \varepsilon$  (dann aber  $s$  in keiner Regel auf rechter Seite)

Interpretation: „ $v$  entspricht der syntaktischen Einheit  $h$ , wenn es im *linken Kontext*  $u$  und *rechten Kontext*  $w$  auftritt“

$L$  heißt *kontextsensitiv*, falls es eine kontextsensitive Grammatik  $G$  mit  $L = L_G$  gibt.

# Beispiel für kontextsensitive Grammatik

$G = [\{a,b,c\}, \{S,X,Y,Z\}, \{S \rightarrow XYZ, X \rightarrow a, X \rightarrow b, aY \rightarrow ab, bY \rightarrow ba, YZ \rightarrow Yc\}, S]$

$L_G = \{abc, bac\}$

# Kontextsensitive Sprachen sind entscheidbar

Erinnerung: Entscheidbar =

$f: X^* \rightarrow \{0,1\}$  mit  $f(x) = 1$  gdw.  $x \in L$   
ist berechenbar

Algorithmische Idee für  $f$ : Ableitungen sind **längenmonoton**,  
d.h. wenn  $v \rightarrow w$ , dann  $|v| \leq |w|$  (bis auf die Ausnahme  $s \rightarrow \varepsilon$ )

Also: Für Eingabe  $w$  alle (endlich vielen) Ableitungen prüfen,  
die Länge  $|w|$  nicht überschreiten)

# Sprachen zu lngenmonotonen Grammatiken sind kontextsensitiv

- lngenmonoton: Regeln  $(u,v)$  mit  $|u| \leq |v|$  (auer  $(s,\epsilon)\dots$ )

- Beweisidee:

ersetze  $(u_1\dots u_m, v_1\dots v_n)$  durch

$(u_1\dots u_m, H_1 u_2 \dots u_m)$

$(H_1 u_2 \dots u_m, H_1 H_2 \dots u_m)$

...

$(H_1 \dots H_{m-1} u_m, H_1 \dots H_m)$

$(H_1 \dots H_m, v_1 H_2 \dots H_m)$

...

$(v_1 \dots v_{m-2} H_{m-1} H_m, v_1 \dots v_{m-1} H_m)$

$(v_1 \dots v_{m-1} H_m, v_1 \dots v_n)$

# Es gibt entscheidbare Sprachen, die nicht kontextsensitiv sind

Idee: Diagonalisierung:

Auflistung aller kontextsensitiven Grammatiken über  $\{a,b\}$ ,

Auflistung aller Wörter über  $\{a,b\}$ , alles berechenbar realisierbar

$L : w_i \in L \text{ gdw. } w_i \notin L_{G_i}$

$L$  entscheidbar, da alle  $L_{G_i}$  entscheidbar

$L \notin CS$ , weil  $L$  sonst bei irgendeinem  $k$  in Liste vorkommen würde, dann aber Widerspruch mit  $w_k$ !

# Kontextsensitive Sprachen werden durch nichtdeterministische linear beschränkte Automaten akzeptiert

- linear beschränkter Automat = Turing-Maschine, die lediglich diejenigen Bandzellen nutzt, auf denen die Eingabe steht ( $\rightarrow$  hat Platzkomplexität  $O(n)$ )
- Idee:
  - wähle nichtdeterministisch Position
  - wähle nichtdeterministisch Regel
  - ersetze, wenn es geht, rechte durch linke Regelseite
  - akzeptiere, wenn nur noch  $s$  übrig bleibt

# Sprachen, die durch nichtdeterministische linear beschränkte Automaten akzeptiert werden, sind kontextsensitiv

- Konstruieren längenmonotone Grammatik
    - (1) erzeuge irgendeine Eingabe + passende Startkonfiguration:  
Ergebnis:  $x_1 \dots x_n \text{ „/“ } z_0 x_1 x_2 \dots x_n$
    - (2) Regeln für Zustandübergänge: (analog Folie 21)  
z.B.  $(z_0 x_i, x_j z_0)$
    - (3) Bei Akzeptierung (Zustand  $z_f$ ) alles rechts vom „/“ aufräumen
- nur solche Zeichenketten enthalten keine Hilfssymbole mehr

## 1.4 Kontextfreie Grammatiken

$G$  ist *kontextfrei*, falls  $R \subseteq H \times (X \cup H)^*$

(also kontextsensitiv, mit leerem linken/rechten Kontext)

$L$  ist *kontextfrei*, falls sie durch eine kontextfreie Grammatik beschrieben werden kann.

Bedeutung: Grundlage für Definition von Programmiersprachen, Markup-Sprachen, Formelsprachen,...

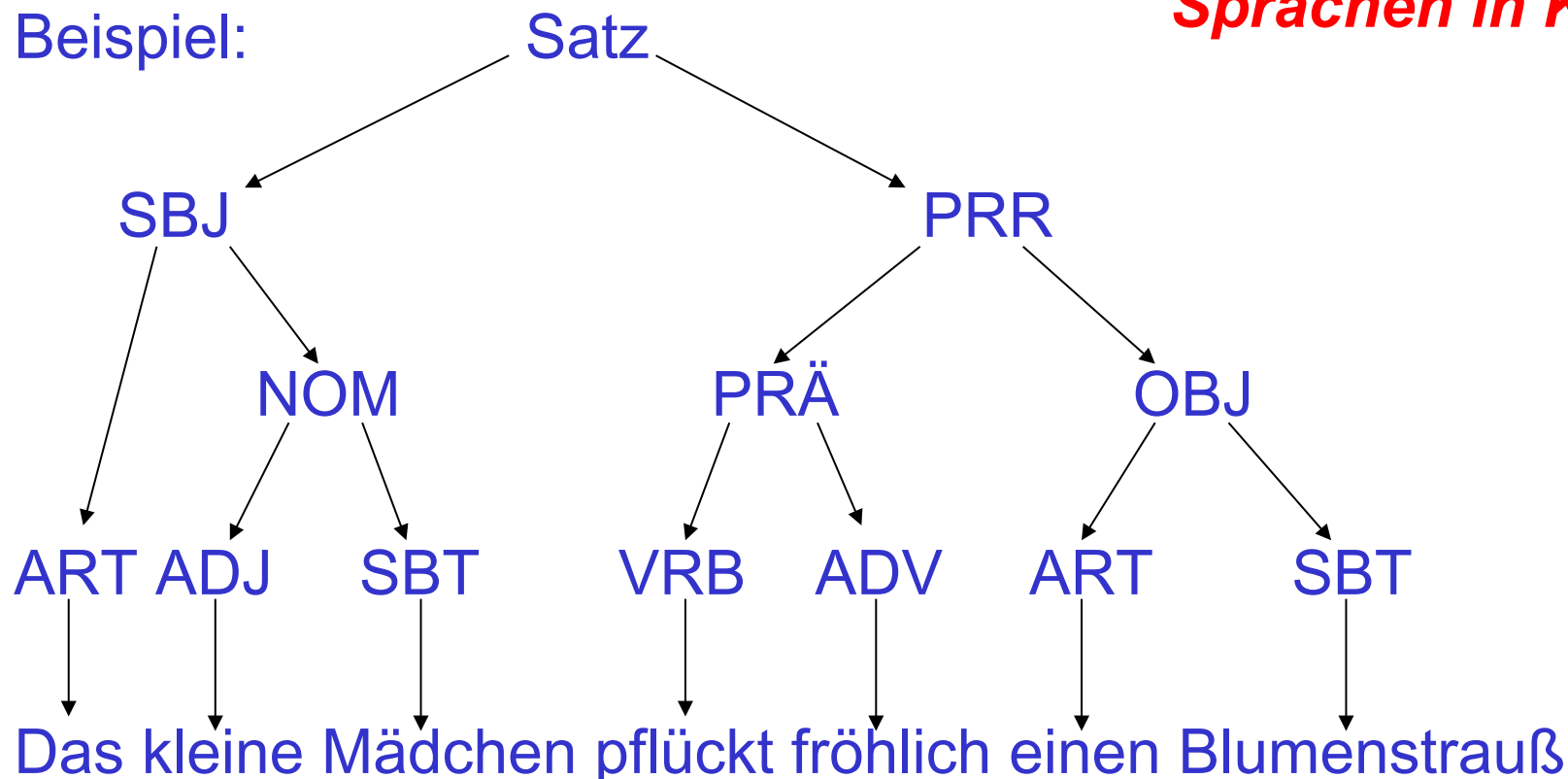


# Ableitungsbaum

Ersetzungen verschiedener Hilfssymbole in einem Wort sind *unabhängig* voneinander → Ableitungsprozess als Baum (*Ableitungsbaum*) darstellbar!

**Mehr zu kontextfreien Sprachen in Kap. 4!**

Beispiel:



## 1.5 Reguläre Sprachen

Eine kontextfreie Grammatik  $G$  heißt *rechtslinear*, falls auf der rechten Seite von Regeln höchstens das letzte Zeichen ein Hilfssymbol ist

Sprache  $L$  heißt *regulär*, falls sie durch eine rechtslineare Grammatik beschrieben werden kann

Bedeutung: Grundlage für

- viele Kommandosprachen,
- Suchmuster,
- Lexik von Programmiersprachen

...

***Mehr zu regulären  
Sprachen in Kap. 3!***

## 1.6 Sprachklassen

Klassen von Sprachen:

ENU - durch beliebige Grammatik beschreibbar  
(rek. aufzählbar)

DEC - entscheidbare Sprachen

CS - kontextsensitive Sprachen

CF - kontextfreie Sprachen

REG - reguläre Sprachen

FIN - endliche Sprachen

# Inklusionen

Satz:  $\text{FIN} \subset \text{REG} \subset \text{CF} \subset \text{CS} \subset \text{DEC} \subset \text{ENU}$

Beweise:

$\text{FIN} \subseteq \text{REG}$ :  $G$  mit Regeln  $s \rightarrow w_1, \dots, s \rightarrow w_n$

$\text{REG} \subseteq \text{CF} \subseteq \text{CS} \subseteq \text{ENU}$ : wegen Struktur der Regeln;  
knifflig:  $\varepsilon$  (wir verzichten hier)

$\text{CS} \subseteq \text{DEC}$ : Folie 123

$\text{DEC} \subset \text{ENU}$ : VL Berechenbarkeit & Komplexität

$\text{CS} \subset \text{DEC}$ : Folie 125

$\text{FIN} \neq \text{REG}$ :  $\{a^i \mid i \in \mathbb{N}\}$ ;  $G$  mit Regeln  $s \rightarrow as, s \rightarrow \varepsilon$

$\text{REG} \neq \text{CF} \rightarrow \text{Kap. 3}$

$\text{CF} \neq \text{CS} \rightarrow \text{Kap. 4}$

# Die Chomsky-Hierarchie

ENU = Typ 0

CS = Typ 1

CF = Typ 2

REG = Typ 3

(FIN = Typ 4)

(auch: Chomsky-Typ  $i$ )

$i > j \rightarrow \text{Typ } i \subset \text{Typ } j$

Typen allein durch Einschränkungen für R definiert

# Die Chomsky-Hierarchie und Akzeptierungsformalismen

ENU = Typ 0 → Aufzählbarkeit durch Turing-Maschinen

CS = Typ 1 → Akzeptierung durch nichtdeterministische  
linear beschränkte Automaten

CF = Typ 2 → Akzeptierung durch nichtdeterministische  
Kellerautomaten (Kap. 4)

REG = Typ 3 → Akzeptierung durch endliche Automaten  
(Kap. 3)

(FIN = Typ 4)

## 1.7 Operationen auf Sprachen

Sprachen sind Mengen, also  $\cap$ ,  $\cup$ , Komplement

Verkettung:  $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

Iteration:  $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$

$$L^0 = \{\varepsilon\}$$

$$L^{i+1} = L^i L$$

# Beispiel

$$\{aa,bb\}\{cc,dd\} = \{aacc,aadd,bbcc,bbdd\}$$

$$\{aa,bb\}^* = \{\varepsilon,aa,bb,aaaa,aabb,bbba,bbbb,aaaaaa,\dots\}$$



# Abschluss bzgl. Operationen

Jede der Klassen REG, CF, CS, ENU ist abgeschlossen bzgl. der Operationen Vereinigung, Verkettung, Iteration

Beweisideen I: jeweils Grammatik erweitern um neues Startsymbol  $s'$  und Regeln

- (Vereinigung:)  $s' \rightarrow s_1, s' \rightarrow s_2$
- (Verkettung, alle außer REG:)  $s' \rightarrow s_1 s_2$
- (Iteration, alle außer REG:)  $s' \rightarrow \varepsilon, s' \rightarrow s'', s'' \rightarrow s s'', s'' \rightarrow s$

Beweisideen II

- Verkettung (REG): Jede Regel  $h \rightarrow w$  ( $w \in X_1^*$ ) aus  $G_1$  ersetzen durch  $h \rightarrow w s_2$
- Iteration (REG): neue Regel  $s \rightarrow \varepsilon$ , jede Regel  $h \rightarrow w$  ( $w \in X_1^*$ ) ersetzen durch  $h \rightarrow w s$

## Kapitel 2

# Automaten

## 2.1 Was ist das?

- Sehr allgemeines Modell für diskrete dynamische Systeme
- Eine abstrakte Maschine, die sich nach festgelegten Regeln (Programm) verhält
- Beispiel: Turing-Maschine

# Anwendung

- Verarbeitung von Sprachen
- Studium von Grenzen der Informatik
- Strukturierung dynamischer Programmabläufe, z.B. Spiele-Engines
- Systemanalyse
- Spezifikation, z.B. Protokolle
- ...

# Bestandteil 1: Zustand

Zustand = Momentaufnahme aller für das weitere Verhalten *wesentlichen* Größen  
(für Automat: diskrete Größen)

Bei Turing-Maschine hieß das: Konfiguration

Bei Automaten: Zustand = Element einer Menge,  
der Zustandsmenge  $Z$

## Bestandteil 2: Ein- / Ausgabe

Ein- und Ausgabe = Modellierung der Wechselwirkung  
mit der Umgebung

Ein- und Ausgaben werden zu Folgen  
zusammengefasst = Wörter

→ Zu einem Automat gehören Alphabete  $X$  und  $Y$

(Turing-Maschinen sind Modell eines geschlossenen  
Systems, also ohne Ein- und Ausgabe)

## Bestandteil 3: Zustandsübergang

- Abhängig vom Zustand
- Abhängig von Eingabe
- Überföhrungsfunktion  $\delta$ : erzeugt neuen Zustand

Deterministisch:  $\delta: Z \times X \rightarrow Z$

oder

Nichtdeterministisch :  $\delta: Z \times X \rightarrow \wp(Z)$

- Erzeugt Ausgabe:  $\lambda : Z \times X \rightarrow Y$

## Bestandteil 4 (optional): Start und Ende

Anfangszustand:  $z_0 \in Z$

Menge von Endzuständen:  $F \subseteq Z$

Arbeit eines Automaten beginnt in  $z_0$ , und kann in einem Zustand aus  $F$  enden.

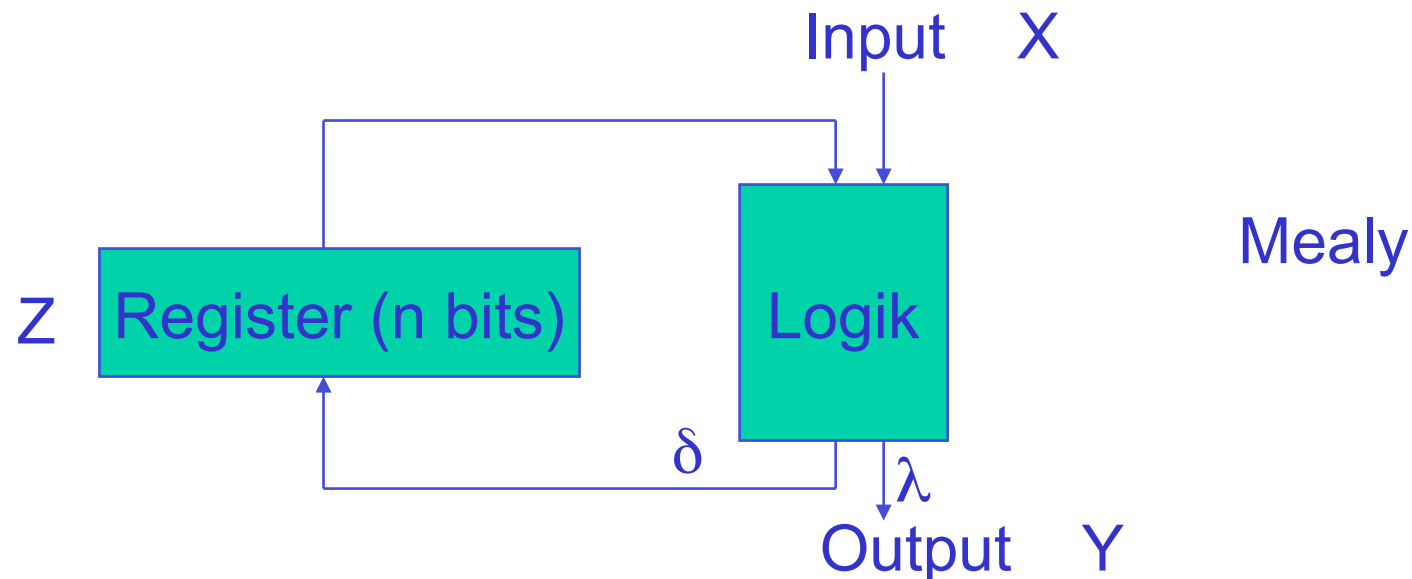


# Klassifikation

- Mealy-Automat: Ein- und Ausgabe wie erklärt, keine Endzustände
- Moore-Automat:  $\lambda : Z \rightarrow Y$
- Akzeptor: Keine Ausgabe, dafür Akzeptierungszustände

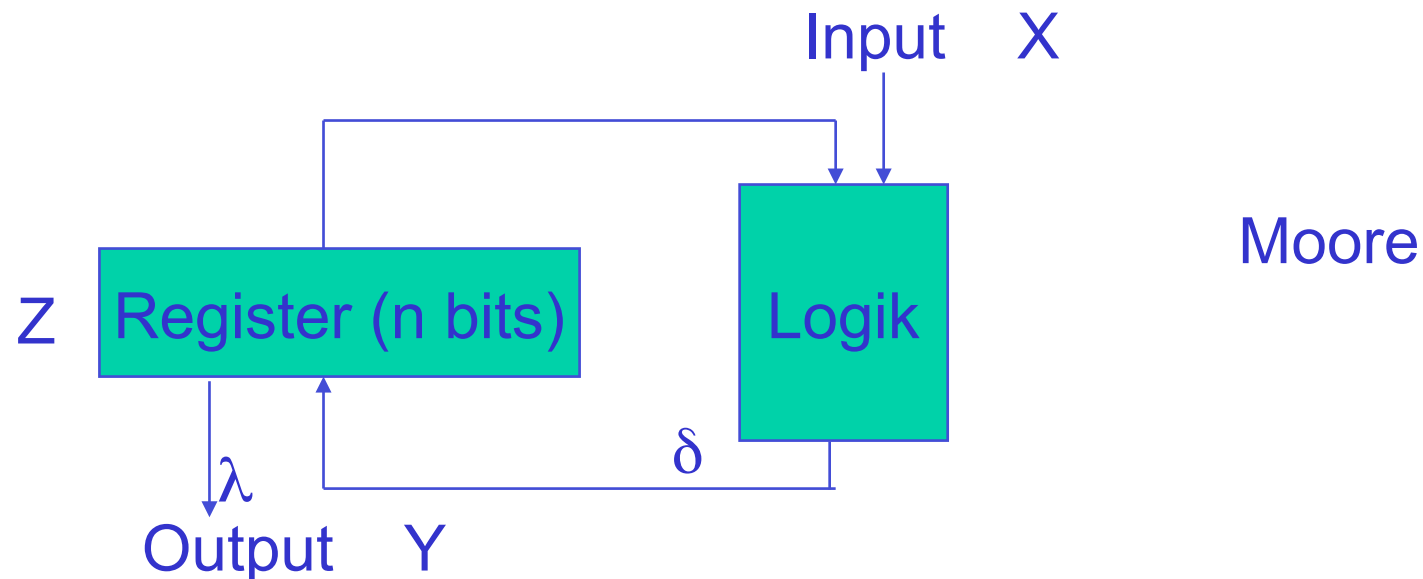
# Anwendung

- Deterministische Melay- und Moore-Automaten definieren Wortfunktionen  $\phi: X^* \rightarrow Y^*$
- Einsatz z.B. zur Formalisierung von diskreten Schaltkreisen



# Anwendung

- Deterministische Melay- und Moore-Automaten definieren Wortfunktionen  $\phi: X^* \rightarrow Y^*$
- Einsatz z.B. zur Formalisierung von diskreten Schaltkreisen



# Anwendung

- Akzeptoren definieren Sprachen
- Einsatz z.B. zur Verarbeitung von Suchmustern und zur lexikalischen Analyse von Programmiersprachen

# Formal: Mealy-Automat

Deterministischer Automat  $A = [Z, X, Y, \delta, \lambda, z_0]$  mit

- $Z$  Menge
- $z_0 \in Z$
- $X, Y$  Alphabete
- $\delta : Z \times X \rightarrow Z$
- $\lambda : Z \times X \rightarrow Y$

Lauf eines Automaten  $A$ , gesteuert durch Eingabefolge

$w = x_0x_1\dots x_n$ :

*die Folge  $z_0z_1\dots z_{n+1}$  von Zuständen mit  $z_{i+1} \in \delta(z_i, x_i)$*

Die von  $A$  definierte (partielle) Wortfunktion  $F_A: X^* \rightarrow Y^*$

$F_A(x_0\dots x_n) = \lambda(z_0, x_0), \dots, \lambda(z_n, x_n)$  wobei  $z_0\dots z_{n+1}$

der durch  $x_0, \dots, x_n$  gesteuerte Lauf von  $A$  ist.

# Eigenschaften von Wortfunktionen

Die Wortfunktion eines Automaten ist *längeninvariant*, d.h. für alle  $A, w$ :  $|w| = |F_A(w)|$

Die Wortfunktion eines Automaten ist *monoton* bzgl. der Relation  $\sqsubseteq$  („ist Anfangsstück von“), d.h. Wenn  $w \sqsubseteq w'$ , so  $F_A(w) \sqsubseteq F_A(w')$

Eine Wortfunktion  $F$  heißt **erzeugbar** bzw. **sequentiell**, falls es einen Automaten gibt mit  $F = F_A$ .

# Eine Wortfunktion ist genau dann erzeugbar, wenn sie längeninvariant und monoton ist

„ $\rightarrow$ “ siehe vorige Folie

„ $\leftarrow$ “ Sei  $F$  längeninvariant und monoton.

Nehmen  $Z = X^*$

$Z_0 = \varepsilon$ ,

$\delta(w, x) = wx$

$\lambda(w, x) = \text{letzter Buchstabe von } F(wx)$

zeigen per Induktion über  $w$ :

- Lauf gesteuert durch  $w$  endet bei  $w$
- $F_A(w) = F(w)$

A)- Lauf gesteuert durch  $\varepsilon$  endet beim Anfangszustand, also  $\varepsilon$

- $F(\varepsilon) = \varepsilon = F_A(\varepsilon)$ , weil  $\varepsilon$  einziges Wort der Länge 0 ist und  $F$  längeninvariant

S)- Vor: Lauf bei  $w$  endet bei  $w$ . Weil  $\delta(w, x) = wx$ , endet also Lauf gesteuert durch  $wx$  ebenfalls bei  $wx$

- Sei  $F_A(w) = F(w)$ . Wegen Monotonie ist  $F(w) \sqsubseteq F(wx)$ , wegen Längeninvarianz hat  $F(wx)$  genau einen Buchstaben mehr als  $F(w)$ .  $F_A(w) = F(w)$ ,  $F_A(wx)$  setzt sich zusammen aus  $F(w)$  (wegen IndVor) und dem letzten Buchstaben von  $F(wx)$  (nach Def.  $\lambda$ ). q.e.d.

# Vergleich Moore-Automat $\leftrightarrow$ Mealy-Automat

Mealy:  $\lambda: Z \times X \rightarrow Y$       Moore  $\lambda': Z' \rightarrow Y$

Zu jedem Moore-Automat existiert ein Mealy-Automat, der die gleiche Wortfunktion realisiert:  $\lambda(z,x) := \lambda'(\delta(z,x))$  für alle  $x$ ;  
neuer Anfangszustand  $z_0^*$  mit  $\delta(z_0^*,x) = z_0$  für alle  $x$  und  
 $\lambda(z_0^*,x) = \lambda'(z_0)$ .

Zu jedem Mealy-Automat existiert ein Moore-Automat, der die gleiche Wortfunktion realisiert (Voraussetzung: leerer Output erlaubt wenigstens im Anfangszustand)

$Z' := \{z_0\} \cup (Z \times Y),$   
 $\delta'([z,y],x) := [\delta(z,x), \lambda(z,x)]$   
 $\lambda'([z,y]) := y$   
 $\lambda'(z_0) := \varepsilon$



# Formal: Akzeptor

Nichtdeterministischer Automat  $A = [Z, X, \delta, z_0, F]$  mit

- $Z$  Menge
- $X$  Alphabet
- $\delta : Z \times X \rightarrow \wp(Z)$
- $z_0 \in Z$
- $F \subseteq Z$

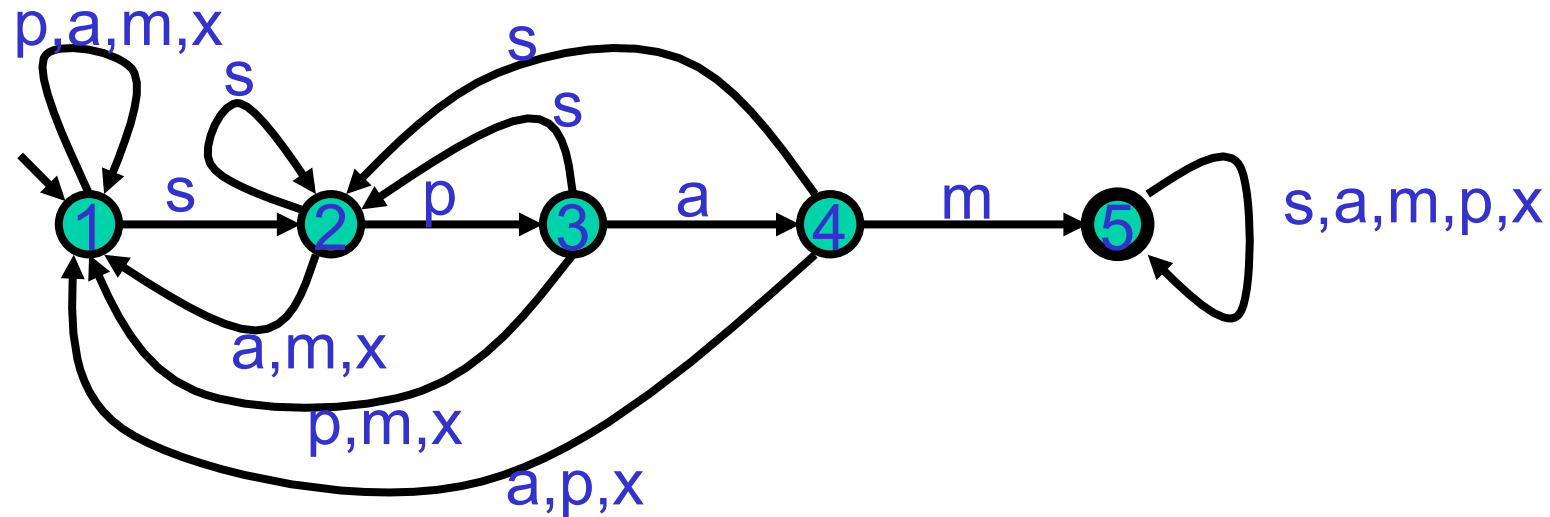
Lauf eines nichtdeterministischen Automaten  $A$ ,  
gesteuert durch Eingabefolge  $w = x_0x_1\dots x_n$ :

*eine* Folge  $z_0z_1\dots z_{n+1}$  von Zuständen mit  $z_{i+1} \in \delta(z_i, x_i)$

Die von  $A$  definierte Sprache  $L_A$ :

$= \{w \mid \text{es gibt einen Lauf von } A \text{ gesteuert durch } w, \text{ der zu einem Endzustand } (z_{n+1} \in F) \text{ f\"uhrt}\}$

# Beispiel deterministischer Akzeptor



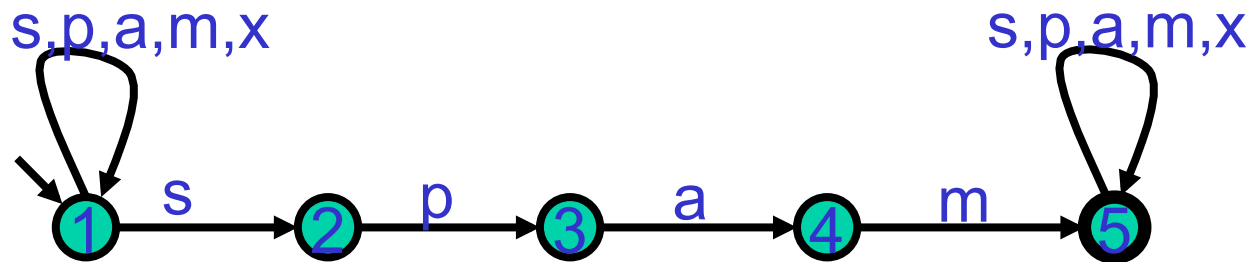
$X = \{a, m, p, s, x\}$      $Z = \{1, 2, 3, 4, 5\}$      $z_0 = 1$      $F = \{5\}$

$\delta(1, s) = 2, \delta(1, p) = 1, \dots$

$\delta(2, s) = 2, \delta(2, p) = 3, \delta(2, a) = 1, \dots$

$L_A = \{w \mid \text{in } w \text{ kommt „spam“ als Teilzeichenreihe vor}\}$

# Beispiel: Nichtdeterministischer Akzeptor



$X = \{a, m, p, s, x\}$        $Z = \{1, 2, 3, 4, 5\}$        $z_0 = 1$        $F = \{5\}$

$\delta(1, s) = \{1, 2\}$ ,  $\delta(1, p) = \{1\}$ , ...

$\delta(2, s) = \emptyset$ ,  $\delta(2, p) = \{3\}$ ,  $\delta(2, a) = \emptyset$ , ...

$L_A = \{w \mid \text{in } w \text{ kommt „spam“ als Teilzeichenreihe vor}\}$

## 2.2 Jede Sprache hat einen Akzeptor

Satz: Zu jedem Alphabet  $X$  und jeder Sprache  $L$  über  $X$  gibt es einen Akzeptor  $A$  mit  $L = L_A$ .

Beweis:

$A = [X^*, X, \delta, \varepsilon, L]$  mit  $\delta(w, x) = wx$  tut es offensichtlich.  
„Freier Automat“

Bem:  $A$  ist sogar deterministisch, aber hat meist unendlich viele Zustände;  
→ für Verarbeitung durch Maschinen ungeeignet

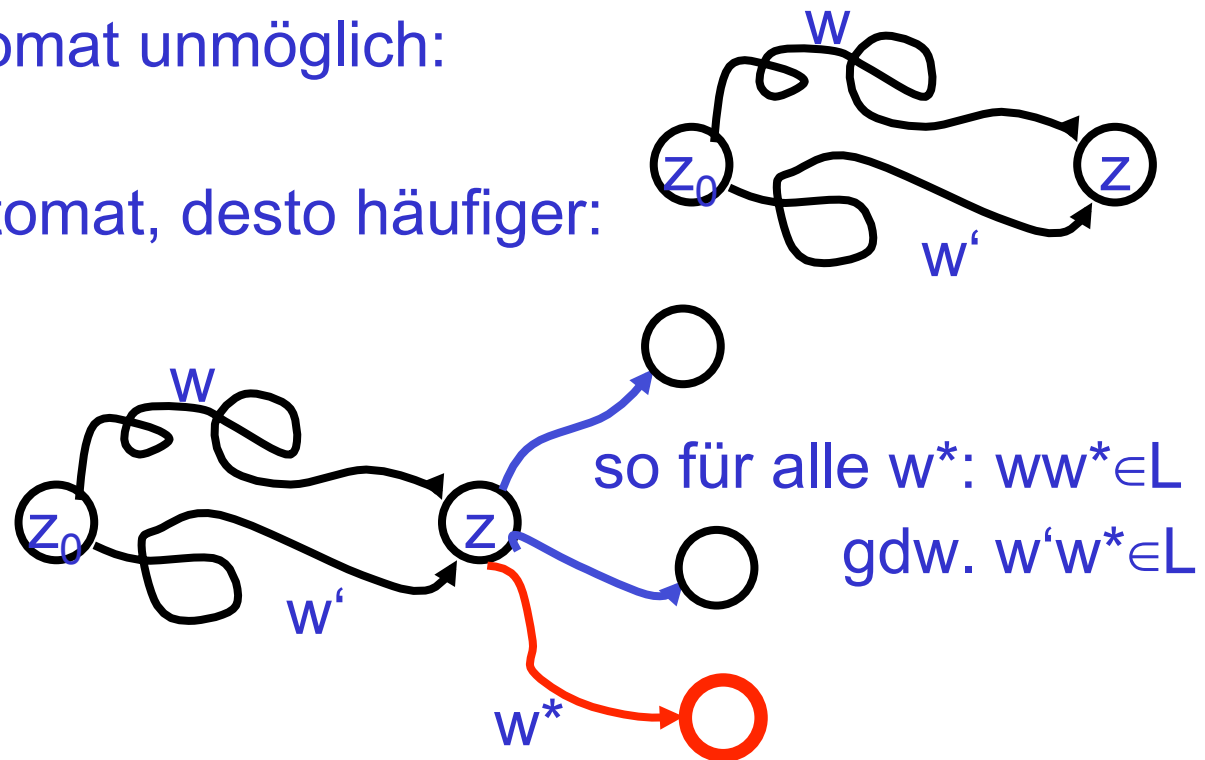
# Es geht vielleicht auch kleiner...

Suchen jetzt, für Sprache  $L$ , den kleinsten (deterministischen) Automat  $A$  mit  $L_A = L$ .

Im freien Automat unmöglich:

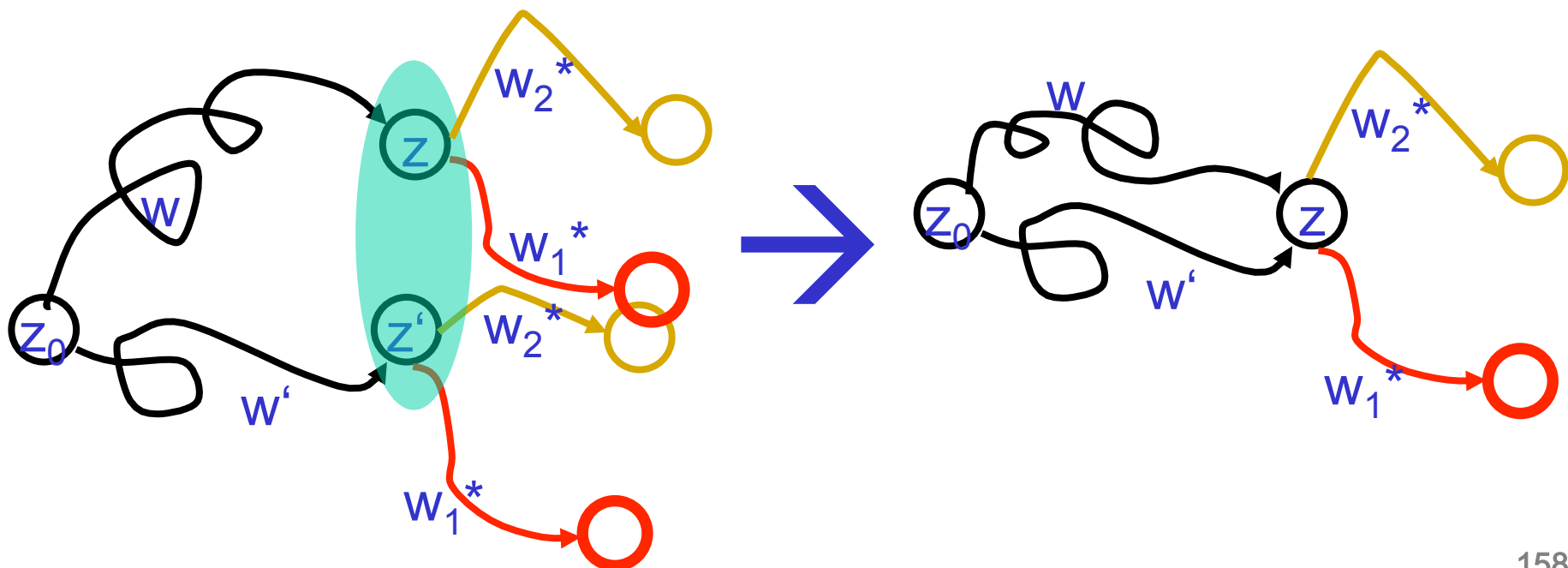
Je kleiner Automat, desto häufiger:

Aber: Wenn



## Es geht vielleicht auch kleiner...

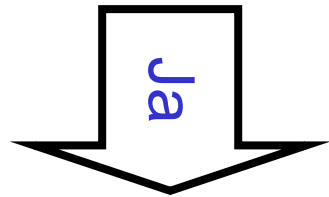
Umgekehrt: Wenn für alle  $w^*$ :  $ww^* \in L$  gdw.  $w'w^* \in L$ ,  
so können der von  $w$  erreichte und der von  $w'$  erreichte  
Zustand verschmolzen werden!



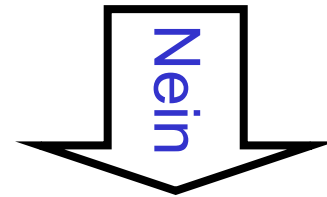
# Es geht auch etwas kleiner ...

Zusammengefasst: gegeben:  $w, w'$

gilt für alle  $w^*$ :  $ww^* \in L$  gdw.  $w'w^* \in L$  ?



die von  $w$  und  $w'$  erreichten  
Zustände *dürfen* gleich sein



die von  $w$  und  $w'$  erreichten  
Zustände *müssen* verschieden sein

→ fassen, ausgehend vom freien Automaten, alle Zustände zusammen, die wir zusammenfassen dürfen!

# Es geht auch etwas kleiner ...

Formal: Definieren Relation  $\sim$  auf  $X^*$ :

$w \sim w'$  gdw. für alle  $w^*$ :  $ww^* \in L$  gdw.  $w'w^* \in L$

$\sim$  ist Äquivalenzrelation

→ können  $X^*$  in Äquivalenzklassen zerlegen:  $X^*/\sim$

Neuer Automat:  $A^* = [X^*/\sim, X, \delta, [\varepsilon], L/\sim]$  mit  $\delta([w], x) = [wx]$ .

Aus unseren Überlegungen folgt:

$A^*$  ist der kleinste Automat mit  $L_{A^*} = L$ .



$\sim$  heißt *Nerode-Relation*



# Beispiel

$L = \{w \mid \text{in } w \text{ kommt die Zeichenreihe „spam“ vor}\}$

$\{s,p,a,m,x\}^*$  zerfällt in die 5 Klassen

$[spam] = L$

$[spa] = \{w \mid w \notin L, w \text{ endet auf „spa“}\}$

$[sp] = \{w \mid w \notin L, w \text{ endet auf „sp“}\}$

$[s] = \{w \mid w \notin L, w \text{ endet auf „s“}\}$

$[\varepsilon] = \text{alle anderen Elemente von } \{s,p,a,m,x\}^*$

mit Fortsetzung  $w^*$  in  $L$   
gdw.  $w^*$  beginnt mit „am“  
oder enthält „spam“

Resultierender Automat: siehe Folie 54

# Hilft das wirklich?

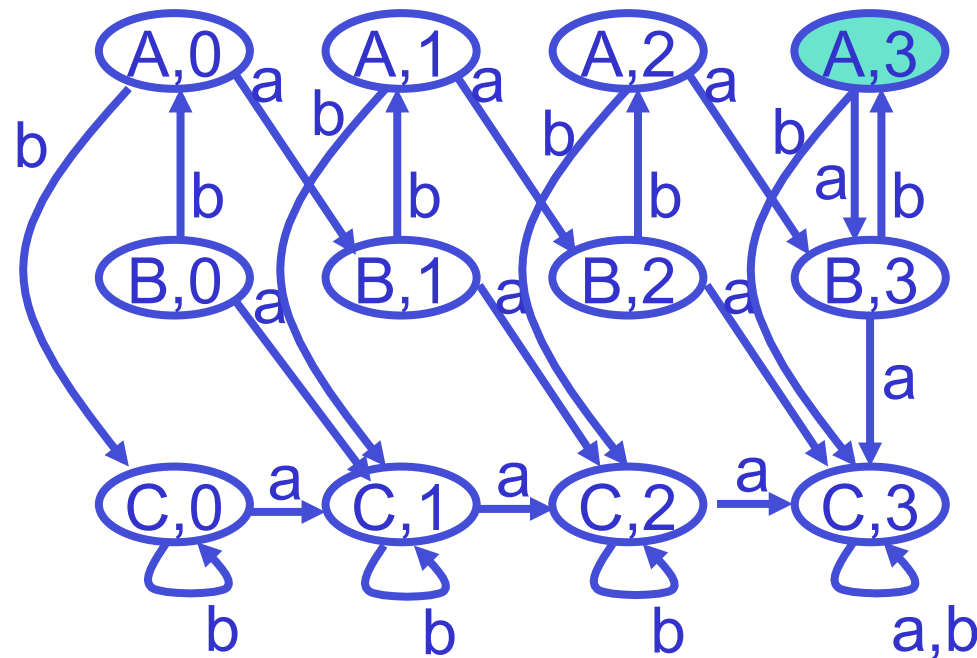
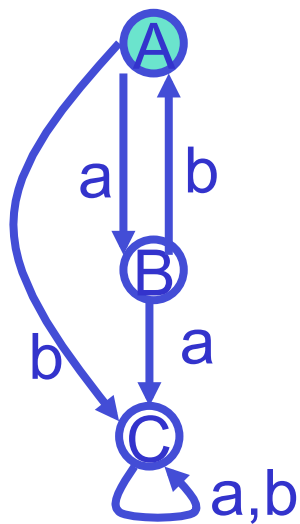
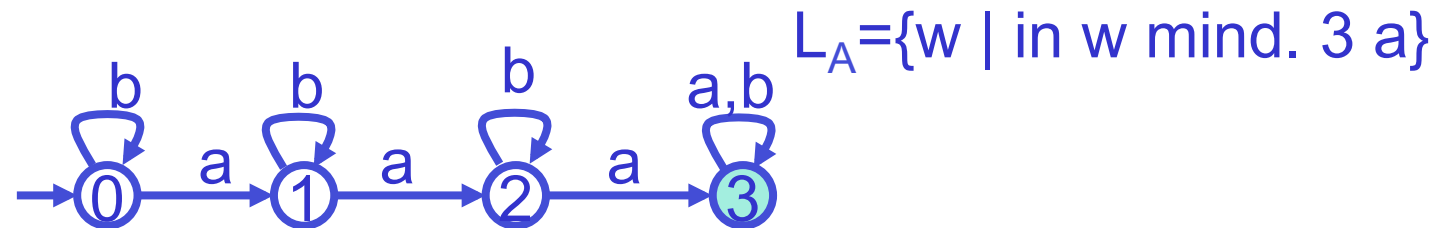
Ansatz liefert kein Rechenverfahren

→ Bessere Beherrschung des Themas für spezielle Sprachklassen → Kapitel 3,4!

## 2.3 Operationen auf Automaten

1. Komplement eines *deterministischen* Automaten:  
Ersetze  $F$  durch  $Z \setminus F$   
Resultat: neuer Automat erkennt  $\overline{L}$
2. Durchschnitt (deterministisch; geht auch nichtdet.)  
Produktautomat  $[Z_1 \times Z_2, X_1 \cap X_2, \delta', [z_{01}, z_{02}], F_1 \times F_2]$   
mit  $\delta'([z_1, z_2], x) = [\delta(z_1, x), \delta(z_2, x)]$   
Resultat: neuer Automat erkennt  $L_1 \cap L_2$
3. Vereinigung (deterministisch)  
gleiches  $\delta'$ , aber Endzustandsmenge  $(F_1 \times Z_2) \cup (Z_1 \times F_2)$

# Beispiel



$$L_{A \cap B} = \{(ab)^i \mid i \geq 3\}$$

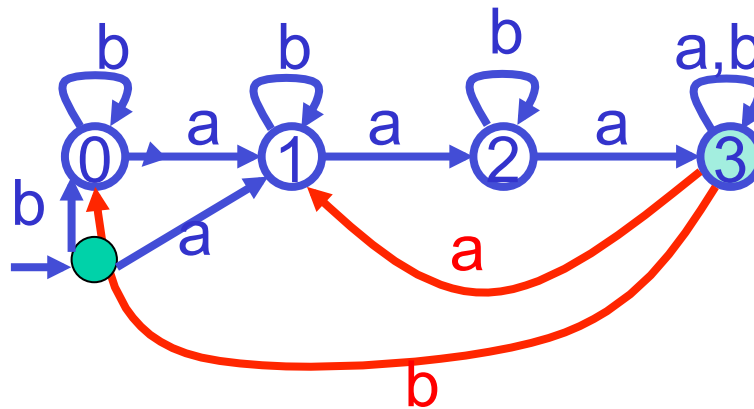
$$L_B = \{(ab)^i \mid i \in \mathbb{N}\}$$

# Operationen auf Automaten

## 4. Iteration (nichtdeterministisch)

- $z_0$  wird Endzustand
- Alle Nachfolger von  $z_0$  werden Nachfolger jedes Endzustands

Beispiel:



## Kapitel 3

# Reguläre Sprachen und endliche Automaten

### 3.1 Jede reguläre Sprache wird durch einen nichtdeterministischen endlichen Automaten erkannt

Ausgangspunkt (Definition regulär): Jede reguläre Sprache wird durch eine rechtslineare Grammatik beschrieben.

Schritt 1: Normalisierung der Grammatik:

Nur noch Regeln der Form  $h \rightarrow xh'$  und  $h \rightarrow \varepsilon$  ( $h, h' \in H, x \in X$ )

Konstruktion:

(a) Für jedes Paar von Regeln  $h \rightarrow wh'$   $h' \rightarrow h''$  Regel  $h \rightarrow wh''$  hinzufügen, solange, bis keine neuen Regeln hinzukommen.

(b) Regeln der Form  $h \rightarrow h'$  streichen

(c) Regeln  $h \rightarrow x_1 x_2 \dots x_n h'$  zu  $h \rightarrow x_1 h_1$   $h_1 \rightarrow x_2 h_2$  ...  $h_{n-1} \rightarrow x_n h'$

Regeln  $h \rightarrow x_1 x_2 \dots x_n$  zu  $h \rightarrow x_1 h_1$   $h_1 \rightarrow x_2 h_2$  ...  $h_n \rightarrow \varepsilon$

(mit „frischen“ Hilfssymbolen  $h_1, \dots$ )

Beobachtung: Beschriebene Sprache bleibt unverändert

# Jede reguläre Sprache wird durch einen nichtdeterministischen endlichen Automaten erkannt

## Schritt 2: Normalisierte Grammatik zu endlichem Automat

Konstruktion: Automat  $[H, X, d, s, F]$  mit

- $H$  ...Menge der Hilfssymbole der Grammatik als Zustände
- $X$  ... Alphabet der Grammatik als Alphabet
- $\delta(h, x) = \{h' \mid h \rightarrow xh' \in R\}$  als Überföhrungsfunktion
- $s$  ... Startsymbol der Grammatik als Anfangszustand
- $F = \{h \mid h \rightarrow_{\varepsilon} \in R\}$  ... Menge der Endzustände

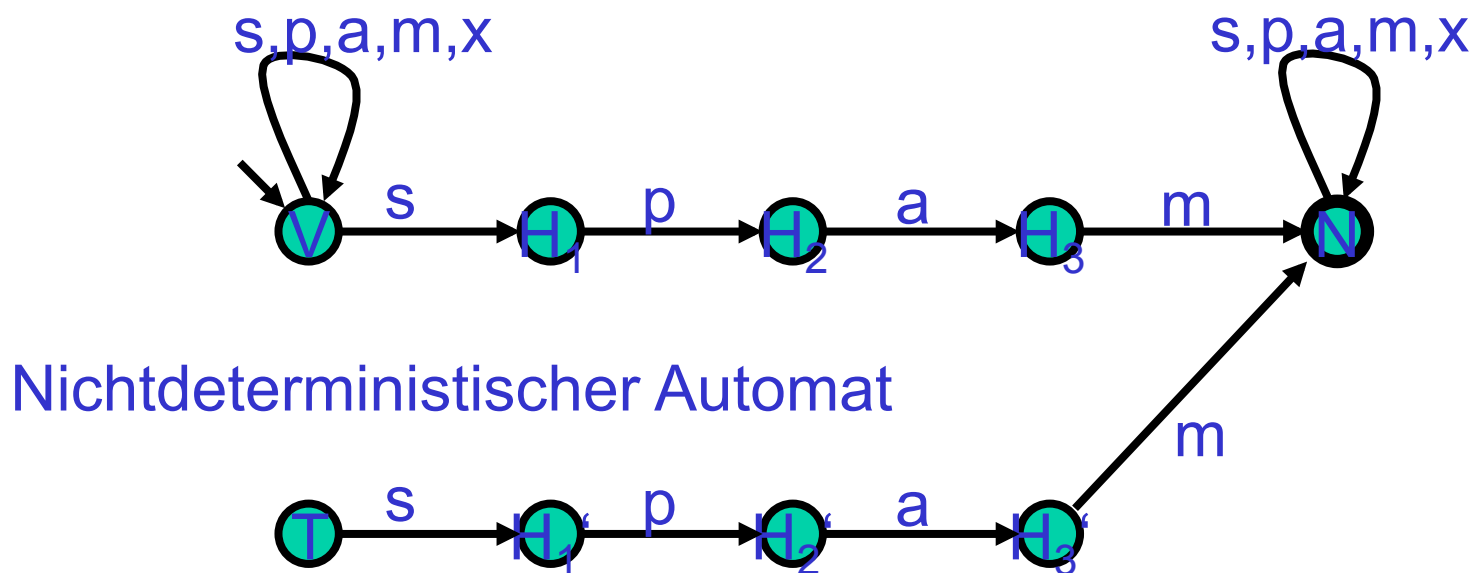


# Beispiel

$G = [ \{a,m,p,s,x\}, \{V,T,N\}, \{V \rightarrow aV, V \rightarrow mV, V \rightarrow pV, V \rightarrow sV, V \rightarrow xV, V \rightarrow T, T \rightarrow spamN, N \rightarrow \varepsilon, N \rightarrow aN, N \rightarrow mN, N \rightarrow pN, N \rightarrow sN, N \rightarrow xN\}, V ]$

Normalisierte Regelmenge:

$V \rightarrow aV, V \rightarrow mV, V \rightarrow pV, V \rightarrow sV, V \rightarrow xV, V \rightarrow sH_1, T \rightarrow sH_1',$   
 $H_1 \rightarrow pH_2, H_1' \rightarrow pH_2', H_2 \rightarrow aH_3, H_2' \rightarrow aH_3', H_3 \rightarrow mN, H_3' \rightarrow mN,$   
 $N \rightarrow \varepsilon, N \rightarrow aN, N \rightarrow mN, N \rightarrow pN, N \rightarrow sN, N \rightarrow xN$



# Jede von einem endlichen nichtdeterministischen Automat erkannte Sprache ist regulär

Konstruktion:  $[H, X, R, z_0]$  mit

$$z \rightarrow xz' \in R \text{ gdw. } z' \in \delta(z, x)$$
$$z \rightarrow \varepsilon \in R \text{ gdw. } z \in F$$

Lemma:  $wz$  genau dann ableitbar, wenn es einen Lauf mit  $w$  gibt, der zu  $z$  führt

Beweis (Induktion über  $|w|$ )

Anfang:  $|w|=0$ , also  $w=\varepsilon$ . Einziger Lauf endet bei  $z_0$ , einzige Ableitung:  $z_0 = \varepsilon z_0$

Schritt:

Vor:  $wz$  genau dann ableitbar, wenn es einen Lauf gibt, der zu  $z$  führt

Beh:  $wxz'$  genau dann ableitbar, wenn es einen Lauf gibt, der zu  $z'$  führt.

Bew: Lauf muss nach  $w$  in einem Zustand  $z^*$  sein mit  $z' \in \delta(z^*, x)$ .

Nach Vor.:  $wz^*$  ableitbar. Wegen  $z' \in \delta(z^*, x)$  ist  $z^* \rightarrow xz' \in R$ , also  $wxz'$  ableitbar.

Ist umgekehrt  $wxz'$  ableitbar, muss auch (einen Schritt früher) ein  $wz^*$  so ableitbar sein, dass eine Regel  $z^* \rightarrow xz'$  existiert. Nach Vor. gibt es einen Lauf mit  $w$  zu  $z^*$ , und, weil  $z^* \rightarrow xz' \in R$ , ist  $z' \in \delta(z^*, x)$ . Also gibt es einen Lauf mit  $wx$  zu  $z'$ . q.e.d.

**Jede von einem endlichen nichtdeterministischen Automaten erkannte Sprache kann auch von einem deterministischen endlichen Automat erkannt werden**

Konstruktion: Zustand des deterministischen Automaten  $D =$   
*Menge von Zuständen* des nichtdeterministischen Automaten  $N$

Ziel: Wenn  $D$  mit  $w$  in  $z$  gelangt, ist  $z$  genau die Menge derjenigen Zustände, in die  $N$  mit  $w$  gelangen *kann*.

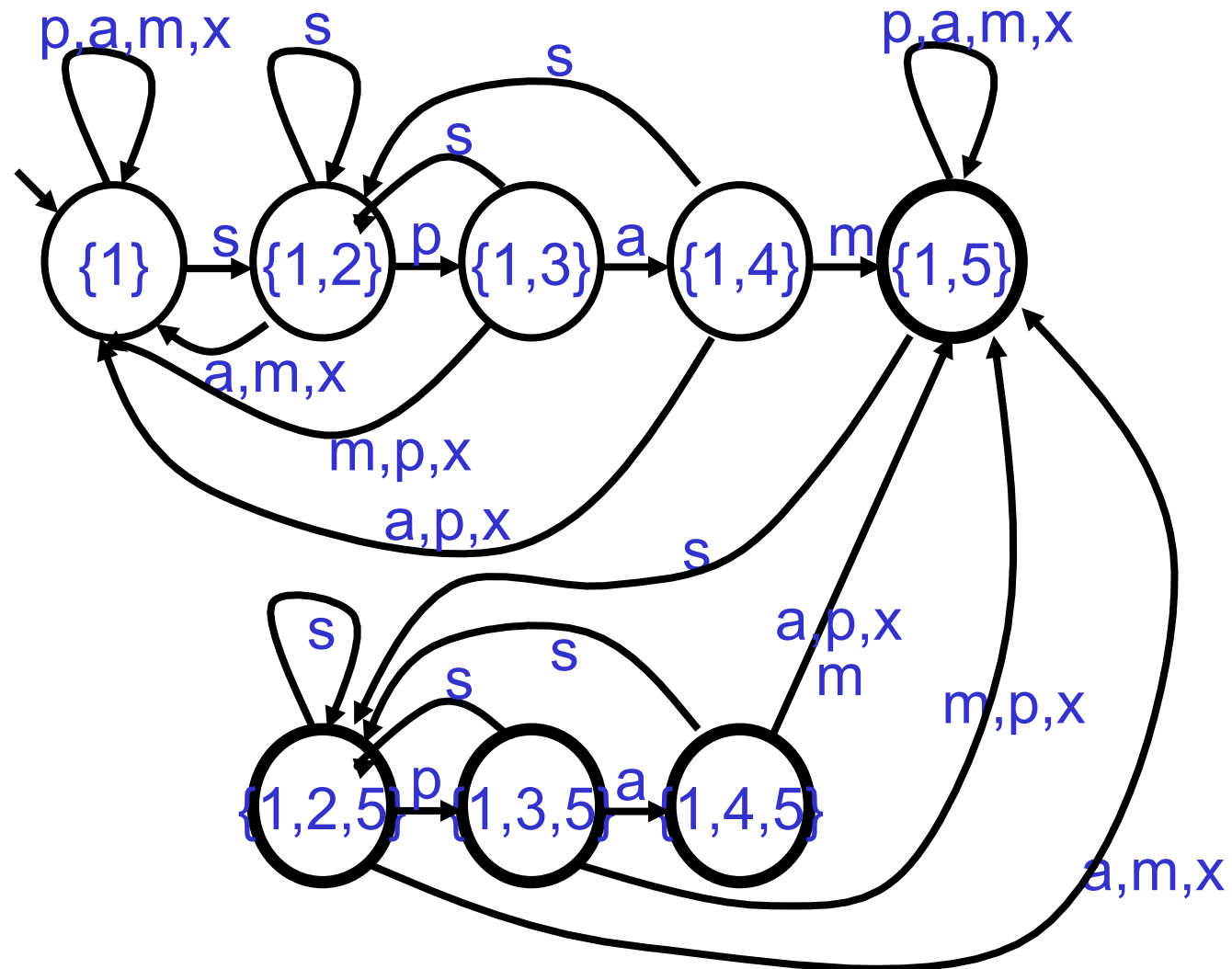
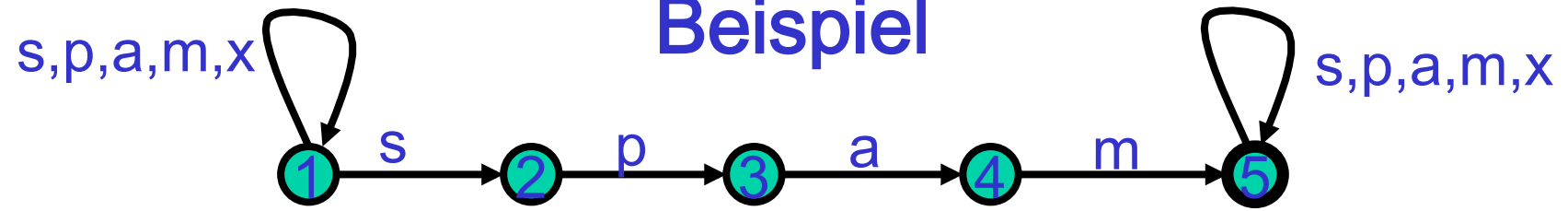
→ Aus  $N = [Z, X, \delta, z_0, F]$  wird

$D = [\wp(Z), X, \delta', \{z_0\}, F']$  mit

$\delta'(M, x) = \bigcup_{z \in M} \delta(z, x)$  und

$F' = \{M \in \wp(Z) \mid M \cap F \neq \emptyset\}.$

# Beispiel



# Jede von einem endlichen nichtdeterministischen Automaten erkannte Sprache kann auch von einem deterministischen endlichen Automat erkannt werden

Satz: Wenn in D Lauf mit  $w$  zu  $M$  führt, ist  $M$  genau die Menge derjenigen Zustände, die durch einen Lauf mit  $w$  in  $N$  erreicht werden können.

Beweis (Induktion über  $|w|$ ).

Anfang ( $w = \varepsilon$ ) klar.

Möge Lauf in D mit  $w$  zu  $M_w$  führen.

Vor:  $M_w$  ist Menge der Zustände, zu denen Lauf mit  $w$  in  $N$  führt

Beh:  $M_{wx}$  ist Menge der Zustände, zu denen Lauf mit  $wx$  in  $N$  führt.

Bew.

- (1) Sei  $z_1 \dots z_n z_{n+1}$  Lauf mit  $wx$  in  $N$ .  $\rightarrow z_1 \dots z_n$  ist Lauf mit  $w$  in  $N \rightarrow z_{n+1} \in \delta(z_n, x)$   
und (nach Vor)  $z_n \in M_w \rightarrow$  (nach Konstruktion von D)  $z_{n+1} \in M_{wx}$
- (2) Sei  $z \in M_{wx} \rightarrow$  (nach Konstruktion von  $M_{wx}$ ) es gibt ein  $z' \in M_w$  mit  $z \in \delta(z', x)$   
 $\rightarrow$  (nach Vor) Es gibt einen Lauf in  $N$  mit  $w$  zu  $z' \rightarrow$  es gibt einen Lauf (über  $z'$ ) mit  $wx$  zu  $z$  in  $N$  q.e.d.

## L ist regulär gdw. L durch linksrekursive Grammatik beschreibbar

Def: Kontextfreie Grammatik  $G$  ist linkslinear, wenn auf der rechten Seite höchstens das erste Zeichen ein Hilfssymbol ist.

Beweis: Wenn  $L$  regulär, so wird  $L$  durch einen (z.B. nichtdet.) Automat  $A = [Z, X, \delta, z_0, F]$  erkannt.

Definieren linkslineare Grammatik  $G = [X, Z \cup \{s\}, R, s]$  mit Regeln

-  $s \rightarrow z$  für alle  $z \in F$

-  $z' \rightarrow zx$  für alle  $x, z, z'$  mit  $z' \in \delta(z, x)$

-  $z_0 \rightarrow \varepsilon$

Behauptung:  $zw$  ableitbar gdw. ein bei  $z$  beginnender Lauf mit  $w$  zu einem Endzustand führt.

Beweis: Analog gezeigter Argumentationen

## Zwischenfazit 3.1

REG ist die Menge der Sprachen

- die durch rechtslineare Grammatik beschreibbar sind
- die durch linkslineare Grammatik beschreibbar sind
- die durch deterministische endliche Automaten erkannt werden
- die durch nichtdeterministische endliche Automaten erkannt werden
- für die nur endlich viele Nerode-Äquivalenzklassen existieren

## 3.2 Abschlusseigenschaften

REG ist abgeschlossen bzgl. der Operationen

- $\cap$  (Produktautomat [Folie 56] endlicher Automaten ist endlich)
  - Komplement (siehe Folie 56)
  - Verkettung
  - Vereinigung
  - Iteration
- } (siehe Folie 34)

Außerdem: Jede endliche Sprache ist regulär.



# Synthese regulärer Sprachen

Satz: Die Menge der regulären Sprachen ist genau die Menge der Sprachen, die sich mittels Vereinigung, Verkettung und Iteration aus max. einelementigen, max. einbuchstabigen Sprachen konstruieren lassen.

Beweis Teil 1:

Jede aus einelementigen einbuchstabigen Sprachen konstruierbare Sprache ist regulär

folgt aus voriger Folie

# Beweis: Jede reguläre Sprache kann durch Vereinigung+Verkettung+Iteration erzeugt werden

$L$  regulär  $\rightarrow$  ex. deterministischer erkennender Automat.

Sei  $Z = \{z_0, z_1, \dots, z_n\}$ .

Definieren Sprachen  $L_{ij}^k$  : Menge der Wörter, für die Lauf, beginnend in  $z_i$ , in  $z_j$  endet und dazwischen nur Zustände aus  $\{z_0, \dots, z_{k-1}\}$  betreten werden.

$L_{ij}^0 = \{x\}$ , falls  $\delta(z_i, x) = z_j$ , sonst  $L_{ij}^0 = \emptyset$ .

$L_{ii}^0 = \{\varepsilon, x\}$ , falls  $\delta(z_i, x) = z_i$ , sonst  $L_{ii}^0 = \{\varepsilon\}$ .

$L_{ij}^{k+1} = L_{ij}^k \cup L_{ik}^k L_{kk}^{k*} L_{kj}^k$

$L = \bigcup_{z_i \in F} L_{0i}^{n+1}$

# Reguläre Ausdrücke

Weitere Beschreibungsform regulärer Sprachen:

Sei  $X$  Alphabet.

- $\varepsilon$  und jedes  $x \in X$  sind reguläre Ausdrücke
- Sind  $P$  und  $Q$  reguläre Ausdrücke, so auch
  - $(P|Q)$
  - $PQ$  und
  - $P^*$

Regulärer Ausdruck beschreibt Sprache (Symbole stehen für die Operationen Vereinigung, Verkettung, Iteration)

# Beispiel

Die bekannte Spam-Sprache kann durch regulären Ausdruck

$(s|p|a|m|x)^*spam(s|p|a|m|x)^*$  beschrieben werden

# Anwendungen

- Reguläre Ausdrücke (und Abwandlungen davon) dienen z.B. zur Formulierung von Suchmustern in Texteditoren
- Reguläre Ausdrücke dienen der Beschreibung der Lexik von Programmiersprachen, z.B.
  - Identifier =  $(A|...|Z|a|...|z)(A|...|z|_|0|...|9)^*$
  - FP-Zahl =  $(\epsilon|-)(\epsilon|0|(1|...|9)(0|...|9)^*).(0|...|9)^*(\epsilon|E(\epsilon|-)(0|...|9)(0|...|9))$
  - ...

# Überführung regulärer Ausdrücke in endliche Automaten

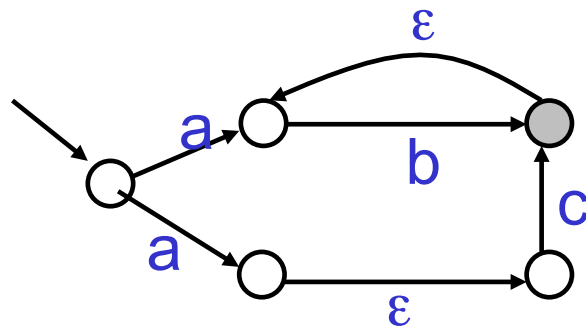
- Sinn: Übersetzung menschenverständlicher regulärer Ausdrücke in maschinenverständliche Automaten
- erster Schritt: Überführung in Automaten mit spontanen Übergängen:

# Automat mit spontanen Übergängen

= Automat mit Übergangsrelation  $d: Z \times (X \cup \{\varepsilon\}) \rightarrow \wp(Z)$

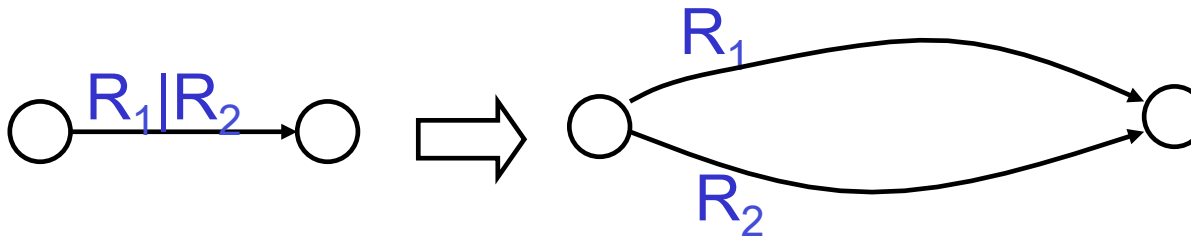
Akzeptierender Lauf: Zustandsfolge vom Initialzustand zu einem Endzustand so, dass Folge der Zeichen an den Übergängen Eingabe ergibt

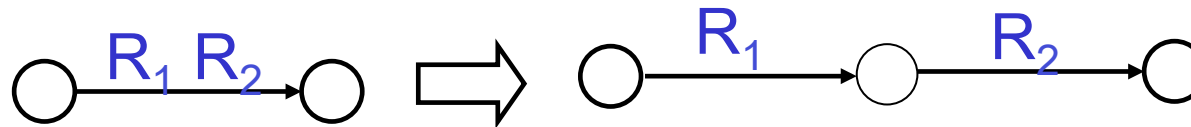
( $\rightarrow$  bei  $\varepsilon$ -Übergang kein Weiterlesen der Eingabe)

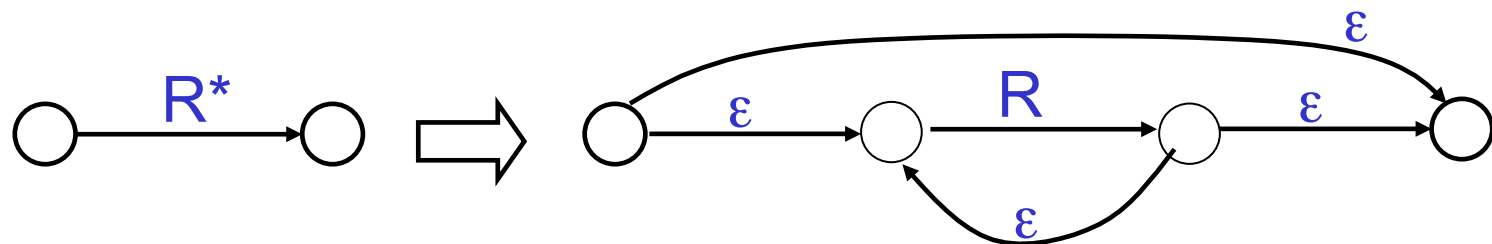


# Umwandlung Regulärer Ausdrücke in NDA

Start:  (R ist gegebener reg. Ausdruck)

Do: 

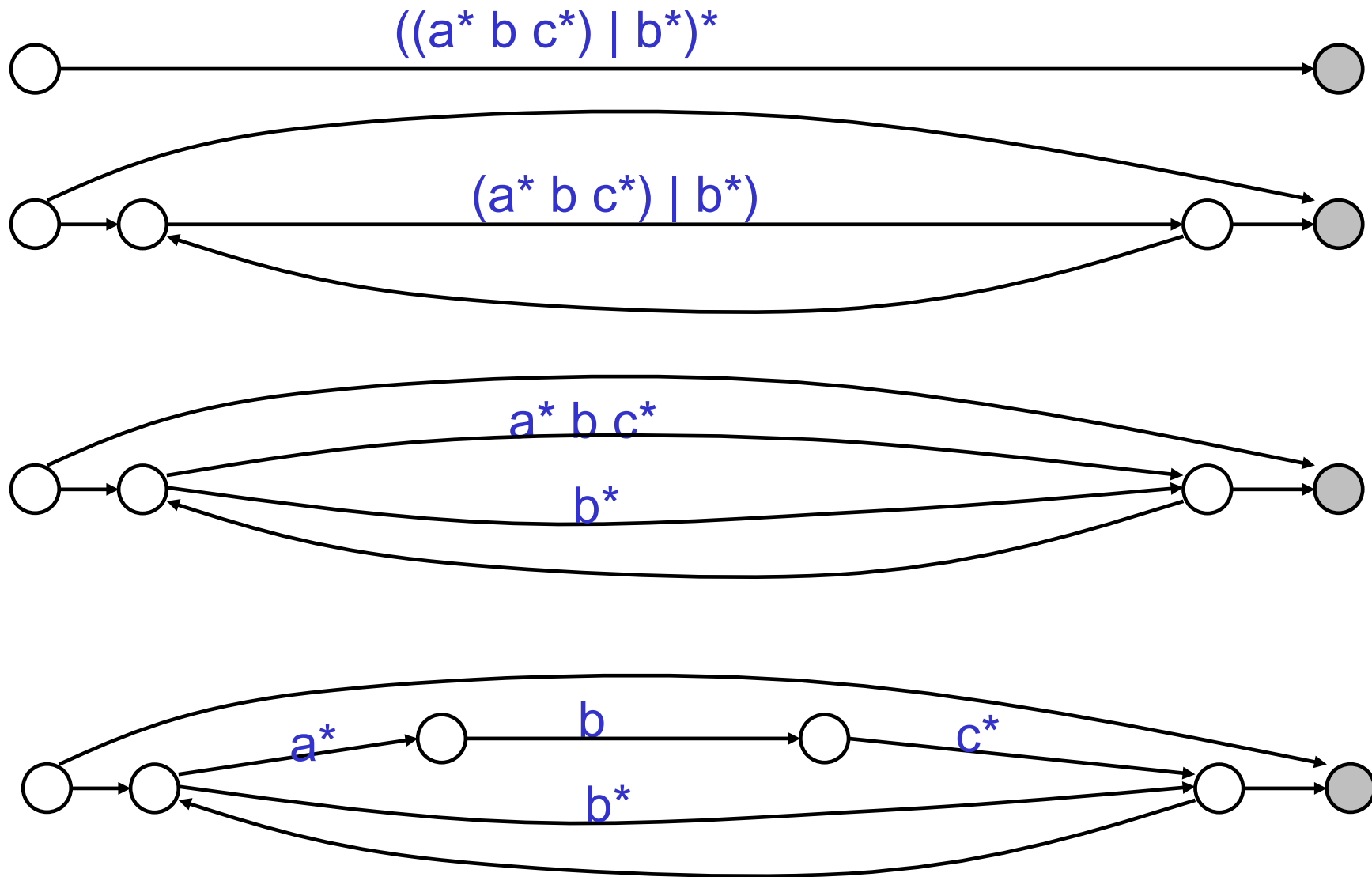




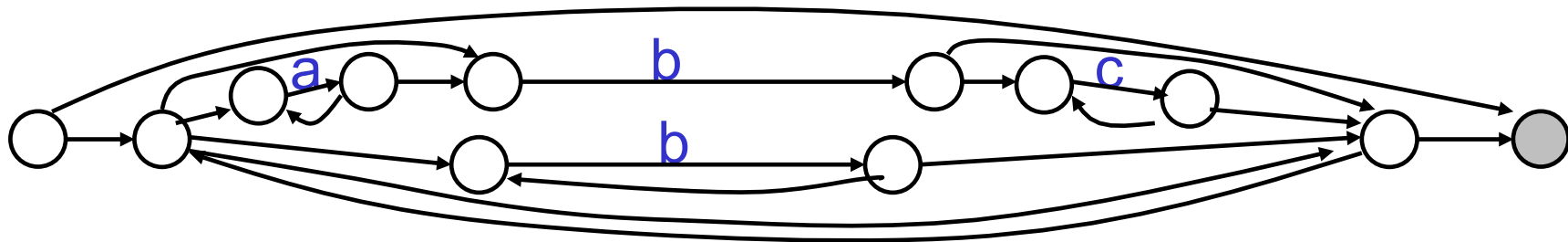
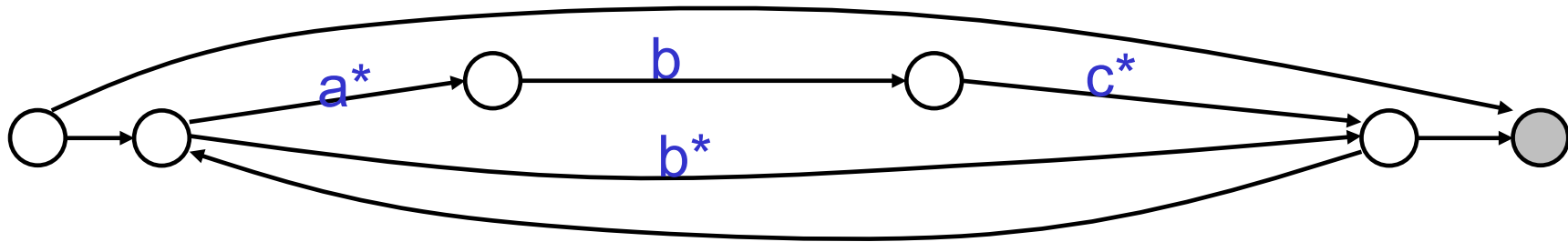
Until: Alle Übergänge einzelne Zeichen oder  $\epsilon$ !  
→ Anzahl der Schritte  $\in O(\text{Länge des Ausdrucks})$



# Beispiel



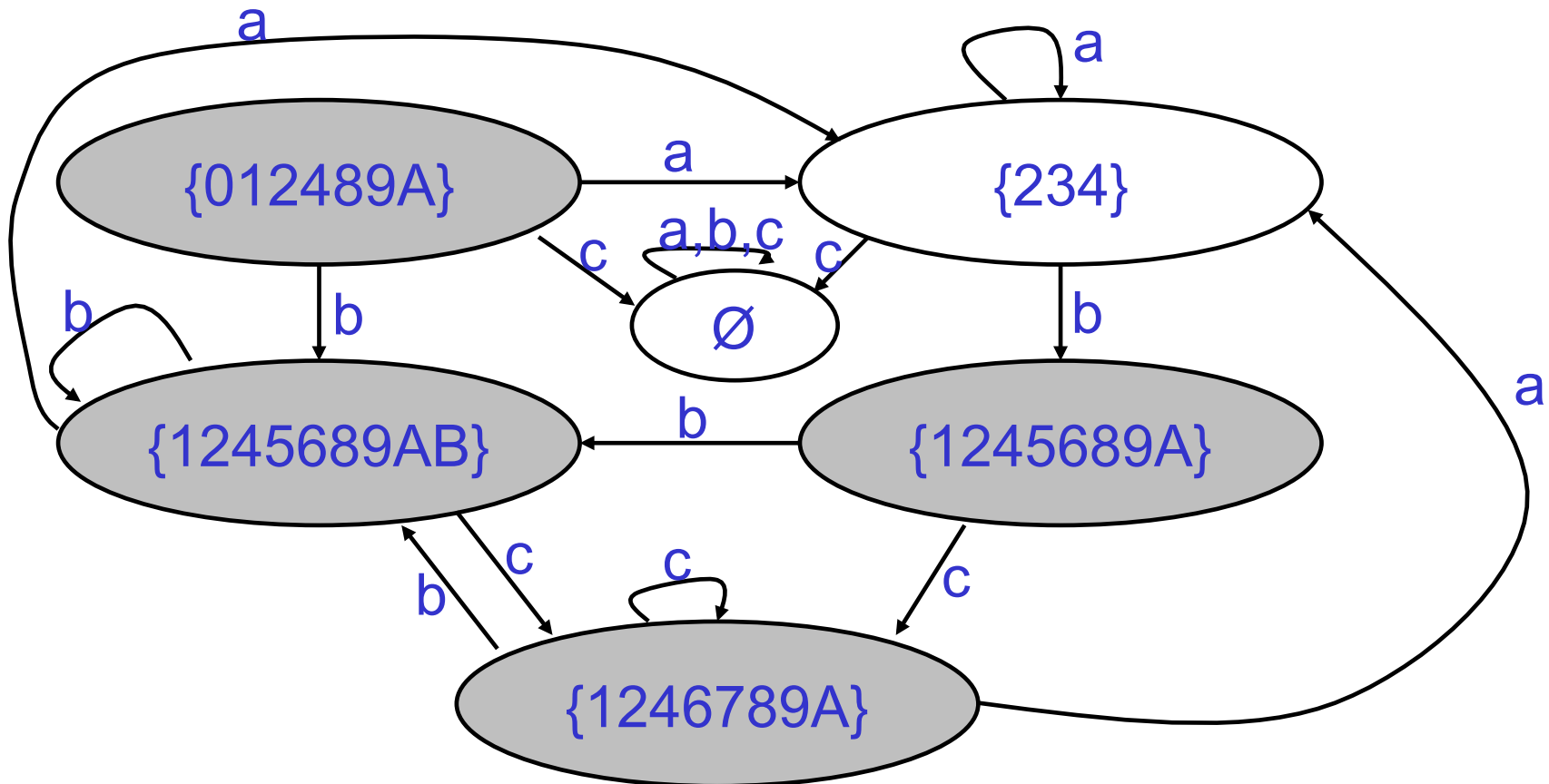
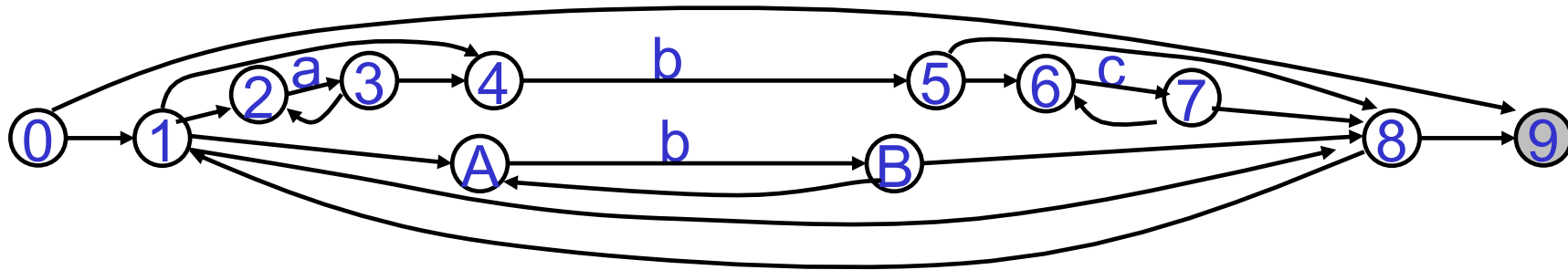
# Beispiel



# Überführung in DEA

- Wie für bisherige nichtdeterministische Automaten:
- Zustände des DEA sind Mengen von Zst. des NDA
- Anfangszustand: Menge der Zustände, die vom Anfangszst. des NDA ohne Eingabe erreichbar
- Folgezustand bei Eingabe  $x$ : Jeder Zustand, der mit  $x$  und beliebig vielen spontanen Übergängen erreichbar ist
- Endzustand: Falls ein Endzustand in Menge enthalten

# Beispiel



## 3.3 Minimierung endlicher Automaten

Geg.: Endlicher deterministischer Automat  $A$ , der Sprache  $L_A$  akzeptiert, oBdA: alle Zustände erreichbar (sonst unerreichbare Zustände streichen)

Gesucht: Endlicher Automat  $A'$  minimaler Größe mit  $L_{A'} = L_A$

Lösungsansatz: Nerode-Relation aus Kapitel 2:

Erinnerung:

$w_1 \sim w_2$  gdw. für alle  $w^*$ :  $w_1 w^* \in L$  gdw.  $w_2 w^* \in L$

Zustände = Äquivalenzklassen bzgl.  $\sim$

Hatten auch bereits überlegt: Wenn  $w_1$  und  $w_2$  in  $A$  zum gleichen Zustand führen, ist  $w_1 \sim w_2$

# Minimierung endlicher Automaten

Bleibt also: Äquivalenz zwischen Zuständen in A

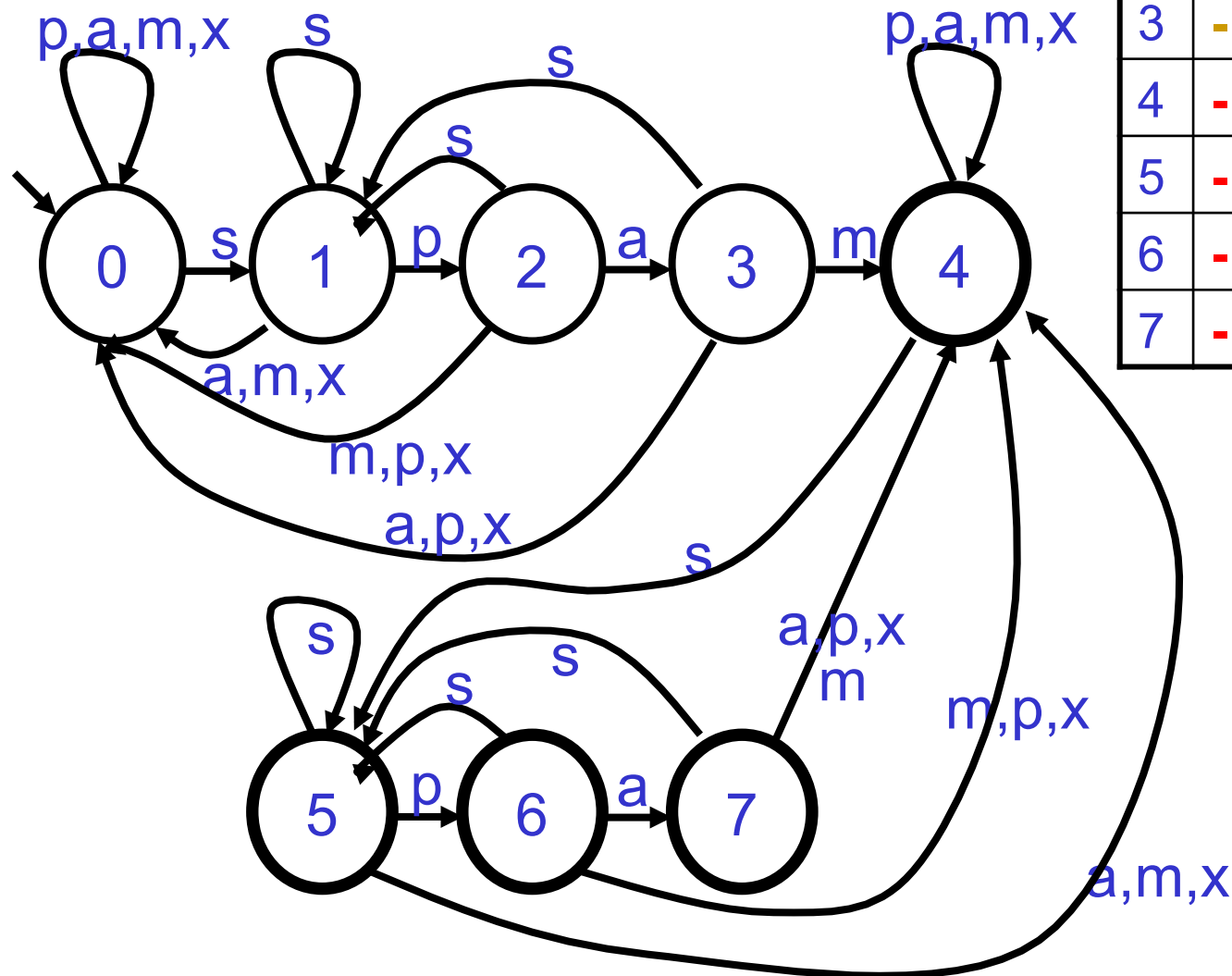
$z_1 \sim z_2$  gdw. die Wörter, die zu  $z_1$  führen, Nerode-äquivalent zu den Wörtern sind, die zu  $z_2$  führen.

Idee: Tabelle  $Z \times Z$ , markieren nicht äquivalente Zustandspaare.

$z_1, z_2$  nichtäquivalent:

1.  $z_1 \in F, z_2 \notin F$  ... Fortsetzung mit leerem Wort
2.  $\delta(z_1, x)$  nicht äquivalent zu  $\delta(z_2, x)$   
... Fortsetzung mit  $x$ ...

# Beispiel



	0	1	2	3	4	5	6	7
0	x	-	-	-	-	-	-	-
1	-	x	-	-	-	-	-	-
2	-	-	x	-	-	-	-	-
3	-	-	-	x	-	-	-	-
4	-	-	-	-	x	-	-	-
5	-	-	-	-	x	x	-	-
6	-	-	-	-	x	x	x	-
7	-	-	-	-	x	x	x	x

1.  $z_1 \in F, z_2 \notin F$
2.  $\delta(\dots)$
3.  $\delta(\dots)$
4.  $\delta(\dots)$
5. Fertig

# Beweis der Korrektheit

Zustandspaare mit „-“ in Tabelle sind nicht äquivalent: siehe Überlegung auf Folie 76

Bleibt zu zeigen: übrige Paare sind äquivalent.

Zu jedem Paar  $z_1, z_2$  nicht äquivalenter Zustände gibt es ein Wort  $x_1 x_2 \dots x_n$ , das von  $z_1$  aus zu  $F$  und von  $z_2$  nicht zu  $F$  führt (oder umgekehrt). Wenn es unter den verbliebenen Paaren nichtäquivalente gibt, wählen wir eines, wo  $n$  minimal ist.

1. Fall  $n = 0 \rightarrow z_1 \in F, z_2 \notin F$  (oder umgekehrt) – wäre im 1. Schritt markiert worden
2. Fall  $n > 0 \rightarrow \delta(z_1, x_1)$  nicht äquivalent zu  $\delta(z_2, x_1)$  (wegen Wort  $x_2 \dots x_n$ ). Wegen Minimalität von  $n$  ist Paar  $\delta(z_1, x_1)$  und  $\delta(z_2, x_1)$  markiert  $\rightarrow z_1, z_2$  ebenfalls markiert.



## 3.4 Wann ist eine Sprache nicht regulär?

Pumping-Lemma für reguläre Sprachen:

Wenn  $L$  regulär ist, so gibt es eine Zahl  $n$  so, dass jedes Wort  $w$  mit  $|w| > n$  so in  $u_1vu_2$  zerlegt werden kann, dass  $|v| > 0$ ,  $|u_1v| \leq n$  und für jedes  $i \in \mathbb{N}$  auch  $u_1v^iu_2 \in L$  ist.

Beweis. Sei  $A$  irgendein deterministischer endlicher Automat, der  $L$  erkennt. Setze  $n = \text{Anzahl der Zustände von } A$ .

Sei  $|w| > n$ . Dann muss ein Lauf von  $w$  wenigstens einen Zst.  $z$  wiederholen. Setze  $u_1 = \text{Teilwort, das von } z_0 \text{ zum ersten } z \text{ führt}$ ,  $u_2 = \text{Teilwort, das vom letzten } z \text{ zu einem Endzustand führt}$ ,  $v$  als alles dazwischen.  $\rightarrow$  Jede  $i$ -fache Wiederholung von  $v$  bei  $z$  führt wieder zu  $z \rightarrow u_1v^iu_2$  wird für beliebige  $i$  akzeptiert.

# Anwendung

Zeigen, dass  $L = \{a^i b^i \mid i \in \mathbb{N}\}$  nicht regulär ist.

Annahme,  $L$  regulär. Betrachten  $a^n b^n$  mit  $n$  aus Pumping-Lemma.  
Weil  $|u_1 v| \leq n$  und  $|v| > 0$ , ist  $v = a^j$  für irgendein  $j > 0$ .

→ (nach Pumping-Lemma):  $a^{i+j} b^i \in L$  im Widerspruch zur Wahl von  $L$ .

Bem.: Es gibt nichtreguläre Sprachen, die nicht mittels Pumping-Lemma „überführt“ werden können.

Beispiel:  $\{a^n b^{m^2} \mid m, n \geq 1\}$ : Jedes Wort zerlegbar:  $u = e \ v = a \ w = \text{Rest}$   
Sprache nicht regulär, siehe ein paar Folien weiter

# Nichtregularität mittels Nerode-Relation

- wissen: Sprache regulär  
gdw durch endlichen deterministischen Automaten beschreibbar  
gdw minimaler Automat hat endlich viele Zustände  
gdw Nerode-Relation hat endlich viele Klassen  
(„endlichen Index“)
- nichtregulär gdw. Nerode-Relation hat unendlich viele Klassen

# Anwendung

$L = \{a^i b^i \mid i \geq 0\}$ :

für  $i \neq j$  sind  $a^i$  und  $a^j$  nicht äquivalent:  $a^i b^i \in L$     $a^j b^i \notin L$

also: unendlich viele Äquivalenzklassen;  $L$  nicht regulär

# Anwendung

- $L = \{a^n b^{m^2} \mid m, n \geq 1\}$ :
- für kein  $m$  sind  $ab^{m^2}, \dots, ab^{(m+1)^2-1}$  äquivalent
- also: unendlich viele Äquivalenzklassen, also  $L$  nicht regulär

## Kapitel 4

# Kontextfreie Sprachen und Kellerautomaten

## 4.1 Bedeutung

- Formelsprachen, z.B.
  - Arithmetik:  $\text{Term} \rightarrow \text{Term} + \text{Term}$   
 $\text{Term} \rightarrow \text{Term} - \text{Term}$   
 $\text{Term} \rightarrow \text{Term} * \text{Term}$   
 $\text{Term} \rightarrow \text{Term} / \text{Term}$   
 $\text{Term} \rightarrow ( \text{Term} )$   
 $\text{Term} \rightarrow \text{Variable}$
  - Aussagenlogik:  $\text{Formel} \rightarrow \neg \text{Formel}$   
 $\text{Formel} \rightarrow \text{Formel} \wedge \text{Formel}$   
 $\text{Formel} \rightarrow \text{Formel} \vee \text{Formel}$   
 $\text{Formel} \rightarrow ( \text{Formel} )$   
 $\text{Formel} \rightarrow \text{Variable}$

# Bedeutung

- Programmiersprachen

Statement → Location = Expression

Statement → if Expression then Statement else Statement end

Statement → while Expression do Statement end

Statement → Statement ; Statement

Statement → Procedurecall

Procedure → identifier ( Parameterlist ) : Type

Localvariables begin Statement end



# Bedeutung

- Markup-Sprachen, z.B. HTML, XML, ...

`<h1> Text <b> Bold-Text </b> Text </h1> ...`

# Bedeutung

Zentrale Fähigkeit kontextfreier Sprachen:  
Klammerung, Neststrukturen

$\{a^i b^i \mid i \in \mathbb{N}\}$  ist kontextfrei:

$S \rightarrow \varepsilon, \quad S \rightarrow a S b$

# Backus-Naur-Form (BNF)

Notation für kontextfreie Grammatiken:

- pro linker Seite eine mit | getrennte Liste aller rechten Seiten
- Hilfssymbole in <...> eingeschlossen

$$\begin{aligned} \langle \text{Term} \rangle ::= & \langle \text{Term} \rangle + \langle \text{Term} \rangle \mid \langle \text{Term} \rangle - \langle \text{Term} \rangle \mid \\ & \langle \text{Term} \rangle * \langle \text{Term} \rangle \mid \langle \text{Term} \rangle / \langle \text{Term} \rangle \mid \\ & ( \langle \text{Term} \rangle ) \mid \langle \text{Variable} \rangle \end{aligned}$$
$$\langle \text{Variable} \rangle ::= \text{identifizier}$$

# Erweiterte Backus-Naur-Form (EBNF)

Expression = Term { („+“ | „-“ ) Term } ;

Zahl = [„-“] Ziffer { Ziffer } ;

{ ... }    Wiederholung    0 ... beliebig oft

[ ... ]    Optional            0 ... 1 mal

( ... )    Klammerung

„ ... “    Terminalzeichenfolgen

Blabla    Nichtterminal

# Erweiterte Backus-Naur-Form (EBNF)

Eine Grammatik für EBNF in BNF:

```
<ebnf> ::= <rule> | <ebnf> <rule>
<rule> ::= <nonterminal> = <expression> ;
<expression> ::= <exp> | <expression> „|“ <exp>
<exp> ::= <term> | <exp> <term>
<term> ::= { <expression> } | [ <expression> ] |
           ( <expression> ) | <nonterminal> |
           <terminal>
<nonterminal> ::= identifier
<terminal> ::= “ ascii-sequence “
```

# EBNF $\rightarrow$ BNF

Idee: Führen zu jedem Teilterm der EBNF ein eigenes Hilfsymbol ein, aus dem wir die diesem Teilterm entsprechenden Zeichenreihen ableiten.

Beispiel: Expression = Term { („+“ | „-“) Term } ;

$\rightarrow$   $\langle \text{Expression} \rangle ::= \langle H_{\text{Term} \{ („+“ | „-“) \text{Term} \}} \rangle$

$\langle H_{\text{Term} \{ („+“ | „-“) \text{Term} \}} \rangle ::= \langle H_{\text{Term}} \rangle \langle H_{\{ („+“ | „-“) \text{Term} \}} \rangle$

$\langle H_{\text{Term}} \rangle ::= \langle \text{Term} \rangle$

$\langle H_{\{ („+“ | „-“) \text{Term} \}} \rangle ::= \epsilon \mid \langle H_{\{ („+“ | „-“) \text{Term} \}} \rangle \langle H_{(„+“ | „-“) \text{Term}} \rangle$

$\langle H_{(„+“ | „-“) \text{Term}} \rangle ::= \langle H_{(„+“ | „-“)} \rangle \langle H_{\text{Term}} \rangle$

$\langle H_{(„+“ | „-“)} \rangle ::= \langle H_{„+“} \rangle \mid \langle H_{„-“} \rangle$

$\langle H_{„+“} \rangle ::= +$

$\langle H_{„-“} \rangle ::= -$

# EBNF $\rightarrow$ BNF systematisch

$nt = E ; \quad \rightarrow \quad \langle nt \rangle ::= H_E$  - Regeln für  $H_E$  wie folgt:

$H_{E_1 \mid E_2} : \quad \langle H_{E_1 \mid E_2} \rangle ::= \langle H_{E_1} \rangle \mid \langle H_{E_2} \rangle$

$H_{E_1 E_2} : \quad \langle H_{E_1 E_2} \rangle ::= \langle H_{E_1} \rangle \langle H_{E_2} \rangle$

$H_{\{E\}} : \quad \langle H_{\{E\}} \rangle ::= \varepsilon \mid \langle H_{\{E\}} \rangle \langle H_E \rangle$

$H_{[E]} : \quad \langle H_{[E]} \rangle ::= \varepsilon \mid \langle H_E \rangle$

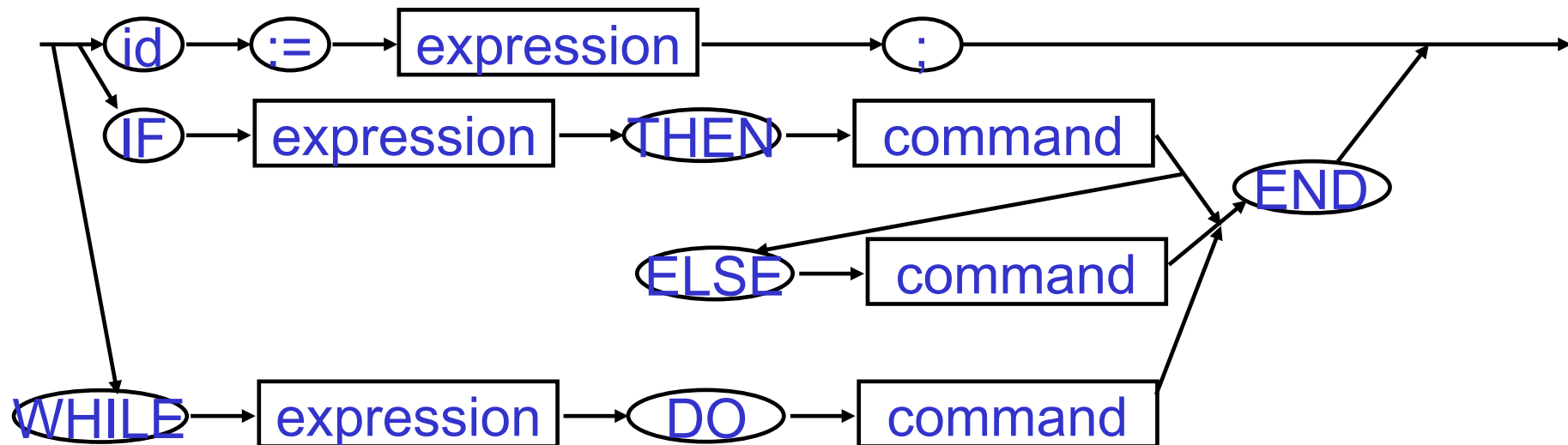
$H_{(E)} : \quad \langle H_{(E)} \rangle ::= \langle H_E \rangle$

$H_{\text{ident}} : \quad \langle H_{\text{ident}} \rangle ::= \langle \text{ident} \rangle$

$H_{\text{"ascii-seq"}} : \quad \langle H_{\text{"ascii-seq"}} \rangle ::= \text{ascii-seq}$

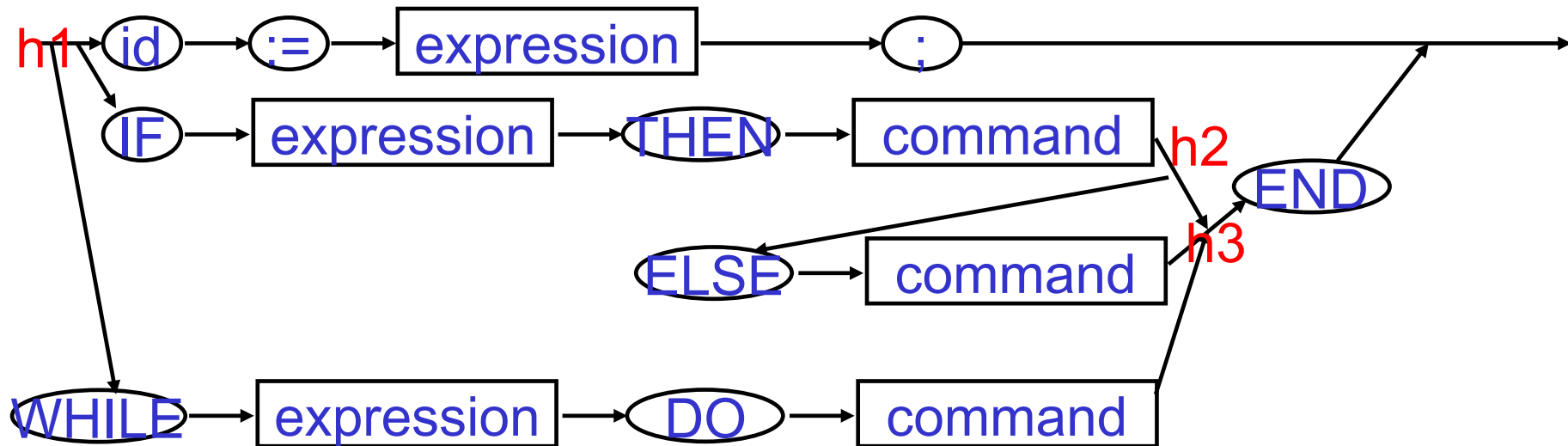
# Syntaxdiagramme

command:





# Umwandlung von Syntaxdiagrammen in kontextfreie Grammatiken



```
<h1> ::= id := <expression> ;  
      | IF <expression> THEN <command> <h2>  
      | WHILE <expression> DO <command> <h3>  
<h2> ::= <h3>  
      | ELSE <command> <h3>  
<h3> ::= END
```

## Bereits erwähnt:

Ableitungsprozess einer kontextfreien Grammatik liefert einen Ableitungsbaum:

$\langle \text{Term} \rangle ::= \langle \text{Term} \rangle + \langle \text{Term} \rangle \mid \langle \text{Term} \rangle * \langle \text{Term} \rangle \mid ( \langle \text{Term} \rangle ) \mid x$

Term

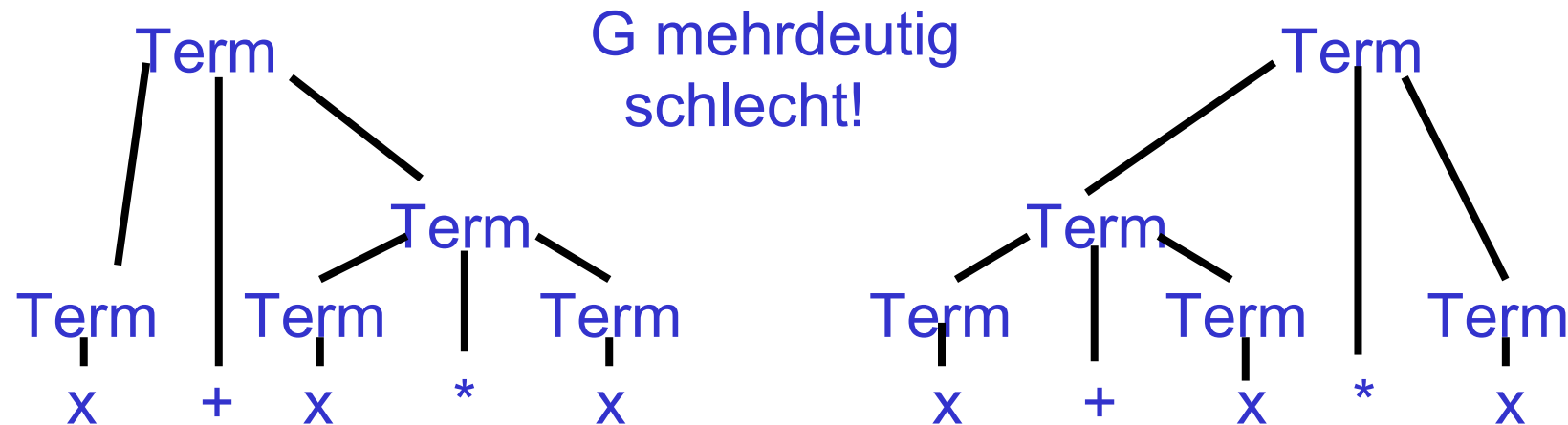
Term + Term

Term + Term \* Term

Term

Term \* Term

Term + Term \* Term



# Eindeutige Grammatiken

Kontextfreie Grammatik heißt eindeutig, falls es zu jedem Wort genau einen Ableitungsbaum gibt

Kann immer noch mehrere Ableitungen geben!

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow x$

$F \rightarrow ( E )$

$E$

$E + T$

$T + T$

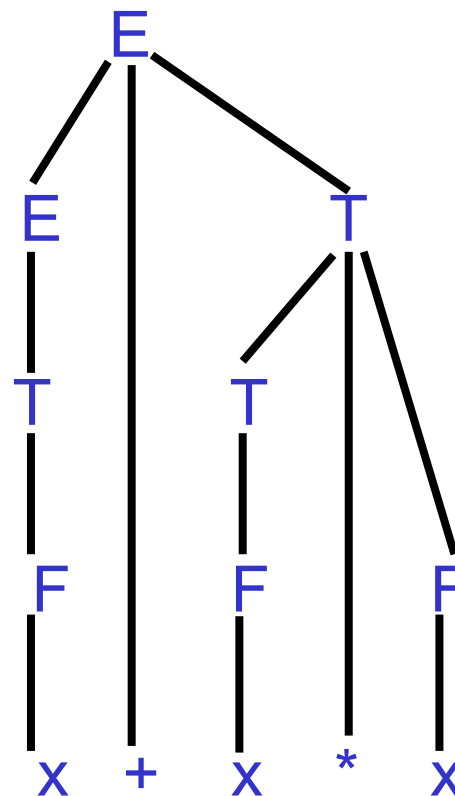
$F + T$

$x + T$

$x + T * F$

$x + F * F$

$x + x * F$



*Linksableitung*

$E$

$E + T$

$E + T * F$

$E + T * x$

$E + F * x$

$E + x * x$

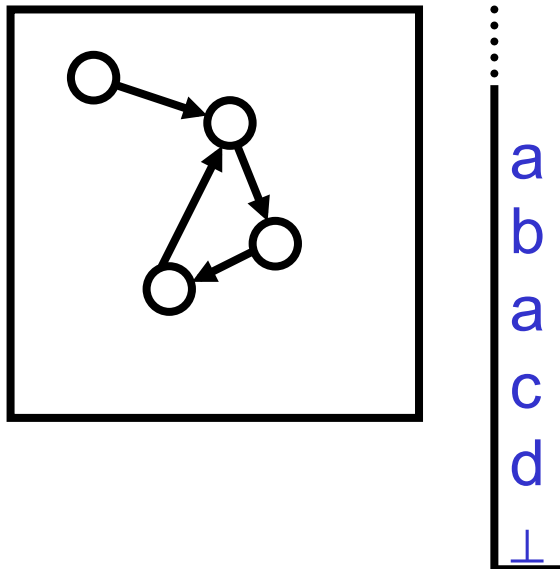
$T + x * x$

$F + x * x$

*Rechtsableitung*

## 4.2 Kellerautomaten

= endlicher Automat + zusätzlicher unbeschränkter Speicher, wo Daten „oben“ eingelagert und „oben“ wieder entfernt werden können



# Kellerautomaten

Formal:

$Z$ : endliche Menge von Zuständen

$X$ : Eingabealphabet

$\Gamma$ : Kellularphabet mit speziellem Startsymbol  $\perp$

$z_0$ : Anfangszustand

$\delta$ : Überföhrungsfunktion  $\delta: Z \times \{X \cup \varepsilon\} \times \Gamma \rightarrow \wp(Z \times \Gamma^*)$

( $Z_f$ : Endzustandsmenge)

# Verhalten eines Kellerautomaten

Beschrieben als (nicht endlicher!) Automat mit  $\varepsilon$ -Übergängen

- Zustände (Konfigurationen):  $Z \times \Gamma^*$
- Alphabet  $X$  (das Eingabealphabet)
- Anfangszustand  $[z_0, \perp]$
- Übergangsfunktion  $d$ :  $[z', wv] \in d([z, wy], x)$  gdw.  $[z', v] \in \delta(z, x, y)$
- Endzustände:  $Z \times \{\varepsilon\}$      *Akzeptanz durch leeren Keller*  
*oder*
- Endzustände:  $Z_f \times \Gamma^*$      *Akzeptanz durch Endzustände*

## Beispiel: leerer Keller

	$z_0 B$	$z_0 \perp$	$z_1 B$	$z_1 \perp$
a	$z_0 BB$	$z_0 \perp B$	--	--
b	--	--	$z_1$	--
$\varepsilon$	$z_1 B$	$z_1 \perp$	--	$z_1$

Lauf mit aaabbb:

$z_0 \perp$   
 a  $z_0 \perp B$   
 a  $z_0 \perp BB$   
 a  $z_0 \perp BBB$   
 $\varepsilon$   $z_1 \perp BBB$   
 b  $z_1 \perp BB$   
 b  $z_1 \perp B$   
 b  $z_1 \perp$   
 $z_1$

## Beispiel: Endzustände

	$z_0 B$	$z_0 \perp$	$z_1 B$	$z_1 \perp$	
a	$z_0 BB$	$z_0 \perp B$	--	--	$Z_f = \{z_2\}$
b	--	--	$z_1$	--	
$\varepsilon$	$z_1 B$	$z_1 \perp$	--	$z_2$	

Lauf mit aaabbb:

$z_0 \perp$   
 a  $z_0 \perp B$   
 a  $z_0 \perp BB$   
 a  $z_0 \perp BBB$   
 $\varepsilon$   $z_1 \perp BBB$   
 b  $z_1 \perp BB$   
 b  $z_1 \perp B$   
 b  $z_1 \perp$   
 b  $z_2$



# Beide Endkriterien sind äquivalent

- Idee 1 (Endzustand  $\rightarrow$  leerer Keller):  
Im Endzustand werden weitere Übergänge eingebaut, die den Keller leeren
- Idee 2 (leerer Keller  $\rightarrow$  Endzustand):  
Wie im Beispiel eben: bei (ggf. neu einzuführendem) Kellern Anfangssymbol in Endzustand übergehen

# Für deterministische Kellerautomaten sind die Endkriterien nicht äquivalent!

mit leerem Keller kann ein deterministischer Kellerautomat nur solche Sprachen akzeptieren, bei denen kein echtes Anfangsstück eines Wortes der Sprache ebenfalls Element der Sprache ist:

$$w \in L \text{ und } w = w_1 w_2 \text{ und } w_2 \neq \varepsilon \rightarrow w_1 \notin L$$

Grund: bereits bei  $w_1$  ist Keller leer, also keine Fortsetzung möglich

## 4.3 Kellerautomat kann beliebige kontextfreie Sprache akzeptieren

2 Varianten:

1. Top-Down:

Starte mit Satzsymbol im Keller, nutze den Keller für Ableitungsprozess, entferne entstehende Terminalsymbole paarweise mit Eingabe  
→ Linksableitung

2. Bottom-Up:

Überführe Terminalsymbole in Keller, wende Grammatik-Regeln rückwärts an und beende, falls Satzsymbol im Keller steht  
→ Rechtsableitung

# Top-Down-akzeptierender Kellerautomat

$\delta(z_0, \varepsilon, \perp) = \{(z_1, \perp s)\}$	für Satzsymbol $s$	INIT
$\delta(z_1, \varepsilon, h) = \{(z_1, w^{-1}) \mid [h, w] \in R\}$	für Hilfssymbol $h$	EXPAND
$\delta(z_1, x, x) = \{(z_1, \varepsilon)\}$	für Terminalzeichen $x$	MATCH

Akzeptierender Lauf des Automaten entspricht Linksableitung  
(das am weitesten links stehende Hilfssymbol steht jeweils am  
Kellereingang)

# Beispiel: Top Down

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow x$

$F \rightarrow ( E )$

$x+x*x$	$z_1$	$\perp E$	INIT	
$x+x*x$	$z_1$	$\perp T+E$	EXPAND	$E \rightarrow E+T$
$x+x*x$	$z_1$	$\perp T+T$	EXPAND	$E \rightarrow T$
$x+x*x$	$z_1$	$\perp T+F$	EXPAND	$T \rightarrow F$
$x+x*x$	$z_1$	$\perp T+x$	EXPAND	$F \rightarrow x$
$+x*x$	$z_1$	$\perp T+$	MATCH	
$x*x$	$z_1$	$\perp T$	MATCH	
$x*x$	$z_1$	$\perp F*T$	EXPAND	$T \rightarrow T*F$
$x*x$	$z_1$	$\perp F*F$	EXPAND	$T \rightarrow F$
$x*x$	$z_1$	$\perp F*x$	EXPAND	$F \rightarrow x$
$*x$	$z_1$	$\perp F*$	MATCH	
$x$	$z_1$	$\perp F$	MATCH	
$x$	$z_1$	$\perp x$	EXPAND	$F \rightarrow x$
	$z_1$	$\perp$	MATCH	

# Bottom-Up-akzeptierender Kellerautomat

$\delta(z_0, x, k) = \{(z_0, kx)\}$	für beliebiges Kellersymbol $k$ und Terminalzeichen $x$	SHIFT
$\delta(z_0, \varepsilon, w) = \{(z_0, h)\}$ (*)	für Regel $h \rightarrow w$	REDUCE
$\delta(z_0, \varepsilon, s) = \{(z_1, \varepsilon)\}$		ACCEPT

(\*) Kurzform für (sei  $w = y_1 \dots y_k$ ):  $[z_{h,w,k}, \varepsilon] \in \delta(z_0, \varepsilon, y_k)$ ,  
 $\{[z_{h,w,k-1}, \varepsilon]\} = \delta(z_0, \varepsilon, y_{k-1}), \dots$   
 $\{[z_{h,w,2}, \varepsilon]\} = \delta(z_0, \varepsilon, y_2)$ ,  
 $\{[z_0, h]\} = \delta(z_0, \varepsilon, y_1)$ .

Akzeptierender Lauf des Automaten entspricht (rückwärts  
gelesen) Rechtsableitung

# Beispiel: Bottom Up

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow x$

$F \rightarrow ( E )$

$x+x*x$	$z_0$	$\perp$	START
$+x*x$		$z_0$	$\perp x$ SHIFT $x$
$+x*x$	$z_0$	$\perp F$	REDUCE $F \rightarrow x$
$+x*x$	$z_0$	$\perp T$	REDUCE $T \rightarrow F$
$+x*x$	$z_0$	$\perp E$	REDUCE $E \rightarrow T$
$x*x$	$z_0$	$\perp E+$	SHIFT $+$
$*x$	$z_0$	$\perp E+x$	SHIFT $x$
$*x$	$z_0$	$\perp E+F$	REDUCE $F \rightarrow x$
$*x$	$z_0$	$\perp E+T$	REDUCE $T \rightarrow F$
$x$	$z_0$	$\perp E+T^*$	SHIFT $*$
	$z_0$	$\perp E+T^*x$	SHIFT $x$
	$z_0$	$\perp E+T^*F$	REDUCE $F \rightarrow x$
	$z_0$	$\perp E+T$	REDUCE $T \rightarrow T^*F$
	$z_0$	$\perp E$	REDUCE $E \rightarrow E+T$
	$z_1$	$\perp$	ACCEPT

## 4.4 Deterministische Spracherkennung Top-Down

Problem: muss für jedes Hilfssymbol  $h$  am Kellereingang entscheiden, welche Regel anzuwenden ist.

Wann? Wenn kompletter Text links von  $h$  bereits verarbeitet ist

Womit? Mit den nächsten  $k$  Zeichen der Eingabe  
(meistens, also auch hier:  $k=1$ )

→LL( $k$ )-Grammatiken

L – left to right analysis

L – leftmost derivation

$k$  –  $k$  characters lookahead



# LL(k)-Eigenschaft

G ist LL(k), falls für je zwei Linksableitungen

$$\begin{aligned} s \rightarrow^* u h v_1 \rightarrow u w_1 v_1 \rightarrow^* u t_1 \quad & u, t_1, t_2 \in X^*, h \in H \\ \text{und} \\ s \rightarrow^* u h v_2 \rightarrow u w_2 v_2 \rightarrow^* u t_2 \quad & v_1, v_2, w_1, w_2 \in (X \cup H)^* \end{aligned}$$

mit  $w_1 \neq w_2$  gilt:  $\text{first}_k(t_1) \neq \text{first}_k(t_2)$

$\text{first}_k(w)$ : w, nach k Zeichen abgeschnitten

# First<sub>1</sub>/Follow<sub>1</sub>-Mengen

- $\text{first}(\varepsilon) = \{\varepsilon\}$
- $\text{first}(x) = \{x\}$  für Terminalsymbol  $x$
- $\text{first}(h) = \bigcup_{h \rightarrow w \in R} \text{first}(w)$
- $\text{first}(w_1 w_2) = \text{first}(w_1)$ , falls aus  $w_1$  nicht  $\varepsilon$  ableitbar ist  
=  $(\text{first}(w_1) \setminus \{\varepsilon\}) \cup \text{first}(w_2)$ , falls doch

$\text{follow}(h)$  = Menge aller Zeichen, die in einer Ableitung unmittelbar hinter  $h$  stehen können

- $\langle \text{EOT} \rangle \in \text{follow}(s)$
- Wenn  $h' \rightarrow w_1 h w_2 \in R$ , so  $\text{first}(w_2) \setminus \{\varepsilon\} \subseteq \text{follow}(h)$
- Wenn  $h' \rightarrow w_1 h w_2 \in R$  und  $\varepsilon \in \text{first}(w_2)$ , so  $\text{follow}(h') \subseteq \text{follow}(h)$

# $\varepsilon$ -Ableitbarkeit

$\varepsilon$  ist ableitbar aus

- Terminalzeichen  $x$ : niemals
- Wort  $\varepsilon$ : immer
- Wenn aus  $y_1, \dots, y_k$   $\varepsilon$  ableitbar ist, so aus  $w = y_1 \dots y_k$
- Wenn aus  $w$   $\varepsilon$  ableitbar ist und  $h \rightarrow w \in R$ , so aus  $h$   $\varepsilon$  ableitbar

# LL(1)-Parser

Auflösung des Nichtdeterminismus in einem Top-Down-Parser für LL(1)-Grammatiken

Einzigster Nichtdeterminismus im Top-Down-Verfahren:  
Wenn Hilfsymbol  $h$  am Kellereingang, mit welcher Regel soll expandiert werden?

Antwort für LL(1)

EXPAND mit  $h \rightarrow w$ , falls nächste Eingabe aus  $\text{first}(w)$

EXPAND mit  $h \rightarrow w$ , falls aus  $w$   $\varepsilon$  ableitbar ist und  
nächste Eingabe aus  $\text{follow}(h)$

LL(1)-Eigenschaft stellt sicher, dass diese Entscheidungen eindeutig sind (d.h., die betreffenden first- und follow-Mengen disjunkt)!

# Beispiel

		Expand	bei Eingabe
$E \rightarrow TR$	$\text{first}(E) = \{\text{id}, (\}$	$E \rightarrow TR$	id, (
$R \rightarrow *E$	$\text{first}(T) = \{\text{id}, (\}$	$R \rightarrow *E$	*
$R \rightarrow /E$	$\text{first}(R) = \{*, /, \varepsilon\}$	$R \rightarrow /E$	/
$R \rightarrow \varepsilon$		$R \rightarrow \varepsilon$	#, )
$T \rightarrow \text{id}$	$\text{follow}(E) = \{\#, )\}$	$T \rightarrow \text{id}$	id
$T \rightarrow (E)$	$\text{follow}(T) = \{*, /, ), \#\}$	$T \rightarrow (E)$	(
	$\text{follow}(R) = \{\#, )\}$		

$\varepsilon$  ableitbar aus R.

# Beispiel: nicht LL(1)

		Expand	bei Eingabe
$E \rightarrow E+T$	$\text{first}(E) = \{x, (\}$	$E \rightarrow E+T$	$x, ($
$E \rightarrow T$	$\text{first}(T) = \{x, (\}$	$E \rightarrow T$	$x, ($
$T \rightarrow T * F$	$\text{first}(F) = \{x, (\}$		
$T \rightarrow F$		...	
$F \rightarrow x$	$\text{follow}(E) = \{\#, ), +\}$		
$F \rightarrow (E)$	$\text{follow}(T) = \{\#, ), +, *\}$		
	$\text{follow}(F) = \{\#, ), +, *\}$		
	$\varepsilon$ nicht ableitbar.		

Bem: linksrekursive  
Grammatiken sind  
niemals LL(1)!

Verletzung der LL(1)-Eigenschaft:

$E \rightarrow E+T \rightarrow T+T \xrightarrow{*} x+x$

$E \rightarrow E+T \rightarrow E+T+T \xrightarrow{*} x+x+x$       $\text{first}(x+x) = \text{first}(x+x+x) = \{x\}$

# Implementation eines Top-Down-Parsers

Methode des rekursiven Abstiegs:

- Pro Hilfssymbol eine Prozedur
- ruft Prozeduren für Hilfssymbole der rechten Seite
- Terminalzeichen werden mit Eingabe verglichen

# Beispiel

$E \rightarrow TR$

$R \rightarrow *E$

$R \rightarrow /E$

$R \rightarrow \varepsilon$

$T \rightarrow \text{id}$

$T \rightarrow (E)$

char current;

```
bool E() {
    bool b;
    b = T();
    if(!b)
        return false;
    b = R();
    return b;
}
```

```
bool R() {
    bool b;
    if(current == '*') {
        read();
        b = E();
        return b;
    }
    if(current == '/') {
        read();
        b = E();
        return b;
    }
    return true;
}
```

```
bool T() {
    bool b;
    if(current == 'id') {
        read();
        return true;
    }
    if(current == '(') {
        read();
        b = E();
        if(!b)
            return false;
        if(current == ')') {
            read();
            return true;
        } else return false;
    }
    return false;
}
```



## 4.5 Deterministische Spracherkennung Bottom-Up

Erinnerung:

Aktionen SHIFT      (Terminalzeichen in Keller überführen)  
                 REDUCE (Regel rückwärts anwenden)

Problem: Wann SHIFT, wann REDUCE

→ SHIFT/REDUCE-Konflikt

Welche Regel beim REDUCE

→ REDUCE/REDUCE-Konflikt

# LR(k)-Grammatiken

L - left to right analysis

R - rightmost derivation

k - k characters lookahead (meist, auch hier:  $k = 1$ )

LR(k)-Eigenschaft:

G heißt LR(k), falls:  $(x, y \dots \in (X \cup H)^*, h \in H^*, z, z' \dots \in X^*)$

$$s \rightarrow^* xhz \rightarrow xyz$$

Wenn  $s \rightarrow^* x'h'z' \rightarrow xyz''$ , so  $x=x', h=h', z'=z''$ .

$$\text{first}_k(z) = \text{first}_k(z'')$$

# Verschiedene Bottom-Up-Verfahren

LR(0)      Entscheidung von Konflikten anhand Zustand

SLR(1)    ... + Follow-Mengen

LALR(1) ... + exact right context (= zustandsabhängige  
Verbesserung der Follow-Mengen)

LR(1)      ... + Zustände werden nach rechtem Kontext  
differenziert

# Zustände eines deterministischen Bottom-Up-Parsers (LR(0),SLR(1),LALR(1))

... repräsentieren Information über Kellerinhalt

LR(0)-Item = Regel mit „Marker“  $\circ$  auf rechter Seite

z.B.  $E \rightarrow E \circ + T$

Lies: „Im Keller steht oben E;

falls der folgende Text  $+ T$  entsprechen sollte,  
könnte dieser mit der gegebenen Regel zu E  
reduziert werden“

Zustand = eine Menge von LR(0)-Items

# Bottom-Up-Parsing mit LR(0)-Items

Anfangszustand:  $\text{closure}(\rightarrow \circ s)$

$\text{closure}$ : wann immer Marker vor einem Hilfssymbol  $h$  steht, wird für jede Regel  $h \rightarrow w$  Item  $h \rightarrow \circ w$  hinzugenommen

In Zustand  $z$ :

- Aktion SHIFT  $x \rightarrow z'$ , falls in mind. 1 Item Marker direkt links von  $x$  steht  
 $z' := \text{closure}(\{h \rightarrow w_1 x \circ w_2 \mid h \rightarrow w_1 \circ x w_2 \in z\})$
- Aktion REDUCE  $r \rightarrow z'$  (für  $r = h \rightarrow w$ ), falls  $h \rightarrow w \circ \in z$   
 $|w|$  Zustände vom Keller entfernen, mit dem neuen Kellereingang  
SHIFT-Aktion mit  $h$  durchführen

$S \rightarrow E\#$      $E \rightarrow E - T$      $E \rightarrow T$      $T \rightarrow \text{id}$      $T \rightarrow (E)$

## Beispiel

$z_0 \rightarrow \circ S$      $\text{id}$     SHIFT  $z_1$   
 $S \rightarrow \circ E\#$      $($     SHIFT  $z_2$   
 $E \rightarrow \circ E - T$      $T$     SHIFT  $z_3$   
 $E \rightarrow \circ T$      $E$     SHIFT  $z_4$   
 $T \rightarrow \circ \text{id}$      $S$     SHIFT  $z_5$   
 $T \rightarrow \circ (E)$

$z_1 \ T \rightarrow \text{id} \circ$     REDUCE  $T \rightarrow \text{id}$

$z_2 \ T \rightarrow (\circ E)$      $\text{id}$     SHIFT  $z_1$   
 $E \rightarrow \circ E - T$      $($     SHIFT  $z_2$   
 $E \rightarrow \circ T$      $T$     SHIFT  $z_3$   
 $T \rightarrow \circ \text{id}$      $E$     SHIFT  $z_6$   
 $T \rightarrow \circ (E)$

$z_3 \ E \rightarrow T \circ$     REDUCE  $E \rightarrow T$

$z_4 \ S \rightarrow E \circ \#$      $-$     SHIFT  $z_7$   
 $E \rightarrow E \circ - T$      $\#$     SHIFT  $z_8$

$z_5 \rightarrow S \circ$     ACCEPT

$z_6 \ E \rightarrow E \circ - T$      $-$     SHIFT  $z_8$   
 $T \rightarrow (E \circ)$      $)$     SHIFT  $z_9$

$z_7 \ E \rightarrow E - \circ T$      $\text{id}$     SHIFT  $z_1$   
 $T \rightarrow \circ \text{id}$      $($     SHIFT  $z_2$   
 $T \rightarrow \circ (E)$      $T$     SHIFT  $z_{10}$

$z_8 \ S \rightarrow E\# \circ$     REDUCE  $S \rightarrow E\#$

$z_9 \ T \rightarrow (E) \circ$     REDUCE  $T \rightarrow (E)$

$z_{10} \ E \rightarrow E - T \circ$     REDUCE  $E \rightarrow E - T$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow id$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id:    SHIFT  $z_1$

id – id – id#       $z_0$      $\perp$

$r_1: S \rightarrow E\#$     $r_2: E \rightarrow E-T$     $r_3: E \rightarrow T$     $r_4: T \rightarrow id$     $r_5: T \rightarrow (E)$   
**Beispiel: Ergebnis in Tabellenform**

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id:      REDUCE  $r_4$  (a)

– id – id#       $z_1 \perp z_0$



$r_1: S \rightarrow E\#$     $r_2: E \rightarrow E-T$     $r_3: E \rightarrow T$     $r_4: T \rightarrow id$     $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_4$  (b)

– id – id#       $z_0$        $\perp$

$r_1: S \rightarrow E\#$     $r_2: E \rightarrow E-T$     $r_3: E \rightarrow T$     $r_4: T \rightarrow id$     $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_3$  (a)

– id – id#       $z_3$        $\perp z_0$

$r_1: S \rightarrow E\#$     $r_2: E \rightarrow E-T$     $r_3: E \rightarrow T$     $r_4: T \rightarrow id$     $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_3$  (b)

– id – id#       $z_0$        $\perp$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow \text{id}$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: SHIFT  $z_7$

– id – id#       $z_4 \perp z_0$

$r_1: S \rightarrow E\#$   
 $r_2: E \rightarrow E-T$   
 $r_3: E \rightarrow T$   
 $r_4: T \rightarrow \text{id}$   
 $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: SHIFT  $z_1$

id – id#       $z_7 \perp z_0 z_4$

$r_1: S \rightarrow E\#$   
 $r_2: E \rightarrow E-T$   
 $r_3: E \rightarrow T$   
 $r_4: T \rightarrow id$   
 $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_4$  (a)

– id#       $z_1 \perp z_0 z_4 z_7$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow id$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_4$  (b)

– id#       $z_7 \perp z_0 z_4$

$r_1: S \rightarrow E\#$     $r_2: E \rightarrow E-T$     $r_3: E \rightarrow T$     $r_4: T \rightarrow \text{id}$     $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_2$  (a)

– id#       $z_{10}$        $\perp z_0 z_4 z_7$



$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow id$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_2$  (b)

– id#       $z_0$        $\perp$

$r_1: S \rightarrow E\#$   $r_2: E \rightarrow E-T$   $r_3: E \rightarrow T$   $r_4: T \rightarrow id$   $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: SHIFT  $z_7$

– id#       $z_4$        $\perp z_0$

$r_1: S \rightarrow E\#$   
 $r_2: E \rightarrow E-T$   
 $r_3: E \rightarrow T$   
 $r_4: T \rightarrow \text{id}$   
 $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: SHIFT  $z_1$

id#       $z_7$        $\perp z_0 z_4$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow id$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	$S z_1$	$R r_4$	$S z_1$	$R r_3$		ACC		$S z_1$	$R r_1$	$R r_5$	$R r_2$
-					$S z_7$		$S z_8$				
(	$S z_2$		$S z_2$					$S z_2$			
)							$S z_9$				
#					$S z_8$						
T	$S z_3$		$S z_3$					$S z_{10}$			
E	$S z_4$		$S z_6$								
S	$S z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_4$

$\#$        $z_1$        $\perp z_0 z_4 z_7$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow \text{id}$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_2$

#       $z_{10}$        $\perp z_0 z_4 z_7$

$r_1: S \rightarrow E\#$   
 $r_2: E \rightarrow E-T$   
 $r_3: E \rightarrow T$   
 $r_4: T \rightarrow \text{id}$   
 $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: SHIFT  $z_8$

#       $z_4$        $\perp z_0$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow \text{id}$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: REDUCE  $r_1$

$z_8 \perp z_0 z_4$

$r_1: S \rightarrow E\#$   
  $r_2: E \rightarrow E-T$   
  $r_3: E \rightarrow T$   
  $r_4: T \rightarrow \text{id}$   
  $r_5: T \rightarrow (E)$

## Beispiel: Ergebnis in Tabellenform

	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$
id	S $z_1$	R $r_4$	S $z_1$	R $r_3$		ACC		S $z_1$	R $r_1$	R $r_5$	R $r_2$
-					S $z_7$		S $z_8$				
(	S $z_2$		S $z_2$					S $z_2$			
)							S $z_9$				
#					S $z_8$						
T	S $z_3$		S $z_3$					S $z_{10}$			
E	S $z_4$		S $z_6$								
S	S $z_5$										

Lauf mit Eingabe id – id – id: ACCEPT

$z_5 \perp z_0$



# LR(0), SLR(1)-Grammatiken

- Die Beispielgrammatik war eine LR(0)-Grammatik:
  - keine SHIFT-REDUCE-Konflikte
  - keine REDUCE-REDUCE-Konflikte

SLR(1): Konflikte auflösbar mittels FOLLOW-Mengen:

Beispiel: SHIFT-REDUCE

Item

$h_1 \rightarrow w_1 \circ x w_2$      $x$ : SHIFT  $x$

$h_2 \rightarrow w_3 \circ$         FOLLOW( $h_2$ ): REDUCE

# Jenseits von SLR(1)

$S \rightarrow id$     $S \rightarrow V := E$     $V \rightarrow id$     $E \rightarrow V$     $E \rightarrow n$

$z_0:$

$\rightarrow \circ S$

$S \rightarrow \circ id$                        $id$  SHIFT  $z_1$

$S \rightarrow \circ V := E$                       ....

$V \rightarrow \circ id$

$z_1:$

$S \rightarrow id \circ$

$V \rightarrow id \circ$

$FOLLOW(S) = \varepsilon$

$FOLLOW(V) = \{\varepsilon, :=\}$

aber:  $\varepsilon \in FOLLOW(V)$  nur wegen  $S \rightarrow V := E$   
– in  $z_1$  gar nicht relevant!

Lösung: LR(1)-Items

# LR(1)-Items

Idee: Erweiterung der Items um rechten Kontext

$[h \rightarrow w_1 \circ w_2, x]$  „ $w_1$  im Keller; wenn jetzt noch  $w_2$  kommt und danach ein  $x$  folgt, kann zu  $h$  reduziert werden“

Start:  $[\rightarrow \circ S, <EOT>]$

Closure: mit  $[h \rightarrow w_1 \circ h' w_2, x]$  auch Items zu Regeln für  $h'$ :

- $[h' \rightarrow w_3, y]$  mit  $y \in \text{FIRST}(w_2)$
- $[h' \rightarrow w_3, x]$ , falls  $\varepsilon \in \text{FIRST}(w_2)$

$r_1: S \rightarrow id$     $r_2: S \rightarrow V := E$     $r_3: V \rightarrow id$     $r_4: E \rightarrow V$     $r_5: E \rightarrow n$

## Beispiel

$z_0: [\rightarrow \circ S, \#]$ $[S \rightarrow \circ id, \#]$ $[S \rightarrow \circ V := E, \#]$ $[V \rightarrow \circ id, :=]$	$id \ S \ z_1$ $V \ S \ z_2$ $S \ S \ z_3$	$z_4: [S \rightarrow V := \circ E, \#]$ $[E \rightarrow \circ V, \#]$ $[E \rightarrow \circ n, \#]$ $[V \rightarrow \circ id, \#]$	$n \ S \ z_5$ $id \ S \ z_6$ $V \ S \ z_7$ $E \ S \ z_8$
$z_1: [S \rightarrow id \circ, \#]$ $[V \rightarrow id \circ, :=]$	$\# \ R \ r_1$ $:= \ R \ r_3$	$z_5: [E \rightarrow n \circ, \#]$ $z_6: [V \rightarrow id \circ, \#]$	$\# \ R \ r_5$ $\# \ R \ r_3$
$z_2: [S \rightarrow V \circ := E, \#]$	$:= \ S \ z_4$	$z_7: [E \rightarrow V \circ, \#]$	$\# \ R \ r_4$
$z_3: [\rightarrow S \circ, \#]$	$\# \ ACC$	$z_8: [S \rightarrow V := \circ E, \#]$	$\# \ R \ r_2$

# LR(1) und LALR(1)-Verfahren

LR(1): benutzt Mengen von LR(1)-Items als Zustände  
→ Problem: große Zustandszahl

LALR(1): fasst Zustände zusammen, wenn sie sich nur in den rechten Kontexten unterscheiden (hinter dem Komma stehen dann *Mengen* von Terminalzeichen)

- Zustandszahl wie bei LR(0) und SLR(1), aber rechter Kontext genauer als FOLLOW
- Prinzip der am weitesten verbreiteten Compilergeneratoren yacc/bison

# Implementierung eines LR/...-Parsers

- unabhängig von LR(0), LR(1), LALR(1), SLR(1)
- tabellengesteuert (Zustand x Eingabe)
  - Einträge SHIFT <neuer Zustand>, REDUCE <regel> und ACCEPT
  - Freie Felder entsprechen Syntaxfehlern

# Implementierung

```
char current;
entry table[#states][#char];
state stack[large_enough];
stackpointer = 0;

stack[0] = z0;

while(true) {
    switch(table[stack[stackpointer]][current]) {
        case SHIFT zi:    read();
                           stack[++stackpointer] = zi;
                           break;
        case REDUCE l→r:  stackpointer -= length(r);
                           Z = table[stack[stackpointer]][1];
                           stack[++stackpointer] = Z;
                           break;
        case ACCEPT:      if(stackpointer == 0
                           && stack[0] == z0
                           && current == <EOT>) return true;
                           else return false;
                           break;
        default:          return false;
    }
}
```

## 4.6 Normalformen/Transformationen

1. Beseitigung von  $\varepsilon$ -Regeln, Kettenregeln
  - zeigt, dass  $CF \subseteq CS$ , verbessert Laufzeit
2. Chomsky-Normalform
  - hilfreich für Komplexität, Pumping-Lemma
3. Greibach-Normalform
  - zeigt, dass  $\varepsilon$ -freie Kellerautomaten möglich sind



# Beseitigung von Kettenregeln

- Für jede zyklische Umbenennung  $h_1 \rightarrow \dots \rightarrow h_i \rightarrow h_1$ :
  - Ersetze überall  $h_i$  durch  $h_1$  (für alle  $i$ )
  - Streiche  $h_1 \rightarrow h_1$
- Numeriere Hilfssymbole so, dass, wenn  $h_i \rightarrow h_j$ , so  $i < j$
- Für alle  $j$  absteigend:
  - für alle  $i < j$  aufsteigend
    - Falls Regel  $h_i \rightarrow h_j$  vorhanden, so
      - streiche  $h_i \rightarrow h_j$
      - für jede Regel  $h_j \rightarrow w$ 
        - » für jede Regel  $h^* \rightarrow w_1 h_i w_2$ 
          - baue zusätzliche Regel  $h^* \rightarrow w_1 w w_2$

# Beseitigung von $\varepsilon$ -Regeln

- Bilde  $E = \{h \mid h \rightarrow^* \varepsilon\}$
- Streiche alle Regeln der Form  $h \rightarrow \varepsilon$
- Falls  $s \in E$ , führe neues Startsymbol  $s'$  ein mit Regeln  $s' \rightarrow s$  und  $s' \rightarrow \varepsilon$
- Für jede Regel  $h \rightarrow w$ :
  - Bilde alle Wörter, die nicht leer sind und aus  $w$  durch Streichen einiger Symbole (mind. 0) aus  $E$  entstehen
  - Für jedes solche Wort  $w'$  führe Regel  $h \rightarrow w'$  ein

# Chomsky-Normalform

- Jede Regel hat eine der Formen:
  - $h \rightarrow x$  ( $x$  Terminalzeichen) oder
  - $h \rightarrow h_1 h_2$  ( $h_1, h_2$  Hilfssymbole)
- Ausnahme:  $s \rightarrow \varepsilon$  falls  $s$  nie auf rechten Seiten von Regeln auftritt

# Konstruktion

- Für jedes Terminalzeichen  $a$ , neues Hilfssymbol  $h_a$  mit Regel  $h_a \rightarrow a$ ; ersetze jedes Terminalzeichen in originalen Regeln durch  $h_a$ .
- Beseitige  $\varepsilon$  (eben gesehen)
- Beseitige Kettenregeln (eben gesehen)
- Für jede Regel der Form  $h \rightarrow h_1 h_2 w$  mit  $\text{length}(w) > 0$ :
  - ersetze durch  $h \rightarrow h' w$      $h' \rightarrow h_1 h_2$  mit neuem  $h'$

# Greibach-Normalform

- Alle Regeln haben die Form  $h \rightarrow xw$  mit  $x \in X$ ,  $w \in H^*$   
(Ausnahme wie immer:  $s \rightarrow \varepsilon$ )

## 4.7 Pumping-Lemma für kontextfreie Sprachen

Ziel: Nachweis, dass  $L$  *nicht* kontextfrei.

Satz: Wenn  $L$  kontextfrei ist, so gibt es ein  $n$ , so dass jedes Wort  $w$  mit  $|w| > n$  zerlegt werden kann in

$$w = w_1 u w_2 v w_3$$

derart, dass  $|uv| > 0$  und für alle  $i$  auch

$$w_1 u^i w_2 v^i w_3 \in L$$

# Beweisidee

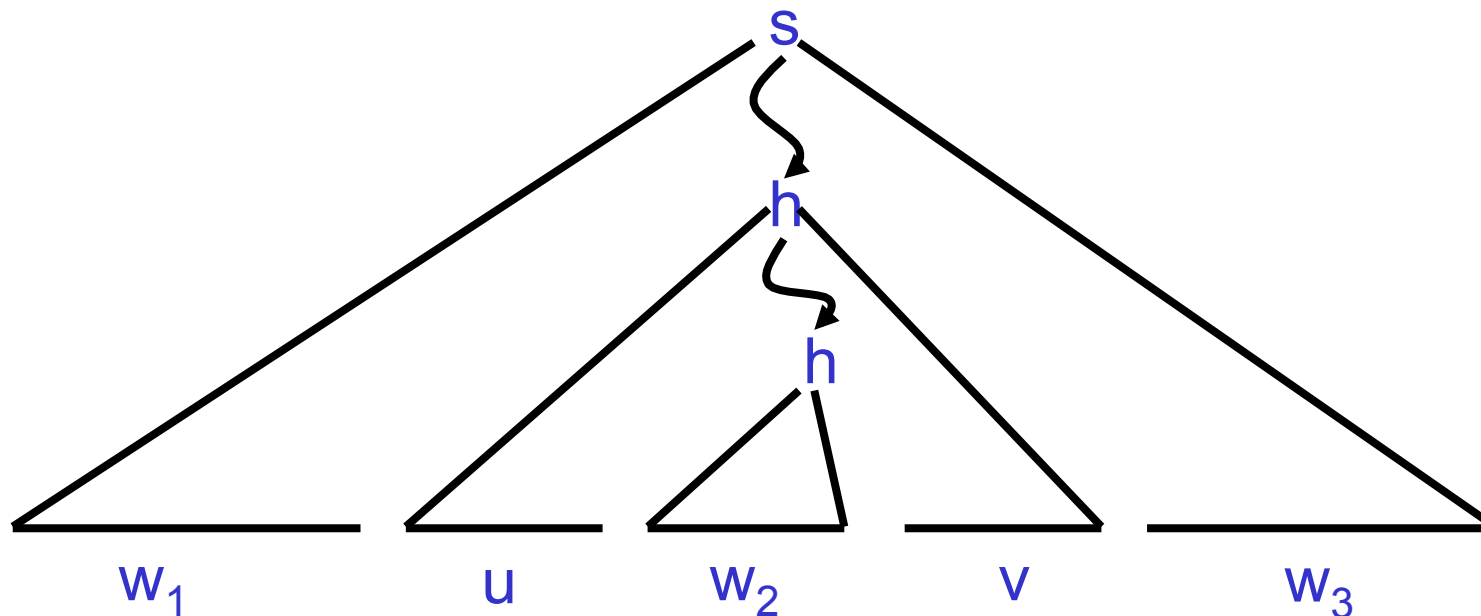
1. Gramatik in Chomsky-Normalform bringen
2.  $j = |H|$   
Setzen  $n = 2^{j+1}$

Was hilft das?

## Beweisidee

Wenn  $|w| > n$ , dann hat Ableitungsbaum Tiefe  $> j$ , weil Baum mit Verzweigungsgrad 2 und Tiefe  $j$  nur max.  $(2^{j+1}-1)$  Knoten haben kann.

Baum mit Tiefe  $> j$  enthält ein mind. ein  $h$  doppelt auf Pfad in die Tiefe.





# Anwendung

Zeigen:  $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$  ist nicht kontextfrei.

Angenommen,  $L$  wäre kontextfrei. Sei  $w = a^i b^i c^i$  ausreichend groß. Wegen Balance der Buchstabenzahlen: Wenn

$u = a^j \rightarrow v$  muss  $b^j c^j$  sein  $\rightarrow$  Widerspruch

$u = b^j \rightarrow v$  muss  $a^j c^j$  sein  $\rightarrow$  Widerspruch

$u = c^j \rightarrow v$  muss  $a^j b^j$  sein  $\rightarrow$  Widerspruch

$u = a^j b^k \rightarrow v$  muss  $c^j$  sein  $\rightarrow$  Widerspruch

$u = b^j c^k \rightarrow v$  muss  $a^j$  sein  $\rightarrow$  Widerspruch

$u = a^j b^i c^k \rightarrow$  Widerspruch

$u = \varepsilon \rightarrow$  Widerspruch

Bem:  $L$  ist kontextsensitiv

## 4.8 Abschlusseigenschaften

CF abgeschlossen gegen

- Iteration
- Vereinigung
- Verkettung (Folie 34)

CF nicht abgeschlossen gegen

- Durchschnitt:  $\{a^m b^n c^n | n, m \in \mathbb{N}\} \cap \{a^n b^n c^m | n, m \in \mathbb{N}\} = \{a^n b^n c^n | n \in \mathbb{N}\}$
- Komplement:  $M \cap N = \overline{\overline{M} \cup \overline{N}}$

## 4.9 Komplexität

Wortproblem für eine Sprache  $L$ :

Gegeben:  $w \in X^*$       Frage:  $w \in L$ ?

Für LR-Sprachen: lineare Laufzeit

(Voraussetzung: Keine Umbenennungen  $H \rightarrow H'$ ,  $\varepsilon$ -frei).

(pro Buchstabe: genau ein SHIFT, höchstens so viele REDUCE wie Buchstaben).

Wie sieht es für allgemeine kontextfreie Sprachen aus?

# CYK-Algorithmus

- (Cocke, Younger, Kasami)

gegeben: Grammatik in Chomsky-NF, Wort  $w = x_1 \dots x_n$

Idee:

- für alle  $i, j$  Menge  $H_{ij}$  derjenigen Hilfssymbole berechnen, aus denen  $x_i \dots x_{i+j-1}$  ableitbar ist
- Berechnung Bottom-Up:
  - $H_{i1} = \{h_{x_i}\}$
  - $j > 1$ :  $H_{ij} = \bigcup_{l=1}^{j-1} \{h: h \rightarrow h' h'' \in R, h' \in H_{il}, h'' \in H_{i+j-l}\}$
  - $w \in L$  gdw.  $s \in H_{1n}$
  - Laufzeit:  $O(n^3)$     Speicherplatz:  $O(n^2)$

# Beispiel

- $S \rightarrow AC \quad S \rightarrow AB \quad C \rightarrow SB \quad A \rightarrow a \quad B \rightarrow b$
- $w = aabb$

$H_{i4}$	$\{S\}$			
$H_{i3}$		$\{C\}$		
$H_{i2}$		$\{S\}$		
$H_{i1}$	$\{A\}$	$\{A\}$	$\{B\}$	$\{B\}$
	a	a	b	b

# Kapitel 5

Jenseits von Sprachen,  
jenseits von Automaten

# Motivation

Diskrete Systeme spezifizieren und verifizieren

Diskretes System:

- Zustände (mit Eigenschaften)
- Zustandsübergänge (deterministisch oder nichtdeterministisch)
- Anfangszustand

... ist also im Prinzip ein Automat ...

... aber ...

- andere Fragestellungen
- (auch) andere Techniken
- nicht immer Sprachen



# Beispiel

Zwei Personen, Dusche, ein Stück Seife.

Ziele:

1. Niemals beide gleichzeitig unter der Dusche  
(Dusche = kritischer Bereich)
2. Wer will, darf irgendwann duschen

Protokoll (Mutex = mutual exclusion):

Anfang: Seife liegt auf dem Tisch

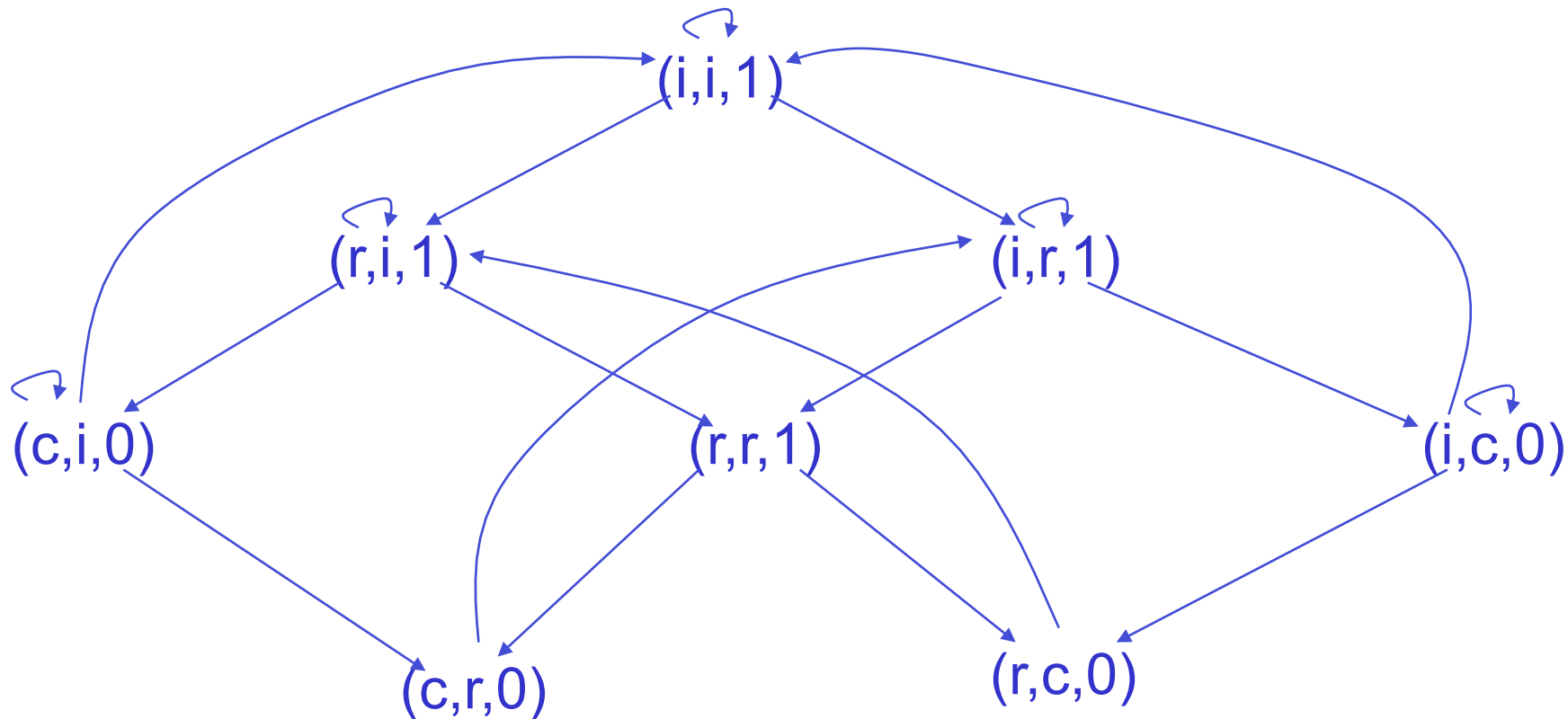
Wer duschen will, nimmt Seife vom Tisch (d.h., wartet, bis sie da liegt), geht duschen, duscht nicht ewig, legt danach Seife wieder hin.

# Beispiel

Zustand Duscher {i = idle, r = request, c = critical}

Zustand Seife {1 = auf dem Tisch, 0 = in Benutzung}

Ablauf = Lauf des Automaten, *unendlich*



# 5.1 Unendliche Pfade

Agenda:

- Temporale Logik = Spezifikation von Eigenschaften reaktiver Systeme = Mengen unendlicher Pfade
- Büchi-Automat = Akzeptierung einer Menge unendlicher Pfade
- Model Checking = Erfüllt System eine Eigenschaft?

# Was ist Temporale Logik?

Es geht nicht um Zeit  $\longrightarrow$  Real Time Logik

Es geht um Eigenschaften von Zuständen und deren Änderung in Systemabläufen

Es ist (bei uns) eine Erweiterung der Aussagenlogik

# Die Temporale Logik LTL

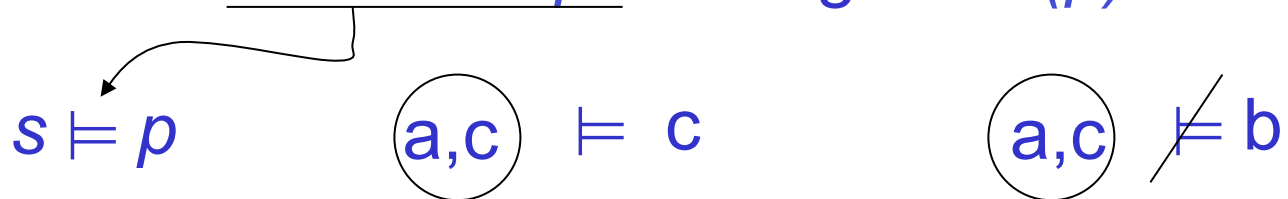
(Linear Time Temporal Logic)

Ausgangspunkt: Eigenschaften von Zuständen

Sei AP eine Menge von atomaren Aussagen.  
Jedes Element von AP ist eine  
**Zustandsformel**

Jeder Zustand liefert eine Belegung der atomaren Aussagen mit Wahrheitswerten:  $s(p) \in \{W, F\}$

Ein Zustand  $s$  erfüllt  $p \in AP$  gdw.  $s(p) = W$ .



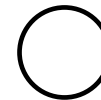
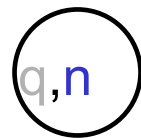
# Zustandseigenschaften

“Mailbox ist leer”

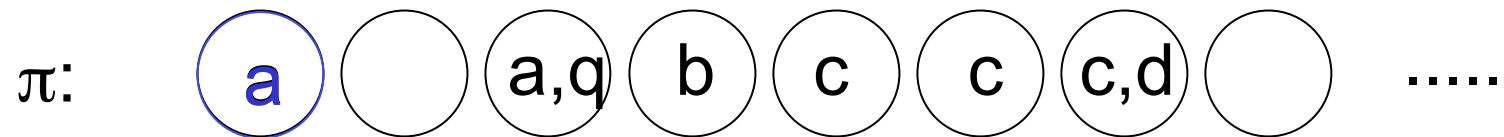
“bin bei Anweisung k”

“ $x[17] > 35$ ”

“nil dereferenziert”



## LTL -Elementare Formeln

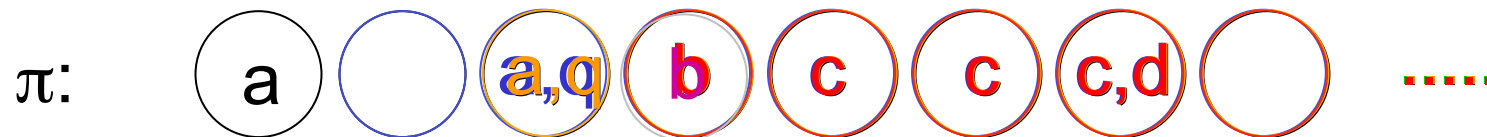


Jede Zustandsformel ist eine Formel

Ein Pfad erfüllt eine Zustandsformel gdw. sein erster Zustand sie erfüllt.

$\pi \models a$        $\pi \not\models c$        $\pi \models \text{true}$

# LTL - Der NACHFOLGER-Operator



Wenn  $\phi$  eine *Formel* ist, so auch  $X \phi$

Ein Pfad  $(s_0 s_1 s_2 s_3 \dots)$  erfüllt  $X \phi$  gdw.  
 $(s_1 s_2 s_3 s_4 \dots)$  erfüllt  $\phi$ .

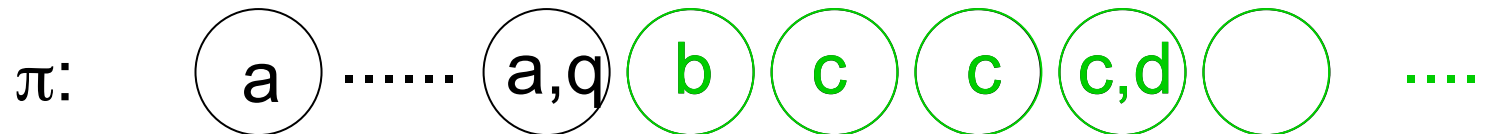
$$\pi \models X \neg a$$

$$\pi \models X X X b$$

Tautologien:  $X \neg \phi \Leftrightarrow \neg X \phi$   
 $X (\phi . \psi) \Leftrightarrow X \phi . X \psi$



# LTL - Der IRGENDWANN-Operator



Falls  $\phi$  eine *Formel* ist, so auch  $F \phi$

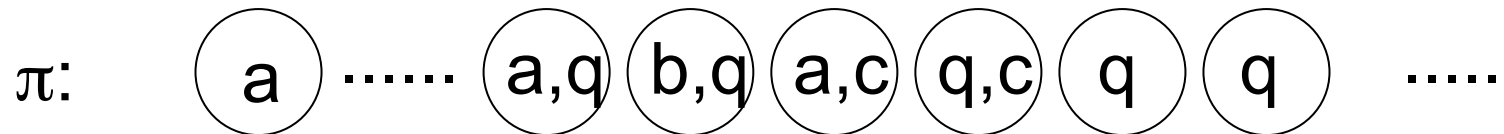
Ein Pfad  $(s_0 s_1 s_2 s_3 \dots)$  **erfüllt**  $F \phi$  gdw.  
 $(s_i s_{i+1} s_{i+2} s_{i+3} \dots)$  erfüllt  $\phi$ , für ein  $i \geq 0$ .

$$\pi \models F b$$

$$\pi \models X a \vee F( b \wedge X c)$$

Tautologien:  $\phi \Rightarrow F \phi$      $X \phi \Rightarrow F \phi$      $F \phi \Leftrightarrow F F \phi$   
 $X F \phi \Leftrightarrow F X \phi$      $F(\phi \boxed{?} \psi) \Rightarrow F \phi \boxed{?} F \psi$   
 $F \phi \Leftrightarrow \phi - X F \phi$      $F(\phi \boxed{?} \psi) \Leftrightarrow F \phi \boxed{?} F \psi$

## LTL - Der IMMER-Operator



Wenn  $\phi$  eine Formel ist, so auch  $G \phi$

Ein Pfad  $(s_0 s_1 s_2 s_3 \dots)$  erfüllt  $G \phi$  gdw.  
 $(s_i s_{i+1} s_{i+2} s_{i+3} \dots)$  erfüllt  $\phi$ , für *alle*  $i \geq 0$ .

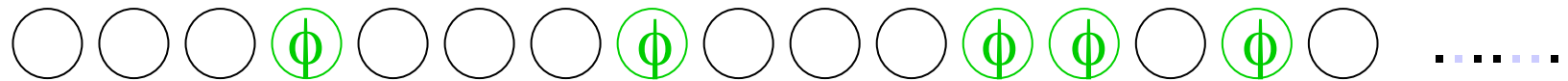
$$\pi \models G (a \vee q)$$

Tautologien:

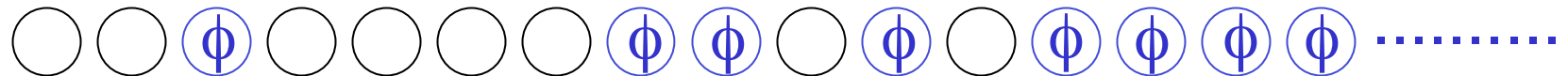
$$\begin{array}{ll}
 G \phi \Rightarrow \phi & G \phi \Rightarrow X \phi \\
 G \phi \Rightarrow F \phi & G \phi \Leftrightarrow G G \phi \quad \text{---} G \phi \Leftrightarrow F \text{---} \phi \\
 G \phi - G \psi \Rightarrow G (\phi - \psi) & G \phi . G \psi \Leftrightarrow G (\phi . \psi) \\
 G \phi \Leftrightarrow \phi . X G \phi &
 \end{array}$$

# Kombinationen von F and G

$G F \phi = \phi$  gilt unendlich oft



$F G \phi = \phi$  stabilisiert

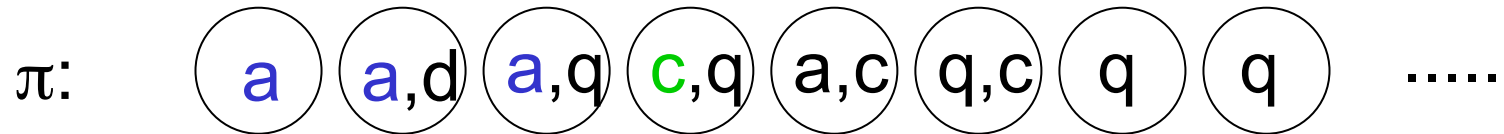


$G (\phi \Rightarrow F \psi) = \phi$  führt zu  $\psi$



Tautologien:  $F G F \phi \Leftrightarrow G F \phi$        $G F G \phi \Leftrightarrow F G \phi$

# LTL - Der BIS-Operator



Wenn  $\phi$  und  $\psi$  Formeln sind, so auch  $\phi \text{ U } \psi$

Ein Pfad  $(s_0 \ s_1 \ s_2 \ s_3 \ \dots)$  erfüllt  $\phi \text{ U } \psi$  gdw.  
 $(s_i \ s_{i+1} \ s_{i+2} \ s_{i+3} \ \dots)$  erfüllt  $\psi$ , für *ein*  $i \geq 0$ ,  
 und  $(s_j \ s_{j+1} \ s_{j+2} \ \dots)$  erfüllt  $\phi$ , für *alle*  $j < i$ .

$\pi \models a \text{ U } c$

Tautologien:  $\psi \Rightarrow \phi \text{ U } \psi$        $F \phi \Leftrightarrow \text{true U } \phi$   
 $\phi \text{ U } \psi \wedge \neg \psi \Rightarrow \phi$        $\phi \text{ U } \psi \Leftrightarrow \psi - (\phi \wedge X (\phi \text{ U } \psi))$

# Model Checking: Grundidee

LTL-Eigenschaft  $\triangleq$  Menge derjenigen Pfade, die  $\phi$  erfüllen

$$\rightarrow L_{\phi}^{\omega}$$

System  $\rightarrow$  Menge derjenigen Pfade, die  
realisiert werden können

$$\rightarrow L_{TS}^{\omega}$$

TS erfüllt  $\phi$  genau dann, wenn jeder Pfad in TS  $\phi$  erfüllt, d.h.

$$L_{TS}^{\omega} \subseteq L_{\phi}^{\omega}$$

# Wie kann man Inklusion entscheiden?

$$L_{TS}^{\omega} \subseteq L_{\phi}^{\omega}$$

$$\Leftrightarrow L_{TS}^{\omega} \cap \overline{L_{\phi}^{\omega}} = \emptyset$$

$$\Leftrightarrow L_{TS}^{\omega} \cap L_{\neg\phi}^{\omega} = \emptyset$$

- Agenda:
1. Automaten, die  $L_{TS}^{\omega}$  und  $L_{\neg\phi}^{\omega}$  akzeptieren
  2. Konstruktion eines Schnittautomaten
  3. Entscheidung, ob Automat die leere Sprache akzeptiert

Problem: Wir reden über **unendliche** Sequenzen!

# Büchi-Automaten

= Automaten mit einem für unendliche Sequenzen geeigneten Akzeptierungskriterium

$$B = [X, Z, Z_0, \delta, F]$$

$X$  – Alphabet

$Z$  – Zustandsmenge

$Z_0$  – Anfangszustandsmenge

$$\delta: Z \times X \rightarrow 2^Z$$

$F = \{F_1, \dots, F_n\}$ ,  $F_i \subseteq Z$  Akzeptierungsmengen

unendliche Sequenz  $\pi = x_0 x_1 x_2 \dots$  in  $X^\omega$

$B$  akzeptiert  $\pi$ : es ex. unendliche Sequenz  $\zeta = z_0 z_1 z_2 \dots$

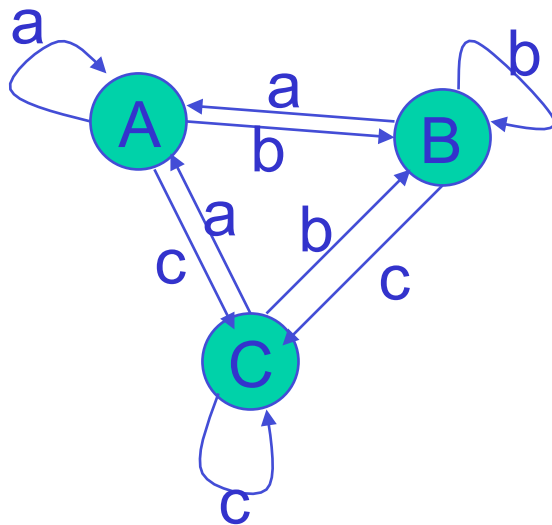
-  $z_0 \in Z_0$ ,  $z_{i+1} \in \delta(z_i, x_i)$ ,

- Für jedes  $F_i \in F$ :  $\zeta$  enthält unendlich oft Elemente aus  $F_i$

$\rightarrow L_B$

# Beispiele

$$X = \{a, b, c\}$$



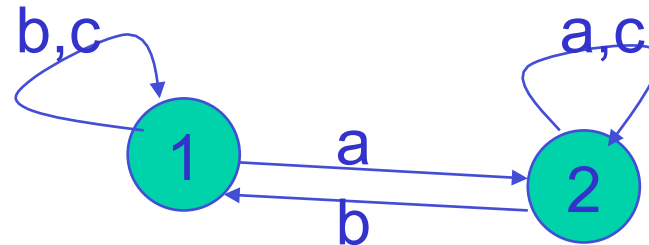
$$Z_0 = Z$$

$$F_1 = \{A, B\}$$

$$F_2 = \{A, C\}$$

$L_B = \{\pi \mid \text{in } \pi \text{ unendlich oft } a$   
 oder unendlich oft } b \text{ und}

unendlich oft } c\}



$$Z_0 = \{1\}$$

$$F_1 = \{1\}$$

$L_B = \{\pi \mid \text{nach jedem } a \text{ in } \pi$   
 kommt irgendwann } b\}



# System und Büchi-Automat

Eine Menge  $L^\omega$  von unendlichen Sequenzen heißt  $\omega$ -regulär, wenn es einen *endlichen* Büchi-Automaten gibt, der  $L^\omega$  akzeptiert.

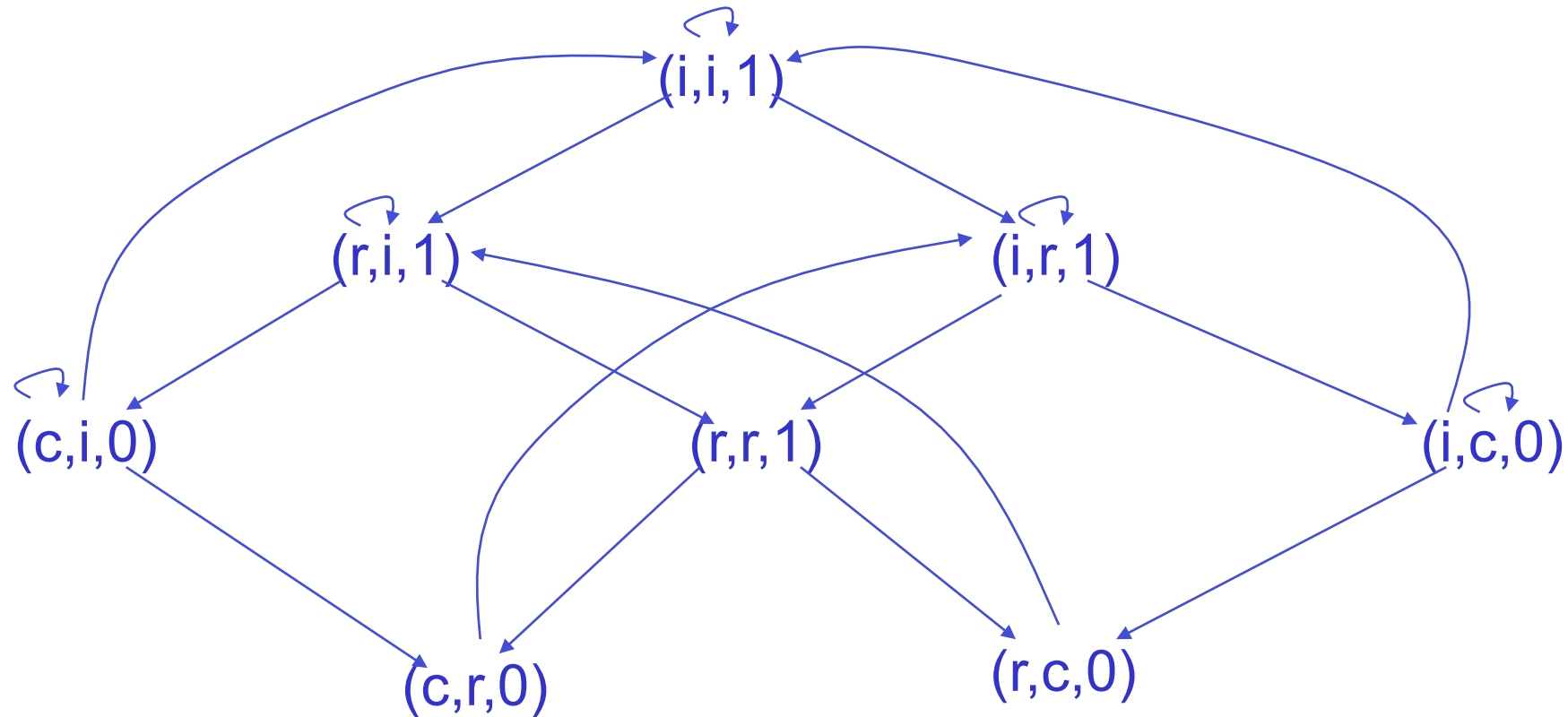
Beobachtung: Jede Menge von unendlichen Abläufen eines endlichen Automaten ist  $\omega$ -regulär.

$X$  = Vektoren von booleschen Werten, die als Wahrheitswerte der elementaren Aussagen interpretiert werden

$$Z = S, \quad Z_0 = I, \quad \delta(z, x) = \begin{cases} \{z' \mid [z, z'] \in E\}, & \text{falls } x \text{ den} \\ & \text{Wahrheitswerten der Aussagen} \\ & \text{in } z \text{ entspricht} \\ \emptyset, & \text{sonst} \end{cases}$$

$$F_1 = Z, \quad F = \{F_1\}$$

# Beispiel



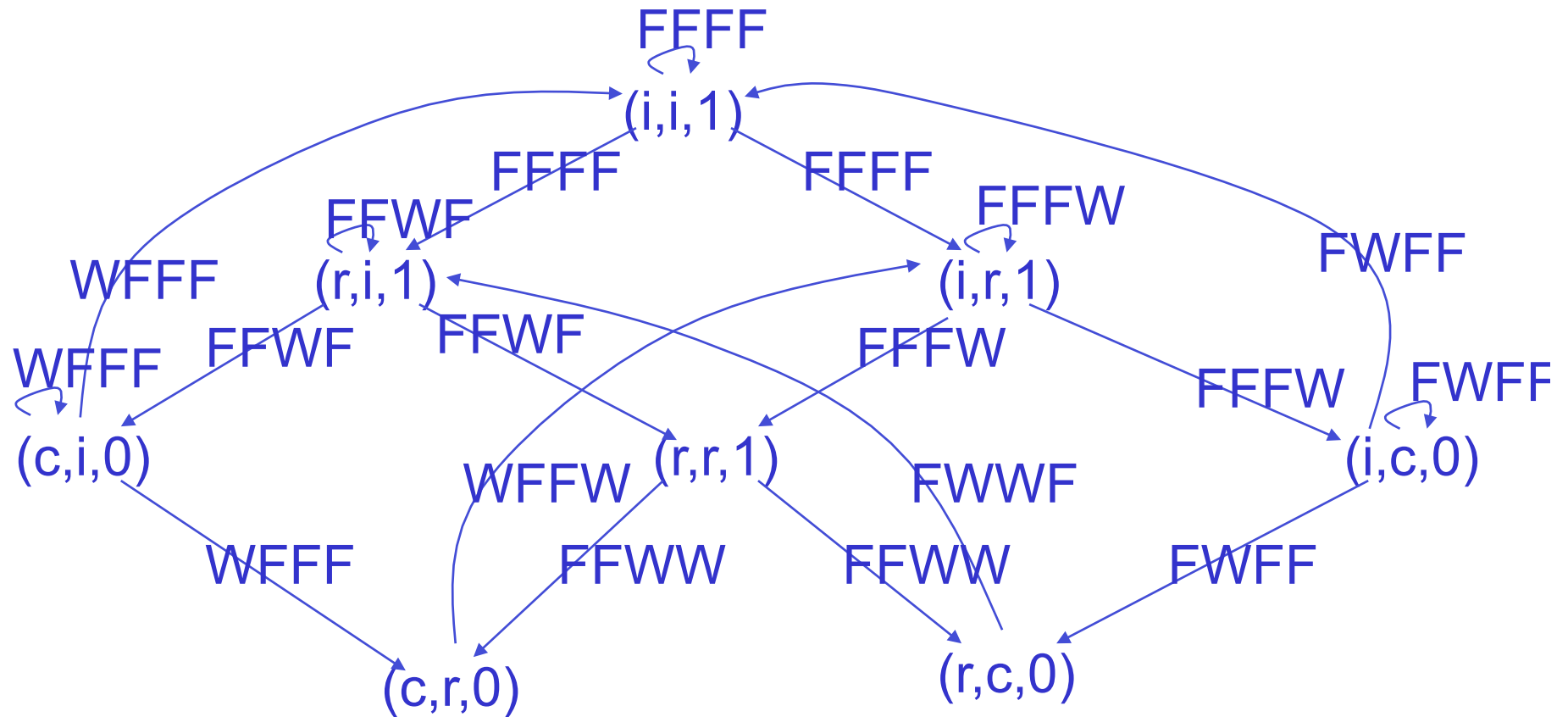
$G (pc1 \neq \text{"critical"} \vee pc2 \neq \text{"critical"})$

$G (pc1 = \text{"request"} \Rightarrow F pc1 = \text{"critical"})$

$G (pc2 = \text{"request"} \Rightarrow F pc2 = \text{"critical"})$

1.  $pc1 = \text{crit}$
2.  $pc2 = \text{crit}$
3.  $pc1 = \text{req}$
4.  $pc2 = \text{req}$

# Beispiele

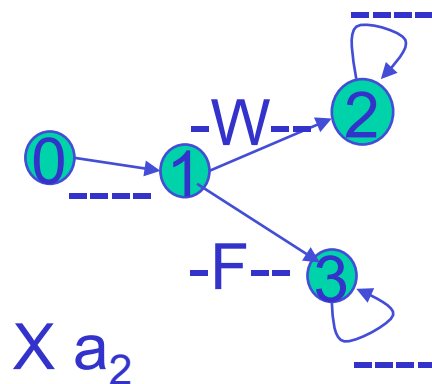


1. pc1 = crit
2. pc2 = crit
3. pc1 = req
4. pc2 = req

# Büchi-Automaten und LTL

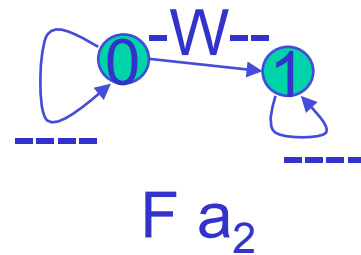
Satz: Zu jeder LTL-Formel  $\phi$  gibt es einen Büchi-Automaten mit  $\max 2^{|\phi|}$  Zuständen, der genau die unendlichen Sequenzen akzeptiert, die  $\phi$  erfüllen.

Beispiele (Eigenschaften  $a_1a_2a_3a_4$ ):



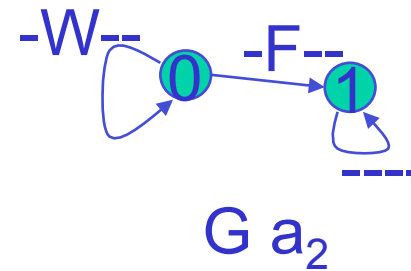
$X a_2$

$Z_0 = \{0\},$   
 $F = \{ \{2\} \}$



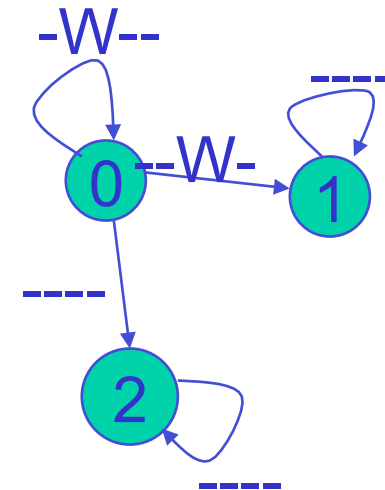
$F a_2$

$Z_0 = \{0\},$   
 $F = \{ \{1\} \}$



$G a_2$

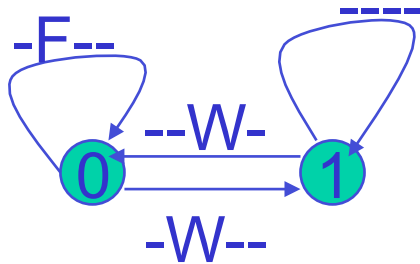
$Z_0 = \{0\},$   
 $F = \{ \{0\} \}$



$a_2 U a_3$

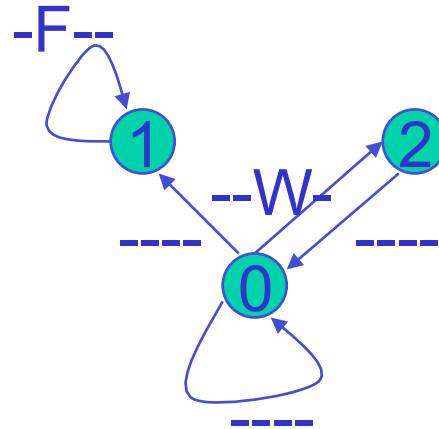
$Z_0 = \{0\},$   
 $F = \{ \{1\} \}$

# Automaten für komplexe Formeln (Bsp)



$G(a_2 \Rightarrow F a_3)$

$Z_0 = \{0\}$   
 $F = \{\{0\}\}$



$(G F a_2) \Rightarrow (G F a_3)$

$Z_0 = \{0\}$   
 $F = \{\{1,2\}\}$

# Zwischenfazit

Haben:

-Büchi-Automat für  $L_{TS}$

-Büchi-Automat für  $L_{\neg\phi}$

Ziel:  $L_{TS} \cap L_{\neg\phi} = \emptyset$  ?

nächster Schritt: geg:  $B_1, B_2$

ges:  $B^*: L_{B^*} = L_{B_1} \cap L_{B_2}$

# Produktautomat

Idee: beide Automaten arbeiten parallel, Akzeptierungsmengen werden nebeneinandergestellt

$$B^1 = [X, Z^1, Z_0^1, \delta^1, F^1]$$

$$B^2 = [X, Z^2, Z_0^2, \delta^2, F^2]$$

$$B^* = [X, Z^*, Z_0^*, \delta^*, F^*]$$

$$Z^* = Z^1 \times Z^2$$

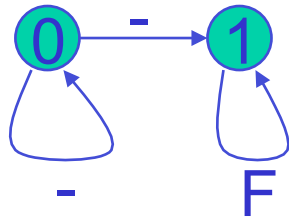
$$Z_0^* = Z_0^1 \times Z_0^2$$

$$\delta^*([z, z'], x) = \delta^1(z, x) \times \delta^2(z', x)$$

$$F^* = \{ F \times Z^2 \mid F \in F^1 \} \cup \{ Z^1 \times F \mid F \in F^2 \}$$

$$\text{Satz: } L_{B^*} = L_{B^1} \cap L_{B^2}$$

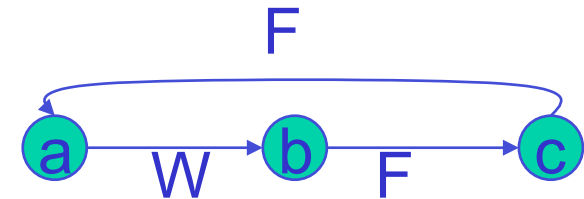
B1:



$Z0 = \{0\}$   
 $F = \{\{1\}\}$

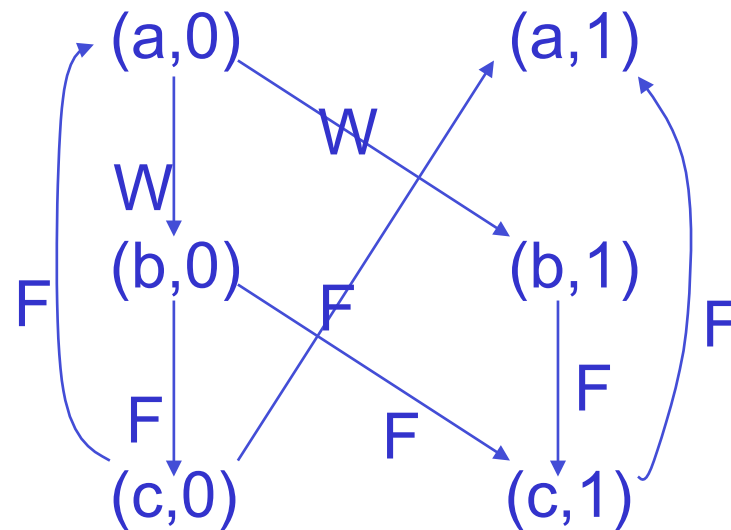
## Beispiel

B2:



$Z0 = \{a\}$   
 $F = \{\{a,b,c\}\}$

B\* :



$Z0 = \{(a,0)\}$   $F = \{Z, \{(a,1), (b,1), (c,1)\}\}$



# Emptiness

geg.: Büchi-Automat  $B$

Frage  $L_B = \emptyset$  ?

Lösung:  $L_B \neq \emptyset$  gdw. es gibt eine nichttriv. SZK in  $B$  (von einem Initialzustand erreichbar), die aus jeder Akzeptierungsmenge Elemente enthält.

“ $\Rightarrow$ ”: Sei  $\pi$  Sequenz, die von  $B$  akzeptiert wird.


$Z^*$  sei Menge der Zustände, die im akzeptierenden Lauf unendlich oft durchlaufen werden.  $Z^*$  ist stark zusammenhängend, und enthält Elemente aus jeder Akzeptierungsmenge

“ $\Leftarrow$ ”: Konstruiere zur SZK eine Sequenz, die vom Initialzustand zur SZK gelangt, und dort zyklisch alle Zst. durchläuft. Diese Sequenz ist offenbar in  $L_B$

nichttriv. = (mehrere Zst.) oder (1 Zustand mit Schleife)

# LTL Model Checking

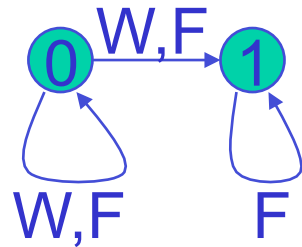
Geg.: TS (implizit),  $\phi$ .

1. Konstruiere  $B_{\neg\phi}$
  2. Konstruiere Produktautomat  $B^*$  aus TS und  $B_{\neg\phi}$
  3. Suche in  $B^*$  nach SZK, die aus jeder Akzeptierungsmenge ein Element enthalten
  4. gefunden  $\rightarrow$  “nein”, bilde Gegenbeispiel aus gefundener SZK
  5. nicht gefunden  $\rightarrow$  “ja”
- 
- 1 Schritt!

$$O(2^{|\phi|} |TS|)$$

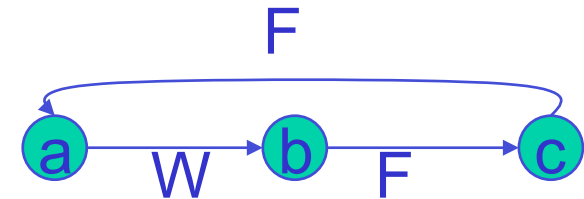
$G F a$  (negiert:  $F G \neg a$ )

## Beispiel



$Z0 = \{0\}$   
 $F = \{ \{1\} \}$

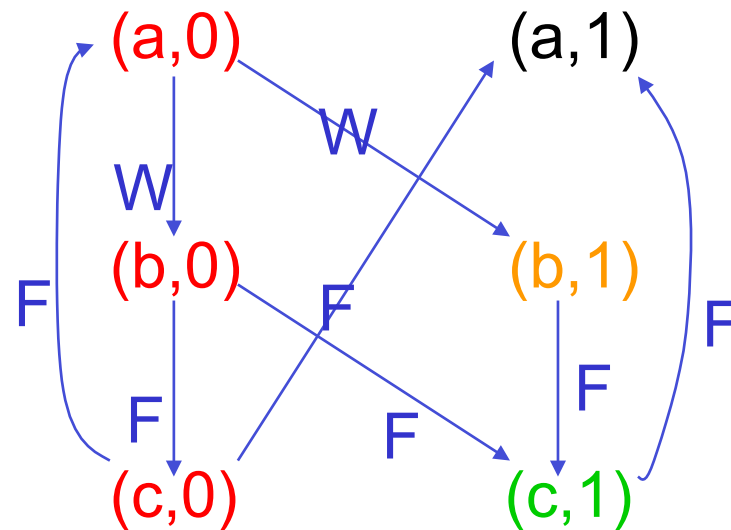
TS:



$Z0 = \{a\}$

$F = \{ \{a,b,c\} \}$

$B^*$  :



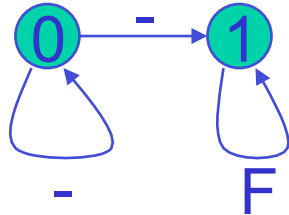
$Z0 = \{(a,0)\}$   $F = \{ Z, \{(a,1), (b,1), (c,1)\} \}$

nur triv. SZK akzeptieren

also: Formel wahr

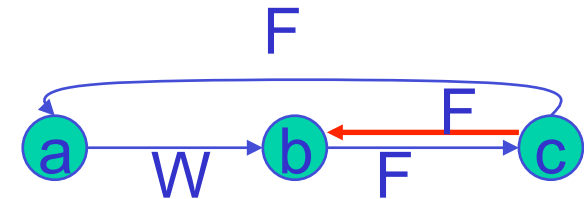
$G F a$  (negiert:  $F G \neg a$ )

## Beispiel



$Z0 = \{0\}$   
 $F = \{ \{1\} \}$

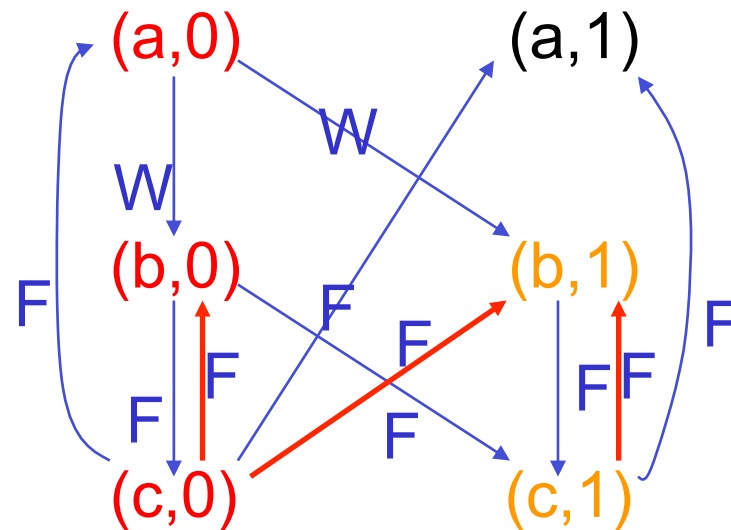
TS:



$Z0 = \{a\}$

$F = \{ \{a, b, c\} \}$

$B^*$  :



$Z0 = \{(a,0)\}$   $F = \{ Z, \{(a,1), (b,1), (c,1)\} \}$

nichttriv. SZK,

→ Formel falsch,

→ Gbsp:  $a (b c)^*$

## 5.2 Automaten als Datenstruktur

- Kontext: Erneut Verifikation: Wollen alle erreichbaren Konfiguration eines Kellerautomaten berechnen.
- Das können unendlich viele sein.
- Hintergrund: Software mit rekursiven Prozeduren.  
Zustand des Kellerautomaten = globale Variablen  
Im Keller: Belegungen lokaler Variablen

# Berechnung erreichbarer Konfigurationen

Ausgangspunkt: Automat, der Anfangskonfiguration akzeptiert

Annahme: max. 2 Elemente pushed in einem Schritt  
(keine prinzipielle Einschränkung, wie wir wissen; Chomsky-NF)

Lösung: Saturation: Für Regel  $[Xy, X'w]$

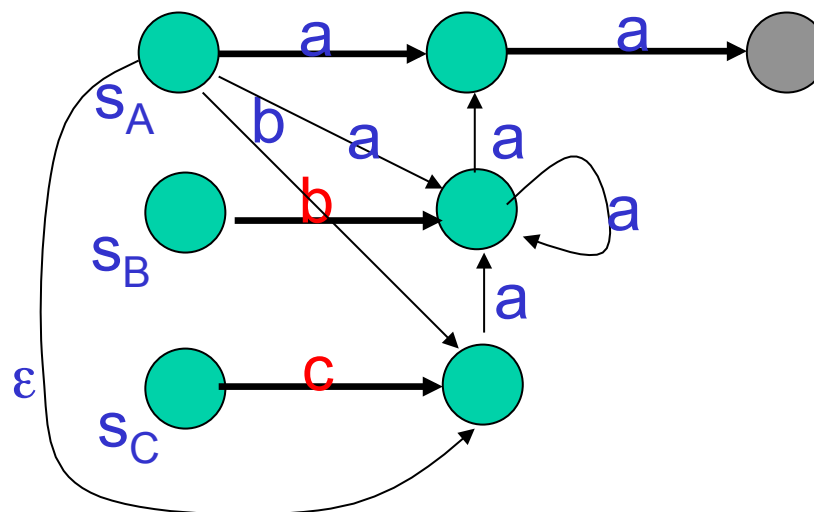
Zustand, der mit  $Xy$  erreicht wird, soll auch von  $X'w$  erreicht werden.

Notwendig: Zwischenzustände für Doppelpush-Regeln

Beispiel:

- 1  $A, a \triangleright B, \textcolor{red}{ba}$
- 2  $B, b \triangleright C, \textcolor{red}{ca}$
- 3  $C, c \triangleright A, b$
- 4  $A, b \triangleright A, \varepsilon$

$C = \{Aaa\}$



# Ergebnis, ganz nebenbei

Satz: Die Menge der erreichbaren Konfigurationen  
eines Kellerautomaten bildet eine reguläre Sprache

# Zusammenfassung Formale Sprachen

1. Sprache = Problem
  - Einordnung in Hierarchie → Gefühl für Komplexität
  - Zusammenhang Grammatikklasse – Maschinenklasse
  - Operationen (Abschlusseigenschaften)
2. Sprache = Repräsentation der Abläufe eines Systems
  - Formaler Zugang, automatische Unterstützung
3. Sprache, Automat = Formales Instrument, Datenstruktur



# Zusammenfassung Gesamt

Wichtigste Kompetenz: Klassifikation  
(Zeithierarchie, Raumhierarchie, Chomsky-Hierarchie)

Verständnis für Tricks, Workarounds, Unzulänglichkeiten

Gefühl für Schwere

Respekt, aber keine Angst vor schweren Problemen