

# Mathe - Grundlagen

Definition → Einem Symbol oder Objekt wird eine Bedeutung zugewiesen.

Satz → Aussage von zuvor definierte Objekte.

Beweis → Bestätigt Allgemeingültigkeit der Aussage auf der vorgegebenen Definition.

↪ Kann als Satz für weitere Beweise oder Definitionen verwendet werden

## Menge

• Mengen sind eindeutig.  $\{1; 2; 2; 3\} = \{1; 2; 3\}$   
↳ falls nicht ist die Rede von einer Multimeng/Bag

• Mengen sind unsortiert  $\{1; 3; 2\} = \{1; 2; 3\}$

Teilmenge:

$$A \subseteq B := \{x \mid x \in A \Rightarrow x \in B\}$$

$$\{1; 2\} \subseteq \{1; 2; 2\} =: M_1 \quad \{\} \subseteq \{1; 2; 3\} =: M_2 \quad \{\{\}\} \not\subseteq \{1; 2; 3\} =: M_3$$

weil:

$$\begin{aligned} 1 &\in M_1 \\ 2 &\in M_1 \end{aligned}$$

weil kein Element  
auch in  $M_2$  ist

weil die Leere Menge nicht als  
Element in  $M_3$  ist.

$$\{\} \notin M_3$$

Die hauptsächlichen Elemente

Mächtigkeit / Kardinalität

• Anzahl der Elemente einer Menge

Potenzmenge  $\mathcal{P}(M)$ :

• Eine Menge aller möglichen Teilmengen von  $M$

$$\cdot |\mathcal{P}(M)| = 2^{|M|}$$

Kartesisches Produkt

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

# Relation

↳ Beziehungen zwischen Objekten

Binäre Relation:  $R \subseteq A \times B \Rightarrow ((a, b) \in R \Leftrightarrow a R b)$

Gleichheits Relation:  $= \subseteq A \times B \Rightarrow ((a, b) \in = \Leftrightarrow a = b)$

Beispiel:  $(42, 42) \in = \Leftrightarrow 42 = 42$

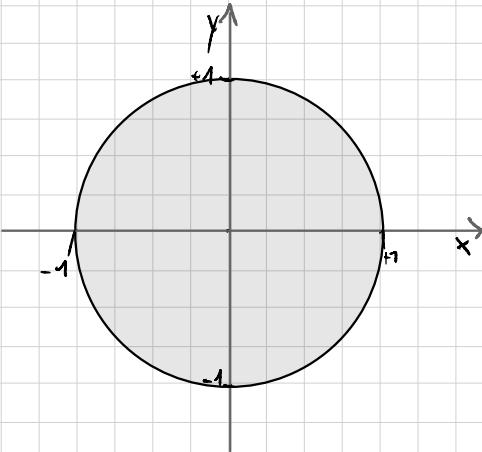
Kleiner Relation:  $< \subseteq A \times B \Rightarrow ((a, b) \in < \Leftrightarrow a < b)$

Beispiel:  $(2, 4) \in < \Leftrightarrow 2 < 4$

↳  $< := \{(a, b) \mid a, b \in \mathbb{N} \wedge (\exists c \in \mathbb{N}: c \neq 0 \Rightarrow a + c = b)\}$

Beispiel Kreisrelation:

$$R := \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 \leq 1\}$$



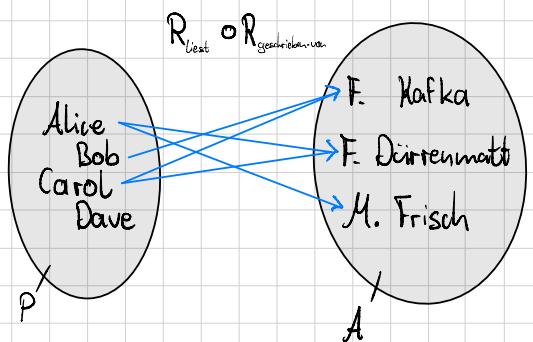
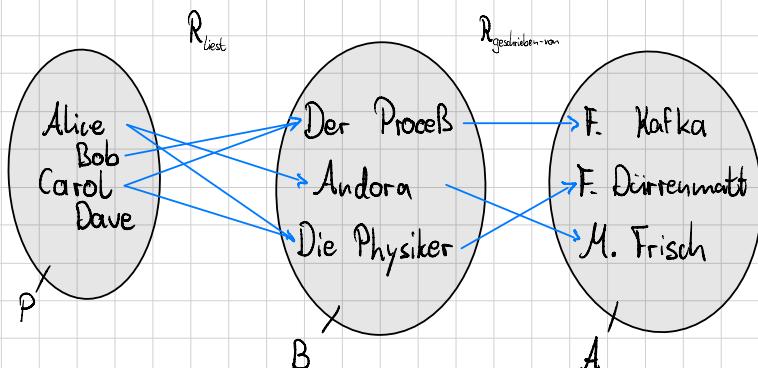
## Komposition

Sei:  $R \subseteq A \times B$ ,  $S \subseteq B \times C$

dann  $R \circ S \subseteq A \times C$

$$R \circ S := \{(a, c) \in A \times C \mid \exists b \in B: (a, b) \in R \wedge (b, c) \in S\}$$

Beispiel:



# Eigenschaften von binären Relationen

- „reflexiv“  $\Rightarrow \forall a \in A: (a, a) \in R$
- „irreflexiv“  $\Rightarrow \forall a \in A: (a, a) \notin R$
- „symmetrisch“  $\Rightarrow \forall a, b \in A: ((a, b) \in R \Rightarrow (b, a) \in R)$
- „asymmetrisch“  $\Rightarrow \forall a, b \in A: ((a, b) \in R \Rightarrow (b, a) \notin R) \quad [a \neq b]$
- „antisymmetrisch“  $\Rightarrow \forall a, b \in A: ((a, b) \in R \wedge (b, a) \in R \Rightarrow a = b)$
- „transitiv“  $\Rightarrow \forall a, b, c \in A: ((a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R)$

	=	< ( $>$ )	$\leq$ ( $\geq$ )
reflexiv	✓ ↗ $(a, a) \in =$	✗ $(a, a) \notin <$	✓ ↗ $(a, a) \in <$
irreflexiv	✗	✓ ↘ $(a, a) \in <$	✗
symmetrisch	✓ ↗ $(a, b) = (b, a)$	✗ $(a, b) \neq (b, a)$	✗ $(a, b) \neq (b, a)$
asymmetrisch	✗	✓ ↘ $(a, b) \neq (b, a)$	✗ $(a, b) = (b, a)$
antisymmetrisch	✓ ↗ $a = b$	✓ $(a, b) \in R \wedge (b, a) \notin R \Rightarrow a \neq b$	✓ ↗ $a \leq b \wedge b \leq a \Rightarrow a = b$
transitiv	✓ ↗ $a = c$	✓ $a < b \wedge b < c \Rightarrow a < c$	✓ ↗ $a \leq b \wedge b \leq c \Rightarrow a \leq c$

→ Halbordnung

↳ Totalordnung, wenn noch  $\forall x, y \in M: x \leq y \vee y \leq x$

Beispiel Kreisrelation:

$$K := \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 \leq 1\}$$

nicht reflexiv: Nicht für alle  $(a, a) \in K$  Bsp.:  $(1, 1) \notin K$

nicht irreflexiv: Nicht für alle  $(a, a) \notin K$  Bsp.:  $(0, 0) \in K$

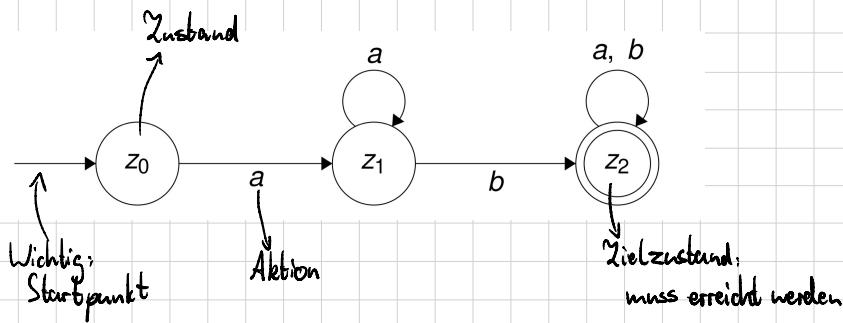
symmetrisch:  $(x, y) \in K \Rightarrow (y, x) \in K$

nicht asymmetrisch:  $\nearrow$

nicht antisymmetrisch:  $(1, 0) \in K \wedge (0, 1) \in K \Rightarrow 1 \neq 0$

nicht transitiv: Nicht für alle  $(a, b) \in K \wedge (b, c) \in K \Rightarrow (a, c) \in K$   
Bsp.:  $(1, 0) \in K \wedge (0, 1) \in K \nRightarrow (1, 1) \in K$

# Endliche Automaten



Anwendungsbeispiele:

- Dig. Schaltungen
- Programmierung
- Wort- u. Spracherkennung (Abstrakt)

↳ Dieser Automat akzeptiert folgende Wörter:

- ab, aab, abab, aaabbba, ...

(die mit einem a starten u. mind. ein b besitzen)

**Definition** (deterministischer endlicher Automat (DFA))

⇒ Besteht aus 5 Komponenten:

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

$\mathcal{Z}$ : „Endliche Menge an Zuständen“

$\Sigma$ : „Endlichen Eingabealphabet“

$\delta$ :  $\mathcal{Z} \times \Sigma \rightarrow \mathcal{Z}$  „Überführungsfunktionen (Zustandsübergänge)“

$z_0$ : „Startzugang“  $z_0 \in \mathcal{Z}$

$E$ : „Endzustände“  $E \subseteq \mathcal{Z}$

⇒ Eigenschaften:

• Arbeiten Taktweise

- Eingabe (hier Wort w) wird pro Schritt (Buchstabe) von links nach rechts gelesen
- Nach einem Takt befindet sich der Automat in einem Zustand
- Wenn die Eingabe fertig gelesen wurde und der Automat sich in einem Endzustand befindet, wird das Wort akzeptiert

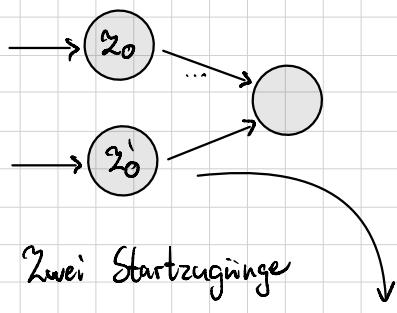
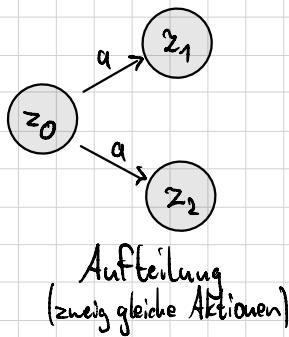
Wortproblem

Eingabe:  $w \in \Sigma^*$       Ausgabe { ja  $w \in L$   
   nein  $w \notin L$

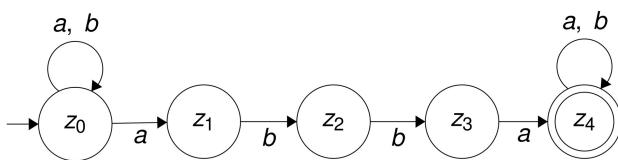
$L \subseteq \Sigma^*$   
Lukas Mensch

⇒ für DFA ist dies in linearer Zeit lösbar

# Nichtdeterministische Endliche Automaten (NFA)



Beispiel:

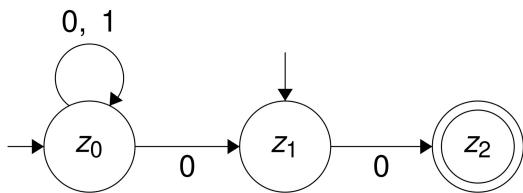


↪ Erkennt Wörter mit dem Substring *abba*

Demnach ändert sich die Definition,  
da es nun eine Menge von  
Startzügen gibt.

$$M = (Z, \Sigma, \delta, S, E)$$

↓ anstatt  $z_0$



↪ Wort endet auf 00 oder w=0

Konstruktion eines DFA aus einem NFA

$$\text{NFA: } M = (Z, \Sigma, \delta, S, E)$$

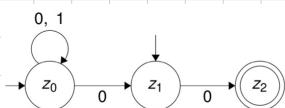
$$Z' = P(Z)$$

$$\Rightarrow \text{DFA: } M' = (Z', \Sigma, \delta', z'_0, E')$$

$\Sigma$ : „Das selbe Alphabet“

$\delta'$ ,  $z'_0$ ,  $E'$ : „wird durch Konstruktion des DFA ermittelt“

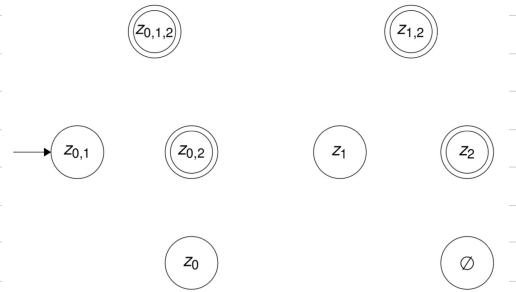
Beispiel



1. Erstellen der Zustandsmenge  $Z'$ :

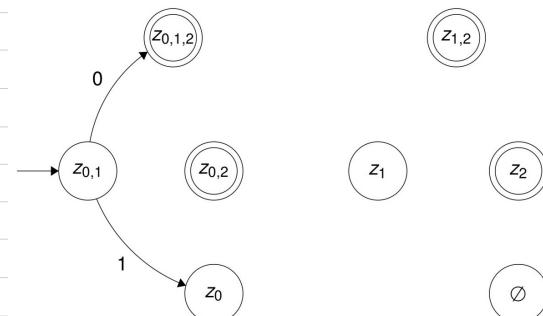


2. Start- und Endzustände markieren

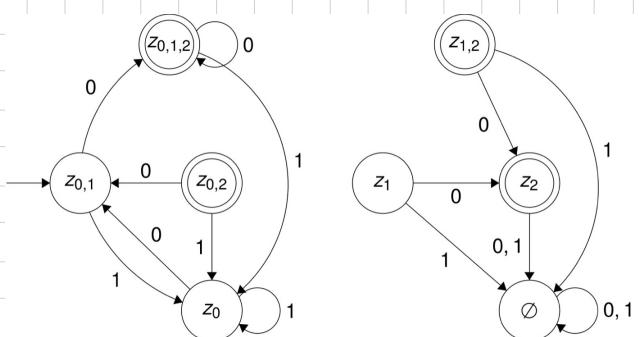


3.

a) Übergänge von  $z_{0,1} \dots$



b) ... und alle anderen Übergänge



## Grammatiken

Definition:

$V$ : endliche Menge an Variablen

$G = (V, \Sigma, P, S)$

$\Sigma$ : endliche Menge an Terminalalphabet

$P$ : endliche Menge an Ableitungsregeln / Produktionen  
 $\cdot y \rightarrow y' \in P$

$S$ :  $S \in V$  ist die Startvariable

## Chomsky-Hierarchie (Grammatiken können in 4 Typen unterteilt werden)

Typ-0: Unbeschränkte Grammatik  
unterliegt keiner Einschränkung

Typ-1: Kontextsensitive Grammatik

Wenn von  $u$  auf  $v$  abgeleitet wird, ist die Wortlänge von  $u$  kleiner gleich als die von  $v$

Für  $u \rightarrow v$  gilt  $|u| \leq |v|$  (Ausnahme  $u \rightarrow \epsilon$ )

Typ-2: Kontextfreie Grammatik

Es wird nur von einer Variablen abgeleitet

$$u \in V$$

Typ-3: Reguläre Grammatik

$u$  wird auf ein Terminalsymbol oder einem Terminalsymbol mit Variable abgeleitet

$$u \rightarrow \Sigma$$

$$u \rightarrow \Sigma^V$$

## Wortproblem

Bezeichnet das Entscheidungsproblem ob ein gegebenes Wort in einer Grammatik enthalten ist oder nicht.

- Für Typ-0-Sprachen im Allgemeinen nicht entscheidbar
- Für Typ-1-Sprachen und höher ist durch einen Algorithmus entscheidbar

## Grammatiken und endliche Automaten

Jede durch einen endlichen Automaten erkannte Sprache ist regulär

⇒ Somit können aus Typ 3 Grammatiken NFA erstellt werden

$$\text{Es gilt } L(G) = L(M)$$

$M(\Sigma, \delta, S^1, E)$  $G(V, \Sigma, P, S)$ 

$$\Sigma = V \cup \{X\}$$

 $X \in V$ 

$$\delta(A, a) = B$$

falls  $A \rightarrow aB \in P$ 

$$\delta(A, a) = X$$

falls  $A \rightarrow a \in P$ 

$$S^1 = \{S\}$$

$$E = \{X\}$$

$$\text{oder } E = \{X, S\}$$

wenn  $S \rightarrow \epsilon \in P$ 

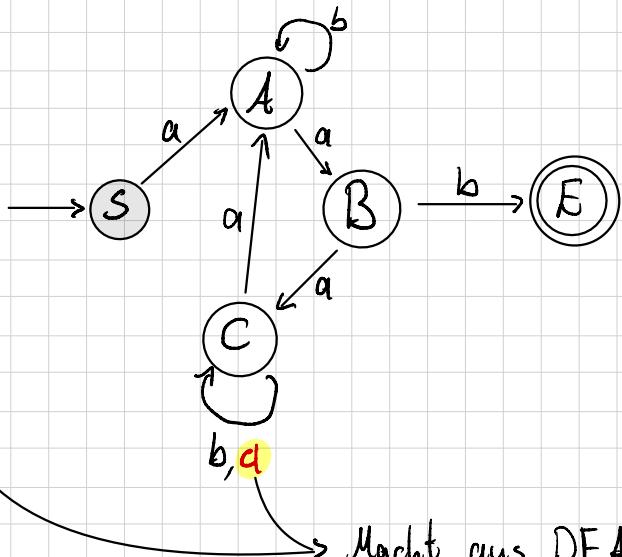
Beispiel:

$$S \rightarrow aA$$

$$A \rightarrow bA \mid aB$$

$$B \rightarrow b \mid ac$$

$$C \rightarrow aA \mid bC \mid aC$$



Macht aus DFA einen NFA

# Kontextfreie Grammatik

## Backus-Naur-Form

- **Variablen** werden in spitze Klammern angegeben: <...>
- **Terminale** werden in Anführungszeichen angegeben: '...'

<Programm> ::= 'PROGRAM' <Bezeichner> 'BEGIN' <Satzfolge> 'END'

<Bezeichner> ::= <Buchstabe> | <Bezeichner> <Buchstabe>

...

⇒ Java in BNF

<https://cs.au.dk/~amoeller/BegAut/JavaBNF.html>

## Erweiterte BNF:

- Null oder Mehrfachwiederholung mit geschweiften Klammern

<Wort> ::= {<Buchstabe>} <Buchstabe>

↪ in RegEx: [a-zA-Z]\*

- Optional mit eckigen Klammern

<Adjektiv> ::= ['un'] <Adjektiv>

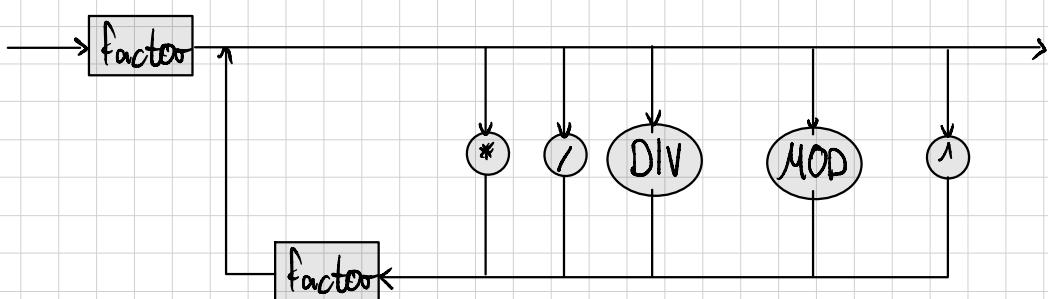
↪ in RegEx: (un)? (logisch)

- 'AAA' ::= "3\*'A'"

↪ A{3}

↪ [a-z]

## Syntaxdiagramme (Möglich bei jeder EBNF):



<factor> ::= '\*'<factor> | '/'<factor> | 'DIV'<factor> | 'MOD'<factor> | '^'<factor>

⇒ Programme, die prüfen, ob ein Wort (Terme, andere Programme, etc.) Lukas Mensch  
in einer Sprache enthalten sind, nennen sich **Parser**.

# Chomsky Normalform (CNF)

Regeln einer Grammatik entsprechen ausschließlich folgender Form:

$$A \rightarrow AB$$

$$A \rightarrow a$$

$$S \rightarrow \epsilon$$

Algorithmus:

- Terminalsymbole in Variablen umschreiben

$$A \rightarrow X_1 \dots a \dots X_n$$

zu:

$$N_a \rightarrow a$$

$$A \rightarrow X_1 \dots N_a \dots X_n$$

- Regeln mit mehr als 2 Variablen auf der rechten Seite auf 2 Variablen herunterbrechen:

$$A \rightarrow X_1 X_2 \dots X_n$$

zu

$$A \rightarrow X_1 A_1; A_1 \rightarrow X_2 A_2;$$

$$\dots; A_{n-2} \rightarrow X_{n-1} X_n$$

- Vereinfachen

$$A \rightarrow B, B \rightarrow X_1 \dots X_n$$

zu

$$A \rightarrow X_1 \dots X_n$$

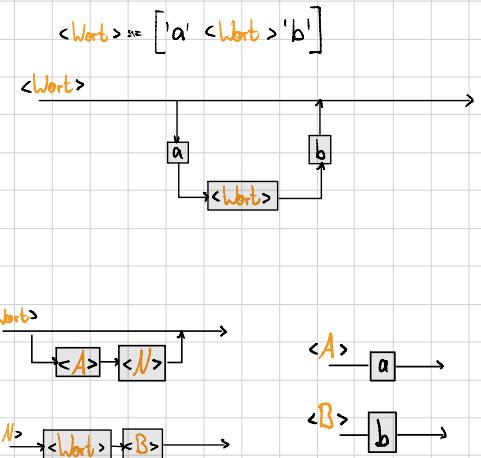
Als Syntaxdiagramm:

In CNF:

$$<\text{Wort}> ::= [<A> <N>];$$

$$<N> ::= <\text{Wort}> <B>;$$

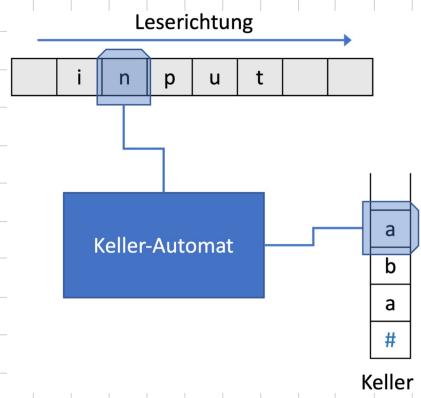
$$<A> ::= a; <B> ::= b;$$



# Kellerautomaten (PDA (engl. Push-Down-Automat))

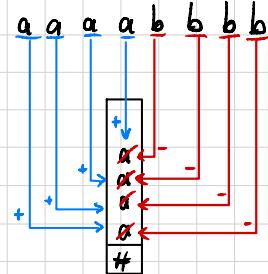
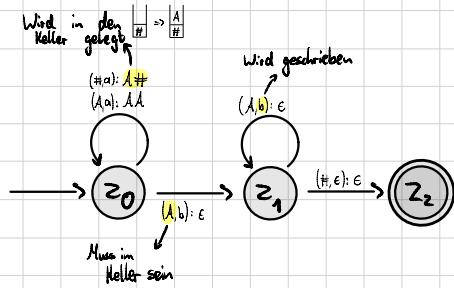
→ Automat hat nun Gedächtnis (den Keller)

- Keller hat ein Last-in First-out (LIFO) Speicher.



Beispiel für Sprache  $L = \{ n \in \mathbb{N} \mid a^n b^n \}$

Beispiel für  $n=4$ :



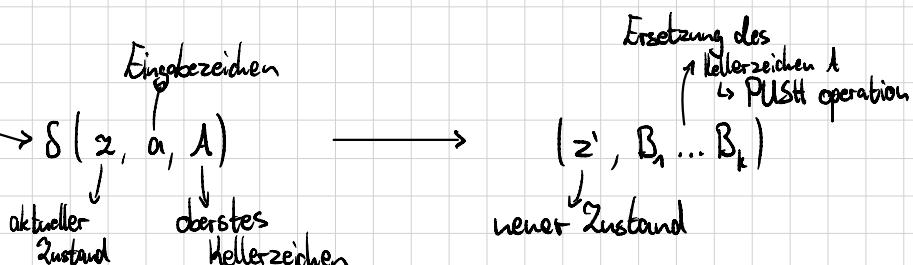
→ Endzustand ist nicht nötig, da die Bedingung gilt,  
dass der Keller am Ende leer sein muss

Definition:

$$M = (\Sigma, \Gamma, \Gamma^*, \delta, z_0, \#)$$

$\Gamma$ : Keller alphabet

$\delta := \Sigma \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times (\Sigma \times \Gamma^*)$ : Übergangsrelationen  
 $\# \in \Gamma$ : unterste Kellerzeichen



Bei POP operation:

$$\delta(z^*, a^*, \lambda^*) \rightarrow (z^*, \epsilon)$$

⇒ jeder Kellerautomat kann in ein Kleberautomat mit nur einem Zustand überführt werden  
(durch Kettentheorie?)

## CYK (Cocke, Younger, Kasami) - Algorithmus

⇒ zum prüfen, ob ein Wort in einer Typ-2 Sprache enthalten ist.

gegeben ist folgende Grammatik mit einem Wort, welches überprüft werden soll:

$$S \rightarrow BC | AC | BA$$

zu testen: abcbac

$$A \rightarrow AA | BB | a$$

$$B \rightarrow BA | b$$

$$C \rightarrow AC | c$$

→ Grammatik muss in CNF sein

1. Tabelle aufstellen und Terminalssymbole in Variablen umwandeln:

a	b	a	b	a	c
A	B	A	B	A	C
∅	SB	∅	SB	SC	

2. Prüfen auf Ableitungsregel von Nachbar ⇒ Ergebnis in die nächste Zeile

a	b	a	b	a	c
A	B	A	B	A	C
∅	SB	∅	SB	SC	

$$\begin{aligned} S &\rightarrow BC | AC | BA \\ A &\rightarrow AA | BB | a \\ B &\rightarrow BA | b \\ C &\rightarrow AC | c \end{aligned}$$

3. Prüfen auf Ableitungsregeln von Nachbarn im Dreieck

a	b	a	b	a	c
A	B	A	B	A	C
∅	SB	∅	SB	SC	
∅					

⇒

a	b	a	b	a	c
A	B	A	B	A	C
∅	SB	∅	SB	SC	
∅	A				

$$\begin{aligned} S &\rightarrow BC | AC | BA \Rightarrow \\ A &\rightarrow AA | BB | a \\ B &\rightarrow BA | b \\ C &\rightarrow AC | c \end{aligned}$$

a	b	a	b	a	c
A	B	A	B	A	C
∅	SB	∅	SB	SC	
∅	A	∅	S		

#### 4. Dreieck Vergrößern für nächste Zeile

a	b	a	b	a	c
A	B	A	B	A	C
Ø	SB	Ø	SB	SC	
Ø	A	Ø	S		
A					

$$\begin{aligned} S &\rightarrow BC \mid AC \mid BA \\ A &\rightarrow AA \mid BB \mid a \\ B &\rightarrow BA \mid b \\ C &\rightarrow AC \mid c \end{aligned}$$

a	b	a	b	a	c
A	B	A	B	A	C
Ø	SB	Ø	SB	SC	
Ø	A	Ø	S		
A					

$$\begin{aligned} S &\rightarrow BC \mid AC \mid BA \\ A &\rightarrow AA \mid BB \mid a \\ B &\rightarrow BA \mid b \\ C &\rightarrow AC \mid c \end{aligned}$$

Hier wird auf SS, SB, BS u. BB geprüft

a	b	a	b	a	c
A	B	A	B	A	C
Ø	SB	Ø	SB	SC	
Ø	A	Ø	S		
A	A	Ø			
A					

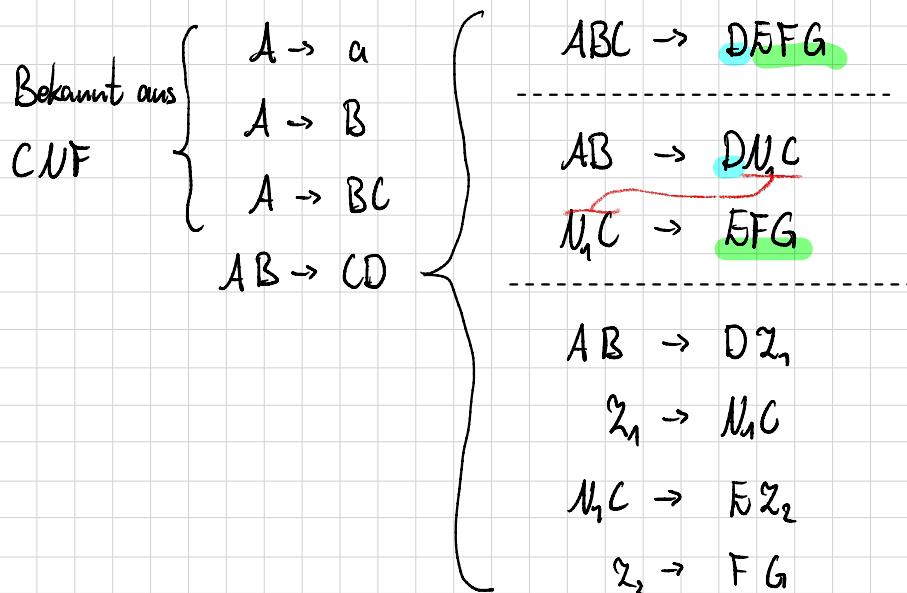
=>

a	b	a	b	a	c
A	B	A	B	A	C
Ø	SB	Ø	SB	SC	
Ø	A	Ø	S		
A	A	Ø			
A	SC				

5. Wenn die Startvariable  
in der letzten Zelle  
ist, ist das Wort  
ein Teil der kontextfreien Sprache

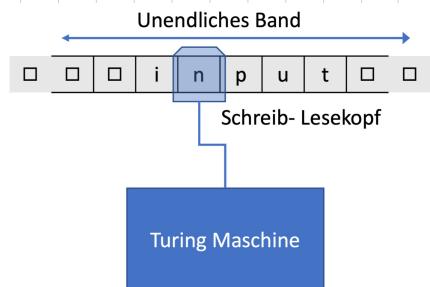
# Kontextsensitive und unbeschränkte Sprachen

## KNF (Kuerzel Normalform)



## Turingmaschine

⇒ Erweiterung durch holen und Schreiben auf unendlichen Band, anstatt nur einer Eingabe



Definition:

$$M = (\Sigma, \Sigma, \Gamma, \delta, z_0, \square, F)$$

$\Gamma$ : Arbeitsalphabet      Bewegungsrichtung

$$\delta: \Sigma \times \Gamma \longrightarrow \Sigma \times \Gamma \times \{L, R, N\} \quad \hat{=} \delta(z, a) = (z', b, X)$$

$\square$ : Blank       $\square \notin \Gamma \setminus \Sigma$

$F$ : Endzustände       $F \subseteq \Sigma$

## Mehrband-Turing-Maschine

... □□□1010□□□□ ...

... □□□aabbab□□□□ ...

↓

... □□□1<sub>a</sub>0<sub>a</sub>1<sub>b</sub>0<sub>a</sub>□□□□ ...

A B C D

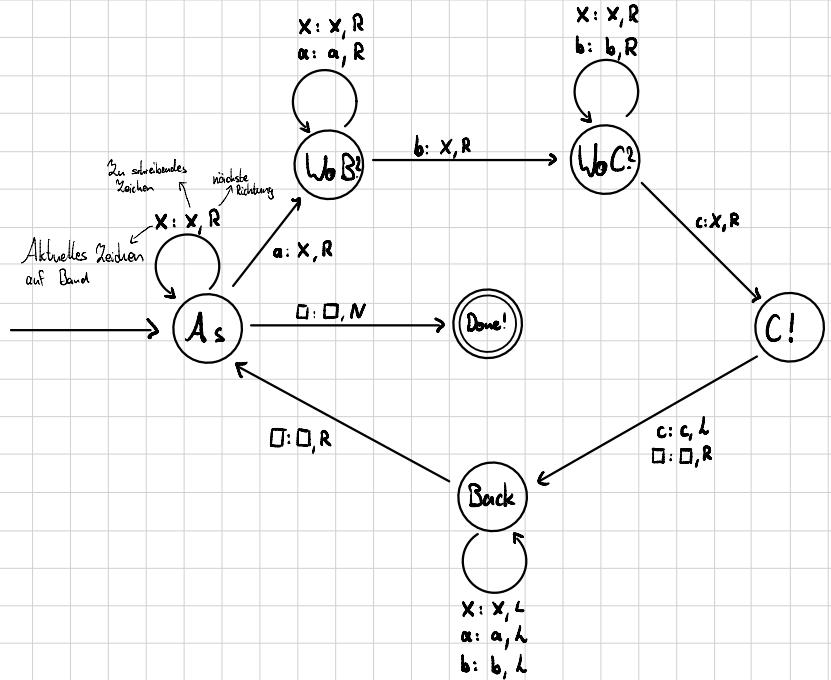
$$\begin{aligned} 1_a &\Rightarrow A \\ 0_a &\Rightarrow B \\ 1_b &\Rightarrow C \\ 0_a &\Rightarrow D \end{aligned}$$

Beispiel:

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

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

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



## Berechenbarkeit

⇒ mit TM lassen sich auch Berechnungen durchführen

↳ benötigtes Eingabesymbol  $\Sigma = \{0, 1, \dots, 9, +, -, \dots\}$

⇒ Grammatiken zum Erstellen bestimmter Zahlen sind ebenso möglich:

Gerade-Positive Zahl:

$$\begin{aligned} S &\rightarrow 2S \mid G \\ 2 &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ G &\rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8 \end{aligned} \quad \left. \begin{array}{l} \text{Typ-3 Grammatik:} \\ \Rightarrow \text{RegEx: } [0-8]^* [02468] \end{array} \right.$$

Typ-1 und Typ-0 Sprachen

- XML stellt eine Typ-1 Grammatik dar
- Typ-0 → rekursiv Auftzähllbar
  - alles was sich berechnen lässt, lässt sich mit einer Typ-0 Grammatik berechnen

Turing-Vollständig / (mächtig)

↳ Ein System, dass alles berechnen kann was eine TM berechnen kann

$\Rightarrow$  Berechenbar ist jede Funktion, die mit einem angegebenen Algorithmus bei jedem Wert aus dem Definitionsbereich stoppt.

$\Rightarrow$  Turing-Vollständigkeit ist ausreichend für eine universelle Programmierbarkeit

$\hookrightarrow$  Berechenbarkeit ist identisch zu allen anderen definierten Systemen

$\hookrightarrow$  Church These

## Rekursiv vs. Rekursiv Auflösbar

charakteristische Funktion einer TM

$$\chi_M(w) = \begin{cases} 1 & \text{wenn TM mit } w \in L \text{ stoppt} \\ 0 & \text{wenn TM mit } w \notin L \text{ stoppt} \\ \perp & \text{wenn TM nicht stoppt} \end{cases}$$

TM entscheidbar:

$$\chi_M(w) = \begin{cases} 1 & \text{wenn } w \in L \\ 0 & \text{wenn } w \notin L \end{cases}$$

TM semi-entscheidbar:

$$\chi_M(w) = \begin{cases} 1 & \text{wenn } w \in L \\ 0 & \text{wenn } w \notin L \end{cases}$$

## WHILE-Programm

Ein WHILE-Programm besteht aus den Symbolen WHILE, LOOP, DO, END, :=, +, -, ;, : und Variablen  $x_1, \dots, x_i, \dots$  und Konstanten

Es sind nur 4 Anweisungen erlaubt:

- Anweisung einer Variablen mit Variablen plus Konstante:

$$x_3 := x_4 + 10$$

- oder minus Konstante

$$x_5 := x_6 - 300$$

Variablen dürfen nur Werte  
 $\geq 0$  zugewiesen werden

- LOOP-Anweisung → Mach Anweisung nach DO so oft wie Wert nach LOOP

LOOP  $x_i$  DO P END

↳ P wird  $x_i$  mal gemacht.

- WHILE-Anweisung → Mach Anweisung nach DO so oft, bis die Bedingung nach WHILE nicht mehr erfüllt ist

WHILE  $x_i \neq 0$  DO P END

↳ P wird solange gemacht bis  $x_i = 0$  ist.

### Anwendungsbeispiele:

- Fallunterscheidung

$$P_{IF_1} := \text{IF } x=0 \text{ THEN P END} := \begin{cases} y := 1; \\ \text{LOOP } x \text{ DO } y := 0 \text{ END}; \\ \text{LOOP } y \text{ DO } P \text{ END}; \end{cases}$$

$$P_{IF_2} := \text{IF } x=c \text{ THEN P END} := \begin{cases} y := 1; \\ \text{LOOP } c \text{ DO } [\text{IF } x \neq 0 \text{ THEN } x := x - 1 \text{ END}] \text{ END}; \\ \text{LOOP } x \text{ DO } y := 0 \text{ END}; \\ \text{LOOP } y \text{ DO } P \text{ END}; \end{cases}$$

$$P_{IF_3} := \text{IF } x=0 \text{ THEN P}_1 \text{ ELSE P}_2 := \begin{cases} y := 1; z := 0 \\ \text{LOOP } x \text{ DO } y = 0; z = 1 \text{ END}; \\ \text{LOOP } y \text{ DO } P_1 \text{ END}; \\ \text{LOOP } z \text{ DO } P_2 \text{ END}; \end{cases}$$

- Addition

$$P_{\text{Addition}} := x_0 := x_1 + x_2 := \begin{cases} x_0 := x_1; \\ \text{LOOP } x_2 \text{ DO } x_0 := x_0 + 1 \text{ END}; \end{cases}$$

## Multiplication

$P_{\text{Multiplikation}} = x_0 := x_1 * x_2 := \begin{cases} x_0 := 0; \\ \text{LOOP } x_2 \text{ DO } x_0 := x_0 + x_1 \text{ END; } \end{cases}$

LOOP mit WHILE ersetzen:

```

    LOOP x DO P END;
    x := y;
    WHILE x ≠ 0 DO
        y := y - 1;
        P;
    END;
  
```

→ Turingmaschine und WHILE-Programme sind gleichmächtig.

(Der Eine kann das machen, was der Andere kann)

→ Jedoch sind LOOP-Programme eine Unterklasse und nicht Turing-vollständig. (Bsp. Ackermannfunktion kann nicht mit LOOP-Programmen berechnet werden)

## Übersicht

$$L(G) = \emptyset ?$$

$$L(G_1) \cap L(G_2) = \emptyset$$

Typ	Sprache	Automat	Regeln (vereinfacht)	Wortproblem	Leerheitsproblem	Äquivalenzproblem	Schnittproblem
Typ-0	Rekursiv aufzählbar	Turingmaschine (TM)	frei	nein	nein	nein	nein
Typ-1	Kontextsensitiv	Linear Beschränkter Automat (LBA)	$w_1 \rightarrow w_2$ $ w_1  \leq  w_2 $	ja $O(e^n)$	nein	nein	nein
Typ-2	Kontextfrei	Kellerautomat (PDA)	$A \rightarrow w_2$	ja $O(n^3)$	ja	nein	nein
Typ-3	Regulär	Endlicher Automat (DFA/NFA)	$A \rightarrow aB$	ja $O(n)$	ja	ja	ja

Wort in der Sprache

$$L(G_1) = L(G_2)$$