Automatentheorie und Formale Sprachen

Sommersemester 2022
(LV 4110)

mittwochs, 11:45 bis 13:15

Prof. Dr. Bernhard Geib

Worum geht es in der Einführung?

- Womit beschäftigt sich die Theoretische Informatik?
  - ✓ Ziele und Merkmale
  - ✓ Themenbereiche und Abgrenzung
- Vorlesungsübersicht
  - ✓ Gliederung und Inhalte
  - ✓ Anwendungsbeispiele
- Organisation der Lehrveranstaltung
  - ✓ Vorlesung, Seminar und Klausur
  - ✓ Ablauf und Vereinbarung zur Leistungsbewertung
  - ✓ Hilfsmittel und Unterrichtsmaterial
  - ✓ Quellen- und Literaturangabe

Die vier wesentlichen Teilgebiete der Informatik:

Praktische Informatik	Angewandte Informatik	Technische Informatik
Theoretische Informatik		

- Die Theoretische Informatik ist das wissenschaftliche Fundament – sozusagen der theoretische Unterbau – der Informatik.
- Die Konzepte der Theoretischen Informatik sind ebenso fundamental wie abstrakt und anspruchsvoll in der Vermittlung.
  - ✓ prinzipielle Lösbarkeit von Problemen
  - ✓ Grenzen der Automatisierung
  - ✓ Modelle, mit deren Hilfe Problemlösungen auf grundsätzliche Machbarkeit hin überprüft und miteinander verglichen werden können

Teilgebiete der Informatik:

Praktische Informatik	Angewandte Informatik	Technische Informatik
Algorithmen, Datenstrukturen, Programmiermethoden Programmiersprachen und Compiler Betriebssysteme und Softwaretechnik	Graphik Datenbanken Künstliche Intelligenz Simulation und Modellierung Textverarbeitung Spezifische Anwendungen	Hardwarekomponenten Schaltnetze, Schaltwerke, Prozessoren Mikroprogrammierung Rechnerorganisation und -architekturen, Rechnernetze

Theoretische Informatik
Automatentheorie und Formale Sprachen Theorie der Berechenbarkeit Komplexitätstheorie und Formale Semantik

Womit beschäftigt sich die Theoretische Informatik?

- **Automatentheorie** (Modellierung der prinzipiellen Funktion einer informationsverarbeitenden Maschine)
- **Algorithmentheorie** (Präzisierung der Begriffe Berechenbarkeit und Algorithmus)
- **Berechenbarkeitstheorie** (Gibt es zu jeder Problemstellung einen Lösungsalgorithmus?)
- **Komplexitätstheorie** (Einschätzung des Aufwandsverhaltens bezüglich Rechenzeit und Speicherplatz)
- **Theorie formaler Sprachen** (Abstraktion von Lexik, Syntax und Semantik)

Ziele und Merkmale der Theoretische Informatik

- Kennenlernen der grundsätzlichen Begriffe, Methoden und Beweistechniken
- Eindringen in die grundlegende Konzepte
- Erkenntnisse im Hinblick auf die praktische Lösbarkeit

und zwar **un**abhängig von konkreten Rechnern und aktuellen Technologien.

Wie beschäftigen uns mit Abstraktionen und Modellbildungen im Zusammenhang mit Problemen, die in irgendeiner Weise mit Hilfe von Computern gelöst werden sollen.

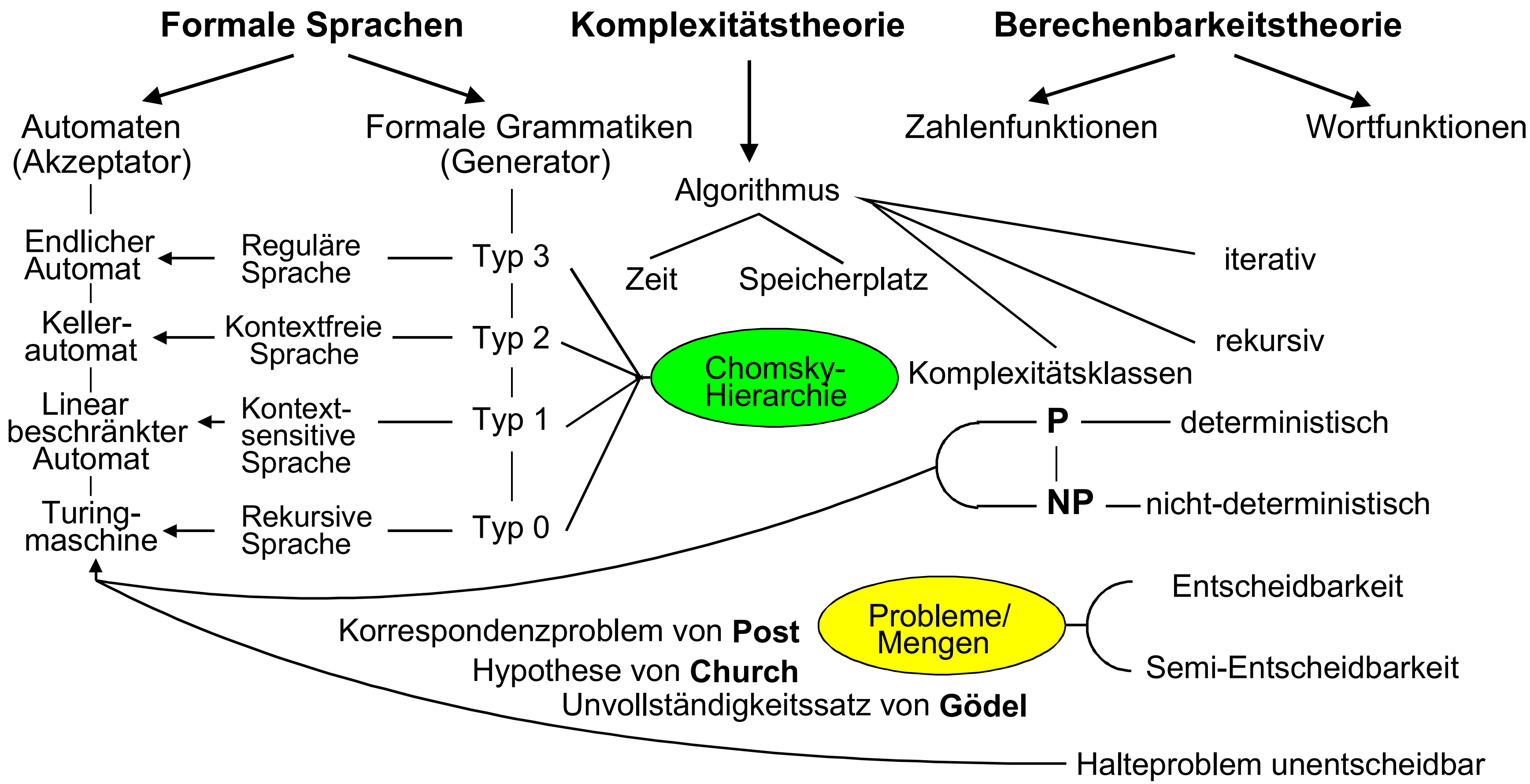
Einführung in die Vorlesung

Grundziel: Unter dem Aspekt der Anwendung werden wichtige Hauptzweige der **Theoretischen Informatik** vorgestellt und anhand ausführlich behandelter Beispiele deren Bezug zu Problemlösungen der Praxis erläutert → **Praktische Informatik**

Gebiete:

- Lexikalische Analyse und Mustererkennung
- Definition von höheren Programmiersprachen (BNF, Syntaxgraphen)
- Compilerbau (Syntaxanalyse und Algorithmierbarkeit)
- Parallele Algorithmen und Berechenbarkeits- bzw. Komplexitätstheorie
- Sicherheitstechnik (Formale Verifikation von Sicherheitseigenschaften)

Zusammenhänge



Gliederung

1. Endliche Automaten (Automatentheorie, Modellierung und Überführungsfunktion, Zustandsgraphen und Funktionstafeln)
2. Reguläre Sprachen und Mengen (Reguläre Ausdrücke, Mengenoperationen und Verknüpfungen, Konstruktion von Automaten, Suchalgorithmen)
3. Grammatiken und Formale Sprachen (Semi-Thue-Systeme, Chomsky-Grammatiken und -Hierarchie, Ableitungsbäume)
4. Kellerautomaten und Kontextfreie Sprachen (Syntaxanalyse von Programmier-sprachen, Pumping-Lemma, Funktionsweise eines Kellerspeichers)
5. Turingmaschinen und Kontextsensitive Sprachen (Monotonie, Funktionsweise der Turingmaschine)
6. Entscheidbarkeit von Problemen und Berechenbarkeit von Algorithmen
7. Problemklassen und Komplexitätstheorie

Aufgabenstellung:

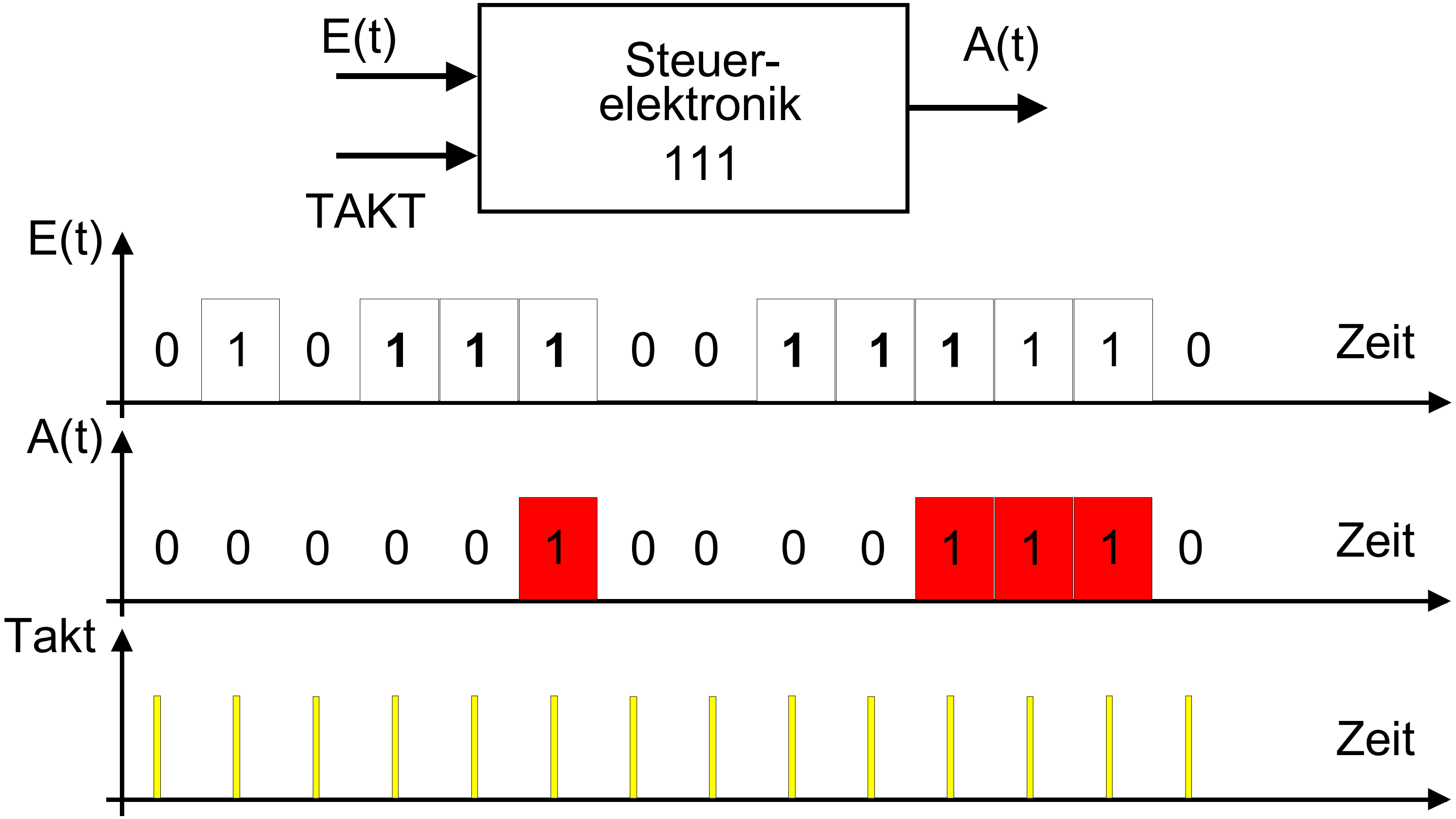
Entwerfen und Realisieren Sie unter Zuhilfenahme der Automatentheorie eine Steuerelektronik, die in einem binären Eingabestrom **E(t)** $\in \Sigma^*$ die Sequenz **111**, d. h. drei hintereinanderfolgende Einsen, erkennt. Am Ausgang der Steuereinheit **A(t)** $\in \Sigma^*$ soll dabei **A = 1** ausgegeben werden, sobald die Sequenz erkannt wurde, ansonsten soll **A = 0** sein.

Zur Lösung der Aufgabe bedienen wir uns dem Modell des deterministischen endlichen Automaten mit einer Ausgabefunktion, kurz **DFAwO**, der aufgrund der Ausgabefunktionalität nun folgende formale Beschreibung erfährt:

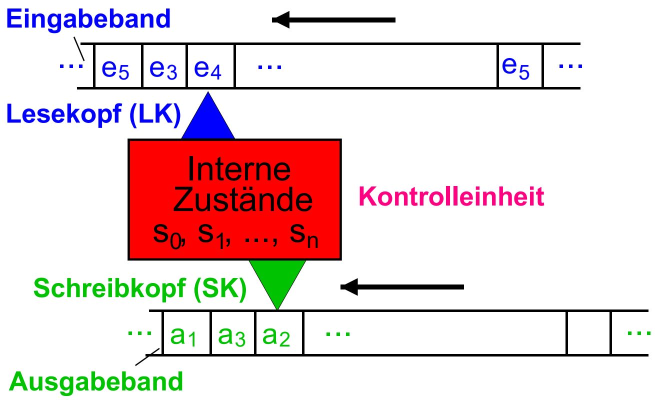
$$\textbf{DFAwO} = (\Sigma = \{0, 1\}, \textbf{S} = \{S_0, S_1, S_2, S_3\}, \delta, \{S_0\}, \textbf{F} = \{S_3\}, \textbf{A} = \{0, 1\}, \alpha)$$

Neben einem Taktgenerator und einigen elementaren Logikgattern (Negation, Konjunktion und Disjunktion) möge Ihnen zur Problemlösung zwei flankengesteuerte JK-Flip-Flops zur Verfügung stehen.

Veranschaulichung:



Zustandsautomat:

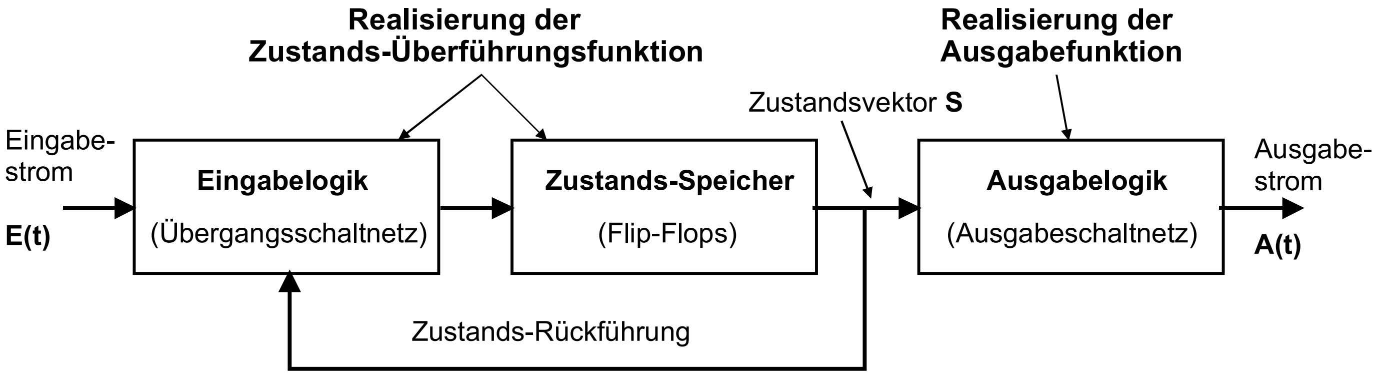


$$\delta : S \times \Sigma \rightarrow S$$

$$\alpha : S \rightarrow A$$

Lösungsidee:

Lese vom Eingabeband E(t) und schreibe auf das Ausgabeband A(t)



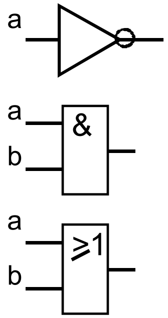
Zustandskodierung:

Logikgatter:

Negation

UND

ODER



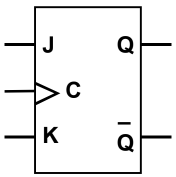
<table>
<tr><th>a</th><th>b</th><th></th><th>a & b</th><th>a | b</th><th>¬a</th></tr>
<tr><td>0</td><td>0</td><td></td><td>0</td><td>0</td><td rowspan="2">1</td></tr>
<tr><td>0</td><td>1</td><td></td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td></td><td>0</td><td>1</td><td rowspan="2">0</td></tr>
<tr><td>1</td><td>1</td><td></td><td>1</td><td>1</td></tr>
</table>

Speicherglieder:

JK-Flip-Flop
(flankengesteuertes)

J = Jump **K** = Kill

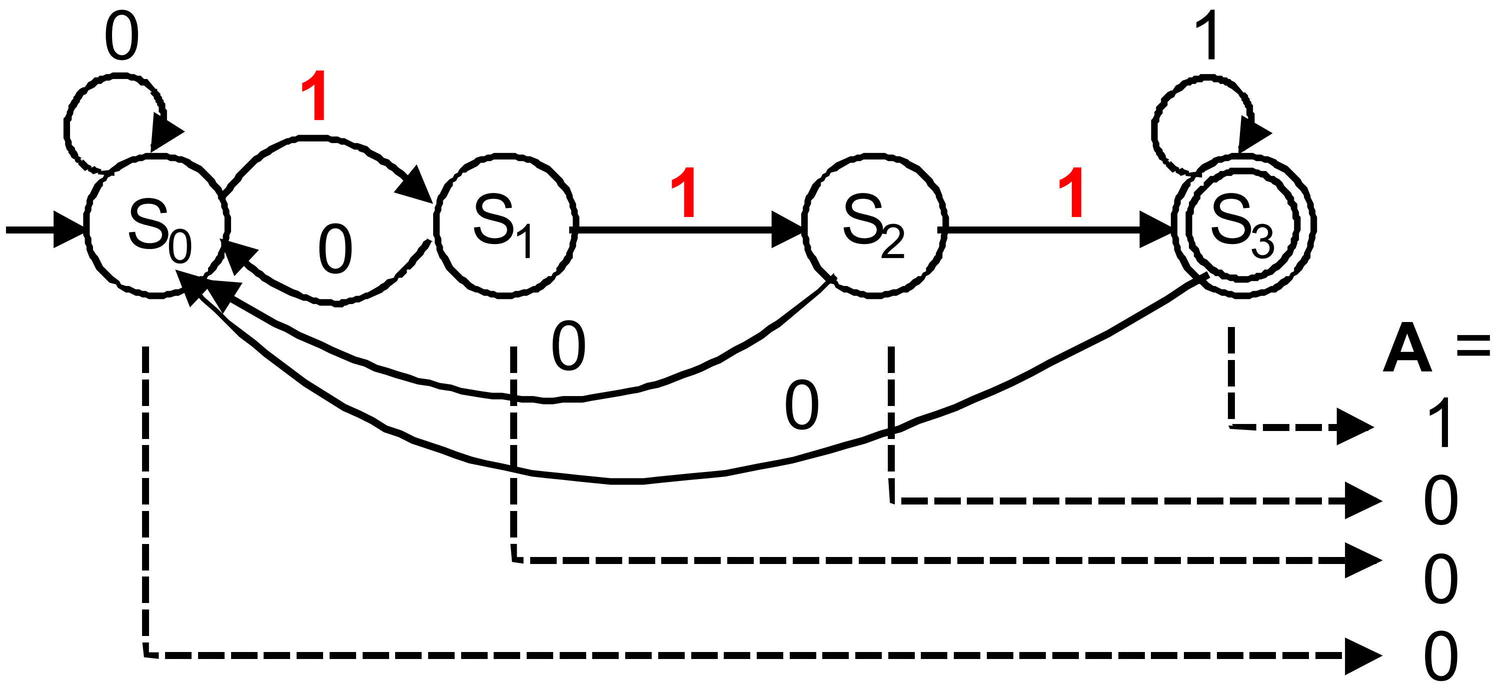
C = Clock



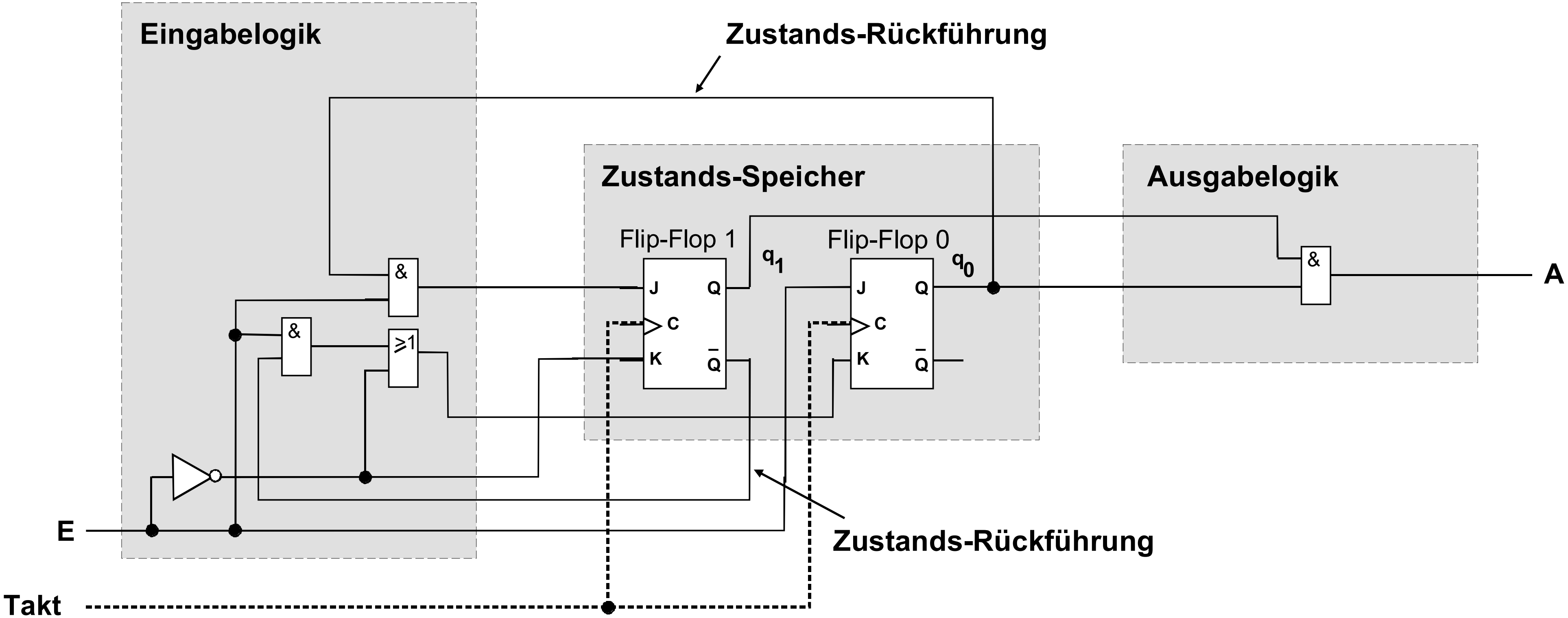
J	K		Q_{neu}	Wirkung
0	0		Q_{alt}	Speichern
0	1		0	Rücksetzen
1	0		1	Setzen
1	1		$\neg Q_{alt}$	Invertieren

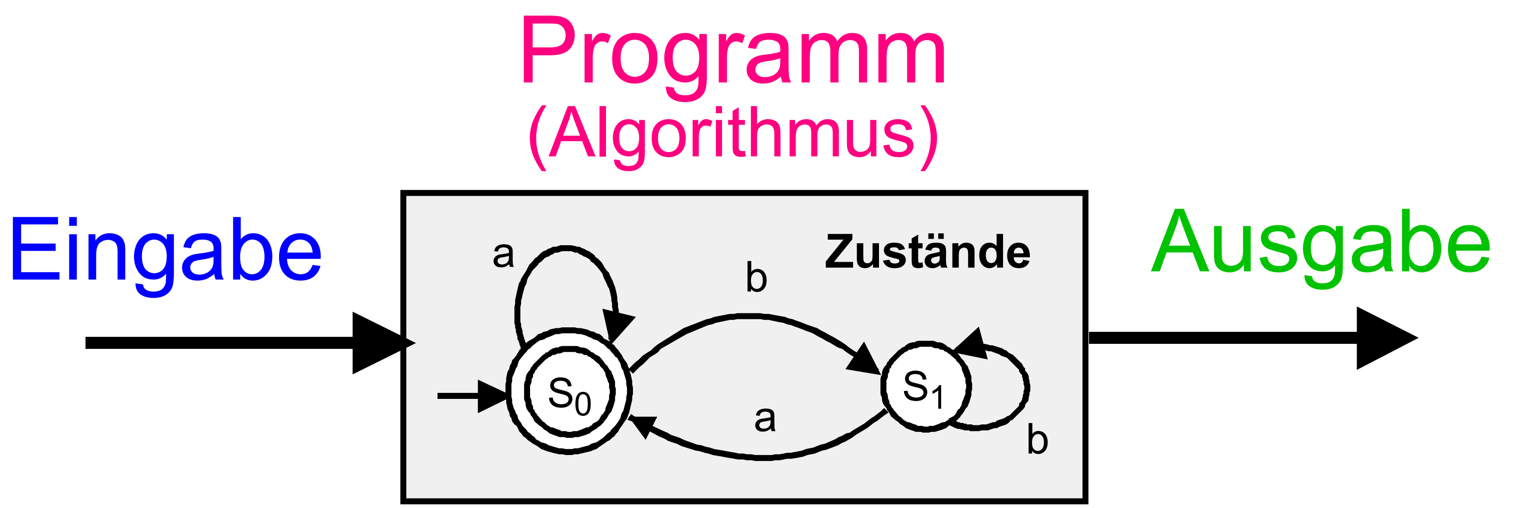
Zustandsdiagramm:

DFAwO = (Σ = {0, 1}, **S** = {S_0, S_1, S_2, S_3}, δ, {S_0}, **F** = {S_3}, **A** = {0, 1}, α)



Ergebnis:


Eingabelogik
Zustands-Rückführung
Zustands-Speicher
Ausgabelogik
Flip-Flop 1
Flip-Flop 0
q1
q0
&
&
≥1
&
J
Q
C
K
Q
J
Q
C
K
Q
A
E
Zustands-Rückführung
Takt



- die **Eingabe** (Übernahme von Daten von außen)
- die **Wertzuweisung** (Zwischenrechnung, Zustände)
- die **Ausgabe** (Übertragung von Variablen nach außen)

Organisation

- Lehrform: 2 SWS Vorlesung, 2 SWS Seminar
- Credits / SWS: **5 cp / 4**
- Gesamtaufwand: **150 h** (etwa 8 h pro Woche)
  - Anwesenheit Vorlesung und Seminar 60 h
  - Vorbereitung und Nachbereitung Vorlesung 30 h
  - Bearbeitung der Übungsaufgaben (Seminar) 60 h
- Ort und Zeit:
  - Vorlesung findet mittwochs um 11:15 Uhr im Raum B 002 statt
  - Seminar erfolgt in kleinen Übungsgruppen gemäß Ankündigung und erfolgter Belegung

- Beim Seminar besteht **anwesenheitspflicht**
  - Eine 75%ige Anwesenheit muss **mindestens** erreicht worden sein
  - Bewerkstelligung von ca. 12 Übungsblättern (je 3 bis 5 Aufgaben)
  - Die Lösungen zu den zur Verfügung gestellten Übungsaufgaben sind zu den jeweilgen Seminarterminen anzufertigen
  - Das Seminar wird benotet (schriftlicher Test plus Übungsaufgaben)
- Klausur am Ende der Vorlesung
  - 90-minütige Klausur (Hilfsmittel: Merkblatt 2 DIN A4 Seiten)
  - Bearbeitung von 6 bis 8 unabhängigen Aufgaben aus dem Themengebiet „Automatentheorie und Formale Sprachen“
  - Die Klausur wird benotet

Folien und Übungsblätter zur Lehrveranstaltung

Befinden sich passwortgeschützt auf dem FB-Server und sind ausschließlich im Rahmen dieser Lehrveranstaltung zu verwenden.

www.cs.hs-rm.de/~rnlab/LVaktuell/AFS/Vorlesung/
www.cs.hs-rm.de/~rnlab/LVaktuell/AFS/Seminar/

Sprechstunde

Außerhalb der Lehrveranstaltungszeiten jeweils

mittwochs zwischen 10:30 und 11:30 Uhr im Raum C 210
oder nach Vereinbarung

Literatur

[1] J. Albert, Th. Ottmann: *Automaten, Sprachen und Maschinen für Anwender*, B.I.-Wissenschaftsverlag, Reihe Informatik/38, Zürich 1983

[2] Uwe Schöning: *Theoretische Informatik kurz gefaßt*, B.I.-Wissenschaftsverlag, Mannheim 1992

[3] Sander, Stucky, Herschel: *Automaten – Sprachen – Berechenbarkeit*, B.G. Teubner, Stuttgart 1992

[4] Ingo Wegner: *Theoretische Informatik – eine algorithmische Einführung*, B.G. Teubner, Stuttgart 1999

[5] M. Broy: *Informatik – Eine grundlegende Einführung*, Teil IV, Theoretische Informatik, Springer, 1995

[6] Daniel I. A. Cohen: *Introduction to Computer Theory*, John Wiley & Sons, Inc., 1997

Automatentheorie und Formale Sprachen

– LV 4110 –

Grammatiken und Formale Sprachen

- Kennenlernen der Begriffe: Formale Sprachen und Ableitung
- Definition eines allgemeinen Erzeugungs- bzw. Ableitungssystems
- Klärung, was man unter der Sprache einer Grammatik versteht
- Definition und Einteilung der Chomsky-Grammatik (Chomsky-Hierarchie bzw. -Typen 0 bis 3)
- Festlegung, was das allgemeine Wortproblem ausdrückt
- Konstruktion einer Grammatik aus gegebenem Automaten und umgekehrt
- Kennenlernen des Vorgehens bei der Erstellung von Ableitungsbäumen

III. Grammatiken und Formale Sprachen

1. Semi-Thue-Systeme
   - 1.1 Idee und Definition des Systems
   - 1.2 Ableitung und Überführung
2. Chomsky-Grammatiken
   - 2.1 Idee und Definition
   - 2.2 Sprache einer Chomsky-Grammatik
3. Chomsky-Hierarchie
4. Endliche Automaten und RL-Grammatiken
   - 4.1 Konstruktion einer Grammatik aus einem Automaten
   - 4.2 Konstruktion eines Automaten aus einer Grammatik

bisher:

Sprachen mit Hilfe von **Automaten** identifiziert und analysiert

jetzt:

Kennenlernen von Formalismen zur **Erzeugung** von Sprachen

$\rightarrow$ Erzeugungssysteme

Solche Erzeugungssysteme, die auf sog. **REGELN**, **PRODUKTIONEN** oder **GRAMMATIKEN** basieren, sind Untersuchungsgegenstand der Theorie der Formalen Sprachen.

III. Grammatiken und Formale Sprachen

1. **Semi-Thue-Systeme**
   - 1.1 **Idee und Definition des Systems**
   - 1.2 Ableitung und Überführung
2. Chomsky-Grammatiken
   - 2.1 Idee und Definition
   - 2.2 Sprache einer Chomsky-Grammatik
3. Chomsky-Hierarchie
4. Endliche Automaten und RL-Grammatiken
   - 4.1 Konstruktion einer Grammatik aus einem Automaten
   - 4.2 Konstruktion eines Automaten aus einer Grammatik

Idee:

- Sprache nicht rein als eine Menge von Wörtern ansehen, sondern definieren, wie man Wörter verändern und manipulieren kann.
- Diese Manipulationen müssen kontrollierbar und nachvollziehbar sein und deshalb nach festen Regeln ablaufen.

Definition:

Ein **Semi-Thue-System** (Norwegischer Mathematiker A. Thue, 1914) wird durch eine endliche Teilmenge **P** $\subseteq \Sigma^* \times \Sigma^*$ bestimmt. Jedes Wortpaar $(\alpha, \beta) \in$ **P** ist eine **Regel** in dem Sinne, dass in einem vorhandenen Ausgangswort **w** ein Teilwort α durch β ersetzt werden kann. Durch die einseitige Ersetzungsrichtung $\alpha \rightarrow \beta$ erklärt sich die Bezeichnung "Semi" .

III. Grammatiken und Formale Sprachen

Definition:

Ein Wort **w** heißt aus einem Wort **v** ***ableitbar***, wenn es durch endlich viele **Ersetzungsschritte** aus **v** entsteht.

Notation:

$v \Rightarrow^* w : \quad v(\alpha 1 \rightarrow \beta 1) \Rightarrow v1(\alpha 2 \rightarrow \beta 2) \Rightarrow \ldots \Rightarrow v_{n-1}(\alpha_n \rightarrow \beta_n) \Rightarrow v_n = w$

kurz:

$\mathbf{v} \Rightarrow^* \mathbf{w} \quad := \quad$ **w** aus **v** **ableitbar** oder **v** in **w** **überführbar**

Ableitung

Beispiel:

$\Sigma = \{a, b, c, d, e\}$ $\qquad$ $v = \{abc\}$ $\qquad$ $w = \{aeb\}$

Regeln des Semi-Thue-Systems **R**:

(1)	$ab \rightarrow ad$	(4)	$ad \rightarrow ae$
(2)	$dc \rightarrow ee$	(5)	$eb \rightarrow b$
(3)	$e \rightarrow b$	(6)	$abc \rightarrow e$

Ableitung: $v \Rightarrow^* w$

$v = \underline{ab}c \;\; (1)\Rightarrow^* \;\; a\underline{d}\underline{c} \;\; (2)\Rightarrow^* \;\; ae\underline{e} \;\; (3)\Rightarrow^* \;\; aeb \; = w$ $\qquad$ q.e.d.

III. Grammatiken und Formale Sprachen

1. Semi-Thue-Systeme
   - 1.1 Idee und Definition des Systems
   - 1.2 Ableitung und Überführung
2. **Chomsky-Grammatiken**
   - **2.1 Idee und Definition**
   - 2.2 Sprache einer Chomsky-Grammatik
3. Chomsky-Hierarchie
4. Endliche Automaten und RL-Grammatiken
   - 4.1 Konstruktion einer Grammatik aus einem Automaten
   - 4.2 Konstruktion eines Automaten aus einer Grammatik

Amerikanischer Sprachwissenschaftler **Noam Chomsky**, 50. Jahren

Idee:

- Das Alphabet Σ besteht aus zwei disjunkten Teilmengen **T** und **N**.
- **T** steht für die Menge der Terminalsymbole, d. h. Menge atomarer Zeichen, aus denen letztlich alle Wörter einer Sprache aufgebaut sind.

  Bsp.: Zeichen a, b, ..., z und 0, 1, ..., 9
- **N** steht für die Menge der **N**onterminalsymbole, die zur Abstraktion und Klassenbildung von Wörtern dienen; sog. syntaktische Kategorien

  Bsp.: Begriffe wie <Name>, <Buchstabe>, <Ziffern>

Ableitung

Definition

Beispiel: „Namensgebung in PASCAL“
hier: erweiterte Backus-Naur-Form (BNF)

<Name>	::=	<Buchstabe> { <Buchstabe> | <Ziffer> }
<Buchstabe>	::=	a | b | c | ... | z
<Ziffer>	::=	0 | 1 | 2 | ... | 9

::=	Überführung oder Ersetzung
|	Auswahl unter mehreren Alternativen
{ ... }	beliebige Wiederholung

Ergebnis: Ein korrekter Name muss mit einem Buchstaben beginnen, anschließend können weitere Buchstaben und Ziffern in beliebiger Zahl folgen.

Definition:

Unter einer ***Chomsky-Grammatik*** verstehen wir ein **Quadrupel** G = (**N**, **T**, **P**, S), wobei **P** ein **Semi-Thue-System** über dem Alphabet $\Sigma^* = (\mathbf{N} \cup \mathbf{T})^*$ ist. Die einzelnen Komponenten haben folgende Bedeutung:

N Menge der Nonterminalsymbole A,B,C, ...

T Menge der Terminalsymbole a,b,c, ...

P Menge der Produktionen (Regeln):

$\mathbf{P} \subset \{ \varphi \rightarrow \psi \mid \varphi, \psi \in (\mathbf{N} \cup \mathbf{T})^*, \varphi \neq \varepsilon \}$

S Startsymbol $\in$ **N**, das in mindestens einer Regel links vorkommt

Interpretation:

- Man hat endliche Menge von Produktionen **P** ($\rightarrow$ auch Regeln genannt), die Terminal- und Nonterminalsymbole beinhalten können.
- Erzeugung von Wörtern beginnt immer mit dem Startsymbol S.
- Linke Seite einer Regel darf nicht das leere Wort ε sein.
- Die erzeugenden Wörter bestehen letztendlich nur aus Terminalzeichen T.

III. Grammatiken und Formale Sprachen

1. Semi-Thue-Systeme
   1.1 Idee und Definition des Systems
   1.2 Ableitung und Überführung
2. Chomsky-Grammatiken
   2.1 Idee und Definition
   2.2 Sprache einer Chomsky-Grammatik
3. Chomsky-Hierarchie
4. Endliche Automaten und RL-Grammatiken
   4.1 Konstruktion einer Grammatik aus einem Automaten
   4.2 Konstruktion eines Automaten aus einer Grammatik

Definition:

Die Sprache einer Chomsky-Grammatik besteht aus allen Worten aus **T***, die aus S ableitbar sind, d. h. für die gilt:

$$L(G) = \{ w \in \mathbf{T}^* \mid S \Rightarrow^* w \}.$$

heißt:

$$S \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_n = w$$

mit $n \in \mathbb{N}_0$

$u_i \in (\mathbf{N} \cup \mathbf{T})^*$

$w \in \mathbf{T}^*$

kurz: $S \Rightarrow^* w$

Ableitung

Definition

Beispiel: „Namensgebung in PASCAL“

Mit

N = { <Name>, <Buchstabe>, <Ziffer> }

T = { a, b, c, ..., z, 0, 1, 2, …, 9 }

P = { <Name> → <Name> <Ziffer>, <Name> → <Buchstabe>,
<Buchstabe> → a | b | c | ... | z, <Ziffer> → 0 | 1 | 2 | ... | 9 }

S = <Name>

gehört das Wort „**a12**“ zur Sprache L(G) obiger Grammatik
G = (**N**, **T**, **P**, S), weil sich folgende Ableitung angeben lässt:

<Name> → <Name> <Ziffer> → <Name> <Ziffer> <Ziffer>
→ <Buchstabe> <Ziffer> <Ziffer> → **a12**

III. Grammatiken und Formale Sprachen

Grundsätzliche Bemerkung:

Die bisherige Definition einer Chomsky-Grammatik ist so allgemein, dass nicht entscheidbar ist, ob ein Wort zur Sprache einer vorgegebenen Grammatik gehört oder nicht (vgl. allgemeines Wortproblem).

Deshalb ist es sinnvoll, Einschränkungen zu treffen, so dass dieses Problem lösbar wird.

$\Rightarrow$ vernünftige Hierarchie von Einschränkungen!

Im folgenden wollen wir uns einer entsprechenden Einteilung (Typ 0 bis Typ 3) zuwenden.

Definition:

Eine Chomsky-Grammatik G = (**N**, **T**, **P**, S) heißt

- **Typ 0** oder **allgemeine** Chomsky-Grammatik, wenn alle Regeln die nicht eingeschränkte Form

$\alpha \rightarrow \beta$ mit $\alpha \neq \varepsilon$ ε = leeres Element

$\alpha, \beta \in (\mathbf{N} \cup \mathbf{T})^*$

haben ($\rightarrow$ Ursprungs-Definition!).

Definition:

Eine Chomsky-Grammatik G = (**N**, **T**, **P**, S) heißt

- **Typ 1** oder **kontextsensitive** Grammatik, wenn alle Regeln die Form

  $\alpha A \beta \to \alpha \gamma \beta$ mit $\gamma \neq \varepsilon$

  $A \in \mathbf{N}$; $\alpha, \beta, \gamma \in (\mathbf{N} \cup \mathbf{T})^*$

  haben mit der Ausnahme, dass S $\to \varepsilon$ dazugehören darf, wenn S in keiner Regel auf der rechten Seite auftritt.

Definition:

Eine Chomsky-Grammatik G = (**N**, **T**, **P**, S) heißt

- **Typ 2** oder **kontextfreie** Grammatik, wenn alle Regeln die Form

$$\mathbf{A} \to \gamma \quad \text{mit} \quad \mathbf{A} \in \mathbf{N}\,; \quad \gamma \in (\mathbf{N} \cup \mathbf{T})^*$$

$$\text{oder} \quad \gamma = \varepsilon$$

haben.

<u>Definition</u>:

Eine Chomsky-Grammatik G = (**N**, **T**, **P**, S) heißt

- **Typ 3** oder **rechtslineare***) Grammatik, wenn alle Regeln die Form

$$A \to \varepsilon \; ; \; A \to a \quad \text{oder} \quad A \to aB \; ;$$

mit $a \in \mathbf{T}$; $A, B \in \mathbf{N}$

haben.

*) d. h. nur ein **N**onterminalsymbol ganz rechts!

Anmerkung:

Jede Typ-3-Grammatik ist auch vom Typ 2, und jede Typ-1-Grammatik auch vom Typ 0.

Definition:

Zwei Grammatiken heißen ***äquivalent***, wenn sie die gleiche Sprache erzeugen.

Definition:

Eine Sprache L(G) heißt vom ***Chomsky-Typ i*** (i=0,1,2,3) wenn G vom Typ i ist. Die Familie der Sprachen vom Typ i wird mit $\mathbf{L}_i$ bezeichnet.

Satz:

Es gelten die Beziehungen: $\mathbf{L}_3 \subset \mathbf{L}_2 \subset \mathbf{L}_1 \subset \mathbf{L}_0$ und

$$\mathbf{L}_3 = \{ L(A) \mid A \text{ ist endlicher Automat} \}$$

III. Grammatiken und Formale Sprachen

Rechts- und linkslineare Grammatiken

Die Regeln der rechts- (oder links-) linearen Grammatiken sind innerhalb der Chomsky-Hierarchie am stärksten eingeschränkt.

rechtslinear: **$A \to aB$**

Nonterminal-zeichen	nur ersetzbar durch	Terminal-zeichen	+ ggf.	Nonterminal-zeichen
A		**a**		**B**

Bei Rechtslinearität erfolgt Einsetzung immer **rechts**, d. h. am Ende des bereits abgeleiteten Wortes.

Da immer nur ein zusätzliches Terminalzeichen hinzugefügt wird, wächst das entstehende Wort in einer Richtung **linear** mit der Anzahl der Ersetzungsschritte.

Satz:

Zu jeder **rechtslinearen** Grammatik gibt es eine **linkslineare**, die die gleiche Sprache erzeugt.

Satz:

Zu jeder rechtslinearen Grammatik mit der Sprache L(G) gibt es einen endlichen Automaten mit

$$L(A) = L(G)$$

und umgekehrt.

III. Grammatiken und Formale Sprachen

1. Semi-Thue-Systeme
   1.1 Idee und Definition des Systems
   1.2 Ableitung und Überführung
2. Chomsky-Grammatiken
   2.1 Idee und Definition
   2.2 Sprache einer Chomsky-Grammatik
3. Chomsky-Hierarchie
4. Endliche Automaten und RL-Grammatiken
   4.1 Konstruktion einer Grammatik aus einem Automaten
   4.2 Konstruktion eines Automaten aus einer Grammatik

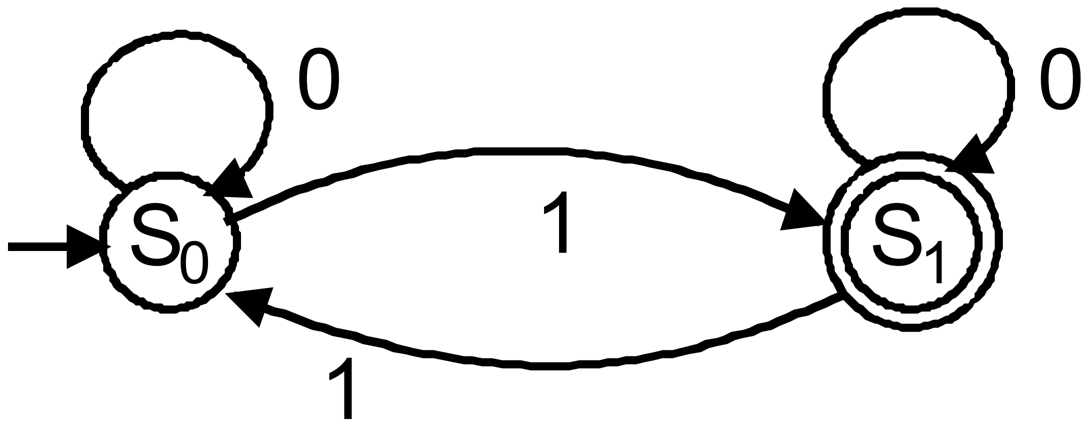
Konstruktion einer RL G aus einem A = ($\mathbf{S}$, s_0, $\mathbf{F}$, Σ, δ):

- verwende die Eingabezeichen Σ als Terminalzeichen **T**
- verwende die Menge **S** der Zustände als Nonterminalzeichen **N**
- verwende s_0 als Startsymbol S
- ersetze jeden Funktionswert $\boldsymbol{\delta(s_1,a) = s_2}$ durch die Regel $\boldsymbol{s_1 \rightarrow a\, s_2}$
- erzeuge für jeden Endzustand s_f eine zusätzliche Regel $\boldsymbol{s_f \rightarrow \varepsilon}$

Ungerade Anzahl von Einsen:

Wir betrachten folgenden DFA, der alle Worte akzeptiert, die eine ungerade Anzahl von Einsen enthalten.

DFA **A** = (Σ, **S**, S_0, δ, **F**) mit Σ = {0, 1}, **S** = {S_0, S_1}, **F** = {S_1} und δ gemäß folgendem Zustandsdiagramm:



Ungerade Anzahl von Einsen:

Hierauf folgt unter Anwendung der Regeln die Grammatik
G = (**N**, **T**, **P**, S) mit

- Eingabezeichen Σ als Terminalzeichen **T** → $\mathbf{T} = \Sigma = \{0, 1\}$
- Zustandsmenge **S** als Nonterminalzeichen **N** → $\mathbf{N} = \{S_0, S_1\}$
- Anfangszustand S_0 als Startsymbol S → $S = S_0$
- Jeder Funktionswert $\boldsymbol{\delta(s_1,a) = s_2}$ eine Regel der Form $\boldsymbol{s_1 \to a\ s_2}$ →

  $\mathbf{P} = \{ S_0 \to 1\ S_1 ,\ S_0 \to 0\ S_0 ,\ S_1 \to 1\ S_0 ,\ S_1 \to 0\ S_1 \}$
- Jeden Endzustand s_f eine zusätzliche Regel $\boldsymbol{s_f \to \varepsilon}$ → $S_1 \to \varepsilon$

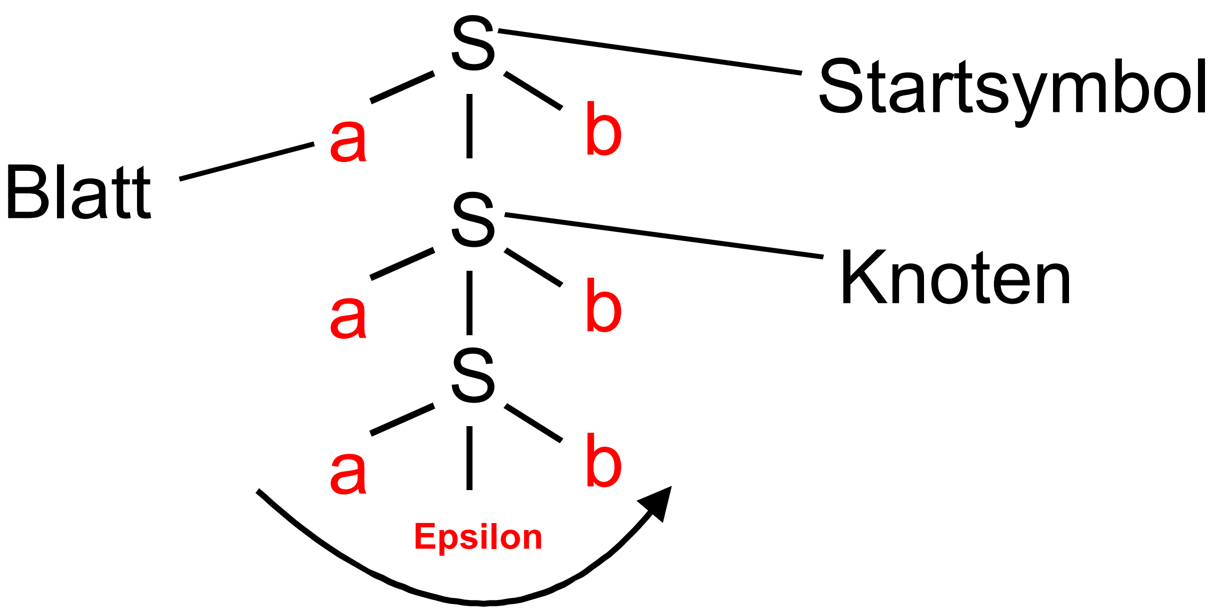
⇒ **P** und δ sind **ähnlich ausdrucksstarke** Konzepte!

III. Grammatiken und Formale Sprachen

<u>Konstruktion eines A aus einer RL G = (**N**, **T**, **P**, S)</u>:

- verwende die Terminalzeichen **T** als Eingabezeichen Σ
- verwende die Nonterminalzeichen **N** als Zustände **S**
- verwende das Startsymbol S als Ausgangszustand s_0
- verwende jedes A, für das $\mathbf{A} \rightarrow \varepsilon$ gilt, als Endzustand s_f
- ersetze jede Produktion $\mathbf{A} \rightarrow \mathbf{aB}$ durch einen Funktionswert $\boldsymbol{\delta}(\mathbf{A},\mathbf{a}) = \mathbf{B}$
- führe für ***alle*** Produktionen $\mathbf{A} \rightarrow \mathbf{a}$ *einen* Endzustand s_f ein und
- bilde für ***jede*** solche Produktion den Funktionswert $\delta(A,a) = s_f$

Ableitungsbaum des Wortes **aaabbb**:



Blätter von links nach rechts gelesen ergeben das Wort w.

Automatentheorie und Formale Sprachen

– LV 4110 –

Reguläre Sprachen und Mengen

- Kennenlernen der Begriffe: Reguläre Sprachen und reguläre Ausdrücke
- Definition der Operationen, die – angewandt auf reguläre Sprachen – wieder reguläre Sprachen erzeugen
- Definition einer Operations-Hierarchie
- Elementarautomaten für die Verkettung, die Potenz und für die Iteration
- Kennenlernen der Vorgehensweise bei der Zusammenführung von Elementarautomaten
- Entwicklung und Anwendung von Suchalgorithmen und Texterkennungsprozeduren: Skelettautomaten, goto- und failure-Funktion

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
    - 1.1 Mengenoperatoren
    - 1.2 Operations-Hierarchie
2. Endliche Automaten und reguläre Sprachen
    - 2.1 Elementarautomaten
    - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
    - 3.1 Deterministische Automaten und einfache Algorithmen
    - 3.2 Ein einfaches pattern-matching Problem
    - 3.3 Gleichzeitiges Suchen nach mehreren Schlüsselworten

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie
2. Endliche Automaten und reguläre Sprachen
   - 2.1 Elementarautomaten
   - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

Die Menge aller Sprachen, die von einem endlichen Automaten akzeptiert werden, nennt man auch die Familie der regulären Sprachen.

Fragestellungen:

1. Welche Operationen auf reguläre Sprachen erzeugen wieder reguläre Sprachen?
2. Wie findet man zu einer regulären Sprache den „einfachsten“ deterministischen Automaten?
3. Welche Probleme sind für reguläre Sprachen algorithmisch lösbar?

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke

- **1.1 Mengenoperatoren**
- 1.2 Operations-Hierarchie

2. Endliche Automaten und reguläre Sprachen

- 2.1 Elementarautomaten
- 2.2 Zusammenführung von Teilautomaten

3. Anwendungen in der Texterkennung

- 3.1 Deterministische Automaten und einfache Algorithmen
- 3.2 Ein einfaches pattern-matching Problem

Definition:

Ein *regulärer Ausdruck* besteht aus Zeichen eines Alphabets und/ oder anderen regulären Ausdrücken, die durch die Operationen *Iteration* (*), *Verkettung* (**...**) oder *Wahlmöglichkeit* (**|**) miteinander verbunden sind. Jedem regulären Ausdruck α entspricht eine Wortmenge $L(\alpha)$ aus Σ^*, die als *reguläre Menge oder reguläre Sprache* bezeichnet wird.

Interpretation:

Ein regulärer Ausdruck kann also verstanden werden als *Formel*, die beschreibt, wie die Wörter einer Sprache, d. h. einer gewissen Untermenge von Σ^* aus den Zeichen des Alphabets Σ, anderen Formeln und den genannten Operationen zu bilden sind.

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - **1.2 Operations-Hierarchie**
2. Endliche Automaten und reguläre Sprachen
   - 2.1 Elementarautomaten
   - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

<u>Operations-Hierarchie</u>:

Ähnlich wie bei algebraischen Formeln gibt es eine Operationen-Hierarchie:

1. Iteration (*)
2. Verkettung (...)
3. Wahlmöglichkeiten (|)

Den Operationen Iteration, Verkettung und Auswahl zur Verknüpfung von regulären Ausdrücken entsprechen im Bereich der zugehörigen regulären Mengen aus Σ^* die *Mengenoperationen:*

Iteration, *Mengenprodukt* und *Vereinigung*.

Regulärer Ausdruck α vs. Endlicher Automat **A**:

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

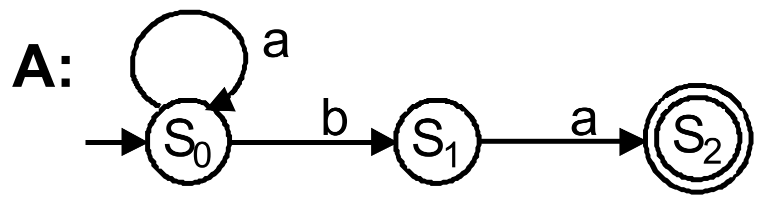
Regulärer Ausdruck: $\alpha = a^*ba$

Anmerkung: Der Stern bedeutet in diesem Zusammenhang die Hintereinanderreihung von a.

Wortmenge $L(\alpha)$:

$L(\alpha) = \{ba, aba, aaba, aaaba, aaaaba, ...\}$
$= T(\mathbf{A})$

Automatenmodell **A**:



DFA **A** := ($\Sigma = \{a, b\}$, $S = \{S_0, S_1, S_2\}$, S_0, δ, $F = \{ S_2\}$)

Definitionen:

Seien L_1, L_2 Mengen von Wörtern über dem Alphabet Σ.

1. Verkettung oder Mengenprodukt

Man definiert als Mengenprodukt von L_1 und L_2 die Menge $L_1 L_2$ durch:

$$L_1 L_2 = \{ w_1 w_2 \in \Sigma^* \mid w_1 \in L_1 \text{ und } w_2 \in L_2 \}$$

Definitionen:

Sei L eine Menge von Wörtern über dem Alphabet Σ.
Ferner sei ε das leere Eingabewort.

2. Potenz

Man definiert die Potenz $L^{(i)}$ von L für $i \geq 0$ durch:

$$L^{(0)} = \{ \varepsilon \} ;$$

$$L^{(1)} = L ;$$

$$L^{(i+1)} = L^{(i)} L .$$

Definitionen:

Sei L eine Menge von Wörtern über dem Alphabet Σ.
Ferner sei ε das leere Eingabewort.

3. Iteration

$\cup$ = Vereinigungsmenge

Man definiert die Iteration L* von L als:

$$L^* = \{ \varepsilon \} \cup \{ w_1 w_2 \ldots w_n \} \mid w_i \in L$$

$$\text{für} \quad i = 1, 2, \ldots, n ; \quad n = 1, 2, \ldots, \infty$$

$$= \{ \varepsilon \} \cup L \cup LL \cup LLL \cup \ldots$$

$$= \{ \varepsilon \} \cup L^{(1)} \cup L^{(2)} \cup L^{(3)} \cup \ldots = \bigcup_{i=0}^{\infty} L^{(i)}$$

Definition:

Nun können wir die Begriffe „regulärer Ausdruck“ und „reguläre Sprache“ **induktiv** wie folgt definieren. Es sei dabei Σ ein endliches Alphabet; dann gilt vereinbarungsgemäß:

(1) ε ist ein regulärer Ausdruck über Σ mit der Sprache $L(\varepsilon) = \{\varepsilon\}$.

(2) Jedes $a \in \Sigma$ ist ein regulärer Ausdruck über Σ mit der Sprache $L(a) = \{a\}$.

(3) Sind α und ß reguläre Ausdrücke über Σ, so sind auch α^*, $\alpha ß$ und $\alpha|ß$ reguläre Ausdrücke und die zugehörigen Sprachen sind:

$$L(\alpha^*) = (L(\alpha))^* \; ; \;\; L(\alpha ß) = L(\alpha)L(ß) \;\; \text{und} \;\; L(\alpha|ß) = L(\alpha) \cup L(ß).$$

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie
2. **Endliche Automaten und reguläre Sprachen**
   - 2.1 Elementarautomaten
   - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

Satz:

Zu jedem regulären Ausdruck α gibt es einen Automaten A (mit genau einem Anfangs- und einem Endzustand), dessen Sprache **T**(A) identisch ist mit der regulären Sprache **L**(α), d. h.

$$\mathbf{T}(A) = \mathbf{L}(\alpha)$$

und dessen Zustandszahl von der Länge des Ausdrucks α abhängt.

Beweis:

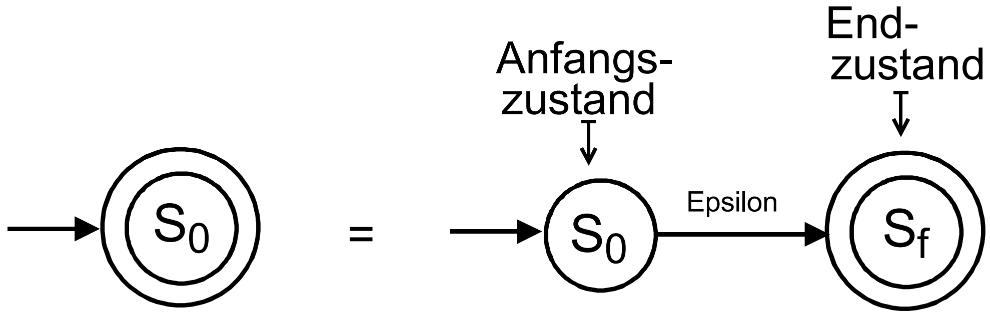
Der Beweis erfolgt durch Angabe von Elementarautomaten für (1) und (2) in der Definition und die Konstruktion entsprechend zusammengesetzter Automaten gemäß Regel (3):

II. Reguläre Sprachen und Mengen

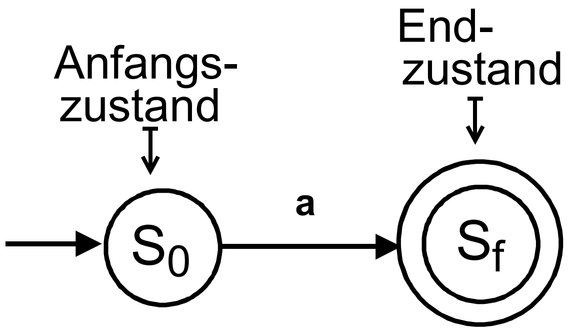
1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie
2. Endliche Automaten und reguläre Sprachen
   - **2.1 Elementarautomaten**
   - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

Elementarautomaten

Elementarautomat für (1) → Aε:

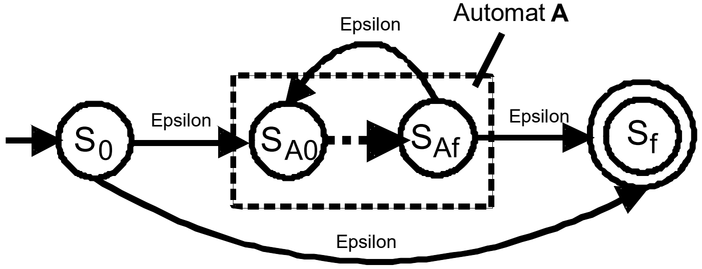


Elementarautomat für (2) → Aa:

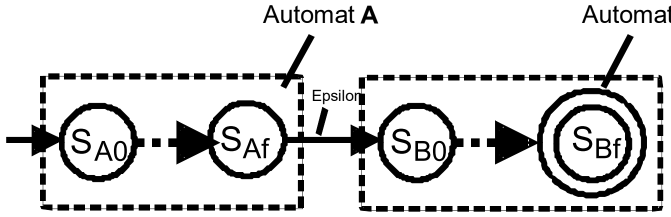


Elementarautomat für (3) mit $L(\alpha) = T(A)$ sowie $L(\beta) = T(B)$:

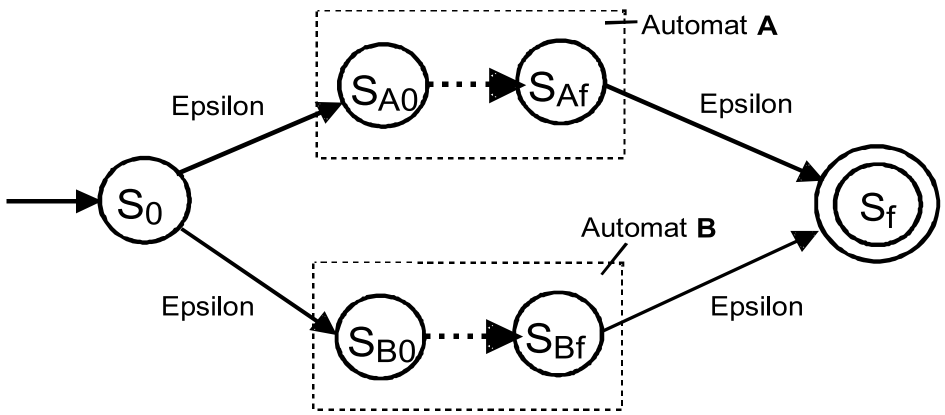
i) für α^*:



ii) für $\alpha ß$:



iii) für $\alpha|ß$:



II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie
2. Endliche Automaten und reguläre Sprachen
   - 2.1 Elementarautomaten
   - **2.2 Zusammenführung von Teilautomaten**
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - 3.2 Ein einfaches pattern-matching Problem

Vorgehensweise:

1. Mit Hilfe der Elementarautomaten für α^*, $\alpha\ \beta$ und $\alpha \mid \beta$ lassen sich durch sukzessive Anwendung Zustandsautomaten mit **spontanen ε-Übergängen** rekonstruieren.

2. Aus diesen sog. **ε-Automaten** können wir dann **nicht-deterministische** Automaten **ohne** ε-Übergänge ableiten, die dieselben Mengen von Worten akzeptieren.

3. Schließlich lassen sich die NFA in **deterministische** Automaten (DFA) überführen (**Teilmengenverfahren**) und letztere ggf. noch optimieren (Zusammenlegen äquivalenter Zustände $\rightarrow$ **Minimal-automat**).

<u>Aufgabe</u>:

Gesucht ist der Zustandsautomat für folgenden regulären Ausdruck:

$$\alpha = a\ (\ a \mid bb\ )^*$$

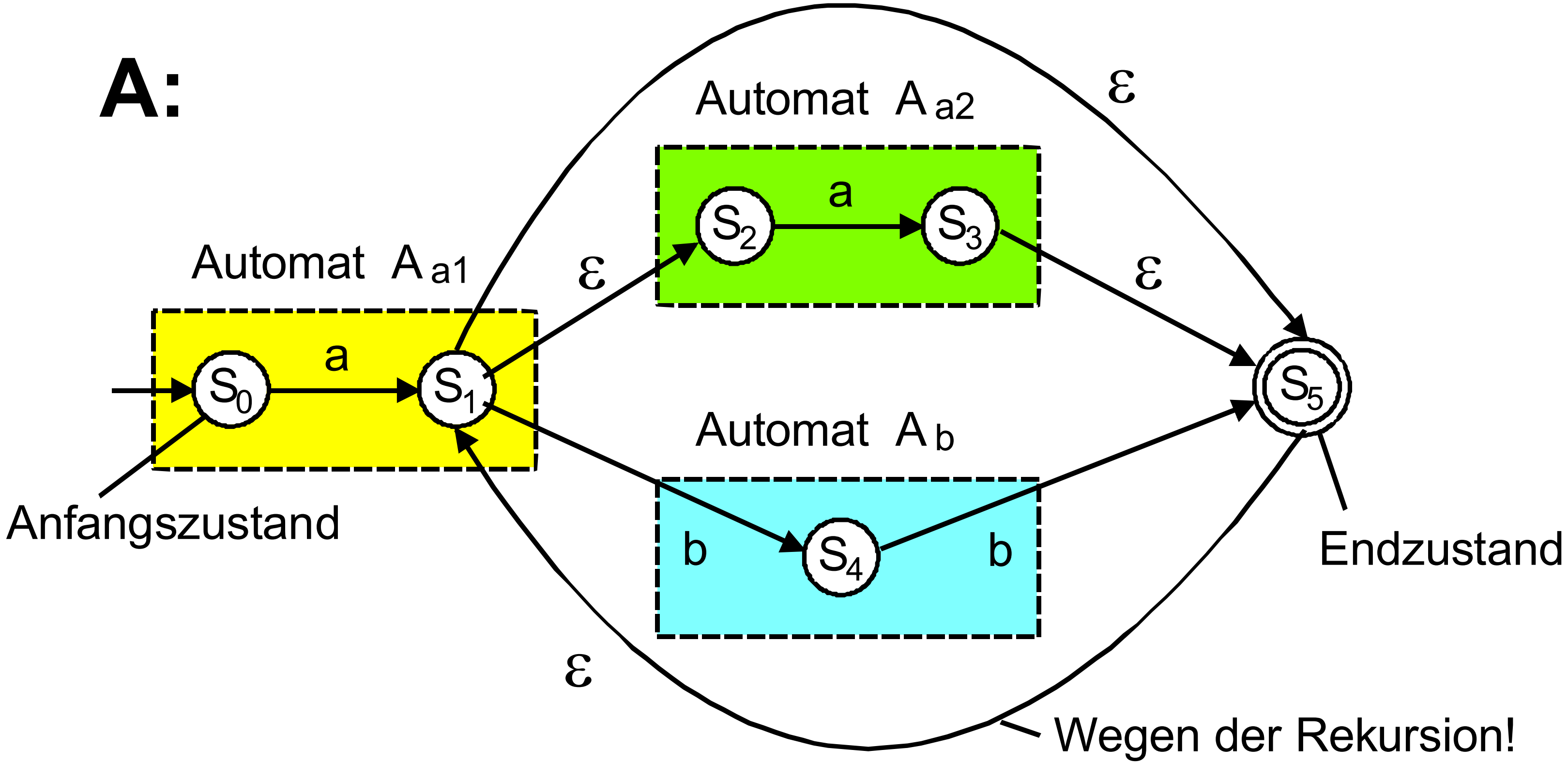
<u>Interpretation</u>:

Alle Wörter, die mit a beginnen, gefolgt von Teilwörtern, die nur aus Zeichen der Form bb oder a bestehen.

<u>Lösungsidee</u>:

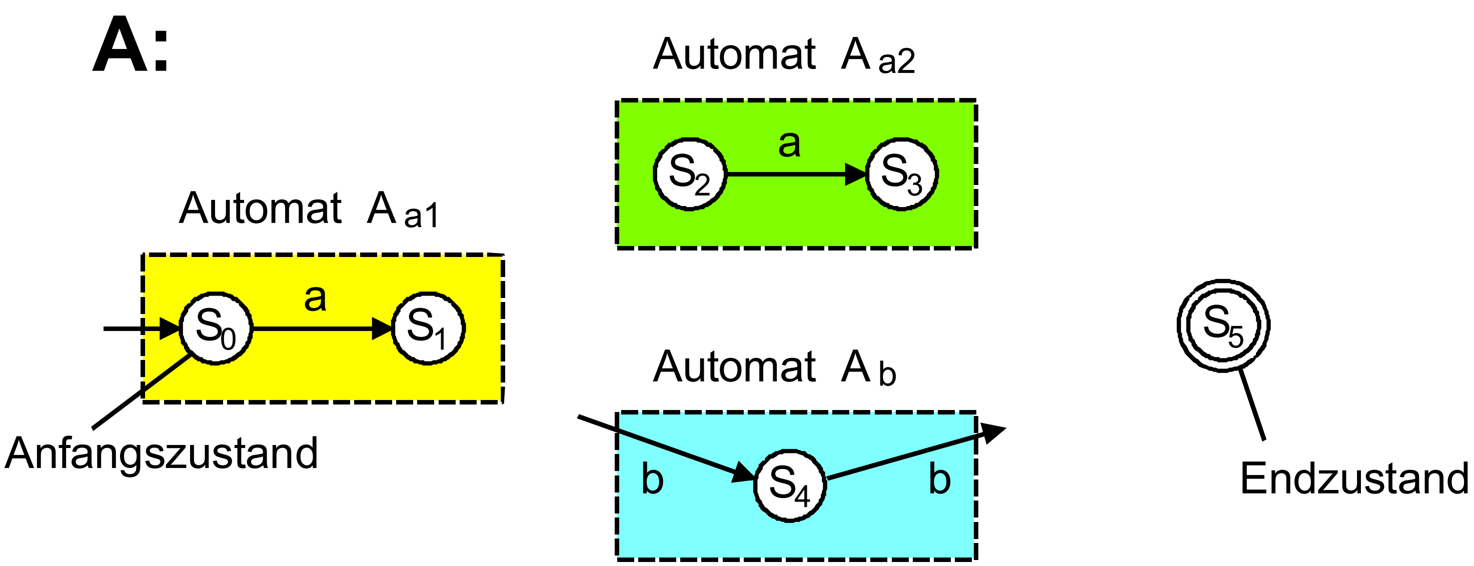
Konstruktion des gesuchten Automaten A aus **drei** Teilautomaten A_{a1}, A_{a2} und A_b in Verbindung mit spontanen ε-Übergängen.

Komposition:



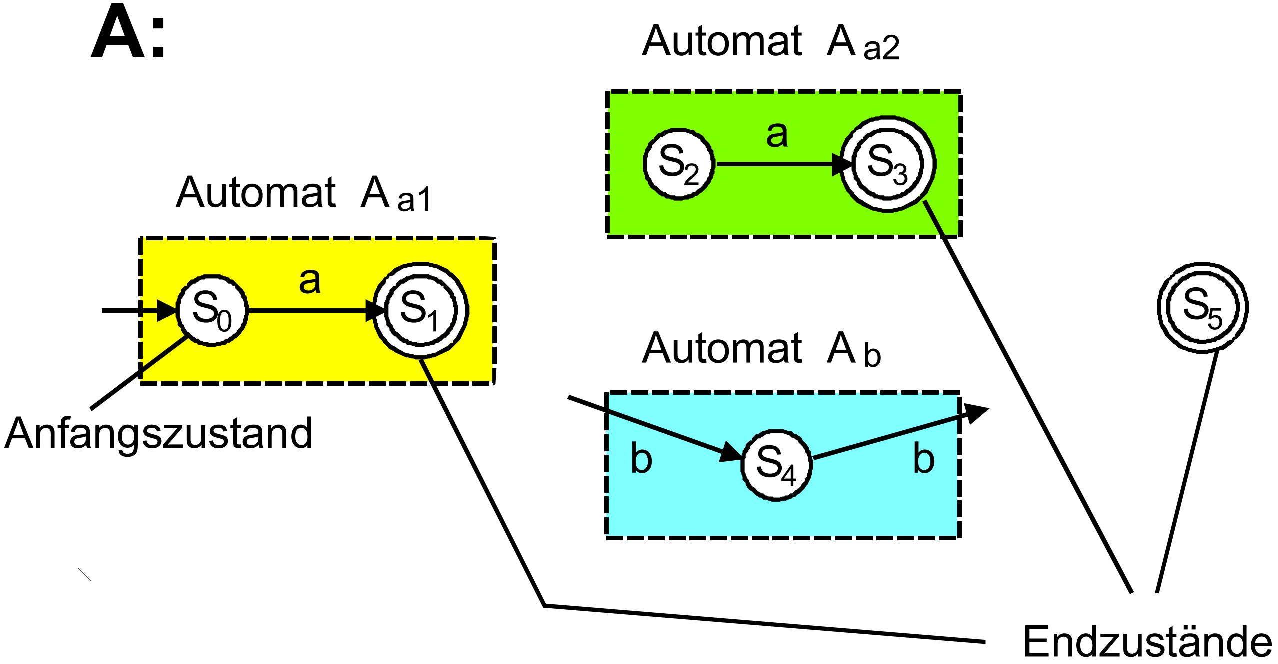
Umwandlung in Automaten ohne ε-Übergänge:

1. Schritt: Übertragen der Zustände

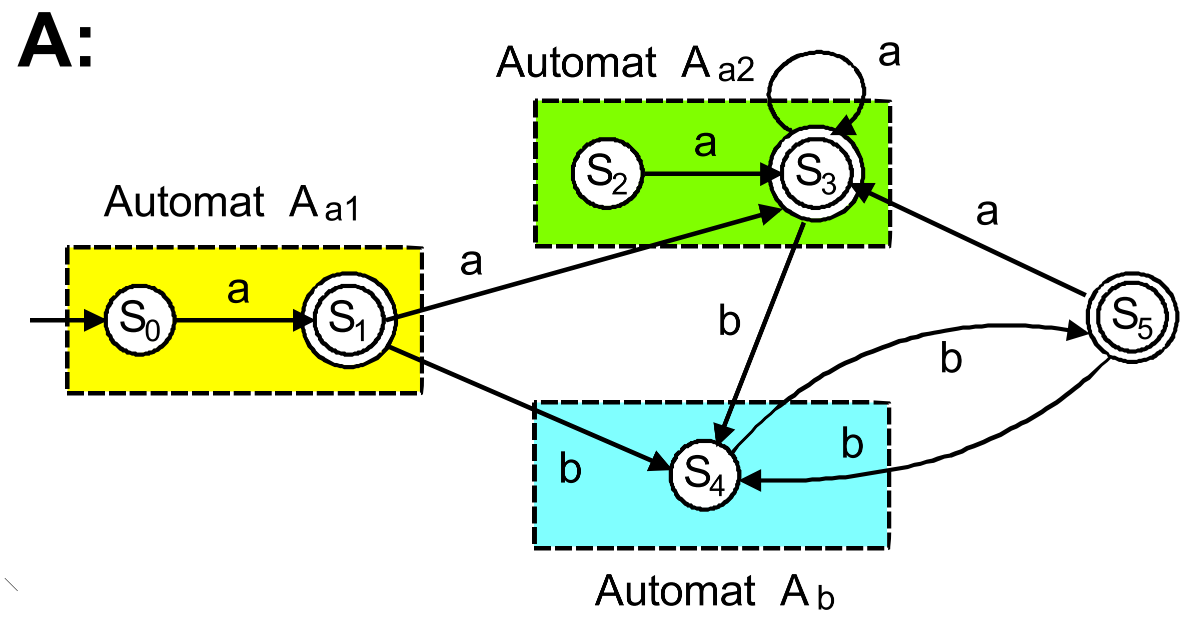


2. Schritt: S_1 und S_3 zu einem Endzustand machen, da man von S_1 bzw. S_3 durch ε-Übergänge in den Endzustand des ε-Automaten kommt.

A:

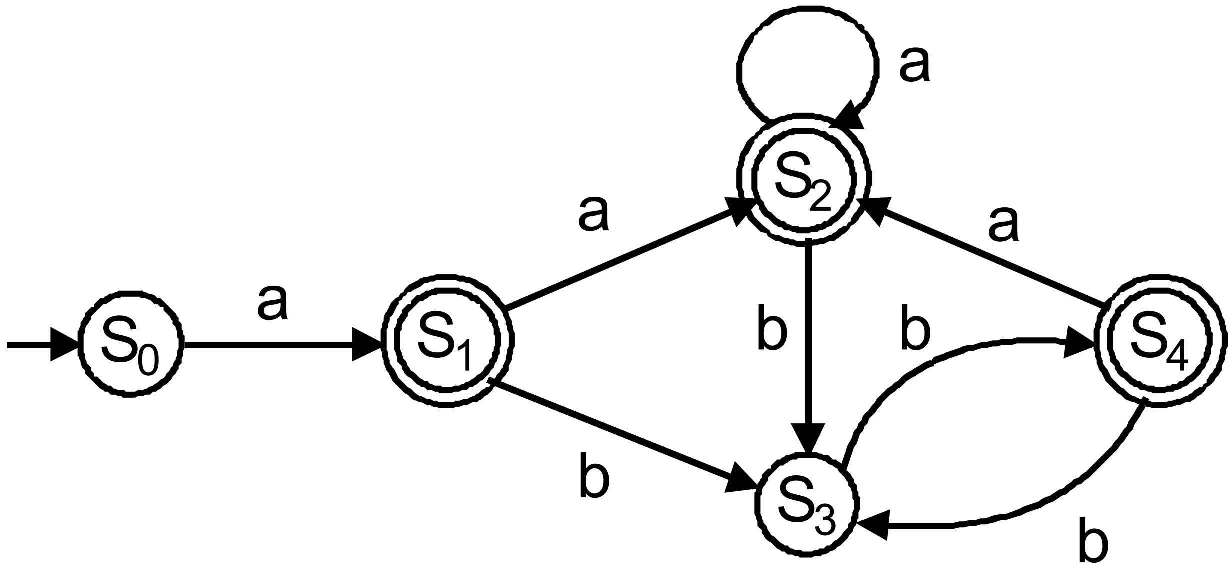


3. Schritt: Ersetzen der ε-Übergänge durch Nicht-ε-Übergänge.



4. Schritt: Entfernen des Zustands S_2, weil dieser nicht erreichbar.

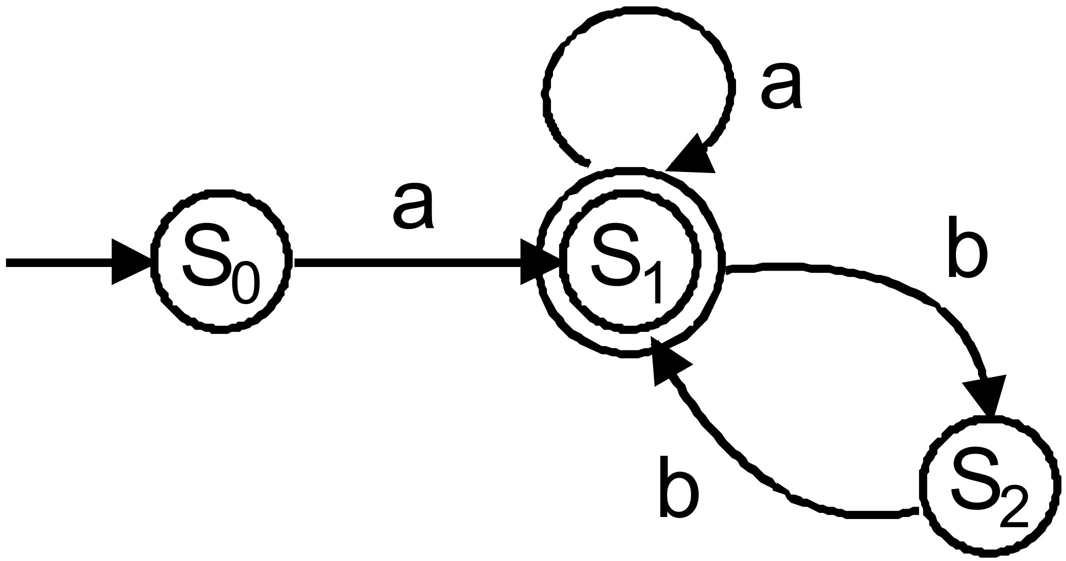
Ergebnis: **A:**



$\rightarrow$ $\Sigma = \{a, b\}$, $\mathbf{S} = \{S_0, S_1, S_2, S_3, S_4\}$ und $\mathbf{F} = \{S_1, S_2, S_4\}$

5. Schritt: Reduzierung zum Minimalautomaten A'.

<u>Endergebnis</u>: **A':**



$\rightarrow \Sigma = \{a, b\}$, $\mathbf{S} = \{S_0, S_1, S_2\}$ und $\mathbf{F} = \{S_1\}$

Ziel:

Umsetzung eines deterministischen endlichen Automaten (DFA) in ein Programm.

Idee:

Zustände des Automaten als Sprungmarken ansehen.

- {neue Sprungmarke mit neuem Zeichen}
  = g(aktuelle Sprungmarke mit aktuellem Zeichen).
- Delimiterzeichen zur Kennzeichnung des Wortendes.
- Unterfunktion FAIL(.), die zur Anwendung kommt, wenn eingelesenes Zeichen im aktuellen Zustand nicht erlaubt ist.

Automaten für reguläre PASCAL-Zahlen


```
                    procedure numbertest(.);
DIGIT := { ´0´, ... ,´9´}
SIGN := {´+´,´-´}
...
Label: 0,1,2,3,4,5,6,7;
0: a := NEXTCHAR(.);
  if a in SIGN then goto 1
  else if a in DIGIT then goto 2 else FAIL(.);
1: a := NEXTCHAR(.);
  if a in DIGIT then goto 2 else FAIL(.);
2: a := NEXTCHAR(.);
...
7: a := NEXTCHAR(.);
  if a in DIGIT then goto 7
  else if a in DELIMITER then write („Zahldarstellung o.k.“)
        else FAIL(.)
```

Es seien:

a = Eingabezeichen

Zustand_neu = g(Zustand_aktuell, a)

```
                         procedure numbertest(.);
...                               << Wenn Eingabezeichen im aktuellen >>
Zustand:=0;                       << Zustand nicht erlaubt ist, liefert g >>
repeat                            << den Wert FAIL                     >>
   a := NEXTCHAR(.)
   if a not in DELIMITER then Zustand := g(Zustand,a)
until (Zustand = FAIL) or (a in DELIMITER)
if a in DELIMITER then write („Zahldarstellung o.k.“)
else ...
```

II. Reguläre Sprachen und Mengen

1. Reguläre Ausdrücke
   - 1.1 Mengenoperatoren
   - 1.2 Operations-Hierarchie
2. Endliche Automaten und reguläre Sprachen
   - 2.1 Elementarautomaten
   - 2.2 Zusammenführung von Teilautomaten
3. Anwendungen in der Texterkennung
   - 3.1 Deterministische Automaten und einfache Algorithmen
   - **3.2 Ein einfaches pattern-matching Problem**

Pattern-matching Problem

Es seien gegeben: ein Text $\mathbf{X} = x_1 x_2 \ldots x_n$

eine Zeichenkette $\mathbf{Y} = y_1 y_2 \ldots y_m$

mit $n >> m$.

Ziel: Jedes Vorkommen von Y in X soll festgestellt werden.

Erster Lösungsansatz:

i = Laufindex über den Text **X**: $i = 0 \ldots n\text{-}m$

j = Laufindex über die Zeichenkette **Y**: $j = 1 \ldots m$

found = logische Variable := ***true***, wenn **Y** in **X** gefunden

false, sonst (nicht gefunden)

Pattern-matching Problem

```
i:= 0;
repeat                                    << über gesamten Text X >>
  found := TRUE;
  j:= 1;
  while (j <= m) and found do             << über ges. Zeichenkette Y >>
     if x(i+j) <> y(j) then found = FALSE;
     j:= j +1;
  endwhile
  if found then write (‚Y kommt vor in X an der Position‘, i+1)
  i:= i+1;
until i > n-m
```

Pattern-matching Problem

Anzahl der Such-Schritte: Suchschritte $\approx (n - m)\, q$ mit $1 < q < m$.

q = durchschnittliche Anzahl der Durchläufe der while-Schleife (abhängig von m und der Art der Zeichenkette)

Anmerkung: Überraschenderweise kann man jedoch einen Algorithmus angeben, der die Frage, ob **Y** Teilwort von **X** ist, in

$$\approx n + m \approx n \,; \qquad \text{da} \qquad m << n$$

Schritten beantwortet.

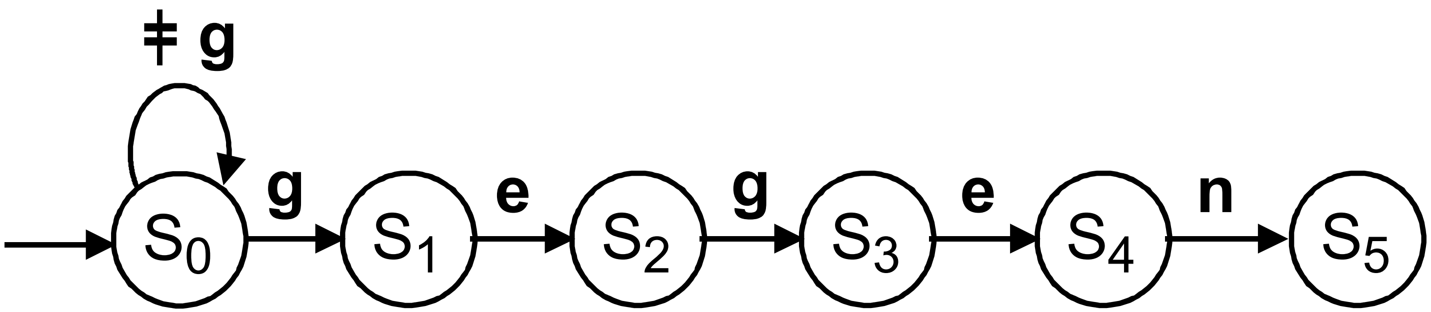
Idee:

Wir konstruieren zur Zeichenkette **Y** einen deterministischen endlichen Automaten, der auf die Eingabe des Textes **X** genau dann in einen Endzustand übergeht, wenn **Y** in **X** entdeckt wurde.

Ein anderer Lösungsweg besteht darin, den deterministischen „Skelettautomaten" zu verwenden, dessen goto-Funktion **g** eindeutig ist.

Übergangsgraph des Skelettautomaten:

Es sei: $X \in \Sigma^* = \{$ gesamte Alphabet $\}$
$Y =$ **y1y2…y5** **=** "gegen" (Schlüsselwort)



$\Rightarrow$ deterministischer Automat ohne Endzustand

Eigenschaften:

- Der Skelettautomat ist im Zustand S_j ($1 \leq j \leq 5$) genau dann, wenn die letzen j gelesenen Buchstaben des Wortes **X** mit den ersten j Buchstaben von **Y** (hier: **Y** = $y_1 y_2 \ldots y_5$ = “gegen“) übereinstimmen.
- Beim Lesen des nächsten Zeichens von **X** sind also zwei Fälle möglich:
  1. Das nächste Zeichen von **X** entspricht dem nächsten Zeichen von Y $\Rightarrow$ Automat geht in den Zustand S_{j+1} über
  2. Oder das nächste Zeichen **a** ist verschieden $\Rightarrow$ Automat geht in den Zustand S_k mit $k \leq j$ zurück, wobei k = größte Zahl derart, dass $y_1 y_2 \ldots y_k$ Endstück von $y_1 y_2 \ldots y_j \mathbf{a}$ ist

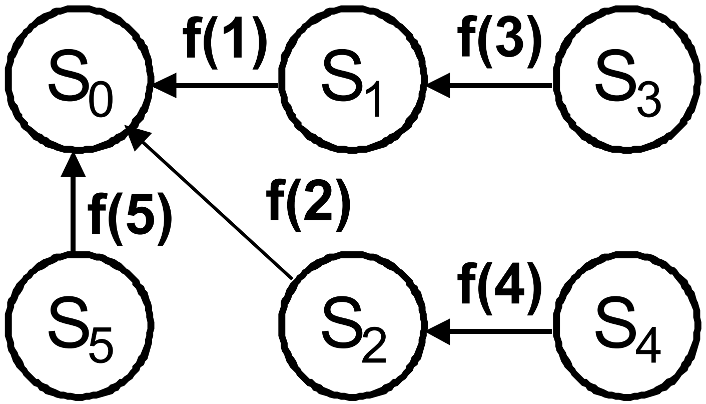
Goto-Funktion g:

Mit der aus dem Zustandsgraphen ablesbaren goto-Funktion **g** erhält man als Texterkennungsprozedur:

Zustand := 0; *<< Anfangszustand >>*
i := 0;
while i < n **do** *<< n = Länge von* ***X*** *>>*
  i := i+1;
  Zustand := g(Zustand, x_i); *<< xi = Eingabezeichen des Textes* ***X*** *>>*
  if Zustand = ERKENNUNGSZUSTAND **then** write('**Y** kommt vor in **X** an der Position', i)
endwhile

Failure-Funktion f:

Um das „Zurückgehen“ im 2. Fall zu bewerkstelligen, definiert man die sog. *failure*-Funktion f, die für unser Beispiel wie folgt dargestellt werden kann:



d. h. die ***failure*-Funktion f** bildet Zustände auf Zustände ab. Der failure-Funktion muss man immer dann folgen, wenn die ***goto*-Funktion g** den Wert FAIL liefert.

Anmerkung zur Failure-Funktion f:

Die ***failure***-Funktion **f: S → S** gibt an, in welchen Zustand man zurückzugehen hat, wenn nach dem Einlesen eines Zeichens x_i die ***goto***-Funktion **g** des Skelettautomaten nicht definiert ist. Dabei wird der Failure-Zustand so gewählt, dass ein möglichst großes Ende des bis dahin eingelesenen **X**-Textes (außer x_i!) wieder mit dem Anfang von **Y** übereinstimmt. Paßt auch da das eingelesene Zeichen x_i nicht, geht man gemäß der failure-Funktion weiter zurück. Spätestens im Anfangszustand endet der Prozeß, weil dort die *goto*-Funktion **g** für alle Zeichen definiert ist.

Algorithmus:

Mit Hilfe der *goto*- und der *failure*-Funktion lautet der Algorithmus:

```
Zustand := 0; i := 0;
while i < n do                          << über alle Zeichen des Textes X >>
  i := i+1;
  while g(Zustand, xi) = FAIL do        << Wenn die goto-Funktion g fehl- >>
     Zustand := f(Zustand);             << schlägt, dann berechne neuen   >>
  endwhile;                             << Zustand mit Hilfe der failure- >>
  Zustand := g(Zustand, xi);            << Funktion f                     >>
  if Zustand = ERKENNUNGSZUSTAND then write(...)
endwhile
```

Algorithmus:

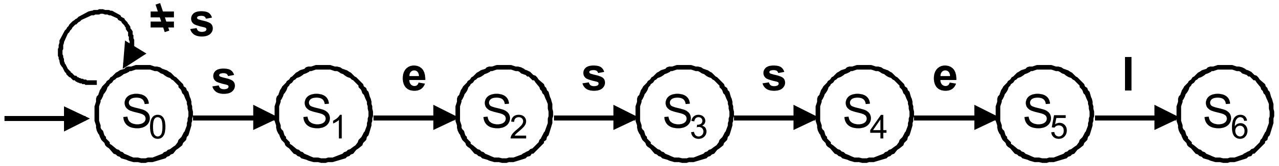
Als Algorithmus für die Funktion **f** ergibt sich also:

```
f(1) := 0;
For s = 2,3,.. ,m do
    t := s-1;
    repeat
        t := f(t);
    until g(t, y_s) ≠ FAIL
    f(s) := g(t, y_s);
enddo;
```

Der Rechenaufwand hierfür hat die Größenordnung m.

Beispiel:

Die Berechnung der **failure-Funktion f(t)** liefert für eine Suche nach dem Wort **„s e s s e l“** gemäß vorstehenden **Algorithmus** folgendes Ergebnis:



$f: S \rightarrow S$

$f(1) = 0$ (per Def.)

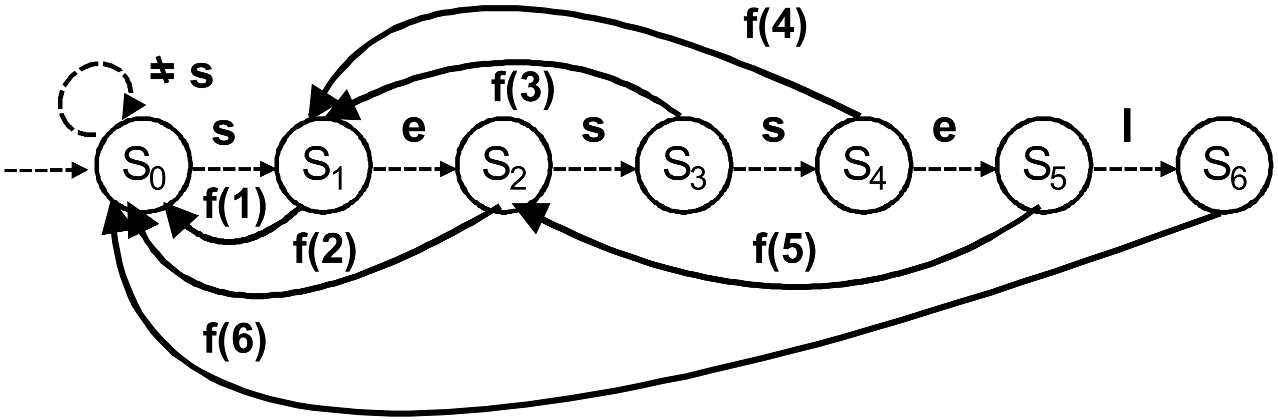
$f(2) = g(0, e) = 0$

$f(3) = g(0, s) = 1$

$f(4) = g(0, s) = 1$

$f(5) = g(1, e) = 2$

$f(6) = g(0, l) = 0$



Automatentheorie und Formale Sprachen

– LV 4110 –

Endliche Automaten

- Kennenlernen der Begriffe: Automat, endlicher Automat, Modell, Zustandsmodell, Graph, Zustandsgraph, Sprache
- Definition der Begriffe: Eingabealphabet, Systemzustände, Start- und Endzustand, Zustandsüberführungsfunktion
- Verwendung von Beschreibungsformen: Zustandsgraph, Funktionstafel, Eingabefolgen, Sprache
- Deterministische endliche Automaten: Komponenten, Eigenschaften, Erweiterung, Sprache, Backus-Naur-Form, Beispiele
- Nicht-deterministische endliche Automaten: Definition, Sprache, Teilmengenkonstruktion und Überführung
- Äquivalenz und Minimierung von Automaten: Minimalautomat, Äquivalenzrelation und -klassen, Algorithmusbeschreibung, Beispiele

I. Endliche Automaten

Was sind Automaten?

Automaten sind selbständig (automatisch) arbeitende Maschinen, die auf gewisse Eingabesignale ihrer Umwelt in einer bestimmten Weise reagieren.

Beispiele:

- Fahrkartenautomat
- Waschmaschine
- Rechenanlage
- Fernsprechanlage
- usw.

Aufbau- und Funktionsbeschreibung:

- Beschreibung kann verbal, z. B. in Umgangssprache erfolgen.
- Für komplexere Prozesse werden abstrakte Modelle benötigt ($\rightarrow$ mehrere Abstraktionsebenen).
- Das einfachste Modell stellt der sog. endliche Automat (**EA**) dar.
- Ein adäquates Mittel zur Funktionsbeschreibung des **EA** sind Zustandsgraphen.
- Daneben existieren noch weitere Beschreibungsformen, wie z. B. Funktionstafeln, Formale Grammatiken etc.

Relationen und Relationseigenschaften:

$A_1, A_2, \ldots, A_n$ seien Mengen

$x_1, x_2, \ldots, x_n$ seien Elemente mit $x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n$

$\Rightarrow$ Implikation (daraus folgt)

$\Leftrightarrow$ Äquivalenz (genau dann wenn)

$\in$ Element von

$\Rightarrow$ $(x_1, x_2, \ldots, x_n)$ ist ein geordnetes Tupel von Elementen über $A_1, A_2, \ldots, A_n$

Kartesisches Produkt:

$$A_1 \times A_2 \times \ldots \times A_n := \{ (x_1, x_2, \ldots, x_n) \mid x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n \}$$

n-stellige Relation:

Satz: Jede Teilmenge **R** der Mengen A_1 x A_2 x ... x A_n heißt eine n-stellige Relation über den Mengen A_1, A_2, ..., A_n.

kurz: $\mathbf{R} \subseteq A_1 \times A_2 \times \ldots \times A_n$

Binäre Relation:

$\Rightarrow$ n = 2 (auch: 2-stellige Relation)

Anmerkung:

Im folgenden sind vor allem binäre Relationen der Art $\mathbf{R} \subseteq \mathbf{M} \times \mathbf{M}$ von Interesse $\Rightarrow$ **R** heißt dann „Relation auf **M**" oder „Relation in **M**".

Anmerkung (Fortsetzung):

$\Leftrightarrow (x, y) \in \mathbf{R} \quad \Leftrightarrow$ sog. Infixnotation für $x\,\mathbf{R}\,y$

heißt: „x steht in der Relation y“

Definition (symmetrisch, reflexiv, transitiv):

Eine binäre Relation **R** auf einer Menge **M** heißt:

symmetrisch	$\Leftrightarrow$	$\forall\, x, y \in \mathbf{M}$:	$(x\,\mathbf{R}\,y \;\Rightarrow\; y\,\mathbf{R}\,x)$
reflexiv	$\Leftrightarrow$	$\forall\, x \in \mathbf{M}$:	$x\,\mathbf{R}\,x$
transitiv	$\Leftrightarrow$	$\forall\, x, y, z \in \mathbf{M}$:	$(x\,\mathbf{R}\,y \wedge y\,\mathbf{R}\,z \;\Rightarrow\; x\,\mathbf{R}\,z)$

Satz: Eine Relation R auf einer Menge M heißt **Äquivalenzrelation**, wenn sie symmetrisch, reflexiv und transitiv ist.

Beispiel:

Wir betrachten die Teilbarkeitsrelation „|“ über den ganzen Zahlen **Z**. Diese Relation ist für $\forall$ m, n $\in$ **Z** definiert durch:

$$(m \mid n \iff \exists k \in \mathbf{Z} : n = k \cdot m)$$

$m \mid n$ heißt: m teilt n.

Zahlenbeispiel:

$7 \mid 21$, denn: $21 = 3 \cdot 7$

Satz: **Die Teilbarkeitsrelation „|“ ist zwar reflexiv und transitiv, aber nicht symmetrisch $\Rightarrow$ ist demnach keine Äquivalenzrelation!**

I. Endliche Automaten

Ein Zustandsgraph besteht aus Knoten und Kanten:

- S_i = Knote S_i kennzeichnet den Systemzustand S_i
- $\xrightarrow{e}$ = Kante e drückt den Zustandsübergang unter Einwirkung der Eingabe e aus.

Kennzeichnung des Start- und Endzustandes:

- $\rightarrow S_0$ = Startzustand S_0
- S_f = Endzustand S_f

An einem Graphen lässt sich sehr leicht nachverfolgen, welche Systemzustände bei der Verarbeitung der Eingabezeichen angenommen werden.

Automat als informationsverarbeitende Maschine:

Eingabe: dient zur Versorgung des Automaten mit Eingabedaten e_i aus den sog. Eingabealphabet $\boldsymbol{\Sigma}$ ($e_i \in \boldsymbol{\Sigma}$)

Interne Zustände: werden als Reaktion auf die Eingabe durchlaufen ($\mathbf{S} = \{s_0, s_1, \ldots, s_n\}$)

Ausgabe: sind die vom Automaten i.d.R. produzierten Ausgabedaten ($a_i \in \mathbf{A}$) mit $\mathbf{A}$ = Ausgabealphabet

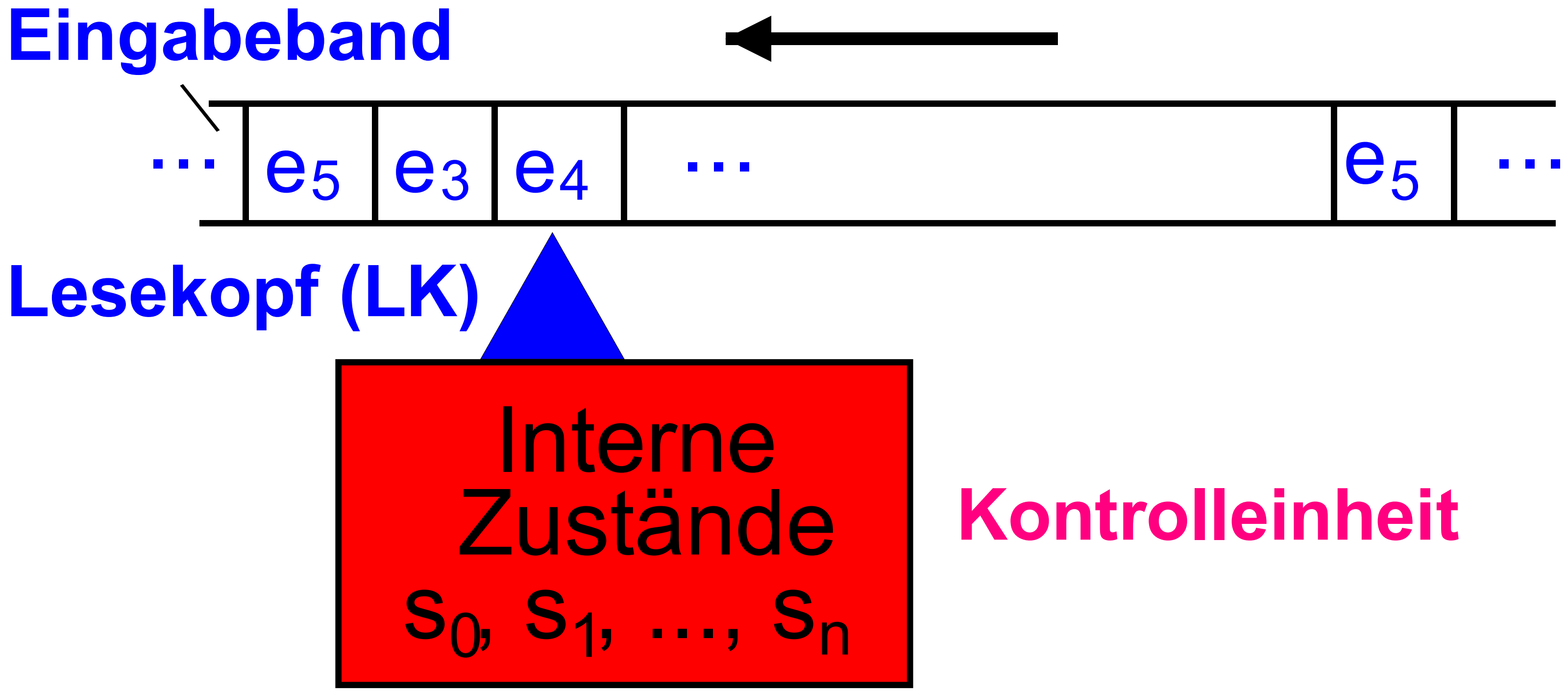
Charakteristisch für einen Automaten ist, dass der Folgezustand neben den Eingabezeichen auch vom momentanen inneren Systemzustand abhängig ist.

Die Zustands-Überführungsfunktion δ muss daher in folgender Form angeschrieben werden:

$$\delta : S \times \Sigma \rightarrow S$$

x := kartesische Produkt

Anschauliche Vorstellung eines endlichen Automaten:



Arbeitsweise eines endlichen Automaten:

BEGIN

Bringe EA *in Zustand* s_0; (* Anfangszustand *)

Setze LK *über linkes Zeichen des Eingabewortes*;

WHILE *Zeichen unter* LK *vorhanden* **DO**

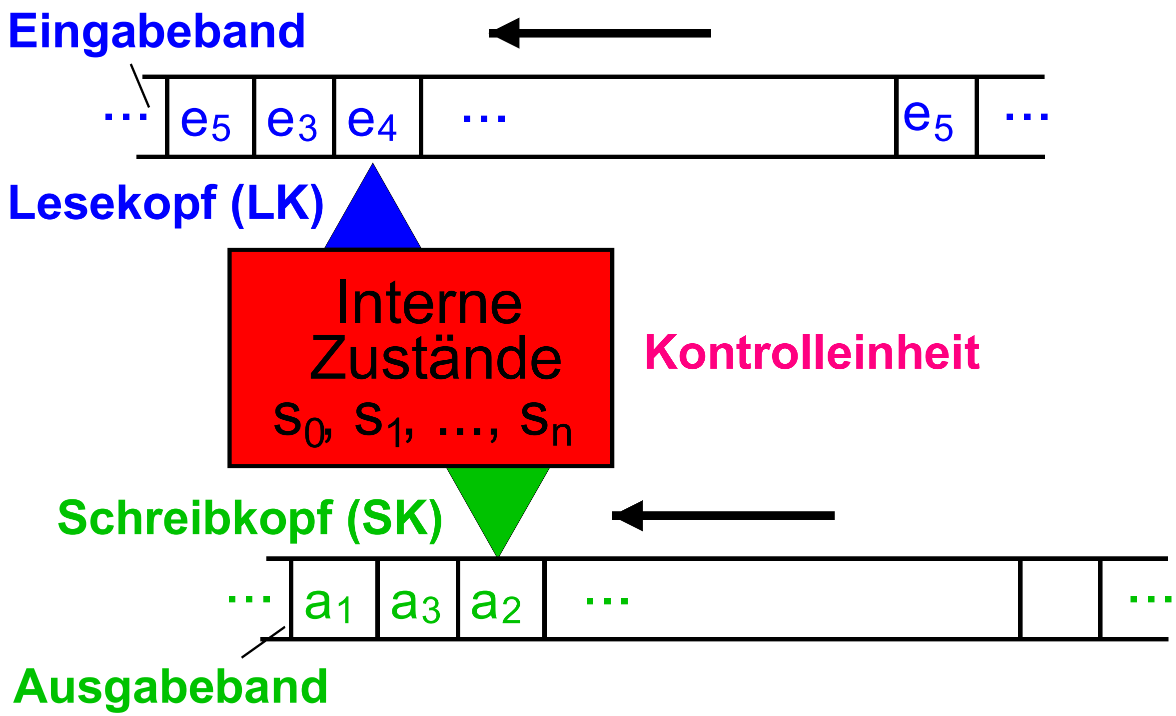
Gehe in Folgezustand gemäß $\boldsymbol{\delta : S \times \Sigma \rightarrow S}$;

Bewege LK *um ein Feld nach rechts*;

END (* WHILE *)

END

Vorstellung eines endlichen Automaten mit Ausgabe:



Arbeitsweise eines endlichen Automaten mit Ausgabe:

BEGIN

Bringe EA *in Zustand* s_0; (* Anfangszustand *)

Setze LK *über linkes Zeichen des Eingabewortes*;

WHILE *Zeichen unter* LK *vorhanden* **DO**

Schreibe das gewünschte Ausgabezeichen $a_i \in \mathbf{A}$

Gehe in Folgezustand gemäß $\boldsymbol{\delta : S \times \Sigma \rightarrow S}$;

Bewege LK *um ein Feld nach rechts*;

END (* WHILE *)

END

Definition:

Die Menge aller Eingabefolgen, die von einem Automaten **A** akzeptiert werden, nennt man die Sprache **T** des Automaten.

kurz:

$T(A)$

Beispiel:

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

$\Rightarrow T(A) = abcabcabc \ldots$

$= (abc)^*$

Beispiel

<u>Spezifikation:</u>

- Einwurfsmöglichkeiten sind **1** €-, **2** €- und **5** €-Münzen.
- In jeder Situation kann der Automat durch Drücken des Rückgabeknopfes (**R**) in den Anfangszustand versetzt werden. (Bereits eingeworfene Münzen werden dann zurückgegeben.)
- Falls **5** € in den Automaten eingeworfen wurden, wird durch Ziehen einer Schublade (**Z**) die Ware zur Entnahme freigegeben.
- Mit der Entnahme (**E**) der Ware wird der Anfangszustand wieder hergestellt.
- Der geleistete Münzeinwurf wird durch die Anzeige (**A**) angezeigt.

<u>Funktionsweise:</u>

Die Funktionsweise des Warenautomaten kann mit Hilfe der **Eingabezeichen**, einer Reihe von **Zuständen** (states) und der **Ausgabe** beschrieben werden.

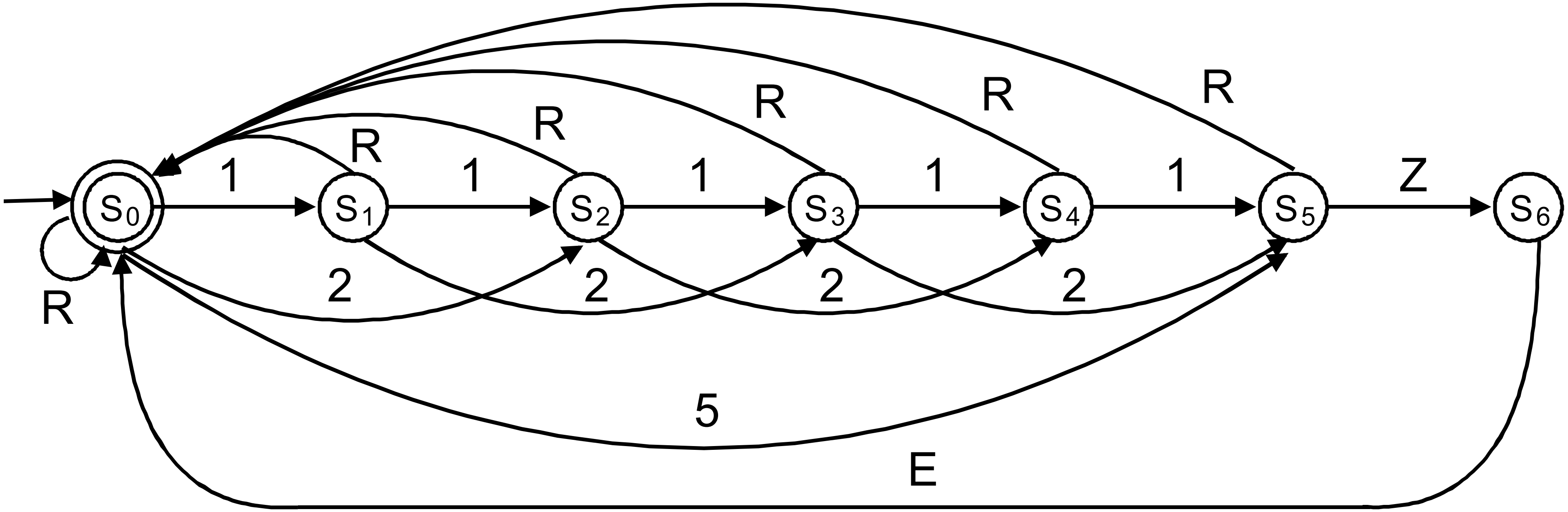
- Eingabezeichen $\mathbf{\Sigma} = \{ 1, 2, 5, R, Z, E \}$
- Zustände $\mathbf{S} = \{ S_0, S_1, \dots, S_6 \}$
- Ausgabe bzw. Endzustand (<u>hier</u>: identisch mit Anfangszustand)

Die Zustandsübergänge bzw. Zustandswechsel lassen sich durch die sogenannte **Überführungsfunktion δ** zum Ausdruck bringen.

- Überführungsfunktion $\boldsymbol{\delta} : \mathbf{S} \times \mathbf{\Sigma} \rightarrow \mathbf{S}$

Beispiel

<u>Graphische Darstellung:</u>



- Im Zustand S_i ($i = 0$ bis 5) genau i € eingeworfen
- Im Zustand S_6 ist Warenentnahme möglich

Tabellarische Darstellung δ:

$s \downarrow$ $\Sigma \rightarrow$	**1**	**2**	**5**	**R**	**Z**	**E**
S_0	S_1	S_2	S_5	S_0		
S_1	S_2	S_3	–	S_0	–	–
S_2	S_3	S_4	–	S_0	–	–
S_3	S_4	S_5	–	S_0	–	–
S_4	S_5	–	–	S_0	–	–
S_5	–	–	–	S_0	S_6	–
S_6	–	–	–	–	–	S_0

Sprache des Automaten:

- Eine besondere Rolle spielt der Zustand S_6, da in diesem Zustand die gewünschte Warenentnahme möglich ist.
- Eingabefolgen, die in den Zustand S_6 führen, sind z. B. **122Z**, **5Z**, aber auch **1R1R...5Z**.
- Es gibt unendlich viele solcher Eingabefolgen.

Alle Eingabefolgen, die in den Zustand S_6 führen sind Worte, gebildet auch Zeichen der Zeichenmenge $\mathbf{\Sigma} = \{ 1, 2, 5, R, Z, E \}$. Sie stellen eine **formale Sprache** über $\mathbf{\Sigma}$ dar, die durch besondere Bildungsregeln gekennzeichnet ist.

Deterministische endliche Automaten

Das 5-Tupel $A = (\mathbf{S}, s_0, \mathbf{F}, \Sigma, \delta)$ bezeichnet einen deterministischen endlichen Automaten, wenn für die einzelnen Komponenten gilt:

S *endliche* Menge der möglichen Zustände des Automaten (Bezeichnung der Elemente mit s, s', s_i usw.)

s_0 Anfangszustand des Automaten, $s_0 \in \mathbf{S}$

F Menge der Endzustände des Automaten, $\mathbf{F} \subseteq \mathbf{S}$

Σ *endliche* Menge der Eingabezeichen (Bezeichnung der Elemente mit a, b, a_i usw.)

δ (*deterministische*) Zustandsüberführungsfunktion, die gewissen Paaren (s,a) des kartesischen Produkts $\mathbf{S} \times \Sigma$ einen Folgezustand s' aus **S** zuordnet. Man schreibt auch $\delta\text{:}\ \mathbf{S} \times \Sigma \rightarrow \mathbf{S}$ und $\delta(s,a) = s'$.

Anmerkung:

Ob die Funktion δ vollständig (→ **totale Funktion)** ist, d.h. für alle Paare (s,a) ein Nachfolgezustand vorhanden ist, spielt manchmal eine Rolle. Man kann dies immer durch Einführung eines ggf. noch nicht vorhandenen Zustands „Fehler“ erreichen, der als Folgezustand für alle Paare (s,a) auftritt, für die δ(s,a) nicht definiert ist (ansonsten: → **partiell definierte Übergangsfunktion)**.

I. Endliche Automaten

1. Sprachgebrauch und Motivation
   - 1.1 Automaten und Zustandsüberführungsfunktion
   - 1.2 Sprache eines Automaten
2. Deterministische endliche Automaten
   - **2.1 Erweiterung der Überführungsfunktion**
   - 2.2 Sprache eines deterministischen Automaten
3. Nicht-deterministische endliche Automaten
   - 3.1 Sprache eines nicht-deterministischen Automaten
   - 3.2 Teilmengenkonstruktion
4. Äquivalenz und Minimierung von Automaten
   - 4.1 Äquivalente und reduzierte Automaten
   - 4.2 Bildung von Äquivalenzklassen

Erweiterung der Übergangsfunktion δ auf Worte aus Σ^*:

Sei Σ^* die Menge aller endlichen Folgen bzw. *Worte* $w = a_1a_2...a_n$, mit $a_i \in \Sigma$ für alle i. Durch iterative Anwendung der (vollständigen) Übergangsfunktion – ausgehend von einem beliebigen $s \in \mathbf{S}$ – erhält man eine Folge von Zuständen s_i mit $\delta(s,a_1) = s_1$, $\delta(s_1,a_2) = s_2$ ... $\delta(s_{n-1},a_n) = s_n$

Interpretation:

Wird im Zustand s das Symbol a_1 vorgelegt, so geht der Automat in den Zustand $\delta(s,a_1) = s_1$ und liest das nächste Symbol a_2, und geht danach in den nächsten Zustand $\delta(s_1,a_2) = s_2$ usw.

Erweiterung der Übergangsfunktion δ auf Worte aus Σ^*:

Wir definieren $\delta(s,w) := s_n$, dann gilt für die erweiterte Funktion δ:

δ: **S** x Σ^* -> **S** und $\delta(s,w) = \delta(s,a_1 w_{-1}) = \delta(\delta(s,a_1),w_{-1})$, wobei w_{-1} den Rest des Wortes w nach Entfernung von a_1 bezeichnet.

Vereinbarungsgemäß sei auch das *leere Wort* ε in Σ^* und wir setzen $\delta(s, \varepsilon) = s$ für alle s.

Interpretation:

Lesen der leere Eingabefolge erzeugt keine Zustandsänderung.

I. Endliche Automaten

1. Sprachgebrauch und Motivation
   - 1.1 Automaten und Zustandsüberführungsfunktion
   - 1.2 Sprache eines Automaten
2. Deterministische endliche Automaten
   - 2.1 Erweiterung der Überführungsfunktion
   - **2.2 Sprache eines deterministischen Automaten**
3. Nicht-deterministische endliche Automaten
   - 3.1 Sprache eines nicht-deterministischen Automaten
   - 3.2 Teilmengenkonstruktion
4. Äquivalenz und Minimierung von Automaten
   - 4.1 Äquivalente und reduzierte Automaten
   - 4.2 Bildung von Äquivalenzklassen

Sprache T(A) eines deterministischen Automaten:

Man sagt ein Wort $w \in \Sigma^*$ werde vom Automaten A *akzeptiert*, wenn $\delta(s_0,w) \in \mathbf{F}$ ist, d.h. wenn die beschriebene iterative Anwendung der Übergangsfunktion ausgehend vom Anfangszustand s_0 auf einen Endzustand $s_f \in \mathbf{F}$ führt.

Definition:

Unter der Sprache T(A) eines deterministischen Automaten versteht man die Menge aller vom Automaten akzeptierten Worte, d.h. :

$T(A) = \{ w \in \Sigma^* \mid \delta(s_0,w) \in \mathbf{F} \}$. Es ist ferner: $\varepsilon \in T(A) \Leftrightarrow s_0 \in \mathbf{F}$.

Bedeutung der Symbole der BNF:

< ... >	spitze Klammern grenzen sogenannte **syntaktische Variable** ein
::=	ist zu lesen als „ist“
|	ist zu lesen als „oder“
{ ... }	geschweifte Klammern zeigen die **Wiederholung** des Klammerinhalts an; Anmerkung: Inhalt **...** kann auch weggelassen werden.

Die BNF-Regeln für eine korrekte Zahlendarstellung:

<number> ::= <unsigned number> | <sign> <unsigned number>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit> {digit}

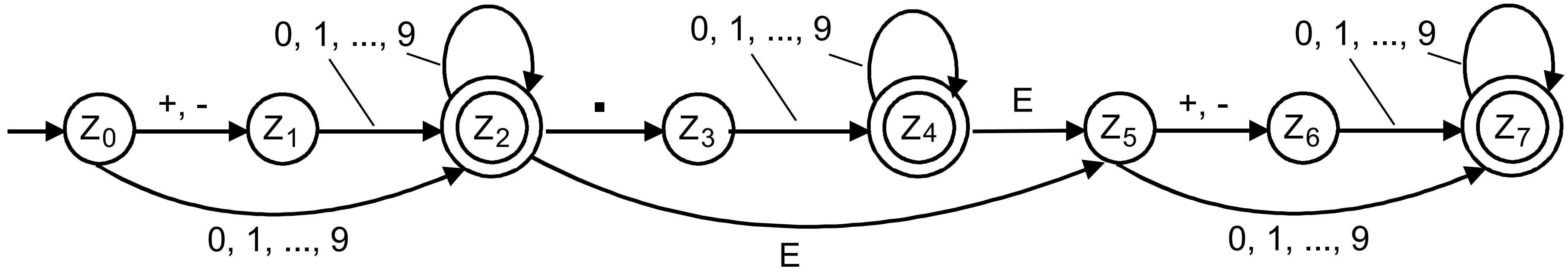
<unsigned real> ::= <unsigned integer> . <digit> {digit} |
<unsigned integer> . <digit> {digit} **E**
<scale factor> | <unsigned integer> **E**
<scale factor>

<scale factor> ::= <unsigned integer> | <sign> <unsigned integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<sign> ::= + | -

Zustandsgraph des Automaten:



Komponenten des Automaten:

Dabei bestehen die einzelnen Komponenten **S**, **F** und Σ aus:

$\mathbf{S}$ = { s0,s1,s2,s3,s4,s5,s6,s7},

$\mathbf{F}$ = { s2,s4,s7} (Doppelkreise im Graphen!) und

Σ = { 0,1,2,3,4,5,6,7,8,9,+,-,.,E }

Funktionstafel des Automaten:

Andererseits kann der Automat auch durch eine Tabelle für die Werte der Übergangsfunktion δ repräsentiert werden:

$S \downarrow$ $\Sigma \rightarrow$	0, … ,9	+, -	.	E
S_0	$\mathbf{S_2}$	S_1	–	–
S_1	$\mathbf{S_2}$	–	–	–
S_2	$\mathbf{S_2}$	–	S_3	S_5
S_3	$\mathbf{S_4}$	–	–	–
S_4	$\mathbf{S_4}$	–	–	S_5
S_5	$\mathbf{S_7}$	S_6	–	–
S_6	$\mathbf{S_7}$	–	–	–
S_7	$\mathbf{S_7}$	–	–	–

Striche bedeuten, dass δ für die entsprechende (s,a)-Kombination nicht definiert ist, bzw. als “Fehlerzustand“ aufzufassen ist, wenn man die Funktion vervollständigt.

I. Endliche Automaten

1. Sprachgebrauch und Motivation
   - 1.1 Automaten und Zustandsüberführungsfunktion
   - 1.2 Sprache eines Automaten
2. Deterministische endliche Automaten
   - 2.1 Erweiterung der Überführungsfunktion
   - 2.2 Sprache eines deterministischen Automaten
3. **Nicht-deterministische endliche Automaten**
   - 3.1 Sprache eines nicht-deterministischen Automaten
   - 3.2 Teilmengenkonstruktion
4. Äquivalenz und Minimierung von Automaten
   - 4.1 Äquivalente und reduzierte Automaten
   - 4.2 Bildung von Äquivalenzklassen

Funktionserweiterung des deterministischen EA:

Bisher:

Gemäß der bisherigen Definition der Zustandsüberführungsfunktion δ erhielten wir höchstens einen Folgezustand $\rightarrow$ **eindeutig**.

Jetzt:

Lassen wir eine Menge **T** möglicher Folgezustände zu $\rightarrow$ Zustandsüberführungsrelation $\rightarrow$ **nicht eindeutig**.

Interpretation:

Ein Element (s, **a**, s‘) aus T ist also zu interpretieren als: „Im Zustand s ist bei Eingabe von **a** ein Übergang in den Zustand s‘ möglich“

Damit kommen wir zum

Nicht-deterministischen endlichen Automaten

kurz: **NFA**

Folge:

Beim Zustandsgraphen kann dasselbe Eingabesymbol an mehreren Kanten stehen, die aus einem Zustand herausführen.

Somit gilt es den nicht-deterministischen endlichen Automaten bzgl. dieser Erweiterung zu definieren!

Definition:

A = ($\mathbf{S}$, $\mathbf{S}_0$, $\mathbf{F}$, Σ, δ) bezeichnet einen nicht-deterministischen endlichen Automaten, wenn **S**, **F** und Σ die gleiche Bedeutung wie bei deterministischen Automaten haben und für die anderen Komponenten gilt:

$\mathbf{S}_0$ Menge der Anfangszustände des Automaten, von denen es mehr als einen geben kann.

δ (nicht deterministische) Übergangs-Relation, die gewissen Paaren (s,a) des kartesischen Produkts $\mathbf{S} \times \Sigma$ i.a. mehrere mögliche Folgezustände, die man in einer Menge $\mathbf{T} \subseteq \mathbf{S}$ zusammenfassen kann, zuordnet. Man schreibt auch $\delta(s,a) = \mathbf{T}$ und δ: $\mathbf{S} \times \Sigma \rightarrow \mathbf{P(S)}$, wobei **P(S)** die **Potenzmenge** von **S** ist.

I. Endliche Automaten

Sprache:

Wir betrachten wieder ein beliebiges Wort $w = a_1a_2...a_n$ aus Σ^*.
Ausgehend von S_0 erzeugen wir iterativ die Mengen:

$\delta(S_0, a_1) = T_1$;

$\delta(T_1, a_2) = T_2$;

...

$\delta(T_{n-1}, a_n) = T_n$

und erhalten mit T_n die Menge aller möglichen Folgezustände, die mit der Übergangsrelation von den Anfangszuständen ausgehend durch Eingabe des Wortes w erreicht werden können.

Definition:

Ist unter den Zuständen $\mathbf{T_n}$ ein Endzustand, dann soll definitionsgemäß das Wort **w** akzeptiert werden.

Ein Wort $\mathbf{w} \in \Sigma^*$ wird von A akzeptiert, wenn

$$\delta(S_0, \mathbf{w}) \cap \mathbf{F} \neq 0 \text{ (leere Menge)}$$

ist.

Als Sprache des nicht-deterministischen Automaten erhalten wir:

$$T(A) = \{ \mathbf{w} \in \Sigma^* \mid \delta(S_0, \mathbf{w}) \cap \mathbf{F} \neq 0 \}$$

Für den Fall des leeren Wortes ε wird hier implizit definiert:

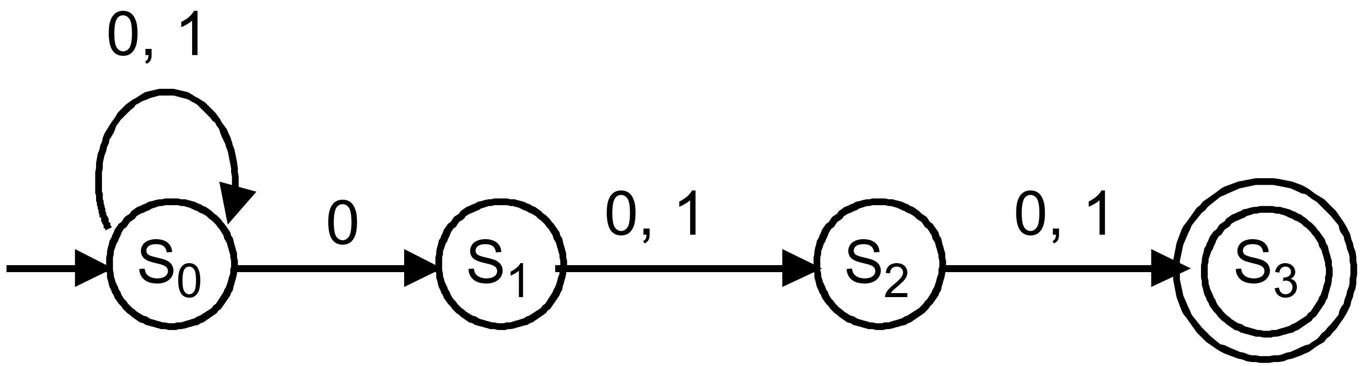
$$\varepsilon \in T(A) \iff S_0 \cap \mathbf{F} \neq 0$$

Beispiel:

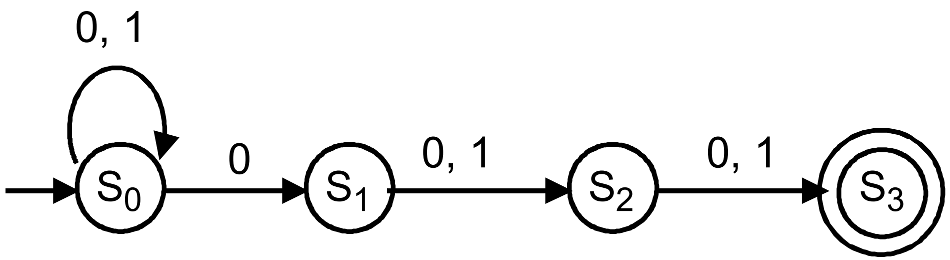
Wir betrachten die Menge aller Worte über dem Alphabet {0, 1}, die als **drittletztes** Symbol eine Null haben, d. h. alle Worte der Gestalt:

$$T(A) = \{ \mathbf{w} \in \Sigma^* \mid w = u0v \text{ mit } u = \{0,1\}^* \text{ und } v = \{00, 01, 10, 11\} \}$$

Zustandsgraph:



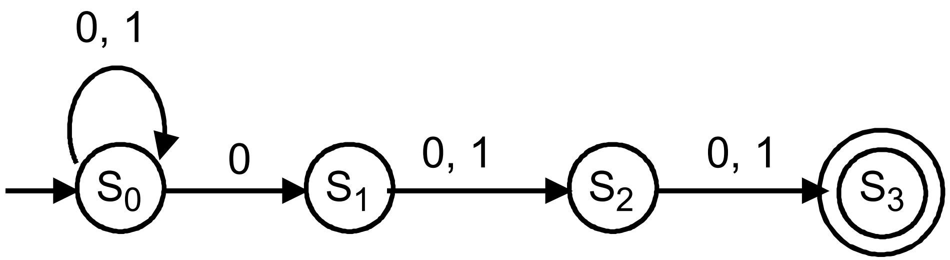
Nicht-Determinismus:



Der Nicht-Determinismus besteht darin, dass vom Zustand S_0 zwei Pfeile mit dem Eingabezeichen 0 ausgehen.

Satz:
Man kann auch einen deterministischen Automaten angeben, der dieselbe Sprache akzeptiert, aber wesentlich mehr Zustände besitzt. Nicht-deterministischer Automat mit $k+1$ Zustände ➔ deterministischer Automat mit 2^k Zustände.

Komponenten des Automaten:



S = $\{S_0, S_1, S_2, S_3\}$
S0 = $\{S_0\}$
F = $\{S_3\}$
Σ = $\{0, 1\}$

Untersuchung der Worte: $w_1 = 10\mathbf{1}01 \notin T(A)$ und $w_2 = 11\mathbf{0}01 \in T(A)$

I. Endliche Automaten

<u>Satz</u>:

Zu jedem nicht deterministischen endlichen Automaten gibt es einen deterministischen, der die gleiche Sprache akzeptiert, d. h.

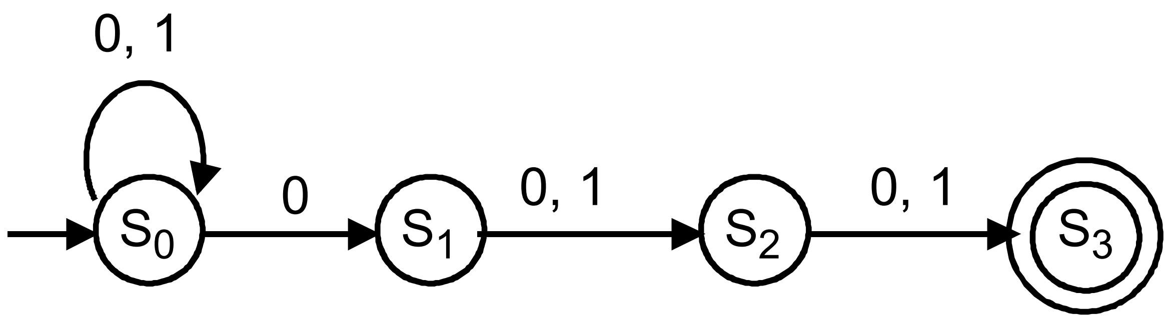
$$T(A) = T(A')$$

<u>Lösungsverfahren</u>:

Teilmengenkonstruktion!

Der Beweis zeigt, daß bei der Konstruktion des deterministischen Automaten u.U. mit großen Zustandsmengen gerechnet werden muß, da eine Menge M mit k Elementen 2^k Untermengen besitzt ($\rightarrow$ sog. Potenzmenge P(M)).

Nicht-deterministischer endlicher Automat:

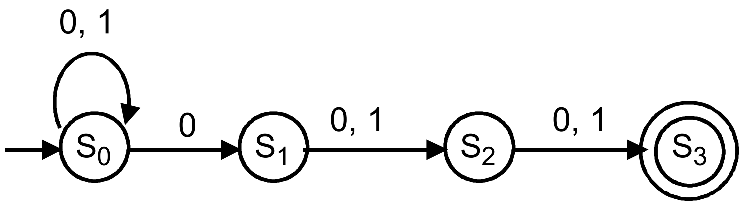


	0	**1**
S_0	S_0, S_1	S_0
S_1	S_2	S_2
S_2	S_3	S_3

Konstruktion von Teilmengen:

Man erhält die Teilmengen durch sukzessives Nachverfolgen aller möglichen Pfade im ursprünglichen Graphen, beginnend mit der Menge der Anfangszustände.

Nicht-deterministischer endlicher Automat:

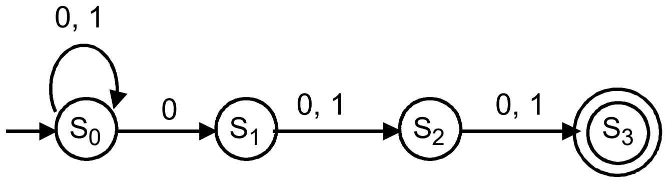


	0	**1**
s_0	s_0, s_1	s_0
s_1	s_2	s_2
s_2	s_3	s_3

Konstruktion von Teilmengen:

Dazu trägt man die Menge der Anfangszustandsmenge in eine Tabelle ein und berechnet hierfür die Zustandsmengen bei Eingabe von **0** und **1**. Neu auftretende Mengen werden unter der Rubrik Zustandsmenge eingetragen und deren Tabelleneinträge für **0** und **1** ermittelt. Dies wird solange fortgesetzt, bis keine neuen Zustandsmengen mehr erscheinen.

Nicht-deterministischer endlicher Automat:

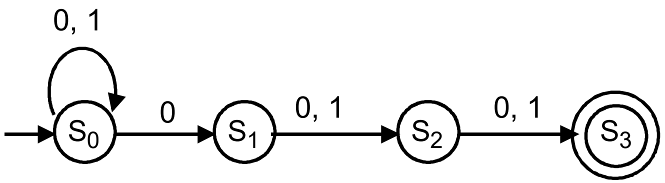


	0	1
s_0	s_0, s_1	s_0
s_1	s_2	s_2
s_2	s_3	s_3

Deterministischer Automat:

	0	1
{s0}	{s0,s1}	{s0}
{s0,s1}	{s0,s1,s2}	{s0,s2}
{s0,s1,s2}	{s0,s1,s2,s3}	{s0,s2,s3}
{s0,s2}	{s0,s1,s3}	{s0,s3}
{s0,s1,s2,s3}	{s0,s1,s2,s3}	{s0,s2,s3}
{s0,s2,s3}	{s0,s1,s3}	{s0,s3}
{s0,s1,s3}	{s0,s1,s2}	{s0,s2}
{s0,s3}	{s0,s1}	{s0}

Nicht-deterministischer endlicher Automat:



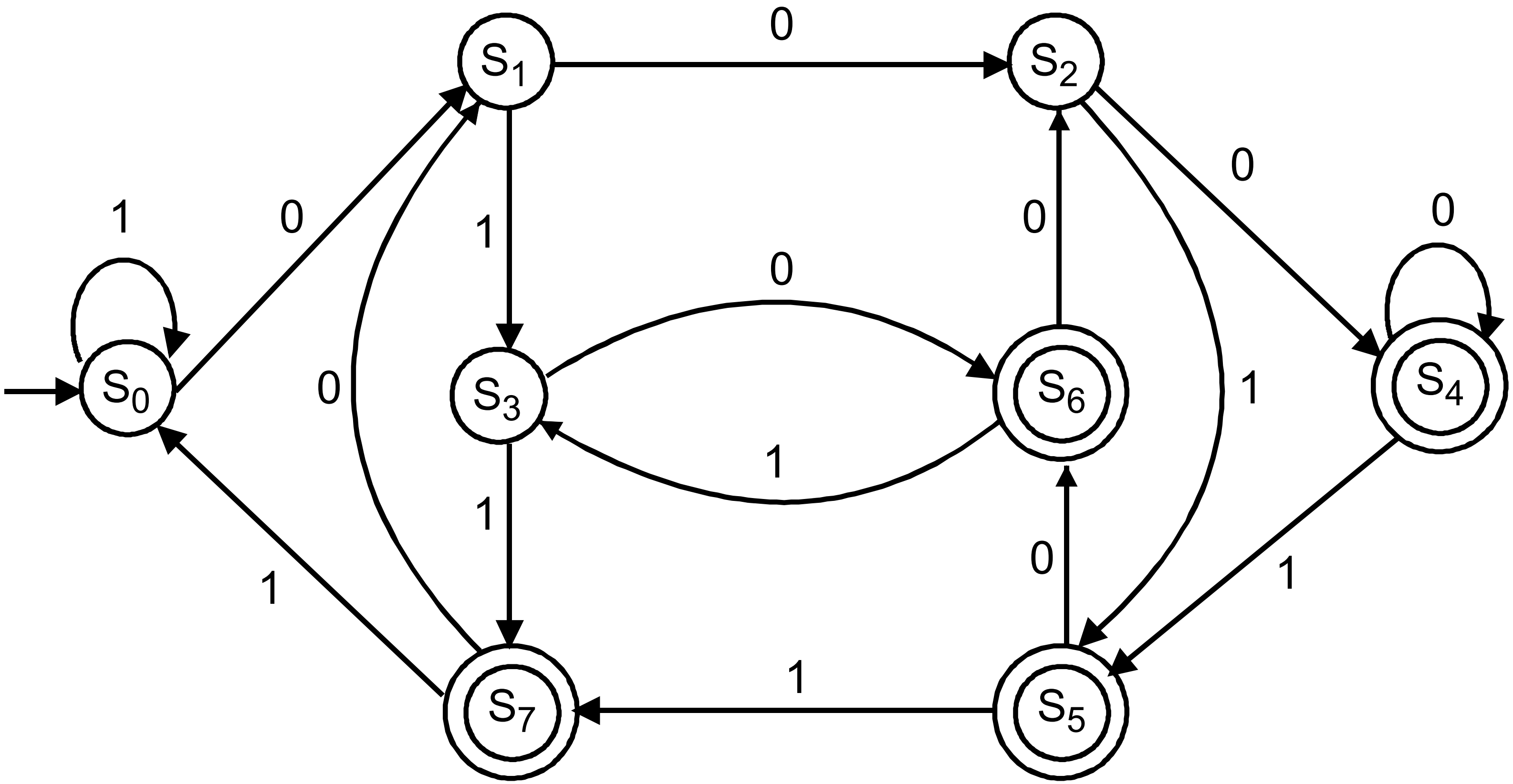
	0	1
s_0	s_0, s_1	s_0
s_1	s_2	s_2
s_2	s_3	s_3

Deterministischer Automat:

	0	1
(0) {s0}	(1) {s0,s1}	(0) {s0}
(1) {s0,s1}	(2) {s0,s1,s2}	(3) {s0,s2}
(2) {s0,s1,s2}	(4) {s0,s1,s2,s3}	(5) {s0,s2,s3}
(3) {s0,s2}	(6) {s0,s1,s3}	(7) {s0,s3}
(4) {s0,s1,s2,s3}	(4) {s0,s1,s2,s3}	(5) {s0,s2,s3}
(5) {s0,s2,s3}	(6) {s0,s1,s3}	(7) {s0,s3}
(6) {s0,s1,s3}	(2) {s0,s1,s2}	(3) {s0,s2}
(7) {s0,s3}	(1) {s0,s1}	(0) {s0}

Teilmengenkonstruktion

Deterministischer endlicher Automat:



<u>Ziel</u>:

Reduktion der Anzahl der Zustände eines Automaten ohne dabei die akzeptierte Sprache des Automaten zu verändern.

➔ Zusammenfassung sog. äquivalenter Zustände auf einen **Minimalautomaten**

<u>Definition</u>:

Zwei Zustände s und s´ eines endlichen deterministischen Automaten heißen *äquivalent* (s =~ s´), wenn die Menge der Worte, die in einen Endzustand führen, für beide identisch ist , d.h. $\delta(s,w) \in F \Leftrightarrow \delta(s',w) \in F$ für alle $w \in \Sigma^*$ gilt. Der Automat heißt *reduziert*, wenn keine zueinander äquivalenten Zustände existieren und jeder Zustand von s_0 aus erreichbar ist.

<u>**Eigenschaften**</u>:

Die Äquivalenzrelation hat die üblichen Eigenschaften der

- Reflexivität,
- Symmetrie und
- Transitivität.

Alle zueinander äquivalenten Zustände können in einer *Äquivalenz-klasse* [s] = { s´ | s´ $=_\sim$ s }, die durch <u>einen</u> Vertreter s repräsentiert werden kann, zusammengefaßt werden.

<u>Satz</u>:

Zu jedem deterministischen endlichen Automaten gibt es einen reduzierten, der die gleiche Sprache akzeptiert.

Beweis:

Beweis durch Angabe eines Algorithmus zur schrittweisen Bildung der Äquivalenzklassen [s] des Automaten A = (**S**, s_0, **F**, Σ, δ) und der Definition des *reduzierten Automaten* A´ = (**S**´, $s´_0$, **F**´, Σ, δ´) mit:

(1) **S**´ = { [s] | s ∈ **S**};

(2) s´0 = s0;

(3) **F**´ = { [s] | s ∈ **F**} und

(4) δ´([s],a) = [δ(s,a)] für alle [s] ∈ S´ und a ∈ Σ

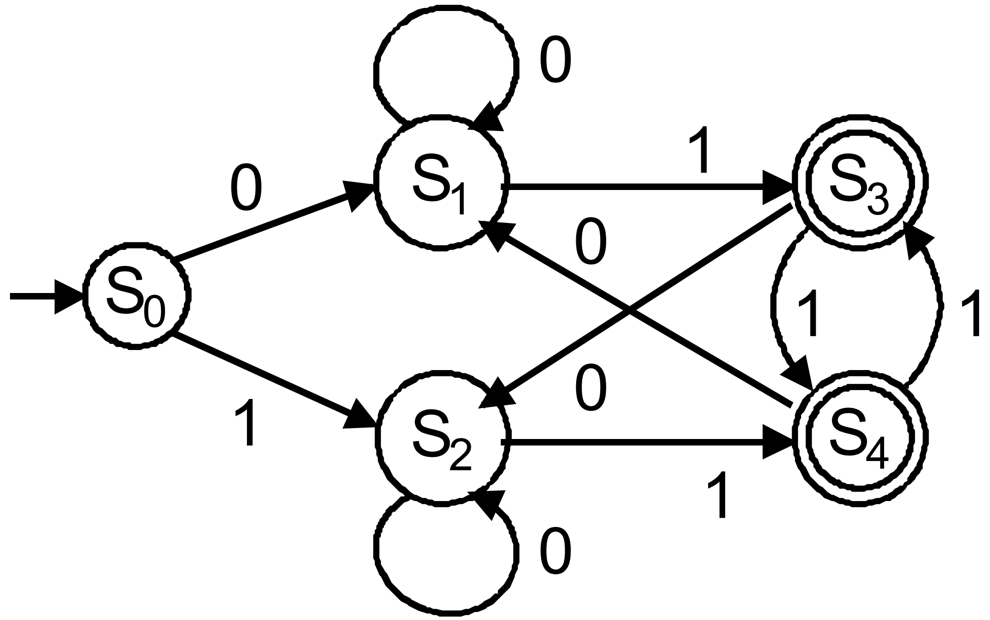
<u>Algorithmus</u>:

Der *Algorithmus zur Bildung der Äquivalenzklassen von A* läuft darauf hinaus die *gröbste Partition* der Menge der Zustände **S** zu finden, bei der die Menge der Folgezustände einer Partitionsmenge für jedes $a \in \Sigma$ wieder eine Partitionsmenge bilden.
Diese Partition stellt dann die Zerlegung in Äquivalenzklassen dar.

Algorithmusbeschreibung:

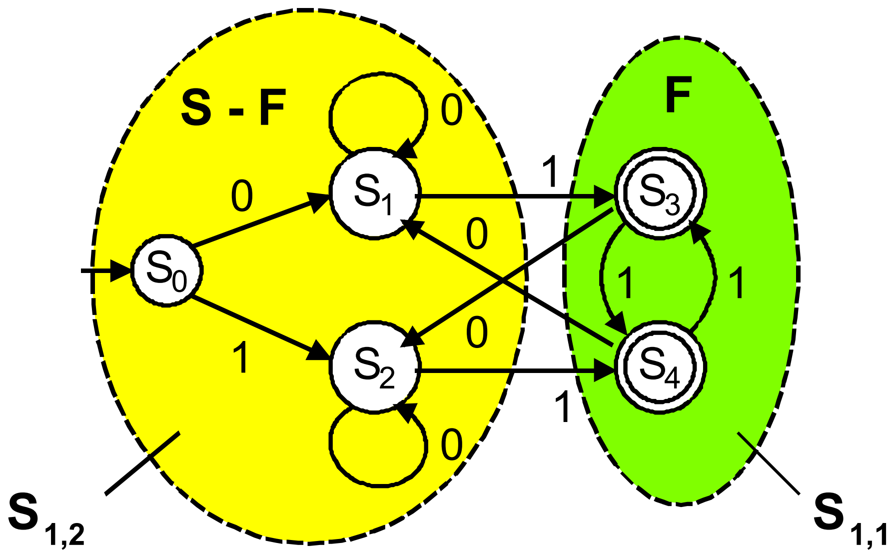
1. Teile die Menge der Zustände **S** in die disjunkten Teilmengen **F** und (**S**-**F**) auf, die offensichtlich nicht beide Elemente der gleichen Äquivalenzklasse enthalten können, da $\delta(s, \varepsilon) \in$ **F** für alle $s \in$ **F** gilt, aber nicht für $s \in$ (**S**-**F**).
2. Die Partitionsmengen werden nun für jedes **a** $\in \Sigma$ untersucht. Liegen für ein bestimmtes **a** die Bilder δ(s, **a**) der Zustände einer Partitionsmenge nicht alle in der gleichen Bildmenge, dann muss die Urbildmenge erneut aufgeteilt werden, d. h. die Partition wird verfeinert und Schritt 2 wiederholt.
3. Der Prozeß endet, wenn keine Verfeinerung der Partition mehr notwendig ist.

Zustandsgraph:



Aufgabe: Gesucht sei der entsprechende **Minimalautomat**.

1. Schritt:



Wir haben zwei disjunkte Partitionsmengen $\mathbf{S_{1,1}}$ und $\mathbf{S_{1,2}}$ mit den Zuständen:

$$S_{1,1} = \{S_3, S_4\} \quad \text{und} \quad S_{1,2} = \{S_0, S_1, S_2\}$$

2. Schritt:

Partitionsmengen werden für jedes **a** $\in \Sigma = \{0, 1\}$ untersucht.

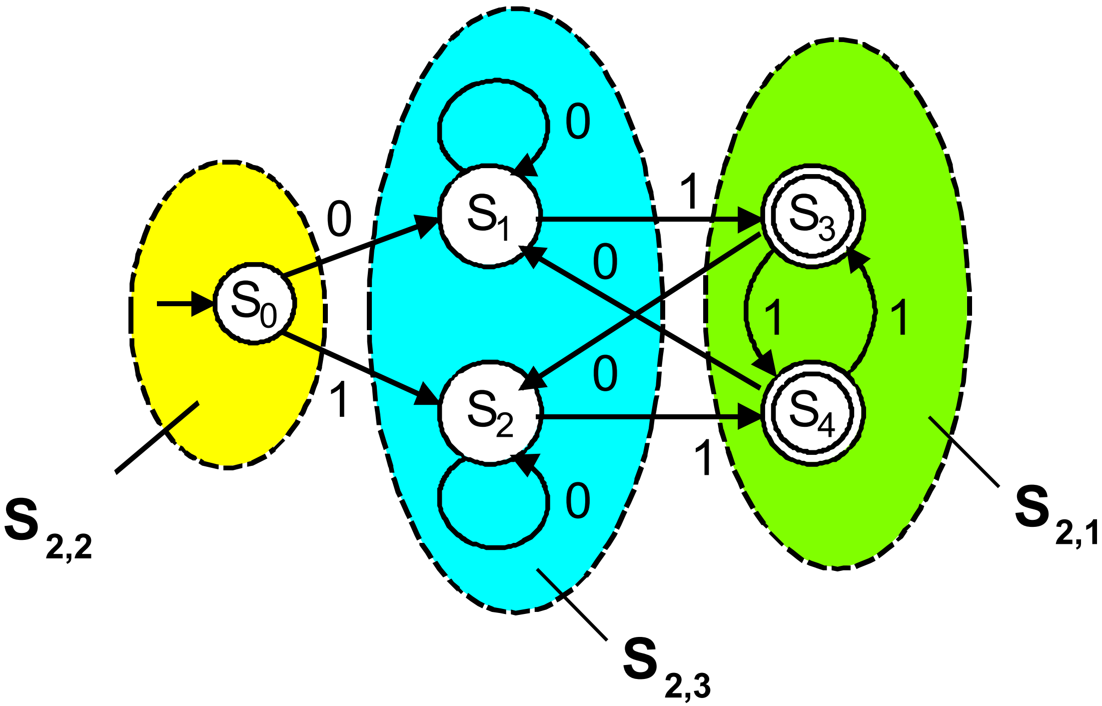
Partition		$S_{1,1}$			$S_{1,2}$		
$\Sigma\downarrow$		S_3	S_4		S_0	S_1	S_2
0		$S_{1,2}$			$S_{1,2}$		
1		$S_{1,1}$			$S_{1,2}$	$S_{1,1}$	

<u>nicht</u> in der gleichen Bildmenge!

$\Rightarrow$ $S_{1,2}$ wird weiter aufgeteilt in: $\{S_0\}$ und $\{S_1, S_2\}$

3. Schritt:

Verfeinerung von $S_{1,2}$



Wiederholung von Schritt 2:

Partitionsmengen werden für jedes **a** $\in \Sigma = \{0, 1\}$ untersucht.

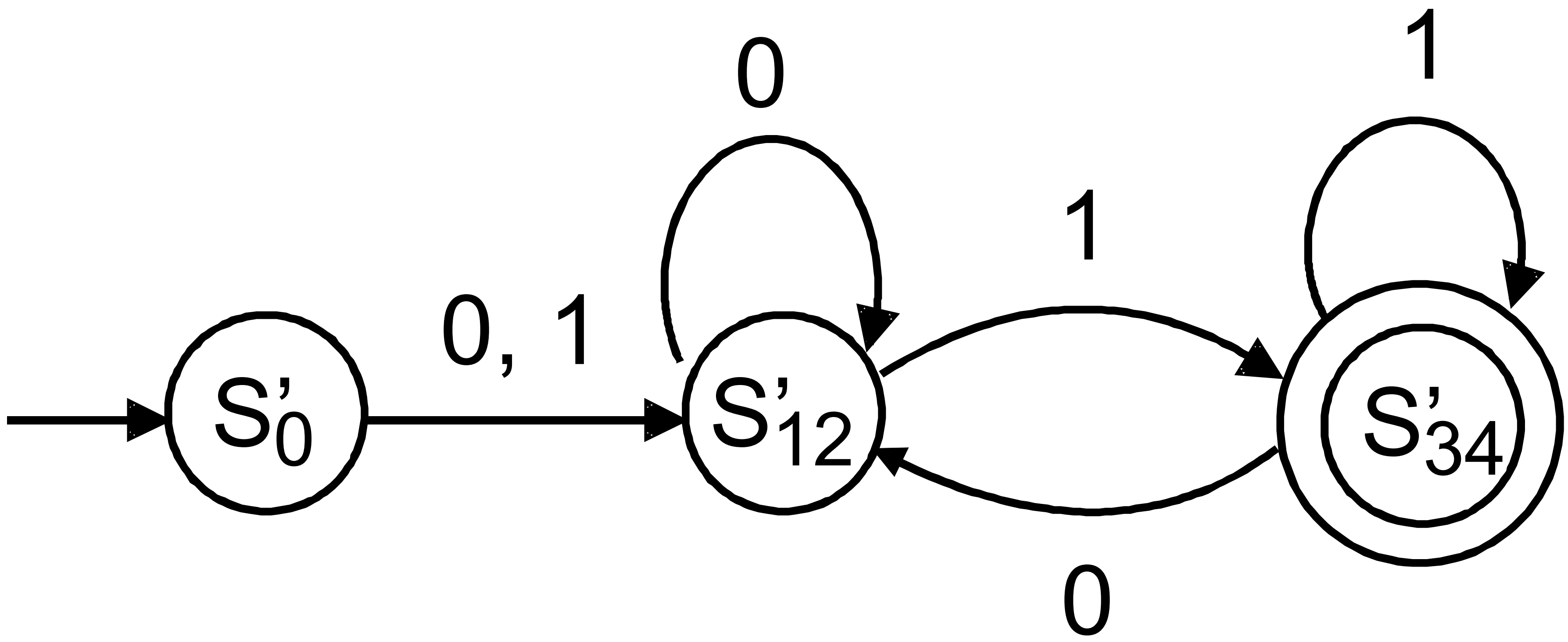
Partition		$S_{2,1}$			$S_{2,2}$		$S_{2,3}$	
$\Sigma\downarrow$		S_3	S_4		S_0		S_1	S_2
0		$S_{2,3}$			$S_{2,3}$		$S_{2,3}$	
1		$S_{2,1}$			$S_{2,3}$		$S_{2,1}$	

jeweils in der gleichen Bildmenge!

$\Rightarrow$ Damit erhalten wir den **Minimalautomaten A´** mit:

$\mathbf{S'_0} := \{S_0\}$; $\mathbf{S'_{12}} := \{S_1, S_2\}$; $\mathbf{S'_{34}} := \{S_3, S_4\}$

Ergebnis:



I. Endliche Automaten

1. Sprachgebrauch und Motivation
   - 1.1 Automaten und Zustandsüberführungsfunktion
   - 1.2 Sprache eines Automaten
2. Deterministische endliche Automaten
   - 2.1 Erweiterung der Überführungsfunktion
   - 2.2 Sprache eines deterministischen Automaten
3. Nicht-deterministische endliche Automaten
   - 3.1 Sprache eines nicht-deterministischen Automaten
   - 3.2 Teilmengenkonstruktion
4. Äquivalenz und Minimierung von Automaten
   - 4.1 Äquivalente und reduzierte Automaten
   - **4.2 Bildung von Äquivalenzklassen**

Äquivalenz und Minimierung von Automaten:

- Zwei Zustände eines DFA heißen **äquivalent**, wenn die Menge der Worte, die in einen Endzustand führen, für beide identisch ist.
- Das Zusammenlegen von äquivalenten Zuständen eines DFA führt schließlich zum sogenannten **reduzierten** bzw. **minimalen** Automaten.
- Dazu haben wir alle zueinander äquivalente Zustände in einer sogenannten **Äquivalenzklasse [s]**, die durch einen **Vertreter s** repräsentiert werden kann, zusammengefasst.
- In diesem Zusammenhang haben wir herausgestellt, dass die **Äquivalenzrelation** die üblichen Eigenschaften **Reflexivität**, **Symmetrie** und **Transitivität** hat.

Idee eines weiteren Algorithmus:

Der betrachtete Algorithmus bestimmt die Äquivalenzklassen durch Markierung aller Zustandspaare, die **nicht äquivalent** sein können:

- Dabei sind $\mathbf{x_0}$ die Anfangsmarkierungen, die sich durch die Unterscheidung der **Endzustände** und **Nicht-Endzustände** ergeben.
- Die Markierungen x_1 und x_2 sind die Zustandspaare, die beim ersten bzw. zweiten Durchlauf als **nicht äquivalent** erkannt werden.
- Alle **nicht ausge-x-ten** Zustandspaare stellen eine Äquivalenzklasse dar und können demzufolge zusammengelegt werden.

Äquivalente Zustände

Beschreibung des Algorithmus zur Ermittlung äquivalenter Zustände:

Eingabe: DFA **A** mit den Systemzuständen **1**, **2**, ..., **n**.

Ausgabe: Erkenntnis, welche Zustände von **A** noch zu verschmelzen sind, um den Minimalautomaten zu erhalten.

Voraussetzung: Zustände, welche vom Anfangszustand aus nicht erreichbar sind, müssen zuvor entfernt worden sein.

Algorithmus:

1. Man vereinbare eine Dreiecksmatrix der Form (Indizes k, i):

$$\begin{array}{llll}
(2, 1) & & & \\
(3, 1) & (3, 2) & & \\
\vdots & & & \\
(n, 1) & (n, 2) & \ldots & (n, n-1)
\end{array}$$

Äquivalente Zustände

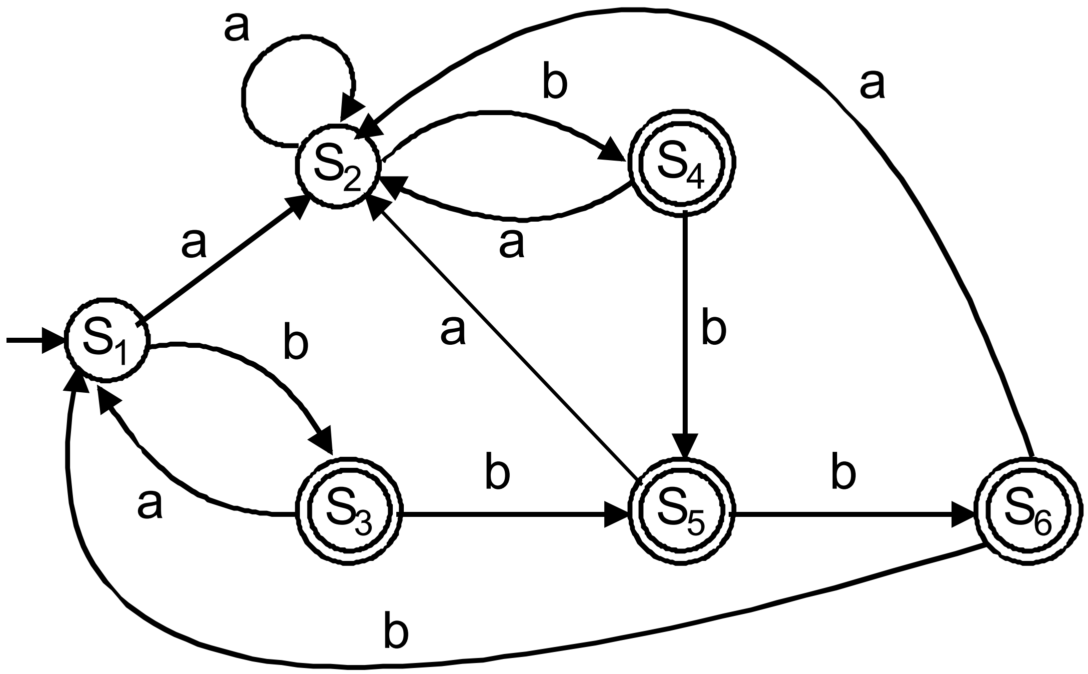
2. Initialisiere Matrixelemente mit dem Wert **1 (markiert)**, falls einer der Indizes **(k, i)** zu einem Endzustand des Automaten gehört und der andere nicht. Anderenfalls initialisiere Matrixelemente mit dem Wert **0 (unmarkiert)**.

3. Für alle mit **0 (unmarkiert)** initialisierten Matrixelemente untersuche, ob das Zustandspaar **($\delta(z_i,a)$, $\delta(z_k,a)$)** für mindestens ein Eingabezeichen $\mathbf{a} \in \Sigma$ zu einem bereits mit **1 (markiert)** initialisierten Matrixelement führt. Wenn ja, dann setze auch den Matrixwert des Elementes **(i, k)** auf **1 (markiert)**.

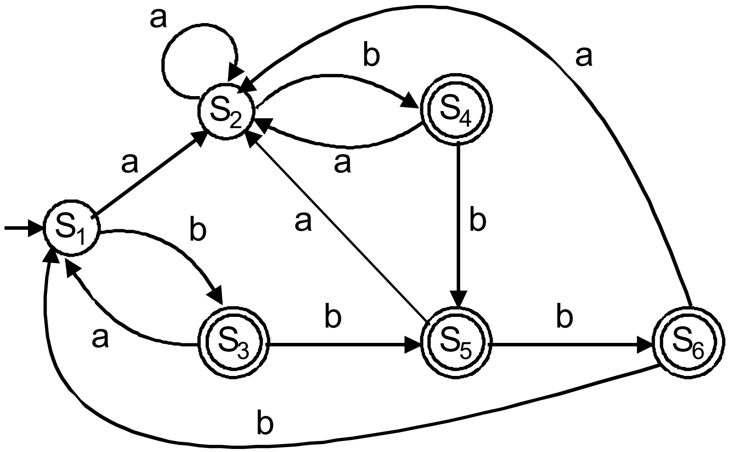
4. Wiederhole Schritt 3, bis sich ein Durchlauf ohne Änderung ergibt.

5. Alle jetzt noch mit **0 (unmarkiert)** besetzten Zustandspaare sind **äquivalente Zustände** und können zu einem Zustand verschmolzen werden.

Zustandsgraph:



Gesucht: Entsprechender Minimalautomat

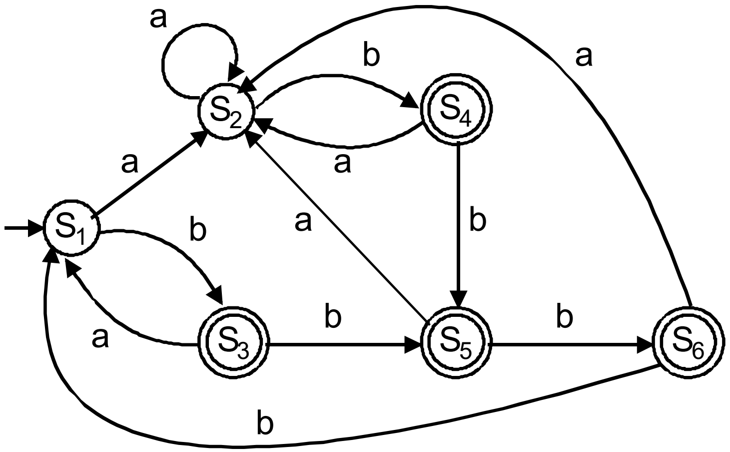


	S_1	S_2	S_3	S_4	S_5
S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0			

k	i		bereits markiert?
2	1	$(\delta(S_2, a), \delta(S_1, a)) = (S_2, S_2) \rightarrow$	nein
		$(\delta(S_2, b), \delta(S_1, b)) = (S_4, S_3) \rightarrow$	nein
4	3	$(\delta(S_4, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$	nein
		$(\delta(S_4, b), \delta(S_3, b)) = (S_5, S_5) \rightarrow$	nein
5	3	$(\delta(S_5, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$	nein
		$(\delta(S_5, b), \delta(S_3, b)) = (S_6, S_5) \rightarrow$	nein

	S_1	S_2	S_3	S_4	S_5
S_2	?				
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0			

Reduzierter Automat



S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0			
	S_1	S_2	S_3	S_4	S_5

k i bereits markiert?

6 3 $(\delta(S_6, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$ nein

$(\delta(S_6, b), \delta(S_3, b)) = (S_1, S_5) \rightarrow$ ja $\rightarrow x_1$

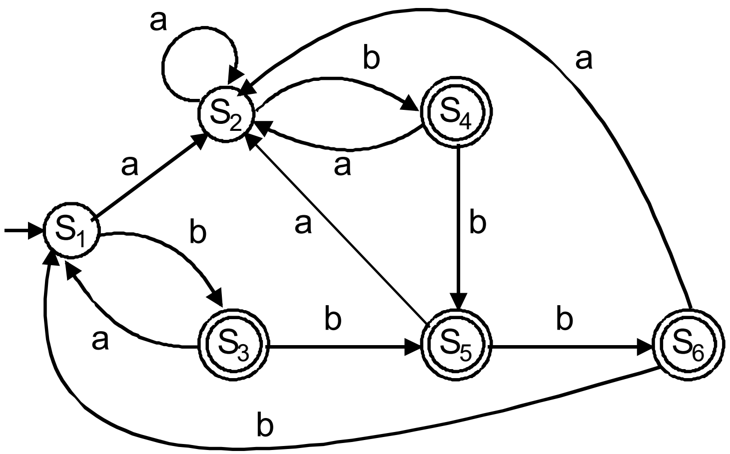
5 4 $(\delta(S_5, a), \delta(S_4, a)) = (S_2, S_2) \rightarrow$ nein

$(\delta(S_5, b), \delta(S_4, b)) = (S_6, S_5) \rightarrow$ nein

6 4 $(\delta(S_6, a), \delta(S_4, a)) = (S_2, S_2) \rightarrow$ nein

$(\delta(S_6, b), \delta(S_4, b)) = (S_1, S_5) \rightarrow$ ja $\rightarrow x_1$

S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0	x_1	x_1	
	S_1	S_2	S_3	S_4	S_5



	S_1	S_2	S_3	S_4	S_5
S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0	x_1	x_1	

k i **bereits markiert?**

6 5 $(\delta(S_6, a), \delta(S_5, a)) = (S_2, S_2) \rightarrow$ nein

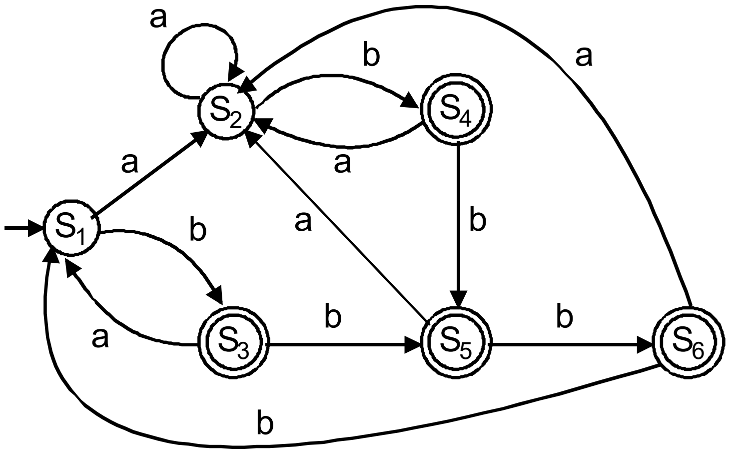
$(\delta(S_6, b), \delta(S_5, b)) = (S_1, S_6) \rightarrow$ ja $\rightarrow x_1$

2. Durchgang

2 1 $(\delta(S_2, a), \delta(S_1, a)) = (S_2, S_2) \rightarrow$ nein

$(\delta(S_2, b), \delta(S_1, b)) = (S_4, S_3) \rightarrow$ nein

	S_1	S_2	S_3	S_4	S_5
S_2	?				
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0	x_1	x_1	x_1



	S_1	S_2	S_3	S_4	S_5
S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0			
S_6	x_0	x_0	x_1	x_1	x_1

k i bereits markiert?

4 3 $(\delta(S_4, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$ nein

$(\delta(S_4, b), \delta(S_3, b)) = (S_5, S_5) \rightarrow$ nein

5 3 $(\delta(S_5, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$ nein

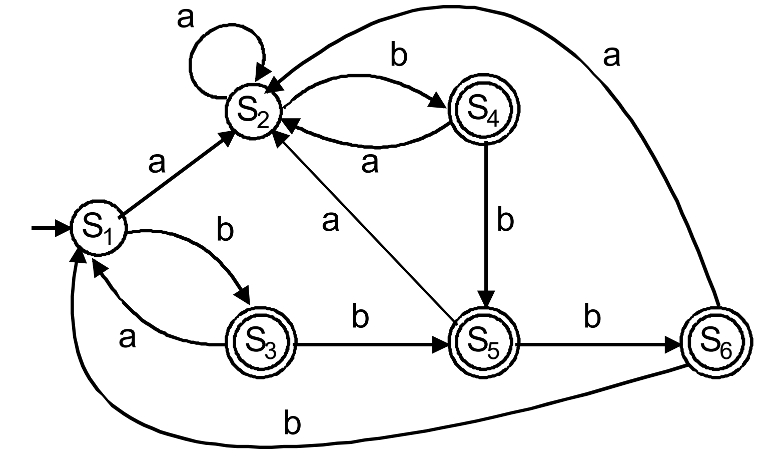
$(\delta(S_5, b), \delta(S_3, b)) = (S_6, S_5) \rightarrow$ ja $\rightarrow x_2$

5 4 $(\delta(S_5, a), \delta(S_4, a)) = (S_2, S_2) \rightarrow$ nein

$(\delta(S_5, b), \delta(S_4, b)) = (S_6, S_5) \rightarrow$ ja $\rightarrow x_2$

	S_1	S_2	S_3	S_4	S_5
S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0	x_2	x_2	
S_6	x_0	x_0	x_1	x_1	x_1

Reduzierter Automat



S_2					
S_3	x_0	x_0			
S_4	x_0	x_0			
S_5	x_0	x_0	x_2	x_2	
S_6	x_0	x_0	x_1	x_1	x_1
	S_1	S_2	S_3	S_4	S_5

k i **bereits markiert?**

3. Durchgang

2 1 $(\delta(S_2, a), \delta(S_1, a)) = (S_2, S_2) \rightarrow$ nein

$(\delta(S_2, b), \delta(S_1, b)) = (S_4, S_3) \rightarrow$ nein

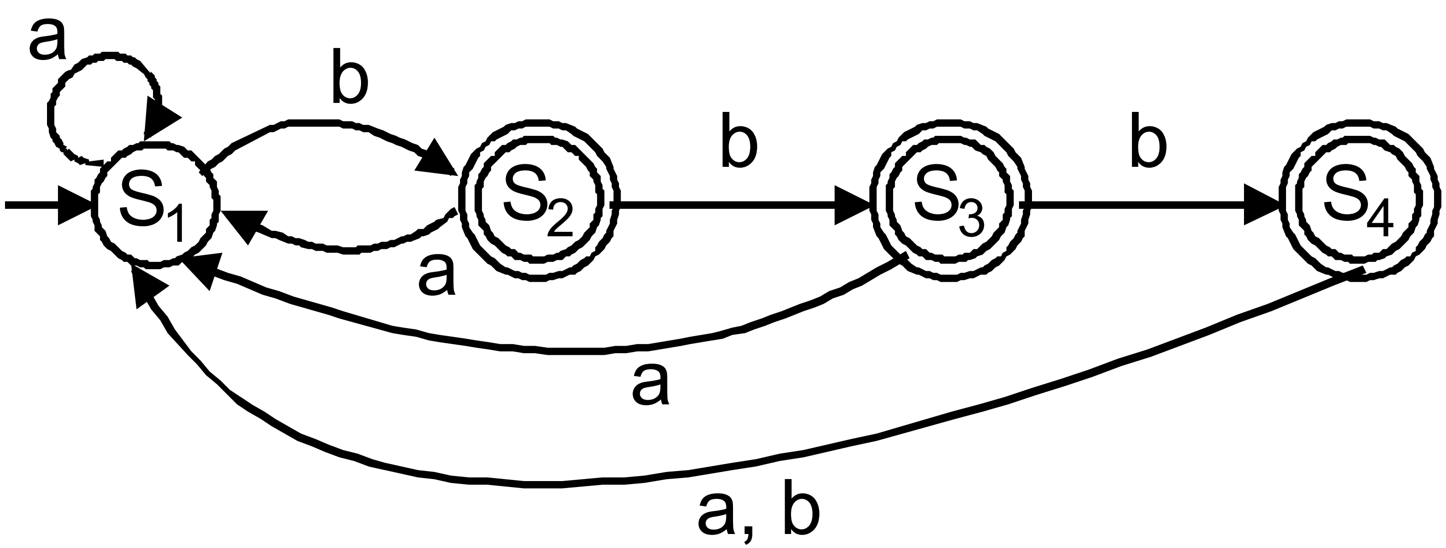
4 3 $(\delta(S_4, a), \delta(S_3, a)) = (S_2, S_1) \rightarrow$ nein

$(\delta(S_4, b), \delta(S_3, b)) = (S_5, S_5) \rightarrow$ nein

S_2	nein				
S_3	x_0	x_0			
S_4	x_0	x_0	nein		
S_5	x_0	x_0	x_2	x_2	
S_6	x_0	x_0	x_1	x_1	x_1
	S_1	S_2	S_3	S_4	S_5

Ergebnis:

Die Zustände S_1 und S_2 bzw. S_3 und S_4 sind **äquivalent** und können damit zusammengelegt werden → **Minimalautomat**



Automatentheorie und Formale Sprachen

– LV 4110 –

Kellerautomaten und Kontextfreie Sprachen

- Kennenlernen der Beschreibungsmöglichkeiten von Programmiersprachen
- Klärung, was man unter der Mehrdeutigkeit einer Sprache sowie der Mehrdeutigkeit einer Grammatik versteht
- Definition von Normalformen kontextfreier Grammatiken
- Kennenlernen des Pumping-Lemmas
- Modellierung und Arbeitsweise eines Kellerautomaten
- Kennenlernen des Zusammenhangs zwischen Kellerautomaten und kontextfreien Grammatiken
- Identifizierung der Probleme bei der Syntaxanalyse

IV. Kellerautomaten und Kontextfreie Sprachen

1. Grammatiken von Programmiersprachen
   - 1.1 Beschreibungsmittel BNF und Syntaxdiagramme
   - 1.2 Definition für eine Chomsky-Grammatik
2. Mehrdeutigkeit bei kontextfreien Grammatiken
   - 2.1 Definition von Mehrdeutigkeiten
   - 2.2 Inhärent mehrdeutige Sprachen
3. Normalformen kontextfreier Grammatiken
   - 3.1 ε-freie Grammatik
   - 3.2 Greibach-Normalform
   - 3.3 Chomsky-Normalform

Fortsetzung:

4. Das Pumping Lemma
   - 4.1 Pumping Lemma für reguläre Sprachen
   - 4.2 Pumping Lemma für kontextfreie Sprachen
5. Kellerautomaten
   - 5.1 Modellbildung
   - 5.2 Deterministische Kellerautomaten
   - 5.3 Sprache des deterministischen Kellerautomaten
   - 5.4 Nicht-deterministische Kellerautomaten
   - 5.5 Sprache des Nicht-deterministischen Kellerautomaten
   - 5.6 Kellerautomaten und kontextfreie Grammatiken
   - 5.7 Das Problem der Syntaxanalyse

Wir erinnern uns:

Chomsky-Grammatiken vom Typ 2 haben die Form:

$$A \rightarrow \gamma \quad \text{mit} \quad A \in N\,; \quad \gamma \in (N \cup T)^*$$

$$\text{oder} \quad \gamma = \varepsilon$$

Diese Regelform ist charakteristisch für die meisten höheren Programmiersprachen (PASCAL, C, ...).

- Backus-Naur-Form
- Syntaxdiagramme

IV. Kellerautomaten und Kontextfreie Sprachen

1. Grammatiken von Programmiersprachen

1.1 Beschreibungsmittel BNF und Syntaxdiagramme

1.2 Definition für eine Chomsky-Grammatik

2. Mehrdeutigkeit bei kontextfreien Grammatiken

2.1 Definition von Mehrdeutigkeiten

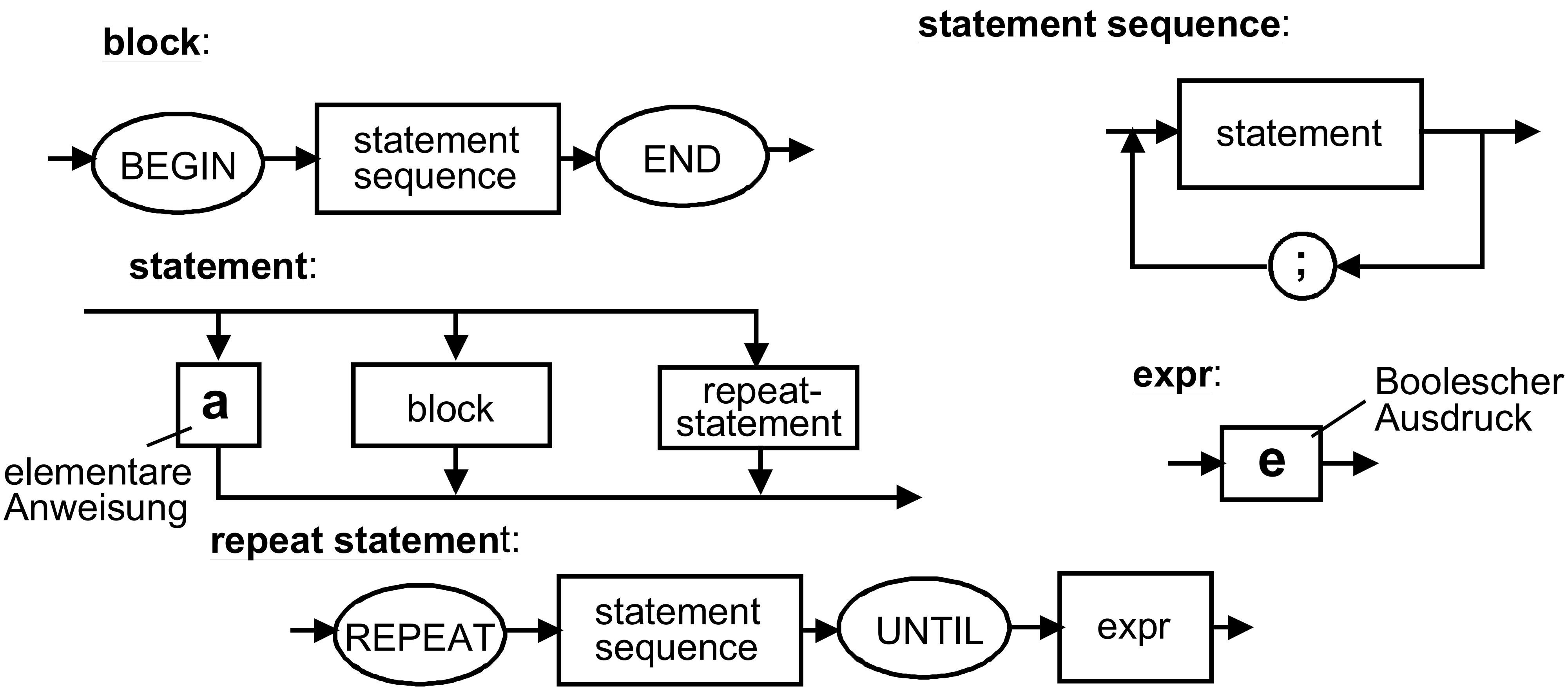
2.2 Inhärent mehrdeutige Sprachen

3. Normalformen kontextfreier Grammatiken

3.1 ε-freie Grammatik

3.2 Greibach-Normalform

3.3 Chomsky-Normalform



block:
BEGIN
statement sequence
END
statement:
a
block
repeat-statement
elementare Anweisung
repeat statement:
REPEAT
statement sequence
UNTIL
expr
statement sequence:
statement
;
expr:
e
Boolescher Ausdruck

Definition der syntaktischen Variablen:

<block>	::= **BEGIN** <statement seq.> **END**
<statement sequence>	::= <statement> { ; <statement> }
<statement>	::= **a** | <block> | <repeat-statement>
<repeat-statement>	::= **REPEAT** <statement sequence> **UNTIL** <expr>
<expr>	::= **e**

IV. Kellerautomaten und Kontextfreie Sprachen

1. Grammatiken von Programmiersprachen
   - 1.1 Beschreibungsmittel BNF und Syntaxdiagramme
   - **1.2 Definition für eine Chomsky-Grammatik**
2. Mehrdeutigkeit bei kontextfreien Grammatiken
   - 2.1 Definition von Mehrdeutigkeiten
   - 2.2 Inhärent mehrdeutige Sprachen
3. Normalformen kontextfreier Grammatiken
   - 3.1 ε-freie Grammatik
   - 3.2 Greibach-Normalform
   - 3.3 Chomsky-Normalform

Definitionen für eine Chomsky-Grammatik:

- Terminalsymbole

  T = { **BEGIN**, **END**, **REPEAT**, **UNTIL**, **a**, **e**, **;** }
- Nonterminalsymbole

  N = { S, A, B, C, D, E } mit

  S = <block>
  A = <statement sequence>
  B = <statement>
  C = <repeat-statement>
  D = <expr>
  E = { ... } (Iteration) bzw. | (Alternative)

Ersetzt man in der BNF das Symbol **::=** durch den Ableitungspfeil $\Rightarrow$, so ergeben sich folgende Ableitungsregeln bzw. Produktionen **P**:

S $\Rightarrow$ **BEGIN** A **END**	(1)
A $\Rightarrow$ B E	(2)
E $\Rightarrow$ ε | **;** A	(3, 4)
B $\Rightarrow$ **a** | S | C	(5, 6, 7)
C $\Rightarrow$ **REPEAT** A **UNTIL** D	(8)
D $\Rightarrow$ **e**	(9)

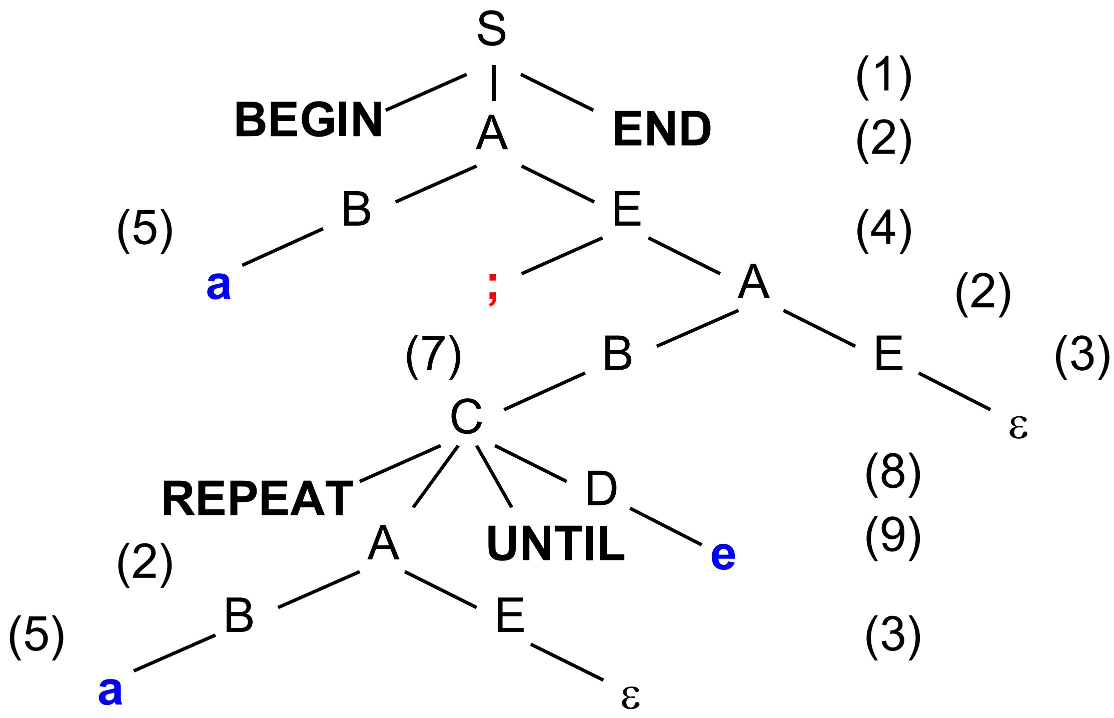
Ergebnis:

G = (**T**, **N**, **P**, S)

Beispiel:

BEGIN a ; Repeat a UNTIL e END

Für dieses Beispiel ergibt sich nebenstehender **Ableitungsbaum**



IV. Kellerautomaten und Kontextfreie Sprachen

1. Grammatiken von Programmiersprachen
   - 1.1 Beschreibungsmittel BNF und Syntaxdiagramme
   - 1.2 Definition für eine Chomsky-Grammatik
2. **Mehrdeutigkeit bei kontextfreien Grammatiken**
   - **2.1 Definition von Mehrdeutigkeiten**
   - 2.2 Inhärent mehrdeutige Sprachen
3. Normalformen kontextfreier Grammatiken
   - 3.1 ε-freie Grammatik
   - 3.2 Greibach-Normalform
   - 3.3 Chomsky-Normalform

Definition:

Eine Grammatik heißt ***mehrdeutig***, wenn es ein Wort in **L(G)** gibt, zu dem Ableitungsbäume von **unterschiedlicher Struktur** existieren.

Bemerkung:

Der Ableitungsbaum ist ein **statisches** Gebilde, zu dem man auf verschiedenen Wegen, d. h. durch in der Reihenfolge der Schritte unterschiedliche Ableitungen, kommen kann. Wenn das Endresultat immer gleich ist, bedeutet dies noch keine Mehrdeutigkeit der Grammatik. Mit anderen Worten: Für die Mehrdeutigkeit sind verschiedene Ableitungsbäume maßgebend. Es reicht nicht aus, dass für ein Wort verschiedene Ableitungen existieren, denn diese können denselben Ableitungsbaum festlegen.

<u>Eindeutige Grammatik</u>:

Wir betrachten folgende **eindeutige** Grammatik G = (**T**, **N**, **P**, S) mit dem Startsymbol S:

T = { **a**, **b**, **c**, **+**, *****, **(**, **)** }

N = { S, T, F, Z }

P = { S ⇒ T | S **+** T , (1, 2)

T ⇒ F | F ***** T , (3, 4)

F ⇒ Z | **(** S **)** , (5, 6)

Z ⇒ **a** | **b** | **c** } (7, 8, 9)

Beispiel:

$$a * b + c$$

Ableitung:

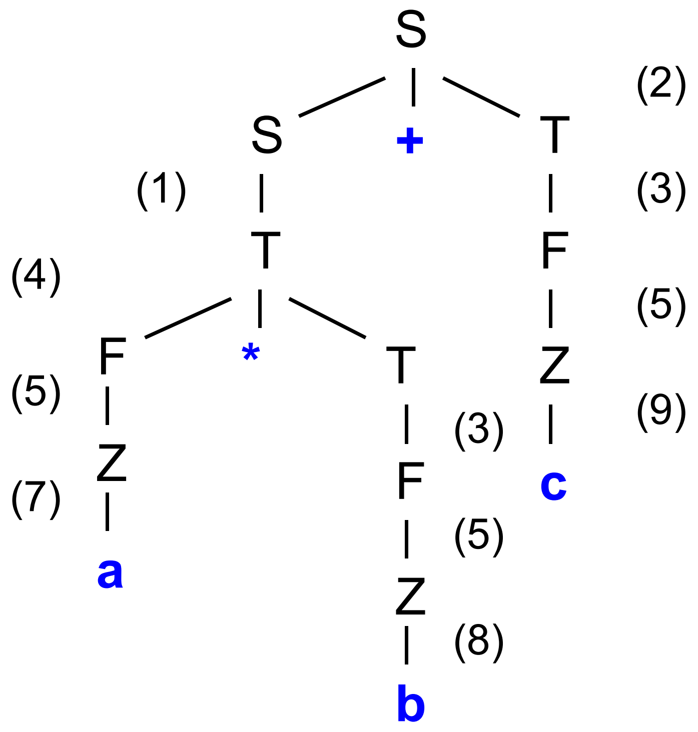
$$S \Rightarrow_{(2)} \underline{S} + T \Rightarrow_{(1)} \underline{T} + T \Rightarrow_{(4)} \underline{F} * T + T$$

$$\Rightarrow_{(5)} \underline{Z} * T + T \Rightarrow_{(7)} a * \underline{T} + T$$

$$\Rightarrow_{(3)} a * \underline{F} + T \Rightarrow_{(5)} a * \underline{Z} + T$$

$$\Rightarrow_{(8)} a * b + \underline{T} \Rightarrow_{(3)} a * b + \underline{F}$$

$$\Rightarrow_{(5)} a * b + \underline{Z} \Rightarrow_{(9)} a * b + c$$



IV. Kellerautomaten und Kontextfreie Sprachen

Folgerung aus vorangegangenem Beispiel:

Zu jeder kontextfreien Grammatik gibt es unendlich viele äquivalente kontextfreie Grammatiken.

Schlußbemerkung:

Mehrdeutige Grammatiken sind unerwünscht, weil sie Interpretationsschwierigkeiten verursachen können. Mehrdeutigkeit kann aber u. U. nicht vermieden werden. Man spricht von einer ***inhärent mehrdeutigen*** *Sprache,* wenn **jede** Grammatik, die die Sprache erzeugt, mehrdeutig ist.
Aus der Mehrdeutigkeit einer Sprache läßt sich auch auf die Mehrdeutigkeit der zugehörigen Grammatik schließen, aber nicht umgekehrt.

IV. Kellerautomaten und Kontextfreie Sprachen

Definition:

Eine kontextfreie Grammatik G = (**N**, **T**, **P**, S) heißt *ε-frei,* wenn es in **P** keine Regel der Form **A → ε** mit A ∈ **N** gibt.

Satz:

Zu jeder kontextfreien Grammatik G mit der Sprache L(G) gibt es eine ε-freie mit der Sprache L(G´) = L(G) - {ε}.

Hinweis zur Konstruktion:

Regel A → ε kann wegfallen, wenn man z. B. die Regel B → aA durch B → a ergänzt.

Beispiel:

Gegeben sei G = (**N**, **T**, **P**, S) mit

P = { $S \Rightarrow AB$, $A \Rightarrow a$, $B \Rightarrow \varepsilon$ } (1, 2, 3) $\Rightarrow$ L(G) = { a }

gezeigt: $S \Rightarrow_{(1)} \underline{A}B \Rightarrow_{(2)} a\underline{B} \Rightarrow_{(3)} a\varepsilon = a$

Es sei nun G' gegeben mit

P = { $S \Rightarrow AB$, $A \Rightarrow a$, $S \Rightarrow A$ } (1, 2, 4) $\Rightarrow$ L(G') = { a }

gezeigt: $S \Rightarrow_{(4)} \underline{A} \Rightarrow_{(2)} a$

Ergebnis:

Durch Hinzufügen der Regel (4) haben wir nun eine **ε-freie** Grammatik G' zum Erzeugen der **gleichen Sprache**.

Anmerkung:

Gegeben sei G = (**N**, **T**, **P**, S) mit

$\mathbf{P} = \{ S \Rightarrow AB,\ A \Rightarrow a,\ B \Rightarrow \varepsilon \}$ (1, 2, 3) $\Rightarrow$ $L(G) = \{ a \}$

gezeigt: $S \Rightarrow_{(1)} \underline{A}B \Rightarrow_{(2)} a\underline{B} \Rightarrow_{(3)} a\varepsilon = a$

Hingegen würde das Entfernen der Regel (3) in G keinen Sinn ergeben, da für G’’ mit

$\mathbf{P} = \{ S \Rightarrow AB,\ A \Rightarrow a \}$ (1, 2) $\Rightarrow$ $L(G'') = \{ \ \}$

gezeigt: $S \Rightarrow_{(1)} \underline{A}B \Rightarrow_{(2)} aB$ und B nicht weiter ersetzbar!

IV. Kellerautomaten und Kontextfreie Sprachen

Definition:

Eine kontextfreie Grammatik G = (**N**, **T**, **P**, S) ist in der **Greibach-Normalform** (kurz GNF), wenn **P** nur Regeln der Form **$A \to a\ \varphi$** mit $A \in$ **N**, $a \in$ **T** und $\varphi \in$ **N*** enthält.

Satz:

Zu jeder kontextfreien Grammatik G mit der Sprache L(G) gibt es eine kontextfreie Grammatik G´ in Greibach-Normalform mit

$$L(G') = L(G) - \{\varepsilon\}.$$

Bemerkung:

Die Fragestellung, ob ein gegebenes Wort oder Programm zum Sprachumfang einer kontextfreien Sprache gehört, ist in der Regel einfacher zu untersuchen, wenn die Grammatik in Greibach-Normalform vorliegt, weil dann das Programm in der Reihenfolge **von links nach rechts** analysiert und abgearbeitet werden kann und der Ableitungsbaum systematischer in dieser Richtung aufgebaut werden kann.

IV. Kellerautomaten und Kontextfreie Sprachen

Definition:

Eine kontextfreie Grammatik G = (**N**, **T**, **P**, S) ist in der **Chomsky-Normalform** (kurz CNF), wenn **P** nur Regeln der Form **A → BC** oder **A → a** mit A, B, C ∈ **N** und a ∈ **T** enthält.

Satz:

Zu jeder kontextfreien Grammatik G mit der Sprache L(G) gibt es eine kontextfreie Grammatik G´ in Chomsky-Normalform mit

$$L(G') = L(G) - \{\varepsilon\}.$$

Beispiel:

Gegeben sei G = (**N**, **T**, **P**, S) mit **N** = { S }, **T** = { **(**, **)** } und

P = { S ⇒ S S, S ⇒ (S), S ⇒ ε } (1, 2, 3)

Mögliche Ableitung:

$S \Rightarrow_{(2)} \textbf{(}\ \underline{S}\ \textbf{)} \Rightarrow_{(1)} \textbf{(}\ \underline{S}\, S\ \textbf{)} \Rightarrow_{(2)} \textbf{(}\,\textbf{(}\ S\ \textbf{)}\ \underline{S}\ \textbf{)} \Rightarrow_{(2)} \textbf{(}\,\textbf{(}\ S\ \textbf{)}\textbf{(}\ S\ \textbf{)}\,\textbf{)}$

$\Rightarrow_{(3)} \textbf{(}\,\textbf{(}\ \varepsilon\ \textbf{)}\textbf{(}\ \underline{S}\ \textbf{)}\,\textbf{)} \Rightarrow_{(2)} \textbf{(}\,\textbf{(}\ \varepsilon\ \textbf{)}\textbf{(}\,\textbf{(}\ \underline{S}\ \textbf{)}\,\textbf{)}\,\textbf{)} \Rightarrow_{(3)} \textbf{(}\,\textbf{(}\ \varepsilon\ \textbf{)}\textbf{(}\,\textbf{(}\ \varepsilon\ \textbf{)}\,\textbf{)}\,\textbf{)}$

d. h.

(( )(( )))

Nun Herleitung einer Grammatik in **CN** ⇒ 2 Schritte

1. Schritt: Umformung von G in ε-freies G'.

$\mathbf{P} = \{ S \Rightarrow S\,S,\ S \Rightarrow K_a\ S\,K_z,\ S \Rightarrow K_a\,K_z,\ K_a \Rightarrow \mathbf{(},\ K_z \Rightarrow \mathbf{)} \}$

neue Regel (3) (1, 2, 3, 4, 5)

2. Schritt: Nonterminalsymbol **H** einführen mit der Regel **H** $\Rightarrow S\,K_z$.

$\mathbf{P} = \{ S \Rightarrow S\,S,\ S \Rightarrow K_a\,\mathbf{H},\ \mathbf{H} \Rightarrow S\,K_z,\ S \Rightarrow K_a\,K_z,\ K_a \Rightarrow \mathbf{(},\ K_z \Rightarrow \mathbf{)} \}$

(1, 2, 3, 4, 5, 6)

Dies entspricht nun der **Chomsky-Normalform**, da nur Regeln der Form: $A \Rightarrow B\,C$ oder $A \Rightarrow a$.

Mögliche Ableitung gemäß **CN**:

$$\mathbf{P} = \{\, S \Rightarrow S\,S,\; S \Rightarrow K_a\, \mathbf{H},\; \mathbf{H} \Rightarrow S\,K_z,\; S \Rightarrow K_a\, K_z,\; K_a \Rightarrow \mathbf{(},\; K_z \Rightarrow \mathbf{)} \,\}$$

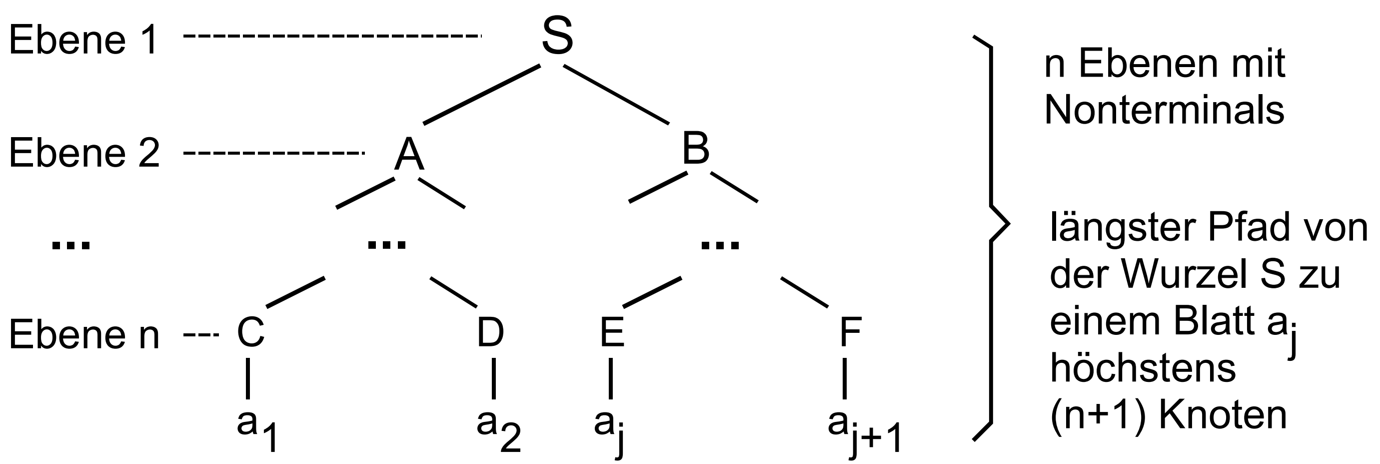
(1, 2, 3, 4, 5, 6)

$$S \Rightarrow_{(2)} \underline{K_a}\,\mathbf{H} \Rightarrow_{(5)} \mathbf{(}\,\underline{\mathbf{H}} \Rightarrow_{(3)} \mathbf{(}\,S\,\underline{K_z} \Rightarrow_{(6)} \mathbf{(}\,\underline{S}\,\mathbf{)} \Rightarrow_{(1)} \mathbf{(}\,\underline{S}\,S\,\mathbf{)} \Rightarrow_{(4)} \mathbf{(}\,\underline{K_a}\,K_z\,\underline{S}\,\mathbf{)}$$

$$\Rightarrow_{(2)} \mathbf{(}\,\underline{K_a}\,K_z\,K_a\,\mathbf{H}\,\mathbf{)} \Rightarrow_{(5)} \mathbf{((}\,\underline{K_z}\,K_a\,\mathbf{H}\,\mathbf{)} \Rightarrow_{(6)} \mathbf{(()}\,\underline{K_a}\,\mathbf{H}\,\mathbf{)} \Rightarrow_{(5)} \mathbf{(()(}\,\underline{\mathbf{H}}\,\mathbf{)}$$

$$\Rightarrow_{(3)} \mathbf{(()(}\,\underline{S}\,K_z\,\mathbf{)} \Rightarrow_{(4)} \mathbf{(()(}\,K_a\,K_z\,\underline{K_z}\,\mathbf{)} \Rightarrow_{(6)} \mathbf{(()(}\,\underline{K_a}\,K_z\,\mathbf{))}$$

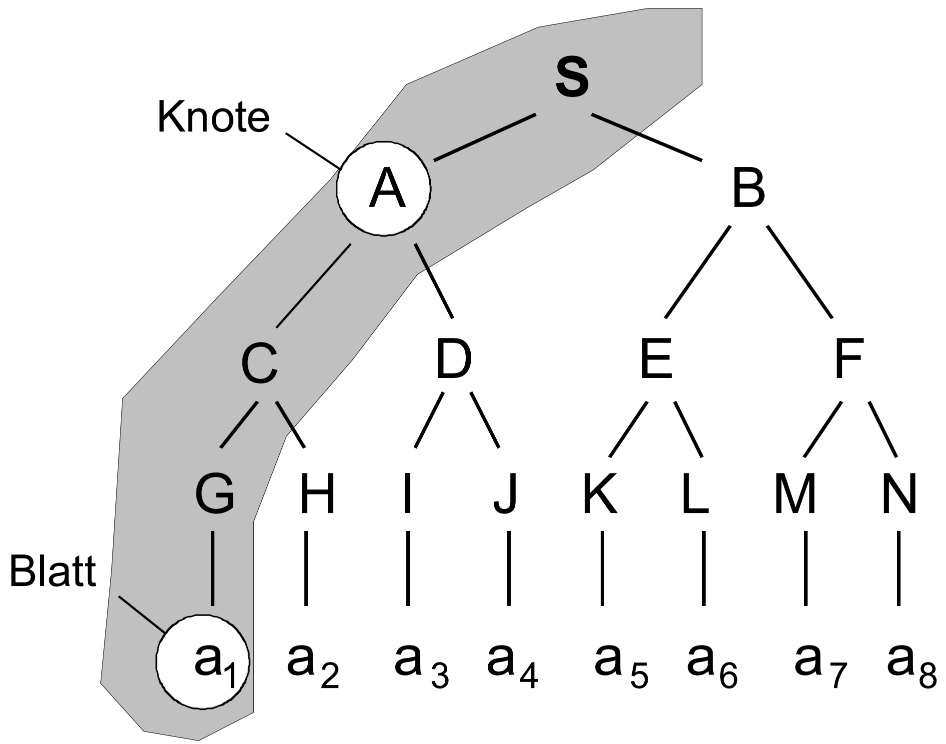
$$\Rightarrow_{(6)} \mathbf{(()((}\,\underline{K_z}\,\mathbf{))} \Rightarrow_{(6)} \boxed{\mathbf{(()(()))}}$$



- Es handelt sich um einen **Binärbaum**, bei dem jeder Knoten genau zwei Sohnknoten hat (mit Ausnahme der letzten Ebene).
- Da sich beim Übergang auf die nächste Ebene (mit Ausnahme der letzten Ebene) die Zeichenmenge **höchstens verdoppeln** kann, gilt für die **Länge** des abgeleiteten Wortes $|w| \leq 2^{n-1}$.

Beispiel:

$n = 4$



Ebene 1

Ebene 2

Ebene 3

Ebene **n = 4**

$| w | = 8 = 2^{3}$

hier:

längster Pfad von S bis a_1 umfasst 5 = (n + 1) Knoten

hier: $\mathbf{n} - 1 = 3$

4. Das Pumping Lemma

4.1 Pumping Lemma für reguläre Sprachen

4.2 Pumping Lemma für kontextfreie Sprachen

5. Kellerautomaten

5.1 Modellbildung
5.2 Deterministische Kellerautomaten
5.3 Sprache des deterministischen Kellerautomaten
5.4 Nicht-deterministische Kellerautomaten
5.5 Sprache des Nicht-deterministischen Kellerautomaten
5.6 Kellerautomaten und kontextfreie Grammatiken
5.7 Das Problem der Syntaxanalyse

Satz:

Es sei **L(G)** eine regüläre Sprache. Dann gibt es eine von **G** abhängige Zahl **k**, so dass jedes Wort $\mathbf{w} \in \mathbf{L(G)}$ mit der Länge $\mathbf{|w| \geq k}$ in der Form

$$\mathbf{w = xyz} \qquad \text{mit} \qquad \mathbf{x, y, z \in T^*}$$

geschrieben werden kann, wobei gilt:

a) $|xy| \leq k$

b) $y \neq \varepsilon$

c) $xy^iz \in L(G)$ für $i \geq 0$.

<u>Satz</u>:

Es sei **L(G)** eine <u>kontextfreie</u> Sprache. Dann gibt es eine von **G** abhängige Zahl **k**, so dass jedes Wort $\mathbf{w} \in \mathbf{L(G)}$ mit der Länge $|\mathbf{w}| \geq \mathbf{k}$ in der Form

$$\mathbf{w = xuyvz} \qquad \text{mit} \qquad \mathbf{x, u, y, v, z} \in \mathbf{T^*}$$

geschrieben werden kann, wobei gilt:

a) $|uyv| \leq k$

b) $uv \neq \varepsilon$

c) $xu^iyv^iz \in L(G)$ für $i \geq 0$.

Beweisvorbereitung:

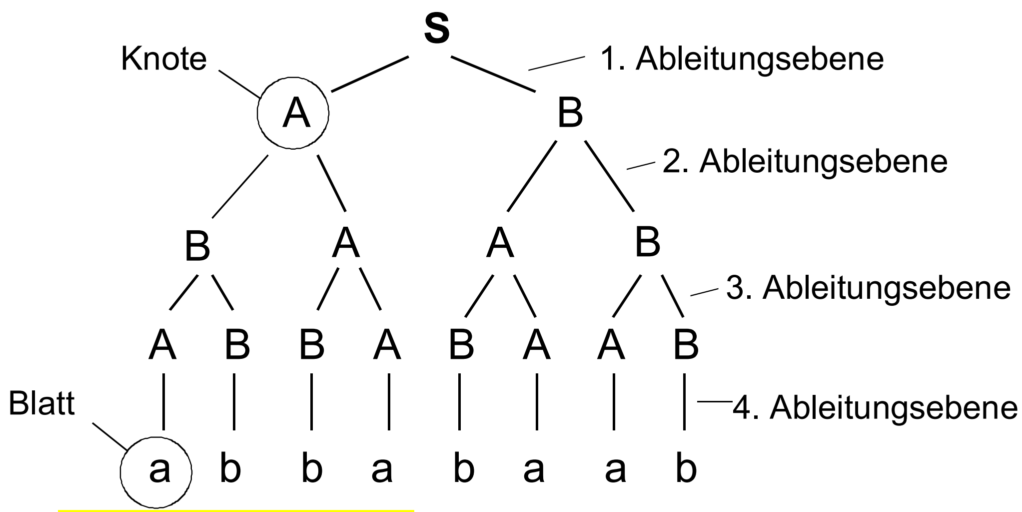
Ausgangspunkt: Sei **G** in der Chomsky-Normalform (CNF), d. h. o. E. nur Regeln der Form:

$\mathbf{A} \rightarrow \mathbf{a}$, wobei $\mathbf{a} \in \mathbf{T}$

oder $\mathbf{A} \rightarrow \mathbf{BC}$, wobei $\mathbf{B}, \mathbf{C} \in \mathbf{N}$.

Wir wählen: $\mathbf{k = 2^n}$, $\mathbf{n = \#N}$ (Anzahl der Nichtterminale einschließlich des Startsymbols **S**)

Ableitung von **w**: Der Ableitungsbaum für ein Wort $\mathbf{w} \in \mathbf{L(G)}$ mit voraussetzungsgemäß $\mathbf{|w| \geq k = 2^n > 2^{n-1}}$ Blätter ist ein Binärbaum (mit dem Startsymbol **S**)



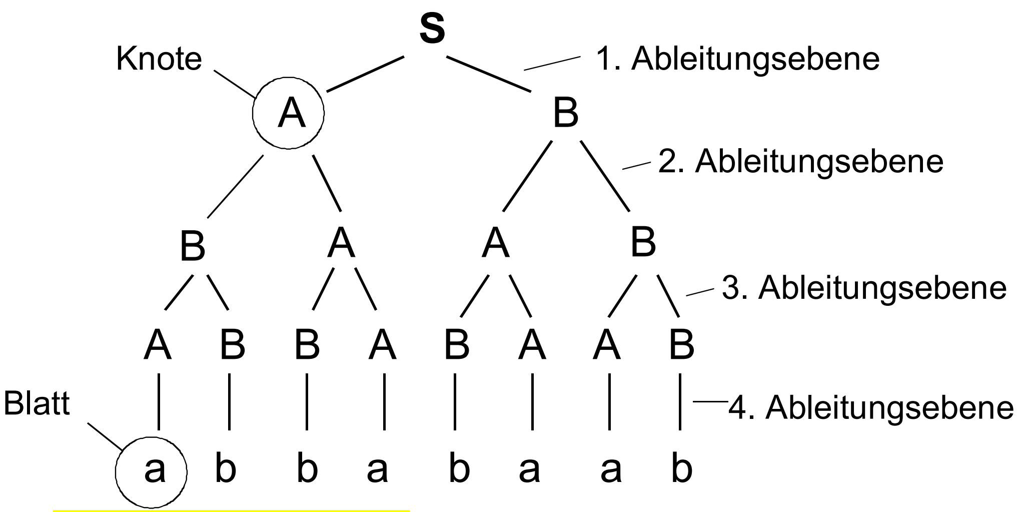
$|w| \geq k = 2^3 = 8 \Rightarrow$ **4 Ableitungsebenen**

Knoten mit **genau einem** Blatt repräsentieren eine Regel der Form **$A \to a$**.

Alle anderen Knoten mögen für eine Regel der Form **$A \to BC$** stehen und haben **genau zwei** Blätter.

hier: **$n = 3$**, weil **$N = \{S, A, B\}$**

Um in einem solchen Binärbaum **2^n** Blätter zu erhalten, benötigt man mindestens **$n + 1$ Ableitungsebenen**.
Um von dem Startsymbol **S** aus zu einem Blatt zu gelangen, ergibt sich eine Knotenfolge von mindestens **$n + 1$ Nonterminals** (einschließlich dem Startsymbol **S**).



$A \to a$.

$A \to BC$

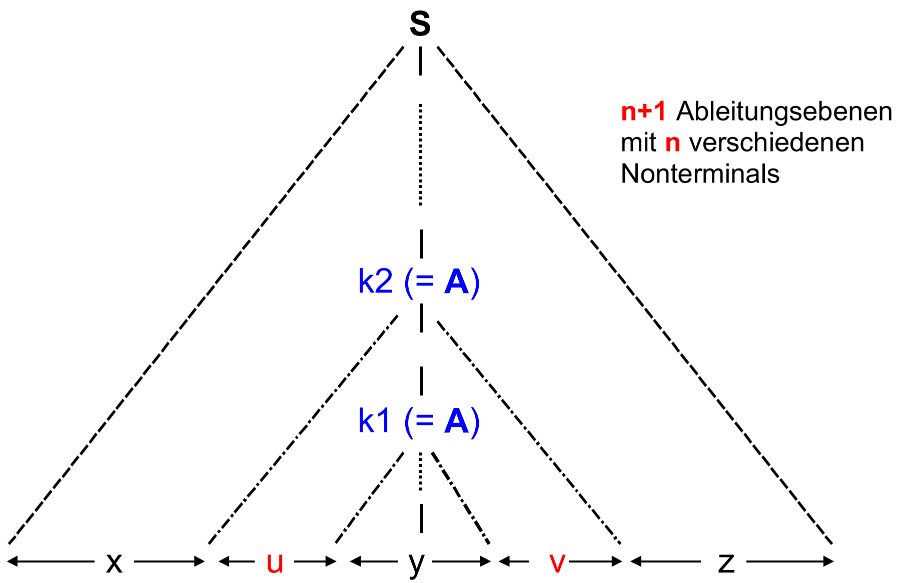
$|w| \geq k = 2^n$

$|w| \geq k = 2^3 = 8 \Rightarrow$ **4 Ableitungsebenen** hier: **$n = 3$** , und **$N = \{S, A, B\}$**

Folgerung:

Da definitionsgemäß jedoch nur **n** verschiedene Nonterminals existieren (vgl. S. 37 **$n = \#N$**), tritt mindestens **ein** Nonterminal mehrfach auf.

Wir bezeichnen dieses Nonterminal im folgenden mit **A**.



Bezeichnung:

Wir bezeichnen die letzte Wiederholung von **A** mit k1, die vorletzte mit k2 und erhalten nebenstehenden Ableitungsbaum:

Es ist also:

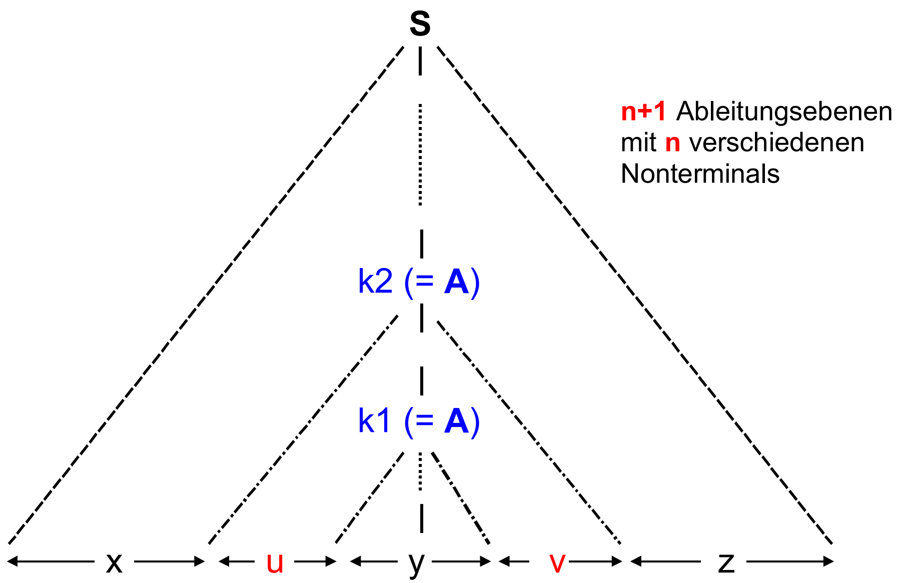
y aus k1

uyv aus k2 und

w = xuyvz aus **S** abgeleitet

1. Folgerung:

Da uyv aus k2 erzeugt wird und für diese Ableitung höchstens **n+1** Ableitungsschritte benötigt werden, gilt: $|uyv| \leq \mathbf{2^n}$ bzw. $|uyv| \leq \mathbf{k}$.



Es sind:

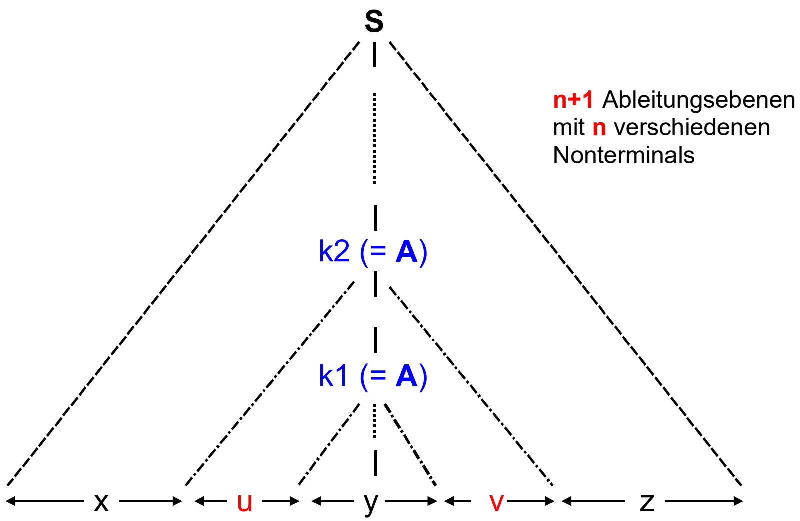
y aus k1

uyv aus k2 und

w = xuyvz aus **S** abgeleitet

2. Folgerung:

k2 hat **genau** **zwei** Nachfolger. Aus dem einen Nachfolger geht später y hervor, aus dem anderen u oder v. Also ist: $uv \neq \varepsilon$.



Es sind:

y aus k1

uyv aus k2 und

w = xuyvz aus **S** abgeleitet

3. Folgerung:

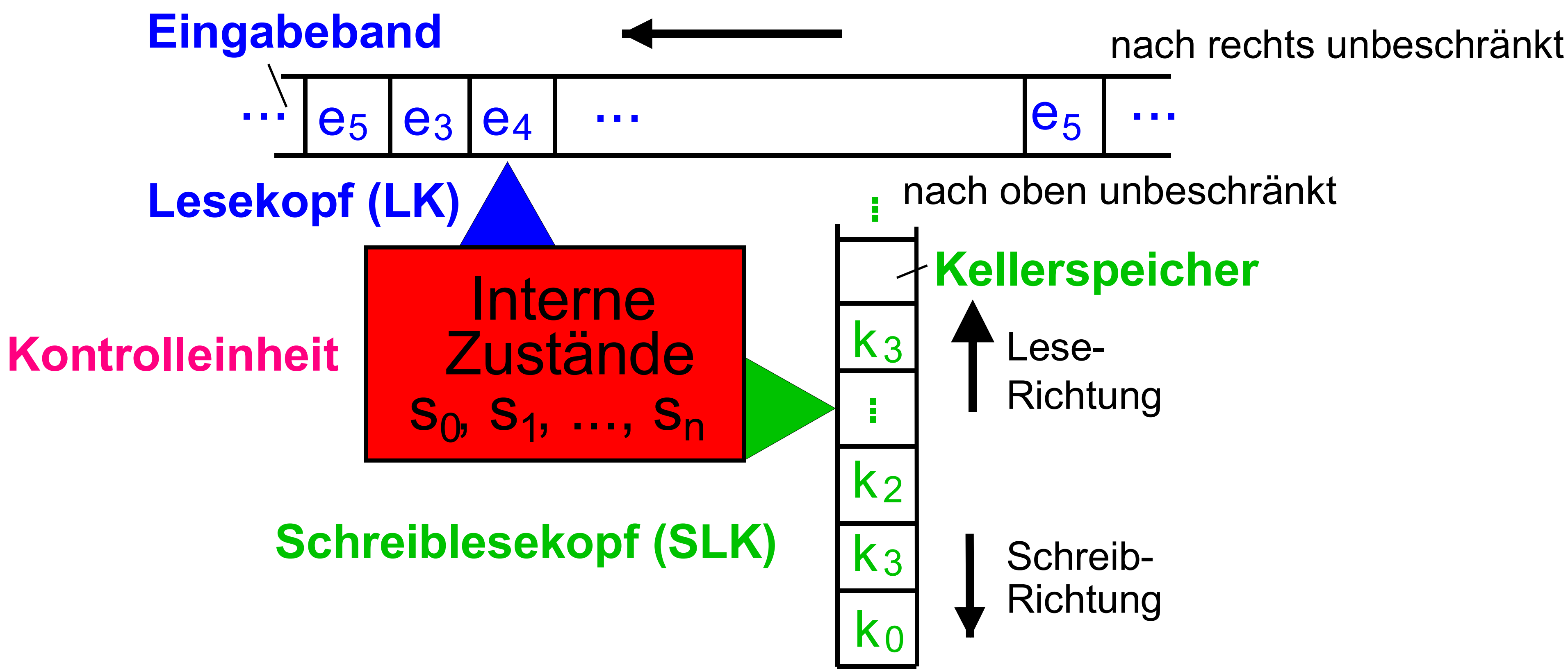
Da man die Ableitungsschritte des Teilbaums ab k1 bereits früher, d. h. bereits ab k2 hätte vornehmen können und diejenigen Ableitungen ab k2 auch ab k1 hätten wiederholt werden können (und zwar beliebig oft) entstehen Wörter der Form: xyz, xuyvz, xuuyvvz, xuuuyvvvz etc. Also ist: $xu^iyv^iz \in L(G)$ für $\mathbf{i} \geq 0$.

bisher:

- Endliche Automaten betrachtet, die dadurch charakterisiert sind, dass die Anzahl ihrer Zustände endlich ist.
- Folglich war ein solcher Automat auch nicht in der Lage, **un**beschränkt viele Informationen zu speichern.

Ausweg:

- Naheliegende Erweiterung besteht darin, einen endlichen Automaten mit einem unbeschränkt großen Speicher zu versehen.
- Dieser Speicher würde es erlauben, die Vergangenheit der Verarbeitung eines Wortes in gewissem Umfang festzuhalten.



Eingabeband
nach rechts unbeschränkt
... e_5 e_3 e_4 ... e_5 ...
Lesekopf (LK)
nach oben unbeschränkt
Kellerspeicher
Kontrolleinheit
Interne Zustände
$s_0, s_1, \ldots, s_n$
k_3
k_2
k_3
k_0
Lese-Richtung
Schreib-Richtung
Schreiblesekopf (SLK)

Ein Verarbeitungsschritt besteht darin, dass der Automat das Zeichen unter dem Lesekopf (LK) und unter dem Schreib-/Lesekopf (SLK) liest und **– in Abhängigkeit vom aktuellen Zustand –** seinen Zustand verändert **sowie das oberste Kellerzeichen durch eine Folge von Zeichen ersetzt**. Anschließend wird der LK um eine Position nach rechts und der SLK auf das oberste Kellerzeichen positioniert.

Definition:

Das Septupel **KA** = (**S**, s_0, **F**, Σ, **K**, k_0, δ) bezeichnet einen deterministischen endlichen Kellerautomaten, wenn für die die einzelnen Komponenten gilt:

- **S** endliche Menge der *internen Zustände* des Automaten
- s_0 interner Anfangszustand, $s_0 \in \mathbf{S}$
- **F** Menge der internen Endzustände, $\mathbf{F} \subseteq \mathbf{S}$
- Σ *endliche* Menge der Eingabezeichen
- **K** endliche Menge der Kellerzeichen k_i
- k_0 Kellerstartzeichen (unterstes Zeichen auf dem Kellerband)
- δ (*deterministische*) Überführungsfunktion mit δ: $\mathbf{S} \times (\Sigma \cup \{\varepsilon\}) \times \mathbf{K} \to \mathbf{S} \times \mathbf{K}^*$

Eigenschaften:

Ist für $s \in \mathbf{S}$, $a \in \Sigma$ und $k \in \mathbf{K}$

$\delta(s, a, k)$ definiert,

so ist

$\delta(s, \varepsilon, k)$ undefiniert.

Damit wird gewährleistet, dass es höchstens eine Möglichkeit gibt, δ anzuwenden.

Ähnlich wie ein endlicher Automat **EA** ist ein **KA** ein Akzeptator, d. h. ein Eingabewort $\mathbf{w} \in \Sigma^*$ wird akzeptiert, wenn sich der Automat nach der Verarbeitung des Wortes in einem **Endzustand** befindet, andernfalls wird es nicht akzeptiert.

Eigenschaften:

Die Überführungsfunktion

$$\delta: \mathbf{S} \times (\Sigma \cup \{\varepsilon\}) \times \mathbf{K} \rightarrow \mathbf{S} \times \mathbf{K}^*$$

bedeutet, dass ein geordnetes Tripel **(s, a, k)** in ein Paar **(s‘, v)** überführt wird, wobei

s‘ = der neue Zustand und

v = ein Wort aus Kellerzeichen ist, durch das das oberste Kellerzeichen ersetzt wird.

Deterministische Kellerautomaten

Ausgangskonfiguration: $(s_0, a_1a_2...a_n, k_0)$

Übergangsverhalten: $\rightarrow$ schrittweise Bearbeitung des Eingabewortes $a_1a_2...a_n$

Solange entweder $\delta(s, a, k)$ für das aktuelle Eingabezeichen $a \in \Sigma$ oder $\delta(s, \varepsilon, k)$ definiert ist, führe aus:

Setze $(s', w) := \delta(s, x, k)$ (mit $x = a$ oder $x = \varepsilon$)
Entferne k aus dem Keller (Pop-Funktion)
Schreibe w in den Keller (Push-Funktion)
Gehe in den internen Zustand s' über
Falls $x \neq \varepsilon$
    gehe ein Zeichen weiter auf dem Eingabeband.

Definition:

Es sei **KA** = (**S**, s_0, **F**, Σ, **K**, k_0, δ) ein Kellerautomat.

Die **Sprache des KA** besteht aus **allen Worten** von Σ*, bei denen sich der Kellerautomat nach der Verarbeitung des letzten Zeichens auf dem Eingabeband in einem internen Endzustand befindet und das Kellerband wieder in der Ausgangsposition steht. Diese Situation bezeichnet man auch als eine

→ **(akzeptierende) Endkonfiguration**: (s_f, , k_0) mit $s_f \in$ **F**

→ d. h. auch, dass das Eingabeband leer ist!

Beispiel: $\Rightarrow$ Deterministischer **KA** für $\mathbf{a^n b^n}$ mit $n > 0$

Wir wollen zeigen, dass der **KA** mit

$$\Sigma = \{a, b\} \qquad \mathbf{S} = \{S_0\, , S_1\} \qquad \mathbf{F} = \{S_1\} \qquad \mathbf{K} = \{k_0, a\}$$

sowie der Zustandsüberführungsfunktion δ

gemäß:

$$\delta(S_0, a, k_0) = (S_0, ak_0) \quad (1) \qquad \delta(S_0, a, a) = (S_0, aa) \quad (2)$$
$$\delta(S_0, b, a) = (S_1, \varepsilon) \quad (3) \qquad \delta(S_1, b, a) = (S_1, \varepsilon) \quad (4)$$

genau die Wörter der Sprache $L(\mathbf{KA}) = \{\mathbf{a^n b^n} \mid n \in \mathbf{IN}\}$ akzeptiert.

Deterministische Kellerautomaten

Konfigurationsfolge beim Akzeptieren von a^3b^3:

1) Schritt 0 = **Startkonfiguration**

2) Schritt 6 = **akzept. Endkonfiguration**

$\delta(S_0, a, k_0) = (S_0, ak_0)$ (1)

$\delta(S_0, a, a) = (S_0, aa)$ (2)

$\delta(S_0, b, a) = (S_1, \varepsilon)$ (3)

$\delta(S_1, b, a) = (S_1, \varepsilon)$ (4)

Schritt	Gl(…)	Eingabeband	Kellerband	Zustand	Konfiguration
0		a a a b b b	k_0	S_0	**(S_0, a a a b b b, k_0)** 1)
1	1	a a a b b b	a k_0	S_0	(S_0, a a b b b, a k_0)
2	2	a a a b b b	a a k_0	S_0	(S_0, a b b b, a a k_0)
3	2	a a a b b b	a a a k_0	S_0	(S_0, b b b, a a a k_0)
4	3	a a a b b b	a a k_0	S_1	(S_1, b b, a a k_0)
5	4	a a a b b b	a k_0	S_1	(S_1, b, a k_0)
6	4	a a a b b b _	k_0	S_1	**(S_1, , k_0)** 2)

4. Das Pumping Lemma
   - 4.1 Pumping Lemma für reguläre Sprachen
   - 4.2 Pumping Lemma für kontextfreie Sprachen
5. Kellerautomaten
   - 5.1 Modellbildung
   - 5.2 Deterministische Kellerautomaten
   - 5.3 Sprache des deterministischen Kellerautomaten
   - **5.4 Nicht-deterministische Kellerautomaten**
   - 5.5 Sprache des Nicht-deterministischen Kellerautomaten
   - 5.6 Kellerautomaten und kontextfreie Grammatiken
   - 5.7 Das Problem der Syntaxanalyse

Definition:

Ein nicht-deterministischer Kellerautomat ist definiert durch ein Septupel **KA** = (**S**, s_0, **F**, Σ, **K**, k_0, δ), wenn alle Komponenten – außer δ – dieselbe Bedeutung haben wie beim deterministischen KA und δ eine *nicht-deterministische* Überführungsfunktion von **S** x (Σ ∪ {ε}) x **K** in die Potenzmenge von **S** x **K*** darstellt. Das bedeutet, dass es Konfigurationen des Kellerautomaten gibt, für die mehrere Nachfolgekonfigurationen existieren.

Die beim deterministischen KA getroffene Einschränkung bzgl. der Definiertheit von δ kann entfallen; man kann δ als eine **totale** Funktion annehmen.

Definition:

Die Sprache des nicht-deterministischen Kellerautomaten KA besteht aus allen Worten aus Σ^*, bei denen es *möglich ist*, daß sich nach dem Lesen des letzten Eingabezeichens der KA in einer das Wort akzeptierenden Endkonfiguration befindet.

Satz:

Im Gegensatz zum endlichen Automaten **EA** besteht zwischen nicht-deterministischen Kellerautomaten und deterministischen Kellerautomaten **keine** Äquivalenz und demzufolge auch keine Überführungsmöglichkeiten.

Satz:

Zu jeder kontextfreien Grammatik G gibt es einen nicht-deterministischen Kellerautomaten KA und umgekehrt mit

$$L(KA) = L(G).$$

Beweis:

Nur für eine Richtung durch Konstruktion des KA zu vorgegebener Grammatik G = (**N**, **T**, **P**, S_G).

Idee:

Die Anwendung von Regeln wird mit Hilfe des Kellerspeichers simuliert. Dabei wird gleichzeitig das Eingabewort w verarbeitet.

<u>Input</u>: (gegeben!)

→ kontextfreie Grammatik $G = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S_G)$.

<u>Output</u>: (gesucht!)

→ nicht-deterministischer $\mathbf{KA} = (\mathbf{S}, s_0, \mathbf{F}, \Sigma, \mathbf{K}, k_0, \delta)$ mit $L(KA) = L(G)$.

<u>Algorithmusbeschreibung</u>:

- Zustände des KA: $\mathbf{S} = \{s_0, s_f\}$
- Eingabezeichen des KA: $\Sigma = \mathbf{T}$ (d. h. Terminals von G)
- Kellerzeichen des KA: $\mathbf{K} = \{k0\} \cup \mathbf{T} \cup \mathbf{N}$
- Endzustand des KA: $\mathbf{F} = \{Sf\}$

- Überführungsfunktion des KA:

$\delta(s_0, \varepsilon, k_0) = (s_f, S_G k_0)$; am Anfang wird das Startsymbol S_G in den Keller geschrieben

$\delta(s_f, \varepsilon, A) = (s_f, \gamma)$ mit $A \in \mathbf{N}$; für jede Regel $A \to \gamma$ von G bzw. $\in$ P, wobei oberstes Kellerzeichen = 1. Zeichen von γ

$\delta(s_f, a, a) = (s_f, \varepsilon)$ mit $a \in \mathbf{T}$; für alle Eingabezeichen bzw. Terminals **T**

Umkehrung:

Zum Nachweis, daß L(G) = L(KA) gilt, geht man davon aus, dass ein Wort der Sprache L(G) auf dem Eingabeband des Kellerautomaten steht und der KA in der Ausgangssituation ist.

Man beachte, dass als "Eingabezeichen" auch ε erlaubt ist, falls auf dem Kellerband ein Nonterminal an oberster Stelle steht. Schritt für Schritt wird vom KA entweder eine Erzeugungsregel für das Eingabewort auf das Kellerband geschrieben oder ein Eingabezeichen abgearbeitet, bis das letzte Zeichen auf dem Eingabeband erreicht ist.

Beispiel: $\Rightarrow$ Konstruktion eines nicht-deterministischen Kellerautomaten **KA**

Gegeben sei die Grammatik G = (**N**, **T**, **P**, S_G) mit **N** = {S_G}, **T** = {a, b}, dem Startsymbol S_G und den beiden Regeln

$$\mathbf{P} = \{ S_G \Rightarrow a\, S_G\, b,\ S_G \Rightarrow \varepsilon \},$$

welche die Sprache L(G) = {$\mathbf{a^n b^n}$ | n $\in$ **IN**} erzeugt.

Gesucht sei der entsprechende **KA** mit dem Startzustand S_0 in der algebraischen Form:

$$\mathbf{KA} = (\mathbf{S}, S_0, \mathbf{F}, \Sigma, \mathbf{K}, k_0, \delta).$$

Lösung:

Für den gesuchten **KA** ergibt sich:

$\Sigma = \{ a, b \}$ **S** $= \{ S_0 , S_f \}$ **F** $= \{ S_f \}$ **K** $= \{ S_G, k_0, a, b \}$

und die Zustandsüberführungsfunktion δ gemäß:

$\delta(S_0, \varepsilon, k_0) = (S_f, S_G\, k_0)$ (1)

$\delta(S_f, \varepsilon, S_G) = (S_f, a\, S_G\, b) \mid (S_f, \varepsilon)$ (2), (2')

$\delta(S_f, a, a) = (S_f, \varepsilon)$ (3)

$\delta(S_f, b, b) = (S_f, \varepsilon)$ (4)

Konfigurationsfolge beim Akzeptieren von **aabb**:

$\delta(S_0, \varepsilon, k_0) = (S_f, S_G\, k_0)$ (1)

$\delta(S_f, \varepsilon, S_G) = (S_f, a\, S_G\, b) \mid (S_f, \varepsilon)$ (2), (2')

$\delta(S_f, a, a) = (S_f, \varepsilon)$ (3)

$\delta(S_f, b, b) = (S_f, \varepsilon)$ (4)

$(S_0, aabb, k_0) \Rightarrow_{(1)} (S_f, aabb, S_G\, k_0) \Rightarrow_{(2)} (S_f, aabb, a\, S_G\, b\, k_0)$

$\Rightarrow_{(3)} (S_f, abb, S_G\, b\, k_0) \Rightarrow_{(2)} (S_f, abb, a\, S_G\, b\, b\, k_0)$

$\Rightarrow_{(3)} (S_f, bb, S_G\, b\, b\, k_0) \Rightarrow_{(2')} (S_f, bb, b\, b\, k_0)$

$\Rightarrow_{(4)} (S_f, b, b\, k_0) \Rightarrow_{(4)} (S_f, \ , k_0)$ = **akzeptierter Endzustand**

aabb $\in$ L(**KA**) = L(G) q. e. d.

- höhere Programmiersprachen können als kontextfreie Sprachen angesehen werden
- syntaktische Korrektheit kann auf das Wortproblem bei formalen Sprachen zurückgeführt werden
- Fragestellung: gehört ein gegebenes Wort $\mathbf{w} \in \mathbf{T^*}$ zur kontext-freien Sprache L(G)?

Merke:

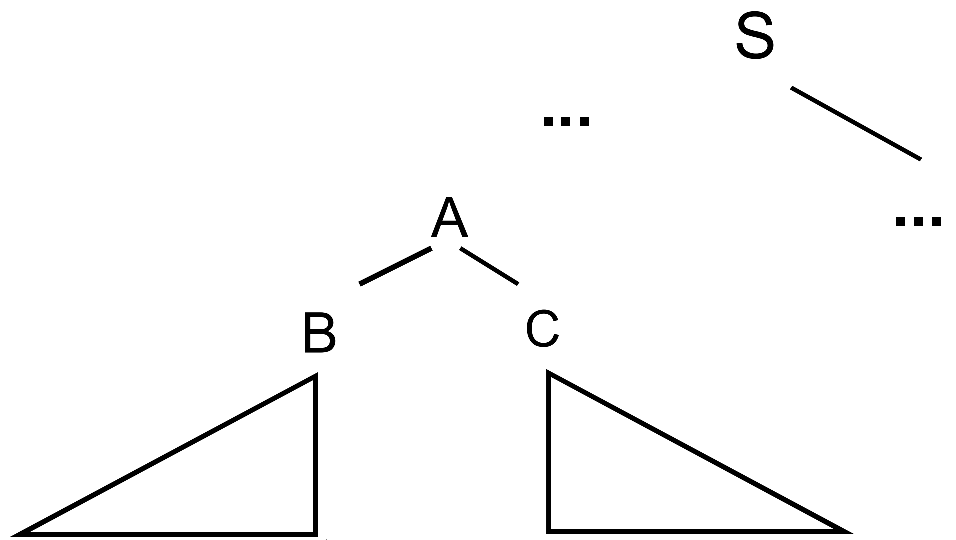
Zur Grammatik einer höheren Programmiersprache gibt es i. a. keinen zugehörigen deterministischen Kellerautomaten

$\rightarrow$ infolge dessen ist auch kein deterministisches Analyseverfahren herleitbar $\rightarrow$ theoretische Lösung ist das **CYK**-Verfahren

- Wenn ein Wort **$w = a$** der Länge 1 abgeleitet werden kann, dann sicherlich nur aufgrund der Regel **$A \to a$**.
- Ist aber **$w = a_1 a_2 \ldots a_n$** ($n \geq 2$), dann kann **w** aus **A** nur deshalb ableitbar sein, weil eine Regel der Form **$A \to BC$** angewandt worden ist.
- Von **B** aus wird dann ein Anfangsstück von **w** abgeleitet und von **C** aus das Endstück.

Folgerung:

Es muss also ein **k** mit **$1 \leq k \leq n$** geben, so dass gilt:



$$\mathbf{w} = \underbrace{a_1\, a_2 \ \dots \ a_k}\quad \underbrace{a_{k+1} \ \dots \ a_n} := \mathbf{w_1\, w_2}$$

⇒ Damit ist es möglich, das Wortproblem für **w** der Länge **n** auf zwei entsprechende Entscheidungen für die Wörter $\mathbf{w_1}$ der Länge **k** und $\mathbf{w_2}$ der Länge **n-k** zurückzuführen.

Beschreibung des Algorithmus:

- Grammatik sei in der **Chomsky-Normalform** gegeben, d.h. es gibt nur Produktionen der Form $\mathbf{A \rightarrow a}$ oder $\mathbf{A \rightarrow BC}$.
- Sei $\mathbf{w = a_1a_2 \ldots a_n}$ zu untersuchen.
- Betrachte Teilwort $a_ia_{i+1}\ldots a_j$ mit $1 \leq i \leq j \leq n$.
- Fasse alle Nonterminals A, aus denen $a_ia_{i+1}\ldots a_j$ ableitbar ist zur Menge $M_{ij} = \{ A \mid A \Rightarrow a_ia_{i+1}\ldots a_j \}$ zusammen.
- Dann gilt: $\mathbf{w \in L(G)} \quad \Leftrightarrow \quad \mathbf{M_{1,n}}$ **enthält S**.

Konstruktion der Mengen $M_{i,j}$ - es gibt n(n+1)/2 davon - schrittweise:

0. Stufe: i=1,2,...n; j=i; $M_{1,1} = \{ A \mid A \to a_1\}$, $M_{2,2} = \{ A \mid A \to a_2\}$, $M_{3,3} = \dots$

1. Stufe: i=1,2,...n-1; j=i+1;

$M_{1,2} = \{ A \mid A \to a_1a_2\} = \{ A \to BC \mid B \in M_{1,1}$ und $C \in M_{2,2} \}$;

$M_{2,3} = \dots$

....

s. Stufe: i=1,2,...n-s; j=i+s: $M_{1,s+1} = \{ A \mid A \to a_1a_2 \dots a_{s+1} \}$

$= \{ A \to BC \mid B \in M_{1,k}$ und $C \in M_{k+1,s+1}$; $k = 1,2, .. s \}$; ...

Entscheidung nach **$O(n^3)$** Zeit-Schritten, ob **$S \in M_{1,n}$**

CYK-Algorithmus: // Eingabewort sei $w = a_1 a_2 \dots a_n$

FOR $i = 1, \dots, n$ **DO**

$M[i, i] := \{A \in M: \exists A \to a_i\}$;

ENDDO

FOR $s = 1, \dots, n - 1$ **DO**

FOR $i = 1, \dots, n - s$ **DO**

$M[i, i + s] := \{ \}$;

FOR $k = i, \dots, i + s - 1$ **DO**

$M[i, i + s] := M[i, i + s] \cup \{ A \in M: \exists A \to BC$ mit $B \in M[i, k] \wedge C \in M[k + 1, i + s] \}$;

Fortsetzung:

```
            ENDDO
        ENDDO
ENDDO
IF S ∈ M[1, n] THEN
        RETURN "JA, w ∈ L(G)."
ELSE
        RETURN "NEIN, w ∉ L(G)."
ENDIF
```

Dreieckstabelle:

A → □= = = = = = = **B** = = = = = = = >

$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{1,4}$	…	$\mathbf{M_{1,n}}$	
↑ $\mathbf{a_1}$	$M_{2,2}$	$M_{2,3}$	$M_{2,4}$	…	$M_{2,n}$	□
	↑ $\mathbf{a_2}$	$M_{3,3}$	$M_{3,4}$	…	$M_{3,n}$	‖ ‖
		↑ $\mathbf{a_3}$	$M_{4,4}$	…	$M_{4,n}$	**C**
			↑ $\mathbf{a_4}$	…		‖ ‖
				…	$M_{n,n}$	V
					↑ $\mathbf{a_n}$	

Beispiel: L(**G**) = {$\mathbf{a^n b^n}$ | n $\in$ **IN**} und w = **aabb**

S $\rightarrow$ AC | AB ; C $\rightarrow$ SB ; A $\rightarrow$ a ; B $\rightarrow$ b

A $\rightarrow$ = = = = **B** = = = >

{A}	{-}	{-}	**{S}**	
↑ **a**	{A}	{S}	{C}	‖ **C** ‖
	↑ **a**	{B}	{-}	V
		↑ **b**	{B}	
			↑ **b**	

Der **CYK**-Algorithmus terminiert also mit der Antwort "**JA**" $\Rightarrow$ w $\in$ L(**G**).

Prinzipien der Syntaxanalyse

- Abarbeitung des Wortes bzw. des Programms grundsätzlich von links nach rechts
- Parsing-Verfahren:
  - bottom-up → Ableitungsbaum von unten nach oben aufgebaut
  - top-down → Ableitungsbaum von oben nach unten aufgebaut
  - oder gemischt

Automatentheorie und Formale Sprachen

– LV 4110 –

Problemklassen und Komplexitätstheorie

- Effizienz und Laufzeit von Programmen
- Herausstellen von Komplexitätsklassen
- Laufzeitkomplexität und Problemgröße
- Definitionen der Klassen P und NP
- Erläuterung und Einordnung des Knapsack-Problems
- Definition und Interpretation der NP-Vollständigkeit
- Erläuterung und Einordnung des Erfüllbarkeitsproblems (SAT)
- Lösungsalgorithmen für NP-harte Probleme

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel

2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Gibt es einen Effizienzgewinn beim Übergang zu „mächtigeren“ Rechenmodellen als TM oder RAM

bzw.

wie effizient kann ein Problem grundsätzlich gelöst werden ?

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. **Herausstellen von Komplexitätsklassen**
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Hintergründe:

- Wir haben **Entscheidungsprobleme**, **Berechenbarkeitsprobleme**, **Approximationsprobleme**, **Erfüllbarkeitsprobleme**, **Aufzählungsprobleme**, **Suchprobleme** bis hin zu **Optimierungsproblemen** zu lösen.
- Dazu stehen uns **Erkennungsprozeduren**, **Such- und Findeprozeduren**, **Prüf- und Entscheidungsprozeduren** sowie **Verarbeitungsprozeduren** zur Verfügung.
- In diesem Zusammenhang interessieren wir uns im allgemeinen für die **Laufzeit von Algorithmen** (Berechnungen) bzw. für die **„Größe“ von Problemen**.

Fragestellungen:

- In welchem Zusammenhang stehen Sprachen und Probleme? Gibt es überhaupt einen Zusammenhang?
- Gibt es eine Korrespondenz zwischen Entscheidungs-, Berechenbarkeits- und Optimierungsproblemen?
- Gibt es womöglich unterschiedliche Problemklassen?
- Gibt es für jedes Problem einen Lösungsalgorithmus bzw. existiert zu jedem Problem überhaupt eine Lösung?
- Gibt es ein Beweissystem (z. B. aus Axiomen und Schlussregeln), mit dem man feststellen kann, ob ein Problem algorithmisch (un)lösbar ist?

Hinter diesen Fragestellungen verbirgt sich eine der wichtigsten offenen Fragen der theoretischen Informatik:

Ist **P ≠ NP?**

Mit anderen Worten:

- Besitzen deterministische Turing-Maschinen eine andere **Zeitkomplexität** als nicht deterministische Turing-Maschinen?
- Erzielen nicht deterministische Turing-Maschinen eine höhere **Aussagekraft** als deterministische Turing-Maschinen?

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. **Laufzeitkomplexität und O-Notation**
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Hintergrund:

Ziel ist es, einen möglichst **effizienten Algorithmus** für ein bestimmtes Problem zu finden.

Definition (Rechenzeit):

Sei **TM** eine deterministische Turingmaschine auf dem Eingabealphabet Σ. Dann ist die worst case Rechenzeit $\mathbf{t_{TM}(n)}$ die maximale Anzahl von Rechenschritten, die **TM** auf Eingaben aus Σ^n macht.

Definition (Zeitkomplexitätsfunktion $T_{TM}(n) : \mathbb{N} \to \mathbb{N}$):

$\mathbf{T_{TM}(n) = \max\{m\}}$, so dass eine deterministische Turing-Maschine **TM** bei Eingabe $\mathbf{x \in \Sigma^n}$ **m** Berechnungsschritte (Übergänge) benötigt, bis ein Endzustand erreicht wird.

Definition (O-Notation):

Für eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ definieren wir

$$f(n) = O(g(n)) = \{ f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists\, c, n_0 : \forall\, n \geq n_0 : f(n) \leq c \cdot g(n) \}$$

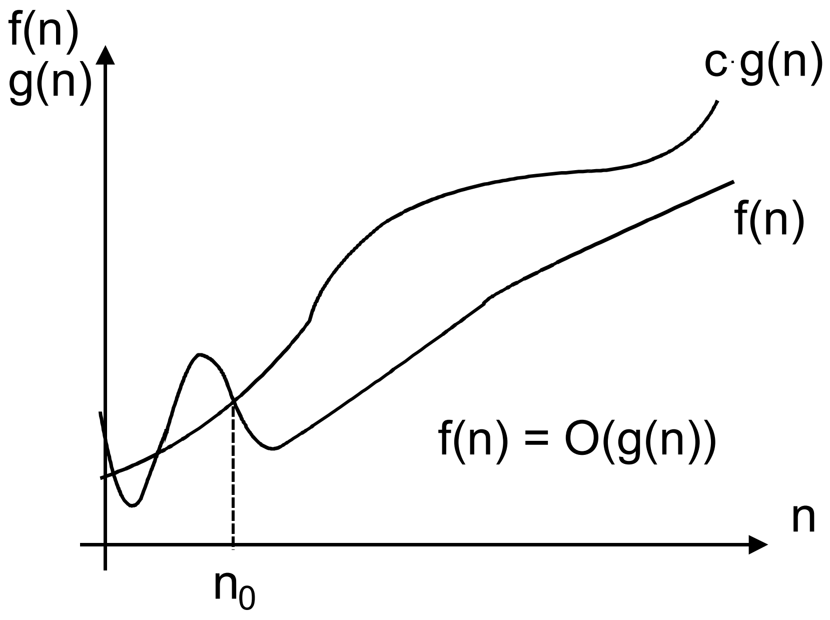
und können so die Laufzeitfunktion f(n) durch eine asymptotische obere Schranke g(n) begrenzen $\Rightarrow$ in Worten: f(n) wächst nicht schneller als g(n)

Die **Komplexität** eines Algorithmus ist O(g(n)), wenn die Laufzeit $T_{TM}(n)$ in O(g(n)) ist, beispielsweise geschrieben als: $f(n) = 2 \cdot n^2 + 4 \cdot n \in O(n^2)$

Dabei bezeichnen:

f(n), g(n)	Zeitkomplexitätsfunktionen (positiv)
n	Eingabelänge
c, n_0	positive Konstanten

Verläufe und Beispiele:



Fehlerterm bei der Approximation:

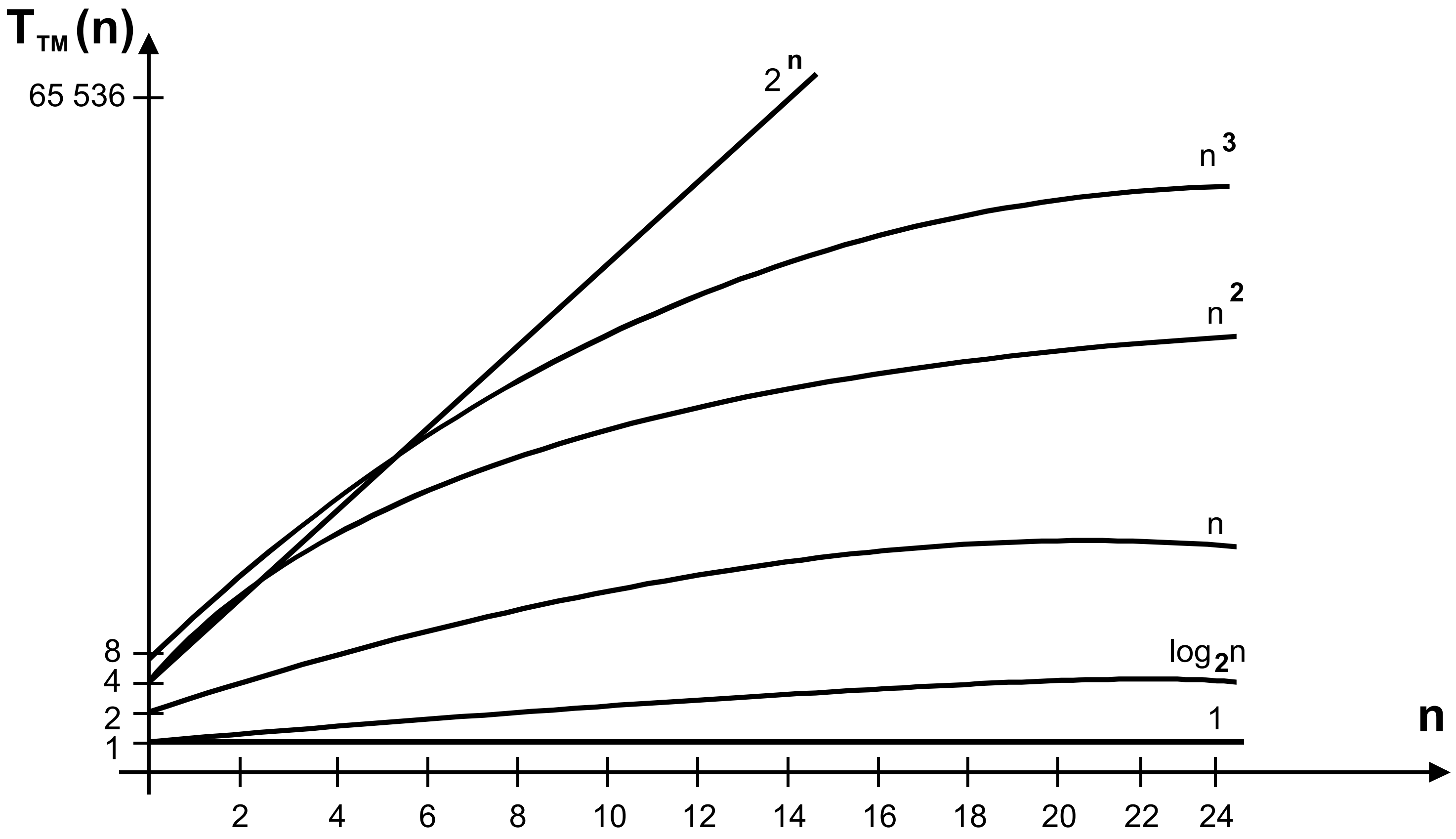
$e^x = 1 + x + x^2 / 2 + O(x^3)$ für $\forall\, x \to 0$ drückt aus, dass die vernachlässigten Summanden höherer Ordnung kleiner sind als der konstante Wert x^3, wenn x nur klein genug ist.

CPU Time eines Algorithmus:

$$T_{TM}(n) = f(n) = 10 \log(n) + 5 (\log(n))^3 + 7 n + 3 n^2 + 6 n^3$$

$\Rightarrow f(n) = O(n^3)$

Verläufe (Wachstumsraten):



Beispiel:

Ein Beispiel für die Größenordnung bzw. Hierarchie ist die folgende Anordnung:

1	$\log n$	$n \quad n^2 \quad n^3 \quad \ldots \quad n^k$	2^n
↑	↑	↑	↑
konstant	logarithmisch	polynomiell	exponentiell

Satz:

Sei **k** eine Konstante und **p** ein Polynom. Ein Algorithmus ist **polynomiell**, wenn seine Zeitkomplexität $T_{TM}(n) \in O(n^k)$ ist. Ein Algorithmus ist dagegen **exponentiell**, wenn seine Laufzeit $T_{TM}(n) \in O(2^{p(n)})$ ist.

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. **Definitionen der Klassen P und NP**
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Definition (Klasse P):

P ist eine Klasse der Probleme, für die es eine deterministische Turing-maschine **TM** gibt, deren worst case Rechenzeit $\mathbf{t_{TM}(n)}$ polynomiell beschränkt ist, d. h. es existiert ein Polynom **p** mit

$$\mathbf{T_{TM}(n) \leq p(n)}$$

Definition (Klasse NP):

NP (nicht deterministisch polynomiell) ist die Klasse der Entscheidungs-probleme, für die es eine nicht deterministische Turingmaschine **NTM** gibt, deren worst case Rechenzeit $\mathbf{t_{NTM}(n)}$ polynomiell beschränkt ist, d. h.

$$\mathbf{T_{NTM}(n) \leq p(n)}$$

Beispiel:

Das Entscheidungsproblem **PRIMES** besteht darin, zu entscheiden, ob es sich bei einer gegebenen natürlichen Zahl **z** > 1 um eine Primzahl handelt. Dabei sei die Zahl **z** zur Basis $\mathbf{b} \in \mathbf{IN}$ dargestellt.

Die dazugehörige Sprache sei mit $\mathbf{L_b}$ = **L [PRIMES, b]** bezeichnet.

Satz:

Sei $\mathbf{L_1}$ **:= L [PRIMES, 1]**. Erst 2002[1)] konnte gezeigt werden, dass gilt:

$\mathbf{L_1}$ liegt in P

d. h. es gibt eine DTM, deren Laufzeit von der Ordnung $\mathbf{O(n^3)}$ und damit polynomial beschränkt ist.

1) Drei indische Mathematiker: M. Agrawal, N. Kayal und N. Saxena

Beispiel:

Ein bedeutender Algorithmus ist der **Euklidische Algorithmus** zur Berechnung des größten gemeinsamen Teilers (kurz **ggT**(n, m)).

Satz:

Der Euklidische Algorithmus zur Berechnung des **ggT** zweier natürlicher Zahlen n und m ist **polynomial** und damit **effektiv berechenbar**.

Genauer gesagt gilt für seine Laufzeit:

$$\mathbf{T(n, m) = O\left((\log_2 n + \log_2 m)^2\right)}$$

Beispiel:

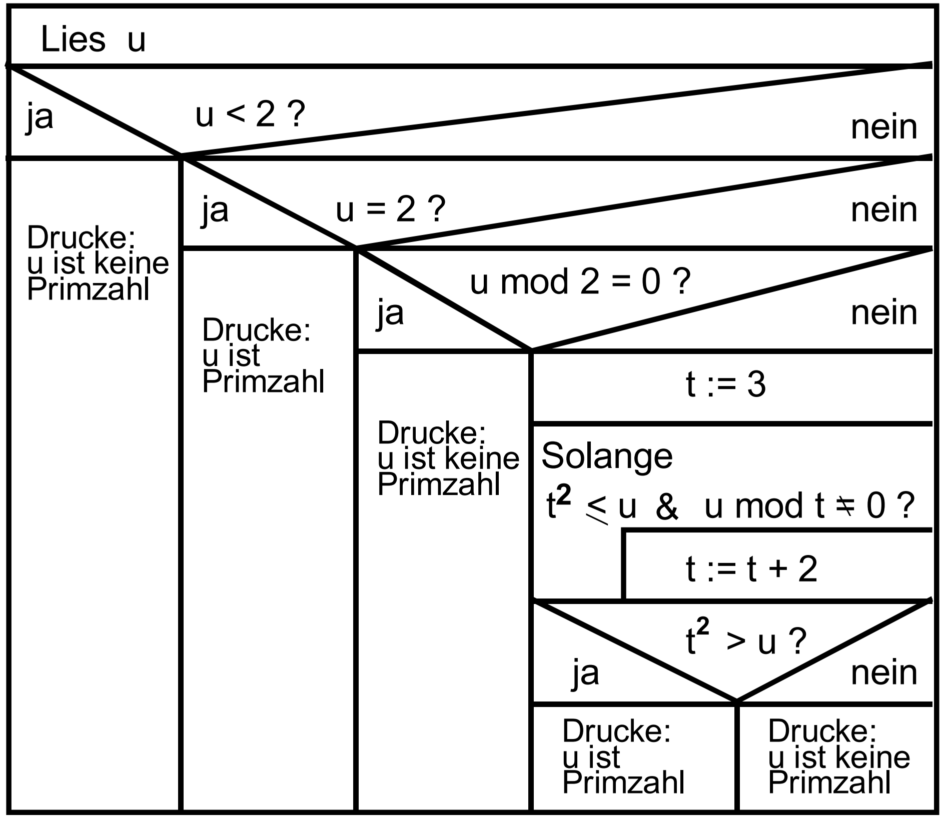
Liefert ein Entscheidungsalgorithmus zu der Frage, ob n zusammengesetzt (also beispielsweise $n = p \cdot q$) ist, als Beleg der Antwort „JA“ einen Faktor q von n, so lässt sich in polynomialer Zeit nachprüfen, ob $q \mid n$.

Es gilt nämlich: $q \mid n \quad \text{gdw} \quad (\exists\, k \in \mathbf{Z}) : n = k \cdot q \quad \Leftrightarrow \quad \mathrm{ggT}(q, n) = |\, q \,|$

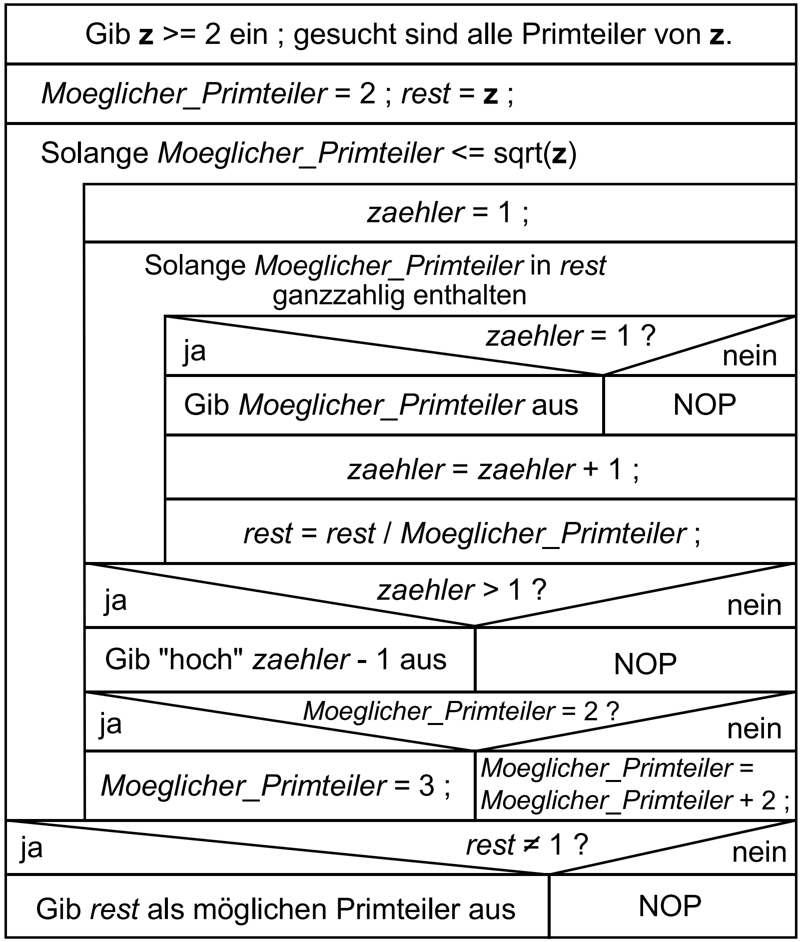
Das **Faktorisierungsproblem** (d. h. die Zerlegung von n in ihre Primfaktoren) ist also in **NP**, aber man weiß nicht, ob es in **P** ist.

Satz:

Die Klasse **NP** besteht aus denjenigen Entscheidungsalgorithmen, bei denen es einen Algorithmus gibt, der im Falle einer **JA**-Antwort durch den Algorithmus des Entscheidungsproblems die Korrektheit der Antwort in polynomialer Zeit testet (d. h. Antwort plus Beleg).



Lies u
ja
u < 2 ?
nein
Drucke: u ist keine Primzahl
ja
u = 2 ?
nein
Drucke: u ist Primzahl
u mod 2 = 0 ?
ja
nein
Drucke: u ist keine Primzahl
t := 3
Solange
t² ≤ u & u mod t ≠ 0 ?
t := t + 2
t² > u ?
ja
nein
Drucke: u ist Primzahl
Drucke: u ist keine Primzahl



Gib z >= 2 ein ; gesucht sind alle Primteiler von z.
Moeglicher_Primteiler = 2 ; rest = z ;
Solange Moeglicher_Primteiler <= sqrt(z)
zaehler = 1 ;
Solange Moeglicher_Primteiler in rest ganzzahlig enthalten
zaehler = 1 ?
ja
nein
Gib Moeglicher_Primteiler aus
NOP
zaehler = zaehler + 1 ;
rest = rest / Moeglicher_Primteiler ;
zaehler > 1 ?
ja
nein
Gib "hoch" zaehler - 1 aus
NOP
Moeglicher_Primteiler = 2 ?
ja
nein
Moeglicher_Primteiler = 3 ;
Moeglicher_Primteiler = Moeglicher_Primteiler + 2 ;
rest ≠ 1 ?
ja
nein
Gib rest als möglichen Primteiler aus
NOP

VII. Problemklassen und Komplexitätstheorie

Definition (KNAPSACK **KP**):

Gegeben sei ein Rucksack und **n** Objekte mit Gewichten $g_1, g_2, ..., g_n \in \mathbb{N}$ sowie eine Gewichtsschranke **G**. Zusätzlich seien $a_1, a_2, ..., a_n \in \mathbb{N}$ die Nutzenwerte für die Objekte. Bei $x_i = 1$ wird ein Objekt i eingepackt und bei $x_i = 0$ nicht.

Variante 1: Gibt es zu einem gegebenen Nutzenwert **A** eine Bepackung des Rucksackes, die das Gewichtslimit **G** respektiert und mindestens den Nutzen **A** erreicht?

Also ob: $a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_n \cdot x_n \geq \mathbf{A}$

unter der Nebenbedingung: $g_1 \cdot x_1 + g_2 \cdot x_2 + ... + g_n \cdot x_n \leq \mathbf{G}$

Variante 2: Berechne den **größten erreichbaren Nutzen A**!

Variante 3: Berechne eine **optimale Bepackung** des Rucksackes!

Satz:

Die Entscheidungsvarianten 1 bis 3 von KP sind in **NP**.

Satz:

Trivialerweise gilt:

$$\mathbf{P} \subseteq \mathbf{NP}$$

Satz:

Für jede Sprache **L** $\in$ **NP** gibt es ein Polynom **p** und eine **deterministische** Turingmaschine **TM**, so dass **TM** die Sprache **L** in

exponentieller Zeit $2^{p(n)}$ akzeptiert.

Dabei ist:

n = Länge der Eingabesymbole (aus Σ^n)

Satz:

KP ist NP-vollständig[1)].

1) also mindestens so komplex, wie jedes andere Problem in **NP**.

Algorithmus:

Die Lösung des KP-Problems führt auf die sog. Bellmansche Optimalitäts-gleichung.

Satz:

Das Rucksack-Problem (KP) kann in der Zeit ($\rightarrow$ pseudopolynomiell)

$$O(nG)$$

gelöst werden (**n** = Anzahl der Objekte, **G** = Gewichtsschranke).

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. **NP-Vollständigkeit**
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. NP-harte Probleme
9. Problemstellung der PARTITION

Definition (Polynomiell reduzierbar):

Es seien L_1 und L_2 Sprachen über Σ_1 bzw. Σ_2. Dann heißt L_1 polynomiell auf L_2 reduzierbar,

Notation: $\mathbf{L_1} \leq_p \mathbf{L_2}$ (**p** steht für **polynom**iell)

wenn es eine polynomielle Transformation von L_1 nach L_2 gibt, d. h. wenn es eine von einer deterministischen Turingmaschine **TM** in polynomieller Zeit **berechenbare** Funktion

$$\mathbf{f} : \Sigma^*_1 \rightarrow \Sigma^*_2$$

gibt, so dass für alle $w \in \Sigma^*_1$ gilt:

$$w \in \mathbf{L}_1 \Leftrightarrow \mathbf{f}(w) \in \mathbf{L}_2$$

Beispiel:

Seien $\mathbf{L_1}$ die Sprache der ungeraden Zahlen in Dezimaldarstellung über $\Sigma_1 = \{0, 1, ..., 9\}$ und $\mathbf{L_2}$ die Sprache der Wörter gerader Länge über dem Alphabet $\Sigma_2 = \{a, b\}$.

Dann kann $\mathbf{L_1}$ polynomial in $\mathbf{L_2}$ transformiert werden, d. h. $\mathbf{L_1} \leq_p \mathbf{L_2}$.

Beweisidee:

Sei $f : \Sigma_1^* \rightarrow \Sigma_2^*$ definiert als

$$f(z) := \begin{cases} \mathbf{aa}, & \text{falls } z \text{ ungerade} \\ \mathbf{a}, & \text{sonst} \end{cases}$$

Dann kann f von einer DTM berechnet werden.

Fortsetzung der Beweisidee:

Eine solche DTM erhält als Eingabe eine Dezimalzahl z. Zunächst löscht sie alle Zeichen der Eingabe bis auf das Letzte. Wenn das letzte Zeichen der Eingabe ungerade ist, schreibt sie **aa**, ansonsten **a** auf das Band. Die Laufzeit der DTM ist hierbei im Wesentlichen **logarithmisch** in der Größe der Eingabe. Damit ist die Laufzeit polynomial beschränkt.

Sei $z \in \Sigma_1^*$. Dann ist $f(z) \in \mathbf{L_2} \Leftrightarrow f(z) = aa \Leftrightarrow z$ ist ungerade $\Leftrightarrow z \in \mathbf{L_1}$.

Damit ist f eine polynomiale Transformation von $\mathbf{L_1}$ in $\mathbf{L_2}$.

Definition (NP-Vollständigkeit):

Eine Sprache **L** heißt NP-vollständig, wenn **L** $\in$ **NP** ist und für alle **L'** $\in$ **NP** gilt:

$$L' \leq_p L$$

Interpretation:

L ist also **NP**-vollständig, wenn **L** selber zu **NP** gehört und jedes Problem in **NP** bzgl. $\leq$p nicht schwieriger als **L** ist. **NP**-vollständige Probleme müssen also selber in **NP** enthalten sein.

$$\text{NP-vollständig} \subseteq \text{NP}$$

NP-vollständige Probleme sind also **mindestens so komplex**, wie jedes andere Problem in **NP**.

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. **Das Erfüllbarkeitsproblem (SAT und 3SAT)**
8. NP-harte Probleme
9. Problemstellung der PARTITION

Bemerkungen:

Falls irgendwann jemand einen polynomiellen Lösungsalgorithmus für ein **NP**-vollständiges Problem **L** finden würde, dann hätte man einen polynomiellen Lösungsalgorithmus für **jedes** Problem in **NP**.

Es würde dann gelten, dass

$$\mathbf{P = NP}$$

(eines der z. Z. größte offene Problem der theoretischen Informatik!)

Obwohl man bisher noch **nicht** nachweisen konnte, dass es keinen polynomiellen Lösungsalgorithmus für **NP**-vollständige Probleme gibt, konnte man für eine Reihe wichtiger Probleme zeigen, dass sie **NP**-vollständig sind.

Das bekannteste Beispiel ist das **Erfüllbarkeitsproblem**, kurz **SAT**.
(**SAT** = **Sat**isfiability)

Definition (SAT):

Sei $\mathbf{X} = \{ x_1, x_2, ..., x_m \}$ eine Menge von **booleschen** Variablen (x_i und $\bar{x}_i$ heißen auch **Literale**). Eine **Wahrheitsbelegung** von **X** ist eine Funktion

$$\mathbf{f} : \mathbf{X} \to \{ \text{wahr, falsch} \}.$$

Eine **Klausel** k ist ein **boolescher** Ausdruck der Form

$$y_{k1} \vee y_{k2} \vee \dots \vee y_{ks}$$

(d. h. Disjunktion von Literalen) mit

$$y_{ki} \in \{ x_1, x_2, ..., x_m \} \cup \{ \bar{x}_1, \bar{x}_2, ..., \bar{x}_m \} \cup \{ \text{wahr, falsch} \}.$$

Dann ist **SAT** wie folgt definiert:

Existiert eine **Wahrheitsbelegung** von **X**, so dass **alle Klauseln** (k = 1, 2, ... n) den Wahrheitswert **wahr** annehmen?

Beispiel:

Gegeben seien eine Menge von Variablen **X** = { x_1, x_2 } und eine Menge von Klauseln **C** = { c_1, c_2 } über **X** gegeben mit

$$c_1 = x_1 \vee \overline{x}_2 \quad \text{und} \quad c_2 = \overline{x}_1 \vee x_2 \ .$$

Mit der Wahrheitsbelegung

$$f(x_1) = f(x_2) = \text{wahr}$$

(d. h. $x_1 = x_2 = 1$) wird **C** erfüllt.

Satz (von Steven Cook, 1971):

SAT ist **NP**-vollständig.

Definition (3SAT):

Gegeben: Eine Menge **X** von Variablen und eine Menge **C** von Klauseln **C**, wobei jede Klausel **genau drei Literale** enthält.

Frage: Existiert eine erfüllende Wahrheitsbelegung für **C**?

Satz:

Das Problem **3SAT** ist **NP**-vollständig.

Anmerkung:

Um zu zeigen, dass ein Problem **NP**-vollständig ist, wird häufig **3SAT** auf das Problem reduziert.

Beispiel (3SAT):

Gegeben seien eine Menge von Variablen $\mathbf{X} = \{ x_1, x_2, x_3 \}$ und eine Menge von Klauseln $\mathbf{C} = \{ c_1, c_2, c_3, c_4 \}$ über $\mathbf{X}$, wobei $m = |\mathbf{X}| = 3$ und $n = |\mathbf{C}| = 4$.

$$c_1 = x_1 \vee x_2 \vee x_3$$
$$c_2 = x_1 \vee \bar{x}_2 \vee x_3$$
$$c_3 = x_1 \vee \bar{x}_2 \vee \bar{x}_3$$
$$c_4 = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$$

Mit der Wahrheitsbelegung

$$f(x_1) = f(x_2) = \text{wahr} \quad \text{und} \quad f(x_3) = \text{falsch},$$

d. h. $(x_1 , x_2 , x_3) = (1 , 1 , 0)$, wird $\mathbf{C}$ erfüllt.

VII. Problemklassen und Komplexitätstheorie

1. Intension in diesem Kapitel
2. Herausstellen von Komplexitätsklassen
3. Laufzeitkomplexität und O-Notation
4. Definitionen der Klassen P und NP
5. Erläuterung und Einordnung des Knapsack-Problems
6. NP-Vollständigkeit
7. Das Erfüllbarkeitsproblem (SAT und 3SAT)
8. **NP-harte Probleme**
9. Problemstellung der PARTITION

Vorbemerkung:

Wenn auch bislang der Versuch eines Beweises für **N = NP** nicht gelungen ist, so konnte man in der Vergangenheit für manche Probleme jedoch schon zeigen, dass man damit jedes andere Problem in **NP** lösen kann. Offensichtlich gelang es dabei aber noch nicht zu zeigen, dass diese Probleme selbst auch in **NP** sind.

Für diese Klasse von Problemen führte man die Bezeichnung **NP** hart ein.

Definition (NP hart):

Eine Sprache **L** heißt NP-hart, wenn für alle **L'** $\in$ **NP** gilt:

$$L' \leq_p L$$

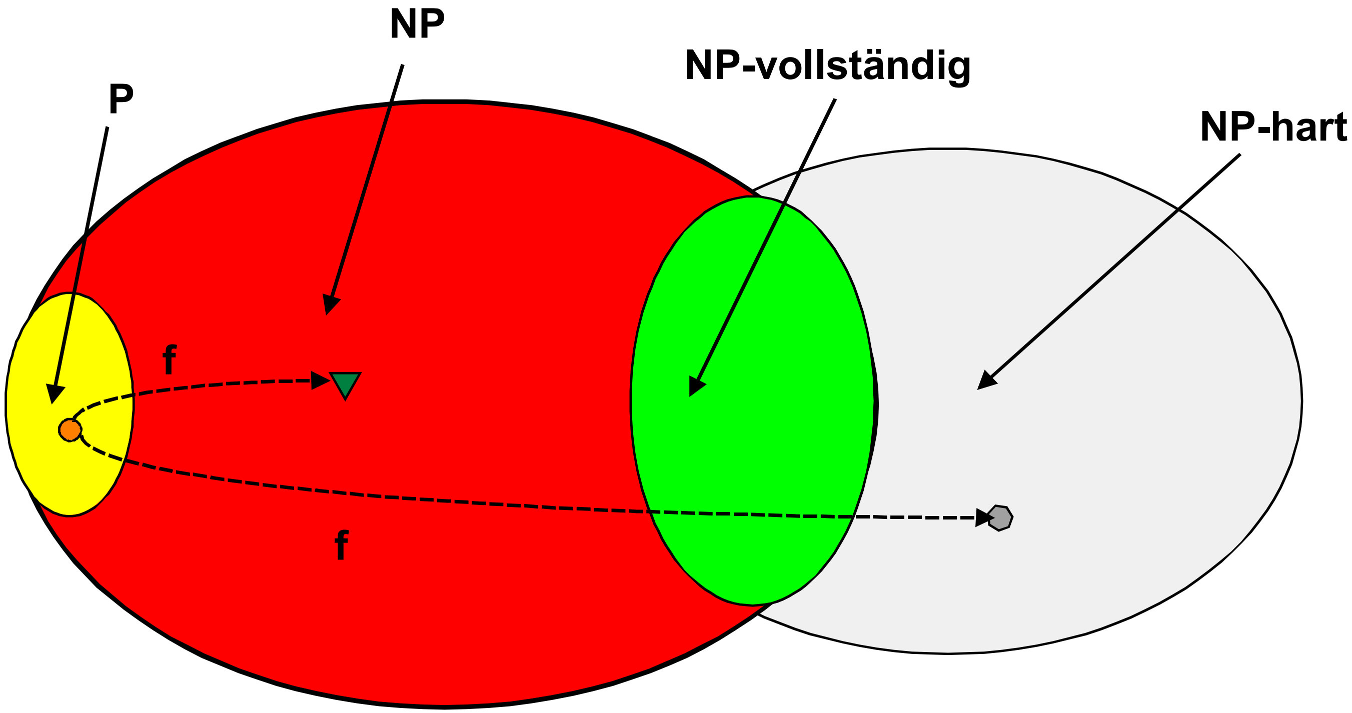
Interpretation:

Ein Problem, das **NP** hart ist, muss selbst nicht notwendigerweise in **NP** enthalten sein. Wir nennen ein Problem **NP** hart, wenn es **mindestens** so schwer ist, wie alle **NP**-vollständigen Probleme.

Klar ist dagegen, dass ein **NP**-vollständiges Problem immer auch **NP** hart ist.

$$\textbf{NP vollständig} \subseteq \textbf{NP hart}$$

Zusammenhänge: **P** $\subseteq$ **NP** und **NP-vollständig** $\subseteq$ **NP**-hart



f
-------> = polynomial transformierbar

f ist transitiv

VII. Problemklassen und Komplexitätstheorie

Definition (PARTITION):

Gegeben sind $b_1, b_2, ..., b_n \in$ **IN**. Gibt es eine Teilmenge **K** $\subseteq \{1, 2, ..., n\}$, so dass die Summe aller b_k, $k \in$ **K**, gleich der Summe aller b_j, $j \notin$ **K** ist?

Satz:

PARTITION ist NP-vollständig.

Beweis:

PARTITION $\in$ **NP**, da wir **K** raten können. Das vollständige Raten (vollständige Aufzählung) entspricht dabei dem **Nicht-Determinismus**.

Beispiel (PARTITION):

Gegeben sei die Zahlenmenge (b_1, b_2, ..., b_8) = (2, 4, 5, 6, 7, 9, 10, 13).

Die gesuchte Teilmenge **K** $\subseteq$ { 1, 2, ..., 8 } ist dann gleich { 1, 2, 4, 5, 6 }, weil:

$$b_1 + b_2 + b_4 + b_5 + b_6 = b_3 + b_7 + b_8 = 28$$

Automatentheorie und Formale Sprachen

– LV 4110 –

Entscheidbarkeit und Berechenbarkeit

- Verwendung von unterschiedlichen Rechnermodellen
- Kennenlernen der Begriffe: Berechenbarkeit und Entscheidbarkeit
- Definitionen für Entscheidungsverfahren und Aufzählverfahren
- Klärung, was man unter einer berechenbaren Funktion versteht
- Definitionen für abzählbare und überabzählbare Mengen
- Beispiele für Turing-berechenbare Funktionen
- Kennenlernen, was die Church-Turing'sche These ausdrückt
- Herausstellen von nicht entscheidbaren Problemen und Sprachen
- Erweiterungen der Turing-Maschine

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. Vergleich zwischen Register- und Turingmaschine
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. Erweiterungen der Turing-Maschine

VI. Entscheidbarkeit und Berechenbarkeit

1. **Hauptfrage in diesem Kapitel**
2. Vergleich zwischen Register- und Turingmaschine
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. Erweiterungen der Turing-Maschine

Besitzt jedes Problem, das mathematisch exakt formulierbar ist, eine algorithmische Lösung

oder

gibt es Grenzen der

Berechenbarkeit ?

Im Jahre **1900** präsentierte **David Hilbert** auf einem Mathematiker-Kongress eine Liste mit **23** ungelösten Problemen:

Problem Nr. 10:
Gebe einen Algorithmus an, der für jede diophantische Gleichung feststellt, ob sie eine ganzzahlige Lösung besitzt oder nicht!

Beispiele für diophantische Gleichungen:

$x^2 + y^2 - z^2 = 0$ Lösung: $x = 3$, $y = 4$, $z = 5$

$6x^{18} - x + 3 = 0$ hat keine ganzzahlige Lösung, da $\forall\, x \in \mathbf{Z}:\ 6x^{18} > x - 3$

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. **Vergleich zwischen Register- und Turingmaschine**
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. Erweiterungen der Turing-Maschine

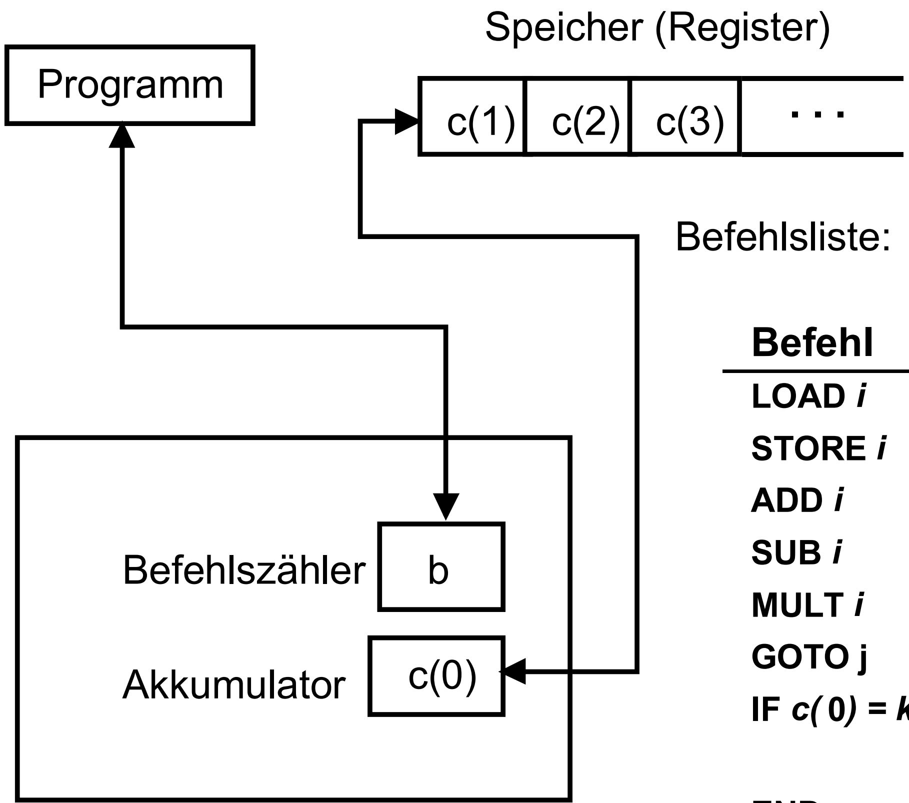
Modellbildung:

- Ein sehr realistisches Rechnermodell (nach der Vorstellung eines wirklichen Rechners) ist die **Registermaschine** (**R**andom **A**ccess **M**aschine, kurz **RAM)**.
- Zwar scheint die **Turing-Maschine** (kurz **TM**) nicht besonders realistisch zu sein, dient sie jedoch als Rechnermodell, das sich für „allgemeine“ theoretische Aussagen hervorragend eignet.
- Man kann zeigen (tun wir hier nicht!), dass die **TM** „gleichwertig“ zur **RAM** ist, die – wie deren Arbeitsweise demonstrieren wird – wiederum einen wirklichen Rechner modelliert.

Arbeitsweise:

- Die RAM besteht aus einem **Befehlszähler**, einem **Akkumulator**, aus **Registern** und aus einem **Programm**.
- Die Inhalte von Befehlszähler, Akkumulator und Registern sind natürliche Zahlen. In den ersten Registern steht die Eingabe.
- Die Register bilden den (unendlichen) Speicher der RAM und haben alle eine eindeutige Adresse. Der Inhalt *(**content**)* des Registers ***i*** sei mit ***c(i)*** bezeichnet.
- Das Programm besteht aus einer Folge von Befehlen, wobei die Programmzeilen durchnummeriert (0, 1, 2, ...) sind.
- Der Befehlszähler startet bei **Null**, und enthält die Nummer des nächsten auszuführenden Befehls.

Schematische Darstellung der RAM:



Befehlsliste:

Befehl	**Wirkung**
LOAD *i*	***c*(0) := *c*(*i*); *b* := *b* + 1**
STORE *i*	***c*(*i*) := *c*(0); *b* := *b* + 1**
ADD *i*	***c*(0) := *c*(0) + *c*(*i*); *b* := *b* + 1**
SUB *i*	***c*(0) := max{ 0, *c*(0) - *c*(*i*)}; *b* := *b* + 1**
MULT *i*	***c*(0) := *c*(0) * *c*(*i*); *b* := *b* + 1**
GOTO j	***b* := *j***
IF *c*(0) = *k* GOTO *j*	***b* : = *j* *falls c*(0) = *k***
	b* := *b* + 1 *sonst
END	***b* := *b***

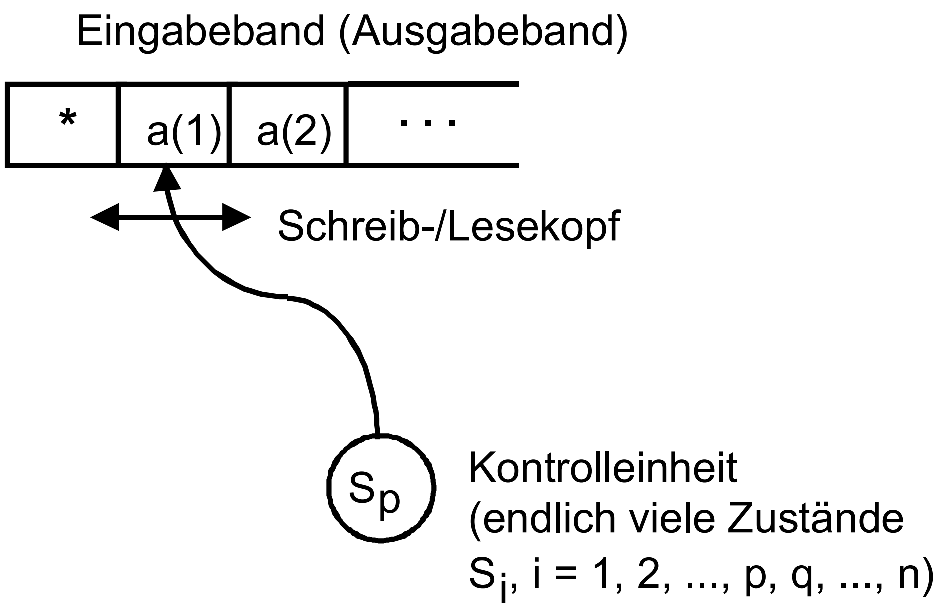
Beispiel: f(x) = 0, falls x gerade, sonst Endlosschleife

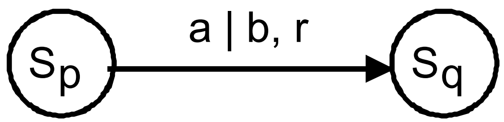
```
0  :  READ                1    // Wert x in Reg. c(1) ablegen
1  :  LOAD                1    // c(1) = x in Akkumulator laden
2  :  DIV                 =2   // dividiere Akku durch 2 (integer!)
3  :  MULT                =2   // multipliziere Akku mit 2 (integer!)
4  :  STORE               2    // Akkumulator in c(2) ablegen
5  :  LOAD                1    // c(1) in Akkumulator laden
6  :  SUB                 2    // davon nun c(2) subtrahieren
7  :  IF c(0) = 0  GOTO   10   // bedingter Sprung zu 10 (Ausgabe)
8  :  SUB                 =1   // Akkumulator wird bzw. bleibt 0
9  :  IF c(0) = 0  GOTO   8    // Rücksprung zu 8 (Endlosschleife)
10 :  WRITE               =0   // c(0) war 0 und somit 0 ausgeben
11 :  END                      // Befehlszähler b bleibt 11
```

Arbeitsweise:

- Die TM besteht aus einem einseitig unendlichen **Eingabe-** und **Ausgabeband** mit einem freibeweglichen **Schreib-/Lesekopf**.
- Der Schreib-/Lesekopf wird von einer endlichen **Kontrolleinheit** (endlich viele Zustände) gesteuert.
- Das Eingabe- und Ausgabeband enthält eine Folge von Symbolen, welche zu Beginn der Berechnung die Eingabe darstellt.
- Die Kontrolleinheit entspricht dem Befehlszähler der RAM und die Zellen des Eingabe- und Ausgabebandes den Registern der RAM.
- Die Zustandsübergänge der TM werden anhand einer Maschinentafel vollzogen. Dabei bewegt sich der Schreib-/Lesekopf eine Stelle nach recht, nach links oder überhaupt nicht.

Schematische Darstellung der TM:





Maschinentafel:

	b_1	a	...	b
S_0				
S_1				
...				
S_p		(S_q, b, r)		
S_q				
...				
S_n				

Beispiel: $L(TM) = \{ w \mid w = a^n b^n c^n \text{ für } n = 1, 2, \ldots \}$

$\Sigma = \{a, b, c\}$; $\mathbf{B} = \{a, b, c, *, A, B, C\}$; $\mathbf{S} = \{S_0, S_1, \ldots, S_7\}$; $\mathbf{F} = \{S_7\}$

	a	**b**	**c**	**A**	**B**	**C**	**$*$**
S_0	–	–	–	–	–	–	$(S_1, *, r)$
S_1	(S_2, A, r)	–	–	–	(S_5, B, r)	–	–
S_2	(S_2, a, r)	(S_3, B, r)	–	–	(S_2, B, r)	–	–
S_3	–	(S_3, b, r)	(S_4, C, l)	–	–	(S_3, C, r)	–
S_4	(S_4, a, l)	(S_4, b, l)	–	(S_1, A, r)	(S_4, B, l)	(S_4, C, l)	–
S_5	–	–	–	–	(S_5, B, r)	(S_5, C, r)	$(S_6, *, l)$
S_6	–	–	–	(S_6, A, l)	(S_6, B, l)	(S_6, C, l)	$(S_7, *, h)$

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. Vergleich zwischen Register- und Turingmaschine
3. **Definition für einen Algorithmus**
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. Erweiterungen der Turing-Maschine

<u>Definition</u> (Algorithmus):

Es sei **P** eine **Problemklasse** und **A** die Menge der konkreten **Problemausprägungen**, d. h. die Menge derjenigen Daten, die ein Problem beschreiben.

Dann verstehen wir unter einem **Algorithmus A_P** zu der Klasse **P** ein allgemeines, deterministisches Verfahren, welches – auf richtige Anfangsdaten $a \in \mathbf{A}$ angewendet – nach <u>endlich</u> vielen Schritten hält und die Lösung des Problems liefert (d. h. ein Element der Lösungsmenge **B**).

Wir schreiben: $\mathbf{A_P} : \mathbf{A} \rightarrow \mathbf{B}$,

d. h. $\forall\, a \in \mathbf{A} : \exists\, b \in \mathbf{B} : \mathbf{A_P}(a) = b$

VI. Entscheidbarkeit und Berechenbarkeit

Definition (Entscheidungs- und Aufzählverfahren):

Einen Algorithmus $\mathbf{A_P} : \mathbf{A} \rightarrow \mathbf{B}$ nennen wir auch

- ein Entscheidungsverfahren, falls

  $\mathbf{B}$ = (True, False) bzw. auch allgemeiner falls
  $|\mathbf{B}| = 2$

- ein Aufzählverfahren, falls

  $\mathbf{A} = \mathrm{IN}$

Beispiel 1 (Primzahlen):

geg: Zahl $Z \in \mathrm{IN}$ ges: Algorithmus, der entscheiden kann, ob Z eine Primzahl ist oder nicht d. h. {2, 3, 5, 7, 11, 13, ...} ?

$\Rightarrow$ Entscheidungsverfahren

Beispiel 2 (Primzahlen):

geg: **A** = { ∀ Primzahlen }

= { p | p ist nicht das Vielfache einer ganzen Zahl mit Ausnahme der 1 und p selbst)}

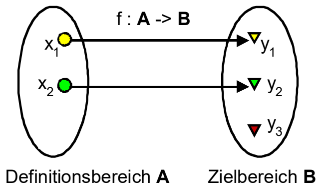
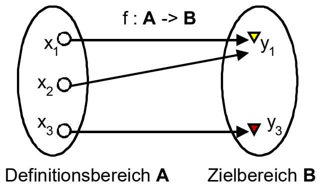
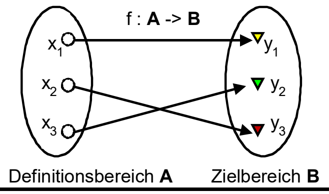
ges: $A_P(1), A_P(2), A_P(3), \ldots$

⇒ Aufzählverfahren

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. Vergleich zwischen Register- und Turingmaschine
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. **Berechenbare Funktionen und entscheidbare Mengen**
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. Erweiterungen der Turing-Maschine

Definition (injektiv, surjektiv, bijektiv):

Injektion: 	$\forall\ x_1, x_2 \in \mathbf{A} : x_1 \neq x_2$ $\Rightarrow f(x_1) \neq f(x_2)$ $\|A\| \leq \|B\|$ (Kardinalität) auch **linkseindeutig** genannt Bsp.: $\mathbf{R} \to \mathbf{R} : f(x) = \arctan(x)$ $\mathbf{N} \to \mathbf{N} : f(n) = n^2$; aber: $\mathbf{Z} \to \mathbf{Z} : f(n) = n^2$ ist nicht injektiv	Zwei verschiedene Elemente von **A** werden auf verschiedene Elemente von **B** abgebildet – oder: Jedem y-Wert im Zielbereich besitz **nur genau einen** x-Wert im Definitionsbereich; daher wiederholt sich bei einer injektiven Funktion also nie ein Funktionswert.
Surjektion: 	$\forall\ y_i \in \mathbf{B} : \exists$ mind. ein $x_i \in \mathbf{A}$: $f(x_i) = y_i$ $\|A\| \geq \|B\|$ auch **rechtstotal** genannt Bsp.: $\mathbf{Z} \to \mathbf{N} : f(x) = \mathrm{abs}(x)$	Jedes Element der Zielmenge B ist ein Funktionswert, d. h. **alle möglichen** y-Werte im Zielbereich werden angenommen – oder: Bildbereich von f stimmt mit der Zielmenge von f überein.
Bijektion: 	$\forall\ y_i \in \mathbf{B} : \exists$ genau ein $x_i \in \mathbf{A}$: $f(x_i) = y_i$ $\|A\| = \|B\|$ Bsp.: $\mathbf{\{0 .. 255\}} \to \mathbf{\{0, 1\}^8}$: f(x) = 8-stell. Binärzahl	f heißt bijektiv, falls es sowohl injektiv als auch surjektiv ist. Geanu dann existiert auch eine **Umkehrfunktion** $x = f^{-1}(y)$

Definition (berechenbare Funktion):

Eine Funktion $f : M_1 \rightarrow M_2$ heißt berechenbar, falls es einen Algorithmus

$$A_f : M_1 \rightarrow M_2$$

gibt mit

$$A_f(w) = f(w) \quad \text{für} \quad \forall\, w \in M_1$$

Interpretation:

Zu jedem Argument $w \in M_1$ kann man also den Funktionswert $f(w) \in M_2$ in endlich vielen Schritten berechnen.

<u>Beispiel</u> (berechenbare Funktion):

Für $M_1 = \mathrm{IN}$ und $M_2 = \mathrm{IN}$ ist

$$f : \mathrm{IN} \rightarrow \mathrm{IN} \text{ mit } f(n) = n \cdot (n + 1) / 2$$

eine <u>berechenbare</u> Funktion, da es für diese Funktion einen Algorithmus gibt (Aufgabe 1.3 – vgl. Übungsblatt 1).

<u>Anmerkung:</u>

Es ist keineswegs eine triviale Frage, ob <u>jede</u> gegebene Funktion berechenbar ist, also durch einen Algorithmus realisiert werden kann. Wir werden sehen, dass die Frage zu verneinen ist.

Definition (abzählbare Menge, überabzählbare Menge):

Es sei M eine Menge. Wir nennen M abzählbar, wenn sich M bijektiv auf die Menge IN (der natürlichen Zahlen) oder eine Teilmenge von **IN** abbilden lässt. Andernfalls nennen wir M überabzählbar.

Merke:

- Jede endliche Menge ist abzählbar.
- Ist A ein (endliches) Alphabet, so ist A* abzählbar (wegen lexikographischer Ordnung).

Satz:

Es sei Σ ein Alphabet und somit endlich. Dann gibt es überabzählbar viele Funktionen $\mathbf{f : \Sigma^* \to \Sigma^*}$, von denen allerdings nur abzählbar viele berechenbar sind. (Beweis durch Wiederspruch! – vgl. Übungsblatt 12)

Definition (entscheidbare Menge):

a) Es seien $M_1 \subseteq M_2 \subseteq \Sigma^*$.

M₁ heißt entscheidbar relativ zu M_2, wenn es einen Algorithmus

$$A_{M1M2} : M_2 \rightarrow \{ \text{True}, \text{False} \}$$

gibt, mit dessen Hilfe man **zu jedem** Element $w \in M_2$ feststellen kann, ob es zu M_1 gehört oder nicht.

kurz: $\forall\, w \in M_2 : A_{M1M2}(w) = „ w \in M_1 “$

b) Es sei $M \subseteq \Sigma^*$.

M heißt absolut entscheidbar (oder kurz entscheidbar), wenn M relativ zu Σ^* entscheidbar ist. $\rightarrow$ Entscheidbarkeit ist eine Mengeneigenschaft und nicht etwa die Eigenschaft eines einzelnen Objektes!

Beispiele:

a) Die Menge

$$M := \{\, n \in \mathbb{N} \mid n \text{ ist eine Primzahl} \,\}$$

ist entscheidbar relativ zu IN, denn es muss zu einer vorgegebenen Zahl $n \in$ IN nur getestet werden, ob ein $m < n$ mit $m \neq 1$ existiert, das n teilt („|“). Wenn ja (d.h. es $\exists$ ein m) $\Rightarrow$ n ist **keine** Primzahl; sonst (d. h. es $\exists$ kein m) $\Rightarrow$ n ist eine Primzahl!

kurz:

$$\text{Für } \mathbf{n \in IN \;\exists\; m \;\; (m \neq 1 \wedge m < n) \,\big|\, (m \text{ „|“ } n) \;\Rightarrow\; n \notin IP}$$

b) Jede **endliche** Menge ist entscheidbar; ein entsprechender Algorithmus muss lediglich eine endliche Menge oder Liste durchsuchen, um die Entscheidung treffen zu können.

c) Das allgemeine Wortproblem für **Typ-0-Grammatiken** ist nicht entscheidbar, d. h. es gibt **keinen** Algorithmus, der zu einer beliebig vorgegebenen Grammatik **G** und einem ebenfalls vorgegebenen Wort **w** entscheiden kann, ob $w \in L(G)$ gilt oder nicht.

Definition (aufzählbare Menge):

Eine Menge $M \subseteq \Sigma^*$ heißt aufzählbar, wenn es eine Funktion

$$f : \mathbb{N}_0 \rightarrow M$$

gibt, die surjektiv und berechenbar ist. Wir sagen dann: „M wird durch f aufgezählt“, d. h. $M = \{ f(0), f(1), f(2), f(1), ... \}$

$\Rightarrow$ durch die Funktion f werden die Elemente von M in eine feste Reihenfolge gebracht, wobei nicht ausgeschlossen ist, dass ein Element von M mehrfach aufgezählt wird.

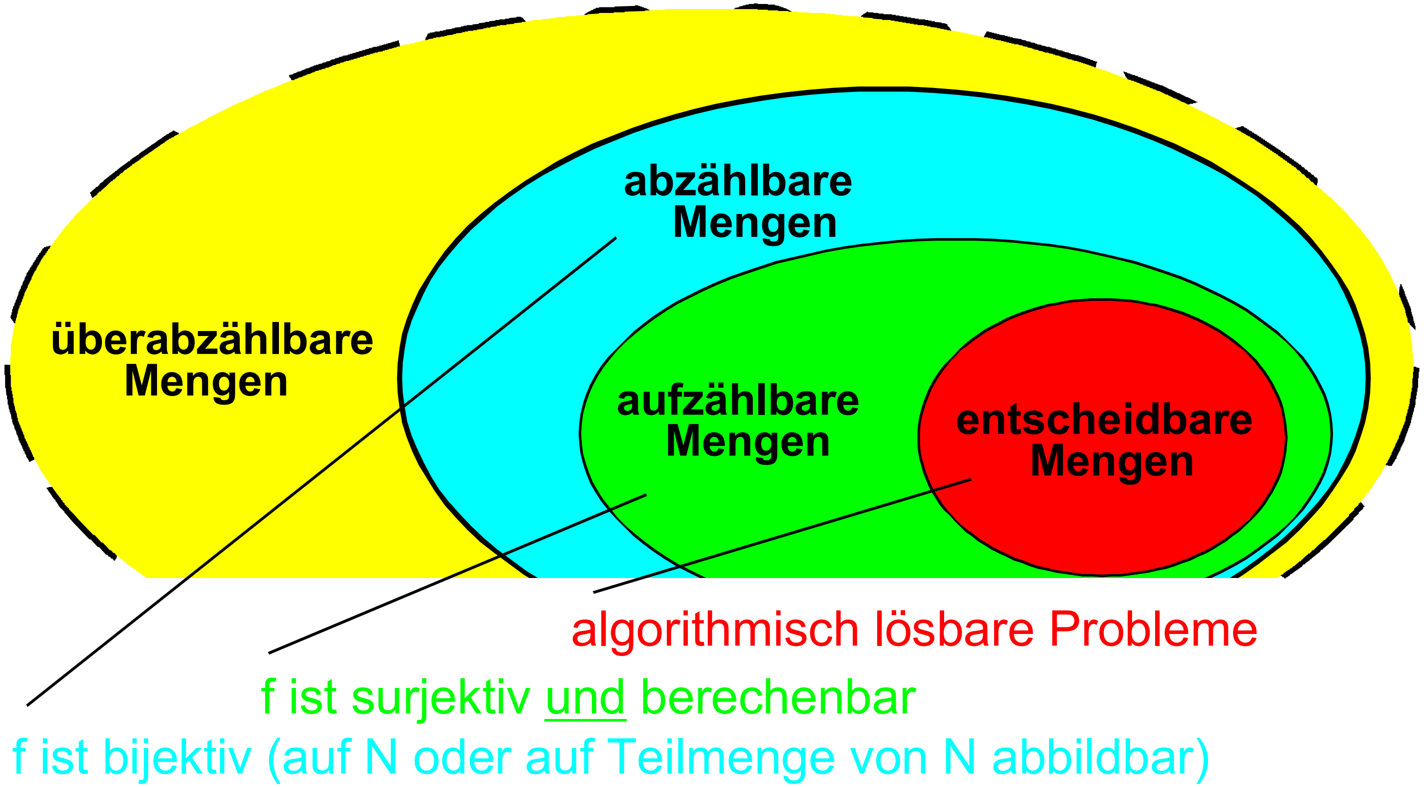
Satz:

Für eine Menge $M \subseteq \Sigma^*$ gilt:

a) M ist **auf**zählbar $\Rightarrow$ M ist **ab**zählbar; die Umkehrung gilt i.a. nicht!

b) M ist **entscheid**bar $\Rightarrow$ M ist **auf**zählbar

Veranschaulichung der Zusammenhänge:



VI. Entscheidbarkeit und Berechenbarkeit

Definition (Turing-Berechenbarkeit):

Es sei **TM** = (S, So, F, Σ, B, δ) eine **Turing-Maschine** und M_1, $M_2 \subseteq \Sigma^*$.

a) Wir sagen: Eine **TM** **realisiert** eine Funktion $f : M_1 \rightarrow M_2$, wenn folgendes gilt:

1) Für $\forall\ w \in M_1$ gilt: $(*, w, s_0) \Rightarrow (*, f(w), s_f)$, wobei $(*, f(w), s_f)$ eine sog. Finalkonfiguration ist, d. h. die Maschine hält an, und s_f ist ein Endzustand.

2) Andernfalls, d. h. für $w \notin M_1$, geht die Maschine nie in eine Final-konfiguration über, und das Verhalten der Maschine ist unbestimmt.

b) Eine Funktion $f : M_1 \rightarrow M_2$ heißt Turing-**berechenbar**, wenn es eine **TM** gibt, die bei Eingabe von $w \in \Sigma^*$ den Funktionswert $f(w) \in \mathbf{B}^*$ berechnet.

Beispiele:

a) Die Funktion $\mathbf{f : IN_0 \rightarrow IN}$ mit $\mathbf{f(n) = n + 1}$ ist Turing-berechenbar.

b) Die Funktion $\mathbf{f : IN_0 \rightarrow IN_0}$ mit $\mathbf{f(n_1, n_2) = n_1 + n_2}$ ist ebenfalls Turing-berechenbar.

zu a)

$* \; 1 \; 1 \; 1 \; \ldots \; 1 \; * \; * \; \ldots \quad \Rightarrow \quad * \; 1 \; 1 \; 1 \; \ldots \; 1 \; 1 \; * \; \ldots$

n mal Eins entspricht dem Wert **n**

n+1 mal Eins entspricht dem ***Wert*** **n** + **1**

zu b)

$* \; 1 \; 1 \; 1 \; \ldots \; 1 \; * \; 1 \; 1 \; 1 \; \ldots \; 1 \; * \; * \; \ldots \Rightarrow \; * \; 1 \; 1 \; 1 \; \ldots \; 1 \; 1 \; 1 \; 1 \; \ldots \; 1 \; * \; * \; \ldots$

Wort 1 entspricht $\mathbf{n_1}$

Wort 2 entspricht $\mathbf{n_2}$

Wort 1 und Wort 2 direkt hintereinander entspricht dem ***Wert*** $\mathbf{n_1}$ + $\mathbf{n_2}$

VI. Entscheidbarkeit und Berechenbarkeit

Satz (Church-Turing'sche These):

Jede (im intuitiven Sinne) berechenbare Funktion ist auch Turing-berechenbar und umgekehrt.

Dieser Satz ist nicht beweisbar; zur Aussage des Satzes hat aber noch niemand ein Gegenbeispiel angeben können.

Folgerung:

Aufgrund der vorgenommenen Definition und der Church-Turing'sche These gilt nun:

Jede aufzählbare bzw. entscheidbare Menge ist auch Turing-aufzählbar bzw. Turing-entscheidbar und umgekehrt.

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. Vergleich zwischen Register- und Turingmaschine
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. **Das Post'sche Korrespondenzproblem**
9. Erweiterungen der Turing-Maschine

Definition (PKP):

Beim Post'schen Korrespondenzproblem (PKP) ist eine endliche Folge von Wortpaaren

$$K = \big((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \big)$$

über einem endlichen Alphabet Σ gegeben.

Es gilt weiter $x_i \notin \varepsilon$ und $y_i \notin \varepsilon$. Gefragt ist nun, ob es eine endliche Folge von Indizes $i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, n\}$ gibt, so dass gilt:

$$x_{i1}\ x_{i2} \ldots x_{ik} = y_{i1}\ y_{i2} \ldots y_{ik}$$

Satz:

Das Post'schen Korrespondenzproblem (PKP) ist nicht entscheidbar.

Beispiel 1:

$$K = \big((1, 111), (10111, 10), (10, 0)\big)$$

hat die Lösung (2, 1, 1, 3), denn es gilt:

$$x_2\ x_1\ x_1\ x_3 = 10111\ 1\ 1\ 10 = 10\ 111\ 111\ 0 = y_2\ y_1\ y_1\ y_3$$

Beispiel 2:

$$K = \big((10, 101), (011, 11), (101, 011)\big)$$

hat keine Lösung.

Beispiel 3:

$$K = \big((001, 0), (01, 011), (01, 101), (10, 011)\big)$$

hat eine Lösung der Länge 66.

VI. Entscheidbarkeit und Berechenbarkeit

1. Hauptfrage in diesem Kapitel
2. Vergleich zwischen Register- und Turingmaschine
3. Definition für einen Algorithmus
4. Definitionen für Entscheidungs- und Aufzählverfahren
5. Berechenbare Funktionen und entscheidbare Mengen
6. Definition der Turing-Berechenbarkeit
7. Die Church-Turing'sche These
8. Das Post'sche Korrespondenzproblem
9. **Erweiterungen der Turing-Maschine**

Folgende Erweiterungsmöglichkeiten existieren bei einer TM:

1. Mehrere Schreib-/Leseköpfe
2. Mehrere Bänder
3. Mehrere Schreib-/Leseköpfe für mehrere Bänder
4. Mehrdimensionale Bänder

Satz:

Es hat sich gezeigt, dass keine dieser Erweiterungen mehr leistet, als eine „normale" Turing-Maschine. Man kann z. B. beweisen, dass jede **k**-Band Turing-Maschine durch eine **1**-Band Turing-Maschine simuliert werden kann. Das gleiche gilt auch für alle anderen angegebenen Modifikationen.

Automatentheorie und Formale Sprachen

– LV 4110 –

Turingmaschinen und Kontextsensitive Sprachen

- Kennenlernen der Zusammenhänge zwischen unterschiedlichen Sprach-Typen
- Untersuchung der Sprache $L(G) = \{a^n b^n c^n \mid n = 0, 1, 2, \ldots\}$
- Definition eines Maschinenmodells zur allgemeinen Beschreibung von Algorithmen
- Kennenlernen der Arbeitsweise einer Turingmaschine
- Anwendung der Technik des Zusammensetzens elementarer TM-Operationen
- Kennenlernen der Turingmaschine als Akzeptor
- Identifizierung des Halteproblems bei einer beschränkten Bandlänge

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. Die Sprache vom Typ 0
3. Turingmaschinen
   - 3.1 Modell einer Turingmaschine
   - 3.2 Arbeitsweise einer Turingmaschine
   - 3.3 Beispiele für elementare Operationen einer TM
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - 3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen

Wir erinnern uns:

Chomsky-Grammatiken vom **Typ 1** haben die Form:

$$\alpha A \beta \to \alpha \gamma \beta \quad \text{mit} \quad \gamma \neq \varepsilon$$

$$A \in N \,; \qquad \alpha, \beta, \gamma \in (N \cup T)^*$$

mit der Ausnahme, dass

$$S \to \varepsilon$$

dazugehören darf, wenn **S** in keiner Regel auf der rechten Seite auftritt.

Gedankenexperiment:

Wählt man: $\alpha, \beta = \varepsilon \quad \Rightarrow \quad A \rightarrow \gamma$ (entspr. Typ 2)

Folge: Die Regeln der **kontextsensitiven** Grammatik gehen in solche einer **kontextfreien** Grammatik über.

Da es zu einer **kontextfreien** Grammatik außerdem immer eine **äquivalente ε-freie** Grammatik gibt, folgt weiter, dass die Menge der **kontextsensitiven Sprachen** die Menge der **kontextfreien Sprachen** umfaßt und damit eine echte **Obermenge** darstellt.

<u>Definitionen für eine monotone Chomsky-Grammatik</u>:

Eine Chomsky-Grammatik $G = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S)$ heißt **monoton**, wenn sie – abgesehen von der Regel $\mathbf{S} \to \varepsilon$ – nur Regeln der Form

$$\varphi \to \gamma \qquad \text{mit} \qquad |\varphi| \leq |\gamma|\,; \qquad \varphi, \gamma \in (\mathbf{N} \cup \mathbf{T})^*$$

hat.

<u>Satz</u>:

Jede kontextsensitive Grammatik ist auch monoton und zu jeder monotonen gibt es eine **äquivalente**, kontextsensitive Grammatik (vgl. Beispiel).

Beispiel: $\Rightarrow$ Monotone Grammatik für $L = \{a^n b^n c^n \mid n \in \mathbb{N}_0\}$

Wir betrachten die Grammatik G = (**T**, **N**, **P**, S) mit dem Startsymbol S sowie **T** = { a, b, c }, **N** = { A, C, S} und den Produktionen **P**:

$$
\begin{aligned}
\mathbf{P} = \{\ & S \Rightarrow aAbc \mid abc\ , && (1, 1') \\
& A \Rightarrow aAbC \mid abC\ , && (2, 2') \\
& Cb \Rightarrow bC\ , && (3) \\
& Cc \Rightarrow cc\ \ \} && (4)
\end{aligned}
$$

- G ist monoton.
- G ist nicht **kontextsensitiv**, weil (3) nicht die Form $\alpha A\beta \rightarrow \alpha\gamma\beta$.

Monotone Grammatiken

$S \Rightarrow aAbc \mid abc\ (1)\ ,\ A \Rightarrow aAbC \mid abC\ (2)\ ,\ Cb \Rightarrow bC\ (3)\ ,\ Cc \Rightarrow cc\ (4)$

Ableitung von $w = a^3b^3c^3$:

$S \Rightarrow_{(1)} a\underline{A}bc \Rightarrow_{(2)} aa\underline{A}bCbc \Rightarrow_{(2')} aaab\underline{Cb}Cbc \Rightarrow_{(3)} aaabbC\underline{Cb}c$

$\Rightarrow_{(3)} aaabb\underline{Cb}Cc \Rightarrow_{(3)} aaabbbC\underline{Cc} \Rightarrow_{(4)} aaabbb\underline{Cc}c$

$\Rightarrow_{(4)} aaabbbccc = a^3b^3c^3$

- G ist zwar monoton, aber <u>nicht</u> kontextsensitiv.
- Jetzt eine zu G **äquivalente kontextsensitive** Grammatik G'.
- Dazu muss Regel (3) geändert werden.

$S \Rightarrow aA\mathbf{B}c \mid abc$ (1) , $A \Rightarrow aA\mathbf{B}C \mid abC$ (2) , $Cc \Rightarrow cc$ (4)

$CB \Rightarrow HB$ (3) , $HB \Rightarrow HC$ (3') , $HC \Rightarrow BC$ (3'') , $B \Rightarrow b$ (3''')

Ableitung von $w = a^3b^3c^3$:

$$S \Rightarrow_{(1)} a\underline{A}Bc \Rightarrow_{(2)} aa\underline{A}BCBc \Rightarrow_{(2')} aaabCB\underline{CB}c \Rightarrow_{(3)} aaabCB\underline{HB}c$$
$$\Rightarrow_{(3')} aaabCB\underline{HC}c \Rightarrow_{(3'')} aaab\underline{CB}BCc \Rightarrow_{(3)} aaab\underline{HB}BCc$$
$$\Rightarrow_{(3')} aaab\underline{HC}BCc \Rightarrow_{(3'')} aaabB\underline{CB}Cc \Rightarrow_{(3)} aaabB\underline{HB}Cc$$
$$\Rightarrow_{(3')} aaabB\underline{HC}Cc \Rightarrow_{(3'')} aaab\underline{B}BCCc \Rightarrow_{(3''')} aaabb\underline{B}CCc$$
$$\Rightarrow_{(3''')} aaabbbC\underline{Cc} \Rightarrow_{(4)} aaabbb\underline{Cc}c \Rightarrow_{(4)} aaabbbccc = a^3b^3c^3$$

- G ist jetzt monoton <u>und</u> kontextsensitiv, d. h. L(G) vom **Typ 1**.

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. **Die Sprache vom Typ 0**
3. Turingmaschinen
   - 3.1 Modell einer Turingmaschine
   - 3.2 Arbeitsweise einer Turingmaschine
   - 3.3 Beispiele für elementare Operationen einer TM
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - 3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen

Allgemeine Sprachen

<u>Wir erinnern uns</u>:
Chomsky-Grammatiken vom **Typ 0** haben die Form:

$$\alpha \to \beta \qquad \text{mit} \qquad \alpha, \beta \in (\mathbf{N} \cup \mathbf{T})^*$$

ohne sonstige Einschränkungen.

<u>Anmerkung</u>:

- $\forall$ Sprachen vom Typ > 0 sind auch vom Typ = 0.
- Dass auch Sprachen existieren, die vom Typ = 0 sind, aber nicht vom Typ > 0, ist **bisher** nur über einen rein mathematischen Existenzbeweis nachvollziehbar ($\to$ ein explizites Beispiel **bislang** nicht bekannt).
- Bei jeder Konstruktion einer Grammatik vom Typ = 0 hat sich bisher gezeigt, dass die zugehörige Sprache auch von einer Typ-1-Grammatik erzeugt werden kann.

Die Familie L1 der kontextsensitiven Sprachen ist eine echte Obermenge der kontextfreien Sprachen L2.

Die Familie L0 der allgemeinen Sprachen ist eine echte Obermenge der kontext-sensitiven bzw. monotonen Sprachen L1.

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. Die Sprache vom Typ 0
3. **Turingmaschinen**
   - 3.1 Modell einer Turingmaschine
   - 3.2 Arbeitsweise einer Turingmaschine
   - 3.3 Beispiele für elementare Operationen einer TM
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - 3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen

- Wir wollen nun **kontextsensitive** (Typ 1) und **allgemeine** (Typ 0) Sprachen betrachten.
- Wir suchen nach einem **Maschinenmodell**, das diese **beiden** Sprachen beschreiben kann.

$\Rightarrow$ Es muss offensichtlich allgemeiner sein, als der **KA**, dessen wesentliche Beschränkung die Zugriffsmöglichkeit auf den Kellerspeicher darstellt, d. h. jeweils nur auf das **oberste** Zeichen des Kellers zugreifbar.

Alan M. Turing (engl. Mathematiker und Kryptoanalytiker) beschrieb 1936 ein nach ihm benanntes Maschinenmodell – sog. **Turingmaschine** – im Zusammenhang mit der **Berechenbarkeit von Funktionen** und der Frage, was man unter einem **Algorithmus** zu verstehen hat.

(1912 – 1954)

Alan Mathison Turing

- britischer Mathematiker und Kryptoanalytiker (Bletchley Park, 1943)
- einflussreichster Theoretiker der Computerentwicklung (Colossus)
- legte die theoretischen Grundlagen der frühen Informatik (Berechen- und Entscheidbarkeit)
- maßgeblich an der Entzifferung von Enigma-verschlüsselten Funksprüchen beteiligt

Von 1945 bis 1948 im National Physical Laboratory, Teddington, tätig am Design der **A**utomatic **C**omputing **E**ngine (ACE)

Übersicht:

Verschlüsselungsgerät der Deutschen Wehrmacht im Zweiten Weltkrieg (1939 – 1945)

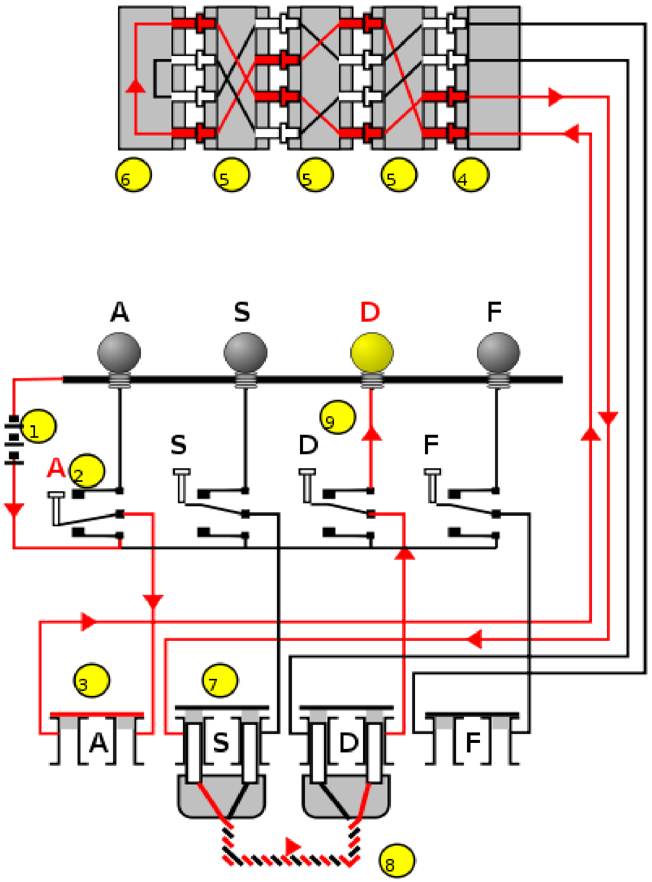
Two Design Principles for secure ciphers:

- Confusion

  The ciphertext statistics should depend on the plaintext statistics in a manner **too complicated** to be exploited by the enemy cryptanalyst.

- Diffusion

  Each digit of the plaintext (and/or secret key) should influence **many digits** of the ciphertext.



Komponenten und Funktionsweise:

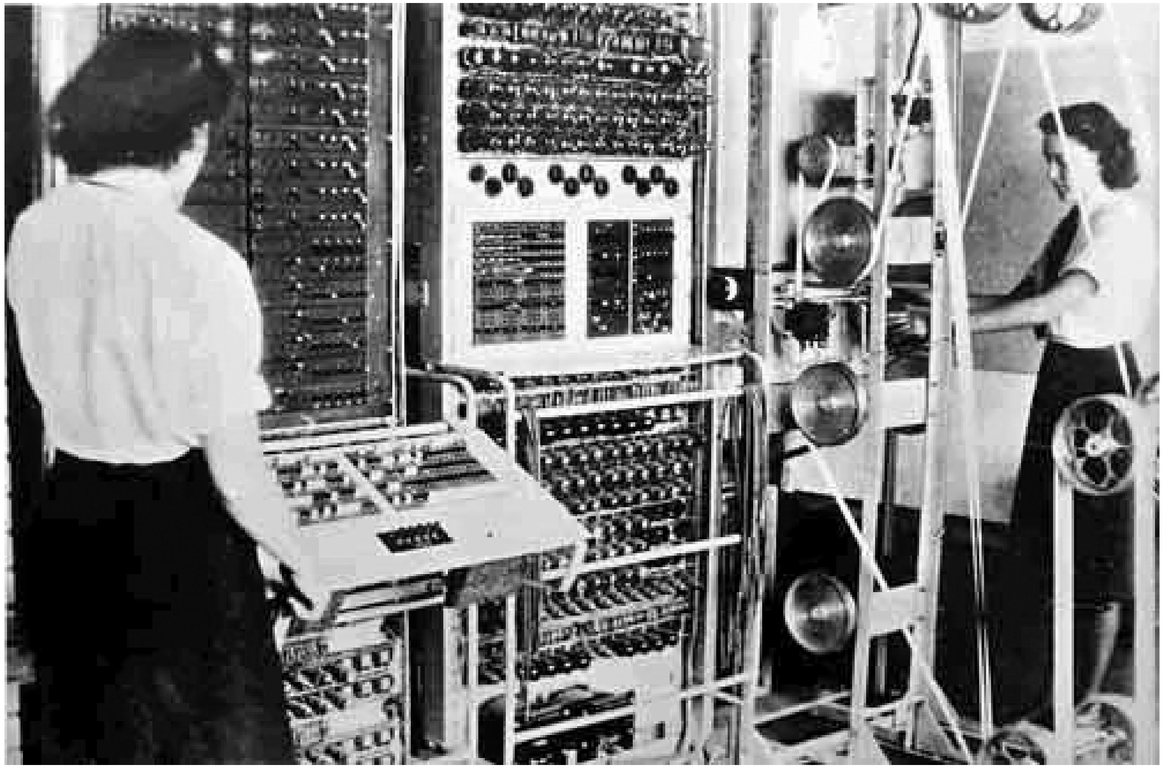
1	Batterie
2	Tastatur
3,7	Steckerbrett
8	Steckkabel
5	Walzensatz (dreht sich)
4	Eintrittswalze (fest)
6	Umkehrwalze (fest)
9	Lampenfeld

Übersicht:

Electronic **N**umerical **I**ntegrator **a**nd **C**omputer (**ENIAC**), USA – 1946

- erste rein elektronische Universalrechner der U.S. Army
- entwickelt ab 1942 bis 1946 an der University of Pennsylvania
- benutzt Elektronenröhren zur Repräsentation von Zahlen und elektrische Pulse für deren Übertragung

Funktionen: add, sub, mult, div, sqrt



Übersicht:

COLOSSUS, UK – 1943

- Röhrenrechner (Computer der ersten Gen.)
- Militärischer Einsatz im Zweiten Weltkrieg
- Erzeugung von Zufallszahlen
- Zum Chiffrieren Bildung von XOR

Entwurf in Bletchley-Park (1943) nach der Idee von Alan M. Turing

Z3 (Konrad Zuse, 1941, Berlin)

- **erster funktionsfähiger Digitalrechner** (Universalrechner) weltweit
- in **elektromagnetischer Relaistechnik** aufgebaut 600 Relais (RW) und 1400 Relais (SW)
- verwendete eine **binäre Gleitkommaarithmetik**
- erst 1998 fand man heraus, dass die Z3 der Definition eines **turingmächtigen Computers** genügt
- sie wurde 1944 bei einem Bombenangriff zerstört

Ein **funktionsfähiger Nachbau** (Deutsches Museum in München)

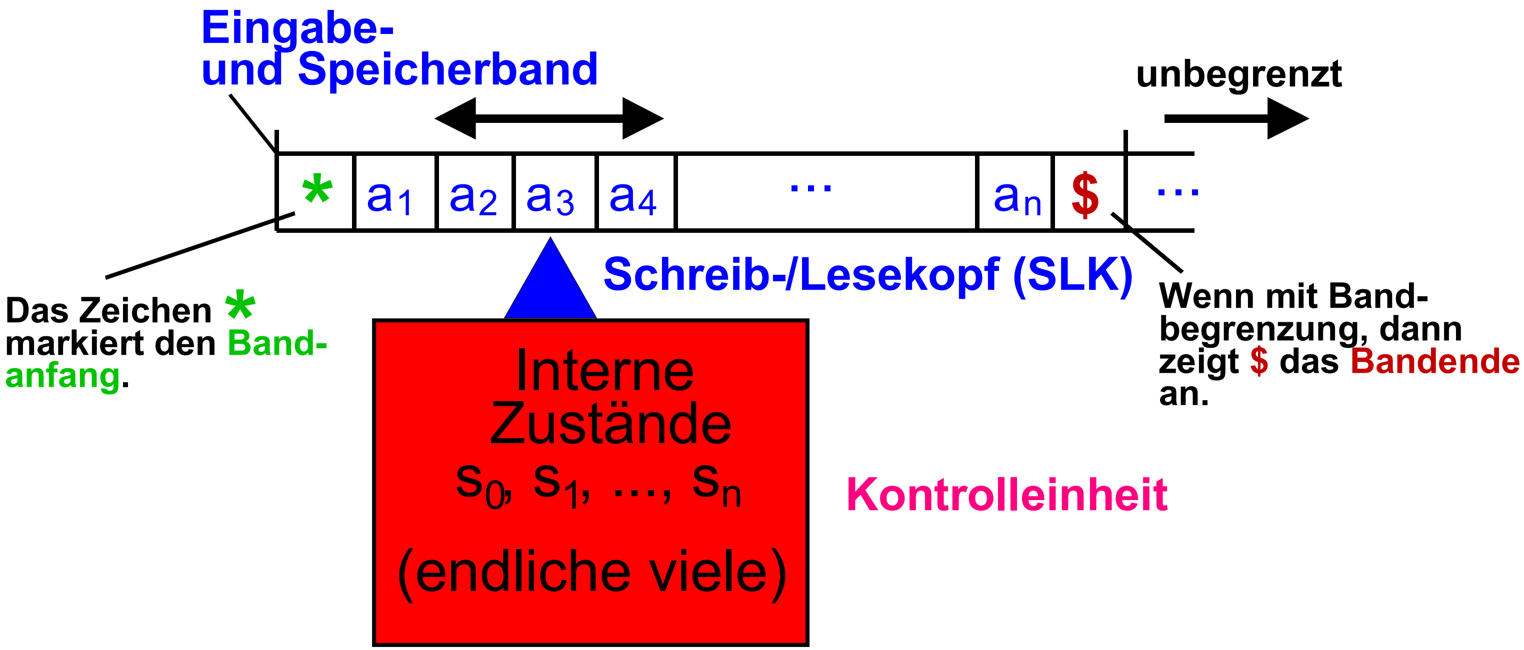
Eigenschaften der ersten Computer

Computer-Name	Land	Jahr	GkA[1)]	binär	elektro-nisch	program-mierbar	Turing-fähig
Zuse Z3	DE	1941	ja	ja	nein	ja, (Lochst)	ja
Colossus	UK	1943	nein	ja	ja	teilweise	nein
ENIAC	USA	1946	nein	nein	ja	teilweise	ja

1) **G**leit**k**omma-**A**rithmetik

V. Turingmaschinen und Kontextsensitive Sprachen

Turingmaschine



- Turingmaschine besteht aus einer **Kontrolleinheit** mit einem **Schreib-/Lesekopf (SLK)**
- sowie aus einem **einseitig unbegrenzten** Speicherband (→ Eingabe und Ausgabe).

Funktion des SLK:

- Mit dem **SLK** kann die Maschine **genau ein** Zeichen lesen bzw. überschreiben.
- Der **SLK** kann um eine Position nach links oder nach rechts gerückt werden.

$\Rightarrow$

Die wesentliche Eigenschaft einer **TM** ist, dass **jedes** Feld des Eingabe- und Speicherbandes **gelesen** und **verändert** werden kann.

Damit entspricht das Eingabe- und Speicherband dem Hauptspeicher eines modernen Rechners.

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. Die Sprache vom Typ 0
3. Turingmaschinen
   - 3.1 Modell einer Turingmaschine
   - **3.2 Arbeitsweise einer Turingmaschine**
   - 3.3 Beispiele für elementare Operationen einer TM
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - 3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen

Arbeitsweise der TM:

Der ***typische elementare Arbeitsschritt*** einer Turingmaschine besteht darin, das Zeichen eines Arbeitsfeldes zu lesen, in Abhängigkeit vom eingelesenen Zeichen und dem gegenwärtigen Maschinenzustand das gleiche oder ein anderes Zeichen in das Arbeitsfeld zu schreiben und anschließend gegebenenfalls auf ein Nachbarfeld zu positionieren.

Definition:

TM = (**S**, s_0, **F**, Σ, **B**, δ) bezeichnet eine (deterministische) Turingmaschine, wenn für die einzelnen Komponenten gilt:

- **S** endliche Menge der ***internen*** Zustände der Maschine
- s_0 interner Anfangszustand, $s_0 \in$ **S**
- **F** Menge der internen Endzustände, **F** $\in$ **S**
- Σ endliche Menge der Eingabezeichen
- **B** endliche **Menge der Bandzeichen** (inkl. Σ und eines Leerzeichens *, das nicht als Eingabezeichen eines Wortes auftritt)
- δ (*deterministische*) Überführungsfunktion δ: **S** x **B** $\rightarrow$ **S** x **B** x **X**, wobei **X** = {l, r, h} die möglichen Bewegungen des SLK darstellt

Erklärung zur Überführungsfunktion einer TM:

$$\delta\text{: } \mathbf{S} \times \mathbf{B} \rightarrow \mathbf{S} \times \mathbf{B} \times \mathbf{X}$$

Die Zuordnung (s, b) $\rightarrow$ (s‘, b‘, **X**) mit **X** $\in$ {l, r, h} bedeutet, dass – falls sich die Maschine im aktuellen Zustand s befindet und das Feld unter dem SLK mit dem Zeichen b beschriftet ist – sie in den Zustand s‘ übergeht, b durch b‘ ersetzt wird und der SLK entweder um eine Position nach links (**X** = l) bzw. nach rechts (**X** = r) geht oder an der aktuellen Position (**X** = h) verharrt.

$\Rightarrow$ Die Tabelle von δ wird auch als ***Maschinentafel*** oder als ***Programm*** der Turingmaschine bezeichnet.

Turingmaschine

Zustands- bzw. Maschinentafel

	b_1	b_2	b		...		b_m
S_0							
S_1							
...							
s			(s‘, b‘, X)				
...							
S_n							

- SLK zu Anfang ganz links auf dem ersten Feld des Bands → i. a. *
- TM hält an, falls δ für das aktuelle Paar (s, b) nicht definiert ist

Beispiel: ⇒ Turing-Maschine **TM** = (**S**, S_0, **F**, Σ, **B**, δ) zum Löschen des Speichers

Für

$$\Sigma = \{0, 1\}\ ,\ \mathbf{B} = \{0, 1, *\}\ ,\ \mathbf{S} = \{s_0, s_1\}\ \text{ und }\ \mathbf{F} = \{s_1\}$$

ist eine **TM** gesucht, deren SLK nach rechts zum nächsten Leerzeichen (*) läuft und dabei alle Felder löscht, d. h. 0 oder 1 jeweils durch ein Leerzeichen * ersetzt.

Maschinentafel:

δ	**0**	**1**	*
$\mathbf{S_0}$	(S_0, *, r)	(S_0, *, r)	(S_1, *, h)

<u>Funktionsweise</u>:

t = 0:

*	**0**	**1**	*	*	…

Bandanfang —↑ ↑ S_0 ↑ = SLK

t = 1:

*	*	**1**	*	*	…

↑ S_0

t = 2:

*	*	*	*	*	…

Endkonfiguration ↑ S_1

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. Die Sprache vom Typ 0
3. Turingmaschinen
   - 3.1 Modell einer Turingmaschine
   - 3.2 Arbeitsweise einer Turingmaschine
   - **3.3 Beispiele für elementare Operationen einer TM**
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - 3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen

- Ein Zeichen nach rechts (**r**):

  $\delta(s_0, x) = (s_f, x, r)$ für beliebiges Bandzeichen x.

- Linkes Wortende suchen (**L**):

  $\delta(s_0, x) = (s_0, x, l)$ für $x \neq *$; $\delta(s_0, *) = (s_f, *, h)$

- Aktuelles Zeichen a auf erstes Freizeichen rechts verschieben (**VR**): (Für jedes Eingabezeichen $a \in \Sigma$ wird dazu ein innerer Zustand s_a benötigt).

$\delta(s_0, a) = (s_a, *, r)$; $\delta(s_a, x) = (s_a, x, r)$ für $x \neq *$; $\delta(s_a, *) = (s_f, a, h)$

- Aktuelles Zeichen a auf erstes Freizeichen links kopieren (**KL**):

$\delta(s_0, a) = (s_a, a, l)$; $\delta(s_a, x) = (s_a, x, l)$ für $x \neq *$; $\delta(s_a, *) = (s_f, a, h)$

- Bedingtes Anhalten in Abhängigkeit vom Feldinhalt * (**B(*)**):

$\delta(s_0, x) = (s_{f1}, x, h)$ für $x \neq *$; $\delta(s_0, *) = (s_{f2}, *, h)$

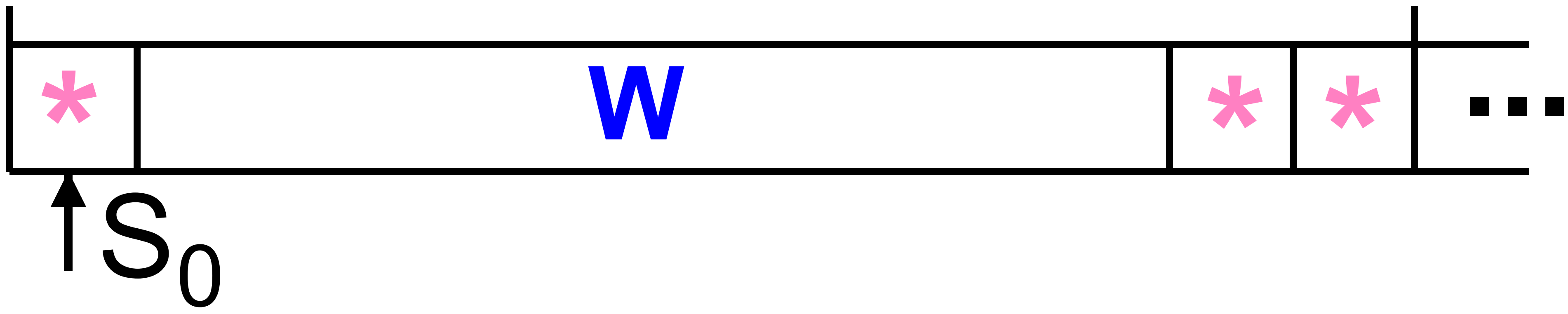
- Zusammengesetzte Turingmaschine zum Kopieren eines Wortes:
  (SLK stehe im Ausgangszustand s_0 auf dem Freizeichen links des Eingabewortes.)

```
r  ---->   VR-VR-KL-VL-r-B(*)  ---->  ende
      |                            |
      ---------------<--------------
```

V. Turingmaschinen und Kontextsensitive Sprachen

<u>Ausgangslage</u> (Initialkonfiguration):

In der Ausgangssituation möge ein Wort w aus Σ*, eingegrenzt durch zwei Leerzeichen, auf dem Band stehen. Die Maschine sei im Anfangszustand s_0 und der SLK stehe auf dem Leerzeichen am Anfang (des Bandes).



Definition (Sprache einer TM = (**S**, s_0, **F**, Σ, **B**, δ)):

Das Wort w wird von der Turingmaschine TM ***akzeptiert***, wenn nach einer Folge von Zwischenschritten eine Endsituation entsteht, bei der sich die TM in einem **internen Endzustand** s $\in$ **F** befindet **und** der SLK wieder auf dem Leerzeichen am Anfang des Bandes steht. Auf den in der Endsituation vorhandenen Bandinhalt kommt es **nicht** an.

Unter der *Sprache* ***L(TM)*** *einer Turingmaschine* ***TM*** versteht man alle Worte, die von der TM akzeptiert werden.

Endzustand:

Beispiel: $\Rightarrow$ Turing-Maschine **TM** = (**S**, S_0, **F**, Σ, **B**, δ) zum Erkennen der Sprache $L = \{ a^n b^n c^n \mid n = 1,2,\ldots\}$

hier:
n = 3

*	a	a	a	b	b	b	c	c	c	*	...

$\uparrow S_0$

Grundidee: $\uparrow$ = **SLK**

- am weitesten links stehendes **a** markieren $\Rightarrow$ durch **A** ersetzen
- das erste **b** suchen und durch **B** ersetzen
- wenn dies erfolgreich, dann erstes **c** suchen und durch **C** ersetzen
- danach läuft **SLK** zurück und Vorgang beginnt von neuem

$\Rightarrow$ Wird kein **a** mehr gefunden, so darf auch kein **b** und kein **c** mehr auf dem Band stehen.

hier:
n = 3

*	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	*	...

↑ S_0

Umsetzung:

- S_0: Anfangszustand ⇒ **SLK** auf *
- S_1: **SLK** erwartet ein **a**, dann ersetze **a** ⇒ **A**, oder **SLK** liest **B** ⇒ kein **a** mehr vorhanden
- S_2: **SLK** geht nach rechts, um erstes **b** zu finden, dann ersetze **b** ⇒ **B**
- S_3: analog zu S_2, lediglich mit **c** ⇒ **C**

hier:
n = 3

*	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	*	**...**

↑ S_0

Umsetzung (Fortsetzung):

- S_4: **SLK** wieder ganz nach links auf das erste **A** (von rechts);
  SLK wird dann auf dem nächsten rechten Feld positioniert
- S_5: prüft, ob kein **b** und kein **c** mehr auf dem Band vorhanden
- S_6: bringt **SLK** wieder in die Ausgangslage
- S_7: einziger Endzustand $\Rightarrow$ **SLK** wieder auf *

$\Rightarrow$ folglich:

hier:
n = 3

*	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	*	...

↑ S_0

Ergebnis:

Turing-Maschine **TM** = (**S**, S_0, **F**, Σ, **B**, δ) mit

S_0 = Anfangszustand

Σ = { **a**, **b**, **c** };

B = { **a**, **b**, **c**, * , **A**, **B**, **C** };

S = { S_0, S_1, ... , S_7 };

F = { S_7 }

Maschinentafel der **TM**:

	a	b	c	A	B	C	*
S_0	-	-	-	-	-	-	(S_1,*, r)
S_1	(S_2,**A**, r)	-	-	-	(S_5,**B**, r)	-	-
S_2	(S_2,**a**, r)	(S_3,**B**, r)	-	-	(S_2,**B**, r)	-	-
S_3	-	(S_3,**b**, r)	(S_4,**C**, l)	-	-	(S_3,**C**, r)	-
S_4	(S_4,**a**, l)	(S_4,**b**, l)	-	(S_1,**A**, r)	(S_4,**B**, l)	(S_4,**C**, l)	-
S_5	-	-	-	-	(S_5,**B**, r)	(S_5,**C**, r)	(S_6,*, l)
S_6	-	-	-	(S_6,**A**, l)	(S_6,**B**, l)	(S_6,**C**, l)	(S_7,*, h)

Konfigurationsfolge für w = abc:

$t = 0$	* ↑ S_0	a	b	c	*	*	*	...
$t = 1$	*	a ↑ S_1	b	c	*	*	*	...
$t = 2$	*	A	b ↑ S_2	c	*	*	*	...
$t = 3$	*	A	B	c ↑ S_3	*	*	*	...
$t = 4$	*	A	B ↑ S_4	C	*	*	*	...
$t = 5$	*	A ↑ S_4	B	C	*	*	*	...

$t = 6$	*	A	B ↑ S_1	C	*	*	*	...
$t = 7$	*	A	B	C ↑ S_5	*	*	*	...
$t = 8$	*	A	B	C	* ↑ S_5	*	*	...
$t = 9$	*	A	B	C ↑ S_6	*	*	*	...
$t = 10$	*	A	B ↑ S_6	C	*	*	*	...
$t = 11$	*	A ↑ S_6	B	C	*	*	*	...

t = 12	* ↑ S_6	**A**	**B**	**C**	*	*	*	...
t = 13	* ↑ S_7	**A**	**B**	**C**	*	*	*	...

$S_7 \in \mathbf{F}$

$\Rightarrow$ $w = abc \in L(\mathbf{TM})$

V. Turingmaschinen und Kontextsensitive Sprachen

Eine Turingmaschine zur Berechnung der Funktion:

$$f(TM) := \begin{cases} 1, & \text{falls } w = 1^n0^n \text{ für } n > 0 \\ 0, & \text{sonst} \end{cases}$$

Anfangskonfiguration (* = Leerzeichen, S_0 = Anfangszustand):

* ↑ S_0	**Eingabewort w**	*	*	*	...

Endkonfiguration (S_f = Endzustand):

*	*	*	**0 / 1** ↑ S_f	*	*	*	*	...

TM = (S, S_0, F, Σ, B, δ)

mit **S** = { S_0, S_1, ..., S_6 }, **F** = { S_6 }, Σ = { 0, 1 } und **B** = { 0, 1, * }:

Maschinentafel:

δ	**0**	**1**	*****
S_0	-	-	(S_1, *, r)
S_1	(S_5, 0, r)	(S_2, *, r)	(S_5, 1, r)
S_2	(S_2, 0, r)	(S_2, 1, r)	(S_3, *, l)
S_3	(S_4, *, l)	(S_5, 0, r)	(S_5, 0, r)
S_4	(S_4, 0, l)	(S_4, 1, l)	(S_1, *, r)
S_5	(S_6, *, l)	(S_6, *, l)	(S_6, *, l)

Funktionsberechnung für w = **1100**:

t = 0	* ↑ S_0	1	1	0	0	*	*	…
t = 1	*	1 ↑ S_1	1	0	0	*	*	…
t = 2	*	*	1 ↑ S_2	0	0	*	*	…
t = 3	*	*	1	0 ↑ S_2	0	*	*	…
t = 4	*	*	1	0	0 ↑ S_2	*	*	…
t = 5	*	*	1	0	0	* ↑ S_2	*	…

t = 6	*	*	1	0	0 ↑ S_3	*	*	…
t = 7	*	*	1	0 ↑ S_4	*	*	*	…
t = 8	*	*	1 ↑ S_4	0	*	*	*	…
t = 9	*	* ↑ S_4	1	0	*	*	*	…
t = 10	*	*	1 ↑ S_1	0	*	*	*	…
t = 11	*	*	*	0 ↑ S_2	*	*	*	…

t = 12	*	*	*	0	* ↑ S_2	*	*	...
t = 13	*	*	*	0 ↑ S_3	*	*	*	...
t = 14	*	*	* ↑ S_4	*	*	*	*	...
t = 15	*	*	*	* ↑ S_1	*	*	*	...
t = 16	*	*	*	**1**	* ↑ S_5	*	*	...
t = 17	*	*	*	**1** ↑ S_6	*	*	*	...

⇒ Funktionswert f(w = 1100) = 1

Nichtdeterministische Turingmaschinen:

Bei diesen gilt für die Überführungsfunktion

$$\delta: \mathbf{S} \times \mathbf{B} \rightarrow P(\mathbf{S} \times \mathbf{B} \times \mathbf{X}).$$

Turingmaschinen mit mehreren Bändern:

Bei diesen wird auf mehreren gleichartigen Bändern gearbeitet. Die Überführungsfunktion hat dabei die Form

$$\delta: \mathbf{S} \times \mathbf{B}^k \rightarrow \mathbf{S} \times \mathbf{B}^k \times \mathbf{X}^k$$

Solche Maschinen arbeiten **effizienter** als eine Turingmaschine mit nur einem Band.

Sätze:

1. Zu jeder als Akzeptor entworfenen nichtdeterministischen Turingmaschine gibt es eine deterministische, die die **gleiche** Sprache akzeptiert.
2. Zu jeder als Akzeptor entworfenen Turingmaschine mit mehreren Bändern gibt es eine mit **nur einem** Band, die die **gleiche** Sprache akzeptiert.
3. Zu jeder Sprache **L(G)** einer Grammatik vom **Typ 0** gibt es eine Turingmaschine **TM** mit **L(G) = L(TM)** und umgekehrt.
4. Es ist **nicht entscheidbar**, ob eine beliebige TM bei der Abarbeitung eines beliebigen Wortes anhält oder nicht (sog. **Halteproblem** bei TM → **Terminierung eines Algorithmus**).

V. Turingmaschinen und Kontextsensitive Sprachen

1. Kontextsensitive Sprachen
2. Die Sprache vom Typ 0
3. Turingmaschinen
   - 3.1 Modell einer Turingmaschine
   - 3.2 Arbeitsweise einer Turingmaschine
   - 3.3 Beispiele für elementare Operationen einer TM
   - 3.4 Turingmaschinen als Akzeptoren
   - 3.5 Turingmaschinen zur Funktionsberechnung
   - **3.6 Linear beschränkte Automaten (LBA) und Typ-1-Sprachen**

Bei **Linear beschränkte Automaten** (LBA) handelt sich um als Akzeptoren entworfene Turingmaschinen, die hinsichtlich ihrer Arbeitsweise einer bestimmten Beschränkung unterliegen: ***Die zur Verfügung stehende Länge des Bandes ist linear abhängig von der Länge des zu untersuchenden Eingabewortes.***

Satz:

Zu jeder Sprache L(G) einer Grammatik vom Typ 1 (monotone Grammatiken) gibt es einen (nichtdeterministischen) LBA mit

L(G) = L(LBA)

und umgekehrt.

Bemerkungen:

Ob ***deterministische*** linear beschränkte Automaten die gleiche Mächtigkeit als Akzeptoren haben wie die ***nichtdeterministischen***, ist eine Frage, die im Gegensatz zu den anderen behandelten Automatentypen bisher **noch nicht** beantwortet werden konnte.

Das ***Halteproblem*** für **LBA** ist entscheidbar, das heißt:

> Für jedes Wort und jeden LBA ist entscheidbar, ob der LBA das Wort akzeptiert oder nicht.