

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

# **Informatik IV**

Kurzkriptum zur Vorlesung

Sommersemester 2004

F. Kröger

INSTITUT FÜR INFORMATIK

Lehr- und Forschungseinheit für Programmierung und Softwaretechnik

# Inhaltsverzeichnis

<b>Einleitung</b>	<b>3</b>
<b>1 Grundlegende Begriffe und Konzepte</b>	<b>5</b>
1.1 Sprachen . . . . .	5
1.2 Turing-Maschinen . . . . .	9
1.3 Berechnungsbegriffe für Turing-Maschinen . . . . .	12
<b>2 Berechenbare Funktionen</b>	<b>16</b>
2.1 Programme . . . . .	16
2.2 Rekursive Funktionen . . . . .	18
2.3 WHILE- und LOOP-Programme . . . . .	23
2.4 Einige Hauptergebnisse der Berechenbarkeitstheorie . . . . .	26
<b>3 Analyse und Erzeugung von Sprachen</b>	<b>30</b>
3.1 Analyse von Sprachen . . . . .	30
3.2 Entscheidungsprobleme . . . . .	31
3.3 Semi-Thue-Systeme und Chomsky-Grammatiken . . . . .	34
3.4 Erzeugung von Sprachen . . . . .	36
<b>4 Reguläre Sprachen</b>	<b>39</b>
4.1 Reguläre Grammatiken und endliche Automaten . . . . .	39
4.2 Reguläre Ausdrücke . . . . .	42
4.3 Minimalautomaten . . . . .	44
<b>5 Kontextfreie und kontextsensitive Sprachen</b>	<b>47</b>
5.1 Kontextfreie Sprachen . . . . .	47
5.2 Einige Eigenschaften kontextfreier Grammatiken . . . . .	49
5.3 Kellerautomaten . . . . .	53
5.4 Kontextsensitive Sprachen . . . . .	55
<b>6 Komplexität von Entscheidungsproblemen</b>	<b>59</b>
6.1 Komplexitätsklassen . . . . .	59
6.2 NP-vollständige Probleme . . . . .	62
6.3 Probleme außerhalb NP . . . . .	64

# Einleitung

## Allgemeine Ziele der Theoretischen Informatik

- Formale (d.h. mathematische), meist abstrahierende Modellbildungen von Phänomenen der „praktischen“ Informatik.
- Theoriebildung für diese Modelle, d.h.: Auffinden mathematischer Aussagen über die Begriffe.
- (zu erwarten:) Erkenntnisse, Anwendungsmöglichkeiten u.Ä. in Bezug auf die ursprünglichen Phänomene.

## Vorlesung Informatik IV

Inhalt: Einführung in einige (Kern-) Gebiete der Theoretischen Informatik.

Ausgangspunkt (Kernthema der Informatik): Verarbeitung von Daten durch Algorithmen.

Grundlegende Fragestellungen dazu:

- Was bedeutet es, dass eine Datenverarbeitungsaufgabe überhaupt algorithmisch lösbar ist, und welche Aufgaben sind algorithmisch lösbar, welche nicht?  
(Theorie: **Berechenbarkeit**)
- Wie lassen sich Algorithmen (als Programme einer Programmiersprache) syntaktisch definieren, und wie lässt sich algorithmisch feststellen, ob ein gegebenes Programm syntaktisch korrekt ist?  
(Theorie: **Formale Sprachen und Automaten**)
- Was bedeutet es, dass algorithmisch zu lösende Aufgaben „weniger“, „mehr“, „besonders“ abarbeitungs-aufwändig lösbar sind, und welche Aufgaben sind von welchem Aufwand?  
(Theorie: **Komplexitätstheorie**)

## Einige geschichtliche Meilensteine

Ab  $\approx$  1900: **Entscheidungsproblem** (Hilbert): Kann man für eine gegebene Formel eines Logikkalküls (z.B. Prädikatenlogik 1. Stufe) „entscheiden“, ob sie allgemeingültig ist?

$\approx$  1930-1940: Präzisierung der Begriffe „Algorithmus“, „Berechenbarkeit“, „Entscheidbarkeit“, ... (Gödel, Péter, Church, Kleene, Post u.a.).

ab  $\approx$  1955: Sprachbeschreibung durch Grammatiken, erkennende Automaten (Chomsky, Backus, Rabin, Scott, Mealy u.a.).

ab 1964: Präzisierung des Begriffs der Komplexität (Cook, Blum u.a.).

**Literaturhinweise**

Asteroth/Baier: *Theoretische Informatik*.  
Pearson Studium 2002.

Erk/Priese: *Theoretische Informatik. Eine umfassende Einführung*.  
Springer 2000.

Hopcroft/Motwani/Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*.  
Addison-Wesley 2002 (2. Auflage).

Sander/Stucky/Herschel: *Automaten-Sprachen-Berechenbarkeit*.  
Teubner 1992.

Schöning: *Theoretische Informatik - kurz gefaßt*.  
Spektrum Akademischer Verlag 2001 (4. Auflage).

Sudkamp: *Languages and Machines: An Introduction to the Theory of Computer Science*.  
Addison-Wesley 1988.

Wagner: *Einführung in die Theoretische Informatik*.  
Springer 1994.

Wegener: *Theoretische Informatik*.  
Teubner 1993.

Winter: *Theoretische Informatik*.  
Oldenbourg 2002.

# Kapitel 1

## Grundlegende Begriffe und Konzepte

### 1.1 Sprachen

#### Formalisierung der Fragestellungen

Fragestellungen aus der Einleitung:

1. Was bedeutet es, dass eine Datenverarbeitungsaufgabe algorithmisch lösbar ist, und welche Aufgaben sind algorithmisch lösbar, welche nicht?

Formal (Daten sind durch Zeichenketten dargestellt):

- Wie kann formal präzisiert werden, dass eine Funktion, die Zeichenketten aufeinander abbildet, algorithmisch „berechenbar“ ist, und welche derartigen Funktionen sind berechenbar, welche nicht?
2. Wie lassen sich Programmiersprachen syntaktisch definieren, und wie lässt sich algorithmisch feststellen, ob ein gegebenes Programm syntaktisch korrekt ist?

Formal (Programme sind Zeichenketten):

- Wie lassen sich Mengen von Zeichenketten konstruktiv beschreiben, und wie lässt sich algorithmisch feststellen, ob eine gegebene Zeichenkette Element einer gegebenen Menge von Zeichenketten ist?
3. Was bedeutet es, dass algorithmisch zu lösende Aufgaben „weniger“, „mehr“, „besonders“ abarbeitungs-aufwändig lösbar sind, und welche Aufgaben sind von welchem Aufwand?

Formal (gemäß 1. und 2.):

- Wie können Berechnungsaufwände für lösbare Aufgaben der Zeichenketten-„Bearbeitung“ klassifiziert werden, und wie ordnen sich konkrete Aufgaben in eine solche Klassifizierung ein?

#### Grundbegriffe

**Alphabet:** endliche Menge  $\Sigma$ ; Elemente heißen (in diesem Kontext) **Zeichen** (*Symbole*).

**Wort** (*Zeichenkette*, *Zeichenreihe*) über einem Alphabet  $\Sigma$ : Endliche Folge von Zeichen  $a_1, a_2, \dots, a_n \in \Sigma$  (geschrieben:  $a_1 a_2 \dots a_n$ ).

Bezeichnungen:

- $\Sigma^*$ : Menge aller Wörter über  $\Sigma$ .
- $\varepsilon$ : *leeres Wort*.
- $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ .

**Länge**  $|w|$  eines Wortes  $w$ : Anzahl der in  $w$  auftretenden Zeichen ( $|\varepsilon| = 0$ ).

$|w|_a$  ( $a \in \Sigma$ ): Anzahl der Vorkommen von  $a$ 's in  $w$ .

**Konkatenation** von Wörtern:  $\circ : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ ,  
 $a_1 \dots a_n \circ b_1 \dots b_m = a_1 \dots a_n b_1 \dots b_m$ .

**Teilwort**  $u$  eines Wortes  $v$ : Es gibt  $x, y \in \Sigma^*$  mit  $v = x \circ u \circ y$   
 ( $x = \varepsilon$ :  $u$  **Präfix** von  $v$ ).

**Spiegelung (Revertierung)** von Wörtern:  $R : \Sigma^* \rightarrow \Sigma^*$ ,  $w \mapsto w^R$ ,  
 definiert durch  $\varepsilon^R = \varepsilon$ ,  
 $(a \circ u)^R = u^R \circ a$  für  $a \in \Sigma, u \in \Sigma^*$ .

### Einige Eigenschaften von Wörtern

Für beliebige  $u, v, w \in \Sigma^*$ ,  $a \in \Sigma$  gilt:

- $\varepsilon$  und  $w$  sind Präfixe von  $w$ .
- $\varepsilon \circ w = w \circ \varepsilon = w$ .
- $u \circ (v \circ w) = (u \circ v) \circ w$ .
- $|v \circ w| = |v| + |w|$ .
- $|v \circ w|_a = |v|_a + |w|_a$ .
- $(v \circ w)^R = w^R \circ v^R$ .

**Schreibweisen** (für Zeichen oder Wörter  $u, v, w$ )

- $uv$  statt  $u \circ v$ ,  $uvw$  statt  $(uv)w, \dots$
- $w^n$  für  $\underbrace{ww \dots w}_{n\text{-mal}}$  ( $w^0 = \varepsilon$ ).

### Sprachen

**Definition.** Eine (*formale*) **Sprache** über einem Alphabet  $\Sigma$  ist eine Teilmenge von  $\Sigma^*$ . Die leere Teilmenge  $\emptyset$  heißt *leere Sprache*.

Sind  $L_1, L_2 \subseteq \Sigma^*$  zwei Sprachen, so sind ihr **Durchschnitt** und ihre **Vereinigung** gegeben durch  $L_1 \cap L_2$  bzw.  $L_1 \cup L_2$ . Das **Komplement**  $\bar{L}$  einer Sprache  $L \subseteq \Sigma^*$  ist  $\Sigma^* \setminus L$ .

Weitere Operationen ( $L, L_1, L_2 \subseteq \Sigma^*$ ):

- **Sprachprodukt (Konkatenation von Sprachen):**

$$L_1 \circ L_2 = \{vw \mid v \in L_1, w \in L_2\}.$$

(Schreibweise auch:  $L_1L_2$ .)

- **Sprachpotenz:**

$$L^n = \{w_1w_2 \dots w_n \mid w_i \in L \text{ für } i \in \mathbb{N}\} \quad (\text{für } n \in \mathbb{N}).$$

(Beachte:  $L^0 = \{\varepsilon\}$ ,  $L^1 = L$ ,  $L^mL^n = L^{m+n}$ ,  $(L^m)^n = L^{m \cdot n}$ .)

- **Kleene-Stern:**

$$L^* = \bigcup_{n \geq 0} L^n \quad (\text{außerdem: } L^+ = \bigcup_{n > 0} L^n).$$

- **Spiegelung:**

$$L^R = \{w^R \mid w \in L\}.$$

## Einige Eigenschaften von Sprachen

**Satz 1.1.1** Für beliebige  $L, L_1, L_2 \subseteq \Sigma^*$  gilt:

- $\overline{\overline{L}} = L$ .
- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ .
- $L(L_1 \cup L_2) = LL_1 \cup LL_2$ .
- $(L^*)^* = L^*$ .
- $(L_1 \cup L_2)^R = L_1^R \cup L_2^R$ .
- $(L_1L_2)^R = L_2^R L_1^R$ .
- $(L^*)^R = (L^R)^*$ .

## Interpretation von Sprachen und Funktionen

- Sprachen  $L \subseteq \Sigma^*$  und Funktionen  $f : \Sigma^* \rightarrow \Sigma^*$  bilden einen einheitlichen Rahmen für die formale Behandlung der in der Einleitung genannten Fragestellungen. Sprachen  $L$  können dabei Darstellungen (**Kodierungen**) konkreter Datenmengen, Funktionen  $f$  Kodierungen von Funktionen auf konkreten Daten sein. Die konkreten Datenmengen und Funktionen sind **Interpretationen** von  $L$  bzw.  $f$ .
- Mögliche Sichtweisen bei der Behandlung von Sprachen  $L \subseteq \Sigma^*$  und Funktionen  $f : \Sigma^* \rightarrow \Sigma^*$ :
  - „Uninterpretiert“: Betrachtet wird nur die Zeichenketten-„Gestalt“ der bei  $L$  bzw.  $f$  behandelten Wörter ohne Bezug auf irgendeine Interpretation.
  - „Interpretiert“:  $L$  bzw.  $f$  werden im Hinblick auf eine konkrete Interpretation betrachtet.

### Beispiele interpretierter Sprachen

- Sei  $\Sigma = \{0, 1\}$ .

$$L_{bin} = \{w \in \Sigma^* \mid w = 0 \text{ oder } w \text{ beginnt mit dem Zeichen } 1\}.$$

Interpretation: Jedes  $w \in L_{bin}$  ist die Binärdarstellung (ohne führende Nullen) einer natürlichen Zahl.

- Sei  $\Sigma = \{\mid\}$ .

$$L_{prim} = \{\mid^p \in \Sigma^* \mid p \text{ ist Primzahl}\}.$$

Interpretation: Jedes  $w = \mid^p \in L_{prim}$  stellt eine Primzahl  $p$  (in **Unär- (Strich-) Darstellung**) dar.

- Sei  $\Sigma = \{\mid, \circ\}$ ,  $k \in \mathbb{N}$ .

$$N_k = \{\mid^{n_1} \circ \mid^{n_2} \circ \dots \circ \mid^{n_k} \in \Sigma^* \mid n_1, n_2, \dots, n_k \in \mathbb{N}\}.$$

Interpretation: Jedes  $w = \mid^{n_1} \circ \mid^{n_2} \circ \dots \circ \mid^{n_k} \in N_k$  stellt das  $k$ -Tupel  $(n_1, n_2, \dots, n_k) \in \mathbb{N}^k$  (gemäß der Unärdarstellung der einzelnen  $n_i$ ) dar.

- Sei  $\Sigma = \{\mid, \circ, (, )\}$ .

$$L_{graph} = \{(\mid^{i_1} \circ \mid^{j_1})(\mid^{i_2} \circ \mid^{j_2}) \dots (\mid^{i_m} \circ \mid^{j_m}) \in \Sigma^* \mid i_1, \dots, i_m, j_1, \dots, j_m \in \mathbb{N}\}.$$

Interpretation: Jedes  $w = (\mid^{i_1} \circ \mid^{j_1}) \dots (\mid^{i_m} \circ \mid^{j_m}) \in L_{graph}$  stellt einen Graphen mit der Knotenmenge  $\{v_{i_1}, v_{j_1} \dots\}$  und der Kantenmenge  $\{(v_{i_1}, v_{j_1}), \dots, (v_{i_m}, v_{j_m})\}$  dar.

### Zahlmengen und -funktionen

- Jede Teilmenge  $M$  von  $\mathbb{N}^k$  ( $k \in \mathbb{N}$ ) ist Interpretation der Teilsprache

$$L_M = \{\mid^{n_1} \circ \mid^{n_2} \circ \dots \circ \mid^{n_k} \in N_k \mid (n_1, n_2, \dots, n_k) \in M\}$$

von  $N_k$  (aus obigem Beispiel). Solche Mengen  $M$  heißen **Zahlmengen**.

- Jede Funktion  $f : \{\mid, \circ\}^* \rightarrow \{\mid, \circ\}^*$  mit der Eigenschaft

Falls  $f(w)$  für  $w = \mid^{n_1} \circ \mid^{n_2} \circ \dots \circ \mid^{n_k} \in N_k$  definiert ist, so ist  $f(w) = \mid^m$  für ein (von  $n_1, n_2, \dots, n_k$  abhängiges)  $m \in \mathbb{N}$ .

kann interpretiert werden als (partielle) Funktion  $f_z : \mathbb{N}^k \rightarrow \mathbb{N}$  mit

$$f_z(n_1, n_2, \dots, n_k) = \begin{cases} m, & \text{falls } f(\mid^{n_1} \circ \mid^{n_2} \circ \dots \circ \mid^{n_k}) \text{ definiert} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Zur Unterscheidung heißen Funktionen  $\mathbb{N}^k \rightarrow \mathbb{N}$  auch **Zahlfunktionen**, Funktionen  $\Sigma^* \rightarrow \Sigma^*$  **Wortfunktionen**.

- Zahlfunktionen der speziellen Art  $\mathbb{N}^k \rightarrow \{0, 1\}$  heißen auch (**Zahl- Prädikate**). Sie interpretieren Wortfunktionen  $f : \{\mid, \circ\}^* \rightarrow \{\mid, \circ\}^*$  mit  $f(w) \in \{\varepsilon, \mid\}$  für  $w \in N_k$ .

## Formale Bearbeitungs-Modelle

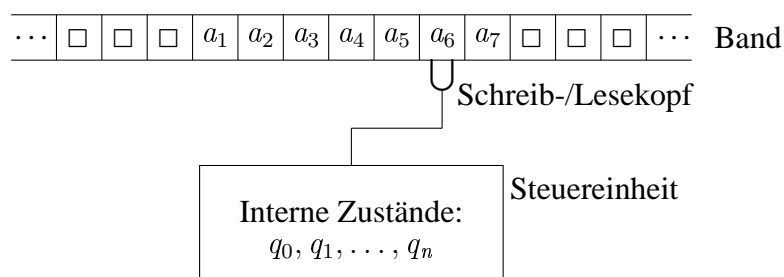
- Die zu Beginn des Abschnitts formulierten Fragestellungen konzentrieren sich auf die algorithmische Bearbeitung von Wörtern und Sprachen, genauer:
  - Berechnung von (i.Allg. partiellen) Funktionen  $f : \Sigma^* \rightarrow \Sigma^*$  (für ein Alphabet  $\Sigma$ ),
  - Konstruktive Beschreibung (**Erzeugung**) von Sprachen und Feststellung des Enthaltenseins von Wörtern in Sprachen (**Analyse** von Sprachen),
  - Aufwand solcher algorithmischen Verfahren.
- Zur formalen Behandlung der Fragen braucht man ein formales Modell für den intuitiven Begriff der „algorithmischen Bearbeitung“. Zwei grundlegende derartige Modelle sind:
  - **Turing-Maschinen:**  
eingeführt 1936 vom englischen Mathematiker A. TURING als ein grundlegendes „mechanisch“, „maschinell“ orientiertes Modell zur „automatischen“ Bearbeitung von Zeichenketten,
  - **Semi-Thue-Systeme** (speziell in der Form von **Chomsky-Grammatiken**):  
entwickelt ab 1914 vom norwegischen Mathematiker A. THUE zur systematischen Manipulation von Zeichenketten; im Bereich der Linguistik modifiziert verwendet ab 1956 vom amerikanischen Sprachwissenschaftler N. CHOMSKY.

(Turing-Maschinen werden im Folgenden eingeführt, Semi-Thue-Systeme in Abschnitt 3.3.)

## 1.2 Turing-Maschinen

### Grundidee

Eine Turing-Maschine kann man sich vorstellen als eine (hypothetische) Maschine, die aus einer Steuereinheit und einem (beidseitig unendlichen) Speicherband besteht:



Das Band ist in einzelne Felder unterteilt, die jeweils genau ein Zeichen aufnehmen können. Die Steuereinheit kann endlich viele interne Zustände annehmen und gemäß einer „Überföhrungsfunktion“ einen Schreib-/Lesekopf über die Bandfelder bewegen (pro Rechenschritt jeweils um 1 Feld nach „links“ oder nach „rechts“). Ferner kann sie dabei den Inhalt des jeweils aktuellen („Arbeits-“) Feldes lesen und/oder überschreiben. Die Kennzeichnung von „leeren“ Feldern geschieht mit einem ausgezeichneten Leerzeichen ( $\square$ ).

### Nichtdeterministische Turing-Maschinen

**Definition.** Eine (*nichtdeterministische*) *Turing-Maschine* (*NTM*, *TM*)  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  ist gegeben durch:

- eine endliche Menge  $Q$  von **Zuständen**,
- ein **Eingabealphabet**  $\Sigma$ ,
- ein **Bandalphabet**  $\Gamma \supseteq \Sigma$  mit einem **Leerzeichen**  $\square \in \Gamma \setminus \Sigma$ ,
- eine totale **Überföhrungsfunktion**  $\delta : Q \times \Gamma \rightarrow \mathfrak{P}(Q \times \Gamma \times \{L, R, N\})$ ,
- einen **Startzustand**  $q_0 \in Q$ ,
- einen **akzeptierenden Zustand**  $q_+ \in Q$ .

Eine **Konfiguration** von  $M$  ist ein Element  $K \in \Gamma^* \times Q \times \Gamma^*$ .

**Beispiel:** NTM  $M_{bsp1}$  mit  $Q = \{q_0, q_1, q_2, q_3\}$ ,  $\Sigma = \{a, b\}$ ,  $\Gamma = \{a, b, \square\}$ ,  $q_+ = q_2$  und der Überföhrungsfunktion

$$\begin{aligned} \delta : (q_0, a) &\mapsto \{(q_0, b, R)\}, \\ (q_0, b) &\mapsto \{(q_0, a, R)\}, \\ (q_0, \square) &\mapsto \{(q_1, \square, L)\}, \\ (q_1, a) &\mapsto \{(q_1, a, L)\}, \\ (q_1, b) &\mapsto \{(q_1, b, L), (q_1, b, N)\}, \\ (q_1, \square) &\mapsto \{(q_2, \square, R)\}, \\ (q_2, a) &\mapsto \{(q_2, b, R), (q_2, \square, N), (q_3, a, R)\}, \\ (q, c) &\mapsto \emptyset \quad \text{für alle anderen } (q, c) \in Q \times \Sigma. \end{aligned}$$

### Rechnungen von Turing-Maschinen

Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine NTM. Die binäre **Übergangsrelation**  $\vdash_M$  auf der Menge der Konfigurationen von  $M$  ist definiert wie folgt: Ist  $(q', c, X) \in \delta(q, b_1)$ , so gilt

$$(a_1 \dots a_m, q, b_1 \dots b_n) \vdash_M \begin{cases} (a_1 \dots a_m, q', cb_2 \dots b_n), & \text{falls } X = N \\ (a_1 \dots a_m c, q', b_2 \dots b_n), & \text{falls } X = R \\ (a_1 \dots a_{m-1}, q', a_m cb_2 \dots b_n), & \text{falls } X = L. \end{cases}$$

(Falls  $a_1 \dots a_m = \varepsilon$ , so ist im dritten Fall  $a_m = \square$ .)

Ist  $K \vdash_M K'$ , so heißt  $K'$  **Nachfolgekonfiguration** von  $K$ . Eine Konfiguration heißt **final**, wenn sie keine Nachfolgekonfiguration hat.

$\vdash_M^*$  sei die reflexive, transitive Hülle von  $\vdash_M$ . D.h.: Es gilt  $K \vdash_M^* K'$ , wenn es Konfigurationen  $K_0, \dots, K_l$  ( $l \in \mathbb{N}$ ) gibt mit

$$K = K_0 \vdash_M K_1 \vdash_M K_2 \vdash_M \dots \vdash_M K_l = K'.$$

Diese Folge der  $K_0, \dots, K_l$  heißt **Rechnung** der **Länge**  $l$  von  $M$ . Die Gesamtanzahl der verschiedenen Felder des Bandes, auf denen der Schreib-/Lesekopf in  $K_0, \dots, K_l$  zu stehen kommt, heißt **Bandverbrauch** der Rechnung.

(Wenn der Kontext klar ist, schreiben wir meist  $\vdash$  bzw.  $\vdash^*$  statt  $\vdash_M$  und  $\vdash_M^*$ ).

### Beschreibung von Turing-Maschinen

Turing-Maschinen (genauer: ihre Überföhrungsfunktionen) werden in verschiedenen Notationen angegeben, z.B.:

- Angabe jedes  $(q', b, X) \in \delta(q, a)$  in der Form

$$q \quad a \quad q' \quad b \quad X$$

Beispiel  $M_{bsp1}$ :

$$\begin{array}{l} q_0 \quad a \quad q_0 \quad b \quad R \\ q_0 \quad b \quad q_0 \quad a \quad R \\ q_0 \quad \square \quad q_1 \quad \square \quad L \\ q_1 \quad a \quad q_1 \quad a \quad L \\ q_1 \quad b \quad q_1 \quad b \quad L \\ q_1 \quad b \quad q_1 \quad b \quad N \\ q_1 \quad \square \quad q_2 \quad \square \quad R \\ q_2 \quad a \quad q_2 \quad b \quad R \\ q_2 \quad a \quad q_2 \quad \square \quad N \\ q_2 \quad a \quad q_3 \quad a \quad R \end{array}$$

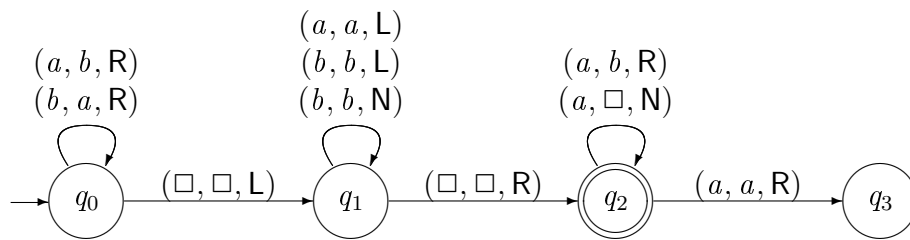
- Jedes  $(q', b, X) \in \delta(q, a)$  ist Eintrag in einer **Zustandstafel** (Matrix mit den Zuständen als Zeilen- und den Zeichen als Spaltennamen) in Zeile  $q$  und Spalte  $a$ .

Beispiel  $M_{bsp1}$ :

	$a$	$b$	$\square$
$q_0$	$(q_0, b, R)$	$(q_0, a, R)$	$(q_1, \square, L)$
$q_1$	$(q_1, a, L)$	$(q_1, b, L)$ $(q_1, b, N)$	$(q_2, \square, R)$
$q_2$	$(q_2, b, R)$ $(q_2, \square, N)$ $(q_3, a, R)$	—	—
$q_3$	—	—	—

- Darstellung als **Zustandsübergangsgraph** mit den Zuständen als Knoten. Der Startzustand und der akzeptierende Zustand werden (etwa durch einen „Eingangspfeil“ bzw. durch „Doppelumrandung“) gekennzeichnet. Für jeden möglichen Zustandsübergang von  $q$  nach  $q'$  enthält der Graph eine gerichtete Kante, die mit allen  $(a, b, X)$  mit  $(q', b, X) \in \delta(q, a)$  markiert ist.

Beispiel  $M_{bsp1}$ :



### Deterministische Turing-Maschinen

**Definition.** Eine Turing-Maschine  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  heißt **deterministisch (DTM)**, wenn  $\delta(q, a)$  für alle  $q \in Q, a \in \Gamma$  höchstens ein Element enthält.

Die Überföhrungsfunktion  $\delta$  einer DTM kann auch als (partielle) Funktion

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$$

(mit  $\delta(q, a) = (q', b, X)$  statt  $\delta(q, a) = \{(q', b, X)\}$  und undefiniertem  $\delta(q, a)$  statt  $\delta(q, a) = \emptyset$ ) verstanden werden.

**Beispiel:** Die folgende Abänderung  $M_{bsp2}$  von  $M_{bsp1}$  ist deterministisch:

	$a$	$b$	$\square$
$q_0$	$(q_0, b, R)$	$(q_0, a, R)$	$(q_1, \square, L)$
$q_1$	$(q_1, a, L)$	$(q_1, b, L)$	$(q_2, \square, R)$
$q_2$	$(q_3, a, N)$	—	—
$q_3$	—	—	—

## 1.3 Berechnungsbegriffe für Turing-Maschinen

### Turing-Maschinen als Akzeptoren

**Definition.** Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine Turing-Maschine.  $M$  **akzeptiert (erkennt)** das Wort  $w \in \Sigma^*$ , falls es eine finale Konfiguration  $(\alpha, q_+, \beta)$  gibt mit  $(\varepsilon, q_0, w) \vdash_M^* (\alpha, q_+, \beta)$ . Die betreffende Rechnung heißt **akzeptierend**. ( $M$  heißt in diesem Kontext auch **Akzeptor**.) Die Menge  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$  heißt die **von  $M$  akzeptierte Sprache**.

**Beispiele:**

1. Die NTM
- $M_{bsp3}$
- mit der Zustandstafel

	$a$	$b$	$\square$
$q_0$	$(q_0, a, R)$ $(q_1, a, R)$	$(q_0, b, R)$	—
$q_1$	—	$(q_2, b, R)$	—
$q_2$	$(q_+, a, N)$	—	—
$q_+$	—	—	—

akzeptiert die Sprache

$$L_{aba} = \{w \in \{a, b\}^* \mid aba \text{ ist Teilwort von } w\}.$$

2. Die DTM
- $M_{bsp4}$
- mit der Zustandstafel

	$a$	$b$	$\square$
$q_0$	$(q_1, a, R)$	$(q_0, b, R)$	—
$q_1$	$(q_1, a, R)$	$(q_2, b, R)$	—
$q_2$	$(q_+, a, N)$	$(q_0, b, R)$	—
$q_+$	—	—	—

akzeptiert ebenfalls die Sprache  $L_{aba}$ .**Determinismus versus Nichtdeterminismus**

Die beiden Beispiele  $M_{bsp3}$  und  $M_{bsp4}$  zeigen einen charakteristischen Unterschied zwischen deterministischen und („echt“) nichtdeterministischen Berechnungen: Beide TMen durchlaufen das Eingabewort von links nach rechts. Die DTM  $M_{bsp4}$  versucht bei jedem  $a$ , anschließend ein  $b$  und dann ein weiteres  $a$  zu finden. Das Grundprinzip lässt sich kurz so beschreiben:

- (1) „Suche“ die Stelle, an der das Teilwort  $aba$  beginnt.
- (2) „Verifiziere“ das Teilwort  $aba$ .

Dagegen „überläuft“ die NTM  $M_{bsp3}$  (in einer akzeptierenden Rechnung) alle auftretenden Zeichen  $a$ , die nicht Beginn eines Teilworts  $aba$  sind, und beginnt die Verifikation von  $aba$  erst bei einem geeigneten  $a$ . Dieses Grundprinzip kann so beschrieben werden:

- (1) „Errate“ die Stelle, an der das Teilwort  $aba$  beginnt.
- (2) „Verifiziere“ das Teilwort  $aba$ .

**Entscheidung von Sprachen**

**Definition.** Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine Turing-Maschine.  $M$  *terminiert für (die Eingabe)*  $w \in \Sigma^*$ , wenn es keine unendliche Folge  $K_0, K_1, K_2, \dots$  von Konfigurationen von  $M$  gibt mit  $K_0 = (\varepsilon, q_0, w)$  und  $K_i \vdash K_{i+1}$  für alle  $i \in \mathbb{N}$ .

**Definition.** Seien  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine Turing-Maschine und  $L \subseteq \Sigma^*$  eine Sprache.  $M$  *entscheidet*  $L$ , wenn  $M$  für alle  $w \in \Sigma^*$  terminiert und  $L = \mathcal{L}(M)$  ist.

**Beispiel:** Die TMen  $M_{bsp3}$  und  $M_{bsp4}$  in den obigen Beispielen entscheiden beide die Sprache  $L_{aba}$ .

### Erzeugung von Sprachen

**Definition.** Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine Turing-Maschine.  $M$  *erzeugt* das Wort  $w \in \Sigma^*$ , falls es eine finale Konfiguration  $(\varepsilon, q_+, w)$  gibt mit  $(\varepsilon, q_0, \varepsilon) \vdash_M^* (\varepsilon, q_+, w)$ .

Die Menge  $\mathcal{L}_{erz}(M) = \{w \in \Sigma^* \mid M \text{ erzeugt } w\}$  heißt die *von  $M$  erzeugte Sprache*.

**Beispiel:** Die NTM  $M_{bsp5}$  mit der Zustandstafel

	$a$	$b$	$\square$
$q_0$	—	—	$(q_0, a, \text{L})$ $(q_0, b, \text{L})$ $(q_1, a, \text{L})$
$q_1$	—	—	$(q_2, b, \text{L})$
$q_2$	—	—	$(q_3, a, \text{L})$
$q_3$	—	—	$(q_3, a, \text{L})$ $(q_3, b, \text{L})$ $(q_+, \square, \text{R})$
$q_+$	—	—	—

erzeugt die Sprache  $L_{aba}$ .

### Berechnung von Funktionen

**Definition.** Seien  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine deterministische Turing-Maschine und  $f : \Sigma^* \rightarrow \Sigma^*$  eine Funktion.  $M$  *berechnet*  $f$ , falls gilt:

1. Ist  $f(w)$  definiert, so ist  $(\varepsilon, q_0, w) \vdash_M^* (\varepsilon, q, v)$  für ein  $q \in Q$  und  $v = f(w)$ , und  $(\varepsilon, q, v)$  ist final.
2. Ist  $f(w)$  undefiniert, so terminiert  $M$  für  $w$  nicht.

**Bemerkung:** Für den Begriff der Berechnung einer Funktion ist der akzeptierende Zustand  $q_+$  der betreffenden DTM  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  offensichtlich ohne Bedeutung. Wir schreiben in derartigem Kontext auch einfacher  $M = (Q, \Sigma, \Gamma, \delta, q_0)$ .

### Beispiele

1. Die DTM  $M_{bsp2}$  aus Abschnitt 1.2 mit der Zustandstafel

	$a$	$b$	$\square$
$q_0$	$(q_0, b, \text{R})$	$(q_0, a, \text{R})$	$(q_1, \square, \text{L})$
$q_1$	$(q_1, a, \text{L})$	$(q_1, b, \text{L})$	$(q_2, \square, \text{R})$
$q_2$	$(q_3, a, \text{N})$	—	—
$q_3$	—	—	—

berechnet die (totale) Funktion  $f : \{a, b\}^* \rightarrow \{a, b\}^*$  mit

$$f(w_1 w_2 \dots w_n) = w'_1 w'_2 \dots w'_n \quad \text{und}$$

$$w'_i = \begin{cases} b, & \text{falls } w_i = a \\ a, & \text{falls } w_i = b \end{cases} \quad (i = 1, \dots, n).$$

2. Die DTM  $M_{add}$  mit  $\Sigma = \{ |, \circ \}$  und der Zustandstafel

		○	□
$q_0$	$(q_0,  , R)$	$(q_1,  , L)$	—
$q_1$	$(q_1,  , L)$	—	$(q_2, \square, R)$
$q_2$	$(q_3, \square, R)$	—	—
$q_3$	—	—	—

berechnet die Funktion  $f : \{ |, \circ \}^* \rightarrow \{ |, \circ \}^*$  mit

$$f(|^{n_1} \circ |^{n_2}) = |^{n_1+n_2}.$$

### Berechnung von Zahlfunktionen

Ist  $M$  eine DTM, die eine Funktion  $f : \{ |, \circ \}^* \rightarrow \{ |, \circ \}^*$  berechnet, die als Zahlfunktion (oder Zahlprädikat)  $f_z : \mathbb{N}^k \rightarrow \mathbb{N}$  interpretierbar ist, so sagen wir auch: „ $M$  berechnet  $f_z$ “.

### Aufwand algorithmischer Berechnungen

Als Aufwand algorithmischer Verfahren gilt typischerweise die Rechenzeit und/oder der Speicherbedarf. Im Turing-Maschinen-Modell ist die Rechenzeit durch die Länge einer Rechnung, der Speicherbedarf durch ihren Bandverbrauch formalisiert.

### Varianten von Turing-Maschinen

Neben dem hier definierten Turing-Maschinen-Modell gibt es Varianten, z.B.:

- Turing-Maschinen mit mehr als einem akzeptierenden Zustand.
- Turing-Maschinen mit nur einseitig unendlichem Band.
- **Mehrband-TM**: Eine **2-Band-TM** zum Beispiel besitzt zwei Bänder, und die Überföhrungsfunktion ist von der Art

$$\delta : Q \times \Gamma \times \Gamma \rightarrow \mathfrak{P}(Q \times \Gamma \times \Gamma \times \{L, R, N\} \times \{L, R, N\}),$$

wobei  $(q', b_1, b_2, X_1, X_2) \in \delta(q, a_1, a_2)$  informell bedeutet: Liest die Maschine im Zustand  $q$  auf dem ersten Band  $a_1$  und auf dem zweiten Band  $a_2$ , so kann sie diese Zeichen durch  $b_1$  bzw.  $b_2$  ersetzen, die (unabhängigen) Schreib-/Leseköppe gemäß  $X_1$  bzw.  $X_2$  bewegen und in den Zustand  $q'$  übergöhen.

Alle diese TM-Modelle sind bezüglich ihrer Verarbeitungs-„Mächtigkeit“ gleich.

# Kapitel 2

## Berechenbare Funktionen

### 2.1 Programme

#### Turing-Berechenbarkeit

- Zur Erinnerung (Abschnitt 1.3):  
Eine DTM  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  berechnet eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$ , falls gilt:
  1. Ist  $f(w)$  definiert, so ist  $(\varepsilon, q_0, w) \vdash_M^* (\varepsilon, q, v)$  für ein  $q \in Q$  und  $v = f(w)$ , und  $(\varepsilon, q, v)$  ist final.
  2. Ist  $f(w)$  undefiniert, so terminiert  $M$  für  $w$  nicht.
- Eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt (**Turing-**) **berechenbar**, wenn es eine DTM gibt, die  $f$  berechnet.

#### Die Churchsche These

- Frage: Ist der Begriff der Turing-Berechenbarkeit „umfassend“?
- **Churchsche These:** Jede im intuitiven Sinn algorithmisch berechenbare Funktion ist (in kodierter Form als Funktion  $f : \Sigma^* \rightarrow \Sigma^*$ ) Turing-berechenbar. D.h.: Der Begriff der Turing-Berechenbarkeit ist tatsächlich eine adäquate, umfassende Formalisierung des intuitiven Berechenbarkeits-Begriffs.
- Wesentliches Argument für diese These: Es wurden viele andere „plausible“ Formalisierungen des Berechenbarkeits-Begriffs vorgenommen, die sich alle als gleichmächtig zur Turing-Berechenbarkeit erwiesen haben.
- In diesem Kapitel: Illustration dieser Tatsache anhand einiger anderer Formalisierungen.

Einschränkung dabei: Wir betrachten nur Funktionen  $f : \{ |, \circ \}^* \rightarrow \{ |, \circ \}^*$  der Art

$$f : |^{n_1} \circ |^{n_2} \circ \dots \circ |^{n_k} \mapsto |^{f_z(n_1, n_2, \dots, n_k)} \quad (\text{falls definiert})$$

(dies bedeutet keine wirkliche Einschränkung der Allgemeinheit), die als Zahlfunktionen  $f_z : \mathbb{N}^k \rightarrow \mathbb{N}$  interpretiert werden können.

Sprechweise: Falls  $f$  berechenbar ist, so: „ $f_z$  ist (Turing-) berechenbar“.

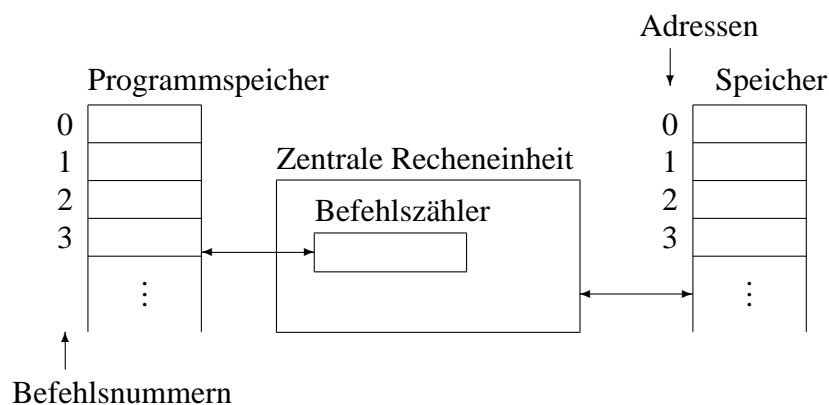
- Churchsche These in dieser speziell interpretierten Version (o.B.d.A.): Jede im intuitiven Sinn algorithmisch berechenbare Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  ist Turing-berechenbar.

## Register-Maschinen

- Die Überföhrungsfunktion  $\delta$  einer (D)TM kann als ein Programm in einer maschinennahen Programmiersprache für eine sehr einfache Rechenmaschine (gemäß dem Bild in Abschnitt 1.2) aufgefasst werden.
- Andere Berechenbarkeitsbegriffe: Programme auf anderen („realistischeren“) Maschinen-Modellen.

### **Register-Maschinen (RAM, URM):**

Eine solche Maschine hat einen Speicher mit adressierbaren Registern (Speicherzellen), in denen Wörter gespeichert werden können, und einen ebenso adressierbaren Programmspeicher, in dem Befehle einer zugehörigen Maschinensprache gespeichert werden können:



## Programmierung von Register-Maschinen

- Typische Befehle der Maschinensprache einer RAM sind „Transport-“ Befehle für Wörter im Speicher, „Sprünge“ auf bestimmte Befehle sowie Befehle für einfache Wortfunktionen.
- Man kann formal definieren, was es heißt, dass eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  (bzw. eine Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ) **RAM-berechenbar** (d.h. durch ein Programm auf einer Register-Maschine berechenbar) ist. Es gilt dann:  $f$  ist genau dann RAM-berechenbar, wenn sie Turing-berechenbar ist.
- Analog zu „realen“ Programmiersprachen: Man kann „höhere“ (problemorientierte) Programmiersprachen zur Programmierung von Registermaschinen verwenden und die Berechenbarkeit einer Funktion durch ein Programm einer solchen Sprache definieren. Dabei typisch:
  - Die Programmiersprache nimmt keinen Bezug mehr auf das zugrunde liegende Maschinen-Modell.
  - Programme verwenden die zu verarbeitenden Daten (im Fall von Zahlfunktionen: natürliche Zahlen) direkt und nicht in einer Kodierung.

- Zwei grundlegende Paradigmen bei problemorientierter Programmierung:
  - Imperative Programmierung: Variablen als symbolische Adressen von Speicherzellen einer Register-Maschine; Berechnung durch gesteuerte Abfolge von Anweisungen zur Veränderung von Werten im Speicher; wesentliches Steuerungskonzept: Wiederholungsanweisungen.
  - Funktionale Programmierung: Direkte („mathematisch orientierte“) Darstellung von Funktionen ohne jeglichen Bezug auf Speicherung von Werten; Berechnung durch gesteuerte Abfolge von Funktions-Auswertungen; wesentliches Steuerungskonzept: Rekursion.

## 2.2 Rekursive Funktionen

### Die Programmiersprache PRIMREK

PRIMREK: Funktionale Programmiersprache zur Formulierung von Algorithmen, die Zahlfunktionen berechnen.

### Syntax von PRIMREK

Sei  $\{e_1, e_2, e_3, \dots\}$  eine (fest vorgegebene unendliche) Menge von *Parametern*. Ein PRIMREK-*Programm* hat die Gestalt

$$\mathbf{fn}(e_1, \dots, e_k) : t$$

oder die Gestalt

$$\mathbf{fn}(e_1, \dots, e_{k+1}) : t \mid t'$$

( $k \in \mathbb{N}$ ), wobei  $t$  ein  $k$ -Term und  $t'$  ein  $(k+2)$ -Term sind. Für  $l \in \mathbb{N}$  ist ein  *$l$ -Term* dabei wie folgt induktiv definiert:

1. 0 und alle Parameter  $e_i$  mit  $1 \leq i \leq l$  sind  $l$ -Terme.
2. Ist  $t$  ein  $l$ -Term, so ist  $t + 1$  ein  $l$ -Term.
3. Ist  $P \equiv \mathbf{fn}(e_1, \dots, e_m) : \dots$  ein PRIMREK-Programm und sind  $t_1, \dots, t_m$   $l$ -Terme, so ist  $P(t_1, \dots, t_m)$  ein  $l$ -Term.

### Semantik von PRIMREK

Die von einem PRIMREK-Programm  $P$  *berechnete* Funktion  $f_P$  ist direkt angebar:

- Ist  $P$  von der Gestalt  $\mathbf{fn}(e_1, \dots, e_k) : t$ , so ist  $f_P : \mathbb{N}^k \rightarrow \mathbb{N}$  mit:

$$f_P(x_1, \dots, x_k) = \llbracket t \rrbracket_{x_1, \dots, x_k}.$$

- Ist  $P$  von der Gestalt  $\mathbf{fn}(e_1, \dots, e_{k+1}) : t \mid t'$ , so ist  $f_P : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  die rekursiv definierte Funktion

$$\begin{aligned} f_P(x_1, \dots, x_k, 0) &= \llbracket t \rrbracket_{x_1, \dots, x_k}, \\ f_P(x_1, \dots, x_k, y + 1) &= \llbracket t' \rrbracket_{x_1, \dots, x_k, y, f_P(x_1, \dots, x_k, y)}. \end{aligned}$$

Dabei ist  $\llbracket t \rrbracket_{y_1, \dots, y_l}$  für einen  $l$ -Term  $t$  rekursiv definiert durch:

1.  $\llbracket 0 \rrbracket_{y_1, \dots, y_l} = 0$  und  $\llbracket e_i \rrbracket_{y_1, \dots, y_l} = y_i$ .
2.  $\llbracket t + 1 \rrbracket_{y_1, \dots, y_l} = \llbracket t \rrbracket_{y_1, \dots, y_l} + 1$ .
3.  $\llbracket P(t_1, \dots, t_m) \rrbracket_{y_1, \dots, y_l} = f_P(\llbracket t_1 \rrbracket_{y_1, \dots, y_l}, \dots, \llbracket t_m \rrbracket_{y_1, \dots, y_l})$ .

### PRIMREK-Berechenbarkeit

**Definition.** Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt **PRIMREK-berechenbar**, wenn es ein PRIMREK-Programm gibt, das  $f$  berechnet.

#### Beispiele:

1.  $\mathbf{fn}(e_1, \dots, e_k) : 0$  berechnet die ( **$k$ -stellige Nullfunktion**)

$$C^k : \mathbb{N}^k \rightarrow \mathbb{N}, \quad C^k(x_1, \dots, x_k) = 0.$$

2.  $\mathbf{fn}(e_1, \dots, e_k) : e_i$  berechnet die ( **$k$ -stellige Projektion**)

$$U_i^k : \mathbb{N}^k \rightarrow \mathbb{N}, \quad U_i^k(x_1, \dots, x_k) = x_i.$$

3.  $\mathbf{fn}(e_1) : e_1 + 1$  berechnet die **Nachfolgerfunktion**

$$S : \mathbb{N} \rightarrow \mathbb{N}, \quad S(x) = x + 1.$$

4.  $\mathbf{fn}(e_1, e_2) : e_1 \mid e_3 + 1$  berechnet die Addition  $x_1 + x_2$  auf  $\mathbb{N}$ .

5.  $\mathbf{fn}(e_1) : 0 + 1 \mid 0$  berechnet das Prädikat  $null : \mathbb{N} \rightarrow \{0, 1\}$  mit  $null(x) = 1 \iff x = 0$ .

6.  $\mathbf{fn}(e_1, e_2) :$

$\mathbf{fn}(e_1, e_2, e_3) : e_1 \mid e_2$  ( $\mathbf{fn}(e_1, e_2) : e_1 \mid e_3 + 1$  ( $e_1, e_2$ ),  $0 + 1 + 1$ ,  $\mathbf{fn}(e_1) : 1 \mid 0$  ( $e_1$ ))  
berechnet die Funktion  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit

$$f(x_1, x_2) = \begin{cases} 2 & \text{falls } x_1 = 0 \\ x_1 + x_2, & \text{sonst.} \end{cases}$$

### Schreibweisen

- $x_1, x_2, \dots, x, y, \dots$  für  $e_1, e_2, \dots$ , dabei  $y$  jeweils für  $e_{k+1}$  in  $\mathbf{fn}(e_1, \dots, e_{k+1}) : t \mid t'$ .
- $rec$  für  $e_{k+2}$  im  $(k+2)$ -Term  $t'$  von  $\mathbf{fn}(e_1, \dots, e_{k+1}) : t \mid t'$ .

- $n$  für  $0 + \underbrace{1 + 1 + \dots + 1}_n$ .
- $f_P$  für  $P$  in  $l$ -Termen  $P(t_1, \dots, t_m)$ , gegebenenfalls auch in Infix- oder sonstigen üblichen Schreibweisen.
- **if**  $f_P(t'_1, \dots, t'_m)$  **then**  $t_1$  **else**  $t_2$  **endif** für  $\mathbf{fn}(e_1, e_2, e_3) : e_1 \mid e_2(t_2, t_1, P(t'_1, \dots, t'_m))$ , falls  $f_P$  ein Prädikat ist.

Z.B. Schreibweise der Beispiele 4. und 6. von oben:

$\mathbf{fn}(x, y) : x \mid \mathit{rec} + 1,$   
 $\mathbf{fn}(x, y) : \mathbf{if} \mathit{null}(x) \mathbf{then} 2 \mathbf{else} x + y \mathbf{endif}.$

### Primitive Rekursion

- Die PRIMREK-berechenbaren Funktionen heißen auch *primitiv-rekursiv*. Jede derartige Funktion ist total.
- Die primitiv-rekursiven Funktionen lassen sich auch ohne Bezugnahme auf die Programmiersprache PRIMREK „rein mathematisch“ induktiv wie folgt definieren (*algebraisch* darstellen):

1. Die Funktionen  $C^k, U_i^k$  ( $1 \leq i \leq k$ ),  $S$  sind primitiv-rekursiv.

2. Sind  $g : \mathbb{N}^n \rightarrow \mathbb{N}, h_1, \dots, h_n : \mathbb{N}^k \rightarrow \mathbb{N}$  primitiv-rekursive Funktionen, so ist die (durch Einsetzung der  $h_i$  in  $g$  gewonnene) Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  mit

$$f(x_1, \dots, x_k) = g(h_1(x_1, \dots, x_k), \dots, h_n(x_1, \dots, x_k))$$

primitiv-rekursiv.

3. Sind  $g : \mathbb{N}^k \rightarrow \mathbb{N}$  und  $h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$  primitiv-rekursive Funktionen, so ist die Funktion  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  mit

$$\begin{aligned} f(x_1, \dots, x_k, 0) &= g(x_1, \dots, x_k), \\ f(x_1, \dots, x_k, y + 1) &= h(x_1, \dots, x_k, y, f(x_1, \dots, x_k, y)) \end{aligned}$$

primitiv-rekursiv.

- Das Rekursionschema in Punkt 3 heißt (ebenso wie das entsprechende Schema in PRIMREK) *primitive Rekursion*.
- Beispiel: Die Addition wird algebraisch dargestellt durch

$$\begin{aligned} \mathit{add}(x, 0) &= U_1^1(x), \\ \mathit{add}(x, y + 1) &= S(U_3^3(x, y, \mathit{add}(x, y))). \end{aligned}$$

### Einige primitiv-rekursive Funktionen und Prädikate

- „Grund-“ Funktionen und Prädikate (mit jeweiligen PRIMREK-Programmen):

$$\begin{aligned}
x + y &: \mathbf{fn}(x, y) : x \mid \mathit{rec} + 1 \\
y \div 1 &: \mathbf{fn}(y) : 0 \mid y \\
x \div y &: \mathbf{fn}(x, y) : x \mid \mathit{rec} \div 1 \\
x * y &: \mathbf{fn}(x, y) : 0 \mid \mathit{rec} + x \\
x^y &: \mathbf{fn}(x, y) : 1 \mid \mathit{rec} * x \\
y! &: \mathbf{fn}(y) : 1 \mid \mathit{rec} * (y + 1) \\
\mathit{null}(y) &: \mathbf{fn}(y) : 1 \mid 0 \\
x \leq y &: \mathbf{fn}(x, y) : \mathit{null}(x \div y) \\
x < y &: \mathbf{fn}(x, y) : 1 \div (y \leq x) \\
x = y &: \mathbf{fn}(x, y) : (x \leq y) * (y \leq x) \\
x \neq y &: \mathbf{fn}(x, y) : 1 \div (x = y)
\end{aligned}$$

- Funktionen und Prädikate zur Teilbarkeit von Zahlen:

$$\begin{aligned}
y \mathit{div} x &: \mathbf{fn}(x, y) : 0 \mid \\
&\quad \mathbf{if} \ y < (x * (\mathit{rec} + 1)) \ \mathbf{then} \ \mathit{rec} \ \mathbf{else} \ \mathit{rec} + 1 \ \mathbf{endif} \\
x \mathit{mod} y &: \mathbf{fn}(x, y) : x \div (y * (x \mathit{div} y)) \\
x \mid y &: \mathbf{fn}(x, y) : (y \mathit{mod} x) = 0 \\
\mathit{teiler}(x, y) &: \mathbf{fn}(x, y) : 0 \mid \mathit{rec} + ((y + 1) \mid x) \\
\mathit{prim}(y) &: \mathbf{fn}(y) : \mathit{teiler}(y, y) = 2 \\
\mathit{klpz}(x, y) &: \mathbf{fn}(x, y) : 1 \mid \\
&\quad \mathbf{if} \ \mathit{rec} \leq y \ \mathbf{then} \ \mathit{rec} \\
&\quad \quad \mathbf{else} \ (y + 2) \div \mathit{prim}(y + 1) \ \mathbf{endif} \\
\mathit{pz}(y) &: \mathbf{fn}(y) : 0 \mid \mathit{klpz}(\mathit{rec}, \mathit{rec}! + 1)
\end{aligned}$$

### Kodierungsfunktionen

- Für jedes  $k \in \mathbb{N}$  können  $k$ -Tupel  $(x_1, \dots, x_k) \in \mathbb{N}^k$  durch die Funktion  $\gamma_k : \mathbb{N}^k \rightarrow \mathbb{N}$  mit

$$\begin{aligned}
\gamma_0() &= 0, \\
\gamma_k(x_1, \dots, x_k) &= 2^{x_1} * 3^{x_2} * \dots * \mathit{pz}(k)^{x_k} \quad \text{für } k > 0
\end{aligned}$$

als natürliche Zahlen kodiert werden. Jedes  $\gamma_k$  ist primitiv-rekursiv.

- Die Funktion  $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit

$$\pi(x, y) = \begin{cases} \text{Exponent von } \mathit{pz}(x) \text{ in der Primfaktorzerlegung von } y, & \text{falls } x, y > 0 \\ 0 & \text{sonst.} \end{cases}$$

ist primitiv-rekursiv, und es gilt:

$$\pi(i, \gamma_k(x_1, \dots, x_k)) = \begin{cases} x_i, & \text{falls } 1 \leq i \leq k \\ 0 & \text{sonst.} \end{cases}$$

- Schreibweisen:  $\gamma$  für  $\gamma_k$  (falls  $k$  aus dem Kontext ersichtlich).  
 $\pi_i(x)$  für  $\pi(i, x)$ .

### Weitere Rekursionsformen

In funktionaler Programmierung (alternativ: mathematischen Funktionsdefinitionen): weitere Rekursionsformen erlaubt/gebräuchlich, z.B.:

- Statt rekursiver Rückgriff „von  $y+1$  auf  $y$ “: Rückgriff auf  $y$  und/oder (auch mehrere) kleinere Werte.
- **Simultane Rekursion**: Definition von  $f_i$  für  $i = 1, \dots, n$  durch:

$$\begin{aligned} f_i(x_1, \dots, x_k, 0) &= g_i(x_1, \dots, x_k), \\ f_i(x_1, \dots, x_k, y+1) &= h_i(x_1, \dots, x_k, y, f_1(x_1, \dots, x_k, y), \dots, f_n(x_1, \dots, x_k, y)). \end{aligned}$$

Diese Rekursionsarten können durch primitive Rekursion ausgedrückt werden, führen also nicht aus der Klasse der primitiv-rekursiven Funktionen heraus.

### Die Ackermann-Funktion

Die **Ackermann-Funktion**  $ack : \mathbb{N}^2 \rightarrow \mathbb{N}$  ist definiert durch:

$$\begin{aligned} ack(0, y) &= y + 1, \\ ack(x + 1, 0) &= ack(x, 1), \\ ack(x + 1, y + 1) &= ack(x, ack(x + 1, y)). \end{aligned}$$

( $ack$  ist eine totale Funktion.)

Es gilt (siehe Abschnitt 2.3):  $ack$  ist nicht primitiv-rekursiv.

### $\mu$ -Rekursion

- $\mu$ REK: Erweiterung von PRIMREK um Programme mit der zusätzlichen Rekursionsform (nur in algebraischer Notation)

$$f(x_1, \dots, x_k, y) = \begin{cases} y, & \text{falls } g(x_1, \dots, x_k, y) = 0 \\ f(x_1, \dots, x_k, y + 1) & \text{sonst} \end{cases}$$

( $\mu$ -Rekursion). Für die so definierte Funktion  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  gilt:

$$f(x_1, \dots, x_k, y) = \begin{cases} \min(M_y^g), & \text{falls } M_y^g \neq \emptyset \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

wobei  $M_m^g$  für eine Funktion  $g$  und  $m \in \mathbb{N}$  definiert ist durch:

$$M_m^g = \{i \in \mathbb{N} \mid i \geq m, g(x_1, \dots, x_k, j) \text{ ist definiert für alle } j \text{ mit } m \leq j \leq i \text{ und } g(x_1, \dots, x_k, i) = 0\}.$$

- Eine Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt  $\mu$ -**rekursiv**, wenn es ein  $\mu$ REK-Programm gibt, das  $f$  berechnet.

- Für eine Funktion  $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  sei die Funktion  $\mu(g) : \mathbb{N}^k \rightarrow \mathbb{N}$  definiert durch:

$$\mu(g)(x_1, \dots, x_k) = \begin{cases} \min(M_0^g), & \text{falls } M_0^g \neq \emptyset \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Ist  $g$   $\mu$ -rekursiv, so ist auch  $\mu(g)$   $\mu$ -rekursiv.

- Mit  $h(x_1, \dots, x_k, z) = g(x_1, \dots, x_k, m + z)$  ist  $\min(M_m^g) = \min(M_0^h) + m$  (falls definiert). Daher lassen sich die  $\mu$ -rekursiven Funktionen algebraisch wie folgt darstellen:
  1. Die Funktionen  $C^k, S, U_i^k$  ( $1 \leq i \leq k$ ),  $k \in \mathbb{N}$ , sind  $\mu$ -rekursiv.
  2. Sind  $g, h_1, \dots, h_n, h$   $\mu$ -rekursive Funktionen, so sind die durch Einsetzung der  $h_i$  in  $g$  sowie die durch primitive Rekursion aus  $g$  und  $h$  gewonnenen Funktionen  $\mu$ -rekursiv.
  3. Ist  $g$  eine  $\mu$ -rekursive Funktion, so ist  $\mu(g)$  eine  $\mu$ -rekursive Funktion.
- Jede primitiv-rekursive Funktion ist auch  $\mu$ -rekursiv.

### Die $\mu$ -Rekursivität der Ackermann-Funktion

**Satz 2.2.1** Die Ackermann-Funktion ist  $\mu$ -rekursiv.

## 2.3 WHILE- und LOOP-Programme

### Die Programmiersprachen WHILE und LOOP

WHILE: Eine höhere imperative Programmiersprache zur Formulierung von Algorithmen, die Zahlfunktionen berechnen.

LOOP: Teilsprache von WHILE.

### Syntax von WHILE und LOOP

Sei  $\mathcal{V} = \{a_0, a_1, a_2, \dots\}$  eine (fest vorgegebene unendliche) Menge von **Variablen**. Ein **WHILE-Programm** ist eine nicht-leere endliche Folge von **Anweisungen** (jeweils durch Strichpunkte „;“ getrennt). Jede Anweisung ist entweder eine **Zuweisung** der Form

$$a_i := 0 \quad \text{oder} \quad a_i := a_j \quad \text{oder} \quad a_i := a_j + 1$$

( $a_i$  und  $a_j$  nicht notwendig verschieden) oder eine **Wiederholungsanweisung** der Form

**loop**  $a_i$  **do**  $P$  **enddo**      (LOOP-Schleife)

oder

**while**  $a_i \neq 0$  **do**  $P$  **enddo**      (WHILE-Schleife)

mit einem WHILE-Programm  $P$ .

Ein LOOP-*Programm* ist ein derartiges Programm, das keine WHILE-Schleifen enthält.

### Semantik von WHILE und LOOP

- Ein (*Variablen-*) *Zustand* ist eine Abbildung  $\mathcal{V} \rightarrow \mathbb{N}$ , die jeder Variablen  $a_i \in \mathcal{V}$  eine natürliche Zahl als *Wert (-Belegung)* zuordnet.
- Ein *Ablauf* eines WHILE-Programms  $P_1; P_2; \dots; P_l$  startet in einem Zustand und verändert diesen (d.h. die Werte von Variablen) schrittweise durch Nacheinander-Ausführung der Anweisungen  $P_1, P_2, \dots, P_l$ . Nach Ausführung von  $P_l$  ist der Ablauf *beendet*.

- Die Wirkung der möglichen Anweisungen ist wie folgt:

$a_i := 0$ :	$a_i$ wird mit dem Wert 0 belegt.
$a_i := a_j$ :	$a_i$ wird mit dem (aktuellen) Wert von $a_j$ belegt.
$a_i := a_j + 1$ :	$a_i$ wird mit dem um 1 erhöhten (aktuellen) Wert von $a_j$ belegt.
<b>loop</b> $a_i$ <b>do</b> $P$ <b>enddo</b> :	$P$ wird so oft nacheinander ausgeführt, wie der Wert der Variablen $a_i$ in dem Zustand angibt, in dem die Ausführung der Schleife begonnen wird.
<b>while</b> $a_i \neq 0$ <b>do</b> $P$ <b>enddo</b> :	$P$ wird so lange nacheinander ausgeführt, wie der Wert der Variablen $a_i$ nicht 0 ist. Wird der Wert von $a_i$ nie 0, so wird die Schleife und damit der gesamte Ablauf nicht beendet. (Das Programm <i>terminiert</i> für den betreffenden Startzustand <i>nicht</i> .)

Die Werte der übrigen Variablen bleiben bei den Zuweisungen jeweils unverändert.

### Berechnung von Funktionen mit WHILE und LOOP

**Definition.** Sei  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  eine Funktion. Ein WHILE-Programm  $P$  *berechnet*  $f$ , wenn gilt: Wird der Ablauf von  $P$  in einem Zustand gestartet, in dem die Variablen  $a_1, \dots, a_k$  mit  $x_1, \dots, x_k \in \mathbb{N}$  und alle übrigen Variablen mit 0 belegt sind, so terminiert  $P$  genau dann, wenn  $f(x_1, \dots, x_k)$  definiert ist, und falls  $P$  terminiert, hat die Variable  $a_0$  in dem nach Beendigung des Ablaufs erreichten Zustand den Wert  $f(x_1, \dots, x_k)$ .

$f$  heißt *WHILE-berechenbar* bzw. *LOOP-berechenbar*, wenn es ein WHILE-Programm (LOOP-Programm) gibt, das  $f$  berechnet.

(Offenbar ist jede LOOP-berechenbare Funktion total und auch WHILE-berechenbar.)

### Bemerkungen

- Die Programmiersprachen WHILE und LOOP sind sehr einfach, mächtigere Konstruktionen sind programmierbar, z.B. ( $n \in \mathbb{N}$ ):

- $a_i := n$ :  $a_i := 0; \underbrace{a_i := a_i + 1; \dots; a_i := a_i + 1}_{n\text{-mal}}$
- $a_i := a_j + a_l$ :  $a_i := a_j;$   
**loop**  $a_l$  **do**  $a_i := a_i + 1$  **enddo**
- **if**  $a_i = 0$  **then**  $P_1$  **else**  $P_2$  **endif**:  $a_j := 1; \mathbf{loop} \ a_i \ \mathbf{do} \ a_j := 0 \ \mathbf{enddo};$   
 $a_l := 0; \mathbf{loop} \ a_i \ \mathbf{do} \ a_l := 1 \ \mathbf{enddo};$   
**loop**  $a_j$  **do**  $P_1$  **enddo**;  
**loop**  $a_l$  **do**  $P_2$  **enddo**

- Die Schleifenform **loop** . . . **do** . . . **enddo** entspricht dem Konzept von Laufanweisungen (**for**-Schleifen), die Form **while**  $a_i \neq 0$  **do**  $P$  **enddo** den bedingten Wiederholungsanweisungen in üblichen Programmiersprachen.

### Beispiele

- Das Programm

$$a_0 := a_1; \mathbf{loop} \ a_2 \ \mathbf{do} \ a_0 := a_0 + 1 \ \mathbf{enddo}$$

berechnet die Addition  $x_1 + x_2$  auf  $\mathbb{N}$ .

- Das Programm

$$\mathbf{if} \ a_1 = 0 \ \mathbf{then} \ a_0 := 0$$

$$\mathbf{else} \ a_2 := 0; \mathbf{loop} \ a_1 \ \mathbf{do} \ a_0 := a_2; \ a_2 := a_2 + 1 \ \mathbf{enddo}$$

$$\mathbf{endif}$$

berechnet (mit Verwendung der oben allgemein angegebenen **if**-Konstruktion) die Zahlfunktion  $x \div 1$ .

- Das Programm

$$a_0 := a_1; \mathbf{while} \ a_2 \neq 0 \ \mathbf{do} \ a_0 := a_0 + 1; \ a_2 := a_2 \div 1 \ \mathbf{enddo}$$

(mit Verwendung der Funktion  $x \div 1$  in einer der Zuweisungen) stellt eine weitere Berechnung der Addition dar.

### Primitive Rekursion und LOOP-Berechenbarkeit

**Satz 2.3.1** Jede primitiv-rekursive Funktion ist LOOP-berechenbar.

**Bemerkung:** Es gilt auch die Umkehrung von Satz 2.3.1: Jede LOOP-berechenbare Funktion ist primitiv-rekursiv.

### Zur Ackermann-Funktion

**Satz 2.3.2** Zu jeder LOOP-berechenbaren Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  gibt es  $l \in \mathbb{N}$  mit

$$f(x_1, \dots, x_k) < ack(l, \max(x_1, \dots, x_k)) \quad (\text{für alle } x_1, \dots, x_k \in \mathbb{N}).$$

**Satz 2.3.3** Die Ackermann-Funktion ist nicht LOOP-berechenbar (also auch nicht primitiv-rekursiv).

### Zur Mächtigkeit der WHILE-Berechenbarkeit

**Satz 2.3.4** Jede  $\mu$ -rekursive Funktion ist WHILE-berechenbar.

**Satz 2.3.5** Jede WHILE-berechenbare Funktion ist Turing-berechenbar.

## 2.4 Einige Hauptergebnisse der Berechenbarkeitstheorie

### Kodierung von DTM-Rechnungen (Gödelisierung)

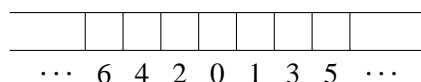
#### 1. Festlegungen

Im Folgenden werden nur DTMen  $M = (Q, \{|\, \circ\}, \Gamma, \delta, q_0)$  betrachtet. (Dazu gehören insbesondere alle DTMen, die Zahlfunktionen berechnen.) Dabei sei

$$\begin{aligned} Q &\subseteq \{q_0, q_1, q_2, \dots\}, \\ \Gamma &\subseteq \{a_0, a_1, a_2, \dots\} \end{aligned}$$

mit zwei fest vorgegebenen Mengen  $\{q_0, q_1, q_2, \dots\}$  und  $\{a_0, a_1, a_2, \dots\}$ . Die Elemente von  $Q$  (außer  $q_0$ ) und  $\Gamma$  (außer  $\square, |, \circ$ ) seien genau die in  $\delta$  vorkommenden  $q_i$  bzw.  $a_j$ .

Die Felder des Bandes von  $M$  seien nummeriert wie folgt:



Zu Beginn jeder Rechnung stehe der Schreib-/Lesekopf auf dem Feld mit der Nummer 1.

## 2. Gödelnummern von Turingmaschinen

Sei  $M$  eine DTM. Jedem Übergang

$$(q_i, a_j) \mapsto (q_{i'}, a_{j'}, X_s) \quad (\text{wobei } X_0 = L, X_1 = R, X_2 = N)$$

vermöge der Überföhrungsfunktion  $\delta$  von  $M$  wird zugeordnet die Zahl

$$\gamma(i, j, i', j', s).$$

$d_1, \dots, d_r$  seien alle derartigen Zahlen (in beliebig gewählter Reihenfolge). Dann sei

$$c(M) = \gamma(d_1, \dots, d_r) \quad (\text{Gödelnummer von } M).$$

## 3. Gödelnummern von Konfigurationen

Sei  $K = (\alpha, q_i, \beta)$  Konfiguration mit

$n_{AF}$  = Nummer des Arbeitsfeldes,

$n_\alpha$  = Nummer des Feldes, auf dem das erste Zeichen von  $\alpha$  steht,

$n_\beta$  = Nummer des Feldes, auf dem das letzte Zeichen von  $\beta$  steht,

$n_{max} = \max(n_\alpha, n_\beta)$ ,

$a_{j_l}$  = Zeichen, das auf Feld  $l$  steht ( $0 \leq l \leq n_{max}$ ).

Dann sei

$$c(K) = \gamma(i, n_{AF}, j_0, \dots, j_{n_{max}}) \quad (\text{Gödelnummer von } K).$$

## 4. Gödelnummern von Rechnungen

Sei  $\mathcal{R}$  eine Rechnung  $K_0 \vdash K_1 \vdash \dots \vdash K_l$  einer DTM. Dann sei

$$c(\mathcal{R}) = \gamma(c(K_0), \dots, c(K_l)).$$

**Universelle Funktionen**

**Lemma 2.4.1** Es gibt primitiv-rekursive Prädikate  $T_k : \mathbb{N}^{k+2} \rightarrow \{0, 1\}$  (für alle  $k \in \mathbb{N}$ ) sowie eine primitiv-rekursive Funktion  $erg : \mathbb{N} \rightarrow \mathbb{N}$  mit folgenden Eigenschaften:

$T_k(x, x_1, \dots, x_k, y) = 0 \iff x$  ist Gödelnummer einer DTM  $M$ , und  $y$  ist Gödelnummer einer mit der Konfiguration  $(\varepsilon, q_0, |^{x_1} \circ |^{x_2} \circ \dots \circ |^{x_k})$  beginnenden und mit einer finalen Konfiguration endenden Rechnung von  $M$ .

$erg(x) = y$ , falls  $x$  Gödelnummer einer mit einer Konfiguration  $(\varepsilon, q, |^y)$  endenden Rechnung ist (sonst: beliebig).

**Definition.** Sei  $k \in \mathbb{N}$ . Die Funktion  $\Phi_k : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  mit

$$\Phi_k(x, x_1, \dots, x_k) = erg(\mu(T_k)(x, x_1, \dots, x_k))$$

heißt **( $k$ -te) universelle Funktion**.

**Satz 2.4.2** Die Funktionen  $\Phi_k$  ( $k \in \mathbb{N}$ ) sind  $\mu$ -rekursiv.

**Satz 2.4.3** Sei  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  eine Turing-berechenbare Funktion.  $f$  werde berechnet durch eine DTM mit der Gödelnummer  $t$ . Dann gilt:

$$f(x_1, \dots, x_k) = \Phi_k(t, x_1, \dots, x_k).$$

$t$  heißt **Index** von  $f$ .

**Satz 2.4.4** Für jede Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  gilt:

$$f \text{ } \mu\text{-rekursiv} \iff f \text{ WHILE-berechenbar} \iff f \text{ Turing-berechenbar.}$$

### Bemerkungen

1. Satz 2.4.4: Alle („umfassenden“) Ansätze zur Formalisierung des Berechenbarkeitsbegriffs sind gleichwertig (einschließlich RAM-Berechenbarkeit sowie weiterer hier nicht behandelter Ansätze). Dies erhärtet die Churchsche These.
2. Sprechweise (in Abschnitt 2.1 zum Teil schon verwendet):  $f$  **berechenbar** (statt  $\mu$ -rekursiv, Turing-/WHILE-/. . .-berechenbar).
3. Aus Satz 2.4.3 folgt: Durchläuft  $t$  alle natürlichen Zahlen, so durchläuft  $\Phi_k(t, \dots)$  alle berechenbaren Funktionen  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ .
4. Aus den Sätzen 2.4.2 und 2.4.4 folgt: Es gibt WHILE- bzw.  $\mu$ REK-Programme und deterministische Turing-Maschinen zur Berechnung der  $\Phi_k$ . Diese heißen **universelle Programme** bzw. **universelle Turingmaschinen** und berechnen bei Eingabe von  $x, x_1, \dots, x_k$  das Ergebnis der Berechnung der deterministischen Turing-Maschine mit Gödelnummer  $x$  für die Eingabe  $x_1, \dots, x_k$ .

### Weitere grundlegende Ergebnisse

#### Satz 2.4.5 (Kleenesches Normalformtheorem)

Jede berechenbare Zahlfunktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  lässt sich darstellen als

$$f(x_1, \dots, x_k) = g(\mu(h)(x_1, \dots, x_k))$$

mit primitiv-rekursiven Zahlfunktionen  $g : \mathbb{N} \rightarrow \mathbb{N}$  und  $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ .

**Bemerkung:** Aus Satz 2.4.5 folgt: Jede berechenbare Zahlfunktion kann durch ein WHILE-Programm berechnet werden, das nur eine **while**-Schleife enthält.

#### Satz 2.4.6 (S-m-n-Theorem)

Zu  $m, n \in \mathbb{N}$  gibt es eine primitiv-rekursive Funktion  $S_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$  mit

$$\Phi_{m+n}(x, x_1, \dots, x_m, y_1, \dots, y_n) = \Phi_n(S_{m,n}(x, x_1, \dots, x_m), y_1, \dots, y_n).$$

**Satz 2.4.7 (Rekursionstheorem)**

Sei  $f : \mathbb{N} \rightarrow \mathbb{N}$  eine totale berechenbare Funktion,  $k \in \mathbb{N}$ . Dann gibt es  $m \in \mathbb{N}$ , so dass für alle  $x_1, \dots, x_k \in \mathbb{N}$  gilt:

$$\Phi_k(f(m), x_1, \dots, x_k) = \Phi_k(m, x_1, \dots, x_k).$$

**Eine nicht berechenbare Funktion**

**Satz 2.4.8** Die Funktion  $h : \mathbb{N} \rightarrow \{0, 1\}$  mit

$$h(x) = 1 \iff \Phi_1(x, x) \text{ ist definiert}$$

ist nicht berechenbar.

# Kapitel 3

## Analyse und Erzeugung von Sprachen

### 3.1 Analyse von Sprachen

#### (Semi-) Entscheidbarkeit

- Zur Erinnerung (Abschnitt 1.3):  
Eine NTM  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  **akzeptiert (erkennt)** das Wort  $w \in \Sigma^*$ , falls es eine finale Konfiguration  $(\alpha, q_+, \beta)$  gibt mit  $(\varepsilon, q_0, w) \vdash_M^* (\alpha, q_+, \beta)$ .  
 $M$  **akzeptiert** eine Sprache  $L \subseteq \Sigma^*$  (d.h.  $L$  ist die von  $M$  akzeptierte Sprache  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$ ), wenn  $M$  genau alle  $w \in L$  akzeptiert.  
 $M$  **entscheidet** eine Sprache  $L \subseteq \Sigma^*$ , wenn  $M$  für alle  $w \in \Sigma^*$  terminiert und genau alle  $w \in L$  akzeptiert.
- Sei  $\Sigma$  ein Alphabet. Eine Sprache  $L \subseteq \Sigma^*$  heißt **semi-entscheidbar**, wenn es eine NTM gibt, die  $L$  akzeptiert.  $L$  heißt **entscheidbar**, wenn es eine NTM gibt, die  $L$  entscheidet. Andernfalls heißt  $L$  **unentscheidbar**.

**Satz 3.1.1** Eine Sprache  $L$  ist genau dann entscheidbar, wenn  $L$  und  $\bar{L}$  semi-entscheidbar sind.

#### Satz 3.1.2

- Zu jeder NTM  $M$  gibt es eine DTM  $M'$  mit  $\mathcal{L}(M') = \mathcal{L}(M)$ .
- Eine Sprache ist genau dann semi-entscheidbar, wenn sie von einer DTM akzeptiert wird.
- Eine Sprache ist genau dann entscheidbar, wenn sie von einer DTM entschieden wird.

#### Charakteristische Funktionen von Sprachen

- Sei  $\Sigma$  ein Alphabet und  $a_+ \in \Sigma$  ein ausgezeichnetes Zeichen. Ist  $L \subseteq \Sigma^*$  eine Sprache, so ist die **charakteristische Funktion**

$$\chi_L : \Sigma^* \rightarrow \Sigma^*$$

von  $L$  definiert durch

$$\chi_L(w) = \begin{cases} a_+, & \text{falls } w \in L \\ \varepsilon & \text{sonst.} \end{cases}$$

- Außerdem: Die Funktion

$$\chi'_L : \Sigma^* \rightarrow \Sigma^*$$

ist definiert durch

$$\chi'_L(w) = \begin{cases} a_+, & \text{falls } w \in L \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

**Satz 3.1.3** Sei  $L$  eine Sprache.

- $L$  ist genau dann entscheidbar, wenn  $\chi_L$  berechenbar ist.
- $L$  ist genau dann semi-entscheidbar, wenn  $\chi'_L$  berechenbar ist.

## 3.2 Entscheidungsprobleme

### Interpretierte Entscheidbarkeitsaussagen

- Sei  $R \subseteq \mathbb{N}^k$ . Die Sprache

$$L_R = \{ |^{n_1} \circ |^{n_2} \circ \dots \circ |^{n_k} \in \{ |, \circ \}^* \mid (n_1, n_2, \dots, n_k) \in R \}$$

kodiert die Zahlmenge  $R$ . Ist  $L_R$  (semi-/un)entscheidbar, so sagen wir auch: „ $R$  ist (semi-/un)entscheidbar“.

- Allgemein: Sei  $L$  eine Sprache, die eine konkrete Datenmenge  $\mathcal{D}$  kodiert. Ist  $L$  (semi-/un)entscheidbar, so sagen wir auch: „ $\mathcal{D}$  ist (semi-/un)entscheidbar“.

**Lemma 3.2.1** Sei  $R \subseteq \mathbb{N}^k$  eine Zahlmenge.

- $R$  ist genau dann entscheidbar, wenn die Zahlfunktion  $ch_R : \mathbb{N}^k \rightarrow \{0, 1\}$  mit

$$ch_R(x_1, \dots, x_k) = \begin{cases} 1, & \text{falls } (x_1, \dots, x_k) \in R \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist. ( $ch_R$  heißt **charakteristische Funktion** von  $R$ .)

- $R$  ist genau dann semi-entscheidbar, wenn die Zahlfunktion  $ch'_R : \mathbb{N}^k \rightarrow \{0, 1\}$  mit

$$ch'_R(x_1, \dots, x_k) = \begin{cases} 1, & \text{falls } (x_1, \dots, x_k) \in R \\ \text{undefiniert} & \text{sonst} \end{cases}$$

berechenbar ist.

## Entscheidungsprobleme

- Ein (**Entscheidungs-**) **Problem** ist eine Aufgabe der Art
  - Entscheide für (beliebig gegebene) „Objekte“  $x_1, \dots, x_n$  (aus „Objektmengen“  $\mathcal{O}_1, \dots, \mathcal{O}_n$ ), ob diese die „Eigenschaft“  $\mathcal{E}$  erfüllen.
- Ein solches Problem wird formal repräsentiert durch die Menge

$$\{(x_1, \dots, x_n) \in \mathcal{O}_1 \times \dots \times \mathcal{O}_n \mid x_1, \dots, x_n \text{ erfüllen } \mathcal{E}\}.$$

Ist diese Menge (d.h. ihre Kodierung als Sprache) (semi-/un)entscheidbar, so sagen wir, dass das betreffende Entscheidungsproblem (semi-/un)entscheidbar (statt (un)entscheidbar oft auch: **(un)lösbar**) ist, oder auch, dass die Eigenschaft  $\mathcal{E}$  für die Objekte  $x_1, \dots, x_n$  (semi-/un)entscheidbar ist.

## Das Halteproblem (für deterministische Turing-Maschinen)

- Die Menge

$$H = \{(n, m) \in \mathbb{N}^2 \mid \Phi_1(n, m) \text{ ist definiert}\}$$

(mit der in Abschnitt 2.4 definierten Funktion  $\Phi_1$ ) repräsentiert das **Halteproblem** (für DTM), informell:

- Entscheide, ob die DTM mit der Gödelnummer  $n$  für die Eingabe  $m$  terminiert.
- Spezieller: **Selbstanwendungsproblem (spezielles Halteproblem)** (für DTM):
  - Entscheide, ob die DTM mit der Gödelnummer  $n$  für die Eingabe  $n$  terminiert.

Als Menge:

$$SANW = \{n \in \mathbb{N} \mid \Phi_1(n, n) \text{ ist definiert}\}.$$

**Satz 3.2.2** Das Selbstanwendungsproblem und das Halteproblem sind unentscheidbar.

**Satz 3.2.3** Das Selbstanwendungsproblem und das Halteproblem sind semi-entscheidbar.

## Reduzierbarkeit

**Definition.** Seien  $\Sigma$  ein Alphabet,  $L_1, L_2 \subseteq \Sigma^*$  zwei Sprachen.  $L_1$  heißt **reduzierbar auf**  $L_2$ , wenn es eine totale berechenbare Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  gibt, so dass für alle  $w \in \Sigma^*$  gilt:

$$w \in L_1 \iff f(w) \in L_2.$$

(Schreibweise:  $L_1 \leq L_2$  oder auch  $f : L_1 \leq L_2$ .)

**Satz 3.2.4** Seien  $\Sigma$  ein Alphabet,  $L_1, L_2 \subseteq \Sigma^*$ ,  $L_1 \leq L_2$ . Ist  $L_2$  entscheidbar, so ist  $L_1$  entscheidbar.

### Der Satz von Rice

#### Satz 3.2.5 (Satz von Rice)

Sei  $\mathcal{F}$  die Menge aller berechenbaren Funktionen  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $\mathcal{G} \subseteq \mathcal{F}$  mit  $\mathcal{G} \neq \emptyset$  und  $\mathcal{G} \neq \mathcal{F}$ . Dann ist die Menge

$$R_{\mathcal{G}} = \{n \in \mathbb{N} \mid f_n \in \mathcal{G}\}$$

(wobei  $f_n(x) = \Phi_1(n, x)$ ) unentscheidbar.

### Bemerkungen

- Der Satz von Rice kann auch auf beliebige von DTMen berechnete Funktionen  $\Sigma^* \rightarrow \Sigma^*$  und auch auf reale (höhersprachliche) Programme übertragen werden. Bedeutung für letztere: Nichttriviale Programmeigenschaften (Terminierung, Korrektheit, Äquivalenz usw.) sind nicht entscheidbar.
- Beispiel für ein Problem, das sogar nicht semi-entscheidbar ist:

$$AEQ = \{(n, m) \in \mathbb{N}^2 \mid \Phi_1(n, x) = \Phi_1(m, x) \text{ für alle } x \in \mathbb{N}\}.$$

(„Entscheide, ob die DTMen mit den Gödelnummern  $n$  und  $m$  die gleiche Funktion berechnen“, **Äquivalenzproblem für DTM**).

- Selbstanwendungsproblem: Häufiger „Bezugspunkt“ für Nachweise der Unentscheidbarkeit von Problemen mit der Methode der Reduktion (Satz 3.2.4).

Ein anderes in solchen Beweisen häufig verwendetes unentscheidbares Problem:

#### **Postisches Korrespondenzproblem (PCP):**

- Entscheide für ein Alphabet  $\Sigma$  und eine endliche Folge  $(x_1, y_1), \dots, (x_k, y_k)$  mit  $x_i, y_i \in \Sigma^+$ : Gibt es eine Folge  $i_1, \dots, i_n$  ( $n \geq 1$ ) von Indizes  $i_j \in \{1, \dots, k\}$ , so dass

$$x_{i_1}x_{i_2}\dots x_{i_n} = y_{i_1}y_{i_2}\dots y_{i_n}.$$

(Die Folge  $i_1, \dots, i_n$  heißt Lösung für das gegebene PCP).

### 3.3 Semi-Thue-Systeme und Chomsky-Grammatiken

#### Semi-Thue-Systeme

**Grundidee:** Semi-Thue-Systeme erlauben die Manipulation von Wörtern durch Regeln zum Ersetzen von Teilwörtern („Textersetzung“).

**Definition.** Ein *Semi-Thue-System*  $T = (\Sigma, P)$  ist gegeben durch:

- ein Alphabet  $\Sigma$ ,
- eine endliche Relation  $P \subseteq \Sigma^* \times \Sigma^*$ .

Die Elemente von  $P$  heißen **Regeln (Produktionen, Produktionsregeln)**.

(Für  $(v, w) \in P$  schreiben wir meist  $v \rightarrow_T w$  oder auch nur  $v \rightarrow w$ .)

Für  $T = (\Sigma, P)$  ist die Relation  $\Rightarrow_T \subseteq \Sigma^* \times \Sigma^*$  gegeben durch:

$$v \Rightarrow_T w \iff \text{es gibt } x, u, u', y \in \Sigma^*, \text{ so dass } v = xuy, w = xu'y \text{ und } u \rightarrow_T u'.$$

Ist  $v \Rightarrow_T w$ , so heißt  $w$  **direkt ableitbar** aus  $v$  (in  $T$ ).

$\Rightarrow_T^*$  sei die reflexive transitive Hülle von  $\Rightarrow_T$ . Ist  $v \Rightarrow_T^* w$ , so heißt  $w$  **ableitbar aus**  $v$  (in  $T$ ). D.h.: Es gilt  $v \Rightarrow_T^* w$ , wenn es  $v_0, \dots, v_l$  ( $l \in \mathbb{N}$ ) gibt mit

$$v = v_0 \Rightarrow_T v_1 \Rightarrow_T v_2 \Rightarrow_T \dots \Rightarrow_T v_l = w.$$

Diese Folge der  $v_0, \dots, v_l$  heißt **Ableitung** (von  $w$  aus  $v$ ), und  $w$  heißt **ableitbar aus**  $v$ .

(Wenn der Kontext klar ist, schreiben wir meist  $\Rightarrow$  bzw.  $\Rightarrow^*$  statt  $\Rightarrow_T$  und  $\Rightarrow_T^*$ ).

#### Beispiel

In dem Semi-Thue-System mit dem Alphabet  $\{a, b\}$  und den Regeln

$$\begin{aligned} aba &\rightarrow a, \\ bba &\rightarrow ba, \\ bba &\rightarrow bab \end{aligned}$$

ist z.B. folgende Ableitung möglich (die jeweils ersetzten Teilwörter sind zur Verdeutlichung unterstrichen):

$$\begin{aligned} ab\underline{bba}a &\Rightarrow ab\underline{baba} \\ &\Rightarrow \underline{abba} \\ &\Rightarrow aba \end{aligned}$$

#### Chomsky-Grammatiken

**Definition.** Seien  $V$  und  $\Sigma$  zwei disjunkte Alphabete. Eine (**Chomsky-**) **Grammatik (Regelgrammatik)**  $G = (V, \Sigma, P, S)$  ist ein Semi-Thue-System  $(V \cup \Sigma, P)$  mit einem ausgezeichneten **Startzeichen**  $S \in V$ .

Die Zeichen von  $V$  heißen (*syntaktische Variablen* (*Nichtterminalzeichen*)), die von  $\Sigma$  *Terminalzeichen*.

**Definition.** Sei  $G = (V, \Sigma, P, S)$  eine Grammatik.  $G$  *erzeugt* ein Wort  $w \in \Sigma^*$ , falls  $S \Rightarrow^* w$  gilt. Die Menge

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid G \text{ erzeugt } w\}$$

heißt die *von  $G$  erzeugte Sprache*.

**Definition.** Sei  $G = (V, \Sigma, P, S)$  eine Grammatik.  $G$  *akzeptiert* ein Wort  $w \in \Sigma^*$ , falls  $w \Rightarrow^* S$  gilt. Die Menge

$$\mathcal{L}_{akz}(G) = \{w \in \Sigma^* \mid G \text{ akzeptiert } w\}$$

heißt die *von  $G$  akzeptierte Sprache*.

### Beispiel

Für die Grammatik  $G_{bsp1}$  mit  $V = \{S, A, C\}$ ,  $\Sigma = \{a, b, c\}$  und den Regeln

$$\begin{aligned} S &\rightarrow \varepsilon, \\ S &\rightarrow aAbc, \\ A &\rightarrow \varepsilon, \\ A &\rightarrow aAbC, \\ Cb &\rightarrow bC, \\ Cc &\rightarrow cc \end{aligned}$$

gilt  $aaabbbccc \in \mathcal{L}(G_{bsp1})$  gemäß der Ableitung

$$\begin{aligned} S &\Rightarrow a\underline{A}bc \\ &\Rightarrow aa\underline{A}bCb \\ &\Rightarrow aaa\underline{A}bCbCb \\ &\Rightarrow aaab\underline{C}bCb \\ &\Rightarrow aaabb\underline{C}Cb \\ &\Rightarrow aaabb\underline{C}bCc \\ &\Rightarrow aaabbb\underline{C}Cc \\ &\Rightarrow aaabbb\underline{C}cc \\ &\Rightarrow aaabbbccc \end{aligned}$$

### Erzeugung und Akzeptanz

Durch „Umkehrung“ ( $v \rightarrow w$  wird zu  $w \rightarrow v$ ) der Regeln von  $G_{bsp1}$  erhält man eine Grammatik  $G_{bsp2}$  mit  $aaabbbccc \in \mathcal{L}_{akz}(G_{bsp2})$ .

(Allgemeiner gilt:  $\mathcal{L}(G_{bsp1}) = \mathcal{L}_{akz}(G_{bsp2})$ .)

**Satz 3.3.1**

- a) Zu jeder Grammatik  $G$  gibt es eine Grammatik  $G'$  mit  $\mathcal{L}(G) = \mathcal{L}_{akz}(G')$ .  
 b) Zu jeder Grammatik  $G$  gibt es eine Grammatik  $G'$  mit  $\mathcal{L}_{akz}(G) = \mathcal{L}(G')$ .

**Bemerkungen**

- Außer der Erzeugung und Akzeptanz (Erkennung) von Sprachen (*generative* bzw. *reduktive* Beschreibung von Sprachen) kann man mit Hilfe von Semi-Thue-Systemen oder Grammatiken auch die Entscheidung von Sprachen und die Berechnung von Funktionen auf Sprachen definieren (mit gleicher Mächtigkeit wie die entsprechenden Begriffe für Turing-Maschinen).
- Man könnte auch (als Grundlage für Aufwands-Begriffe) analog zur Länge und zum Bandverbrauch einer TM-Rechnung die Länge einer Ableitung und die maximal auftretende Länge von Wörtern einer Ableitung in einem Semi-Thue-System bzw. einer Grammatik definieren. (Dies ist allerdings nicht üblich).

**3.4 Erzeugung von Sprachen****Konstruktion (NTM  $\rightarrow$  Grammatik)**

- Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  eine NTM mit  $\Sigma = \{a_1, \dots, a_n\}$ . Die Grammatik  $G_M$  ist gegeben durch  $G_M = (V, \Sigma, P, S)$  mit

$$V = \{S, T, R, L, [, ], A_1, \dots, A_n\} \cup Q \cup (\Gamma \setminus \Sigma)$$

(die neuen Zeichen seien nicht in  $Q$  und  $\Gamma$  enthalten) und den Regeln

$$\begin{array}{ll}
 S \rightarrow [q_0], & \\
 S \rightarrow a_i T[a_i] & \text{für } 1 \leq i \leq n, \\
 T[ \rightarrow [q_0, & \\
 T[ \rightarrow a_i T[A_i] & \text{für } 1 \leq i \leq n, \\
 A_i a_j \rightarrow a_j A_i & \text{für } 1 \leq i, j \leq n, \\
 A_i ] \rightarrow a_i ] & \text{für } 1 \leq i \leq n, \\
 qac \rightarrow bq'c & \text{für } (q', b, R) \in \delta(q, a) \text{ und } c \in \Gamma, \\
 qa] \rightarrow bq'\square] & \text{für } (q', b, R) \in \delta(q, a), \\
 cqa \rightarrow q'cb & \text{für } (q', b, L) \in \delta(q, a) \text{ und } c \in \Gamma, \\
 [qa \rightarrow [q'\square b & \text{für } (q', b, R) \in \delta(q, a), \\
 qa \rightarrow q'b & \text{für } (q', b, N) \in \delta(q, a), \\
 q_+ a \rightarrow R & \text{für } \delta(q_+, a) = \emptyset, \\
 Ra \rightarrow R & \text{für } a \in \Gamma, \\
 R] \rightarrow L, & \\
 aL \rightarrow L & \text{für } a \in \Gamma, \\
 [L \rightarrow \varepsilon. &
 \end{array}$$

- Wirkungsweise von  $G_M$ :
  - Die ersten 6 Regeln erzeugen ein (beliebiges) Wort  $w[q_0w]$  mit  $w \in \Sigma^*$ .
  - Die nächsten 5 Regeln simulieren eine Rechnung  $(\varepsilon, q_0, w) \vdash_M^* \dots$  auf dem Teilwort  $[q_0w]$ . (Jede Konfiguration  $(\alpha, q, \beta)$  von  $M$  wird durch das Teilwort  $[\alpha q \beta]$  repräsentiert.)
  - Falls die simulierte Rechnung akzeptierend ist, wird [...] durch die restlichen Regeln gelöscht.

Es gilt:  $\mathcal{L}(G_M) = \mathcal{L}(M)$ .

### Erzeugung und Semi-Entscheidbarkeit

#### Satz 3.4.1

- a) Zu jeder NTM  $M$  gibt es eine Grammatik  $G$  mit  $\mathcal{L}(G) = \mathcal{L}(M)$ .
- b) Zu jeder Grammatik  $G$  gibt es eine NTM  $M$  mit  $\mathcal{L}(M) = \mathcal{L}(G)$ .

**Satz 3.4.2** Eine Sprache wird genau dann von einer Grammatik erzeugt, wenn sie semi-entscheidbar ist.

### Erzeugung und Aufzählung von Sprachen durch Turing-Maschinen

- In Abschnitt 1.3 auch definiert: Erzeugung von Sprachen durch NTM. Es gilt analog zu Satz 3.4.1: Eine Sprache wird genau dann von einer NTM erzeugt, wenn sie von einer Grammatik erzeugt wird.
- Verwandter Begriff für DTM: Eine Sprache  $L$  über einem Alphabet  $\Sigma$  heißt **rekursiv aufzählbar**, wenn  $L = \emptyset$  ist oder es eine totale berechenbare Funktion  $f : (\Sigma \cup \{|\})^* \rightarrow (\Sigma \cup \{|\})^*$  gibt mit

$$L = \{f(|^n) \mid n \in \mathbb{N}\}.$$

### Bemerkungen

- All diese Begriffe (und einige weitere) sind gleichwertig. Insgesamt sind folgende Aussagen für eine Sprache  $L$  gleichwertig:
  - $L$  ist semi-entscheidbar.
  - $L$  wird von einer NTM akzeptiert.
  - $L$  wird von einer DTM akzeptiert.
  - $L$  wird von einer Grammatik akzeptiert.
  - $L$  wird von einer Grammatik erzeugt.
  - $L$  wird von einer NTM erzeugt.
  - $L$  ist rekursiv aufzählbar.
  - $L$  ist Wertebereich einer berechenbaren Wortfunktion.
  - $L$  ist Definitionsbereich einer berechenbaren Wortfunktion.

- Typische Verwendung der Berechnungsmodelle bezüglich Akzeptanz/Entscheidung und Erzeugung (auch im Folgenden):

Begriff	Berechnungsmodell
Akzeptanz/Entscheidung von Sprachen	Turing-Maschine
Erzeugung von Sprachen	Grammatik

### Wortprobleme für Grammatiken

- Das Problem
  - Entscheide für eine fest gegebene Grammatik  $G = (V, \Sigma, P, S)$ , ob  $w \in \mathcal{L}(G)$  gilt (für  $w \in \Sigma^*$ ).
 heißt *spezielles Wortproblem* (für  $G$ ).
- Das Problem
  - Entscheide, ob  $w \in \mathcal{L}(G)$  gilt (für Grammatik  $G = (V, \Sigma, P, S)$  und  $w \in \Sigma^*$ ).
 heißt *allgemeines Wortproblem* (für Grammatiken).

### Satz 3.4.3

- Das spezielle Wortproblem ist (im Allgemeinen) unentscheidbar.
- Das allgemeine Wortproblem ist unentscheidbar.

# Kapitel 4

## Reguläre Sprachen

### 4.1 Reguläre Grammatiken und endliche Automaten

#### Reguläre Grammatiken

**Definition.** Eine Grammatik  $G = (V, \Sigma, P, S)$  heißt **regulär**, wenn jede Regel von der Form  $A \rightarrow aB$  (**rechtslineare** Regel) oder  $A \rightarrow a$  oder  $A \rightarrow \varepsilon$  ist mit  $A, B \in V$  und  $a \in \Sigma$ .

Eine von einer regulären Grammatik erzeugte Sprache heißt **regulär**.

**Beispiel.** Die Grammatik  $G_{aba} = (\{a, b\}, \{S, A, B, C\}, P, S)$  mit den Regeln

$$\begin{array}{llll} S \rightarrow aS, & A \rightarrow bB, & B \rightarrow aC, & C \rightarrow aC, \\ S \rightarrow aA, & & & C \rightarrow bC, \\ S \rightarrow bS, & & & C \rightarrow \varepsilon \end{array}$$

in  $P$  ist regulär und erzeugt die Sprache (vgl. Abschnitt 1.2)

$$L_{aba} = \{w \in \{a, b\}^* \mid aba \text{ ist Teilwort von } w\}.$$

#### Schreibweise

- Regeln der Form  $A \rightarrow w_1, A \rightarrow w_2, \dots, A \rightarrow w_n$  (d.h. mit gleicher „linker Seite“) werden zusammengefasst zu:

$$A \rightarrow w_1 \mid w_2 \mid \dots \mid w_n.$$

- Anwendung auf das Beispiel von oben:

$$\begin{array}{l} S \rightarrow aS \mid aA \mid bS, \\ A \rightarrow bB, \\ B \rightarrow aC, \\ C \rightarrow aC \mid bC \mid \varepsilon. \end{array}$$

#### Endliche Automaten

- Sei  $M_{aba}$  die NTM mit akzeptierendem Zustand  $q_3$  und der Zustandstafel

	$a$	$b$	$\square$
$q_0$	$(q_0, a, R)$ $(q_1, a, R)$	$(q_0, b, R)$	—
$q_1$	—	$(q_2, b, R)$	—
$q_2$	$(q_3, a, R)$	—	—
$q_3$	$(q_3, a, R)$	$(q_3, b, R)$	—

(Vgl. NTM  $M_{bsp3}$  in Abschnitt 1.2;  $M_{aba}$  akzeptiert  $L_{aba}$ .)

Spezielle Wirkungsweise von  $M_{aba}$ :

- $M_{aba}$  geht nur jeweils „nach rechts“.
  - Die Zeichen der Eingabe werden nicht verändert.
  - $\square$  ist einziges Zeichen von  $\Gamma \setminus \Sigma$ , und die Überföhrungsfunktion ist für kein  $(q, \square)$  definiert. (Nach „Durchlaufen“ der gesamten Eingabe wird somit eine finale Konfiguration erreicht.)
  - Jede akzeptierende Rechnung für eine Eingabe  $w$  endet mit der Konfiguration  $(w, q_3, \varepsilon)$  (d.h. nach Durchlaufen der gesamten Eingabe).
- Eine derartig gegenüber der allgemeinen Form eingeschränkte NTM heißt nichtdeterministischer endlicher Automat. Diese Maschinenart wird üblicherweise – unter Weglassung jetzt überflüssiger TM-Konzepte ( $\Gamma$ , R, L, N, Schreiben von Zeichen) – in nachfolgender eigenständiger Definition präzisiert (wobei allgemein statt nur einem akzeptierenden Zustand eine Menge solcher Zustände zugelassen wird).

**Definition.** Ein (*nichtdeterministischer*) *endlicher Automat (NEA)*  $M = (Q, \Sigma, \delta, q_0, E)$  ist gegeben durch:

- eine endliche Menge  $Q$  von **Zuständen**,
- ein **Eingabealphabet**  $\Sigma$ ,
- eine totale **Überföhrungsfunktion**  $\delta : Q \times \Sigma \rightarrow \mathfrak{P}(Q)$ ,
- einen **Startzustand**  $q_0 \in Q$ ,
- eine Menge  $E \subseteq Q$  von **akzeptierenden Zuständen (Endzuständen)**.

Eine **Konfiguration** von  $M$  ist ein Element  $K \in Q \times \Sigma^*$ .

### Analyse von Sprachen mit endlichen Automaten

Sei  $M = (Q, \Sigma, \delta, q_0, E)$  ein NEA. Die binäre **Übergangsrelation**  $\vdash_M$  (auch kurz:  $\vdash$ ) auf der Menge der Konfigurationen von  $M$  ist definiert wie folgt: Ist  $q' \in \delta(q, a)$ , so gilt

$$(q, aw) \vdash_M (q', w) \quad (w \in \Sigma^*).$$

$\vdash_M^*$  (kurz:  $\vdash^*$ ): reflexive, transitive Hülle von  $\vdash_M$ .

**Nachfolgekongfiguration, Rechnung:** definiert wie bei NTMen (Abschnitt 1.2).

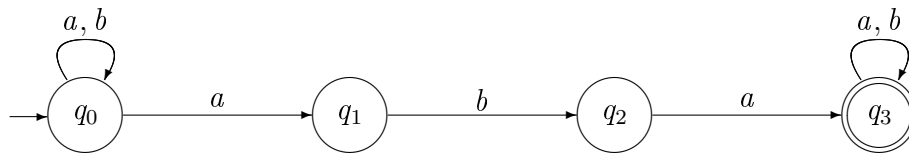
**Definition.** Sei  $M = (Q, \Sigma, \delta, q_0, E)$  ein NEA.  $M$  **akzeptiert** das Wort  $w \in \Sigma^*$ , falls es  $q_+ \in E$  gibt mit  $(q_0, w) \vdash_M^*(q_+, \varepsilon)$ .

Die Menge  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$  heißt die **von  $M$  akzeptierte Sprache**.

### Zustandsübergangsgraphen endlicher Automaten

- Graphische Darstellung von NEAen: wie bei NTMen. Kantenmarkierung jetzt nur noch mit dem jeweils gelesenen Zeichen.

- Beispiel: Darstellung von  $M_{aba}$  von oben:



### Konstruktion (NEA $\rightarrow$ reguläre Grammatik)

- Sei  $M = (Q, \Sigma, \delta, q_0, E)$  ein NEA. Die Grammatik  $G_M$  ist gegeben durch  $G_M = (Q, \Sigma, P, q_0)$  mit den Regeln

$$\begin{aligned} q &\rightarrow aq' && \text{für alle } q, q', a \text{ mit } q' \in \delta(q, a), \\ q &\rightarrow \varepsilon && \text{für alle } q \in E. \end{aligned}$$

- $G_M$  ist regulär, und es gilt:  $\mathcal{L}(G_M) = \mathcal{L}(M)$ .

### Konstruktion (reguläre Grammatik $\rightarrow$ NEA)

- Sei  $G = (V, \Sigma, P, S)$  eine reguläre Grammatik. Der NEA  $M_G$  ist gegeben durch  $M_G = (V', \Sigma, \delta, S, E)$  mit

$$V' = \begin{cases} V \cup \{X\} & \text{mit } X \notin V, \text{ falls } P \text{ eine Regel der Form } A \rightarrow a \text{ enthält} \\ V & \text{sonst,} \end{cases}$$

$$B \in \delta(A, a) \iff P \text{ enthält } A \rightarrow aB \quad (\text{für } B \in V),$$

$$X \in \delta(A, a) \iff P \text{ enthält } A \rightarrow a,$$

$$E = \begin{cases} \{A \in V \mid P \text{ enthält } A \rightarrow \varepsilon\} \cup \{X\}, & \text{falls } X \in V' \\ \{A \in V \mid P \text{ enthält } A \rightarrow \varepsilon\} & \text{sonst.} \end{cases}$$

- Es gilt:  $\mathcal{L}(M_G) = \mathcal{L}(G)$ .

### Ergebnisse

**Satz 4.1.1** Eine Sprache ist genau dann regulär, wenn sie von einem NEA akzeptiert wird.

**Satz 4.1.2** Jede reguläre Sprache ist entscheidbar.

### Deterministische endliche Automaten

**Definition.** Ein NEA  $M = (Q, \Sigma, \delta, q_0, E)$  heißt *deterministisch (DEA)*, wenn  $\delta(q, a)$  für alle  $q \in Q, a \in \Sigma$  höchstens ein Element enthält.

Die Überföhrungsfunktion  $\delta$  eines DEA kann auch als (partielle) Funktion

$$\delta : Q \times \Sigma \rightarrow Q$$

(mit  $\delta(q, a) = q'$  statt  $\delta(q, a) = \{q'\}$  und undefiniertem  $\delta(q, a)$  statt  $\delta(q, a) = \emptyset$ ) verstanden werden.

$M$  heißt **vollständig**, wenn  $\delta(q, a)$  für alle  $q \in Q, a \in \Gamma$  genau ein Element enthält, wenn also  $\delta$  als Funktion  $Q \times \Sigma \rightarrow Q$  total ist. (In diesem Fall endet jede Rechnung mit einer Konfiguration  $(q, \varepsilon)$ .)

**Festlegung:** Im Folgenden werden ausschließlich vollständige DEAs betrachtet.

### Konstruktion (NEA $\rightarrow$ DEA)

- Sei  $N = (Q, \Sigma, \delta, q_0, E)$  ein NEA. Der DEA  $M_N$  ist gegeben durch  $M_N = (\mathfrak{P}(Q), \Sigma, \delta', \{q_0\}, E')$  mit

$$\delta'(U, a) = \bigcup_{q \in U} \delta(q, a) \quad \text{für alle } U \in \mathfrak{P}(Q),$$

$$E' = \{U \in \mathfrak{P}(Q) \mid U \cap E \neq \emptyset\}.$$

- Es gilt:  $\mathcal{L}(M_N) = \mathcal{L}(N)$ .

**Satz 4.1.3** Zu jedem NEA  $M$  gibt es einen DEA  $M'$  mit  $\mathcal{L}(M') = L(M)$ .

### Das Pumping-Lemma

#### Satz 4.1.4 (Pumping-Lemma)

Sei  $L$  eine reguläre Sprache über einem Alphabet  $\Sigma$ . Dann gibt es eine Zahl  $n \in \mathbb{N}$  (**Pumping-Zahl für  $L$** ), so dass gilt: Zu jedem  $x \in L$  mit  $|x| \geq n$  gibt es  $u, v, w \in \Sigma^*$  mit  $x = uvw$  und

1.  $|v| \geq 1$ ,
2.  $|uv| \leq n$ ,
3. Für jedes  $k \in \mathbb{N}$  ist  $uv^k w \in L$ .

## 4.2 Reguläre Ausdrücke

### $\mathcal{R}_\Sigma$ und reguläre Ausdrücke

**Induktive Definition** der Menge  $\mathcal{R}_\Sigma$  von Sprachen über einem Alphabet  $\Sigma$  zusammen mit **regulären Ausdrücken** (über  $\Sigma$ ) zur Beschreibung der jeweiligen Sprache:

1.  $\emptyset$  ist Element von  $\mathcal{R}_\Sigma$  und wird beschrieben durch den regulären Ausdruck  $\emptyset$ .
2.  $\{\varepsilon\}$  ist Element von  $\mathcal{R}_\Sigma$  und wird beschrieben durch den regulären Ausdruck  $\varepsilon$ .

3.  $\{a\}$  ist Element von  $\mathcal{R}_\Sigma$  für jedes  $a \in \Sigma$  und wird beschrieben durch den regulären Ausdruck  $a$ .
4. Sind  $L_r, L_s \in \mathcal{R}_\Sigma$ , beschrieben durch  $r$  bzw.  $s$ , so ist  $L_r \cup L_s \in \mathcal{R}_\Sigma$  und wird beschrieben durch den regulären Ausdruck  $r + s$ .
5. Sind  $L_r, L_s \in \mathcal{R}_\Sigma$ , beschrieben durch  $r$  bzw.  $s$ , so ist  $L_r \circ L_s \in \mathcal{R}_\Sigma$  und wird beschrieben durch den regulären Ausdruck  $r \cdot s$ .
6. Ist  $L_r \in \mathcal{R}_\Sigma$ , beschrieben durch  $r$ , so ist  $L_r^* \in \mathcal{R}_\Sigma$  und wird beschrieben durch den regulären Ausdruck  $r^*$ .

### Schreibweisen und Bezeichnungen

- Bei Zusammensetzung von regulären Ausdrücken: Klammern notwendig.  
Zur Klammerersparnis: Prioritäts-Regelung „ $\cdot$  vor  $+$ “,  
 $r + s + t$  statt  $(r + s) + t$  oder  $r + (s + t)$ ,  
 $r \cdot s \cdot t$  statt  $(r \cdot s) \cdot t$  oder  $r \cdot (s \cdot t)$ .
- $rs$  statt  $r \cdot s$ .
- $\text{REG}_\Sigma$ : Menge aller regulären Ausdrücke über  $\Sigma$ .
- $\mathcal{L}(r)$ : Die durch  $r$  beschriebene Sprache ( $r \in \text{REG}_\Sigma$ ).
- $r, s \in \text{REG}_\Sigma$  heißen *äquivalent*, geschrieben  $r = s$ , wenn  $\mathcal{L}(r) = \mathcal{L}(s)$  gilt.

### Rechnen mit regulären Ausdrücken

Mit Eigenschaften wie in Satz 1.1.1 können Gleichheiten von Sprachen berechnet werden. Für Sprachen aus  $\mathcal{R}_\Sigma$  können solche Rechnungen mit Hilfe regulärer Ausdrücke beschrieben werden.

Einige „Rechenregeln“:

$$\begin{array}{ll}
 r + r = r, & \\
 (r + s) + t = r + (s + t) & \text{(Rechtfertigung für Schreibweise } r + s + t), \\
 (rs)t = r(st) & \text{(Rechtfertigung für Schreibweise } rst), \\
 r(s + t) = rs + rt & \text{(vgl. Satz 1.1.1 c),} \\
 (r^*)^* = r^* & \text{(vgl. Satz 1.1.1 d),} \\
 r^* = rr^* + \varepsilon & \\
 \text{usw.} & 
 \end{array}$$

### Reguläre Ausdrücke und reguläre Sprachen

**Satz 4.2.1** Sei  $\Sigma$  ein Alphabet. Eine Sprache  $L$  über  $\Sigma$  ist genau dann regulär, wenn  $L \in \mathcal{R}_\Sigma$  gilt, d.h. wenn es einen regulären Ausdruck  $r \in \text{REG}_\Sigma$  gibt mit  $\mathcal{L}(r) = L$ .

**Bemerkung:** Aus den Ergebnissen in den Abschnitten 4.1 und 4.2 folgt: Für eine Sprache  $L$  (über einem Alphabet  $\Sigma$ ) sind folgende Aussagen gleichwertig:

- $L$  wird von einer regulären Grammatik erzeugt (d.h.  $L$  ist regulär).
- $L$  wird von einem NEA akzeptiert.
- $L$  wird von einem DEA akzeptiert.
- $L \in \mathcal{R}_\Sigma$ .
- $L$  wird durch einen regulären Ausdruck beschrieben.

### Abschlusseigenschaften regulärer Sprachen

**Satz 4.2.2** Die regulären Sprachen sind abgeschlossen unter den Operationen Vereinigung, Durchschnitt, Komplement, Produkt, Kleene-Stern und Spiegelung. (D.h.: Sind  $L_1, L_2$  reguläre Sprachen, so sind  $L_1 \cup L_2, L_1 \cap L_2, \overline{L_1}, L_1 \circ L_2, L_1^*, L_1^R$  reguläre Sprachen.)

## 4.3 Minimalautomaten

### Der Index einer Sprache

**Definition.** Sei  $L$  eine Sprache über einem Alphabet  $\Sigma$ . Die binäre Relation  $\sim_L \subseteq \Sigma^* \times \Sigma^*$  ist gegeben durch:

$$x \sim_L y \iff \text{für alle } w \in \Sigma^* \text{ gilt } xw \in L \iff yw \in L.$$

#### Lemma 4.3.1

- Falls  $x \sim_L y$ , so  $x \in L \iff y \in L$ .
- $\sim_L$  ist eine Äquivalenzrelation auf  $\Sigma^*$ .

**Definition.** Sei  $L$  eine Sprache über einem Alphabet  $\Sigma$ . Für  $x \in \Sigma^*$  bezeichne  $[x]_{\sim_L}$  die Äquivalenzklasse von  $x$  bzgl.  $\sim_L$ . (D.h.:  $[x]_{\sim_L} = \{y \in \Sigma^* \mid x \sim_L y\}$ .) Der **Index**  $\mathcal{I}(L)$  von  $L$  ist die Anzahl der Äquivalenzklassen in  $\Sigma^*$  bzgl.  $\sim_L$ .

Schreibweise auch kurz:  $[x]$  statt  $[x]_{\sim_L}$ .

**Satz 4.3.2** Sei  $M$  ein DEA mit  $n$  Zuständen. Dann gilt:  $n \geq \mathcal{I}(\mathcal{L}(M))$ .

### Äquivalenzklassenautomaten

Sei  $L$  eine Sprache über einem Alphabet  $\Sigma$  mit  $\mathcal{I}(L) = n$  und den  $n$  verschiedenen Äquivalenzklassen  $[x_1], [x_2], \dots, [x_n]$  bzgl.  $\sim_L$ . Der **Äquivalenzklassenautomat von  $L$**  ist der DEA  $M = (Q, \Sigma, \delta, q_0, E)$  mit

$$\begin{aligned}
Q &= \{[x_1], [x_2], \dots, [x_n]\}, \\
\delta([x], a) &= [xa] \quad \text{für alle } x \in \Sigma^*, a \in \Sigma, \\
q_0 &= [\varepsilon], \\
E &= \{[x] \mid x \in L\}.
\end{aligned}$$

(Leicht nachzurechnen: Die Festlegungen für  $\delta$  und  $E$  sind unabhängig von den Repräsentanten der angegebenen Äquivalenzklassen, der DEA  $M$  ist also wohldefiniert.)

**Satz 4.3.3** Für den Äquivalenzklassenautomaten  $M$  einer Sprache  $L$  gilt:  $\mathcal{L}(M) = L$ .

**Satz 4.3.4 (Satz von Myhill–Nerode)**

Eine Sprache  $L$  ist genau dann regulär, wenn der Index von  $L$  endlich ist.

### Minimalautomaten

- Sätze 4.3.2 und 4.3.3: Ist  $L$  eine reguläre Sprache, so hat jeder DEA, der  $L$  akzeptiert, mindestens  $\mathcal{I}(L)$  Zustände. Der Äquivalenzklassenautomat von  $L$  akzeptiert  $L$  und hat  $\mathcal{I}(L)$  Zustände.
- Der Äquivalenzklassenautomat ist also ein (bis auf die Bezeichnung der Zustände eindeutiger) *Minimalautomat* für  $L$  in dem Sinne, dass es keinen DEA mit weniger Zuständen gibt, der  $L$  ebenfalls akzeptiert.

### Reduzierte Automaten

**Definition.** Sei  $M = (Q, \Sigma, \delta, q_0, E)$  ein DEA, der keine unerreichbaren Zustände enthält (d.h.: zu jedem  $q \in Q$  gibt es  $x \in \Sigma^*$  mit  $(q_0, x) \vdash^* (q, \varepsilon)$ ). Die binären Relationen  $akz_M \subseteq Q \times \Sigma^*$  und  $\approx_M \subseteq Q \times Q$  sind gegeben durch:

$$\begin{aligned}
akz_M(q, x) &\iff (q, x) \vdash^* (q_+, \varepsilon) \text{ für ein } q_+ \in E. \\
q \approx_M q' &\iff \text{für alle } x \in \Sigma^* \text{ gilt } akz_M(q, x) \iff akz_M(q', x).
\end{aligned}$$

Es gilt:  $\approx_M$  ist eine Äquivalenzrelation.

**Definition.** Ein DEA  $M$  ohne unerreichbare Zustände heißt *reduziert*, wenn für beliebige verschiedene Zustände  $q$  und  $q'$  von  $M$  gilt: Es ist nicht  $q \approx_M q'$ .

**Satz 4.3.5** Seien  $M$  ein DEA und  $L = \mathcal{L}(M)$ .  $M$  hat genau dann  $\mathcal{I}(L)$  Zustände, wenn  $M$  reduziert ist.

### Iterative Bestimmung von $\approx_M$

- Für  $k \in \mathbb{N}$  seien die Relationen  $R_k$  definiert durch:

$$q R_k q' \iff akz_M(q, x) \iff akz_M(q', x) \text{ für alle } x \in \Sigma^*, |x| \leq k.$$

Dann gilt:

$$\begin{aligned} \approx_M &= \bigcap_{k=0}^{\infty} R_k, \\ q R_0 q' &\iff q \in E \iff q' \in E, \\ q R_{k+1} q' &\iff q R_k q' \text{ und} \\ &\quad akz_M(q, aw) \iff akz_M(q', aw) \text{ f\u00fcr alle } a \in \Sigma, |w| \leq k \\ &\iff q R_k q' \text{ und } \delta(q, a) R_k \delta(q', a) \text{ f\u00fcr alle } a \in \Sigma. \end{aligned}$$

- Somit:  $R_{k+1} \subseteq R_k$  f\u00fcr alle  $k$ , und es gibt ein  $k_0$ , so dass  $R_{k_0} = R_{k_0+1} = R_{k_0+2} = \dots = \approx_M$  (da  $Q$  endlich ist). Die Relation  $\approx_M$ , die den gesuchten DEA charakterisiert, kann also iterativ bestimmt werden:

- Man beginnt mit  $R_0$ .
- In jedem Iterationsschritt bestimmt man  $R_{k+1}$  aus  $R_k$ , indem man aus  $R_k$  die Zustandspaare  $(q, q')$  entfernt, f\u00fcr die es ein  $a \in \Sigma$  gibt, so dass nicht  $\delta(q, a) R_k \delta(q', a)$  gilt.
- Wenn es solche Zustandspaare nicht mehr gibt, ist die Berechnung von  $\approx_M$  beendet.

### Minimalautomaten-Konstruktion

Gegeben sei ein DEA  $M = (Q, \Sigma, \delta, q_0, E)$  (ohne unerreichbare Zust\u00e4nde). Der folgende Algorithmus konstruiert einen Minimalautomaten  $M'$  zu  $M$ . Als Datenstruktur wird eine Tabelle aller Zustandspaare  $(q, q')$  von  $M$  mit  $q \neq q'$  verwendet.

1. Markiere alle Paare  $(q, q')$  mit  $q \in E, q' \notin E$  oder umgekehrt.  
(\* Diese Paare sind nicht in  $R_0$  \*).
2. Wiederhole so lange, bis sich die Tabelle nicht mehr \u00e4ndert:  
F\u00fcr jedes unmarkierte Paar  $(q, q')$  in der Tabelle:  
Falls es  $a \in \Sigma$  gibt, so dass  $(\delta(q, a), \delta(q', a))$  markiert ist, so markiere  $(q, q')$ .  
(\* Sukzessive Markierung der Paare, die nicht in  $R_1, R_2, \dots$  sind; am Ende der Iteration gilt  $q \approx_M q'$  f\u00fcr alle unmarkierten Paare  $(q, q')$  \*).
3. Bilde maximale Mengen von Zust\u00e4nden, f\u00fcr die paarweise gilt, dass  $(q, q')$  in der Tabelle unmarkiert ist.
4. Diese Mengen (\u00c4quivalenzklassen  $[q]$  in  $Q$  bzgl.  $\approx_M$ ) sind die Zust\u00e4nde von  $M'$ .  
Anfangszustand von  $M'$ :  $[q_0]$ ,  
\u00dcberf\u00fchrungsfunktion von  $M'$ :  $\delta'([q], a) = [\delta(q, a)]$  f\u00fcr alle  $[q]$  und  $a \in \Sigma$ ,  
Endzustandsmenge von  $M'$ :  $\{[q] \mid q \in E\}$ .

# Kapitel 5

## Kontextfreie und kontextsensitive Sprachen

### 5.1 Kontextfreie Sprachen

#### Kontextfreie Grammatiken

Reguläre Grammatiken reichen bereits für einfache Syntaxdefinitionen von Programmiersprachen nicht aus.

**Definition.** Eine Grammatik  $G = (V, \Sigma, P, S)$  heißt **kontextfrei** (abgekürzt: **kf**), wenn jede Regel von der Form  $A \rightarrow x$  ist mit  $A \in V$  und  $x \in (V \cup \Sigma)^*$  (**kontextfreie** Regel). Eine von einer kontextfreien Grammatik erzeugte Sprache heißt **kontextfrei**.

#### Beispiele

1. Die Grammatik  $G_{ab} = (\{S\}, \{a, b\}, P, S)$  mit den Regeln

$$S \rightarrow \varepsilon \mid aSb$$

(in analoger Kurzschreibweise mit „|“ wie bei regulären Grammatiken) ist kontextfrei und erzeugt die (damit also kontextfreie) Sprache  $\{a^i b^i \in \{a, b\}^* \mid i \in \mathbb{N}\}$ .

2. Die Grammatik  $G = (\{A, T, F, I\}, \{(\,), +, -, *, /, x, y, z\}, P, A)$  mit den Regeln

$$\begin{aligned} A &\rightarrow T \mid A+T \mid A-T, \\ T &\rightarrow F \mid T*F \mid T/F, \\ F &\rightarrow I \mid (A), \\ I &\rightarrow x \mid y \mid z \end{aligned}$$

ist kontextfrei und erzeugt eine Sprache der „arithmetischen Ausdrücke“ (über  $x, y, z$ ).

3. (Zum Zusammenhang mit der Linguistik): Die Grammatik mit den Variablen  $\langle \text{Satz} \rangle$  ( $= S$ ),  $\langle \text{Subjekt} \rangle$ , ..., den Terminalzeichen  $\text{der, die, ...}$  und den Regeln

$$\begin{aligned} \langle \text{Satz} \rangle &\rightarrow \langle \text{Subjekt} \rangle \langle \text{intransitives Prädikat} \rangle \\ \langle \text{Subjekt} \rangle &\rightarrow \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle \\ \langle \text{Artikel} \rangle &\rightarrow \text{der} \mid \text{die} \mid \text{das} \\ \langle \text{Attribut} \rangle &\rightarrow \varepsilon \mid \text{hungrige} \mid \text{große} \\ \langle \text{Substantiv} \rangle &\rightarrow \text{Katze} \mid \text{Mensch} \mid \text{Tier} \\ \langle \text{intransitives Prädikat} \rangle &\rightarrow \langle \text{Verb} \rangle \langle \text{Adverb} \rangle \\ \langle \text{Verb} \rangle &\rightarrow \text{schläft} \mid \text{singt} \mid \text{läuft} \\ \langle \text{Adverb} \rangle &\rightarrow \text{laut} \mid \text{leise} \mid \text{schnell} \end{aligned}$$

erzeugt einfache Sätze der deutschen Sprache.

## Reguläre und kontextfreie Sprachen

### Satz 5.1.1

- Jede reguläre Grammatik ist auch eine kontextfreie Grammatik, und jede reguläre Sprache ist auch eine kontextfreie Sprache.
- Es gibt kontextfreie Sprachen, die nicht regulär sind.

## Die Backus-Naur-Form kontextfreier Grammatiken

Kontextfreie Grammatiken werden zur Syntaxdefinition von Programmiersprachen ausgiebig verwendet, dabei meist aber etwas kompakter in **Backus-Naur-Form (BNF)**, oft auch: **erweiterte BNF, EBNF**) aufgeschrieben:

- Die schon verwendete Schreibweise

$$A \rightarrow w_1 \mid w_2 \mid \dots \mid w_n$$

für die  $n$  Regeln  $A \rightarrow w_1, A \rightarrow w_2, \dots, A \rightarrow w_n$  ist bereits ein Teil dieser Notation.

- Außerdem werden noch folgende Abkürzungen verwendet:

$$\begin{aligned} A \rightarrow x \{ w'_1 \mid \dots \mid w'_l \} y & \quad \text{für} \quad A \rightarrow xw'_1y \mid \dots \mid xw'_ly, \\ A \rightarrow x \{ \delta \}^* y & \quad \text{für} \quad A \rightarrow xBy, B \rightarrow \varepsilon \mid \{ \delta \} B, \\ A \rightarrow x \{ \delta \}^+ y & \quad \text{für} \quad A \rightarrow xBy, B \rightarrow \delta \mid \{ \delta \} B, \\ A \rightarrow x [ \delta ] y & \quad \text{für} \quad A \rightarrow xy \mid x \{ \delta \} y \end{aligned}$$

(mit  $\delta$  von der Form  $w'_1 \mid w'_2 \mid \dots \mid w'_l$ ).

## Abschlusseigenschaften kontextfreier Sprachen

**Satz 5.1.2** Die kontextfreien Sprachen sind abgeschlossen unter den Operationen Vereinigung, Produkt, Potenz, Kleene-Stern und Spiegelung. (D.h.: Sind  $L, L_1, L_2$  kontextfreie Sprachen,  $n \in \mathbb{N}$ , so sind  $L_1 \cup L_2, L_1 \circ L_2, L^n, L^*, L^R$  kontextfreie Sprachen.)

### Bemerkungen:

- Die kontextfreien Sprachen sind (im Gegensatz zu den regulären Sprachen) nicht abgeschlossen unter den Operationen Durchschnitt und Komplementbildung.
- Es gilt aber noch:  $L_1$  kontextfrei,  $L_2$  regulär  $\implies L_1 \cap L_2$  kontextfrei.

## 5.2 Einige Eigenschaften kontextfreier Grammatiken

### Ableitungsbäume

Eine Ableitung

$$S = w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n = w$$

eines Wortes  $w \in \Sigma^*$  in einer kontextfreien (also auch in einer regulären) Grammatik  $G = (V, \Sigma, P, S)$  ist graphisch darstellbar durch einen **Ableitungsbaum** (*parsing tree*) für  $w$ .

Dieser Baum  $\mathcal{A}$  lässt sich schrittweise aufbauen:

- $\mathcal{A}$  besitzt zunächst nur die Wurzel  $S$  als einzigen Knoten;
- $\mathcal{A}$  wird sukzessive für  $i = 1, \dots, n$  erweitert wie folgt: Wird im Ableitungsschritt  $w_{i-1} \Rightarrow w_i$  eine Regel  $A \rightarrow x_1 \dots x_m$  ( $x_j \in V \cup \Sigma$  für  $j = 1, \dots, m$ ) angewendet, so werden an den Knoten  $A$  von  $\mathcal{A}$  die Nachfolgeknoten  $x_1, \dots, x_m$  angefügt (im Falle einer Regel  $A \rightarrow \varepsilon$  ein Nachfolgeknoten  $\varepsilon$ ).

### Links- und Rechtsableitung

In einer kontextfreien Grammatik kann es für ein Wort verschiedene Ableitungen geben, je nachdem, welche Variable in einem Schritt als nächste ersetzt wird.

Zwei spezielle „Ableitungsstrategien“:

**Definition.** Eine Ableitung  $S = w_0 \Rightarrow^* w_n = w$  in einer kontextfreien Grammatik heißt **Linksableitung**, wenn in jedem Ableitungsschritt  $w_{i-1} \Rightarrow w_i$  ( $i = 1, \dots, n$ ) die am weitesten links stehende Variable in  $w_{i-1}$  ersetzt wird. Die Ableitung heißt **Rechtsableitung**, wenn in jedem Ableitungsschritt die am weitesten rechts stehende Variable ersetzt wird.

### Mehrdeutigkeit

**Definition.** Eine kontextfreie Grammatik  $G$  heißt **eindeutig**, wenn jedes  $w \in \mathcal{L}(G)$  genau einen Ableitungsbaum besitzt, andernfalls heißt sie **mehrdeutig**. Eine kontextfreie Sprache  $L$  heißt **inhärent mehrdeutig**, wenn jede Grammatik  $G$  mit  $\mathcal{L}(G) = L$  mehrdeutig ist.

### Bemerkungen:

- Die Grammatik  $G = (\{S\}, \{a\}, \{S \rightarrow a \mid S + S \mid S * S\}, S)$  ist mehrdeutig. Die von ihr erzeugte Sprache ist allerdings nicht inhärent mehrdeutig: Die Grammatik  $G'$ , die aus  $G$  entsteht, wenn man die Regeln von  $G$  durch

$$\begin{aligned} S &\rightarrow T \mid S + T, \\ T &\rightarrow a \mid a * T \end{aligned}$$

ersetzt, erzeugt die gleiche Sprache und ist eindeutig.

- Die Sprache  $\{a^i b^j c^k \in \{a, b, c\} \mid i = j \text{ oder } j = k\}$  ist kontextfrei und inhärent mehrdeutig.

### $\varepsilon$ -Freiheit und Kettenregeln

**Definition.** Eine Grammatik  $G = (V, \Sigma, P, S)$  heißt  $\varepsilon$ -frei, wenn gilt:  $G$  enthält keine Regel der Form  $u \rightarrow \varepsilon$  mit  $u \neq S$ , und falls  $G$  die Regel  $S \rightarrow \varepsilon$  enthält, so gilt für alle Regeln  $u \rightarrow v$  von  $G$ , dass  $S$  nicht in  $v$  vorkommt.

**Definition.** Eine Regel einer Grammatik  $G = (V, \Sigma, P, S)$  der Form  $A \rightarrow B$  mit  $A, B \in V$  heißt **Kettenregel**.

### Konstruktion (kf Grammatik $\rightarrow \varepsilon$ -freie kf Grammatik)

- Sei  $G = (V, \Sigma, P, S)$  eine kf Grammatik. Die Grammatik  $G_\varepsilon = (V_\varepsilon, \Sigma, P_\varepsilon, S_\varepsilon)$  ist gegeben wie folgt:
  1. Bestimme die Menge  $U_\varepsilon \subseteq V$  durch:
    - (a)  $U_\varepsilon$  enthält zunächst alle  $A \in V$  mit  $A \rightarrow \varepsilon \in P$ .
    - (b) Erweitere  $U_\varepsilon$  um alle  $A \in V$ , für die es eine Regel  $A \rightarrow A_1 \dots A_k$  mit  $A_1, \dots, A_k \in U_\varepsilon$  in  $P$  gibt.
    - (c) Wiederhole Schritt (b) so lange, bis keine neuen Elemente zu  $U_\varepsilon$  mehr hinzukommen.
  2. Setze  $V_\varepsilon = V$  und  $S_\varepsilon = S$ , falls  $S \notin U_\varepsilon$  (d.h.:  $\varepsilon \notin \mathcal{L}(G)$ ). Andernfalls setze  $V_\varepsilon = V \cup \{S'\}$  (mit einer neuen Variablen  $S'$ ) und  $S_\varepsilon = S'$ .
  3. Bestimme die Regelmengemenge  $P_\varepsilon$  aus  $P$  durch:
    - (a) Entferne aus  $P$  alle Regeln der Form  $A \rightarrow \varepsilon$ .
    - (b) Nimm für jede Regel  $A \rightarrow xBy$  mit  $B \in U_\varepsilon$  und  $xy \neq \varepsilon$  eine neue Regel  $A \rightarrow xy$  hinzu.
    - (c) Wiederhole Schritt (b) so lange, bis keine neuen Regeln mehr hinzukommen.
    - (d) Falls  $\varepsilon \in \mathcal{L}(G)$ , so nimm die Regeln  $S' \rightarrow S$  und  $S' \rightarrow \varepsilon$  hinzu.
- Es gilt:  $G_\varepsilon$  ist kontextfrei und  $\varepsilon$ -frei, und es ist  $\mathcal{L}(G_\varepsilon) = \mathcal{L}(G)$ .

### Konstruktion ( $\varepsilon$ -freie kf Grammatik $\rightarrow \varepsilon$ -freie kf Grammatik ohne Kettenregeln)

- Sei  $G = (V, \Sigma, P, S)$  eine  $\varepsilon$ -freie kf Grammatik. Die Grammatik  $G_{oK} = (V, \Sigma, P_{oK}, S)$  ist gegeben wie folgt:
  1. Entferne alle Regel-, „Zyklen“ durch:
    - (a) Sind  $A_1, \dots, A_k \in V$  und  $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_k \rightarrow A_1$  Regeln in  $P$ , so ersetze alle  $A_2, \dots, A_k$  in allen Regeln von  $P$  durch  $A_1$  und entferne alle Regeln der Form  $C \rightarrow C$ .

- (b) Wiederhole Schritt (a) so lange, bis keine derartigen Zyklen  $A_1, \dots, A_k$  mehr vorhanden sind.
2. Bestimme eine Nummerierung  $B_1, \dots, B_n$  der Variablen von  $V$  so, dass gilt:  
Falls  $B_i \rightarrow B_j \in P$ , so  $i < j$ .
  3. Entferne sukzessive für  $k = n - 1, \dots, 2, 1$  und alle  $j > k$  alle Regeln der Form  $B_k \rightarrow B_j$ , und füge für jede Regel  $B_j \rightarrow x$  die Regel  $B_k \rightarrow x$  hinzu.
- Es gilt:  $G_{oK}$  ist  $\varepsilon$ -frei, kontextfrei, enthält keine Kettenregeln, und es ist  $\mathcal{L}(G_{oK}) = \mathcal{L}(G)$ .

## Ergebnisse

**Satz 5.2.1** Zu jeder kontextfreien Grammatik  $G$  gibt es eine  $\varepsilon$ -freie kontextfreie Grammatik  $G'$  mit  $\mathcal{L}(G') = \mathcal{L}(G)$ .

**Satz 5.2.2** Zu jeder kontextfreien Grammatik  $G$  gibt es eine  $\varepsilon$ -freie kontextfreie Grammatik  $G'$  ohne Kettenregeln mit  $\mathcal{L}(G') = \mathcal{L}(G)$ .

## Normalformen

**Definition.** Eine kontextfreie Grammatik  $G = (V, \Sigma, P, S)$  ist in **Chomsky-Normalform**, wenn sie  $\varepsilon$ -frei ist und wenn alle Regeln von  $G$  (mit der eventuellen Ausnahme von  $S \rightarrow \varepsilon$ ) von der Form  $A \rightarrow BC$  oder  $A \rightarrow a$  sind ( $A, B, C \in V, a \in \Sigma$ ).

**Bemerkung:** Die Chomsky-Normalform ist hauptsächlich für theoretische Betrachtungen von kontextfreien Grammatiken hilfreich. Es gibt noch weitere Normalformen dieser Grammatiken. Speziell wichtig für die praktische Anwendung bei der Syntaxanalyse von Programmen ist folgende Normalform:

**Definition.** Eine kontextfreie Grammatik  $G = (V, \Sigma, P, S)$  ist in **Greibach-Normalform**, wenn sie  $\varepsilon$ -frei ist und wenn alle Regeln von  $G$  (mit der eventuellen Ausnahme von  $S \rightarrow \varepsilon$ ) von der Form  $A \rightarrow aB_1 \dots B_k$  ( $A, B_i \in V, a \in \Sigma, k \geq 0$ ) sind.

## Konstruktion (kf Grammatik $\rightarrow$ kf Grammatik in Chomsky-Normalform)

- Sei  $G = (V, \Sigma, P, S)$  eine beliebige kf Grammatik. Die Grammatik  $G_{CNF} = (V_{CNF}, \Sigma, P_{CNF}, S)$  ist gegeben wie folgt:
  1. Konstruiere aus  $G$  gemäß den obigen Verfahren eine  $\varepsilon$ -freie kf Grammatik  $G'$  ohne Kettenregeln.
  2. Ersetze in  $G'$  jedes  $a \in \Sigma$ , das in (mindestens) einer Regel  $A \rightarrow x$  mit  $|x| \geq 2$  vorkommt, in diesen Regeln durch eine neue Variable  $C_a$ , und füge die Regel  $C_a \rightarrow a$  hinzu.

3. Führe für jede Regel  $A \rightarrow B_1 B_2 \dots B_n$  (mit Variablen  $B_1, \dots, B_n$ ,  $n > 2$ ) neue Variablen  $D_1, \dots, D_{n-2}$  ein, und ersetze diese Regeln durch die neuen Regeln

$$A \rightarrow B_1 D_1, D_1 \rightarrow B_2 D_2, \dots, D_{n-2} \rightarrow B_{n-1} B_n.$$

- Es gilt:  $G_{CNF}$  ist in Chomsky-Normalform, und es ist  $\mathcal{L}(G_{CNF}) = \mathcal{L}(G)$ .

### Ergebnisse

**Satz 5.2.3** Zu jeder kontextfreien Grammatik  $G$  gibt es eine Grammatik  $G'$  in Chomsky-Normalform mit  $\mathcal{L}(G') = \mathcal{L}(G)$ .

**Bemerkung:** Es gilt auch:

Zu jeder kontextfreien Grammatik  $G$  gibt es eine Grammatik  $G'$  in Greibach-Normalform mit  $\mathcal{L}(G') = \mathcal{L}(G)$ .

### Das Pumping-Lemma für kontextfreie Sprachen

#### Satz 5.2.4 (Pumping-Lemma für kontextfreie Sprachen)

Sei  $L$  eine kontextfreie Sprache über einem Alphabet  $\Sigma$ . Dann gibt es eine (Pumping-) Zahl  $n \in \mathbb{N}$  (für  $L$ ), so dass gilt: Zu jedem  $z \in L$  mit  $|z| \geq n$  gibt es  $u, v, w, x, y \in \Sigma^*$  mit  $z = uvwxy$  und

1.  $|vx| \geq 1$ ,
2.  $|vwx| \leq n$ ,
3. Für jedes  $k \in \mathbb{N}$  ist  $uv^k wx^k y \in L$ .

### Entscheidbarkeit kontextfreier Sprachen

**Satz 5.2.5** Jede kontextfreie Sprache ist entscheidbar.

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

Eingabe: Grammatik  $G = (V, \Sigma, P, S)$  in Chomsky-Normalform,

$$w = a_1 \dots a_n \in \Sigma^*;$$

1. FOR  $i = 1, \dots, n$  DO  $V_{i1} := \{A \mid A \rightarrow a_i \in P\}$  ENDDO (\*  $l = 1$  \*)
2. FOR  $l = 2, \dots, n$  DO (\* Iteration über  $l$  \*)
  - FOR  $i = 1, \dots, n - l + 1$  DO (\*  $i > n - l + 1 \implies i + l - 1 > n$  \*)
    - $V_{il} := \emptyset$ ;
    - FOR  $k = 1, \dots, l - 1$  DO
      - $V_{il} := V_{il} \cup \{A \mid A \rightarrow BC \in P, B \in V_{ik}, C \in V_{i+k, l-k}\}$
  - ENDDO
- ENDDO

Ausgabe: „Ja“ (d.h.:  $w \in \mathcal{L}(G)$ ), falls  $S \in V_{1n}$ , sonst „Nein“.

## 5.3 Kellerautomaten

### Kellerautomaten

**Definition.** Ein (*nichtdeterministischer*) **Kellerautomat** (*pushdown automaton*, **KA**)  $M = (Q, \Sigma, \Gamma, \delta, q_0, E)$  ist gegeben durch:

- eine endliche Menge  $Q$  von **Zuständen**,
- ein **Eingabealphabet**  $\Sigma$ ,
- ein **Kelleralphabet**  $\Gamma$  mit einem ausgezeichneten **Startzeichen**  $\# \in \Gamma$ ,
- eine totale **Überföhrungsfunktion**  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathfrak{P}(Q \times \Gamma^*)$  mit der Eigenschaft, dass  $\delta(q, a, A)$  für jedes  $(q, a, A) \in Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$  endlich ist,
- einen **Startzustand**  $q_0 \in Q$ ,
- eine Menge  $E \subseteq Q$  von **akzeptierenden Zuständen** (**Endzuständen**).

Eine **Konfiguration** von  $M$  ist ein Tripel  $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ .

### Analyse von Sprachen mit Kellerautomaten

Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, E)$  ein KA. Die binäre Relation  $\vdash_M$  (auch kurz:  $\vdash$ ) auf der Menge der Konfigurationen von  $M$  ist definiert durch

$$(q, ax, A\beta) \vdash_M (q', x, \gamma\beta) \iff (q', \gamma) \in \delta(q, a, A) \\ \text{(für } a \in \Sigma \cup \{\varepsilon\}, x \in \Sigma^*, A \in \Gamma, \beta, \gamma \in \Gamma^* \text{)}.$$

$\vdash_M^*$  (kurz:  $\vdash^*$ ): reflexive, transitive Hölle von  $\vdash_M$ .

**Definition.** Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, E)$  ein KA.  $M$  **akzeptiert** ein Wort  $w \in \Sigma^*$ , falls es  $q_+ \in E$  gibt mit  $(q_0, w, \#) \vdash_M^* (q_+, \varepsilon, \gamma)$  für ein  $\gamma \in \Gamma^*$ .

Die Menge  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$  heißt die **von  $M$  akzeptierte Sprache**.

### Beispiel

Der Kellerautomat  $M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{A, B, \#\}, \delta, q_0, \{q_0, q_3\})$  mit

$$\begin{aligned} \delta(q_0, a, \#) &= \{(q_1, A\#)\}, \\ \delta(q_0, b, \#) &= \{(q_1, B\#)\}, \\ \delta(q_1, a, B) &= \{(q_1, AB)\}, \\ \delta(q_1, b, A) &= \{(q_1, BA)\}, \\ \delta(q_1, a, A) &= \{(q_1, AA), (q_2, \varepsilon)\}, \\ \delta(q_1, b, B) &= \{(q_1, BB), (q_2, \varepsilon)\}, \\ \delta(q_2, a, A) &= \{(q_2, \varepsilon)\}, \\ \delta(q_2, b, B) &= \{(q_2, \varepsilon)\}, \\ \delta(q_2, \varepsilon, \#) &= \{(q_3, \#)\} \end{aligned}$$

(und  $\delta(\dots) = \emptyset$  in allen sonstigen Fällen) akzeptiert die Sprache  $\{ww^R \mid w \in \{a, b\}^*\}$ .

**Konstruktion (kf Grammatik  $\rightarrow$  KA)**

- Sei  $G = (V, \Sigma, P, S)$  eine kontextfreie Grammatik. Der KA  $M_G$  ist gegeben durch  $M_G = (\{q_0, q_1, q_+\}, \Sigma, V \cup \Sigma \cup \{\#\}, \delta, q_0, \{q_+\})$  mit

$$\begin{aligned}\delta(q_0, \varepsilon, \#) &= \{(q_1, S\#)\}, \\ \delta(q_1, \varepsilon, A) &= \{(q_1, x) \mid A \rightarrow x \in P\} \quad \text{für alle } A \in V, \\ \delta(q_1, a, a) &= \{(q_1, \varepsilon)\} \quad \text{für alle } a \in \Sigma, \\ \delta(q_1, \varepsilon, \#) &= \{(q_+, \#)\}\end{aligned}$$

und  $\delta(\dots) = \emptyset$  in allen sonstigen Fällen.

- Es gilt:  $\mathcal{L}(M_G) = \mathcal{L}(G)$ .

**Konstruktion (KA  $\rightarrow$  kf Grammatik)**

- Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, E)$  ein KA. Die Grammatik  $G_M$  ist gegeben durch  $G_M = (\bar{Q} \times \Gamma \times \bar{Q} \cup \{S\}, \Sigma, P, S)$  mit  $\bar{Q} = Q \cup \{\bar{q}\}$ ,  $\bar{q} \notin Q$  und den Regeln

$$\begin{aligned}S &\rightarrow (q_0, \#, q) \quad \text{für alle } q \in E \cup \{\bar{q}\}, \\ (q, A, q_{m+1}) &\rightarrow a(q_1, B_1, q_2)(q_2, B_2, q_3) \dots (q_m, B_m, q_{m+1}) \\ &\quad \text{für alle } (q_1, B_1 \dots B_m) \in \delta(q, a, A) \text{ und} \\ &\quad \text{für alle Kombinationen beliebiger } q_2, \dots, q_{m+1} \in Q \cup \{\bar{q}\}, \\ (q, A, \bar{q}) &\rightarrow \varepsilon \quad \text{für alle } q \in E \cup \{\bar{q}\}, A \in \Gamma.\end{aligned}$$

- Es gilt:  $G_M$  ist kontextfrei, und es ist  $\mathcal{L}(G_M) = \mathcal{L}(M)$ .

**Ergebnis**

**Satz 5.3.1** Eine Sprache ist genau dann kontextfrei, wenn sie von einem KA akzeptiert wird.

**Deterministische Kellerautomaten**

**Definition.** Ein KA  $M = (Q, \Sigma, \Gamma, \delta, q_0, E)$  heißt **deterministisch (DKA)**, wenn für alle  $q \in Q$ ,  $a \in \Sigma$ ,  $A \in \Gamma$  gilt:

$$|\delta(q, a, A)| + |\delta(q, \varepsilon, A)| \leq 1.$$

**Bemerkung:** DKAen sind immer noch „mächtiger“ als endliche Automaten, jedoch weniger mächtig als KAen. Beispiel: Die Sprache

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

ist kontextfrei (siehe oben), es gibt aber keinen DKA  $M$  mit  $\mathcal{L}(M) = L$ .

Die durch DKAen akzeptierten Sprachen bilden also eine eigene Sprachklasse.

**Definition.** Eine Sprache  $L$  heißt *deterministisch kontextfrei*, wenn es einen DKA  $M$  gibt mit  $\mathcal{L}(M) = L$ .

**Bemerkung:** Diese Sprachen spielen in der Praxis eine große Rolle, insbesondere da sie in linearer Rechenzeit entscheidbar sind. Es gibt auch spezielle kontextfreie Grammatiken – genannt: *LR(k)-Grammatiken* –, die genau solche Sprachen erzeugen.

## 5.4 Kontextsensitive Sprachen

### Monotone Grammatiken

**Definition.** Eine Grammatik  $G = (V, \Sigma, P, S)$  heißt *monoton*, wenn sie  $\varepsilon$ -frei ist und für jede Regel  $u \rightarrow v$  (mit der eventuellen Ausnahme von  $S \rightarrow \varepsilon$ ) gilt:  $|u| \leq |v|$ .

**Bemerkung:**

Jede  $\varepsilon$ -freie kontextfreie Grammatik ist monoton.

**Beispiel**

Die Grammatik  $G_{abc} = (\{S, A, C\}, \{a, b, c\}, P, S)$  mit den Regeln

$$\begin{aligned} S &\rightarrow \varepsilon \mid abc \mid aAbc, \\ A &\rightarrow aAbC \mid abC, \\ Cb &\rightarrow bC, \\ Cc &\rightarrow cc \end{aligned}$$

ist monoton. (Vgl. die ganz ähnliche, allerdings nicht  $\varepsilon$ -freie Grammatik  $G_{bsp1}$  in Abschnitt 3.3.) Es gilt:  $\mathcal{L}(G_{abc}) = \{a^i b^i c^i \in \{a, b, c\}^* \mid i \in \mathbb{N}\}$ .

### Entscheidbarkeit des Wortproblems für monotone Grammatiken

**Algorithmus:**

```

Eingabe: Grammatik  $G = (V, \Sigma, P, S)$ ,  $w \in \Sigma^*$ ;
1.  $n := |w|$ 
2. IF  $n = 0$  THEN  $T := \{u \mid S \rightarrow u \text{ Regel von } G\}$ 
   ELSE  $T := \emptyset$ ;
    $T_{neu} := \{S\}$ ;
   WHILE  $w \notin T$  AND  $T \neq T_{neu}$  DO
      $T := T_{neu}$ ;
      $T_{neu} := T \cup \{u \in (V \cup \Sigma)^* \mid |u| \leq n \text{ und es gibt } u' \in T \text{ mit } u' \Rightarrow u\}$ 
   ENDDO
   ENDIF
Ausgabe: „Ja“ (d.h.:  $w \in \mathcal{L}(G)$ ), falls  $w \in T$ , sonst „Nein“.

```

**Satz 5.4.1** Das (allgemeine) Wortproblem für monotone Grammatiken ist entscheidbar.

### Kontextsensitive Grammatiken

Eine Grammatik  $G = (V, \Sigma, P, S)$  heißt **kontextsensitiv**, wenn  $G$   $\varepsilon$ -frei ist und jede Regel (mit der eventuellen Ausnahme von  $S \rightarrow \varepsilon$ ) von der Form  $uAv \rightarrow uxv$  ist mit  $A \in V$ ,  $u, v \in (V \cup \Sigma)^*$  und  $x \in (V \cup \Sigma)^+$  (**kontextsensitive** Regel).

Eine von einer kontextsensitiven Grammatik erzeugte Sprache heißt **kontextsensitiv**.

### Beispiel:

Die Grammatik  $G = (\{S, A, B\}, \{a, b\}, P, S)$  mit den Regeln

$$\begin{aligned} S &\rightarrow aAB, \\ A &\rightarrow a, \\ aAab &\rightarrow aaAab, \\ AB &\rightarrow AabA, \\ bA &\rightarrow bAab \mid bb \end{aligned}$$

ist kontextsensitiv.

### Bemerkungen

- Jede  $\varepsilon$ -freie kontextfreie Grammatik ist kontextsensitiv, und jede kontextfreie Sprache ist auch eine kontextsensitive Sprache.
- Jede kontextsensitive Grammatik ist monoton.
- Jede kontextsensitive Sprache ist entscheidbar.

### Konstruktion (monotone Grammatik $\rightarrow$ kontextsensitive Grammatik)

- Sei  $G = (V, \Sigma, P, S)$  eine monotone Grammatik. Die Grammatik  $G_{ks} = (V_{ks}, \Sigma, P_{ks}, S)$  ist gegeben wie folgt:

1. Füge für jedes  $a \in \Sigma$  zu  $V$  eine neue Variable  $C_a$  hinzu, ersetze in jeder Regel von  $P$  jedes  $a \in \Sigma$  durch  $C_a$ , und füge die Regeln  $C_a \rightarrow a$  hinzu.
2. Füge für jede nicht kontextsensitive Regel

$$A_1 \dots A_m \rightarrow B_1 \dots B_n \quad (A_1, \dots, A_m, B_1, \dots, B_n \text{ Variablen})$$

mit  $n \geq m \geq 2$  weitere neue Variablen  $D_1, \dots, D_m$  hinzu, und ersetze diese Regeln durch die neuen Regeln

$$\begin{aligned} A_1 A_2 \dots A_m &\rightarrow D_1 A_2 \dots A_m, \\ D_1 A_2 \dots A_m &\rightarrow D_1 D_2 \dots A_m, \\ &\vdots \\ D_1 D_2 \dots D_{m-1} A_m &\rightarrow D_1 D_2 \dots D_{m-1} D_m B_{m+1} \dots B_n, \\ D_1 D_2 \dots D_m B_{m+1} \dots B_n &\rightarrow B_1 D_2 \dots D_m B_{m+1} \dots B_n, \\ &\vdots \\ B_1 \dots B_{m-1} D_m B_{m+1} \dots B_n &\rightarrow B_1 \dots B_{m-1} B_m B_{m+1} \dots B_n. \end{aligned}$$

- Es gilt:  $G_{ks}$  ist kontextsensitiv, und es ist  $\mathcal{L}(G_{ks}) = \mathcal{L}(G)$ .

## Ergebnisse

**Satz 5.4.2** Eine Sprache ist genau dann kontextsensitiv, wenn sie von einer monotonen Grammatik erzeugt wird.

**Satz 5.4.3** Es gibt kontextsensitive Sprachen, die nicht kontextfrei sind.

## Linear beschränkte Automaten

**Definition.** Eine nichtdeterministische Turing-Maschine  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_+)$  mit  $<, > \in \Gamma$  heißt **linear beschränkter Automat (LBA)**, wenn für alle  $q \in Q$  gilt:

- Falls  $(q', b, X) \in \delta(q, <)$ , so  $X \neq L$  und  $b = <$ ,
- Falls  $(q', b, X) \in \delta(q, >)$ , so  $X \neq R$  und  $b = >$ ,

$M$  **akzeptiert** eine Zeichenreihe  $w \in \Sigma^*$ , falls es eine finale Konfiguration  $(\alpha, q_+, \beta)$  gibt mit  $(<, q_0, w>) \vdash_M^* (\alpha, q_+, \beta)$ .

Die Menge  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$  heißt die **von  $M$  akzeptierte Sprache**.

## Beispiel

Der LBA mit  $Q = \{q_0, q_1, q_2, q_3, q_4, q_+\}$ ,  $\Sigma = \{a, b, c\}$ ,  $\Gamma = \{a, b, c, A, B, C, <, >, \square\}$  und folgendem  $\delta$  akzeptiert  $\{a^i b^i c^i \in \{a, b, c\}^* \mid i \in \mathbb{N}\}$ :

$$\begin{aligned} \delta : \quad & (q_0, a) \mapsto \{(q_1, A, R)\}, \\ & (q_0, B) \mapsto \{(q_4, B, R)\}, \\ & (q_0, >) \mapsto \{(q_+, >, N)\}, \\ & (q_1, a) \mapsto \{(q_1, a, R)\}, \\ & (q_1, b) \mapsto \{(q_2, B, R)\}, \\ & (q_1, B) \mapsto \{(q_1, B, R)\}, \\ & (q_2, b) \mapsto \{(q_2, b, R)\}, \\ & (q_2, C) \mapsto \{(q_2, C, R)\}, \\ & (q_2, c) \mapsto \{(q_3, C, L)\}, \\ & (q_3, a) \mapsto \{(q_3, a, L)\}, \\ & (q_3, b) \mapsto \{(q_3, b, L)\}, \\ & (q_3, A) \mapsto \{(q_0, A, R)\}, \\ & (q_3, B) \mapsto \{(q_3, B, L)\}, \\ & (q_3, C) \mapsto \{(q_3, C, L)\}, \\ & (q_4, B) \mapsto \{(q_4, B, R)\}, \\ & (q_4, C) \mapsto \{(q_4, C, R)\}, \\ & (q_4, >) \mapsto \{(q_+, >, N)\}, \\ & (\dots) \mapsto \emptyset \text{ sonst.} \end{aligned}$$

## Die Chomsky-Hierarchie

- Zusammenfassung (mit einigen weiteren üblichen Bezeichnungen):

Sprachbezeichnung	Andere Bezeichnung	Darstellung möglich durch
Regelsprache	Typ-0-Sprache	uneingeschränkte Grammatik DTM NTM
kontextsensitiv	Typ-1-Sprache	kontextsensitive Grammatik monotone Grammatik LBA
kontextfrei	Typ-2-Sprache	kontextfreie Grammatik KA
deterministisch kontextfrei	–	LR( $k$ )-Grammatik DKA
regulär	Typ-3-Sprache	reguläre Grammatik DEA NEA regulärer Ausdruck

- Uneingeschränkte, kontextsensitive, kontextfreie und reguläre Grammatiken heißen auch **Typ-0-**, **Typ-1-**, **Typ-2-** bzw. **Typ-3-Grammatiken**. Für die entsprechenden Sprachklassen der **Typ- $i$ -Sprachen** ( $i = 0, 1, 2, 3$ , bezeichnet mit  $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ ) gilt:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0.$$

- Die vier Sprachklassen  $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$  (bzw. die zugehörigen Grammatikklassen) bilden die **Chomsky-Hierarchie** formaler Sprachen.

# Kapitel 6

## Komplexität von Entscheidungsproblemen

### 6.1 Komplexitätsklassen

#### Komplexität

- Grundlegender Begriff zur Behandlung von Fragestellungen in Bezug auf den Aufwand von algorithmischen Berechnungsverfahren: **Komplexität**
  - eines Algorithmus: Berechnungsaufwand des Algorithmus,
  - einer (Datenverarbeitungs-) Aufgabe: minimale Komplexität aller Algorithmen, die die Aufgabe lösen.
- Für die Komplexität können verschiedene Kenngrößen relevant sein. Die wichtigsten Arten von Komplexität beziehen sich auf:
  - die Rechenzeit („Zeitkomplexität“),
  - den Speicherbedarf („Platzkomplexität“).
- (Typische) „Messung“ der Komplexität: „der Größenordnung nach“ (in *O-Notation*) in Abhängigkeit von der „Größe“ der Aufgabe (*asymptotische* Komplexität).
- Zwei Grundzüge der Komplexitätstheorie:
  - *Spezielle Komplexitätstheorie*: Bestimmung von oberen und unteren Schranken für die Komplexität von Aufgaben. (Fortführung: Entwicklung möglichst *effizienter* Algorithmen.)
  - *Allgemeine Komplexitätstheorie*: Einteilung der Komplexitäten von Aufgaben in Klassen; Existenz von Aufgaben in diesen Klassen; Zusammenhang zwischen den Klassen; usw. (hier ausschließlich verfolgt).

#### Formale Komplexitätsbegriffe

Zur Erinnerung (Abschnitt 1.2):

Für eine Rechnung  $K_0 \vdash_M K_1 \vdash_M K_2 \vdash_M \dots \vdash_M K_l$  einer NTM ist  $l$  ihre **Länge**, und die Gesamtanzahl der verschiedenen Felder des Bandes, auf denen der Schreib-/Lesekopf in der Rechnung zu stehen kommt, heißt ihr **Bandverbrauch**.

**Definition.** Sei  $M$  eine nichtdeterministische Turing-Maschine. Die **Zeitkomplexität**  $t_M : \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$  von  $M$  ist gegeben durch

$$t_M(n) = \text{maximale Länge von Rechnungen von } M \text{ mit Eingaben } w \text{ mit } |w| = n.$$

Die **Platzkomplexität**  $s_M : \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$  von  $M$  ist gegeben durch

$$s_M(n) = \text{maximaler Bandverbrauch von Rechnungen von } M \\ \text{mit Eingaben } w \text{ mit } |w| = n.$$

(Terminiert  $M$  für eine Eingabe nicht, wird als maximale Länge  $\infty$  angesetzt. Der Bandverbrauch kann dann ebenfalls  $\infty$  sein.)

**Bemerkung:** In der Literatur findet man häufig eine etwas kompliziertere Definition, die für die nachfolgenden Betrachtungen aber gleichwertig zu der angegebenen ist.

### Beispiel

Für die NTM  $M_{bsp3}$  aus Abschnitt 1.3 mit der Zustandstafel

	$a$	$b$	$\square$
$q_0$	$(q_0, a, R)$ $(q_1, a, R)$	$(q_0, b, R)$	—
$q_1$	—	$(q_2, b, R)$	—
$q_2$	$(q_+, a, N)$	—	—
$q_+$	—	—	—

gilt:  $t_M(n) = O(n)$  und  $s_M(n) = O(n)$ .

### Komplexitätsklassen

- Im Folgenden ausschließlich betrachtet: Komplexität von Algorithmen zur Entscheidung von Sprachen (interpretiert: Komplexität von Entscheidungsverfahren für Probleme im Sinne von Abschnitt 3.2).
- **Komplexitätsklassen:** Klassen von Sprachen, die von bestimmten TMen mit einer gewissen Zeit- bzw. Platzkomplexität entschieden werden. Beispiel:

$$\text{NLIN} = \{L \mid \text{es gibt eine NTM } M \text{ mit } t_M(n) = O(n), \text{ die } L \text{ entscheidet}\}.$$

### Die Komplexitätsklassen P und NP

**Definition.** Die Komplexitätsklassen P und NP sind definiert durch:

$$\text{P} = \{L \mid \text{es gibt eine DTM } M, \text{ die } L \text{ entscheidet,} \\ \text{und } r \in \mathbb{N} \text{ mit } t_M(n) = O(n^r)\}.$$

$$\text{NP} = \{L \mid \text{es gibt eine NTM } M, \text{ die } L \text{ entscheidet,} \\ \text{und } r \in \mathbb{N} \text{ mit } t_M(n) = O(n^r)\}.$$

**Bemerkungen:**

- Offensichtlich gilt:  $P \subseteq NP$ .
- Offene Frage:  $P \stackrel{?}{=} NP$  (**P=NP - Problem**).  
Praktische Bedeutung:
  - Probleme außerhalb von P: „Praktisch“ unlösbar (da zu ineffizient).
  - Viele wichtige Probleme sind in NP. Wäre  $P = NP$ , könnte man zumindest auf eine brauchbare Algorithmisierung mit *polynomieller* Zeitkomplexität hoffen.
- Vermutung:  $P \neq NP$ .

**Einige Eigenschaften von P und NP**

1. P ist abgeschlossen unter den Operationen Vereinigung, Durchschnitt und Komplement.
2. NP ist abgeschlossen unter den Operationen Vereinigung, Durchschnitt (Abschluss unter Komplementbildung: unbekannt).
3. Es ist  $NP \subseteq EXP$  mit

$$EXP = \{L \mid \text{es gibt eine DTM } M, \text{ die } L \text{ entscheidet,} \\ \text{und } r \in \mathbb{N} \text{ mit } t_M(n) = O(2^{n^r})\}.$$

D.h.: Jedes Problem in NP ist deterministisch mit *exponentieller* Zeitkomplexität lösbar (vgl. Bemerkungen zum P=NP - Problem).

**Bemerkung**

Die Komplexitätsklassen P und NP (und EXP) sind „unbeeinflusst“ vom sehr primitiven Berechnungsmodell der Turing-Maschinen.

**Beispiel:** Der CYK-Algorithmus zur Entscheidung des Wortproblems für eine kontextfreie Grammatik in Chomsky-Normalform (Abschnitt 5.2) hat – bezogen auf die in der Pseudocode-Formulierung verwendeten (und realistischen) „Einzelschritte“ – eine Zeitkomplexität von  $O(n^3)$  (für eine feste Grammatik), ist also polynomiell.

„Implementierung“ des Algorithmus auf einer DTM:  $O(n^r)$  (mit  $r \geq 3$ ), also auch polynomiell.

Insgesamt: Das genannte Wortproblem ist „deterministisch mit polynomieller Zeitkomplexität“ lösbar (d.h. in P) – unabhängig vom gewählten Berechnungsmodell.

## 6.2 NP-vollständige Probleme

### Polynomielle Reduzierbarkeit

**Definition.** Seien  $\Sigma$  ein Alphabet,  $L_1, L_2 \subseteq \Sigma^*$  zwei Sprachen.  $L_1$  heißt *polynomiell reduzierbar auf*  $L_2$  (geschrieben:  $L_1 \leq_p L_2$ ), wenn es eine totale berechenbare Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  gibt, so dass gilt:

- a)  $w \in L_1 \iff f(w) \in L_2$  für alle  $w \in \Sigma^*$ .
- b) Es gibt eine DTM  $M$ , die  $f$  berechnet, und  $r \in \mathbb{N}$  mit  $t_M(n) = O(n^r)$ .

**Satz 6.2.1** Seien  $L_1, L_2$  Sprachen mit  $L_1 \leq_p L_2$ . Dann gilt:

- a) Ist  $L_2 \in \mathbf{P}$ , so ist  $L_1 \in \mathbf{P}$ .
- b) Ist  $L_2 \in \mathbf{NP}$ , so ist  $L_1 \in \mathbf{NP}$ .

### NP-Vollständigkeit

**Definition.** Eine Sprache  $L$  heißt *NP-hart*, wenn  $L' \leq_p L$  für alle  $L' \in \mathbf{NP}$  gilt.  $L$  heißt *NP-vollständig*, wenn  $L \in \mathbf{NP}$  und  $L$  NP-hart ist.

**Satz 6.2.2** Sei  $L$  NP-vollständig. Dann gilt:

$$L \in \mathbf{P} \iff \mathbf{P} = \mathbf{NP}.$$

**Satz 6.2.3** Sei  $L'$  NP-vollständig. Dann gilt:

Ist  $L \in \mathbf{NP}$  und  $L' \leq_p L$ , so ist  $L$  NP-vollständig.

**Bemerkung:** Bedeutung der Sätze 6.2.2 und 6.2.3:

- Könnte man für *eine* NP-vollständige Sprache zeigen, dass sie in  $\mathbf{P}$  ist, so wäre  $\mathbf{P} = \mathbf{NP}$ .
- Methode zum Nachweis der NP-Vollständigkeit einer Sprache  $L$ : Zeige  $L \in \mathbf{NP}$  und  $L' \leq_p L$  für eine andere NP-vollständige Sprache  $L'$ .

### Das Erfüllbarkeitsproblem der Aussagenlogik

Sei  $V = \{v_1, v_2, v_3, \dots\}$  eine abzählbare Menge von *Aussagenvariablen*. Die Menge  $\mathcal{F}$  der *Formeln* (der *Aussagenlogik*) ist induktiv definiert wie folgt:

1.  $v \in V \implies v \in \mathcal{F}$ .
2.  $A, B \in \mathcal{F} \implies \neg A, (A \wedge B), (A \vee B) \in \mathcal{F}$ .

Eine *Belegung* ist eine Abbildung  $\varphi : V \rightarrow \{0, 1\}$ . Jede Belegung  $\varphi$  wird fortgesetzt auf  $\mathcal{F}$  wie folgt:

$$\begin{aligned}\varphi(\neg A) &= 1 - \varphi(A); \\ \varphi(A \wedge B) &= \min(\varphi(A), \varphi(B)); \\ \varphi(A \vee B) &= \max(\varphi(A), \varphi(B)).\end{aligned}$$

Eine Formel  $A \in \mathcal{F}$  heißt **erfüllbar**, wenn es eine Belegung  $\varphi$  gibt mit  $\varphi(A) = 1$ .

**Erfüllbarkeitsproblem der Aussagenlogik (SAT):**

- Entscheide, ob eine (beliebig gegebene) Formel  $A$  der Aussagenlogik erfüllbar ist.

Kodierung von *SAT* als Sprache:

- Jedes  $v_i \in V$  wird kodiert durch  $|^i$ . Damit:  $A \in \mathcal{F}$  ist Wort über  $\{ |, \neg, \wedge, \vee, (, ) \}$ .
- Sprache *SAT* damit:

$$SAT = \{ w \in \{ |, \neg, \wedge, \vee, (, ) \}^* \mid w \text{ ist Kodierung einer erfüllbaren Formel der Aussagenlogik} \}.$$

**Satz 6.2.4** *SAT* ist NP-vollständig.

**Eine Variante von SAT**

Eine Formel  $A \in \mathcal{F}$  heißt in **konjunktiver Normalform (kNF)**, wenn

$$A = B_1 \wedge \dots \wedge B_n$$

und

$$B_i = B_{i1} \vee \dots \vee B_{im_i} \quad (i = 1, \dots, n)$$

gilt ( $n, m_i \geq 1$ ), wobei  $B_{ij} = v$  oder  $B_{ij} = \neg v$  für ein  $v \in V$  ist.

$$SATNF = \{ w \in \{ |, \neg, \wedge, \vee, (, ) \}^* \mid w \text{ ist Kodierung einer erfüllbaren Formel der Aussagenlogik in kNF} \}.$$

**Satz 6.2.5** *SATNF* ist NP-vollständig.

**Einige weitere NP-vollständige Probleme**

1. **3-Erfüllbarkeitsproblem**

Entscheide, ob eine Formel in kNF  $B_1 \wedge \dots \wedge B_n$ , wobei  $B_i = B_{i1} \vee \dots \vee B_{im_i}$ ,  $m_i \leq 3$  ( $i = 1, \dots, n$ ) gilt, erfüllbar ist.

2. **Cliquen-Problem**

Entscheide für einen Graphen  $G$  und  $k \in \mathbb{N}$ , ob  $G$  eine  $k$ -Clique enthält.

3. **Hamilton-Kreis-Problem**

Entscheide, ob ein Graph  $G$  einen Hamilton-Kreis besitzt.

**4. Rucksack-Problem**

Entscheide für eine Folge  $\alpha = (i_1, \dots, i_n)$  ganzer Zahlen und  $k \in \mathbb{Z}$ , ob es eine Teilfolge  $i_{j_1}, \dots, i_{j_r}$  von  $\alpha$  gibt mit  $\sum_{l=1}^r i_{j_l} = k$ .

**5. Traveling-Salesman-Problem**

Entscheide für  $n$  Städte, für die alle Entfernungen voneinander gegeben sind, und eine Zahl  $k$ , ob es eine "Rundreise" (die alle Städte besucht) gibt, deren Gesamtlänge  $\leq k$  ist.

**6. Prozessor-Zuteilungs-Problem**

Entscheide für eine Menge  $M = \{J_1, \dots, J_n\}$  von Aufträgen,  $t \in \mathbb{N}$  (Zeitschranke,  $t \geq 1$ ),  $p \in \mathbb{N}$  (Anzahl der Prozessoren,  $p \geq 1$ ) und eine partielle Ordnung  $<$  auf  $M$ , ob es eine Abbildung  $f : M \rightarrow \{1, \dots, t\}$  gibt derart, dass höchstens  $p$  Aufträge auf die gleiche Zahl abgebildet werden und dass gilt:  $J_i < J_k \implies f(J_i) < f(J_k)$ .

**Das Cliquesproblem**

Sei  $G$  ein Graph mit der Knotenmenge  $V$  und der Kantenmenge  $E$ . Eine  **$k$ -Clique** von  $G$  ist eine Teilmenge  $V'$  von  $V$  mit  $|V'| = k$ , so dass  $(u, v) \in E$  für alle  $u, v \in V'$ ,  $u \neq v$  gilt.

Kodierung des Problems als Sprache (vgl. Sprache  $L_{graph}$  in Abschnitt 1.1; Knotenkodierung:  $|^i$ , Kantenkodierung:  $(|^i \circ |^j)$ ):

$$CLIQUE = \{w = |^k(|^{i_1} \circ |^{j_1}) \dots (|^{i_m} \circ |^{j_m}) \in \{|, \circ, (, )\}^* \mid \text{der durch } w \text{ (ohne } |^k) \text{ kodierte Graph besitzt eine } k\text{-Clique}\}.$$

**Satz 6.2.6**  $CLIQUE$  ist NP-vollständig.

**6.3 Probleme außerhalb NP****Die Komplexitätsklassen PSPACE und NPSPACE**

**Definition.** Die *Komplexitätsklassen* PSPACE und NPSPACE sind definiert durch:

$$PSPACE = \{L \mid \text{es gibt eine DTM } M, \text{ die } L \text{ entscheidet, und } r \in \mathbb{N} \text{ mit } s_M(n) = O(n^r)\}.$$

$$NPSPACE = \{L \mid \text{es gibt eine NTM } M, \text{ die } L \text{ entscheidet, und } r \in \mathbb{N} \text{ mit } s_M(n) = O(n^r)\}.$$

**Bemerkungen:**

- Es gilt:
  - PSPACE = NPSPACE.
  - NP  $\subseteq$  PSPACE  $\subseteq$  EXP.

- Offene Frage:  $NP \stackrel{?}{=} PSPACE$ .

### PSPACE-Vollständigkeit

**Definition.** Eine Sprache  $L$  heißt **PSPACE-hart**, wenn  $L' \leq_p L$  für alle  $L' \in PSPACE$  gilt.  $L$  heißt **PSPACE-vollständig**, wenn  $L \in PSPACE$  und  $L$  PSPACE-hart ist.

**Beispiele:** Folgende Probleme sind PSPACE-vollständig:

- Erfüllbarkeitsproblem für Formeln der **quantifizierten Aussagenlogik (quantifizierte Boolesche Formeln)**.
- Wortproblem für kontext-sensitive Grammatiken.
- Äquivalenzproblem für reguläre Grammatiken (Erzeugen zwei Grammatiken die gleiche Sprache?).
- Probleme für diverse Zweipersonen-Brettspiele wie Dame, Go u.a.: Gibt es eine Gewinnstrategie?

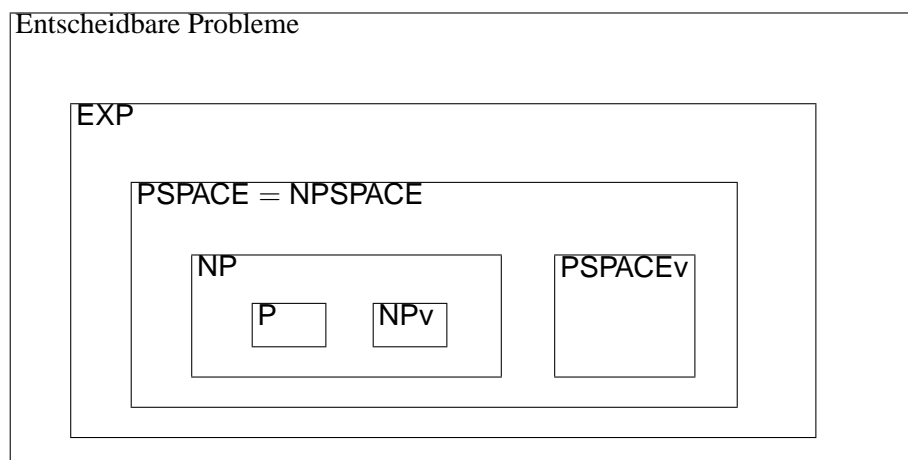
Es gilt: Gibt es eine  $L$  PSPACE-vollständige Sprache mit  $L \in NP$ , so ist  $NP = PSPACE$ .

### Existenz von schwierigen Problemen

**Satz 6.3.1** Für jede totale berechenbare Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  gibt es eine entscheidbare Sprache  $L$  mit der Eigenschaft, dass für jede DTM  $M$ , die  $L$  entscheidet, gilt: Es gibt  $n \in \mathbb{N}$  mit  $t_M(n) > f(n)$ .

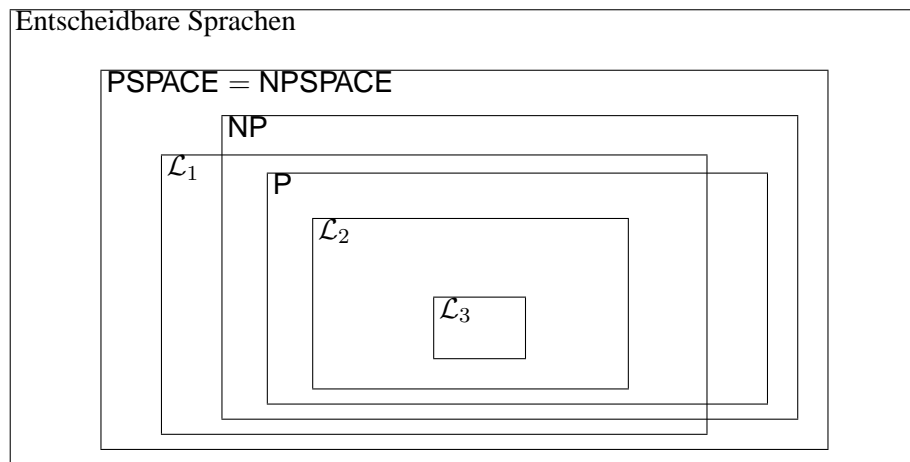
### Zusammenfassung

- (Wahrscheinliches) zusammenfassendes Bild:



(NPv, PSPACEv: Klasse der NP- bzw. PSPACE-vollständigen Probleme.)

- Einige Zusammenhänge zur Chomsky-Hierarchie:



- **Bemerkung:** In der Praxis sind schwierige Probleme (außerhalb von  $P$ ) oft von großer Wichtigkeit. Ansätze zur praktischen Lösung:
  - Spezielle algorithmische Methoden, die gewisse Praxisziele erreichen (z.B.: dynamisches Programmieren, branch-and-bound-Methoden).
  - Probabilistische Verfahren (richtiges Ergebnis oder brauchbare Komplexität nur mit gewisser Wahrscheinlichkeit).
  - Näherungsverfahren (z.B.: Traveling-Salesman-Problem: „suboptimale Routen“).