

# Zusammenfassung Informatik IV

Thomas Marc Jähnel  
jaehnel@in.tum.de

22. August 2003

## Inhaltsverzeichnis

<b>1</b>	<b>Formale Sprachen</b>	<b>3</b>
1.1	Relationen und Graphen . . . . .	3
1.1.1	Zweistellige Relationen . . . . .	3
1.1.2	Wege in Graphen und Hüllenbildung . . . . .	4
1.2	Grammatiken . . . . .	4
1.2.1	Reduktive und generative Grammatiken . . . . .	5
1.2.2	Die Sprachhierarchie nach Chomsky . . . . .	5
1.2.3	Strukturgraphen und Strukturbäume . . . . .	5
1.3	Chomsky-3-Sprachen und endliche Automaten . . . . .	6
1.3.1	Reguläre Ausdrücke . . . . .	6
1.3.2	Endliche Automaten . . . . .	6
1.3.3	Pumping-Lemma . . . . .	6
1.3.4	Minimale Automaten . . . . .	7
1.4	Kontextfreie Sprachen und Kellerautomaten . . . . .	7
1.4.1	Die BNF-Notation . . . . .	7
1.4.2	Kellerautomaten . . . . .	7
1.4.3	Greibach-Normalform . . . . .	8
1.5	Deterministische Grammatiken . . . . .	9
1.5.1	LL(k)-Sprachen . . . . .	9
1.5.2	LR(k)-Sprachen . . . . .	10
1.5.3	Rekursiver Abstieg . . . . .	10
<b>2</b>	<b>Berechenbarkeit</b>	<b>10</b>
2.1	Hypothetische Maschienen . . . . .	10
2.1.1	Turingmaschienen . . . . .	10
2.1.2	Registermaschienen . . . . .	11
2.2	Rekursive Funktionen . . . . .	12
2.2.1	Primitiv rekursive Funktionen . . . . .	12
2.2.2	$\mu$ -Rekursion . . . . .	13
2.2.3	Allgemein rekursive Funktionen . . . . .	13
2.2.4	Churchsche These . . . . .	13
2.3	Entscheidbarkeit . . . . .	13
2.3.1	Entscheidbare Prädikate . . . . .	14
2.3.2	Rekursive und rekursiv aufzählbare Mengen . . . . .	14

<b>3</b>	<b>Komplexitätstheorie</b>	<b>14</b>
3.1	Komplexitätsmaße . . . . .	14
3.1.1	Zeitkomplexität . . . . .	14
3.1.2	Bandkomplexität . . . . .	15
3.1.3	Zeit- und Bandkomplexität von Problemen . . . . .	15
3.1.4	Polynomiale und nichtdeterministisch polynomiale Zeitkomplexität . . . . .	15
3.1.5	Backtracking-Nichtdeterminismus in Programmiersprachen . . . . .	15
3.2	NP-Vollständigkeit . . . . .	15
3.2.1	NP-Vollständigkeitsbeweise . . . . .	16
3.3	Effiziente Algorithmen für NP-vollständige Probleme . . . . .	16
<b>4</b>	<b>Effiziente Algorithmen und Datenstrukturen</b>	<b>17</b>
4.1	Sortieralgorithmen . . . . .	17
4.2	Wege in Graphen . . . . .	17
4.3	Bäume . . . . .	17
4.4	Hashing . . . . .	17

# 1 Formale Sprachen

In diesem Kapitel geht es um *formale Sprachen* und deren theoretische Grundlagen. Zu Beginn werden grundlegende Konzepte der *Relationenalgebra* besprochen, danach wird das Konzept der *Grammatik* eingeführt.

## 1.1 Relationen und Graphen

### 1.1.1 Zweistellige Relationen

Gegeben sei eine Menge  $M$ . Eine *zweistellige*<sup>1</sup> *Relation*  $R$  über der *Grundmenge*  $M$  ist eine Teilmenge von  $M \times M$ . Oft wird auch die Infixnotation verwendet:

$$x R y \iff (x, y) \in R \quad (1)$$

Folgende elementare Relationen existieren:

- *Nullrelation*:  $O_M = \emptyset$
- *vollständige Relation*:  $L_M = M \times M$
- *Identitätsrelation*:  $I_M = \{(x, y) \in M \times M : x = y\}$

Mit folgenden Eigenschaften können Relationen klassifiziert werden. Eine Relation  $R$  heißt

- *reflexiv*, falls  $I_M \subseteq R$
- *symmetrisch*, falls  $R = R^T$
- *antisymmetrisch*, falls  $R \cap R^T \subseteq I_M$
- *asymmetrisch*, falls  $R \cap R^T = O_M$
- *transitiv*, falls  $R \circ R \subseteq R$
- *irreflexiv*, falls  $I_M \cap R = O_M$
- *linkseindeutig*, falls  $R \circ R^T \subseteq I_M$
- *rechtseindeutig*, falls  $R^T \cap R \subseteq I_M$
- *linkstotal*, falls  $I_M \subseteq R \circ R^T$
- *rechtstotal*, falls  $I_M \subseteq R^T \circ R$

Weitere Klassifizierungen sind:

- *partielle Ordnung*: reflexiv, transitiv und antisymmetrisch
- *lineare Ordnung*: partielle Ordnung mit  $R \cap R^T = I_M$
- *Äquivalenzrelation*: transitiv, reflexiv, symmetrisch
- *partielle Funktion*: rechtseindeutig
- *totale Funktion*: rechts- und linkstotal

<sup>1</sup> auch dyadisch oder binär genannt

### 1.1.2 Wege in Graphen und Hüllenbildung

Häufig werden Relationen nicht direkt angegeben, sondern aus einer weniger umfassenden Relation durch *Hüllenbildung* erzeugt.

Die Hüllenbildung läßt sich am Besten an einem Beispiel erläutern. Sei  $M = \{a, b, c\}$  und  $R = \{(a, b), (b, c)\}$ .

- Die *reflexive* Hülle:  $R^{refl} = \{(a, a), (a, b), (b, b), (b, c), (c, c)\}$
- Die *transitive* Hülle:  $R^+ = \{(a, b), (b, c), (a, c)\}$
- Die *reflexiv, transitive* Hülle:  $R^* = \{(a, a), (a, b), (b, b), (b, c), (a, c), (c, c)\}$
- Die *symmetrische* Hülle:  $R^{sym} = \{(a, b), (b, c), (c, b), (b, a)\}$
- Die *symmetrisch, transitiv, reflexive* Hülle:  $R^\otimes = L_M$

Ausserdem werden folgende Abkürzungen verwendet: Für eine Relation  $\rightarrow$  steht  $\rightarrow^*$  für  $R^*$  und  $\leftrightarrow^*$  für  $R^\otimes$ .

**Definition 1.1** Eine Relation heißt *noethersch* wenn in  $R$  keine unendlich fortgesetzten Wege existieren

Beispiele:

- *Nicht-Noethersche* Relation mit Zyklus:  $M = \{a, b, c\}$  und  $R = \{(a, b), (b, c), (c, a)\}$ .
- *Unedliche noethersche* Relation:  $M = N$  und  $R = \{(x, y) \in N \times N : x = y + 1\}$

**Definition 1.2** *Konfluenz und Church-Rosser-Eigenschaft nachtragen!*

## 1.2 Grammatiken

**Definition 1.3** Bei eine Semi-Thou-Grammatik ist durch ein Tripel  $G = \{M, \rightarrow, S\}$  definiert

Bei einer *Chomsky-Grammatik* wird die Menge  $M$  noch in zwei Partitionen unterteilt:

**Definition 1.4** Eine Chomsky-Grammatik  $G$  ist ein *Quadrupel*  $G = (V_N, V_T, \rightarrow, S)$ , wobei  $V_N =$  Menge der Nonterminalsymbole,  $V_T =$  Menge der Terminalsymbole,  $\rightarrow =$  Menge von Ersetzungsregeln und  $S =$  Satzsymbol, *Wurzel oder auch Startzustand*

Die Regeln sind ähnlich zu Textersetzungssystemen in folgender Form angegeben  $\{A \rightarrow aB, B \rightarrow bb, \dots\}$ , wobei auch mehrere Regeln im "BNF-Style" in einer Zeile zusammengefasst werden können  $\{A \rightarrow b|bb\}$

Man sagt, eine Zeichenkette  $\beta$  ist die *direkte Ableitung* einer Zeichenkette  $\alpha$ , wenn eine Regel existiert, die  $\beta$  in einem Schritt in  $\alpha$  überführt. Eine *Ableitung* dagegen, heißt nur, das nach Anwendung beliebiger Regeln<sup>2</sup> irgendwann der Zustand  $\beta$  erreicht werden muss. Ein *Satz* ist eine Satzform die nur aus *Terminalsymbolen* besteht. Die *Sprache* ist die Menge aller Sätze.

<sup>2</sup>die natürlich zur Grammatik gehören müssen

### 1.2.1 Reduktive und generative Grammatiken

Man spricht von einer *generativen* Grammatik, wenn alle Wörter der Sprache ausgehend von der Wurzel abgeleitet werden. Bei einer *reduktiven* Grammatik wird die Sprache anhand ihrer akzeptierten Wörter definiert.

### 1.2.2 Die Sprachhierarchie nach Chomsky

Chomsky-Grammatiken lassen sich anhand der äusseren Form ihrer Regeln klassifizieren. Man spricht von einer *Typ-0-Grammatik* wenn man keine der anderen Eigenschaften feststellen kann.

1. Eine Ersetzungsregel der Form

$$u \circ a \circ v \rightarrow u \circ \langle b \rangle \circ v, \text{ mit } u, v \in (V_T \cup V_N)^*, a \in (V_T \cup V_N)^+, b \in V_N \quad (2)$$

heißt *kontextsensitiv*. Sind alle Ersetzungsregeln kontextsensitiv, so spricht man von einer *kontextsensitiven* Grammatik, oder auch *Chomsky-1-Grammatik*. Chomsky-1-Grammatiken sind *wortlängenmonoton*<sup>3</sup>.

2. Eine Regel der Form

$$a \rightarrow \langle b \rangle \text{ mit } a \in (V_T \cup V_N)^*, b \in V_N \quad (3)$$

heißt *kontextfrei*. Sind alle Regeln kontextfrei so spricht man von einer *Chomsky-2-Grammatik*.

3. Eine kontextfreie Regel der Form

$$m \circ \langle a \rangle \circ n \rightarrow \langle b \rangle \text{ mit } m, n \in V_T^+, a, b \in V_N \quad (4)$$

heißt *beidseitig linear*. Ist zusätzlich  $n = \varepsilon$ , so heißt die Regel *rechtslinear*, mit  $m = \varepsilon$  *linkslinear*. Links- und rechtslineare Regeln heißen auch *einseitig linear*. Eine Regel der Form

$$w \rightarrow \langle b \rangle \text{ mit } w \in V_T^+, b \in V_N \quad (5)$$

heißt *terminal*. Eine Chomsky-Grammatik, deren sämtliche Regeln terminal oder einseitig linear sind<sup>4</sup> heißt *reguläre*, oder *Chomsky-3-Grammatik*.

Hieraus ergeben sich folgende Klassifizierungen für formale Sprachen:

- *REG* die Menge der *regulären Sprachen*
- *CFL* die Menge der *kontextfreien Sprachen*
- *CSL* die Menge der *kontextsensitiven Sprachen*
- *COL* die Menge der *Chomsky-0-Sprachen*

### 1.2.3 Strukturgraphen und Strukturbäume

Ein *Strukturgraph* ist die graphische Darstellung einer Ableitung. Eine Grammatik heißt *eindeutig*, wenn für jedes Wort alle Ableitungen die zur Wurzel führen strukturell äquivalent sind.

<sup>3</sup>die Länge der Wörter ändert sich nicht

<sup>4</sup>wobei hier nur entweder rechts- oder linkslineare Regeln vorkommen dürfen

### 1.3 Chomsky-3-Sprachen und endliche Automaten

Formale Sprachen, die durch Chomsky-3-Grammatiken akzeptiert werden, können stets auch durch *reguläre Ausdrücke* oder durch *endliche Automaten* dargestellt werden.

#### 1.3.1 Reguläre Ausdrücke

Diesen Absatz spar ich mir. Jeder der schon mal Perl programmiert hat kennt die Dinger eh!

#### 1.3.2 Endliche Automaten

Endliche Automaten sind ein sehr einfaches Modell für informationsverarbeitende Maschinen.

**Definition 1.5** Ein endlicher Automat ist definiert durch

$$A = (S, T, s_0, S_Z, \delta) \quad (6)$$

mit folgenden Bestandteilen:

- eine endliche Menge von Zuständen  $S$
- eine endliche Menge von Eingangszeichen  $T$
- ein Anfangszustand  $s_0 \in T$
- eine Menge von Endzuständen  $S_Z \subseteq S$
- eine Übergangsfunktion (bzw. Relation)  $\delta : S \times (T \cup \{\varepsilon\}) \rightarrow \wp(S)$

Man unterscheidet weiterhin zwischen *deterministischen* und *nichtdeterministischen* endlichen Automaten. Ein nichtdeterministischer Automat kann aus einem Zustand mit einem bestimmten Eingabezeichen in *mehrere* Folgezustände übergehen.

Ein Übergang der Form  $s_1 \xrightarrow{\varepsilon} s_2$  heißt  $\varepsilon$ -Übergang.  $\varepsilon$ -Übergänge nennt man *trivial*, wenn gilt  $\delta(s, \varepsilon) \subseteq \{s\}$ . Sind für alle Zustände eines Automaten die  $\varepsilon$ -Übergänge trivial, so heißt der Automat  $\varepsilon$ -frei.

**Definition 1.6** Zu jedem endlichen Automaten existiert ein  $\varepsilon$ -freier endlicher Automat, der die gleiche formale Sprache akzeptiert.

#### 1.3.3 Pumping-Lemma

Um nachzuweisen, daß eine formale Sprache (nicht) regulär ist, kann das *Pumping-Lemma* verwendet werden.

**Definition 1.7** Zu jeder regulären Sprache  $L$  existiert ein  $n$ , so daß sich alle Wörter  $w \in L$  mit  $|w| \geq n$  in Wörter  $x, y, z \in T^*$  zerlegen lassen mit

$$w = x \circ y \circ z \quad (7)$$

mit

- $|y| \geq 1$
- $|x \circ y| \leq n$
- $x \circ y^i \circ z \in L \forall i \in \mathbb{N}$

### 1.3.4 Minimale Automaten

Für jede reguläre Sprache läßt sich ein *minimaler*<sup>5</sup> Automat konstruieren. Dazu geht man, ausgehend von einem  $\epsilon$ -freien, deterministischen, endlichen Automaten, wie folgt vor:

- Man entfernt alle Zustände aus  $S$ , die vom Anfangszustand  $s_0$  nicht erreichbar sind
- man faßt alle äquivalenten Zustände zusammen

## 1.4 Kontextfreie Sprachen und Kellerautomaten

Weil Chomsky-2-Grammatiken mächtiger sind als Chomsky-3-Grammatiken, sind reguläre Ausdrücke und endliche Automaten im allgemeinen nicht geeignet um Typ-2-Sprachen zu beschreiben. Chomsky-2-Sprachen lassen sich aber durch *BNF-Ausdrücke* und *Kellerautomaten* darstellen.

### 1.4.1 Die BNF-Notation

Die *BNF-Notation* sollte ja bereits bekannt sein. Im Prinzip werden reguläre Ausdrücke mit *Hilfszeichen* und *rekursiven Gleichungen* erweitert.

$$\{a^n b^n : n \in \mathbb{N} \setminus \{0\}\} \quad (8)$$

Die formale Sprache aus 8 wird durch folgende BNF-Notation beschrieben:

$$\langle Z \rangle ::= ab \mid a \langle Z \rangle b \quad (9)$$

### 1.4.2 Kellerautomaten

Da wie schon erwähnt endliche Automaten nicht genügen um nichtreguläre, kontextfreie Sprachen zu beschreiben, führen wir dazu die sog. *Kellerautomaten* ein.

**Definition 1.8** Ein Kellerautomat ist definiert durch

$$KA = (S, T, K, \delta, s_0, k_0, S_Z) \quad (10)$$

mit folgenden Bedeutungen:

- Zustandsmenge  $S$
- Menge der Eingabezeichen  $T$
- Menge der Kellerzeichen  $K$
- endliche Übergangsrelation  $\delta$
- Anfangszustand  $s_0 \in S$
- Kellerstartsymbol  $k_0 \in K$
- Menge der Endzustände  $S_Z \subseteq S$

<sup>5</sup>bezogen auf die Anzahl seiner Zustände

Ein Kellerautomat verarbeitet ein Wort  $w \in T^*$ , indem er schrittweise die Zeichen von  $w$  liest und dabei in Abhängigkeit vom Kellerzustand in einen neuen Zustand wechselt, wobei er dabei das oberste Zeichen vom Keller nimmt und durch eine Sequenz<sup>6</sup> ersetzt. Ein Kellerautomat *akzeptiert* ein Wort, wenn die vollständige Verarbeitung der Wortes zu einem Endzustand in  $S_Z$  führt.

**Beispiel** Ein Kellerautomat, der die Sprache aus Gleichung 8 akzeptiert hat folgende Form:

$$KA = (\{s_0, s_1, s_2\}, \{a, b\}, \{|\}, 0\}, \delta, s_0, k_0, \{s_2\}) \quad (11)$$

mit der Übergangsfunktion  $\delta$ :

- $\delta(s_0, a, 0) = \{(s_0, < | 0 >)\}$
- $\delta(s_0, a, |) = \{(s_0, < || >)\}$
- $\delta(s_0, b, |) = \{(s_1, \varepsilon)\}$
- $\delta(s_1, b, |) = \{(s_1, \varepsilon)\}$
- $\delta(s_1, \varepsilon, 0) = \{(s_2, \varepsilon)\}$

Die graphische Darstellung ist Broy S. 245 zu entnehmen.

### 1.4.3 Greibach-Normalform

Um bei einer gegebenen Grammatik einen dazu passenden Kellerautomaten zu finden, läßt sich die *Greibach-Normalform* dieser Grammatik verwenden. Die Konstruktion ist auch ohne die Greibach-Normalform möglich, jedoch erheblich aufwändiger.

**Definition 1.9** Eine kontextfreie Grammatik  $G = (T, N, \rightarrow, Z)$  ist in Greibach-Normalform, wenn jede Ersetzungsregel die folgende Gestalt hat:

$$< a > \circ w \rightarrow x \text{ mit } a \in T, w \in N^*, x \in N \quad (12)$$

Es wird also bei jeder Ersetzung genau ein Terminalzeichen am Rand der Regel verarbeitet. Ausserdem gilt weiter: Jeder von einer  $\varepsilon$ -freien, kontextfreien Grammatik erzeugte Sprachschatz kann auch durch eine kontextfreie Grammatik in Greibach-Normalform erzeugt werden.

Mit Hilfe der letzten Aussage läßt sich also ein Kellerautomat zu jeder dieser Sprachen finden. Die Konstruktion eines Kellerautomaten sieht dann so aus:

$$KA = (\{s_0\}, T, N, \delta, s_0, Z, \{s_0\}) \quad (13)$$

Die Übergangsrelation wird spezifiziert durch:

$$\delta(s_0, a, k) = \{(s_0, w) : < a > \circ w \rightarrow k\} \forall k \in N \quad (14)$$

Es gilt für  $v \in T^*, x \in N^*$ :

$$v \rightarrow^* x \Leftrightarrow (s_0, v, x) \rightarrow^* (s_0, \varepsilon, \varepsilon) \quad (15)$$

**Beispiel** Die formale Sprache aus 8 soll noch einmal als Beispiel dienen. Die Grammatik in Greibach-Normalform zu 8 sieht so aus:

<sup>6</sup>die auch leer sein darf

- $T = \{a, b\}$
- $N = \{Z, U\}$

mit den Regeln

- $aU \rightarrow Z$
- $aZU \rightarrow Z$
- $b \rightarrow U$

Somit erhalten wir den Kellerautomaten aus 13 mit folgender Übergangsrelation:

$$\delta(s_0, a, Z) = \{(s_0, \langle U \rangle), (s_0, \langle ZU \rangle)\} \quad (16)$$

$$\delta(s_0, b, U) = \{(s_0, \epsilon)\} \quad (17)$$

## 1.5 Deterministische Grammatiken

*Deterministische Grammatiken* sind solche, deren Sprachen von deterministischen Kellerautomaten erkannt werden. Sie sind eindeutig und erlauben die Erkennung von Sätzen in  $O(n)$  Schritten. Programmiersprachen sind *kontextfreie, deterministische* Sprachen. Aufgrund dieses Zusammenhangs mit Programmiersprachen, spricht man hier auch von *Syntaxanalyseverfahren* statt von Erkennung. Es werden zwei grundlegende Verfahren zur Syntaxanalyse unterschieden:

**Definition 1.10** *Bei den sog. absteigenden oder top-down Verfahren geht man vom Satzsymbol aus und ersetzt es wiederholt durch Satzformen. Es handelt sich also um ein generatives Verfahren.*

**Definition 1.11** *Bei einem aufsteigenden oder bottom-up Verfahren wird mit der Zeichenkette begonnen und reduktiv so lange höhere Satzformen gebildet, bis man zu einem Satzsymbol gelangt.*

Bei beiden Verfahren wird die Zeichenkette von links nach rechts gelesen und es wird ein *Look-ahead* oder *Vorgriff* verwendet, indem man  $k$  Zeichen hinter dem zuletzt gelesenen aufnimmt. Dabei genügt ein look-ahead von *einem* Zeichen in den meisten Fällen.

### 1.5.1 LL(k)-Sprachen

**Definition 1.12** *Eine Grammatik wird LL(k) genannt, wenn man bei der absteigenden Syntaxanalyse von links nach rechts in jeder Situation, in der man zwischen Alternativen wählen muß, durch einen look-ahead von  $k$  Zeichen entscheiden kann, welche Alternative die richtige ist.*

Bei der  $LL(1)$ -Analyse handelt es sich im wesentlichen um die Konstruktion eines Kellerautomaten aus dem vorangegangenen Abschnitt, mit dem Unterschied, daß der Kellerautomat verzweigt, während hier durch die  $LL(1)$ -Bedingung sofort die richtige Alternative gewählt werden kann.

Die  $LL(1)$ -Analyse wird im *Compilerbau* ausgiebig benutzt.

### 1.5.2 LR(k)-Sprachen

**Definition 1.13** Eine Grammatik wird  $LR(k)$  genannt, wenn bei der aufsteigenden Syntaxanalyse von links nach rechts in jeder Situation durch Betrachtung des gesamten bisher gelesenen Teils der Kette oder durch Reduktion aus ihr entstandenen Satzform und von  $k$  Vorgriffszeichen eindeutig festgestellt werden kann, ob der gelesene Teil eine vollständige Phrase enthält und wenn ja, wie lange sie ist und zu welchem Nonterminalsymbol sie reduziert werden muß.

Alle von links nach rechts analysierbaren Grammatiken sind  $LR(k)$ . Die Grammatiken von Programmiersprachen sind sogar fast ausschließlich  $LR(1)$ , und die  $LR(1)$ -Analyse ist die leistungsfähigste Analysetechnik der Compilerbaus. Dazu werden jedoch umfangreiche Tabellen benutzt, die sich effizient nur mittels Programmen erzeugen lassen, und die viel Speicherplatz benötigen.

Wichtige Sätze über  $LR(k)$ -Grammatiken:

- Es ist nur entscheidbar ob eine gegebene Grammatik  $LR(k)$  ist, für ein gegebenes  $k$
- Zu jeder  $LR(k)$ -Grammatik mit  $k > 1$  gibt es eine äquivalente  $LR(1)$ -Grammatik
- Zu jeder  $LR(k)$ -Grammatik mit Endemarkierung nach jedem Satz gibt es eine äquivalente  $LR(0)$ -Grammatik

### 1.5.3 Rekursiver Abstieg

Ein Verfahren zur Zerteilung von Wörtern einer kontextfreien Sprache. Das Verfahren orientiert sich direkt an der BNF-Darstellung der Syntax. Für jedes nichtterminale Zeichen wird eine Erkennungsprozedur eingeführt, die eine vollständige disjunkte Fallunterscheidung gemäß der BNF-Darstellung macht.

Das Verfahren im Allgemeinen jedoch nicht sequentiell, sondern arbeitet auf dem ganzen Wort.

## 2 Berechenbarkeit

Eine Funktion  $f(x)$  heißt *berechenbar*, wenn ein Algorithmus für  $f$  existiert, der  $\forall x$  eine Ausgabe zu  $f(x)$  liefert.

Da die bisher vorgestellten Modelle von Maschinen nicht genügen um den Begriff der Berechenbarkeit zu erfassen, führen wir ein noch allgemeineres Modell ein.

### 2.1 Hypothetische Maschinen

s

#### 2.1.1 Turingmaschinen

Eine *Turingmaschine* besteht aus einem Band von Zellen, einem Schreib/Lesekopf und einer Steuereinheit mit einer endlichen Menge von Kontrollzeichen.

Das Band wird als beidseitig unendlich angenommen, es soll jedoch in der Regel nur eine endliche Teilmenge betrachtet werden. Hierbei wird # als Platzhalter für leere Zellen verwendet. Ein Tripel aus

$$(s_2, t_2, z) \in \delta(s_1, t_1) \quad (18)$$

ist dabei ein möglicher Berechnungsschritt. Hier wird also ausgehend vom Zustand  $s_1$  mit dem aktuellen Zeichen  $t_1$  der neue Zustand  $s_2$  eingenommen, das Zeichen  $t_2$  an der *aktuellen* Position geschrieben und danach der Schreib/Lesekopf in "Richtung"  $z$  bewegt.

Eine Turingmaschine läßt sich auch graphisch durch einen endlichen Automaten darstellen, wobei eine Kante von  $s_1$  nach  $s_2$  existiert und mit  $(e, a, m)$  beschriftet ist, falls  $(s_2, a, m) \in \delta(s_1, e)$ <sup>7</sup>.

**Definition 2.1** Die Konfiguration einer Turingmaschine ist das *Quadrupel*  $(s, l, a, r)$  mit

- $s$  = aktueller Zustand
- $l$  = alles links vom Schreib/Lesekopf
- $a$  = Zeichen an der aktuellen Position
- $r$  = alles rechts vom Kopf

Eine Berechnung heißt *terminal*, wenn gilt:  $\delta(s, a) = \emptyset$

Eine Berechnung heißt *vollständig*, wenn sie endlich ist und in einer terminalen Konfiguration endet, oder wenn sie unendlich ist.

Da Turingmaschinen eine Darstellung von Algorithmen sind, läßt sich somit unser anfangs erwähnter Berechenbarkeitsbegriff darauf stützen. Eine Funktion heißt *Turing-berechenbar* falls gilt:

1. Es existiert eine endliche vollständige Berechnung mit der terminalen Konfiguration  $(s_x, r, \#, \varepsilon)$  genau dann, wenn die Funktion  $f$  für  $t$  definiert ist und  $f(t) = r$  gilt
2. Es existiert eine unendliche Berechnung genau dann, wenn  $f$  für  $t$  nicht definiert ist

Turingmaschinen lassen sich auch als sog. *Komposition* aneinanderhängen. Daraus resultieren wieder Turing-berechenbare Funktionen.

### 2.1.2 Registermaschinen

*Registermaschinen* kommen den heute gebräuchlichen Rechnerarchitekturen etwas näher als die Turingmaschine. Wir beschränken uns hier auf Registermaschinen mit einer *endlichen* Zahl von Registern, deren jeweiliger Speicherplatz aber *unbeschränkt* ist.

Auf Registermaschinen läßt sich die folgende, induktiv definierte Menge an Programmen ausführen:

- $\varepsilon$  ist ein Programm (leeres Programm)
- $succ_i$  und  $pred_i$  sind Programme<sup>8</sup>
- Sind  $M_1$  und  $M_2$  Programme, so ist auch  $M_1; M_2$  ein Programm
- Ist  $M$  ein Programm, so ist auch  $while_i(M)$  ein Programm

<sup>7</sup>Beispiel siehe Broy S.267

<sup>8</sup>mit  $i$  als Registernummer

## 2.2 Rekursive Funktionen

Unsere bisherige Betrachtung des Berechenbarkeitsbegriffs war an hypothetischen Konstrukten orientiert. Wir wollen nun Algorithmen durch *rekursive Funktionen* darstellen.

### 2.2.1 Primitiv rekursive Funktionen

*Primitiv rekursive Funktionen* sind  $n$ -stellige, totale Funktionen

$$f : N^n \rightarrow N \quad (19)$$

die sich durch Komposition und die Anwendung des Schemas der primitiven Rekursion aus einer Menge einfacher Grundfunktionen gewinnen lassen.

Als *Grundfunktionen* verwenden wir folgende Abbildungen:

$$\begin{aligned} \text{succ} & : N \rightarrow N \\ \text{zero}^{(0)} & : \rightarrow N \\ \text{zero}^{(1)} & : N \rightarrow N \\ \pi_i^n & : N^n \rightarrow N \end{aligned}$$

wobei die letzte Funktion folgendermaßen definiert ist:  $\pi_i^n(x_1, x_2, \dots, x_n) = x_i$

**Definition 2.2** *Durch das Schema der primitiven Rekursion lassen sich durch Komposition aus den vorgegebenen Funktionen weitere gewinnen. Seien*

$$\begin{aligned} g & : N^k \rightarrow N \\ h & : N^{k+2} \rightarrow N \end{aligned}$$

*vorgegebene Funktionen, dann spezifiziert das Schema*

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

*induktiv eindeutig eine neue Funktion*

$$f : N^{k+1} \rightarrow N \quad (20)$$

Primitive Rekursion entspricht einer statischen Wiederholung<sup>9</sup> bei der ein Parameter die Rekursionstiefe festlegt. Daraus ergibt sich, dass bei primitiver Rekursion rekursive Aufrufe *immer* terminieren.

Mit dem Funktional  $pr()$  hat sich eine andere Schreibweise für das in 20 eingebürgert. Man schreibt also auch:

$$f = pr(g, h) \quad (21)$$

**Beispiel Addition**

$$\text{add}(x, 0) = x \quad g(x) = \pi_1^1(x) \quad (22)$$

$$\text{add}(x, \text{succ}(y)) = \text{succ}(\text{add}(x, y)) \quad h(x) = \text{succ}(\pi_3^3(x, y, \text{add}(x, y))) \quad (23)$$

<sup>9</sup>wie *for*-Schleifen

Daraus ergibt sich:

$$add = pr(\pi_1^1, succ \circ [\pi_3^3]) \quad (24)$$

Primitiv rekursive Funktionen sind alle total und Turing-berechenbar. Da es Turing-berechenbare Funktionen gibt, die partiell sind, folgt daraus, dass es nicht alle Turing-berechenbaren Funktionen primitiv rekursiv sind.

Dies lässt sich auch mit Hilfe der Ackermann-Funktion beweisen:

$$ack(n, m) = \begin{cases} m + 1 & \text{falls } n = 0 \\ ack(n - 1, 1) & \text{falls } n > 0, m = 0 \\ ack(n - 1, ack(n, m - 1)) & \text{falls } n > 0, m > 0 \end{cases}$$

Zuerst zeigt man, mittels Induktion über  $n$ , dass die Ackermann-Funktion aus einer Schaar von primitiv rekursiven Funktionen besteht. Danach wird mittels eines Widerspruchsbeweises gezeigt, dass  $ack$  schneller wächst als jede primitiv rekursive Funktion.

### 2.2.2 $\mu$ -Rekursion

Aus dem vorangegangenen “Beweis” ergibt sich die Notwendigkeit einer Erweiterung der rekursiven Funktionen, um auch partielle Funktionen abdecken zu können.

Sei die partielle Funktion

$$f : N^{k+1} \rightarrow N \text{ mit } k \in N \quad (25)$$

gegeben. So ist die Partielle Funktion definiert durch

$$\mu(f) : N^k \rightarrow N \quad (26)$$

mit der Gleichung:

$$\mu(f)(x_1, \dots, x_k) = \min\{y \in N : f(x_1, \dots, x_k, y) = 0\} \quad (27)$$

Beispiele S. 283 ff

### 2.2.3 Allgemein rekursive Funktionen

Die  $\mu$ -Rekursion ist eine Spezialfall der *allgemeinen Rekursion*. Erklärung der allg. rek. Funktionen ???.

### 2.2.4 Churchsche These

Die churchsche These besagt

**Definition 2.3** *Jede intuitiv berechenbare Funktion ist partiell rekursiv*

Diese These lässt sich auf Grund des *informellen* Begriffs “intuitiv” nicht beweisen.

## 2.3 Entscheidbarkeit

Nachdem nun ausführlich der Begriff Berechenbarkeit definiert wurde, geht es jetzt darum, welcher Bereich damit abgedeckt ist und was für Funktionen *nicht* berechenbar sind.

### 2.3.1 Entscheidbare Prädikate

Ein Prädikat  $p$  heißt *entscheidbar*, wenn es einen Algorithmus gibt, der für jeden Satz von Argumenten  $x_1, \dots, x_n$  den *booleschen* Wert  $p(x_1, \dots, x_n)$  berechnet. Das Prädikat heißt *positiv semientscheidbar* falls es einen Algorithmus gibt, der in endlicher Zeit die *true*-Werte richtig berechnet und bei *false*-Ergebnissen nicht terminiert. *Negativ semientscheidbar* ist das Gegenteil davon.

Nichtentscheidbare Funktionen müssen nicht generell für alle Werte nichtentscheidbar sein. Es ist durchaus möglich, dass auf Teilintervallen eine Entscheidbarkeit oder Berechenbarkeit gegeben ist.

### 2.3.2 Rekursive und rekursiv aufzählbare Mengen

Von besonderer Bedeutung sind Prädikate, die die Zugehörigkeit eines Elements zu einer Menge testen. Solche Prädikate heißen *charakteristische Prädikate* der Menge.

Sätze dazu:

1. Jede kontextsensitive Sprache ist rekursiv
2. Jede Chomsky-Sprache ist rekursiv aufzählbar
3. Die Menge von Argumenten für die eine partiell rekursive Funktion nicht terminiert, ist i.A. nicht rekursiv aufzählbar
4. Eine Sprache  $S$  über der Zeichenmenge  $T$  ist genau dann rekursiv, wenn sowohl die Menge  $S$  als auch die Menge  $T^* \setminus S$  rekursiv aufzählbar ist
5. Eine Teilmenge einer rekursiv aufzählbaren Menge ist genau dann rekursiv aufzählbar, wenn ihr charakteristisches Prädikat positiv semientscheidbar ist.

Eine wichtige Anwendung dieser Theorie ist die mathematische Logik.

## 3 Komplexitätstheorie

### 3.1 Komplexitätsmaße

Die hier definierten Begriffe zur Komplexität stützen sich auf die mehrband<sup>10</sup> Turingmaschine.

#### 3.1.1 Zeitkomplexität

Wenn eine Turingmaschine ein Problem in  $n$  Schritten lösen kann so heißt sie  $T(n)$ -Zeitbeschränkte Turingmaschine. Man sagt auch die Turingmaschine hat die *Zeitkomplexität*  $T(n)$ .

Im folgenden wollen wir davon ausgehen:

$$T(n) \geq n + 1 \quad (28)$$

Das heißt, daß wir nur Turingmaschinen betrachten, die immer das gesamte Eingabewort betrachten. Damit werden zwar einige triviale Probleme ausgeschlossen, aber dafür komplexere Probleme vereinfacht.

<sup>10</sup>um realistischere Abschätzungen zu bekommen

### 3.1.2 Bandkomplexität

Die Bandkomplexität zielt nicht auf den Rechenaufwand ab sondern soll den Aufwand an erforderlichem Speicher beschreiben.

Man spricht von einer Bandkomplexität  $S(n)$  wenn für alle Bänder  $i$  gilt, das der Speicherplatzbedarf um Wörter der Länge  $n$  zu verarbeiten  $b_i \leq S(n)$  ist.

### 3.1.3 Zeit- und Bandkomplexität von Problemen

Zeit- und Bandbeschränkte Turingmaschinen lassen sich mit einen konstanten Faktor  $c$  multiplizieren ohne das sich die Grundaussage verändert. Bei Zeitkomplexität gilt das allerdings nur wenn folgende Gleichung erfüllt ist:

$$\inf_{n \rightarrow \infty} \frac{T(n)}{n} = \infty \quad (29)$$

**Definition 3.1** Eine Funktion  $S : N \rightarrow N$  heißt voll Band-konstruierbar, wenn eine  $S(n)$ -bandbeschränkte Turingmaschine existiert, für die gilt: Für alle Zahlen  $n \in N$  existiert ein Eingabewort der Länge  $n$ , für das die Turingmaschine tatsächlich  $S(n)$  Speicherzellen benötigt.

**Definition 3.2** Satz von Savitch: Jedes Akzeptanzproblem in  $NSPACE(S(n))$  liegt auch in  $DSPACE(S(n)^2)$ , wenn  $S$  voll Band-konstruierbar ist und  $\forall n \in N : S(n) \geq \log_2(n)$

### 3.1.4 Polynomiale und nichtdeterministisch polynomiale Zeitkomplexität

Die Menge der Probleme, die in polynomial beschränkter Zeit auf *deterministischen* Maschinen gelöst werden kann bezeichnet man mit  $P$ .  $NP$  bezeichnet den Fall der polynomial zeitbeschränkten *nichtdeterministischen* Lösungen.

Für bandbeschränkte Komplexität läßt sich die selbe Aussage machen, jedoch folgt aus dem Satz von Savitch, dass  $P_{SPACE} = NP_{SPACE}$  ist.

### 3.1.5 Backtracking-Nichtdeterminismus in Programmiersprachen

Broy führt hier einen Nichtdeterminismus in seiner Vorlesungssprache ein. Ein Programm kann also an einer bestimmten Stelle nichtdeterminiert sein. Man bietet mehrere Möglichkeiten der Fortführung an. . .

## 3.2 NP-Vollständigkeit

Es ist bisher noch nicht gelungen die Aussage  $P = NP$  zu zeigen oder zu widerlegen. Es wird jedoch angenommen, dass  $P \neq NP$  gilt, denn es gibt viele Probleme für die bisher kein *effizienter* Algorithmus gefunden wurde und man deshalb annimmt, dass sie nur in  $NP$  nicht aber in  $P$  liegen. Probleme in  $NP$  sind jedoch immer in polynomieller Zeit *überprüfbar*, das heißt es gibt Algorithmen in  $P$  um zu zeigen, dass eine gegebene Lösung korrekt ist oder nicht.

*NP-vollständige* Probleme nennt man nun die Klasse der "schwierigsten" Probleme in  $NP$ . Dazu benötigt man folgenden Begriff:

**Definition 3.3** Problem  $B$  heißt polynomiell reduzierbar auf ein Problem  $A$ , wenn es eine Transformation mit polynomieller Laufzeit gibt, die  $B$  nach  $A$  transformiert.

Gelingt es also ein einziges  $NP$ -vollständiges Problem durch einen Algorithmus in polynomieller Laufzeit zu lösen, ist somit bewiesen, dass  $P = NP$  gilt, denn jedes Problem läßt sich auf ein  $NP$ -vollständiges reduzieren.

Beispiele für  $NP$ -vollständige Probleme sind:

1. *Clique*: Gegeben ist ein ungerichteter Graph mit seinen Kanten. Die Frage ist: Gibt es einen Teilgraph davon mit  $k$  Elementen, die jeweils paarweise miteinander verbunden sind.
2. *Gerichteter Hamiltonkreis*: Gibt es in dem gegebenen Graphen einen Hamiltonkreis<sup>11</sup>
3. *Travelling Salesman problem*: Gesucht ist eine Rundreise durch alle gegebenen Städte mit fest vorgegebener Distanz.

### 3.2.1 NP-Vollständigkeitsbeweise

$NP$ -Vollständigkeitsbeweise werden geführt, indem man folgende Schritte macht:

- Zeige das das Problem  $A \in NP$
- Wähle ein bekanntes  $NP$ -vollständiges Problem  $X$  aus und finde eine Transformation die  $A$  auf  $X$  reduziert

## 3.3 Effiziente Algorithmen für $NP$ -vollständige Probleme

Es gibt verschiedene Tricks mit denen  $NP$ -vollständige Probleme zumindest in Teilbereichen effizient gelöst werden können

- *Branch and Bound* Geschicktes Durchsuchen im Lösungsraum. Einfach auszuwertende Zweige werden am Anfang behandelt. Bei frühzeitiger Erkennung, wird ein bestimmter Weg schneller abgebrochen.
- *Approximative Verfahren*: Algorithmen, die *annähernd* optimale Lösungen liefern, dafür aber in polynomiller Zeit lösbar sind.
- *Dynamische Programmierung*: Mit erhöhtem Speicherbedarf aber geringerer Laufzeit Tabellen von bereits berechneten Lösungen anlegen, um den aufwand bei gleichen Teilstrukturen in Bäumen zu reduzieren
- *Probabilistische Algorithmen*: Algorithmen die nur mit einer gewissen Wahrscheinlichkeit ein korrektes Ergebnis liefern, oder nur mit einer gewissen Wahrscheinlichkeit terminieren.
- *Einschränkung der Problemklasse*: Einfachere Probleme betrachten
- *Heuristische Methoden*: Häufig lassen sich auch heuristische Verfahren finden die gut funktionieren.
- *Greedy-Algorithmen*: Greedy-Algorithmen versuchen die Laufzeit zu optimieren dadurch, dass sie immer die *lokal* beste Lösung verwenden und weiterarbeiten.

<sup>11</sup>jeder Knoten genau 1mal, geschlossener Pfad

## 4 Effiziente Algorithmen und Datenstrukturen

Wir betrachten nun die Zeit- und Speicherkomplexität von verschiedenen Algorithmen.

### 4.1 Sortieralgorithmen

Name	Zeitavarage	Zeitworst	Speicher	Anwendung
Insert Bubble Select	$O(n^2)$	$O(n^2)$	-	Bei sehr kurzen Sequenzen
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	günstig wenn im RAM sortiert wird
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$	
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n)$	

### 4.2 Wege in Graphen

Der *Warshall*-Algorithmus berechnet die transitive Hülle eines Graphen. Er läßt sich mit Hilfe von arrays von einer ursprünglichen Komplexität von  $O(3^n)$  auf  $O(n^3)$  optimieren.

### 4.3 Bäume

Obwohl Bäume schon im Detail besprochen wurden, werden hier noch spezielle Bäume mit bestimmten Eigenschaften vorgestellt, die für bestimmte Aufgaben besonders gut geeignet sind.

Ein Baum läßt sich nach einem vorgegebenen Schema linear in einem Array ablegen. Dies ist nur effizient, wenn der Baum ausgeglichen und vollständig ist, und die Struktur nicht während der Laufzeit geändert werden muss.

**Definition 4.1** *Ein AVL-Baum ist ein sortierter Binärbaum, bei dem sich in allen Teilbäume die Höhe des linken und rechten Teils um maximal 1 unterscheidet. Deswegen nennt man ihn auch balanciert.*

Es gibt auch noch B-Bäume. Ist mir aber zu theoretisch :p

### 4.4 Hashing

Beim Hashing<sup>12</sup> wird eine sog. Hashfunktion dazu benutzt Schlüssel auf ein Array abzubilden und somit den Zugriff auf gespeicherte Elemente zu ermöglichen. Die Elemente selber liegen dabei in einem linearen Array und die Zuordnung von hash-key und index im array wird in einer Tabelle gespeichert.

<sup>12</sup>bei Broy *Streuspeicherverfahren* genannt