Einführung in die Informatik 4
Vorlesungsmitschrift

Simon Bierbaum Martin von Gagern

Sommersemester 2003 – Stand 15. August 2003

Inhaltsverzeichnis

1 Formale Sprachen **2**

1.1 Relationen, Graphen . . . 2

1.1.1 Relationen . . . 2

1.1.2 Wege in Graphen / Hüllenbildung . . . 3

1.2 Grammatiken . . . 4

1.2.1 Reduktive und generative Grammatik . . . 5

1.2.2 Die Sprachhierarchie nach Chomsky . . . 7

1.2.3 Strukturelle Äquivalenz von Ableitungen . . . 7

1.2.4 Sackgassen, unendliche Ableitungen . . . 8

1.3 Chomsky-3-Sprachen, endlich Automaten, reguläre Ausdrücke . . . 9

1.3.1 Reguläre Ausdrücke . . . 9

1.3.2 Endliche Automaten . . . 10

1.3.3 Äquivalenz der Darstellungsform . . . 11

1.3.4 Äquivalenz RA, EA, Chomsky-3-Grammatiken . . . 11

1.3.5 Minimale Automaten . . . 13

1.4 Kontextfreie Sprachen (KfS) und Kellerautomaten (KA) . . . 14

1.4.1 BNF-Notation . . . 14

1.4.2 Kellerautomaten . . . 14

1.4.3 Äquivalenz von Kellerautomaten und KfS . . . 15

1.4.4 Greibach-Normalform . . . 16

1.4.5 LR(k)-Sprachen (LR = Left-Rightmost) . . . 17

1.4.6 LL(k)-Grammatiken (Left-leftmost) . . . 18

1.4.7 Rekursiver Abstieg . . . 19

1.4.8 Das Pumping-Lemma für KfS . . . 19

1.5 Kontextsensitive Grammatiken (KsG) . . . 20

2 Berechenbarkeit **21**

2.1 Hypothetische Maschinen . . . 21

2.1.1 Turing-Maschine . . . 21

2.1.2 Register-Maschinen . . . 23

2.2 Rekursive Funktionen . . . 24

2.2.1 Primitiv rekursive Funktionen . . . 24

2.2.2 μ-rekursive Funktionen . . . 26

2.2.3 Allgemeine Bemerkungen zur Rekursion . . . 27

2.3 Äquivalenz der Berechenbarkeitsbegriffe . . . 27
2.3.1 Äquivalenz von μ-Berechenbarkeit und TM-Berechenbarkeit . . . 27
2.3.2 Äquivalenz RM/TM-Berechenbarkeit . . . 28
2.3.3 Church's These . . . 28
2.4 Entscheidbarkeit . . . 28
2.4.1 Nichtberechenbare Funktionen . . . 28
2.4.2 (Nicht)Entscheidbare Prädikate . . . 29
2.4.3 Rekursive und rekursiv aufzählbare Mengen . . . 30

3 Komplexitätstheorie **31**
3.1 Komplexitätsmaße . . . 31
3.1.1 Zeitkomplexität . . . 31
3.1.2 Bandkomplexität . . . 32
3.1.3 Zeit- und Bandkomplexitätsklassen . . . 32
3.1.4 Polynomiale und nichtdeterministisch polynomiale Zeitkomplexität . . . 33
3.1.5 Nichtdeterminismus in Algorithmen – Backtracking . . . 33

Vorlesungstage

10.4.2003 . . . 1
11.4.2003 . . . 3
17.4.2003 . . . 5
24.4.2003 . . . 7
25.4.2003 . . . 9
2.5.2003 . . . 11
8.5.2003 . . . 13
9.5.2003 . . . 16
15.5.2003 . . . 18
16.5.2003 . . . 22
22.5.2003 . . . 24
23.5.2003 . . . 26
30.5.2003 . . . 28
5.6.2003 . . . 30
6.6.2003 . . . 32
12.6.2003 . . . 33

10.4.2003

Themen

Formale Sprachen	↔ Beschreibungsmächtigkeit
Berechenbarkeit	↔ Grenze der Algorithmik
Komplexität	↔ Aufwand in Berechnungen für Probleme
Effiziente Algorithmen	↔ aufwändige Algorithmen optimieren
Logikprogrammierung	
Effiziente Datenstrukturen	↔ dito für Datenstrukturen
Datenbankmodelle	

Kapitel 1

Formale Sprachen

Zeichensatz A
$A*$ Menge der Wörter über A
$L \subseteq A*$ formale Sprache

Frage: Mit welchen formalen Mitteln können wir Sprachen $L \subseteq A*$ beschreiben?

1.1 Relationen, Graphen

1.1.1 Relationen

Gegeben Grundmenge M. Zweistellige (binäre, dyadische) Relation $R \subseteq M \times M$

Zur Notation: $(x, y) \in R$ gleichwertig zu xRy

Beispiel: $\leq \subseteq \mathbb{N} \times \mathbb{N} \qquad (1,3) \in \leq \qquad 1 \leq 3$

Operationen auf Relationen: $R_1, R_2 \subseteq M \times M$:

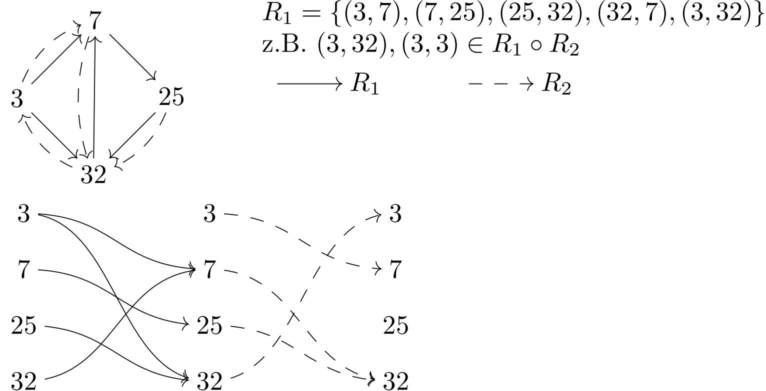
Durchschnitt	$R_1 \cap R_2$
Vereinigung	$R_1 \cup R_2$
Komplement	$R^- = \{(x,y) \in M \times M : (x,y) \notin R\}$
Konverse Relation	$R^T = \{(x,y) \in M \times M : (y,x) \in R\}$
Relationenprodukt	$R_1 \circ R_2 = \{(x,z) \in M \times M : \exists y \in M : (x,y) \in R_1 \wedge (y,z) \in R_2\}$

Monotonie: $R_1 \subseteq R_1' \wedge R_2 \subseteq R_2' \Rightarrow R_1 \circ R_2 \subseteq R_1' \circ R_2'$

Frage: Wie Relationen definieren?

Durch Aufzählung: $\{(3,7), (2,8), (12,33), \ldots\}$
Logisch: $\{x, y) : P(x,y)\}$ (Prädikat)

Graphisch:



$R_1 = \{(3,7), (7,25), (25,32), (32,7), (3,32)\}$
z.B. $(3,32), (3,3) \in R_1 \circ R_2$
$\longrightarrow R_1 \qquad \dashrightarrow R_2$

Boolesche Matrix:

	3	7	25	32
3	O	L	O	L
7	O	O	L	O
25	O	O	O	L
32	O	L	O	O

Spezielle Relationen über M:

Nullrelation:	$O_M = \emptyset$
Vollständige Relation:	$L_M = (M \times M)$
Identität:	$I_M = \{(x, y) \in M \times M : x = y\}$
	$O_M^- = L_M$
	I_M^-

Eigenschaften von Relationen $R \subseteq M \times M$:

reflexiv:	$I_M \subseteq R$	
symetrisch:	$R = R^T$	
antisymetrisch:	$R \cap R^T \subseteq I_M$	
asymetrisch:	$R \cap R^T = O_M$	
transitiv:	$R \circ R \subseteq R$	
irreflexiv:	$I_M \cap R = O_M$	(schlingenfreier Graph)
linkseindeutig:	$R \circ R^T \subseteq I_M$	(injektiv)
rechtseindeutig:	$R^T \circ R \subseteq I_M$	
rechtstotal:	$I_M \subseteq R^T \circ R$	(surjektiv)
linkstotal:	$I_M \subseteq R \circ R^T$	

Relationsarten:

Quasiordnung, Präordnung:	transitiv, reflexiv
Striktordnung:	transitiv, asymetrisch
partielle Ordnung:	transitiv, antisymetrisch, reflexiv
Lineare Ordnung:	transitiv, antisymetrisch, reflexiv, $R \cup R^T = L_M$
Äquivalenzrelation:	transitiv, reflexiv, symetrisch
partielle Funktion:	rechtseindeutig
totale Funktion:	rechtseindeutig, linkstotal

Für eine gegebene Relation R über M heißt ein Element $x \in M$

maximal g.d.w. $\forall y \in M : xRy \Rightarrow x = y$ („kein anderes ist größer“)

größtes Element g.d.w. $\forall y \in M : yRx$ („größer als alle anderen“)

11.4.2003

Eine zweistellige Relation nenn wir auch **gerichteter Graph**, eine symetrische zweistellige Relation auch **ungerichteter Graph**. Wir können für Graph R Kantenmarkierungen $\beta : R \to S$ Knotenmarkierungen $\alpha : M \to S'$ einführen.

1.1.2 Wege in Graphen / Hüllenbildung

Iteriertes Relationenprodukt: $R \subseteq MxM$ $\quad R^0 = I_M$ $\quad R^{i+1} = R^i \circ R$

Ein **Weg** im Graph R von x_0 nach x_n ist eine Folge von „Knoten“ $x_0, \ldots, x_n \in M$ mit $\forall i, 1 \leq i \leq n : x_{i-1}Rx_i$

Satz: Ein **Weg** von x nach y der Länge n existiert genau dann, wenn xR^ny. Beweis: Induktion über $n \in \mathbb{N}$

Ist R partielle Ordnung, dann heißen Wege auch **Ketten**. Wir lassen dann auch unendliche Wege zu. R heißt **noethersch**, wenn es keine unendlichen Wege gibt, in denen sich Knoten nicht wiederholen (streng aufsteigende Ketten).

> Aus dem Buch:
>
> Eine Folge $\{x_i\}_{i \in \mathbb{N}}$ von Elementen heißt **unendlich fortgesetzter Weg** in R ausgehend von x_0, wenn gilt $\forall i \in \mathbb{N} : x_iRx_{i+1}$

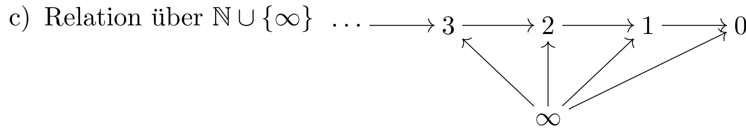
Ist einer Relation eine partielle Ordnung, so heißen unendlich fortgesetzte Wege auch **Ketten**. Eine Relation R heißt **Noethersch**, wenn in R keine unendlich fortgesetzten Wege existieren.

Beispiel: Graph über $\mathbb{N}$

a) $0 \to 1 \to 2 \to 3 \to \dots$
Diese Ordnung ist nicht noethersch

b) $\dots \to 3 \to 2 \to 1 \to 0$
Diese Ordnung ist noethersch

c) Relation über $\mathbb{N} \cup \{\infty\}$



Diese Ordnung ist auch noethersch.

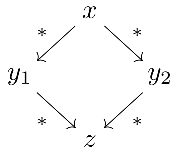
Hüllenbildung

Gegeben Relation $R \subseteq M \times M$

Reflexive Hülle	$R \cup I_M = R^{refl} = \bigcap\{T \subseteq M \times M : R \subseteq T \wedge T \text{ reflexiv}\}$
Transitive Hülle	$R^+ = \bigcap\{T \subseteq M \times M : R \subseteq T \wedge T \text{ transitiv}\}$
Reflexiv-transitive Hülle	$(R \cup I_M)^+ = R^* = \bigcap\{T \subseteq M \times M : R \subseteq T \wedge T \text{ transitiv und reflexiv}\}$
Symetrische Hülle	$R \cup R^T = R^{sym} = \bigcap\{T \subseteq M \times M : R \subseteq T \wedge T \text{ symetrisch}\}$
Symetrisch transitiv reflexive Hülle	$R^{\otimes} = \bigcap\{T \subseteq M \times M : R \subseteq T \wedge T \text{ symetrisch, transitiv und reflexiv}\}$

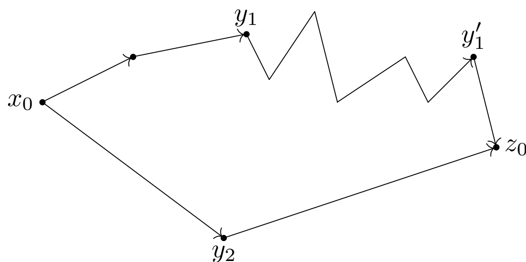
Zur Notation: Relation $\to$, wir schreiben
$\xrightarrow{*}$ für transitive, reflexive Hülle
$\xleftrightarrow{*}$ für reflexiv, symetrisch, transitive Hülle

Eine Relation $\to$ heißt konfluent, wenn gilt: $\forall x, y_1, y_2 \in M : (x \xrightarrow{*} y_1 \wedge x \xrightarrow{*} y_2 \Rightarrow \exists z \in M : y_1 \xrightarrow{*} z \wedge y_2 \xrightarrow{*} z)$



Relationenalgebra: $R^{T*} \circ R^* \subseteq R^* \circ R^{T*}$

Sei R konfluent und noethersch, dann gilt: Jeder Weg (jede Berechnung) ausgehend von x endet in einem Punkt z.



Beispiele: siehe später!

1.2 Grammatiken

Formale Sprachen sind in der Regel unendlich und können deshalb nicht durch einfaches Aufzählen beschrieben werden. Deshalb brauchen wir Regeln (endliches Regelsystem), die die Wörter einer Sprache charakterisieren: Wir verwenden Textersetzungsregeln

Zeichenmenge V
$(x, y) \in V^* \times V^* \qquad x \to y$

$\rightarrow \subseteq V^* \times V^*$ Relation
$\rightarrow$ ist eine endliche Menge von Textersetzungsregeln

Wir gewinnen aus $\rightarrow$ eine in der Regel unendliche Relation $\twoheadrightarrow \subseteq V^* \times V^*$

durch Regelanwendung ($\alpha, \omega \in V^*$)

$x \rightarrow y \quad \Rightarrow \quad \alpha \circ x \circ \omega \twoheadrightarrow \alpha \circ y \circ \omega$

$\twoheadrightarrow$ algebraische Hülle, Halbgruppenhülle

$\omega_1 \overset{*}{\twoheadrightarrow} \omega_2$ $\qquad$ ω_1 wird in ω_2 durch eine Folge von Ersetzungsschritten überführt.

$(V^*, \twoheadrightarrow)$ Semi-Thue-System; wenn $\rightarrow$ symetrisch, dann auch Thue-System.

Beispiel: $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

(1) Beispiel: $\mathtt{ba} \rightarrow \mathtt{ab}$
$\mathtt{cb} \rightarrow \mathtt{bc}$
$\mathtt{ca} \rightarrow \mathtt{ac}$
Relation ist noethersch und konfluent!
Jedes Wort wird durch eine terminierende Berechnung auf die Form $\mathtt{a}^i\mathtt{b}^k\mathtt{c}^j$ gebracht

(2) Beispiel: Zusätzliche Regeln zu (1):
$\mathtt{aa} \rightarrow \mathtt{a}$
$\mathtt{bb} \rightarrow \mathtt{b}$
$\mathtt{cc} \rightarrow \mathtt{c}$
Sechs mögliche Ergebnisse: wie oben, aber i, j und $k \in \{0, 1\}$

(3) Beispiel: Zusätzliche Regeln zu (1):
$\mathtt{aa} \rightarrow \mathtt{a}$
$\mathtt{bc} \rightarrow \mathtt{a}$
$\mathtt{ab} \rightarrow \mathtt{b}$
$\mathtt{ac} \rightarrow \mathtt{c}$
Äquivalenzklassen isomorph zu ganzen Zahlen.

Für jedes Semi-Thue-System $(V^*, \twoheadrightarrow)$ erhalten wir durch $\overset{*}{\twoheadleftarrow\!\!\twoheadrightarrow}$ eine Äquivalenzrelation.

$[w]$ bezeichnet die Äquivalenzklasse:

$[w] = \{w' \in V^* : w \overset{*}{\twoheadleftarrow\!\!\twoheadrightarrow} w'\}$

$V^*/\overset{*}{\twoheadleftarrow\!\!\twoheadrightarrow}$ bilden mit Konkatenation als Verknüpfung eine Halbgruppe mit neutralem Element $[\varepsilon]$, genannt Regelgrammatikmonoid.

17.4.2003

1.2.1 Reduktive und generative Grammatik

Eine Grammatik über einer endlichen Zeichenmenge M ist ein Tripel $(M, \rightarrow, Z)$, wobei
$\rightarrow \in M^* \times M^*$, $\quad$ endlich
$Z \in M$ $\quad$ heißt Axiom (oder Wurzel)

$G = (M, \rightarrow, Z)$ heißt auch Semi-Thue-Grammatik. Grammatiken beschreiben formale Sprachen.

Generativ: Ein Wort $w \in M^*$ ist in der durch G beschriebenen formalen Sprache, wenn gilt:

$$<Z> \overset{*}{\twoheadrightarrow} w$$

Generativer Sprachschatz: $L_g(G) = \{w \in M^* : <Z> \overset{*}{\twoheadrightarrow} w\}$

Reduktiv: Ein Wort $w \in M^*$ ist in der durch G beschriebenen formalen Sprache, wenn gilt:

$$w \overset{*}{\twoheadrightarrow} <Z>$$

Reduktiver Sprachschatz: $L_r(G) = \{w \in M^* : w \overset{*}{\twoheadrightarrow} <Z>\}$

Sprechweise: $w \in L_g(G)$: w wird durch G generiert (erzeugt)
$w \in L_r(G)$: w wird durch G erkannt

Eine generative Grammatik heißt ε-frei oder ε-produktionsfrei, wenn die rechte Seite jeder Regel $\neq \varepsilon$.
Eine reduktive Grammatik heißt ε-frei oder ε-produktionsfrei, wenn die linke Seite jeder Regel $\neq \varepsilon$.

Semi-Thue-Grammatiken haben enge Grenzen in der Ausdrucksmächtigkeit.

Beispiel: $M = \{L\}$ Sprache $S \subseteq M^*, S = \{L^{2^k} : k \in \mathbb{N}\}$

Behauptung: Diese Sprache ist durch Semi-Thue-Grammatik nicht beschreibbar.

Beweis: Jede Regel hat die Form $L^i \to L^j$ Wir betrachten den reduktiven Fall.
Wir benötigen mindestens eine Regel der Form $L^i \to L$ mit $i > 1$
L muß Axiom sein
$L^i L^{i-1} \twoheadrightarrow L^i \twoheadrightarrow L; \quad L^i L^{i-1} \notin S$

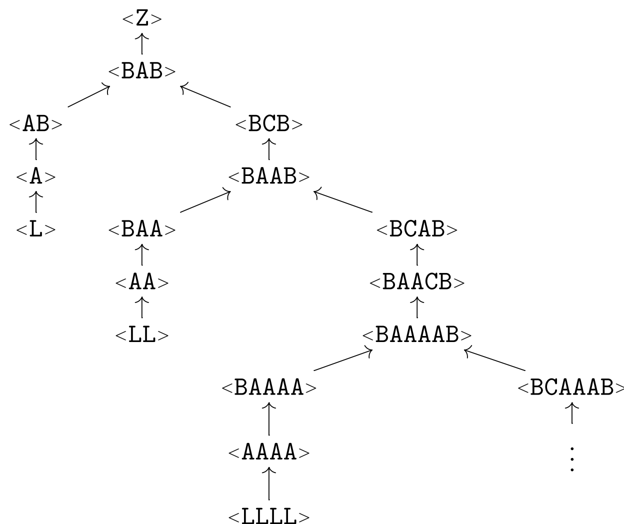
Chomsky-Grammatiken strukturieren den Zeichensatz $M = T \cup N$ mit $T \cap N = \emptyset$
T Terminalzeichen – Zeichen in Worten der formalen Sprache
N Nichtterminalzeichen – Hilfszeichen für die Ableitung

Chomsky-Grammatik $G = (T, N, \to, Z)$ mit

- $(T \cup N, \to, Z)$ Semi-Thue-Grammatik
- $Z \in N$
- $T \cap N = \emptyset$
- $L_g(G) = L_g(T \cup N, \to, Z) \cap T^*$
- $L_r(G) = L_r(T \cup N, \to, Z) \cap T^*$

Beispiel: Reduktive Chomsky-Grammatik zur Beschreibung der Sprache $S = \{L^{2^k} : k \in \mathbb{N}\}$:

- $T = \{\mathtt{L}\}$
- $N = \{\mathtt{Z}, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$
- Z Axiom
- Regeln:
  $\mathtt{L} \to \mathtt{A}$
  $\varepsilon \to \mathtt{B}$
  $\mathtt{AAB} \to \mathtt{CB}$
  $\mathtt{AAC} \to \mathtt{CA}$
  $\mathtt{BC} \to \mathtt{BA}$
  $\mathtt{BAB} \to \mathtt{Z}$



Wir bretrachten den Sprachschatz der Semi-Thue-Grammatik, die wir erhalten, indem wir die ersten zwei Regeln weglassen. <BAB> ist im Sprachschatz.

Beobachtungen:

(1) Jedes Wort im Sprachschatz hat die Form <B> $\circ\, x \circ$ <B>, $x \in \{\mathtt{AC}\}^*$

(2) In Rückwärtsableitung entstehen Cs immer am linken Rand, um die Cs wieder zu entfernen, müssen sie nach rechts geschoben werden und verdoppeln dabei die Anzahl der As.

$\Rightarrow$ d.h. die Wörter aus $\{\mathtt{B}, \mathtt{A}\}^*$ im Sprachschatz haben die Form $\mathtt{BA}^{2^k}\mathtt{B}$

Zwei reduktive (bzw. zwei generative) Grammatiken heißen **äquivalent**, wenn sie die gleiche formale Sprache beschreiben. Eine reduktive Grammatik heißt **wortlängenmonoton**, wenn für jede ihrer Regeln gilt: $w \to w' \;\Rightarrow\; |w| \geq |w'|$. Gilt $|w| > |w'|$, so heißt die Grammatik **strikt wortlängenmonoton**.

Konsequenz für Abb.: $w_0 \twoheadrightarrow w_1 \twoheadrightarrow w_2 \twoheadrightarrow \ldots \twoheadrightarrow w_n \quad \Rightarrow \quad |w_0| \geq |w_1| \geq |w_2| \geq \ldots \geq |w_n|$

Eine Ersetzungsregel $w \to w'$ heißt **separiert**, wenn $w' \in \mathbb{N}^+$.

1.2.2 Die Sprachhierarchie nach Chomsky

Chomsky-Grammatiken lassen sich nach der äußeren Form ihrer Regeln klassifizieren. **Chomsky-0-Sprachen** sind formale Sprachen, die durch Chomsky-Grammatiken beschrieben werden können. Wir beschränken uns im Folgenden auf reduktive Grammatik (generative analog):

Eine Ersetzungsregel der Form $u \circ a \circ v \to u \circ <b> \circ v$ mit $u, v \in (T \cup U)^*, a \in (T \cup N)^+, b \in N$ heißt **kontextsensitiv**. Eine Chomsky-Grammatik heißt **kontextsensitiv** oder **ε-produktionsfreie Chomsky-1-Grammatik**, wenn alle ihrer Regeln kontextsensitiv sind. Trivialerweise gilt für alle Chomsky-1-Grammtiken: Alle ihrer Regeln sind wortlängenmonoton. Eine formale Sprache heißt kontextsensitiv oder Chomsky-1-Sprache, wenn es eine Chomsky-1-Grammatik gibt, die die Sprache beschreibt. Ist eine Sprache S kontextsensitiv, dann wird $S \cup \{\varepsilon\}$ auch kontextsensitiv genannt.

Eine Regel $a \to <b>$ mit $a \in (N \cup T)^*, b \in N$ heißt **kontextfrei**. Sind alle Regeln einer Grammatik kontextfrei, so heißt die Grammatik **Chomsky-2-Grammatik** oder kontextfrei. Jede Sprache, die durch eine Chomsky-2-Grammatik beschreibbar ist, heißt kontextfrei.

Hinweis: BNF entspricht Chomsky-2-Grammatik

Eine kontextfreie Regel der Form $m \circ <a> \circ n \to <b>$ mit $a, b \in N, m, n \in T^+$ heißt **beidseitig linear**. Gilt jedoch $n = \varepsilon$, so heißt die Regel **rechtslinear**. Gilt $m = \varepsilon$, so heißt die Regel **linkslinear**.

Eine Regel der Form $w \to <b>$ mit $w \in T^+, b \in N$ heißt **terminal**. Eine Chomsky-2-Grammatik, deren sämtliche Regeln terminal oder rechtslinear (bzw. terminal oder linkslinear) sind, heißt **Chomsky-3-Grammatik** oder **regulär**. Eine Sprache heißt regulär, wenn sie durch eine reguläre Grammatik beschrieben werden kann (oder auch Chomsky-3-Sprache).

$\mathrm{C_0L}$	Menge der Chomsky-0-Sprachen
CSL	Menge der Chomsky-1-Sprachen
CFL	Menge der Chomsky-2-Sprachen
REG	Menge der Chomsky-3-Sprachen

Satz: $\mathrm{C_0L} \supsetneq \mathrm{CSL} \supsetneq \mathrm{CFL} \supsetneq \mathrm{REG}$

1.2.3 Strukturelle Äquivalenz von Ableitungen

Ableitung über eine Grammatik
$w_0 \twoheadrightarrow w_1 \twoheadrightarrow \ldots \twoheadrightarrow w_n$

Fragen von Interesse:

(1) Was ist ableitbar? $\overset{*}{\twoheadrightarrow}$

(2) Welche Struktur hat eine Ableitung?

<BAAAAB> $\to$ <BAACB> $\to$ <BCAB> $\to$ <BAAB> $\to$ <BCB> $\to$ <BAB> $\to Z$

24.4.2003

Die Anwendung von Regeln $t_1 \to t_1'$ und $t_2 \to t_2'$ in einem Wort $(\alpha, \beta, \gamma \in (T \cup N)^*)$

$$\alpha \circ t_1 \circ \beta \circ t_2 \circ \gamma \twoheadrightarrow \alpha \circ t_1' \circ \beta \circ t_2 \circ \gamma \twoheadrightarrow \alpha \circ t_1' \circ \beta \circ t_2' \circ \gamma$$

$$\alpha \circ t_1 \circ \beta \circ t_2 \circ \gamma \twoheadrightarrow \alpha \circ t_1 \circ \beta \circ t_2' \circ \gamma \twoheadrightarrow \alpha \circ t_1' \circ \beta \circ t_2' \circ \gamma$$

$\alpha \circ t_1 \circ \beta \circ t_2 \circ \gamma \twoheadrightarrow [\alpha \circ t_1' \circ \beta \circ t_2 \circ \gamma \twoheadrightarrow] \alpha \circ t_1 \circ \beta \circ t_2' \circ \gamma \twoheadrightarrow \alpha \circ t_1' \circ \beta \circ t_2' \circ \gamma$

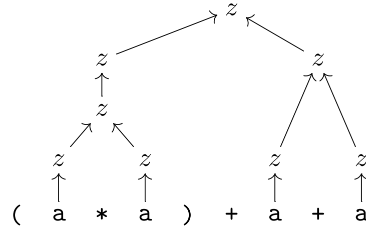
heißt **vertauschbar**.

Bemerkung: Formal ist jede Ableitung ein Wort aus $(T \cup N \cup \{\to\})^*$. Vertauschbarkeit definiert eine Relation auf der Menge der Ableitungen. Wir können zu dieser Relation die transitive, symetrische, reflexive Hülle bilden. Stehen zwei Ableitungen in dieser Relation, dann heißen sie „strukturell äquivalent“.

Chomsky-2-Grammatiken (kontextfreie) können wir aufgrund der einfachen Struktur ihrer Regeln jeder Ableitung auf das Axiom einen Strukturbaum zuordnen.

Beispiel:
$N = \{z\}$, $T = \{$`(`, `)`, `+`, `*`, `a`$\}$, Axiom z

Regeln:
z`+`$z \to z$
z`*`$z \to z$
`a` $\to z$
`(`z`)` $\to z$



Wichtig: Zwei Ableitungen sind genau dann strukturell äquivalent, wenn sie den gleichen Strukturbaum darstellen. Strukturbäume sind also die Normalform in der Klasse der strukturell äquivalenten Ableitungen.

Ein Struikturbaum für eine Ch-2-Grammatik ist ein endlich verzweigter Baum:

(1) Die Wurzel ist das Axiom

(2) Jeder Teilbaum hat folgende Eigenschaft:

  (a) Er ist genau ein Terminalzeichen und hat keine Teilbäume

  (b) Er hat als Wurzel ein Nicht-Terminalzeichen; dann gilt: die Wurzeln seiner Teilbäume können zu dem Wort w konkateniert werden und es gilt $w \to x$ ist eine Regel der Grammatik.

Eine Grammatik heißt **eindeutig**, wenn für jedes Wort $w \in T^*$ mit $w \overset{*}{\twoheadrightarrow} z$ (z Wurzel) alle Ableitungen auf z strukturell äquivalent sind. Dann existiert genau ein Strukturbaum für jedes im Sprachschatz.

Feststellung: Die Grammatik aus obigem Beispiel ist nicht eindeutig.

1.2.4 Sackgassen, unendliche Ableitungen

Ch-2-Grammatiken werden praktisch eingesetzt, um

(1) eine formale Sprache zu definieren

(2) Jedem Wort im Sprachschatz einen Strukturbaum (am besten eindeutig) zuzuordnen.

Ein Wort w heißt „im Sprachschatz“ bzw. reduzierbar oder auch von der Grammatik akzeptiert, wenn gilt $w \overset{*}{\twoheadrightarrow} z$

Aufgaben für eine gegebene Grammatik:

(1) Stelle fest, ob ein gegebenes Wort im Sprachschatz ist

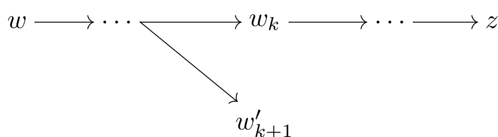
(2) Ermittle (falls es im Sprachschatz ist) den Strukturbaum

Naive Idee: Wende auf das gegebene Wort Regeln an. Wir erhalten eine Ableitung $w \to w_1 \to w_2 \to w_3 \to \dots$
Wir erhalten zwei Möglichkeiten

(1) Die Ableitung führt zu einem Wort, auf das keine Regel mehr anwendbar ist.

(2) Die Ableitung bricht nicht ab und kann unendlich fortgesetzt werden.

Falls (1) gilt und das Wort, auf das keine Regel mehr anwendbar ist, das Axiom ist, so ist w im Sprachschatz. Falls (1) gilt und das Wort, auf das keine Regel mehr anwendbar ist, nicht das Axiom ist, dann können wir daraus nicht schließen, daß das Wort nicht im Sprachschatz ist. Wir sprechen von einer Sackgasse.



Eine unendliche Ableitung (Fall (2)) sagt auch nichts darüber aus, ob das Wort im Sprachschatz ist.

Beispiel: Einfache arithmetische Ausdrücke als Ch-2-Grammatik

$T = \{a, \dots, z, +, *, (,)\}$
$N = \{R, E\}$
Wurzel E

$a \to R$
$\vdots$
$z \to R$
$R * R \to R$
$E + R \to R$
$R \to E$
$(E) \to R$

Neue Grammatik

$T = \{a, \dots, z, +, *, (,)\}$
$N = \{F, R, A, E\}$
Wurzel E

$a \to F$
$\vdots$
$z \to F$
$F * R \to R$
$R + A \to A$
$F \to R$
$R \to A$
$A \to E$
$(E) \to F$

Diese Grammatik ist eindeutig, aber wieder nicht sackgassenfrei.

In der Menge aller Ableitungen für ein Wort über einer Grammatik können wir eine sog. **Linksableitung** (analog **Rechtsabletung**) auszeichnen. Wir erhalten eine Linksableitung. Dies ist eine Ableitung, in der jeweils soweit links wie möglich eine Regel angewendet wird.

Achtung:

(1) In der Regel sind Linksableitungen nicht eindeutig

(2) In der Menge der strukturell äquivalenten Ableitungen sind Linksableitungen eindeutig

Eine Linksableitung führt für ein Wort im Sprachschatz nicht unbedingt zum Axiom.

Akzeptierte Linksableitung: Regeln werden soweit links wie möglich angewendet, Regeln, die nicht zum Axiom führen (in Sackgassen führen), werden übersprungen.

25.4.2003

1.3 Chomsky-3-Sprachen, endlich Automaten, reguläre Ausdrücke

Wir betrachten drei Beschreibungsmittel für formale Sprachen:

- reguläre Ausdrücke
- endliche Automaten
- Chomsky-3-Grammatiken

Wir werden zeigen: Alle drei Beschreibungsmittel haben die gleiche Beschreibungsmächtigkeit und beschreiben die gleiche Klasse formaler Sprachen, genannt **reguläre Sprachen**.

1.3.1 Reguläre Ausdrücke

Gegeben Zeichenmenge T. Die Syntax und Semantik der regulären Ausdrücke (RA) ist wie folgt gegeben:

(1) Einfache reguläre Ausdrücke
x mit $x \in T$ ist RA mit Sprache $\{<x>\}$
ε ist RA mit Sprache $\{\varepsilon\}$
$\{\}$ ist RA mit Sprache $\emptyset$

(2) Zusammengesetzte RA (Seien X und Y RA)
   $[X|Y]$ ist RA mit Sprache $L_X \cup L_Y$
   $[XY]$ ist RA mit Sprache $\{x \circ y : x \in L_X \wedge y \in L_Y\}$
   X^* ist RA mit Sprache $\{x_1 \circ \ldots \circ x_n : n \in \mathbb{N} \wedge x_1, \ldots, x_n \in L_X\}$

wobei L_X (L_Y) die formale Sprache zu RA X (Y) sei.

$X^+ = XX^*$

Gesetze der RA:

- $[[X|Y]|Z] = [X|[Y|Z]]$
- $[[XY]Z] = [X[YZ]]$
- $[X|Y] = [Y|X]$
- $[[X|Y]Z] = [[XZ]|[YZ]]$
- $[Z[X|Y]] = [[ZX]|[ZY]]$
- $\{\}^* = \varepsilon$
- $X\varepsilon = X$
- $x\{\} = \{\}$
- $X^* = XX^* \backslash \varepsilon$
- $X^* = [X|\varepsilon]^*$

1.3.2 Endliche Automaten

Endlicher Automat $A = (S, T, s_0, S_Z, \delta)$

S	endliche Menge von Zuständen
T	endliche Menge von Eingangszeichen
$s_0 \in S$	Anfangszustand
$S_Z \subseteq S$	Menge der Endzustände
$\delta : S \times (T \cup \{\varepsilon\}) \to \delta(S)$	Übergangsfunktion

$\delta(S)$ = Potenzmenge von S

$\delta(s, t)$ ist die Menge der Nachfolgezustände zu $s \in S, t \in T \cup \{\varepsilon\}$

Deterministischer Automat:	$\delta(s, \varepsilon) \subseteq \{s\}$ $|\delta(s,t)| \leq 1$ für alle $s \in S, t \in T$
Partieller Automat:	$\exists t \in T, s \in S : \delta(s,t) = \emptyset$
ε-freier Automat:	$\delta(s, \varepsilon) \subseteq \{s\}$ für alle $s \in S$

Notation: Wir schreiben $s_1 \xrightarrow{a} s_2$ falls $s_2 \in \delta(s_1, a)$

Jedem Wort $w \in T^*$ ordnen wir eine Relation $\xrightarrow{w}^*$ auf den Zuständen zu:

$$\xrightarrow{\varepsilon}^* = (\xrightarrow{\varepsilon})^*$$
$$\xrightarrow{<a>}^* = \xrightarrow{\varepsilon}^* \circ \xrightarrow{a} \circ \xrightarrow{\varepsilon}^*$$
$$\xrightarrow{w_1 \circ w_2}^* = \xrightarrow{w_1}^* \circ \xrightarrow{w_2}^* \qquad w_1, w_2 \in T^*$$

Wir schreiben auch $\delta^*(s, w)$ für $\{s' : s \xrightarrow{w}^* s'\}$

Damit können wir die Sprache charaterisieren, die durch einen Automaten A beschrieben wird.

$L(A) = \{w \in T^* : \exists z \in S_Z : s_0 \xrightarrow{w}^* z\}$

1.3.3 Äquivalenz der Darstellungsform

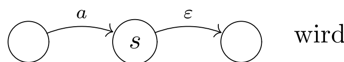
Zunächst beantworten wir die Frage, ob Automaten mit spontanen Übergängen beschreibungsmächtiger sind als Automaten ohne spontane Übergänge.

Satz: Zu jedem endlichen Automaten existiert ein ε-freier endlicher Automat, der die gleiche Sprache akzeptiert.

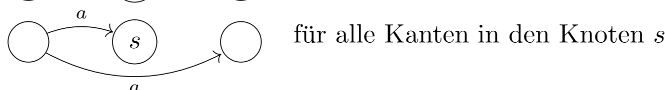
Beweisidee: Wir konstruieren zu einem gegebenen EA (endlichen Automaten) einen ε-freien EA mit gleicher Sprache:

(1) $\xrightarrow{\varepsilon}^*$ - Zyklen: Wir fassen Knoten, die durch $\xrightarrow{\varepsilon}^*$ - Zyklen verbunden sind, zu einem Knoten zusammen. Es entsteht ein Automat ohne $\xrightarrow{\varepsilon}^*$ - Zyklen (ε-Schlingen können weggelassen werden)

(2) $\xrightarrow{\varepsilon}^*$ - Wege (zyklenfrei): Durchschalten von ε-Übergängen. Aus



wird



für alle Kanten in den Knoten s

Satz: Zu jedem ε-freien nichtdeterministischen Automaten existiert ein (ε-freier) deterministischer Automat mit gleichem Sprachschatz.

Beweisidee: Wir konstruieren zu dem Automaten mit Zustandsmenge S einen Automaten mit Zustandsmenge $\delta(S)$

1.3.4 Äquivalenz RA, EA, Chomsky-3-Grammatiken

Satz: Für jeden Automaten existiert ein regulärer Ausdruck, der die gleiche Sprache beschreibt.

Beweis: Konstruktiv! Wir geben zu einem gegebenen EA einen RA mit gleicher Sprache an.
Zustandsmenge: $S = \{s_1, \ldots, s_n\}$
o.B.d.A.: Sei EA ε-frei.
Wir konstruieren eine Familie formaler Sprachen $L(i,j,k) \subseteq T^*$ wobei $i,j \in \{1,\ldots,n\}; k \in \{0,\ldots,n\}$ und die dazugehörige RA $E(i,j,k)$ die jeweils die formale Sprache $L(i,j,k)$ beschreiben.
Dabei sei $L(i,j,k) = \{w \in T^* : s_i \xrightarrow[k]{w} s_j\}$ Dabei betrachten wir in $\xrightarrow[k]{w}$ nur Pfade in Übergangsgraphen mit Zwischenzuständen s_l mit $l \leq k$.
$L(i,j,0) = \{<a> : s_j \in \delta(s_i, a)\}$
$E(i,j,0) = a_1|\ldots|a_q$ entspricht Menge aller $a \in T$ mit $s_i \xrightarrow{a} s_j$
$L(i,j,k+1) = L(i,j,k) \ \cup \ \{u \circ v \circ w : \ u \in L(i,k+1,k) \ \wedge \ v \in L(k+1,k+1,k)^* \ \wedge \ w \in L(k+1,j,k) \ \}$
$E(i,j,k+1) = E(i,j,k)|E(i,k+1,k)E(k+1,k+1,k)^*E(k+1,j,k)$
Die Sprache des EA sind die Wege von s_1 (Anfangszustand) zu Endzuständen $S_Z = \{s_{Z_1}, .., s_{Z_n}\}$
Dies entspricht dem RA $E(1,Z_1,n)|\ldots|E(1,Z_p,n)$

2.5.2003

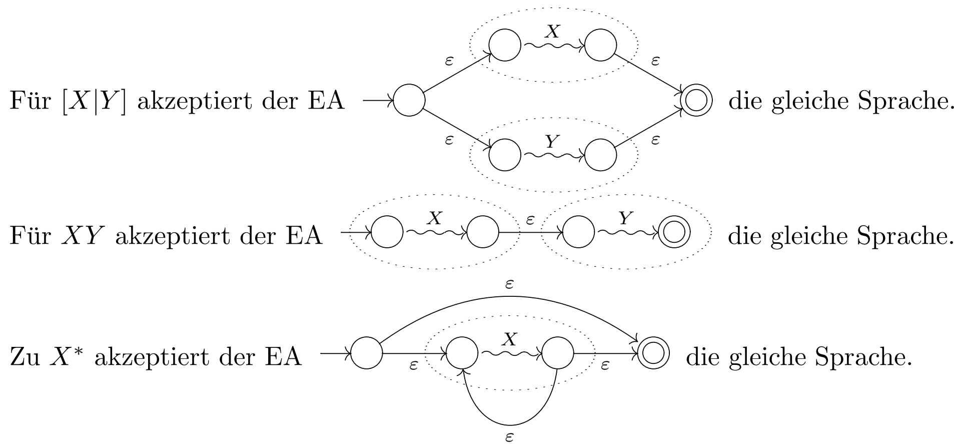
Satz: Zu jedem RA läßt sich ein EA angeben, der die gleiche Sprache beschreibt.

Beweis: Wir geben die Konstruktionsprinzipien an.

(1) Für elementare RA:

Sei $x \in T$, dann ist x RA mit einer Sprache, die durch den EA beschrieben wird.
ε ist RA mit EA:
Automat für leere Sprache:

(2) Seien X und Y RA, die durch die EA und beschrieben werden.



Die Überlegungen zeigen: Für jeden RA können wir einen endlichen Automaten angeben, der die gleiche formale Sprache beschreibt – vorausgesetzt, wir finden bei zusammengesetzten RA EA für die Unterausdrücke. Vollständige Induktion über die Anzahl der Symbole in einem RA liefert die Behauptung.

Satz: Zu jedem ε-freien EA können wir eine Chomsky-3-Grammatik angeben, die die gleiche Sprache beschreibt.

Beweis: Gegeben EA $A = (S, T, s_0, S_z, \delta)$
o.B.d.A. gehe keine Kante nach s_0. Jede Kante (d.h. jeden Übergang $\delta(s, a) = s'$) ordnen wir eine Regel der Grammatik zu. Wir verwenden die Zustände als Nichtterminalzeichen.
Für Übergänge vom Anfangszustand s_0 aus: $\delta(s_0, a) = s_i$ verwenden wir die Regel $a \to s_i$ Für Übergänge $\delta(s_i, a) = s_i \quad (s_i \xrightarrow{a} s_i)$ führen wir die Regel $s_i a \to s_i$ ein. Zu dem Zuständen als Nichtterminalzeichen nehmen wir auch noch das Sonderzichen, die „Wurzel" Z hinzu (o.B.d.A. sei $Z \notin S$). Für jeden Zustand $s_i \in S_z$ nehmen wir die Regel $s_i \to Z$ hinzu. Die entstehende Grammatik akzeptiert die gleiche Sprache.

Satz: Zu jeder Chomsky-3-Grammatik existiert ein endlicher Automat, der die gleiche Sprache akzeptiert.

Beweis: o.B.d.A. sei die Chomsky-3-Grammatik linkslinear und alle Regeln von der Form $sa \to s'$ mit $a \in T$.
Wir geben einen endlichen Automaten an:

Zustände	$S = N \cup \{s_0\}$
Eingangszeichen	T
Anfangszustand	s_0
Endzustände	$\{Z\}$
Übergangsfunktion	$\delta(s, a) = \{s' \in S : sa \to s' \text{ Regel}\}$ für $s \in N, a \in T$
	$\delta(s, \varepsilon) = \{s' \in S : s \to s' \text{ Regel}\}$ für $s \in N$
	$\delta(s_0, a) = \{s' \in S : a \to s' \text{ Regel}\}$ für $a \in T$

Es entsteht ein Automat, der die gleiche Sprache wie die Grammatik beschreibt: Jede Ableitung in der Grammatik entspricht einer Folge von Zustandsübergängen im Automat und umgekehrt.

Fazit:

(1) Die Beschreibungsmittel EA, RA und Chomsky-3-Grammatiken sind gleich mächtig und beschreiben die gleiche Klasse formaler Sprachen.

(2) Die Übergänge sind konstruktiv: Für jeden EA (RA, Chomsky-3-Grammatik) können wir systematisch (durch einen Algorithmus) einen RA (Chomsky-3-Grammatik, EA) angeben, der die gleiche formale Sprache beschreibt. Solche Sprachen heißen regulär.

Wir wenden uns nun der Frage zu, wie wir für eine vorgegebene formale Sprache erkennen können, daß sie regulär ist. Wir beginnen mit einem einfachen Beispiel für eine nichtreguläre Sprache $L_{ab} = \{a^n b^n : n \in \mathbb{N}\backslash\{0\}\}$
Diese Sprache können wir ohne Probleme durch eine Chomsky-2-Grammatik beschreiben:

$$\begin{aligned} T &= \{a, b\} \\ N &= \{Z\} \\ Z &\text{ Wurzel} \\ \text{Regeln:} \quad ab &\to Z \\ aZb &\to Z \end{aligned}$$

Behauptung: Die Sprache L_{ab} ist nicht regulär.

Beweis: Wir zeigen, daß kein EA existiert, der L_{ab} akzeptiert.
Annahme: Ein solcher Automat exisitert:

(1) Der endliche Automat hat nur endlich viele Zustände (sei k die Anzahl der Zustände).

(2) $a^n b^n, a^m b^m$ sind im Sprachschatz, d.h. es existieren die Zustände s_n, s_m, s_z
$s_0 \overset{a^n}{\longrightarrow}{}^* s_n \wedge s_n \overset{b^n}{\longrightarrow}{}^* s_z$
$s_0 \overset{a^m}{\longrightarrow}{}^* s_m \wedge s_m \overset{b^m}{\longrightarrow}{}^* s_z$

Falls $s_n = s_m$ gilt, werden auch Wörter $a^n b^m$ und $a^m b^n$ akzeptiert. Da nur endlich viele Zustände (genau k Zustände) exisiteren, muß es Zahlen n und m geben mit $n \neq m$ und $s_n = s_m$.

Problem bei EA: EA können nur eine endliche Information speichern, können nicht unbeschränkt zählen. Sprachen mit Klammerstrukturen (beliebiges Verschachteln von sich öffnenden und schließenden Klammern) sind nicht regulär.

Satz: Pumping Lemma für reguläre Sprachen
Zu jeder Regulären Sprache L existiert eine Zahl $n \in \mathbb{N}$, sodaß für alle Wörter $w \in L$ mit $|w| \geq n$ gilt:
w läßt sich in Wörter $x, y, z \in T^*$ zerlegen: $w = x \circ y \circ z$ mit $|y| \geq 1, |x \circ y| \leq n, x \circ y^i \circ z \in L$ für alle $i \in \mathbb{N} \backslash \{0\}$

Beweis: L regulär! Es existiert ein deterinistischer, endlicher ε-freier Automat, der L akzeptiert. Sei n die Anzahl seiner Zustände, $w \in L$.
Es gilt $s_0 \overset{w}{\longrightarrow} s_\varepsilon$ Dabei durchläuft der Automat $|w| + 1$ Zustände.
Gilt $|w| \geq n$, so tritt ein Zustand mindestens zweimal auf, d.h. $s_0 \overset{x}{\to} s_k \wedge s_k \overset{y}{\to} s_k \wedge s_k \overset{z}{\to} s_z$ dann gilt auch $s_0 \overset{x}{\to} s_k \wedge s_k \overset{y^i}{\longrightarrow} s_k \wedge s_k \overset{z}{\to} s_z$.

Damit läßt sich zeigen: L_{ab} ist nicht regulär.

1.3.5 Minimale Automaten

Wir zeigen nun, wie wir für eine reguläre Sprache einen minimalen Automaten (kleinste Anzahl Zustände) angeben können. Sei $L \subseteq T^*$: Wir definieren eine Äquivalenzrelation $\sim_L$ auf T^* durch (sei $v, w \in L^*$):
$v \sim_L w \Leftrightarrow_{def} (\forall u \in T^* : v \circ u \in L \Leftrightarrow w \circ u \in L)$
Dadurch erhalten wir Äquivalenzklassen $[v]_L = \{w \in T^* : w \sim_L v\}$

Satz (Myhill-Nerode): L ist genau dann regulär, wenn die Menge der Äquivalenzklassen endlich ist.

Beweisidee: Jede Äquivalenzklasse definiert einen Zustand in einem endlichen Automaten, der die Sprache akzeptiert. Umgekehrt: Jeder Zustand in einem endlichen Automaten entspricht einer Äquivalenzklasse.
Die Beweisidee liefert bereits einen Hinweis, wie wir einen ε-freien, deterministischen, endlichen Automaten mit totaler Übergangsfunktion konstruieren können, der die Sprache L akzeptiert, und eine minimale Anzahl von Zuständen hat.

Wir zeigen nun, wie wir aus einem gegebenen endlichen ε-freien, deterministischen Automaten einen minimalen Automaten konstruieren, indem wir bestimmte Zustände zusammenlegen.

1. Schritt Wir entfernen alle Zustände, die vom Anfangszustand aus nicht erreichbar sind.

2. Schritt Wir konstruieren eine Folge von Prädikaten auf Zuständen $p_i : S' \times S' \to \mathbb{B}$
$p_i(s_1, s_2) : \quad \forall w \in T^* : |w| \leq i \Rightarrow (\exists z \in S_z : s_1 \overset{w}{\twoheadrightarrow}{}^* z) \Leftrightarrow (\exists z \in S_z : s_2 \overset{w}{\twoheadrightarrow}{}^* z)$
p_i läßt sich induktiv definieren:
$p_0(s_1, s_2) = (s_1 \in S_z \Leftrightarrow s_2 \in S_z)$
$p_{i+1}(s_1, s_2) = p_i(s_1, s_2) \wedge \forall x \in T : p_i(\delta(s_1, x), \delta(s_2, x))$

Da der Automat endlich ist, existiert ein k mit $p_k = p_{k+1}$. Das Prädikat p_k definiert mir dann, welche Zustände der Automaten äquivalent sind und zu einem Zustand (im minimalen Automaten) verschmolzen werden können.

8.5.2003

Anwendung: grep, sed, vi/vim, emacs, flex/lex

1.4 Kontextfreie Sprachen (KfS) und Kellerautomaten (KA)

Chomsky-3-Grammatiken genügen i.A. nicht, um komplexe, formale Sprachen wie Programmiersprachen zu beschreiben (z.B. Klammerstrukturen wie in $L_{ab} = \{a^n b^n : n \in \mathbb{N}\}$). Es gilt: REG $\subsetneq$ CFL, d.h. die Klasse der KfS ist mächtiger als die der regulären Sprachen. Chomsky-3-Grammatiken, reguläre Ausdrücke und endliche Automaten eignen sich nicht, um KfS zu beschreiben. Analog zu RA, EA und Chomsky-3-Grammatiken bei regulären Sprachen sind in der Klasse KfS folgende Beschreibungsformen gleich mächtig:

- Chomsky-2-Grammatiken
- BNF-Ausdrücke
- Kellerautomaten

Diese Mittel werden bei Spezifikation und der Syntax-Analyse von Programmiersprachen in Übersetzern und Interpretern eingesetzt.

1.4.1 BNF-Notation

Erweiterung von RA um:

1. Hilfszeichen (Nichtterminale)
2. (rekursive) Gleichungen für Hilfszeichen

Sei T eine Menge von Terminalzeichen und N eine Menge von Nichtterminalzeichen. Eine BNF-Beschreibung einer Sprache L hat die Form:

$$\begin{array}{lcl} x_1 & ::= & E_1 \\ & \vdots & \\ x_n & ::= & E_n \end{array}$$

mit $x_1, \ldots, x_n \in N$ und $E_1, \ldots, E_n$ RA über $T \cup N$.

Beispiel: $<Z> ::= ab|a<Z>b$ beschreibt die KfS $\{a^n b^n : N \in \mathbb{N}\ 0\}$

Für eine BNF-Beschreibung kann für jedes x_i $(1 \leq i \leq n)$ eine Chomsky-2-Grammatik (KfG) angegeben werden. Die RA E_i werden in Chomsky-3-Grammatiken (mit neuem Nichtterminal Z_i als Axiom) umgeformt und für die BNF-Regel $x_i ::= E_i$ wird die Regel $Z_i \to x_i$ in die Grammatik aufgenommen.

Anmerkung: Es gibt zahlreiche Stile und Erweiterungen von BNF. Die Klammern $[\ldots]$ dienen oft der Kennzeichnung optionaler Anteile, z.B. $X[Y]$ für $X|XY$. $\{X\}_n^m$ bezeichnet die n- bis m-malige Wiederholung von X.

1.4.2 Kellerautomaten

Ein Kellerautomat $KA = (S, T, K, \delta, s_o, k_o, S_z)$

- endliche Menge S von Zuständen
- endliche Menge T von Eingabezeichen
- endliche Menge K von Kellerzeichen
- endliche Übergangsrelation $\delta : S \times (T \cup \{\varepsilon\}) \times K \to P(S \times K^*)$
- Anfangszustand $s_0 \in S$
- Endzustände $S_z \subseteq S$

Prinzip: KA verarbeitet ein $w \in T^*$, indem er w von links nach rechts liest und in jedem Schritt, abhängig vom aktuellen Zustand, Eingabezeichen a und oberstem Kellerzeichen k, entsprechend δ:

- in einen neuen Zustand $s' \in S$ übergeht
- das oberste Zeichen k im Keller entfernt
- eine (ggf. leere) Sequenz $u \in K^*$ auf den Keller legt

KA akzeptiert $w \in T^*$, wenn die vollständige Verarbeitung von w von s_0 zu einem Endzustand $s \in S_z$ führt.

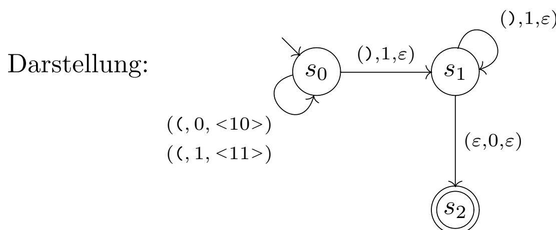
Darstellung: Graph analog zu EA. Knoten: $s \in S$, Kanten: $(a, k, u) \in (T \cup \{\varepsilon\} \times K \times K^*)$ Eine Kante (a, k, u) von s nach s' existiert genau dann, wenn $(s', u) \in \delta(s, a, k)$

Beispiel: KA für $\{ (^n)^n : n \in \mathbb{N}\ \{0\}\}$

$$\begin{aligned} KA &= (S, T, K, \delta, s_0, 0, \{s_2\}) \\ S &= \{s_0, s_1, s_2\} \\ T &= \{ (,) \} \\ K &= \{1, 0\} \end{aligned}$$

$$\begin{aligned} \delta(s_0, (, 0) &= \{(s_0, <10>)\} \\ \delta(s_0, (, 1) &= \{(s_0, <11>)\} \\ \delta(s_0,), 1) &= \{(s_1, \varepsilon)\} \\ \delta(s_1,), 1) &= \{(s_1, \varepsilon)\} \\ \delta(s_1, \varepsilon, 0) &= \{(s_2, \varepsilon)\} \end{aligned}$$

Darstellung:



Arbeitsweise formal:
Wir betrachten **Kellerkonfigurationen** $(s, w, v) \in S \times T^* \times K^*$ mit Kontrollzustand s, (Rest-)Eingabewort w und Kellerzustand v (oberstes Zeichen im Keller). Ein KA induziert eine Übergangsrelation $\rightarrow$ auf Konfigurationen, vermöge:
$(s_1, w, <k> \circ v) \rightarrow (s_2, w, u \circ v)$, falls $(s_2, u) \in \delta(s_1, \varepsilon, k)$ und
$(s_1, <a> \circ w, <k> \circ v) \rightarrow (s_2, w, u \circ v)$, falls $(s_2, u) \in \delta(s_1, a, k)$

$w \in T^*$ wird vom KA akzeptiert, wenn es ein $s \in S_z$ und ein $v \in K^*$ gibt, so das $(s_0, w, <k_0>) \xrightarrow{*} (s, \varepsilon, v)$. Die vom KA akzeptierte Sprache bezeichnen wir mit L(KA).

Anmerkung: Akzeptieren durch leeren Keller ist gleich mächtig, d.h. $w \in T^*$ vom KA akzeptiert, wenn $(s_0, w, \varepsilon) \xrightarrow{*} (s, \varepsilon, \varepsilon)$ mit $s \in S$.
Durch KA erhalten wir für beliebige KfS unmittelbar einen (i.d.R. sehr ineffizienten) Erkennungsalgorithmus.
Im Gegensatz zu EA sind deterministische Kellerautomaten nicht äquivalent zu nichtdeterministischen KA.

1.4.3 Äquivalenz von Kellerautomaten und KfS

Satz: Ist L eine kontextfreie Sprache, dann gibt es einen Kellerautomaten K, so das $L(K) = L$

Idee: Gegeben sei eine KfG $G = (T, N, \rightarrow, Z)$
Gesucht: KA K mit $L(K) = L(G)$
$K = (S, T, T \cup N \cup \{\#\}, \delta, \varepsilon, \#, \{s_e\})$
$S = \{\varepsilon, s_e, s_v\} \cup$ Menge aller (Teil-)Sequenzen der linken Seiten der Grammatik-Regeln.
Sei $<e_1 \dots e_n>, e_i \in N \cup T$, linke Seite einer Regel $<e_1 \dots e_n> \rightarrow A$.
Übergangsrelation δ:

1. Kellern: $(\varepsilon, <ak>) \in \delta(\varepsilon, a, k), a \in T$
2. Regelerkennung:
   a) $(<e_{i-1} \dots e_j>, \varepsilon) \in \delta(<e_i \dots e_j>, \varepsilon, e_{i-1})$
   b) $(<e_i \dots e_{j+1}>, <k>) \in \delta(<e_i \dots e_j>, e_{i+1}, k)$

3. Regelanwendung: $(\varepsilon, <Ak>) \in \delta(<e_1 \dots e_n>, \varepsilon, k)$

4. Akzeptanz:

   a) $(s_v, \varepsilon) \in \delta(\varepsilon, \varepsilon, Z)$
   b) $(s_e, \varepsilon) \in \delta(s_v, \varepsilon, \#)$

K ist nichtdeterministisch und ineffizient. K kellert Eingabe, bis er an beliebiger Stelle mit dem Aufbau der linken Seite einer Regel beginnt.

Beispiel:
KfG $G = (\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{Z}\}, \{\mathtt{aZb} \to \mathtt{Z}, \mathtt{ZZ} \to \mathtt{Z}, \mathtt{ab} \to \mathtt{Z}\})$
$S = \{\varepsilon, s_v, s_e, <\mathtt{a}>, <\mathtt{b}>, <\mathtt{Z}>, <\mathtt{aZ}>, <\mathtt{Zb}>, <\mathtt{ZZ}>, <\mathtt{aZb}>, <\mathtt{ab}>\}$
δ: (sei $x \in T, k \in K$) $\delta(\varepsilon, x, k) = \{(<x>, <k>), (\varepsilon, <xk>)\}$
$k \neq \# \Rightarrow \delta(\varepsilon, \varepsilon, k) = \{(<k>, \varepsilon)\}$
$\delta(<\mathtt{Z}>, \mathtt{b}, k) = \{(<\mathtt{Zb}>, <k>), (<\mathtt{ZZ}>, <k>)\}$
$\delta(\varepsilon, \varepsilon, \mathtt{Z}) = \{(s_v, \varepsilon)\}$
$\vdots$

Ausschnitt aus Ableitungsbaum für $<\mathtt{aabb}>$:
$(\varepsilon, aabb, \#) \stackrel{1}{\to} (\varepsilon, abb, a\#) \stackrel{1}{\to} (\varepsilon, bb, aa\#) \stackrel{1}{\to} (\varepsilon, b, baa\#) \stackrel{1}{\to} (\varepsilon, \varepsilon, bbaa\#) \stackrel{2a}{\to} (b, \varepsilon, baa\#)$ Sackgasse!
Alternative: $(\varepsilon, aabb, \#) \stackrel{1}{\to} (\varepsilon, abb, a\#) \stackrel{2b}{\to} (a, bb, a\#) \stackrel{2b}{\to} (ab, b, a\#) \stackrel{3}{\to} (\varepsilon, b, Za\#) \stackrel{2a2a2b}{\to} (aZb, \varepsilon, \#) \stackrel{3}{\to} (\varepsilon, \varepsilon, Z\#) \stackrel{4a4b}{\to} (s_e, \varepsilon, \#)$

Beobachtung: k nichtdeterministisch und Sackgassen, d.h. sehr ineffizient.

9.5.2003

Satz: Zu jedem KA K (mit $\varepsilon \neq L(K)$) existiert eine kontextfreie Grammatik G mit $L(K) = L(G)$.
Beweis: s. Literatur (Hopcroft/Ullman)

Vorgehensweise bei der Reduktion von Wörter KfS durch KA:

1. Top-Down: Keller initial Axiom. Ableitung eines (Teil-)Wortes $v \in T^*$ im Keller und Vergleich mit (Rest-)Eingabe. Bei Übereinstimmung Kürzung des Eingabeworts und Entfernen von v im Keller.

2. Bottom-Up: Keller initial leer. Schrittweise Lesen / Kellern der Eingabe, Reduktion von Teilwörtern auf Keller. Erfolgreich, wenn Z im Keller und Eingabe leer.

Beide Verfahren i.A. nichtdeterministisch und wegen Suche im Ableitungsbaum ineffizient.

Verbesserungsansatz: Eliminieren des Nichtdeterminismus durch Beschränkung der Regelauswahl; Erfolg abhängig von der KfS.

1.4.4 Greibach-Normalform

KfG $G = (T, N, \to, Z)$ ist in Greibach-Normalform, wenn jede Ersetzungsregel folgende Gestalt hat:
$<a> \circ w \to x$ mit $a \in T, w \in N^*, x \in N$
d.h. Verarbeitung genau eines Terminals $a \in T$ links bei jeder Ersetzung.

Satz: Jeder von einer ε-freien KfG erzeugte Sprachschatz kann auch durch eine KfG in GNF erzeugt werden.
Beweis: Literatur.

Satz: Zu jeder KfG G existiert ein KA mit $L(K) = L(G)$

Beweis: $G = (T, N, \to, Z)$ sei o.B.d.A. in GNF
$K = (\{s_0\}, T, N, \delta, s_0, Z, s_0)$
$\delta(s_0, a, k) = \{(s_0, w) : <a> \circ w \to k\} \quad \forall k \in N$
Es gilt für $v \in T^*, x \in N^* : v \stackrel{*}{\twoheadrightarrow} x \Leftrightarrow (s_0, v, x) \stackrel{*}{\to} (s_0, \varepsilon, \varepsilon)$
Es genügt eine einelementige Zustandsmenge.

Beispiel: KfG $G = (\{a, b\}, \{Z, U\}, \{aU \to Z, aZU \to Z, b \to U\}, Z)$
$K = (\{s_0\}, \{a, b\}, \{Z, U\}, \delta, s_0, Z, \{s_0\})$

$\delta(s_0, a, Z) = \{(s_0, <U>), (s_0, <ZU>)\}$ (Regeln 1a und 1b)
$\delta(s_0, b, U) = \{(s_0, \varepsilon)\}$ (Regel 2, s. folgendes Beispiel)

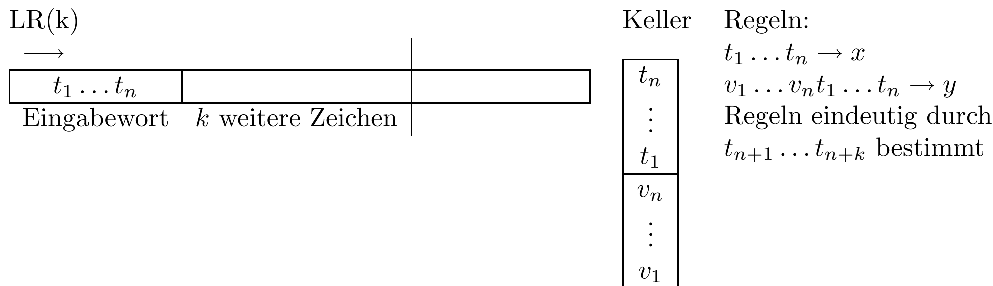
Ausschnitt Reduktionsbaum $<aabb>$:
$(s_0, <aabb>, <Z>) \xrightarrow{1a} (s_0, <abb>, <U>) \rightarrow$ Sackgasse!
$(s_0, <aabb>, <Z>) \xrightarrow{1b} (s_0, <abb>, <ZU>) \xrightarrow{1a} (s_0, <bb>, <UU>) \xrightarrow{2} (s_0, <b>, <U>) \xrightarrow{2} (s_0, \varepsilon, \varepsilon)$.

D.h. stark vereinfachter KA, aber nach wie vor nichtdeterministisch. Benötigt aus praktischer Sicht: eingeschränkte KfS, die beschränkten Nichtdeterminismus des Kellerautomaten verursachen und Sackgassen eliminieren.

1.4.5 LR(k)-Sprachen (LR = Left-Rightmost)

... oder: L → Eingabe von links nach rechts lesen, R → Rechtsableitung, k → k Zeichen Vorschau (Lookahead)

Idee: Bottom-Up. Eingabewort von links nach rechts lesen und dabei Kellern, bis Regel anwendbar ist. Bestimmung der anzuwendenden Regel anhand von höchstens k Zeichen des Restwortes (bzw. rechts der Anwendungsstelle); Rechtskontext.



Beispiel: $t_1 \dots t_n \rightarrow x$; streiche $t_1 \dots t_n$ aus Keller und Eingabewort, lege x auf den Keller.

In einer LR(k)-Sprache kann bei jeder akzeptierenden Linksreduktion durch Betrachtung der k nächsten Zeichen rechts der Anwendungsstelle die anzuwendende Regel eindeutig bestimmt werden.

Bei bestimmten Grammatiken kann die Auswahl einer Regel der Form $<e_1 \dots e_n> \rightarrow A$ für $w = x \circ <e_1 \dots e_n> \circ y$ durch Betrachtung eines endlichen Präfix von y gesteuert werden.

Kontextbedingung (KB): Menge von Wörtern, die die Präfixe von y festlegt, welche die Regelanwendung zulassen.

KfG G heißt **LR-deterministisch**, wenn es für alle Regeln der Grammatik endliche KB gibt, so das der von links nach rechts arbeitende KA für alle Wörter $w \in L(G)$ und alle akzeptierende Reduktionen in jedem Schritt genau eine Regel zuläßt, die Redutkion vollzieht und Sackgassen vermeidet.

Satz: Jede LR-deterministische KfG ist eindeutig.
Beweis: Gemäß Definition von LR-deterministischen KfG gibt es eine eindeutige Einschränkung des Kellerautomaten, der die Ableitung erzeugt.

Definition (LR(k)):
$w[i:k]$ bezeichnet Teilwort $<w_i \dots w_k>$ von $w = <w_1 \dots w_n>, i, k \in \mathbb{N}, 1 \leq i$.
Sei $k \in \mathbb{N}$. Eine azyklische KfG $(T, N, \rightarrow, Z)$ heißt **LR(K)-Grammatik**, wenn für jedes Paar von Ableitungen in Links-Normalform:
$u_1 \circ a_1 \circ x_1 \twoheadrightarrow u_1 \circ <B_1> \circ x_1 \twoheadrightarrow \dots \twoheadrightarrow Z$
$u_2 \circ a_2 \circ x_2 \twoheadrightarrow u_2 \circ <B_2> \circ x_2 \twoheadrightarrow \dots \twoheadrightarrow Z$
wobei $\{a_1 \rightarrow B_1, a_2 \rightarrow B_2\} \subseteq \rightarrow$, gilt:
$u_1 \circ a_1 \circ (x_1[1:k]) = u_2 \circ a_2 \circ (x_2[1:k]) \Rightarrow a_1 = a_2 \wedge B_1 = B_2 \wedge u_1 = u_2$
D.h. durch $x_1[1:k]$ ist die anzuwendende Regel eindeutig festgelegt.

Beispiel (LR(0)):
$G = (\{a, b\}, \{Z, X\}, \{ZX \rightarrow Z, X \rightarrow Z, aZb \rightarrow X, ab \rightarrow X\}, Z)$
Ableitung von $<aabbab>$: $<aabbab> \xrightarrow{4} <aXbab> \xrightarrow{2} <aZbab> \xrightarrow{3} <Xab> \xrightarrow{2} <Zab> \xrightarrow{4} <ZX> \xrightarrow{1} <Z>$.

KA: Kellern bis Regel anwendbar, Regeln anwenden so lange wie möglich:

Keller	Eingabe
ε	aabbab
a	abbab
aa	bbab
aab	bab
aX	bab
aZ	bab
⋮	⋮
Z	

Keller: Präfix des zu reduzierenden Wortes.
Bei LR(0) ist kein Kontext notwendig, um die richtigen Regeln zu wählen.

Satz: Jede LR(k)-Grammatik ist eindeutig. *Beweis:* LR(k)-Grammatiken sind LR-deterministisch.

Beispiel (LR(1)):
$G = (T, N, \rightarrow, Z)$
$N = \{Z, A, P, E\}$
$T = \{(,), +, -, *, /, a\}$

Regel	KB
$A \rightarrow Z$	ε
$E \rightarrow A$	+, -,), ε
$A + E \rightarrow A$	+, -,), ε
$A - E \rightarrow A$	+, -,), ε
$P \rightarrow E$	$t \in T$
$E * P \rightarrow E$	$t \in T$
$E/P \rightarrow E$	$t \in T$
$(A) \rightarrow P$	$t \in T$
$a \rightarrow P$	$t \in T$

Bsp.: $<a + a * a> \twoheadrightarrow \ldots \twoheadrightarrow <A + E * a> \twoheadrightarrow <A + E * P> \twoheadrightarrow <A + E> \twoheadrightarrow \ldots \twoheadrightarrow Z$

LR(k)-Grammatiken erlauben Reduktion von Wörtern durch Kellerautomaten mit akzeptablem Aufwand. Insbesondere LR(1)-Grammatiken werden in der Praxis eingesetzt.

Anmerkung: Es gibt keinen Algorithmus, der für eine beliebige Grammatik G entscheidet, ob G LR(k).

1.4.6 LL(k)-Grammatiken (Left-leftmost)

Gegensatz zu LR(k): top-down-Produktion der Ableitung, ausgehend vom Axiom Z. Dadurch Linksableitung (bzw. Rechtsreduktion) statt Linksreduktion.

Idee (KA):

1) Produktion von $v \in T^*$ ausgehend vom Nichtterminal auf Keller und abhängig vom Rechtskontext.

2) Vergleich des erzeugten v mit Präfix des (Rest-)Eingabewortes

3) Bei Übereinstimmung: Entfernen von v aus Keller und Eingabewort.

Definition LL(k):
Eine azyklische KfG $G = (T, N, \rightarrow, Z)$ heißt LL(k)-Grammatik ($k \in \mathbb{N}$), falls für alle Paare von Ableitungen in Rechtsnormalform ($a_1, a_2, v, u_1, u_2 \in (T \cup N)^*, w \in T^*, B \in N$):
$v \circ u_1 \overset{*}{\twoheadrightarrow} v \circ a_1 \circ w \overset{*}{\twoheadrightarrow} v \circ <B> \circ w \overset{*}{\twoheadrightarrow} Z$
$v \circ u_2 \overset{*}{\twoheadrightarrow} v \circ a_2 \circ w \overset{*}{\twoheadrightarrow} v \circ <B> \circ w \overset{*}{\twoheadrightarrow} Z$
wobei $\{a_1 \rightarrow B, a_2 \rightarrow B\} \subseteq \rightarrow$ gilt:
$u_1[1:k] = u_2[1:k] \Rightarrow a_1 = a_2$
D.h. die Zerteilung ist mit $u_1[1:k]$ eindeutig festgelegt.

15.5.2003

Beispiel: LL(1)-Grammatik $G = (\{Z\}, \{a, +, -\}, \{a \rightarrow Z, -Z \rightarrow Z, +ZZ \rightarrow Z\}, Z)$
Reduktion von <`+-a+aa`> durch KA:

LL-Technik (top-down)		LR-Technik (bottom-up)	
Keller	Resteingabe	Keller	Resteingabe
`Z`	`+-a+aa`	ε	`+-a+aa`
`ZZ+`	`+-a+aa`	`+`	`-a+aa`
`ZZ`	`-a+aa`	`+-`	`a+aa`
`ZZ-`	`-a+aa`	`+-a`	`+aa`
`ZZ`	`a+aa`	`+-Z`	`+aa`
		`+Z`	`+aa`
⋮	⋮	⋮	⋮
`a`	`a`		
ε	ε	`Z`	ε

Jede LL(k)-Grammatik ist auch eine LR(k)-Grammatik, d.h. u.a. jede LL(k)-Grammatik ist eindeutig.

1.4.7 Rekursiver Abstieg

Klassisches Verfahren der Programmierung eines top-down-KA zur Zerteilung von Wörtern einer KfS. Orientiert an BNF:
Für jedes Nichtterminal $x_i \in \mathbb{N}$ der linken Seiten der Regeln $x_i ::= E_1|\dots|E_n$ eine Prozedur, die (rekursiv) eine vollständige Falluntersuchung der rechten Seite der zu x_i gehörenden Regel durchführt.
i.d.R. ebenfalls Steuerung über Präfix des Restwortes.

1.4.8 Das Pumping-Lemma für KfS

KfG $G = (T, N, \rightarrow, Z)$ ist in Chomsky-Normalform (CNF), falls alle Regeln folgende Gestalt haben:
$a \rightarrow A$ oder $BC \rightarrow A$, für $a \in T; A, B, C \in N$

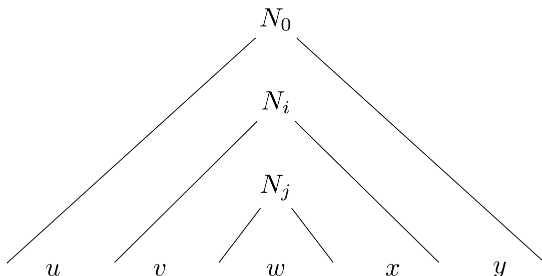
Satz: Zu jeder KfG G mit $\varepsilon \notin L(G)$ gibt es eine äquivalente KfG in CNF.

Satz (Pumping Lemma für KfS): Zu jeder KfS S existiert ein $n \in \mathbb{N}$, sodaß sich alle $z \in L$ mit $|z| \geq n$ in $u, v, w, x, y \in T^*$ zerlegen lassen, $z = u \circ v \circ w \circ x \circ y$ mit:

1. $|v \circ x| \geq 1$
2. $|v \circ w \circ x| \leq n$
3. für alle $i \geq 0 : u \circ v^i \circ w \circ x^i \circ y \in L$

Beweis: Sei $G = (T, N, \rightarrow, N_0)$ CNF-Grammatik mit $L(G) = L, k = |N|, n = 2^k, z \in L$ mit $|z| \geq k$.
Ableitungsbaum T für z ist Binärbaum der Höhe $k+1$ mit $|z| \geq n$ Blättern.
Knoten mit genau einem Sohn ($a \rightarrow A$) oder genau zwei Söhnen ($BC \rightarrow A$)
$h \geq log_2|z| \geq log_2 n = k$
Im Pfad $p_0, \dots, p_{n+1}$ ist T von der Wurzel p_0 zu Blatt p_{h+1} ist die Markierung $N_0, N_1, \dots, N_h$ der Knoten $p_0, \dots, p_h$ eine Folge von $h+1 \geq k+1$ Nichtterminalen. D.h. es gibt $i, j \in \{0, \dots, h\}$, so daß $N_i = N_j = N, i < j$ und $N_{i+1}, \dots, N_h$ paarweise verschieden

Zerlegung von z



1. Da G in CNF ist $v \neq \varepsilon$ oder $x \neq \varepsilon$, d.h. $|vx| \geq 1$

2. Teilbaum T' ab Knoten p_i ist Abletungsbaum für $N_i \overset{*}{\twoheadrightarrow} v \circ w \circ x$
   Aus $N_{i+1}, \ldots, N_h$ verschieden folgt: T' hat Höhe $h - i \leq k$.
   D.h. $|v \circ w \circ x| \leq 2^{h-i} \leq 2^k = n$

3. Ableitungen für $u \circ v^i \circ w \circ x^i \circ y$ ergeben sich aus der Kombination der Möglichkeiten: $N_0 \overset{*}{\twoheadrightarrow} u \circ N \circ y, N \overset{*}{\twoheadrightarrow} w, N \overset{*}{\twoheadrightarrow} v \circ N \circ x$.

1.5 Kontextsensitive Grammatiken (KsG)

es gilt: CFL $\subsetneq$ CSL

KsG sind definitionsgemäß wortlängenmonoton (wlm), d.h. für jeden Schritt $x \twoheadrightarrow y : |y| \leq |x|$

KsG sind sehr allgemein, jede wlm Grammatik ist strukturäquivalent zu einer KsG! Es ist möglich, für jede KsG L einen (ineffizienten) Algorithmus anzugeben, der für $w \in T^*$ entscheidet, ob $w \in L$

Begründung: Bei einer wlm Grammatik ist für jede $w \in T^*$ die Menge der durch Reduktion von w entstehenden Wörter endlich; mögliche unendliche Reduktionen können aufgrund der Wortlängenmonotonität erkannt werden.

Für eine Chomsky-0-Grammatik G_0 existiert i.A. kein solcher Algorithmus! Für G_0 existiert lediglich ein Algorithmus, der für jedes $w \in L(G_0)$ mit Resultat *true* terminiert. Für $w \notin L(G_0)$ kann die Terminierung nicht garantiert werden.

Kapitel 2

Berechenbarkeit

Es gibt mathematisch exakt beschreibbare Probleme, für die es keine Lösungsalgorithmen gibt!

typische Probleme:
Aufgabe: Berechnung einer gegebenen Funktion
Lösung: Algorithmus, der f berechnet.

Definition: f heißt **berechenbar**, wenn es einen Algorithmus gibt, der für jedes Argument x (Eingabe) den Wert $f(x)$ (Ausgabe) berechnet. Nicht berechenbare Funktionen können nicht durch Programme einer Rechenanlage beschrieben werden!

Wir betrachten o.B.d.A. n-stellige $f : \mathbb{N}^n \to \mathbb{N}$, wobei lediglich Repräsentationen von $n \in \mathbb{N}$ zur Verfügung stehen. Wir verwenden die Zeichenmenge T und die injektive, umkehrbare Abbildung $rep : \mathbb{N} \to T^*$. Demnach existiert auch $abs : \{t \in T^* : \exists n \in \mathbb{N} : rep(n) = t\} \to \mathbb{N}$, so daß $abs \circ rep = id$ und $rep \circ abs = id$

Statt abstraktem $f : \mathbb{N} \to \mathbb{N}$ betrachten wir ein konkretes $\bar{f} : (T^*)^n \to T^*$, mit
$f(x_1, \ldots, x_n) = abs(\bar{f}(rep(x_1), \ldots, rep(x_n)))$

Konkrete Algorithmen arbeiten auf Repräsentationen (von $\mathbb{N}$). Für einen realistischen Berechenbarkeitsbegriff müssen die Repräsentationen handhabbar sein. Wir setzen deshalb u.a. voraus, daß T endlich ist.

2.1 Hypothetische Maschinen

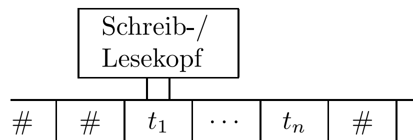
Möglichkeit zur Präzisierung des Algorithmus-Begriffs: hypothetische Maschinen. Mathematische Nachbildung des Zustandsraums und der Übergangsfunktion realer Maschinen mit dem Ziel der Präzisierung und Vereinfachung.

EA und KA genügen nicht, u.a. können KA für KfS aber nicht für KsS L berechnen, ob $w \in L$. Wir betrachten allgemeinere Modelle als KA mit unbeschränkten Mengen von Zuständen.
Mögliche Kritik: unrealistisch, weil technisch nicht realisierbar. . . ?

2.1.1 Turing-Maschine

A.M. Turing, 1936

Beidseitig unendliches Band von Zellen zur Speicherung von Zeichen, ein Schreib-/Lesekopf, Steuereinheit



Ein endlicher Abschnitt des Bandes trägt relevante Informationen, alle anderen Zellen enthalten das Symbol # für die leere Information.

Prinzip: TM liest in jedem Schritt ein Zeichen $t \in T \cup \{\#\}$ unter dem Kopf. Abhängig vom Zustand wird das Zeichen überschrieben, ein neuer Zustand eingenommen und der Kopf um höchstens eine Position nach links

oder rechts bewegt.

Eine Turing-Maschine $TM = (T, S, \delta, s_0)$ umfaßt:

- endliche Menge T von Zeichen, mit denen das Band beschrieben wird (es sei $\# \notin T$)
- endliche Menge S von Zuständen
- endliche Übergangsfunktion bzw. Relation $\delta : S \times (T \cup \{\#\}) \to \delta(S \times (T \cup \{\#\}) \times (\ll, \gg, \downarrow\})$

„$\ll$" bzw. „$\gg$" beschreiben eine Verschiebung des Kopfes um eine Zelle nach links oder rechts und „$\downarrow$" das Beibehalten der Position. Gilt für $TM : \forall t \in T \cup \{\#\}, s \in S : |\delta(s,t)| \leq 1$, so heißt TM **deterministisch**.

Beispiel: Prüfung, ob Anzahl der „L" in $<w_1, \ldots, w_n>, n \geq 1, w_i \in \{\mathtt{L}, \mathtt{0}\}$ gerade.

Zu Beginn sei $\ldots \# w_1 \ldots_n \# \ldots$ auf Band und der Kopf auf w_n positioniert. TM hält mit Bandinhalt $\ldots \#\mathtt{L}\# \ldots$, falls Anzahl der L gerade bzw. mit $\ldots \#\mathtt{0}\# \ldots$ sonst.

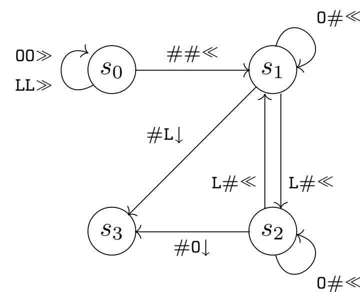
$TM = (\{\mathtt{0}, \mathtt{L}\}, \{s_0, s_1, s_2, s_3\}, \delta, s_0)$

Übergangsfunktion δ:

δ	0	L	#
s_0	$(s_0, \mathtt{0}, \gg)$	$(s_0, \mathtt{L}, \gg)$	$(s_1, \# \ll)$
s_1	$(s_1, \#, \ll)$	$(s_2, \#, \ll)$	$(s_3, \mathtt{L}, \downarrow)$
s_2	$(s_2, \#, \ll)$	$(s_1, \#, \ll)$	$(s_3, \mathtt{0}, \downarrow)$
s_3	$\emptyset$	$\emptyset$	$\emptyset$

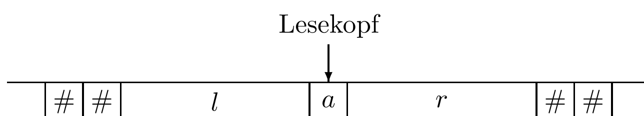
16.5.2003

Das Verhalten einer Turingmaschine kann auch grafisch durch einen Automaten dargestellt werden:



Die Markierung „0#$\ll$" für einen Zustandsübergang von s nach s' bedeutet, daß dieser Übergang im Zustand s ausgeführt werden kann, falls 0 unter dem Lesekopf steht; dann wird # geschrieben und der Lesekopf bewegt sich nach links.

Eine Konfiguration einer Turingmaschine beschreibt den Berechnungszustand. Wir verwenden ein 4-Tupel: $(s, l, a, r) \in S \times (T \cup \{\#\})^* \times (T \cup \{\#\}) \times (T \cup \{\#\})^*$



s = aktueller Zustand

Da das Band nach links und rechts unendlich fortgesetzt ist, sind folgende Konfigurationen äquivalent: $(s, l, a, r) = (s, <\#> \circ l, a, r) = (s, l, a, r \circ <\#>)$

Die Berechnung einer Turingmaschine (ausgehend von einer Konfiguration) besteht aus einer endlichen oder unendlichen Folge von Konfigurationen.

Wir definieren eine Relation auf Konfigurationen $(s, l, a, r) \to (s', l', a', r')$ wie folgt: Es gilt genau eine der folgenden Aussagen:

(0)	$z = \downarrow \wedge l = l' \wedge r = r' \wedge a' = x$	Kopf bleibt
(1)	$z = \ll \wedge l = l' \circ <a'> \wedge r' = <x> \circ r$	Kopf nach links
(2)	$z = \gg \wedge l' = l \circ <x> \wedge r = <a'> \circ r'$	Kopf nach rechts

wobei $(s', x, z) \in \delta(s, a)$

Eine Konfiguration K heißt **terminal**, wenn keine Nachfolgekonfiguration existiert. Die Turingmaschine bleibt stehen, die Berechnung endet. Eine Berechnung heißt **vollständig**, wenn sie unendlich ist oder endlich ist und die letzte Konfiguration terminal ist. Bemerkung: Jede Turingmaschine läßt sich in einer der gebräuchlichen Programmiersprachen (Java, C, Pascal etc.) simulieren.

Wir stützen nun auf Toruingmaschinen den Begriff der Berechenbarkeit ab (genauer: Turing-Berechenbarkeit). Eine partielle Funktion $f : T^* \to T^*$ heißt **turing-berechenbar**, wenn es eine deterministische Turingmaschine gibt, so daß für jedes Wort $t \in T^*$ eine der folgenden Aussagen gilt:

(1) $f(t)$ hat einen Bildpunkt ($f(t)$ ist definiert, genauer der Wert von f für Argument t ist definiert) und es existiert eine vollständige Berechnung: $(s_0, t, \#, \varepsilon) \to \ldots \to (s_e, r, \#, \varepsilon)$ wobei $f(t) = r$.

(2) f ist für Argument t nicht definiert und es existiert eine unendliche Berechnung $(s_0, t, \#, \varepsilon) \to \ldots \to \ldots \to \ldots$

Dieser Berechenbarkeitsbegriff läßt sich auf partielle Abbildungen $f : \mathbb{N}^n \to \mathbb{N}$ übertragen: Dazu müssen wir nur eine Zahldarstellung festlegen. Wir stellen natürliche Zahlen als Strichzahlen dar und verwenden „␣“ als Trennzeichen. f heißt berechenbar, wenn eine Funktion $g : \{1, \sqcup\}^* \to \{1, \sqcup\}^*$ existiert, die berechenbar ist und für alle $x_1, \ldots, x_n \in \mathbb{N}$ gilt: $g(<\sqcup> \circ 1^{x_1} \circ <\sqcup> \circ \ldots \circ <\sqcup>1^{x_n}<\sqcup>) = <\sqcup> \circ 1^y \circ <\sqcup>$ genau dann, wenn $f(x_1, \ldots, x_n) = y$.

Einige Beispiele für turing-berechenbare Funktionen

(1) Konstante
$c : \mathbb{N}^n \to \mathbb{N}$ mit $c(x_1, \ldots, x_n) = k$ für gegebene $k \in \mathbb{N}$

(2) Nachfolgerfunktion
$succ : \mathbb{N} \to \mathbb{N}$ mit $succ(n) = n + 1$

(3) Projektionen ($1 \leq i \leq n$)
$\Pi_i^n : \mathbb{N}^n \to \mathbb{N}$ mit $\Pi_i^n(x_1, \ldots, x_n) = x_i$

(4) Vorgänger
$pred : \mathbb{N} \to \mathbb{N}$ total

$$pred(x) = \begin{cases} x - 1 & \text{falls } x > 0 \\ 0 & \text{sonst} \end{cases}$$

$pred : \mathbb{N} \to \mathbb{N}$ partiell

$$pred(x) = \begin{cases} x - 1 & \text{falls } x > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Komplexe Funktionen können wir durch Funktionskomposition gewinnen. Wir definieren eine verallgemeinerte Komposition für partielle Funktionen: $g : \mathbb{N}^n \to \mathbb{N}, \quad h_i : \mathbb{N}^m \to \mathbb{N}, 1 \leq i \leq n$
Wir definieren $f : \mathbb{N}^m \to \mathbb{N}$ durch $f(x_1, \ldots, x_m) = g(h_1(x_1, \ldots, x_m), \ldots, h_n(x_1, \ldots, x_m))$, falls $h_1, \ldots, h_n$ für $x_1, \ldots, x_n$ definiert und g für $h_1(x_1, \ldots, x_n), \ldots, h_n(x_1, \ldots, x_n)$ definiert, sonst ist f für $(x_1, \ldots, x_n)$ undefiniert.

Satz: f ist turing-berechenbar, wenn $g, h_1, \ldots, h_n$ turing-berechenbar sind.

Notation: Wir schreiben dann für f auch $g \circ [h_1, \ldots, h_n]$

Es gilt: Addition, Multiplikation, Division (d.h. die übliche Arithmetik) ist turing-berechenbar.

Frage: Gibt es Funktionen, die nicht turing-berechenbar sind?
Antwort: Ja, siehe später!

Bemerkung: Ob wir Turingmaschinen mit mehreren Bändern oder nur einseitig unendliche Turingmaschinen verwenden, ändert nichts am Begriff der Turing-Berechenbarkeit. Andere Berechenbarkeitsbegriffe haben sich als äquivalent erwiesen. Dazu folgen zwei Beispiele.

2.1.2 Register-Maschinen

Eine Register-Maschine ist (wie eine Turingmaschine) eine hypothetische Maschine (mathematisch definiert), die eher unseren gebräuchlichen Rechnern entspricht. Eine Register-Maschine mit n Registern (n-RM) besitzt n Register (Speicherplätze) für natürliche Zahlen und ein Programm (n-RM-Programm). Diese Programme haben eine extrem einfache Syntax:

(1) ε ist ein n-RM-Programm (leeres Programm)

(2) $succ_i$ mit $1 \leq i \leq n$ ist ein n-RM-Programm

(3) $pred_i$ mit $1 \leq i \leq n$ ist ein n-RM-Programm

(4) sind M_1 und M_2 n-RM-Programme, so ist $M_1; M_2$ ein n-RM-Programm

(5) ist M ein n-RM-Programm, so ist $while_i(M)$ mit $1 \leq i \leq n$ ein n-RM-Programm

22.5.2003

Semantik von RM durch Zustandsübergangsbeschreibung. Konfiguration einer n-RM: $(s, p) \in \mathbb{N}^n \times n$-Prog

Dabei beschreibt n-Prog die Menge aller Programme für n-RM.

Zustands- bzw. Konfigurationsübergangsfunktion (für n-RM mit $i \in \mathbb{N}, 1 \leq i \leq n$):
$(s, succ_i) \rightarrow ((s_1, \ldots, s_{i-1}, s_i + 1, s_{i+1}, \ldots, s_n), \varepsilon)$
$(s, \text{pred}_i) \rightarrow ((s_1, \ldots, s_{i-1}, k, s_{i+1}, \ldots, s_n), \varepsilon)$ mit $k = \begin{cases} s_i - 1 & \text{falls } s_1 > 0 \\ 0 & \text{sonst} \end{cases}$
$(s, (p_1; p_2)) \rightarrow (s', (p'_1; p_2))$ falls $(s; p_1) \rightarrow (s'; p'_1)$
$(s, (\varepsilon; p_2)) \rightarrow (s; p_2)$
$(s, while_i(p)) \rightarrow \begin{cases} (s, (p; while_i(p))) & \text{falls } s_i > 0 \\ (s, \varepsilon) & \text{sonst} \end{cases}$

Bemerkung: Die Konfiguration der Form (s, ε) sind terminal, d.h. es exisitert keine Nachfolgerfunktion („das Programm terminiert“). Wie gehabt definieren wir Berechnungen (endliche/unendliche) als Folgen von Konfigurationen in der $\rightarrow$-Relation.

Beispiel: RM-Programme

(1) $while_i(pred_i)$ entspricht while $s_i > 0$ do $s_i := s_i - 1$ od und setzt s_i auf Null.
$s_i := k$ entspricht $while_i(pred_i); \underbrace{succ_i; \ldots; succ_i}_{k\text{-mal}}$

(2) Folgende Funktionen können in RM programmiert werden:
- übliche Arithmetik
- übliche Boolesche Algebra (Wahrscheinlichkeitswerte dargestellt durch Zahlen).

Eine Funktion $f : \mathbb{N}^n \rightarrow \mathbb{N}^n$ (partiell!) heißt RM-berechenbar, wenn es ein n-RM-Programm gibt, so dass genau falls $f(s_1, \ldots, s_n) = (s'_1, \ldots, s'_n)$ dann gilt $(s, p) \xrightarrow{*} (s', \varepsilon)$
Auch $g : \mathbb{N}^k \rightarrow \mathbb{N}$ mit $1 \leq k \leq n$ und (für ein $i \in \mathbb{N}, 1 \leq i \leq n$)
$g(x_1, \ldots, x_k) = \Pi_i^n(f(x_1, \ldots, x_k, x_k + 1, \ldots, x_n))$ für gegebene $x_{k+1}, \ldots, x_n$ heißt auch RM-berechenbar.

2.2 Rekursive Funktionen

Die beiden bisher betrachteten Berechnungsmodelle (Turing-Maschinen, Registermaschinen) sind „hypothetische Maschinen“. Jetzt betrachten wir ein stärker durch Beschreibung charakterisiertes Berechnungsmodell, rekursiv definierte Funktionen. Auch dafür können wir Algorithmen zur Auswertung angeben (vgl. Termersetzung).

2.2.1 Primitiv rekursive Funktionen

Wir betrachten n-stellige Funktionen $f : \mathbb{N}^n \rightarrow \mathbb{N}$ (total oder partiell). Wir nennen folgende Funktionen **Grundfunktionen**:

$succ : \mathbb{N} \rightarrow \mathbb{N}$	mit $succ(n) = n + 1$
$zero^{(0)} : \rightarrow \mathbb{N}$	(Konstante Null)
$zero^{(1)} : \mathbb{N} \rightarrow \mathbb{N}$	
$\Pi_i^n : \mathbb{N}^n \rightarrow \mathbb{N}$	i-te Projektion, $1 \leq i \leq n$

Aus einer gegebenen Menge von Funktionen gewinnen wir weitere durch Komposition $g_0[k_1, \dots, k_n]$ oder durch Anwendung des **Schemas der primitiven Rekursion**:
Gegeben seien Funktionen $g : \mathbb{N}^k \to \mathbb{N}$ und $k : \mathbb{N}^{k+2} \to \mathbb{N}$.
Wir definieren $f : \mathbb{N}^{k+1} \to \mathbb{N}$ wie folgt: Für alle $x_1, \dots, x_k, n \in \mathbb{N}$:
$f(x_1, \dots, x_k, 0) = g(x_1, \dots, x_k)$
$f(x_1, \dots, x_k, n+1) = k(x_1, \dots, x_k, n, f(x_1, \dots, x_k, n))$

Dies entspricht einer induktiven Definition von f.

Fakt: Sind g, h totale Funktionen, so ist durch das Schema f eindeutig bestimmt und total.
Beweis: Induktion!
Bemerkung: Sind g und h partiell, so ist f auch eindeutig festgelegt, aber u.U. selbst partiell.

Die Menge der primitiv rekursiven Funktionen (PR) ist induktiv wie folgt definiert:

(1) Die Grundfunktionen sind in PR.

(2) Funktionen, die durch Komposition von Funktionen aus PR gewonnen werden können, sind in PR.

(3) Funktionen, die durch das Schema der primitiven Rekursion über Funktionen $g, h \in$ PR gewonnen werden können, sind in PR.

Wir können, wie für die Komposition, auch für das Schema der primitiven rekursiven Funktionen eine kompakte Notation einführen:
$pr : (\mathbb{N}^k \to \mathbb{N}) \times (\mathbb{N}^{k+2} \to \mathbb{N}) \to (\mathbb{N}^{k+1} \to \mathbb{N})$ mit $f = pr(g, h)$, alle Def. wie oben.

Es gilt:

(1) alle arithmetischen Funktionen (totale Division, totale Subtraktion) sind in PR

(2) alle Funktionen in PR sind total

(3) alle Funktionen in PR sind TM-berechenbar

(4) alle Funktionen in PR sind RM-berechenbar

(5) Da alle Funktionen in PR total sind, existiert trivialerweise eine Funktion, die TM- bzw. RM-berechenbar, aber nicht in PR ist

Wir zeigen nun, daß es eine totale Funktion gibt, die offensichtlich TM/RM-berechenbar ist, aber nicht in PR.
$ack : \mathbb{N}^2 \to \mathbb{N}$

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

Feststellung:

(1) Durch diese Gleichung ist ack eindeutig bestimmt und total.

(2) Wir definieren eine Schar von Funktionen $B_n(m) = ack(n, m)$

Wir erhalten:
$B_0(m) = m + 1$
$B_1(m) = m + 2$
$B_2(m) = 2m + 3$,
$B_3(m) = 2^{m+3} - 3$
Alle Funktionen B_n sind primitiv rekursiv.

B_{n+1} kann durch das Schema der PR aus B_n definiert werden.

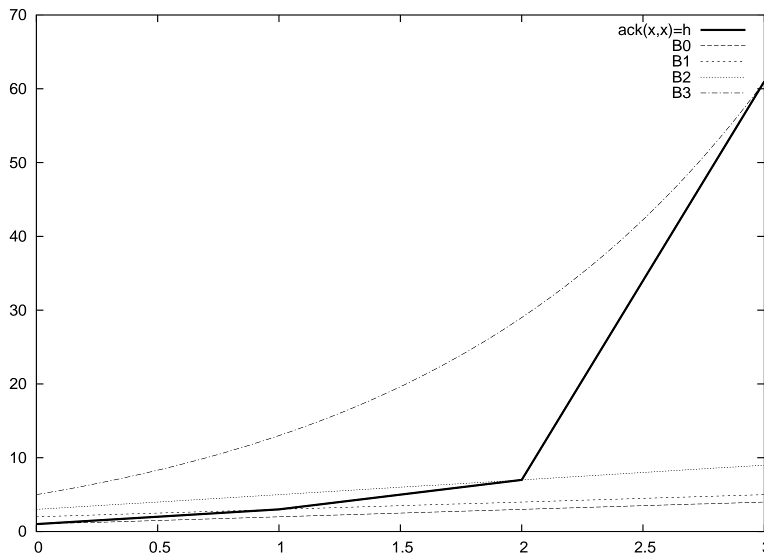
Satz: $ack \notin$ PR

Beweis (Skizze):
Durch Induktion über die Anzahl der Anwendungen des Schemas der primitiven Rekursion können wir zeigen:

Zu jeder Funktion $g \in PR, g : \mathbb{N}^n \to \mathbb{N}$ existiert eine Konstante $c \in \mathbb{N}$, sodas gilt: $g(x_1, \ldots, x_n) < ack(c, \sum_{i=1}^n x_i)$ für alle $x_1, \ldots, x_n \in \mathbb{N}$.

Annahme: $ack \in$ PR. Dann ist $h : \mathbb{N} \to \mathbb{N}$ mit $h(n) = ack(n, n)$ auch in PR.
Es existiert eine Zahl $c \in \mathbb{N}$: $ack(n, n) = h(n) < ack(c, n)$ für alle $n \in \mathbb{N}$.
Mit n=c erhalten wir $ack(c, c) = h(c) < ack(c, c)$↯

In anderen Worten: ack wächst schneller als alle Funktionen in PR:



Fazit:

(1) Es existiert eine totale Funktion, TM-berechenbar/RM-berechenbar, die aber nicht in PR ist.

(2) ack ist offensichtlich durch Rekursion definierbar, also ist der Begriff der PR zu eng.

2.2.2 μ-rekursive Funktionen

Jetzt betrachten wir auch partielle Funktionen, verwenden alle Funktionen aus PR (und alle Konstruktionsprinzipien) und zusätzlich eine weitere Konstruktion, die der μ-Rekursion.

Beispiel: Rekursive Definition $ulam : \mathbb{N} \to \mathbb{N}$

$$ulam(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ ulam(g(n)) & \text{sonst} \end{cases}$$

$$g(n) = \begin{cases} n/2 & \text{falls } n \text{ gerade} \\ 3n + 1 & \text{sonst} \end{cases}$$

23.5.2003

Feststellung: g ist primitiv rekursiv.
Frage: Terminiert $nlam$ für alle Argumente $n \in \mathbb{N}$?
Anwort: Unbekannt!
Vermutung: Antwort ist ja!
Falls $nlam$ immer terminiert, gilt $nlam$(n)=1, d.h. nlam ist dann primitiv rekursiv.

Das Schema der μ-Rekursion:

Gegeben $f : \mathbb{N}^{k+1} \to \mathbb{N}$ partiell

Wir definieren $\mu(f) : \mathbb{N}^k \to \mathbb{N}$ durch $\mu(f)(x_1, \ldots, x_n) = \min\{y \in \mathbb{N} : f(x_1, \ldots, x_n, y) = 0\}$ falls
(*) $\exists y \in \mathbb{N} : f(x_1, \ldots, x_k, n) = 0 \land \forall z \in \mathbb{N} : 0z \leq y \Rightarrow f$ ist def.
für $(x_1, \ldots, x_k, z)$ gilt.

Gilt (*) nicht, dann definieren wir $\mu(f)$ ist für $(x_1, \ldots, x_k)$ nicht definiert.

Achtung: Auch wenn f total ist, kann $\mu(f)$ partiell sein.

Die Menge der μ-rekursiven Funktionen MR definieren wir induktiv wie folgt:

(1) Alle primitiven rekursiven Funktionen sind in MR

(2) Funktionen, die durch Komposition oder das Schema der primitiven Rekursion aus Funktionen in MR gebildet werden, sind in MR.

(3) Falls $f \in$ MR, dann $\mu(f) \in$ MR

Beispiel: μ-rekursive Definition der partiellen Subtraktion:

$$a \dot{-} b = \begin{cases} a - b & \text{falls } a \geq b \\ \textit{undef.} & \text{sonst} \end{cases}$$

Wir definieren

a $\dot{-}$ b $= \mu(h_0)(a, b)$ mit geeignetem $h_0 : \mathbb{N}^3 \to \mathbb{N}$

Wir wählen $h_0(a, b, y) = sub(b + y, a) + sub(a, b + y)$ wobei $sub(a, b) = \begin{cases} a - b & \text{falls } a \geq b \\ 0 & \text{sonst} \end{cases}$

Fälle	sub(b+y, a)+sub(a, b+y)	Fälle
$a > b$	$0 + 0 = 0$	$y = a - b \geq 0$
$a > b$	> 0	$y < a - b, y + b < a$
$b < a$	> 0 keine Nullstelle!	

2.2.3 Allgemeine Bemerkungen zur Rekursion

In der Informatik existieren viele Spielarten von Rekursion zur Definition von Funktionen, Prozeduren, Methoden, Datentypen, Mengen, formalen Sprachen usw. In der rekursiven Definition einer Funktion verwenden wir Gleichungen der Form $f(x) = E$ wobei f in E auftritt oder $f(D) = E$ mit eingeschränkten Ausdrücken D.

Beispiel:
$ack(0, 0) = 1$
$ack(0, m + 1) = ack(0, m) + 1$
$ack(n + 1, 0) = ack(n, 1)$
$ack(n + 1, m + 1) = ack(n, ack(n + 1, m))$

2.3 Äquivalenz der Berechenbarkeitsbegriffe

Wir haben vier Begriffe von Berechenbarkeit eingeführt: Turingmaschinen, Registermaschinen, primitive rekursive FUnktionen und μ-rekursiven Funktionen. PR ist schwächer als TM, RM, MR. Wir zeigen nun die Äquivalenz von TM, RM, MR.

2.3.1 Äquivalenz von μ-Berechenbarkeit und TM-Berechenbarkeit

Der Logiker Kurt Gödel hatte eine Idee zur Darstellung von Zeichenfolgen durch Zahlen entwickelt → Gödelisierung (1933). Eine Gödelisierung ist eine Abbildung $f : A^* \to \mathbb{N}$ wobei A eine endliche Menge von Zeichen sei, mit folgenden Eigenschaften:

(1) f ist injektiv

(2) f ist berechenbar (TM-berechenbar)

(3) es ist berechenbar, ob eine Zahl $n \in \mathbb{N}$ im Bildberech von f liegt (d.h. $\exists w \in A^* : f(w) = n$)

(4) es existiert ein Algorithmus, der zu jeder Zahl im Bildbereich das zugehörige Wort berechnet – „f ist algorithmisch umkehrbar“.

Satz: TM $\cong$ MR

Beweis:

(1) $f \in \text{MR} \Rightarrow f \in \text{TM}$

Beweisidee: für jede Grundfunktion in MR (bzw. PR) können wir eine TM angeben. Das Schema der Komposition, der PR und der μ-Rekursion können wir durch TM „nachbauen".

(2) $f \in \text{TM} \Rightarrow f \in \text{MR}$

Konfigurationen von TM entsprechen Wörtern aus A^* (mit geeignetem A). Wir verwenden eine Gödelisierung: $rep : A^* \to \mathbb{N}$ (einfach zu konstruieren).

Es zeigt sich, daß die Funktion $g : \mathbb{N} \to \mathbb{N}$ mit $k_0 \to k_1 \Leftrightarrow g(rep(k_0)) = rep(k_1)$ primitiv rekursiv ist (d.h. die Nachfolgerfunktion auf Konfigurationen entspricht einer primitiv rekursiven Funktion auf den Darstellungen der Konfigurationen durch natürliche Zahlen).

Ferner definieren wir eine primitiv rekursive Funktion $h : \mathbb{N}^2 \to \mathbb{N}$ durch

$$h(n,m) = \begin{cases} 0 & \text{falls } g^m(n) \text{ einer terminalen Konfiguration entspricht} \\ 1 & \text{sonst} \end{cases}$$

Die Funktion $it : \mathbb{N}^2 \to \mathbb{N}$ sei spezifiziert durch
$it(n,0) = n$
$it(n,m+1) = it(g(n),m)$
it beschreibt die Ausführung von m Schritten der TM

Die Funktion $tm : \mathbb{N} \to \mathbb{N}$ sei definiert durch

$$tm(n) = it(n, \underbrace{\mu(h)(n)}_{\substack{\text{Anzahl der Schritte der TM bis zur Terminierung für Eingabekonfiguration} \\ \text{dargestellt durch } n \text{ (falls TM terminiert)}}})$$

2.3.2 Äquivalenz RM/TM-Berechenbarkeit

Jede RM läßt sich in eine TM übersetzen. Dazu brauchen wir eine Darstellung der Konfigurationen der Registermaschine durch eine Turingmaschine:

(1) Die Registerinhalte werden auf dem TM-Band dargestellt (z.B. Strichzahlen)

(2) Aus RM-Programmen konstruieren wir den endlichen Automaten, der die TM steuert.

Zu einer gegebenen TM kann man eine RM konstruieren, die eine TM simuliert.
Idee: Konfigurationen der TM werden gödelisiert und in den Registern dargestellt. DIe Fahrbewegungen und Zustandsübergänge der TM werden durch Programmschritte der RM simuliert.

2.3.3 Church's These

Alonzo Church (1936): Jede intuitiv berechenbare Funktion ist μ-rekursiv

30.5.2003

2.4 Entscheidbarkeit

Wir haben eine Reihe von Berechenbarkeitsbegriffen kennengelernt, die jeweils auf das gleiche Konzept von „berechenbaren Funktionen" führen (Church'sche These). Wir wenden uns nun der Frage zu, welche Funktionen berechenbar / nicht berechenbar und welche Prädikate entscheidbar, d.h. durch Algorithmen eindeutig mit ja / nein beantwortbar sind.

2.4.1 Nichtberechenbare Funktionen

Jeder Formalismus zur Berechenbarkeit (Turing-Maschinen, Registermaschinen, μ-Rekursionen) definiert Programme / Algorithmen durch Wörter über einem Zeichensatz. D.h. in jedem Fall existiert eine formale Sprache $PRG \subseteq A^*$ mit geeignetem Zeichensatz A^*, s.d.g. jeder Algorithmus wird dargestellt durch ein Wort $p \in PRG$.

Für jede berechenbare Funktion f existiert ein $p \in PRG$, das f berechnet.
Die Menge der Funktionen $\mathbb{N}^n \to \mathbb{N}$ ist über-abzählbar! (Ergebnis der Mengenlehre)
$\mathbb{N}$ ist abzählbar. A^* ist abzählbar (falls A endlich ist). PRG ist abzählbar.

Dies ergibt sofort: Es gibt viel mehr Funktionen $\mathbb{N} \to \mathbb{N}$ als Programme in PRG, d.h. fast jede Funktion ist nicht berechenbar. Wir können Programme durch Zahlen codieren, d.h. es existiert eine Gödelisierung $code : PRG \to \mathbb{N}$

Satz: Es existiert eine totale Funktion $g : \mathbb{N} \to \mathbb{N}$, die nicht berechenbar ist.

Beweis: Wir spezifizieren g durch (für alle $n \in \mathbb{N}$):

$$g(n) = \begin{cases} f_p(n) + 1 & \text{falls } \exists p \in PRG : code(p) = n \text{ und } f_p \text{ definiert für Arg.} \\ 0 & sonst \end{cases}$$

Dabei sei für $p \in PRG$ die Funktion $f_p : \mathbb{N} \to \mathbb{N}$, die durch p berechnete partielle Funktion.
Achtung: Die Spezifikation von g ist eindeutig und konsistent, da *code* eine injektive Funktion ist.
Annahme: g ist berechenbar!
Dann existiert ein Programm $q \in PRG$ mit $f_q = g$. Mit $code(q) = m$ gilt: $g(m) = f_q(m) + 1 = g(m) + 1$ ↯

Bemerkung: g ist zunächst nur von theoretischen Interesse.

2.4.2 (Nicht)Entscheidbare Prädikate

Eine ja/nein-Frage entspricht logisch einem Prädikat. Ein Prädikat

$$p : M \to \mathbb{B}$$

kann auch als Funktion $p' : M \to \{0, 1\} \subseteq \mathbb{N}$ aufgefaßt werden. Damit können wir das Konzept der Berechenbarkeit auf Prädikate übertragen. Ein „berechenbares“ totales Prädikat heißt auch entscheidbar. Ein Prädikat $p : M \to \mathbb{B}$ heißt

(1) entscheidbar, wenn es einen Algorithmus gibt, der für jedes Argument $m \in M$ terminiert und korrekt 1 für $p(m) = true$ und 0 für $p(m) = false$ ausgibt.

(2) positiv semientscheidbar, wenn es einen Algorithmus gibt, der für jedes Argument $m \in M$ terminiert und korrekt 1 ausgibt, falls $p(m) = true$, und falls $p(m) = flase$ gilt entweder 0 ausgibt oder nicht terminiert.

(3) negativ semientscheidbar, falls $\neg p$ positiv semientscheidbar ist.

Beispiele:

(1) Entscheidbare Probleme
  - Gleichheit zweier Zahlen in Binärdarstellung
  - Primzahleigenschaft
  - Zugehörigkeit eines Wortes zum Sprachschatz einer Chomsky-3-Grammatik
  - Existenz der Nullstelle eines Polynoms über den ganzen Zahlen

(2) positiv semientscheibare (aber nicht allgemein entscheidbare) Prädikate
  - Terminierung einer gegebenen Turingmaschine für bestimmte Eingaben
  - Zugehörigkeit eines Wortes zum Sprachschatz einer gegebenen Chomsky-0-Sprache

(3) Negativ semientscheidbar
  - Gleichheit zweier durch primitive Rekursion gegebene Funktionen

(4) Unentscheidbare Probleme (weder positiv noch negativ semientscheidbar)
  - Terminierung einer durch μ-Rekursion beschriebenen Funktion für alle Eingaben

Satz: Das Terminierungsproblem für μ-rekursive ist nicht entscheidbar.

Beweis:
Sei μ-PRG die Menge der Wörter, die μ-rekursive Programme darstellen
Widerspruchsannahme: Terminierungsproblem ist entscheidbar!
Dann exisitert ein μ-rekursives Programm $p \in \mu\text{-}PRG$, das eine Funktion f_p berechnet s.d.g.:

$$f_p(x) = \begin{cases} 0 & \text{falls } \exists q \in \mu\text{-}PRG : code(q) = x \wedge f_q \text{ für } x \text{ nicht definiert} \\ undef & \text{sonst} \end{cases}$$

Wir erhalten für $m = code(p)$: $f_p(m) = \begin{cases} 0 & \text{falls } f_p(m) \text{ } undef. \\ undef & \text{sonst} \end{cases}$ ↯

Widerspruch, da $f_p(m)$ entweder undefiniert ist (dann wäre $f_p(m) = 0$↯) oder $f_p(m) = 0$ (dann wäre $f_p(m)$ undefiniert ↯).

2.4.3 Rekursive und rekursiv aufzählbare Mengen

Wir betrachten jetzt eine Obermenge U, sowie Teilmengen $M \subseteq U$. Ein Prädikat auf U

$$p : U \to \mathbb{B} \quad \text{Charaktermistisches Prädikat zu } M \subseteq U$$

ist dann gegeben durch

$$p(x) = \begin{cases} true & \text{falls } x \in M \\ false & \text{falls } x \notin M \end{cases}$$

Eine Menge heißt **rekursiv**, wenn das charakteristische Prädikat entscheidbar ist.

Satz: Jede Chomsky-1-Sprache ist rekursiv.

Eine totale Abbildung $e : \mathbb{N} \to U$ heißt Aufzählung der Menge M, falls gilt:

$$M = \{e(n) : n \in \mathbb{N}\} = \bigcup_{n \in \mathbb{N}} \{e(n)\}$$

M heißt **rekursiv aufzählbar**, falls e berechenbar ist. Jede rekursiv aufzählbare Menge hat ein positiv semientscheidbares charakteristisches Prädikat.

5.6.2003

Bemerkung: In einer Aufzählung einer Menge $M = \{e(0), e(1), e(2), \ldots\}$ können Elemente beliebig oft vorkommen, aber jedes Element muß mindestens einmal auftreten.

Beispiele:

(1) Die Menge der primitiv rekursiven Funktionen ist rekursiv aufzählbar, aber nicht rekursiv.

(2) Die Menge der Argumente, für die eine μ-rekursive Funktion (TM, RM) terminiert, ist rekursiv aufzählbar, aber nicht rekursiv.

Satz: Jede Chomsky-Sprache ist rekursiv aufzählbar!

Beweis: Die Grammatik der Sprache definiert einen Baum von Ableitungen, aus dem wir ein Aufzählverfahren gewinnen.

Satz: Die Menge der Argumente einer berechenbaren Funktion f, für die f nicht definiert ist (der Algorithmus nicht terminiert), ist i.A. nicht rekursiv aufzählbar.

Beweis: Folgt direkt aus dem Halteproblem.

Satz: Eine Menge $S \subseteq T^*$ ist genau dann rekursiv, wenn S und $T^* \backslash S$ rekursiv aufzählbar sind.

Beweis: Wir zählen S und T^* S parallel auf. Irgendwann tritt jedes Element in einer der Aufzählungen auf. Dann liegst fest, ob es in S oder in $T^* \backslash S$ liegt. Dieses Verfahren terminiert für jedes Element aus T^*.

Kapitel 3

Komplexitätstheorie

Jeder Algorithmus verbraucht bei der Ausführung gewisse Betriebsmittel und erfordert einen Berechnungsaufwand.

Fragen:

(1) Bei einem gegebenen Algorithmusbegriff (TM, RM, μ-Rekursion) können wir nun für ein gegebenes Problem (Beispiel: Multiplikation) fragen, welcher Algorithmus den geringsten Berechnungsaufwand hat.

(2) Wie hängt die Klassifizierung des Berechnungsaufwandes von der Wahl des Algorithmusbegriffs ab?

Wir werden die Aufwandsabschätzung für Algorithmen und Probleme an Turingmaschinen orientieren.

3.1 Komplexitätsmaße

Wir wählen zwei einfache Maßzahlen für den Berechnungsaufwand eines Algorithmus:

- Anzahl der Berechnungsschritte (Zeitkomplexität)
- Anzahl der benötigten („Hilfs-“) Speicherzellen (Bandkomplexität)

3.1.1 Zeitkomplexität

Wir betrachten k-Band-Turingmaschinen. Für jedes Band existiert ein Lese-Schreib-Kopf. Für ein gegebenes Eingabewort (etwa auf Band 1) führt die Turingmaschine eine endliche oder eine unendliche Anzahl von Schritten aus. Terminiert die Turingmaschine für ein Wort w, so bezeichne $b(w)$ die Anzahl der Schritte.

Gilt für $T : \mathbb{N} \to \mathbb{N}$ für alle Wörter $w \in Z^*$ der Länge $n = |w|$: $b(w) \leq T(n)$ so heißt die Turingmaschine (der Algorithmus) $T(n)$-zeitbeschränkt.

Achtung:

- Diese Definition schließt nichtdeterministische Turingmaschinen mit ein. Dann müssen alle Berechnungen für w durch $T(n)$ beschränkt sein.
- Mehrband-Turingmaschinen vermeiden den Aufwand von Kopfbewegungen, um zwischen Argumenten (Beispiel Addition) hin und her zu fahren und liefern “realistischere” Abschätzungen.

Die Definition von Zeitbeschränktheit läßt sich von Algorithmen auf Probleme übertragen: Ein Problem heißt $T(n)$-zeitbeschränkt, wenn es einen Algorithmus gibt, der das Problem berechnet und $T(n)$-zeitbeschränkt ist (nb: Problem entspricht immer der Aufgabe, eine Funktion/ein Prädikat zu berechnen).

Beispiel 1: Das Problem, zu entscheiden, ob ein Wort in der Sprache $L = \{a^k cb^k : k \in \mathbb{N}\}$ ist, ist $(n+1)$-zeitbeschränkt. Dafür ist zu zeigen, daß ein Algorithmus (TM) existiert, der in $n+1$ Schritten für ein Wort w der Länge n entscheidet, ob $w \in L$. Wir sagen, die Turingmaschine ist linear beschränkt.

Beispiel 2: Addition von Zahlen, dargestellt durch Binärzahlen
Eine Turingmaschine für die Addition benötige $\log_2(b+a)+2+2\cdot\log_2(b)+1+1$ Schritte.
Sprechweise: Das Problem ist logarithmisch zeitbeschränkt.

Annahme: Wir betrachten nur Probleme mit $T(n) \geq n+1$, d.h. wir betrachten nur Problemstellungen, bei denen die gesamte Eingabe für den Algorithmus relevant ist.

6.6.2003

3.1.2 Bandkomplexität

Bandkomplexität bewertet den Speicheraufwand eines Algorithmus. Wir messen die Bandkomplexität wieder durch Turingmaschinen über die Anzahl der im Laufe einer Berechnung beschriebenen Speicherzellen.
Wir betrachten Turingmaschinen mit folgender Charakteristik:

- ein Eingabeband, das nicht beschrieben wird und eine Ende-Markierung enthält
- k einseitig unendliche Bänder

Für ein Eingabewort $w \in Z^*$ verwendet die Turingmaschine $b_i(w) \in \mathbb{N}$ Speicherzellen auf ihren i-ten Band $(1 \leq i \leq n)$. Gilt für eine Abbildung $S : \mathbb{N} \to \mathbb{N}$ für alle Wörter w mit $n = |w|$, für alle i, $1 \leq i \leq k$ $b_i(w) \leq S(n)$, so heißt die Turingmaschine $S(n)$-beschränkt. Wir sagen: „Die Turingmaschine hat Bandkomplexität $S(n)$“.

Beispiele: Sprache $L = \{a^n c b^n : n \in \mathbb{N}\}$

Wenn wir die Zahlen in Binärschreibweise notieren, dann benötigen wir $1 + log_2 n$ Speicherzellen, um die a^n Zeichen zu zählen. Wir erhalten einen logarithmisch bandbeschränkten Algorithmus.

Definition: Ein Problem heißt $S(n)$-bandbeschränkt, wenn es einen Algorithmus (eine Turingmaschine) gibt, die das Problem löst und $S(n)$-bandbeschränkt ist.

Wenn wir von Bandkomplexität reden, meinen wir stets $\max\{1, S(n)\}$

Wir sind bei der Abschätzung von Band-/Zeitkomplexität in der Regel nicht an exakten Zahlen interessiert, sondern an Größenordnungen.

Dazu verwenden wir eine Abbildung $g, f : \mathbb{N} \to \mathbb{N}$ um das asymptotische Verhalten abzuschätzen. Wir sagen: „Die Funktion g wächst mit der Ordnung $\Theta(f(n))$“, wenn gilt:

$$\exists c, d \in \mathbb{N}\backslash\{0\}, n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : f(n) \leq d \cdot g(n) \leq c \cdot f(n)$$

Eine Turingmaschine hat Bandkomplexität $S(n)$, falls für jedes Wort $w \in Z^*$ mit $|z| = n$ alle Berechnungen der Turingmaschine höchstens $S(n)$ Speicherzellen pro Band verbrauchen.

3.1.3 Zeit- und Bandkomplexitätsklassen

Eine Problemstellung (eine Aufgabe, eine Berechnunge, eine Funktion/Prädikat) heißt deterministisch $T(n)$-zeitbeschränkt (bzw. deterministisch $T(n)$-bandbeschränkt), wenn es eine deterministische Turingmaschine gibt, die das Problem löst und $T(n)$-zeit(band)beschränkt ist. Analog definiert: nicht deterministische $T(n)$-Zeit(Band)komplexität.

Abkürzungen:	DTIME($T(n)$)	Klasse der Probleme, die deterministisch $T(n)$-zeitbeschränkt sind
	NTIME($T(n)$)	Klasse der Probleme, die nichtdeterministisch $T(n)$-zeitbeschränkt sind
	DSPACE($T(n)$)	Klasse der Probleme, die deterministisch $T(n)$-bandbeschränkt sind
	NSPACE($T(n)$)	Klasse der Probleme, die nichtdeterministisch $T(n)$-bandbeschränkt sind

Wichtig: Hier brauchen wir uns nur für die Größenordnung (asymtotisches Verhalten) zu interessieren, wie folgende Sätze zeigen:

Satz (Bandkompression):
Ist ein Problem $S(n)$-bandbeschränkt, so ist für jede Konstante $c \in \mathbb{R}$ mit $c > 0$ das Problem auch $c \cdot S(n)$-bandbeschränkt.

Beweis:
Konstruiere Turingmaschine, die r Zeichen in ein Zeichen zusammenfaßt.

Korollar:
Für alle $c \in \mathbb{R}$ mit $c > 0$ gilt:
$\text{NSPACE}(S(n)) = \text{NSPACE}(c \cdot S(n))$
$\text{DSPACE}(S(n)) = \text{DSPACE}(c \cdot S(n))$

Satz (Reduktion der Bänderzahl):
Sei M eine $S(n)$-bandbeschränkte Turingmaschine mit k Bändern. Wir können eine Turingmaschine konstruieren, die die k Bänder auf einem Band simuliert.

Satz (linearer Speed-Up):

Wird ein Problem von einer $T(n)$-zeitbeschränkten k-Band-Turingmaschine gelöst, so wird es für jede reelle Zahl $c \in \mathbb{R}, c > 0$ auch durch eine $c \cdot T(n)$-zeitbeschränkte k-Band-Turingmaschine gelöst, falls $k > 1$: $\inf\limits_{n\to\infty} \frac{T(n)}{n} = \infty$

Korollar: Unter dieser Voraussetzung gilt:
$\text{DTIME}(T(n)) = \text{DTIME}(c \cdot T(n))$
$\text{NTIME}(T(n)) = \text{NTIME}(c \cdot T(n))$

Satz (nach Savitch):
Für jedes Akzeptanzproblem (erkennen, ob ein Wort in einer Sprache liegt) gilt: Ist A in $\text{NSPACE}(S(n))$, dann gilt auch A ist in $\text{DSPACE}(S(n)^2)$.

Falls gilt (die Funktion S ist vollbandkostruierbar): Es existiert eine Turingmaschine, so daß gilt: Für alle Zahlen $n \in \mathbb{N}$ existiert ein Eingabewort w mit $|w| = n$, so daß die Turingmaschine tatsächlich $S(n)$ Speicherzellen benötigt.

3.1.4 Polynomiale und nichtdeterministisch polynomiale Zeitkomplexität

Ein Problem heißt (bzgl. Zeitkomplexität) polynomial deterministisch lösbar, wenn es in $\text{DTIME}(n^k)$ mit $k \in \mathbb{N}$ lösbar ist. Analog definieren wir exponentielle Zeitkomplexität mit $\text{DTIME}(2^n)$. Besonders interessant $\text{DTIME}(2^n) \setminus \bigcup_{k\in\mathbb{N}} \text{DTIME}(n^k)$

Dies sind die Probleme, die für größere n theoretisch, aber nicht praktisch berechenbar sind. Wir diskutieren nun zwei Klassen:

$$P = \bigcup_{k\in\mathbb{N}} \text{DTIME}(n^k) \qquad NP = \bigcup_{k\in\mathbb{N}} \text{NTIME}(n^k)$$

Es gilt: $NP \subseteq \text{DTIME}(2^n)$

Frage: Gilt $P = NP$?

12.6.2003

Klarer ist diese Frage für die Band/Speicherkomplexität beantwortet:

$$P_{\text{SPACE}} = \bigcup_{i\geq 1} \text{DSPACE}(n^i) \qquad NP_{\text{SPACE}} = \bigcup_{i\geq 1} \text{NSPACE}(n^i)$$

Nach Satz von Savich gilt $\text{NSPACE}(n^i) \subseteq \text{DSPACE}(n^{2i})$. Dies ergibt sofort $P_{\text{SPACE}} = NP_{\text{SPACE}}$.

Es gilt: $\text{NSPACE}(\log n) \subseteq P \subseteq NP \subseteq P_{\text{SPACE}}$
Dies ergibt eine Klassifizierung von Problemen in solche, die praktisch lösbar sind, und solche, die auf Grund des benötigten Aufwandes (exponentielle Zunahme des Aufwandes abhängig von der Größe der Eingabe) praktisch nicht mehr lösbar sind.

3.1.5 Nichtdeterminismus in Algorithmen – Backtracking

Bisher hatten wir Nichtdeterminismus im wesentlichen in Turingmaschinen studiert. Nun betrachten wir Nichtdeterminismus auf der Ebene von Programmiersprachen.

Nichtdeterminismus liegt in einem System oder Algorithmus vor, wenn bei bestimmten Schritten Wahlmöglichkeiten gegeben sind.

Achtung: Wir betrachten hier keine Annahmen über Wahrscheinlichkeiten für die Wahl der Schritte.

Für funktionale Sprachen können wir Nichtdeterminismus durch folgende Kostruktion einführen: Wir betrachten für Ausdrücke E_1 und E_2 den nichtdeterministischen Ausdruck $E_1 [] E_2$ (Auswahlausdruck). Dieser Ausdruck ergibt bei Auswertung den (einen) Wert von E_1 oder den (einen) Wert E_2.

Bemerkung: Diese Idee läßt sich auch auf Anweisungen übertragen: $S_1 [] S_2$
Auch die Terminierung kann von der nichtdeterministischen Auswahl abhängen. Wir betrachten im Folgenden Algorithmen, bei denen die Terminierung gesichert ist.

Beispiel: Nichtdeterministische Erzeugung von Permutationen

```
{f(n) erzeugt (nichtdeterm.) eine Sequenz, die eine Permutation der Menge {1,...,n} ist}
fct f = (nat n) seq nat:
⌜ if n = 0 then ε
          else einf(f(n − 1), n)
  fi                              ⌟
```

```
{einf(s, n) fügt nichtdeterministisch die Zahl n an einer beliebigen Stelle in s ein}
fct einf = (seq nat s, nat n) seq nat:
⌜ if s = ε then <n>
          else <n> ∘ s [] <first(s)> ∘ einf(rest(s), n)
  fi                                                  ⌟
```

Behauptung: $f(n)$ erzeugt nichtdeterministisch eine Permutation von $\{1, \ldots, n\}$, wobei jede Permutation als Ergebnis auftreten kann.

Beweis: Induktion über $n \in \mathbb{N}$

1. Fall: $n = 0 \Rightarrow f(n)$ liefert ε

2. Fall: Annahme: $f(n-1)$ liefert beliebige Permutation über $\{1, \ldots, n-1\}$. n wird in $f(n-1)$ an beliebiger Stelle eingefügt. Dies liefert eine Permutation von $\{1, \ldots, n\}$; Jede Permutation kann so erzeugt werden.

Bisher haben wir nur Beispiele betrachtet, wo jeder nichtdeterinistische Berechnungspfad zu einem Ergebnis führt, das unseren Vorstellungen genügt. Oft studieren wir nichtdeterministische Berechnungen (Beispiel Turingmaschine), wo gewisse Pfade nicht zu einem brauchbaren Ergebnis führen. Dazu definieren wir einen neuen Ausdruck *failure* der darstellt, daß dieses Ergebnis nicht brauchbar ist.

Beispiel:

```
fct sqrt = (nat n, nat y) nat:
⌜ if   y² > n               then failure
  elif y² ≤ n ≤ (y + 1)²    then y
                            else sqrt(n, 2 ∗ y + 1) [] sqrt(n, 2 ∗ y)
  fi                                                                 ⌟
```

sqrt$(n, 1)$ für $n > 0$ liefert die natürlichzahlige Wurzel, wenn wir vereinbaren, daß nichtdeterministische Berechnungen, die mit *failure* enden, nich akzeptiert werden. Eine nichtdeterministische Berechnung darf nur mit *failure* enden, wenn alle Zweige im „Entscheidungsbaum“ auf *failure* enden.

Beispiel: Erfüllbarkeit von aussagenlogischen Formeln

Wir betrachten eine logische Formel (boolescher Ausdruck), der genau $x_1, \ldots, x_n$ als freie Identifikatoren der Sorte Bool enthält. Der Ausdruck heißt erfüllbar, wenn es Werte $b_1, \ldots, b_n \in \mathbb{B}$ gibt, sodas für $x_1 = b_1, x_2 = b_2, \ldots, x_n = b_n$ der Ausdruck den Wert *true* liefert. Damit ist die Sequenz $<b_1, \ldots, b_n>$ der Nachweis für die Erfüllbarkeit.

Wir stellen den Ausdruck als Funktion dar:
fct ausdruck = (**seq bool** b) **bool**: ...

Ein nichtdeterministischer Ausdruck für die Erfüllbarkeit:

```
fct erfüllbar = (seq bool s) bool:
⌜ if |s| = n then
                 if ausdruck(s) then true
                                else  failure
                 fi
            else  erfüllbar(s ∘ <L>) [] erfüllbar(s ∘ <O>)
  fi                                                      ⌟
```