

Theoretische Informatik

ZUSAMMENFASSUNG DER MODULE THEORETISCHE INFORMATIK 1 & 2

Simon KÖNIG

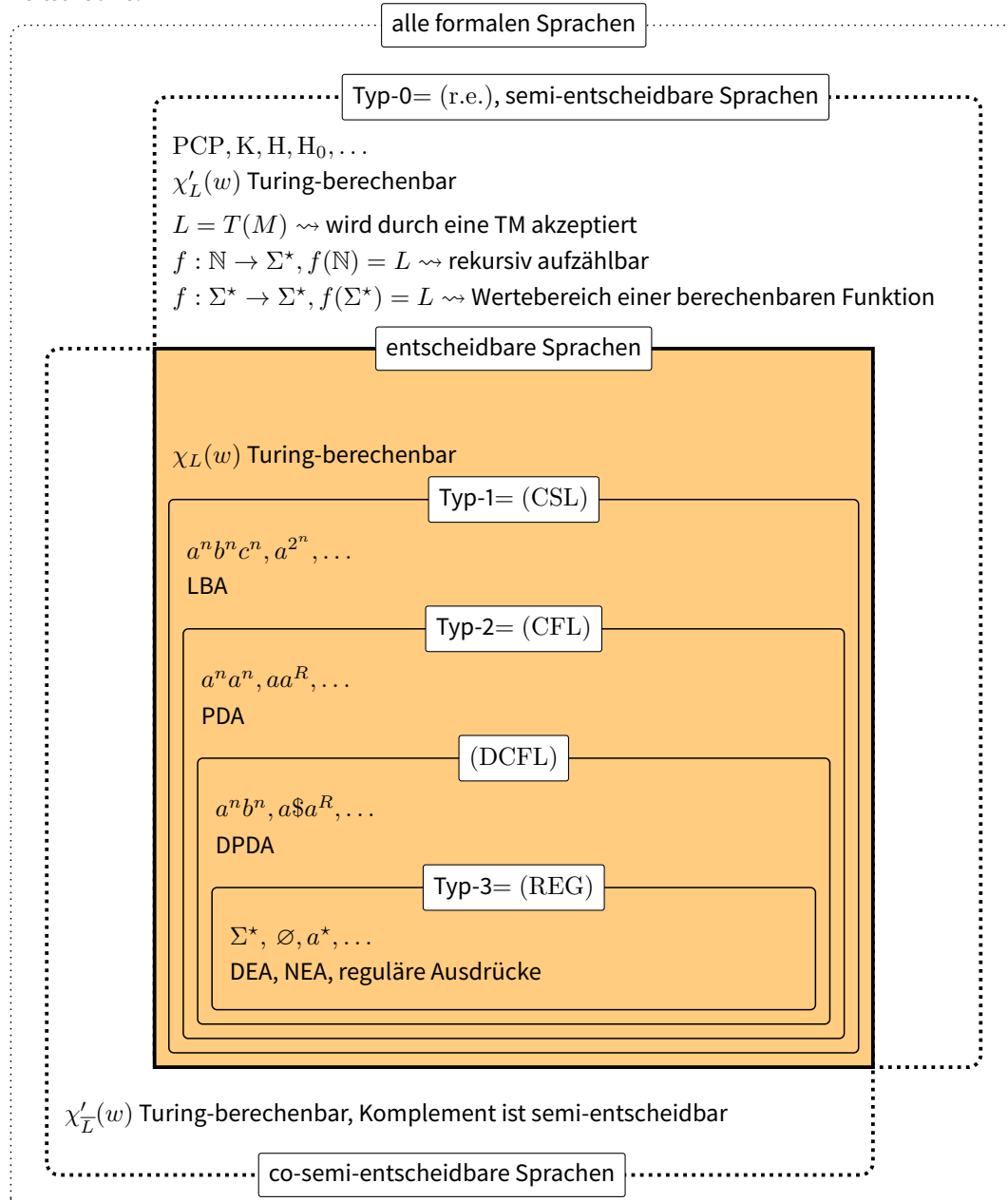
INHALTSVERZEICHNIS

1 Überblick	4
I Grundlagen	5
1 Erste Definitionen	6
2 Grundlagen der Aussagenlogik	7
2.1 Syntax der Aussagenlogik	7
2.2 Semantik der Aussagenlogik	7
2.3 Prädikatenlogik	7
II Formale Sprachen und Automatentheorie	8
1 Endliche Sprachen (FIN)	9
1.1 Beispiele	9
2 Reguläre Sprachen	10
2.1 Automatenmodell DEA	10
2.2 Automatenmodell NEA	10
2.3 Reguläre Ausdrücke	11
2.4 Sätze zu den regulären Sprachen:	11
2.4.1 Pumping-Lemma für Typ-3	11
2.4.2 Myhill-Nerode-Äquivalenz	12
2.4.3 Erkennung durch Monoide – Syntaktisches Monoid	12
2.4.4 Wohldefiniertheit des Produkts der Äquivalenzklassen	13
2.5 Konstruktionsalgorithmus für Minimal-DEAs	13
2.6 Beispiele:	13
2.6.1 Pumping-Lemma für Typ-3	15
2.6.2 Myhill-Nerode-Äquivalenz	15
2.6.3 Beweis durch Abschlusseigenschaften	16
3 Deterministisch kontextfreie Sprachen	17
3.1 Automatenmodell DPDA	17
3.2 Sätze zu (DCFL)	17
3.3 Beispiele	18
4 Kontextfreie Sprachen	19
4.1 Automatenmodell PDA	19
4.2 Sätze zu (CFL)	19
4.2.1 Pumping-Lemma für Typ-2	20
4.2.2 CYK-Algorithmus zur Lösung des Wortproblems für Typ-2	20
4.3 Chomsky-Normalform	20
4.3.1 Umformungsalgorithmus	20

4.4	Greibach-Normalform	21
4.4.1	Umformungsalgorithmus	21
4.5	Beispiele	22
4.5.1	Pumping-Lemma für Typ-2	23
5	Kontextsensitive Sprachen	24
5.1	Automatenmodell Turingmaschine	24
5.1.1	Einschränkung LBA	25
5.2	Sätze zu (CSL)	25
5.2.1	Algorithmus zur Entscheidbarkeit des Wortproblems	25
5.3	Kuroda-Normalform	25
5.3.1	Umformungsalgorithmus	26
5.4	Beispiele	26
III	Berechenbarkeitstheorie und Komplexität	27
1	Rekursiv aufzählbare Sprachen	28
1.1	Sätze zu (r.e.)	28
2	Entscheidbare Sprachen	29
3	Berechenbarkeitstheorie	30
3.1	LOOP-Berechenbarkeit	30
3.1.1	Erlaubte Basisanweisungen	30
3.1.2	Simulierbare Makros	30
3.2	WHILE-Berechenbarkeit	31
3.2.1	Erlaubte Anweisungen	31
3.3	GOTO-Berechenbarkeit	31
3.3.1	Erlaubte Anweisungen	31
3.4	Turing-Berechenbarkeit	31
3.5	Primitive Rekursion	31
3.5.1	Beispiele für primitiv rekursive Funktionen:	32
3.6	μ -Rekursion	32
4	Entscheidbarkeitstheorie	33

1: ÜBERBLICK

Ein kurzer Gesamtüberblick über den Zusammenhang von formalen Sprachen zur Berechenbarkeitstheorie:



1

Grundlagen

1: ERSTE DEFINITIONEN

Definition 1.1: Alphabet

Als Alphabet bezeichnen wir eine endliche, nichtleere Menge, deren Elemente Buchstaben genannt werden.

Dieses wird üblich mit Σ bezeichnet.

Definition 1.2: Freies Monoid über Σ

Ein Monoid ist eine Menge mit einer assoziativen Verknüpfung und einem neutralen Element. Die Menge aller endlichen Zeichenketten, die sich aus Elementen von Σ bilden lassen bilden mit der *Konkatenation* ein Monoid Σ^* .

Das leere Wort (ϵ) bildet das neutrale Element.

Definition 1.3: Grammatik

Eine Grammatik ist ein Quadrupel:

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

V Eine Menge an Zeichen, den Variablen

Σ Das Alphabet, $V \cap \Sigma = \emptyset$

P Die Produktionsmenge, $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$

S Das Startsymbol oder die Startvariable, $S \in V$

2: GRUNDLAGEN DER AUSSAGENLOGIK

2.1 Syntax der Aussagenlogik

- Atomare Formeln: A_i mit $i \in \mathbb{N}$
- F und G Formeln $\rightarrow (F \wedge G), (F \vee G)$ und $\neg F$ auch Formeln

2.2 Semantik der Aussagenlogik

$$D \subseteq \{A_1, A_2, \dots\}$$

Eine Abbildung $\mathcal{A} : D \rightarrow \{0, 1\}$ heißt Belegung. Weiter

- Eine Belegung \mathcal{A} ist **passend** zu einer Formel F , falls alle in F vorkommenden atomaren Variablen zum Definitionsbereich von \mathcal{A} gehören.
- Eine Belegung \mathcal{A} ist **Modell** für eine Formel, falls \mathcal{A} zu F passend ist und $\mathcal{A}(F) = 1$ gilt. Man schreibt dann $\mathcal{A} \models F$.
- F ist **erfüllbar**, falls ein Modell für F existiert.
- F ist **gültig**, falls alle passenden Belegungen Modelle sind. $\rightarrow F$ nennt man dann eine **Tautologie**. Das Komplement einer Tautologie ist unerfüllbar.
- Zwei Formeln F und G heißen **semantisch äquivalent**, wenn alle zu beiden passenden Belegungen \mathcal{A} gilt: $\mathcal{A}(F) = \mathcal{A}(G)$. Man schreibt dann $F \equiv G$.

2.3 Prädikatenlogik

2

Formale Sprachen und Automatentheorie

1: ENDLICHE SPRACHEN (FIN)

$$(\text{FIN}) \subset (\text{REG}) \subset (\text{DCFL}) \subset (\text{CFL}) \subset (\text{CSL}) \subset (\text{REC}) \subset (\text{r.e.})$$

Alle endlichen Sprachen sind regulär.

1.1 Beispiele

- $L_1 = \emptyset$ (leere Sprache ist endlich)
- $L_2 = \Sigma$ (nur die Buchstaben)
- $L_3 = \{aaa, baba\}$ (z.B. nur zwei Wörter)

2: REGULÄRE SPRACHEN

$$\text{Typ-3} = (\text{REG}) \subset (\text{DCFL}) \subset (\text{CFL}) \subset (\text{CSL}) \subset (\text{REC}) \subset (\text{r.e.})$$

2.1 Automatenmodell DEA

Ein deterministischer endlicher Automat ist ein 5-Tupel

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

Z endliche Zustandsmenge

Σ Eingabealphabet

δ Überföhrungsfunktion $\delta : Z \times \Sigma \rightarrow Z$

z_0 Startzustand, $z_0 \in Z$

E Endzustandsmenge, $E \subseteq Z$

Es lässt sich außerdem eine erweiterte Funktion $\hat{\delta}$ definieren:

$$\hat{\delta} : Z \times \Sigma^* \rightarrow Z$$

Mit den folgenden Eigenschaften:

$$\hat{\delta}(z, \epsilon) = z$$

$$\hat{\delta}(z, ax) = \hat{\delta}(\delta(z, a), x)$$

Die von einem deterministischen Automaten M akzeptierte Sprache ist

$$T(M) = \left\{ w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E \right\}$$

2.2 Automatenmodell NEA

Ein nichtdeterministischer endlicher Automat ist ein 5-Tupel

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

Z endliche Zustandsmenge

Σ Eingabealphabet

δ Überföhrungsfunktion $\delta : Z \times \Sigma \rightarrow \text{Pot}(Z)$

S Startzustandsmenge, $S \subseteq Z$

E Endzustandsmenge, $E \subseteq Z$

Der NEA ist formal stärker als der DEA, sie akzeptieren jedoch beide die selbe Sprachklasse. Die akzeptierte Sprache eines nichtdeterministischen endlichen Automaten ist:

$$T(M) = \left\{ w \in \Sigma^* \mid \hat{\delta}(S, w) \cap E \neq \emptyset \right\}$$

2.3 Reguläre Ausdrücke

Die regulären Sprachen lassen sich zusätzlich zu den zwei Automatenmodellen auch durch sog. reguläre Ausdrücke beschreiben. Eine Definition für die Syntax der regulären Ausdrücke ist:

- \emptyset und ϵ sind reguläre Ausdrücke.
- a ist ein regulärer Ausdruck für alle $a \in \Sigma$
- Wenn α und β eguläre Ausdrücke sind, dann sind auch $\alpha\beta$, $(\alpha|\beta)$ und $(\alpha)^*$ reguläre Ausdrücke.

Die Semantik der regulären Ausdrücke ist ebenso induktiv bestimmt:

- $L(\emptyset) = \emptyset$ und $L(\epsilon) = \{\epsilon\}$
- $L(a) = \{a\}$ für jedes $a \in \Sigma$
- $L(\alpha\beta) = L(\alpha)L(\beta)$, $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$, $L((\alpha)^*) = L(\alpha)^*$

2.4 Sätze zu den regulären Sprachen:

- Typ-3 Sprachen können *nicht* inhärent mehrdeutig sein, da sich zu jeder Sprache ein Minimalautomat bilden lässt.
- Die Klasse der Typ-3 Sprachen ist unter allen boole'schen Operationen, Sternoperation und der Konkatenation abgeschlossen.
- Für reguläre Sprachen ist das Wortproblem (in Linearzeit), das Leerheitsproblem, das Äquivalenzproblem sowie das Schnittproblem entscheidbar.
- Alle Typ-2 Sprachen über einem einelementigen Alphabet sind bereits regulär.

2.4.1 Pumping-Lemma für Typ-3

Für jede reguläre Sprache L gibt es ein $n \in \mathbb{N}$, so dass für jedes $x \in L$ mit $|x| \geq n$ eine Zerlegung in drei Teile exisitiert: $x = uvw$, so dass die drei Bedingungen erfüllt sind:

- $|v| \geq 1$
- $|uv| \leq n$
- $\forall i \in \mathbb{N} : uv^i w \in L$ („Pump-Bedingung“)

Gilt die Negation dieser Aussage, also

$$\forall n \in \mathbb{N} : \exists x \in L, |x| \geq n : \forall u, v, w \in \Sigma^*, x = uvw, |v| \geq 1, |uv| \leq n : \exists i \in \mathbb{N} : uv^i w \notin L$$

so ist L nicht regulär!

ABER: Das Pumping-Lemma gibt keine Charakterisierung der Typ-3 Sprachen an! Es gibt auch Sprachen, die nicht Typ-3 sind, die Aussage des Lemmas aber trotzdem erfüllen! Das Pumping-Lemma gibt also nur eine Möglichkeit, zu Beweisen, dass eine Sprache *nicht* regulär ist! (Siehe Unterabschnitt 2.6.1)

2.4.2 Myhill-Nerode-Äquivalenz

Mit der Myhill-Nerode-Äquivalenz ist es möglich nachzuweisen, ob eine Sprache regulär ist.

$$xR_L y \iff [\forall w \in \Sigma^* : xw \in L \iff yw \in L]$$

Bzw. anhand eines DEA (dies führt zu einer Verfeinerung von R_L)

$$xR_M y \iff [\delta(z_0, x) = \delta(z_0, y)]$$

Es gilt:

$$xR_M y \Rightarrow \forall w \in \Sigma^* : \delta(z_0, xw) = \delta(z_0, yw) \Rightarrow xR_L y$$

Die Sprache $L \subseteq \Sigma^*$ ist genau dann regulär, wenn der Index der Myhill-Nerode-Äquivalenz R_L endlich ist.

2.4.3 Erkennung durch Monoide – Syntaktisches Monoid

Sei $L \subseteq \Sigma^*$ eine formale Sprache und M ein Monoid.

M erkennt L , wenn eine Teilmenge $A \subseteq M$ und ein Homomorphismus $\varphi : \Sigma^* \rightarrow M$ existiert, so dass gilt:

$$\begin{aligned} L &= \varphi^{-1}(A) & \text{d.h. } w \in L &\iff \varphi(w) \in A \\ L &= \varphi^{-1}(\varphi(L)) & \text{d.h. } w \in L &\iff \varphi(w) \in \varphi(L) \end{aligned}$$

Weiter kann man für eine konkrete Sprache L die *syntaktische Kongruenz* definieren:

$$w_1 \equiv_L w_2 \iff [\forall x, y \in \Sigma^* : xw_1y \in L \iff xw_2y \in L]$$

Basierend auf dieser Kongruenz definieren wir das Quotientenmonoid der Kongruenz, dessen Elemente die Äquivalenzklassen sind.

Das Quotientenmonoid oder auch syntaktisches Monoid bezüglich der syntaktischen Kongruenz wird mit

$$\text{Synt}(L) := (\Sigma^* / \equiv_L)$$

bezeichnet.

Für jede Sprache L gibt es ein syntaktisches Monoid das L mit dem Homomorphismus

$$\varphi : L \rightarrow \text{Synt}(L), w \mapsto [w]$$

erkennt. Ist $|\text{Synt}(L)|$ endlich, so ist L regulär bzw. erkennbar.

2.4.4 Wohldefiniertheit des Produkts der Äquivalenzklassen

$$[u_1]_{\equiv_L} \cdot [u_2]_{\equiv_L} \stackrel{?}{=} [u_1 u_2]_{\equiv_L} \quad \text{wohldefiniert?}$$

Es gilt:

$$u_1, \widetilde{u_1} \in [u_1]_{\equiv_L} \iff u_1 \equiv_L \widetilde{u_1} \iff [\forall x, y \in \Sigma^* : xu_1y \in L \Leftrightarrow x\widetilde{u_1}y \in L] \quad (2.1)$$

$$u_2, \widetilde{u_2} \in [u_2]_{\equiv_L} \iff u_2 \equiv_L \widetilde{u_2} \iff [\forall x, y \in \Sigma^* : xu_2y \in L \Leftrightarrow x\widetilde{u_2}y \in L] \quad (2.2)$$

Setzt man nun in Gleichung 2.1 für $y = u_2 y'$ ein:

$$u_1, \widetilde{u_1} \in [u_1]_{\equiv_L} \iff u_1 \equiv_L \widetilde{u_1} \iff [\forall x, y' \in \Sigma^* : xu_1 u_2 y' \in L \Leftrightarrow x\widetilde{u_1} u_2 y' \in L]$$

Äquivalent gilt das selbe für $\widetilde{u_2}$. Ebenso gilt das gleiche für Gleichung 2.2 mit $x = x' u_1$:

$$u_2, \widetilde{u_2} \in [u_2]_{\equiv_L} \iff u_2 \equiv_L \widetilde{u_2} \iff [\forall x', y \in \Sigma^* : x' u_1 u_2 y \in L \Leftrightarrow x' u_1 \widetilde{u_2} y \in L]$$

Äquivalent gilt das selbe für $\widetilde{u_1}$. Setzt man nun alle Gleichungen zusammen, erhält man:

$$u_1 u_2 \equiv_L \widetilde{u_1} \widetilde{u_2} \iff [\forall x, y \in \Sigma^* : xu_1 u_2 y \in L \Leftrightarrow x\widetilde{u_1} \widetilde{u_2} y \in L]$$

Dies gilt wie oben gezeigt für alle Kombinationen $u_1 u_2$, $u_1 \widetilde{u_2}$, $\widetilde{u_1} u_2$ und $\widetilde{u_1} \widetilde{u_2}$. Damit erzeugt diese Äquivalenz die Klasse:

$$[u_1 u_2]_{\equiv_L} \text{ mit } u_1 u_2, u_1 \widetilde{u_2}, \widetilde{u_1} u_2, \widetilde{u_1} \widetilde{u_2} \in [u_1 u_2]_{\equiv_L}$$

2.5 Konstruktionsalgorithmus für Minimal-DEAs

Mit dem Beweis zur Myhill-Nerode-Äquivalenz wird ein Automat definiert, dieser ist isomorph zum Minimalautomaten. Der Index der Myhill-Nerode-Äquivalenz ist genau die Anzahl der Zustände des Minimalautomaten.

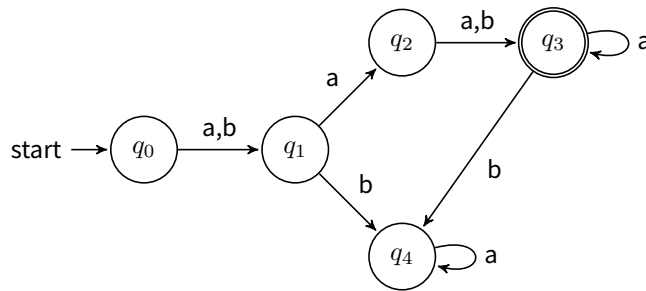
Man kann mit einem einfachen Algorithmus aus einem beliebigen DEA den Minimalautomaten erzeugen:

Wir ermitteln algorithmisch, welche Zustände nicht äquivalent sind und verschmelzen die übrig bleibenden. Nicht äquivalent sind Zustände, bei denen bei Eingabe eines Worts vom einen aus ein Endzustand erreicht wird, vom anderen jedoch nicht.

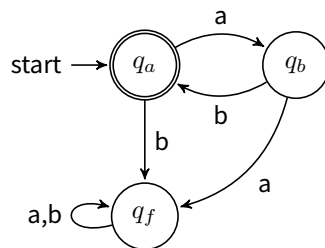
So sind im ersten Schritt Zustandspaare aus Endzustand und Nichtendzustand nicht äquivalent und werden markiert.

2.6 Beispiele:

- $L_1 = \Sigma^*$
- $L_2 = L((a|b)a(a|b)a^*)$

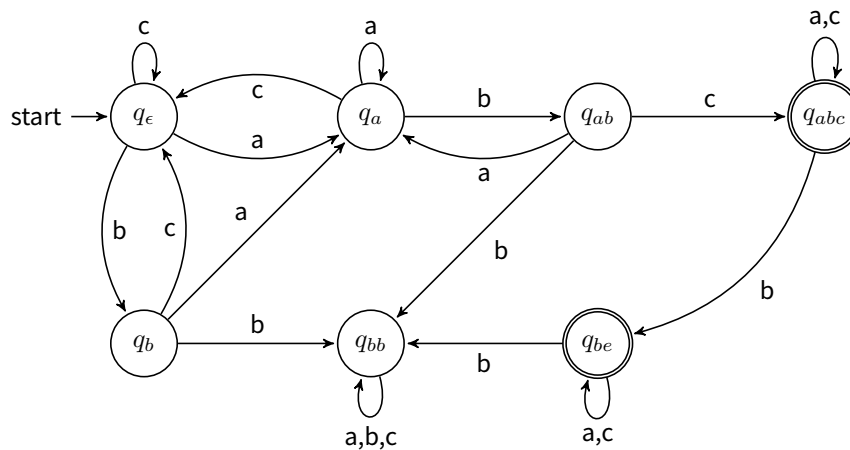


- Automat M mit $L_3 = T(M) = \{(ab)^n \mid n \in \mathbb{N}_0\}$:

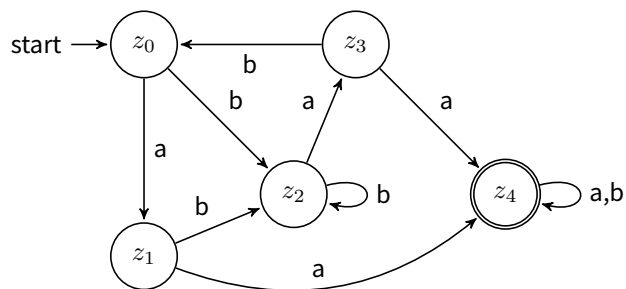


- Automat erkennt die Sprache

$L_3 = \{w \in \{a, b, c\}^* \mid w \text{ enthält das Teilwort } abc \text{ aber nicht das Teilwort } ab\}$:



Konstruktion für den Minimalautomaten



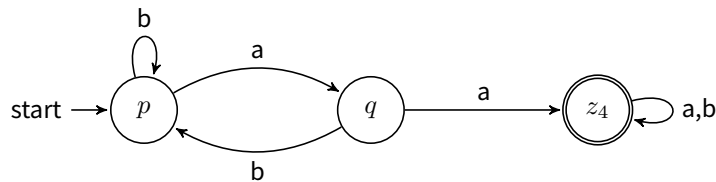
Die Paare $\{z_i, z_4\}$ mit $i = 0, 1, 2, 3$ werden markiert:

z_1				
z_2				
z_3				
z_4	ϵ	ϵ	ϵ	ϵ
	z_0	z_1	z_2	z_3

Durch Testen der Zustandspaare erhält man die untenstehende Tabelle. Hierbei wurden Zeugen für die Inäquivalenz eingetragen.

z_1	a			
z_2		a		
z_3	a		a	
z_4	ϵ	ϵ	ϵ	ϵ
	z_0	z_1	z_2	z_3

Damit lassen sich Zustände zusammenfassen: $p = \{z_0, z_2\}$, $q = \{z_1, z_3\}$. Der Minimalautomat ist also:



$$T(M) = \{waaw' \mid w, w' \in \{a, b\}^*\}$$

2.6.1 Pumping-Lemma für Typ-3

Für die Sprache $L = \{a^n b^n \mid n \geq 1\}$:

Zunächst wählt man ein Wort x , mit $|x| \geq n$:

$$x = a^n b^n \in L, \quad |a^n b^n| = 2n > n$$

Nun zu einer beliebigen Zerlegung $x = uvw$, für die die Bedingungen gelten:

$$x = uvw = a^n b^n$$

$$u = a^{n-l-k} \quad v = a^l \quad w = a^k b^n \text{ mit } l \geq 1$$

$$x = (a^{n-l-k})(a^l)(a^k b^n)$$

Pumpt man nun v mit $v = 0$:

$$x = (a^{n-l-k})(a^l)^i(a^k b^n) = (a^{n-l-k})(a^l)^0(a^k b^n)$$

$$x = (a^{n-l-k})(a^k b^n) = a^{n-l} b^n \notin L, \text{ da } l \geq 1$$

2.6.2 Myhill-Nerode-Äquivalenz

Für die Sprache $L = \{w \in \{a, b\}^* \mid w \text{ enthält das Teilwort } abb\}$:

Finden der Äquivalenzklassen:

$$[\epsilon] = \{\epsilon, b, bb, \dots\} = \{b^n \mid n \in \mathbb{N}_0\}$$

$$[a] = \{b^n a^m \mid n \in \mathbb{N}_0, m \in \mathbb{N}^+\}$$

$$[ab] = \{w \in \{a, b\}^* \mid w \text{ enthält das Teilwort } ab \text{ aber nicht das Teilwort } abb\}$$

$$[abb] = \{wabbw' \mid w, w' \in \{a, b\}^*\}$$

2.6.3 Beweis durch Abschlusseigenschaften

Zu zeigen: $L = \{b^n a^m \mid n, m \in \mathbb{N}_0, n \neq m\} \notin (\text{REG})$

Annahme:

$$L \in (\text{REG})$$

Mit dem Abschluss gegen Komplement folgt

$$\overline{L} \in (\text{REG})$$

Wegen Abschluss gegen Schnitt folgt dann auch

$$\overline{L} \cap L(b^* a^*) \in (\text{REG})$$

Jedoch gilt

$$\overline{L} \cap L(b^* a^*) = \{b^n a^n \mid n \in \mathbb{N}_0\} \in (\text{DCFL}) \not\supseteq (\text{REG})$$

Widerspruch! $L \notin (\text{REG})$

3: DETERMINISTISCH KONTEXTFREIE SPRACHEN

$$(\text{DCFL}) \subset (\text{CFL}) \subset (\text{CSL}) \subset (\text{REC}) \subset (\text{r.e.})$$

3.1 Automatenmodell DPDA

Der deterministische Kellerautomat ist ähnlich definiert wie ein nichtdeterministischer (Siehe Abschnitt 4.1).

Der Unterschied zum PDA liegt dabei, dass beim DPDA in jeder Situation nur ein Übergang möglich sein darf,

$$\forall z \in Z, a \in \Sigma, A \in \Gamma : |\delta(z, a, A)| + |\delta(z, \epsilon, A)| \leq 1$$

und der DPDA akzeptiert nicht durch leeren Keller sondern durch Endzustände.

EIN DETERMINISTISCHER KELLERAUTOMAT IST EIN 7-TUPEL

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$$

Z endliche Zustandsmenge

Σ Eingabealphabet

Γ Kelleralphabet

δ Überföhrungsfunktion $\delta : Z \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Z \times \Gamma^*$

z_0 Startzustand, $z_0 \in Z$

$\#$ Keller-Bottom-Symbol $\# \in \Gamma \setminus \{\Sigma\}$

E Endzustandsmenge $E \subseteq Z$

AKZEPTIERTE SPRACHE EINES DETERMINISTISCHEN PDA:

$$N(M) := \{w \in \Sigma^* \mid \exists e \in E, V \in \Gamma^* : (z_0, w, \#) \vdash^* (e, \epsilon, V)\}$$

dies wird auch als *Akzeptieren durch Endzustand* bezeichnet.

Beide Akzeptanzarten, (durch Endzustand und leeren Keller, siehe Abschnitt 4.1) sind äquivalent.

3.2 Sätze zu (DCFL)

- Eine deterministisch kontextfreie Sprache geschnitten mit einer regulären Sprache ist wieder eine deterministisch kontextfreie Sprache.

$$L_1 \in (\text{DCFL}), L_2 \in (\text{REG}) \Rightarrow L_1 \cap L_2 \in (\text{DCFL})$$

- Die Klasse der deterministisch kontextfreien Sprachen ist nur abgeschlossen unter Komplement.
- Das Leerheitsproblem (Markieren von produktiven Variablen), das Wortproblem (in Linearzeit mit Kellerautomat) und das Äquivalenzproblem sind entscheidbar.

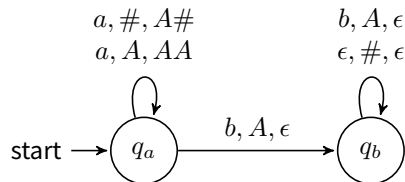
Zum Äquivalenzproblem:

$$\begin{aligned} L = L' &\Leftrightarrow L \subseteq L' \wedge L' \subseteq L \\ &\Leftrightarrow L \cap \overline{L'} = \emptyset \wedge L' \cap \overline{L} = \emptyset \end{aligned}$$

entscheidbar, da Abschluss unter Komplement und Leerheitsproblem entscheidbar.

3.3 Beispiele

- $L_1 = \{w\$w^R \mid w \in \Sigma^*\}$ (markierte Palindrome)
- Deterministischer Kellerautomat, der die Sprache $L_2 = \{a^n b^n \mid n \geq 1\}$ akzeptiert:



Konfigurationsübergänge bei Eingabewort $aaabb$:

$$\begin{aligned} (z_a, aaabb, \#) &\vdash (z_a, aabb, A\#) \\ &\vdash (z_a, abb, A\#) \\ &\vdash (z_a, bb, AAA\#) \\ &\vdash (z_a, b, AA\#) \\ &\vdash (z_a, \epsilon, A\#) \\ &\rightsquigarrow aaabb \notin N(M) = L_2 \end{aligned}$$

4: KONTEXTFREIE SPRACHEN

$$\text{Typ-2} = (\text{CFL}) \subset (\text{CSL}) \subset (\text{REC}) \subset (\text{r.e.})$$

4.1 Automatenmodell PDA

Der Push DownAutomat (Kellerautomat) ist ähnlich definiert wie ein nichtdeterministischer endlicher Automat (Siehe Abschnitt 2.2).

Der PDA jedoch hat einen entscheidenden Unterschied, er hat einen sogenannten *Kellerspeicher*, in dem Informationen zwischengespeichert werden können.

EIN KELLERAUTOMAT IST EIN 6-TUPEL

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

Z endliche Zustandsmenge

Σ Eingabealphabet

Γ Kelleralphabet

δ Überföhrungsfunktion $\delta : Z \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \text{Pot}_e(Z \times \Gamma^*)$

z_0 Startzustand, $z_0 \in Z$

$\#$ Keller-Bottom-Symbol $\# \in \Gamma \setminus \{\Sigma\}$

AKZEPTIERTE SPRACHE EINES NICHTDETERMINISTISCHEN PDA

$$N(M) := \{w \in \Sigma^* \mid \exists z \in Z : (z_0, w, \#) \vdash^* (z, \epsilon, \epsilon)\}$$

dies wird auch als *Akzeptieren durch leeren Keller* bezeichnet.

Insbesondere beim deterministischen Kellerautomaten gibt es eine zweite Definition der akzeptierten Sprache, beide Definitionen sind äquivalent (Siehe Abschnitt 3.1).

KONFIGURATION DES PDA Eine Konfiguration ist jedes Element k aus der Menge $Z \times \Sigma^* \times \Gamma^*$

Einen Konfigurationsübergang stellt man durch \vdash dar.

Die aktuelle Konfiguration sei $(z, a_1 a_2 \dots a_n, A_1 \dots A_m)$, möglich sind jetzt die Übergänge aus $\delta(z, a_1, A_1)$ und $\delta(z, \epsilon, A_1)$.

Ein möglicher Übergang ist also für $(z', B_1 \dots B_k) \in \delta(z, a_1, A_1)$:

$$(z, a_1 a_2 \dots a_n, A_1 \dots A_m) \vdash (z', a_2 \dots a_n, B_1 \dots B_k A_2 \dots A_m)$$

4.2 Sätze zu (CFL)

- Alle Typ-2 Sprachen über einem einelementigen Alphabet sind bereits regulär.

- Die Kontextfreien Sprachen sind gegen Substitution abgeschlossen.
- Die Klasse der kontextfreien Sprachen ist abgeschlossen unter Sternoperation, Vereinigung und Konkatenation.

Zur Vereinigung: $G_1 = (V_1, \Sigma, P_1, S_1), G_2 = (V_2, \Sigma, P_2, S_2), V_1 \cap V_2 = \emptyset, S \notin V_1 \cup V_2$.

$$G = (V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{(S, S_1), (S, S_2)\}, S) \rightsquigarrow L(G) = L(G_1) \cup L(G_2)$$

- Das Wortproblem ($\mathcal{O}(n^3)$) sowie das Leerheits- und Endlichkeitsproblem ist entscheidbar (Siehe Unterabschnitt 4.2.2).
- Die Klasse der Typ-2 Sprachen ist identisch zu der durch EBNF beschreibbaren Sprachen.

4.2.1 Pumping-Lemma für Typ-2

Sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache, dann gibt es eine Zahl n so, dass für alle $z \in L$ mit $|z| \geq n$ eine Zerlegung mit $z = uvwxy$ in $u, v, w, x, y \in \Sigma^*$ existiert für die die drei Bedingungen erfüllt sind:

- $|vx| \geq 1$
- $|vwx| \leq n$
- $\forall i \in \mathbb{N} : uv^iwx^iy \in L$

4.2.2 CYK-Algorithmus zur Lösung des Wortproblems für Typ-2

Mit dem CYK-Algorithmus ist das Wortproblem für Typ-2 in $\mathcal{O}(n^3)$ entscheidbar. Hierfür werden alle Ableitungsmöglichkeiten in einer Tabelle geordnet dargestellt. Ist am Ende die Startvariable als Startknoten für die Ableitung möglich, so ist das Wort in L .

4.3 Chomsky-Normalform

Eine Typ-2 Grammatik (V, Σ, P, S) ist in Chomsky-Normalform (CNF), wenn gilt:

$$\forall (u, v) \in P : v \in V^2 \cup \Sigma$$

Zu jeder Typ-2 Grammatik existiert eine Grammatik G' in CNF, für die gilt $L(G) = L(G')$!
Für alle Ableitungen in CNF gilt: Die Ableitung eines Wortes der Länge n benötigt genau $2n - 1$ Schritte!

4.3.1 Umformungsalgorithmus

1. Zunächst wollen wir erreichen, dass folgendes gilt: $(u, v) \in P \Rightarrow (|v| > 1 \vee v \in \Sigma)$

(a) **Ringableitungen entfernen:**

Eine Ringableitung liegt vor, wenn es Variablen V_1, \dots, V_r gibt, die sich im Kreis in einander ableiten lassen, d.h. es gibt Regeln $V_i \rightarrow V_{i+1}$ und $V_r \rightarrow V_1$.

Um dies loszuwerden, werden alle Variablen V_i durch eine neue Variable V ersetzt. Überflüssige Regeln wie $V \rightarrow V$ werden gelöscht.

(b) **Variablen anordnen:**

Man legt eine Ordnung der Variablen fest: $V = \{A_1, A_2, \dots, A_n\}$, hierfür muss gelten:

$$A_i \rightarrow A_j \in P \Leftrightarrow i < j$$

Falls dies nicht gilt, müssen Abkürzungen verwendet werden, also alle Produktionen von A_j werden eingesetzt:

$$P = (P \setminus \{A_i \rightarrow A_j\}) \cup \{(A_i, w) \mid (A_j, w) \in P\}$$

2. Jetzt gilt für jede Regel $(u, v) \in P$ entweder $v \in \Sigma$ oder $|v| \geq 2$.

Für letztere Regeln werden nun **Pseudoterminal**e eingeführt. Es werden neue Variablen und Produktionen für jedes Terminalsymbol hinzugefügt, z.B. $V_a \rightarrow a$.

3. **Letzter Schritt:** Alle rechten Seiten mit $|v| > 2$ müssen nun noch auf Länge 2 gekürzt werden. Hierfür werden wiederum neue Variablen eingefügt:

$$A \rightarrow C_1 C_2 C_3$$

Wird gekürzt zu

$$A \rightarrow C_1 D_1$$

$$D_1 \rightarrow C_2 C_3$$

4.4 Greibach-Normalform

Eine Typ-2 Grammatik (V, Σ, P, S) ist in Greibach-Normalform (GNF), wenn gilt:

$$\forall (u, v) \in P : v \in \Sigma V^*$$

Zu jeder Typ-2 Grammatik existiert eine Grammatik G' in GNF, für die gilt $L(G) = L(G')$!

4.4.1 Umformungsalgorithmus

1. Der erste Algorithmus:

Der erste Algorithmus hat zum Ziel, dass alle Produktionen einer bestimmten Ordnung unterliegen. Hierfür werden zunächst die Variablen angeordnet:

$$V = \{A_1, A_2, \dots, A_m\}$$

Nun sollen nur Regeln $A_i \rightarrow A_j \alpha$ in P sein, wenn $i < j$ gilt.

Um dies zu erreichen wird solange eingesetzt bis keine Regeln dieser Form vorliegen.

Für alle $A_i \rightarrow A_j \alpha \in P$ mit $i > j$ werden alle Produktionsregeln

$$A_j \rightarrow \beta_1 \mid \dots \mid \beta_r$$

Eingesetzt zu

$$A_i \rightarrow \beta_1 \alpha \mid \dots \mid \beta_r \alpha$$

Und schließlich die Regel $A_i \rightarrow A_j \alpha$ aus P gestrichen.

2. Beseitigung von Linksrekursion:

Alle Produktionsregeln sind von der Form:

$$A \rightarrow A\alpha_1 | \dots | A\alpha_k | \beta_1 | \dots | \beta_l$$

Diese können durch diese $2k + 2l$ Regeln ersetzt werden:

$$A \rightarrow \beta_1 | \dots | \beta_l$$

$$A \rightarrow \beta_1 B | \dots | \beta_l B$$

$$B \rightarrow \alpha_1 | \dots | \alpha_k$$

$$B \rightarrow \alpha_1 B | \dots | \alpha_k B$$

Nun sind keinerlei Linksrekursionen mehr vorhanden!

3. Der zweite Algorithmus:

Der zweite Algorithmus hat zum Ziel, dass alle rechten Seiten mit einem Terminalsymbol beginnen.

Da die Regeln mit A_m auf der linken Seite nur in ein Terminal übergehen können, da sie am Ende der Variablenanordnung stehen müssen wir nur alle A_m -Produktionen bei den A_{m-1} -Regeln einsetzen und so weiter.

4. Der letzte Schritt:

Als letztes müssen wir überprüfen ob die B -Regeln aus der Beseitigung der Linksrekursionen die gewünschte Form haben. Da aber alle B -Produktionen entweder mit einem A_i oder einem Terminal beginnen, muss wieder nur eingesetzt werden.

Schließlich müssen die Terminalsymbole, die in allen Produktionen weiter hinten auftreten durch Pseudoterminale ersetzt werden.

4.5 Beispiele

- $L_1 = \{ww^R \mid w \in \Sigma^*\}$ (unmarkierte Palindrome)
- Korrekt geklammerte arithmetische Ausdrücke (Dyck-Wörter)

CYK-Algorithmus

Produktionsregeln der Grammatik (CNF):

$$S \rightarrow AX|YB, A \rightarrow XA|AB|a, B \rightarrow XY|BB, X \rightarrow YA|a, Y \rightarrow XX|b$$

Eingabewort: *abbaab*

Länge	a	b	b	a	a	b
1	{A, X}	{Y}	{Y}	{A, X}	{A, X}	{Y}
2	{B}	∅	{X}	{A, S, Y}	{B}	
3	∅	∅	{A, Y, X}	{A}		
4	∅	{X}	{B, X}			
5	{S, Y}	{B, S}				
6	{A, B}					

Da die unterste Zelle nun $\{A, B\}$ enthält, ist $w = abbaab \notin L$

4.5.1 Pumping-Lemma für Typ-2

- Für die Sprache $L = \{a^n b^n c^n \mid n \geq 1\}$:

Zunächst wählt man ein Wort z , mit $|z| \geq n$:

$$x = a^n b^n c^n \in L, \quad |a^n b^n c^n| = 3n > n$$

Nun zu einer beliebigen Zerlegung $z = uvwxy$, für die die Bedingungen gelten. Da $|vwx| \leq n$, können v und x nur maximal zwei unterschiedliche Buchstaben beinhalten, niemals jedoch a , b und c .

Damit kann uv^iwx^iy nicht in L sein für ein $i \neq 1$. □

- Für die Sprache $L = \{a^n \mid n \text{ ist eine Quadratzahl}\}$:

Zunächst wählt man ein Wort x , mit $|x| \geq n$:

$$z = a^{n^2} \in L$$

Bei jeder Zerlegung $z = uvwxy$ sind nur as in den Teilwörtern v und x , die gepumpt werden.

Betrachtet man die Länge $|vx| = r$, so gilt:

$$|uv^iwx^iy| = n^2 + r(i - 1)$$

Insbesondere gilt für das Wort

$$\begin{aligned} z' &= uv^2wx^2y = a^s \\ |z'| &= n^2 + r = s \end{aligned}$$

Das hieße, $n^2 + r$ müsste eine Quadratzahl sein damit x' wiederum in L läge. Für r gilt aber die Ausgangsbedingung des Pumping-Lemmas!

$$|vwx| = r + |w| \leq n$$

s kann damit unmöglich eine Quadratzahl sein. □

5: KONTEXTSENSITIVE SPRACHEN

Typ-1 = (CSL) \subset (REC) \subset (r.e.)

5.1 Automatenmodell Turingmaschine

EINE TURINGMASCHINE IST EIN 7-TUPEL

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

Z endliche Zustandsmenge

Σ Eingabealphabet $\Sigma \subseteq \Gamma$

Γ Bandalphabet

δ Überföhrungsfunktion $\delta : Z \times \Gamma \rightarrow Z \times \Gamma \times \{L, R, N\}$

z_0 Startzustand, $z_0 \in Z$

\square Leersymbol, $\square \in \Gamma \setminus \Sigma$

E Endzustandsmenge $E \subseteq Z$

KONFIGURATION Eine Konfiguration k einer Turingmaschine definieren wir als ein Element der Menge

$$k \in \Gamma^* Z \Gamma^*$$

Das bedeutet, dass sich die Turingmaschine in der Konfiguration $k = \alpha z \beta$ im Zustand z befindet und der Schreib- Lesekopf das erste Symbol von β liest.

DIE AKZEPTIERTE SPRACHE EINER TURINGMASCHINE Die akzeptierte Sprache einer Turingmaschine ist definiert als

$$T(M) := \{w \in \Sigma^* \mid \exists \alpha, \beta \in \Gamma^*, e \in E : z_0 w \vdash^* \alpha e \beta\}$$

Wenn $w \notin T(M)$ können drei verschiedene Dinge geschehen:

1. Die Turingmaschine hält an und ist nicht in einem Endzustand.
2. Die Turingmaschine läuft weiter und kommt irgendwann in eine Schleife
 $z_0 w \vdash^* \alpha z \beta \vdash^* \alpha' z' \beta' \vdash^* \alpha z \beta \vdash^* \dots$
3. Die Turingmaschine rechnet endlos, ohne in eine Schleife zu kommen (Halteproblem).

5.1.1 Einschränkung LBA

Ein linear beschränkter Automat ist eine Turingmaschine, die bei der Verarbeitung der Eingabe niemals den Platz der Eingabe auf dem Arbeitsband verlässt.

Um das zu erreichen muss das letzte Symbol der Eingabe markiert sein um den rechten Rand der Eingabe erkennbar zu machen, deshalb definieren wir:

$$\Sigma' = \Sigma \cup \{\hat{a} \mid a \in \Sigma\}$$

Eine nichtdeterministische Turingmaschine nennen wir einen linear beschränkten Automaten, wenn gilt:

$$\forall a_1 a_2 \dots a_{n-1} \hat{a}_n \in \Sigma^+, \alpha, \beta \in \Gamma^*, z \in Z \text{ mit } z_0 a_1 a_2 \dots a_{n-1} \hat{a}_n \vdash^* \alpha z \beta : |\alpha \beta| \leq n$$

Für die akzeptierte Sprache gilt dann:

$$T(M) := \{a_1 \dots a_n \in \Sigma^* \mid \exists \alpha, \beta \in \Gamma^*, e \in E : z_0 a_1 \dots a_n \vdash^* \alpha e \beta\}$$

Die Klasse der durch LBAs erkannten Sprachen ist gleich der Typ-1 Sprachen!

Dabei ist die Frage ob nichtdeterministische und deterministische LBAs gleich mächtig sind noch offen.

5.2 Sätze zu (CSL)

- Die Klasse der kontextsensitiven Sprachen ist abgeschlossen unter allen Operationen!
- Für die kontextsensitiven Sprachen und alle höheren Sprachklassen gibt es die sog. Epsilon-Sonderregel, die oft verwendet wird um trotz der Eigenschaft nichtverkürzend ein Wort der Länge Null zu erreichen:

$$(S, \epsilon) \in P \text{ aber } \forall (u, v) \in P : S \not\subseteq v$$

5.2.1 Algorithmus zur Entscheidbarkeit des Wortproblems

Die Funktion $\text{Abl}_n(X)$ wird iteriert angewendet, bis sich entweder X nicht mehr ändert ($w \notin L(G)$) oder das gesuchte Wort w in X enthalten ist ($w \in L(G)$).

Dabei ist n die Länge des gesuchten Worts w , also $|w|$.

Die Funktion $\text{Abl}_n(X)$ ist für eine Grammatik G wie folgt definiert:

$$\text{Abl}_n(X) := X \cup \{w \in (V \cup \Sigma)^* \mid |w| \leq n \wedge \exists y \in X : y \Rightarrow_G w\}$$

5.3 Kuroda-Normalform

Eine Typ-1 Grammatik (V, Σ, P, S) ist in Kuroda-Normalform wenn alle Regeln von einem der vier Typen sind:

- $A \rightarrow a$
- $A \rightarrow B$
- $A \rightarrow BC$
- $AB \rightarrow CD$

Wobei $A, B, C, D \in V$ und $a \in \Sigma$ ist.

Zu jeder Typ-1 Grammatik existiert eine äquivalente Grammatik in Kuroda-Normalform

5.3.1 Umformungsalgorithmus

1. **Pseudoterminale:** Zunächst alle Terminalsymbole, die nicht alleine auf einer rechten Seite stehen durch Pseudoterminale ersetzen.

Jetzt stören noch die Regeln $A_1 \dots A_n \rightarrow B_1 \dots B_m$ mit $1 \leq n \leq m, m > 2$

2. Für Regeln der Form $A \rightarrow B_1 \dots B_m, m > 2$ können wir die gleiche Methode wie bei den Typ-2 Normalformen anwenden (Siehe Unterabschnitt 4.3.1).
3. Regeln der Form $A_1 \dots A_n \rightarrow B_1 \dots B_m$ mit $2 \leq n \leq m, m > 2$ ersetzen wir durch:

$$A_1 A_2 \rightarrow B_1 D_1$$

$$D_1 A_2 A_1 \dots A_n \rightarrow B_2 \dots B_m$$

5.4 Beispiele

- $L_1 = \{a^n b^n c^n \mid n \geq 1\}$ (drei und mehr gleiche Exponenten)
- $L_2 = \{a^{n^2} \mid n \in \mathbb{N}^+\}$

3

Berechenbarkeitstheorie und Komplexität

1: REKURSIV AUFZÄHLBARE SPRACHEN

Typ-0 = (r.e.)

Eine Turingmaschine M akzeptiert die Sprache L , wenn sie nach endlicher Zeit stoppt und JA ausgibt. Bei Eingaben, die nicht zu L gehören, rechnet sie unendlich lang weiter. Diese Sprachen L gehören dann zu (r.e.)

1.1 Sätze zu (r.e.)

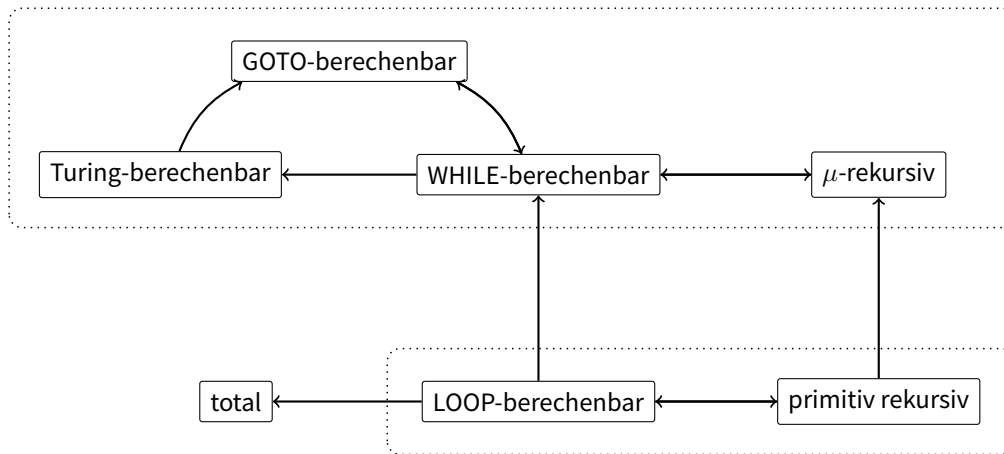
- Die Klasse der Typ-0 Sprachen ist abgeschlossen unter Sternoperation, Vereinigung, Schnitt und Konkatenation.
- Die Klasse der durch Turingmaschinen erkennbaren Sprachen ist gleich der Klasse der rekursiv aufzählbaren.

2: ENTSCHEIDBARE SPRACHEN

$$(\text{REC}) \subset (\text{r.e.})$$

Eine Turingmaschine *entscheidet* die Sprache L , wenn sie für jede Eingabe nach endlicher Zeit stoppt und JA oder NEIN ausgibt, je nachdem ob $w \in L$ oder nicht. Diese Sprachen L gehören dann zur Klasse der entscheidbaren Sprachen: (REC).

3: BERECHENBARKEITSTHEORIE



3.1 LOOP-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt LOOP-berechenbar, falls es ein LOOP-Program P gibt, das gestartet auf der Eingabe n_1, n_2, \dots, n_k in den Variablen x_1, x_2, \dots, x_n nach endlich vielen Schritten hält und die Variable x_0 den Wert $f(n_1, \dots, n_k)$ beinhaltet.

3.1.1 Erlaubte Basisanweisungen

- $x_i := x_j + c$ bzw. $x_i := x_j - c$ mit $c \in \mathbb{N}$
- LOOP x_i DO P END
- Hintereinanderausführung

3.1.2 Simulierbare Makros

- Wertzuweisungen $x_i := x_j$ und $x_i := c$
- IF $x_i > c$ THEN P END
- Übliche Arithmetische Operationen (Multiplikation und Division), sogar mod

3.2 WHILE-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt WHILE-berechenbar, falls es ein WHILE-Program P gibt, das gestartet auf der Eingabe n_1, n_2, \dots, n_k in den Variablen x_1, x_2, \dots, x_n nach endlich vielen Schritten hält (falls das Ergebnis definiert ist) und die Variable x_0 den Wert $f(n_1, \dots, n_k)$ beinhaltet. Ist $f(n_1, \dots, n_k)$ undefiniert, so hält P nicht.

3.2.1 Erlaubte Anweisungen

- In WHILE-Programmen sind alle Anweisungen von LOOP-Programmen erlaubt sowie die WHILE-Schleife:
- WHILE $x_i \neq 0$ DO P END

3.3 GOTO-Berechenbarkeit

3.3.1 Erlaubte Anweisungen

- Berechnungen und Zuweisungen: $x_i := x_j \pm c$
- Marken: M_i
- GOTO M_i
- HALT
- IF $x_i = c$ THEN GOTO M_j

3.4 Turing-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt Turing-berechenbar, falls eine deterministische Turingmaschine existiert, die $f(n_1, \dots, n_k) = m$ berechnet indem, sie gestartet auf dem k -Tupel (n_1, \dots, n_k) nach endlich vielen Berechnungsschritten einen Endzustand erreicht und dann m auf dem Band steht. Falls das Ergebnis für die Eingabe undefiniert ist, terminiert die Maschine nie.

3.5 Primitive Rekursion

Eine Funktion ist genau dann primitiv rekursiv, wenn sie LOOP-berechenbar ist.

- Konstante Funktionen sind primitiv rekursiv
- Projektionen sind primitiv rekursiv
- Die Nachfolgerfunktion $s : \mathbb{N} \rightarrow \mathbb{N}, n \mapsto n + 1$ ist primitiv rekursiv
- Verkettungen von primitiv rekursiven Funktionen sind primitiv rekursiv

- Funktionen, die durch primitive Rekursion aus primitiv rekursiven Funktionen entstehen, sind primitiv rekursiv:

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

Das heißt, g, h primitiv rekursiv $\Rightarrow f$ primitiv rekursiv.

3.5.1 Beispiele für primitiv rekursive Funktionen:

- Cantor'sche Paarungsfunktion

$$c(x, y) = \binom{x+y+1}{2} + x \quad (\text{bijektiv})$$

mit den Umkehrungsfunktionen

$$e(c(x, y)) = x \text{ und } f(c(x, y)) = y$$

•

3.6 μ -Rekursion

$$\mu f(x_1, \dots, x_k) = \min \{n \in \mathbb{N} \mid f(n, x_1, \dots, x_k) = 0 \wedge \forall m < n : f(m, x_1, \dots, x_k) > 0\}$$

Der μ -Operator liefert den kleinsten Eingabewert n der ersten Variable, bei der die Funktion 0 ausgibt. Dabei muss insbesondere f für alle Werte kleiner als n definiert sein, da sonst eine Berechnung nicht möglich wäre.

Satz 3.1: Satz von Kleene

Sei $f : \mathbb{N}^n \rightarrow \mathbb{N}$ eine μ -rekursive Funktion. Dann existieren zwei $(n+1)$ -stellige primitiv rekursive Funktionen p und q mit

$$f(x_1, \dots, x_n) = p(x_1, \dots, x_n, \mu q(x_1, \dots, x_n))$$

4: ENTSCHEIDBARKEITSTHEORIE