Inhaltsverzeichnis

Teil I

Automaten, Reguläre Ausdrücke und Grammatiken

1 Endliche Automaten

Ein *endlicher* Automat ist ein mathematisches Modell eines Systems mit Ein- und Ausgaben. Ein solches System befindet sich immer in einer aus einer endlichen Anzahl möglichen internen Konfigurationen (Zustände).

Hierbei unterscheidet, man zwischen *deterministischen* und *nicht deterministischen* Automaten. Der deterministische Automat ist so aufgebaut, daß für jeden Zustand genau festgelegt ist, welcher Zustand ihm bei Einlesen eines Eingabesymbols folgt. Beim nichtdeterministischen Automaten können mehrere mögliche Folgezustände auf einen Zustand existieren.

1.1 derministische Automaten

Der deterministische endliche[1] Automat wird formal dargestellt als:

$$A = (\Sigma, S, \delta, s_0, F)$$

Dabei gilt:

Σ ist das Eingabealphabet, also die Menge aller gültigen Eingaben

S bezeichnet die Zustandsmenge des Automaten A. Diese beschreibt alle gültigen Zustände, die der Automat annehmen kann.

δ ist die Zustandsüberführungsfunktion des Automaten mit $\delta : S \times \Sigma = S$. In der Zustandüberfürungsmenge *kann* für jeden Zustand in Kombination mit jedem Eingabesymbol ein Eintrag definert sein. In der Regel besteht die Menge der Zustandsüberführungen jedoch nur aus einer Teilmenge dieses Kreuzproduktes. δ muß also nicht für alle Paare $(s, a) \in S \times \Sigma$ definiert sein.

s_0 ist der Startzustand des Automaten. s_0 ist in der Zustandsmenge des Automaten enthalten da es auch ein Zustand ist: $s_0 \in S$

F (von *Final*?) ist die Menge der Endzustände des Automaten, wobei gilt $F \subseteq S$. Ein Automat kann also beliebig viele Endzustände besitzen. In einigen Fällen kann er sogar so aufgebaut sein, daß alle Zustände gleichzeitig Endzustände sind.

Der Automat heißt Deterministisch, wenn für jeden beliebigen Zustand in Kombination mit jedem Eingabesymbol höchstens ein Folgezustand existiert. Die Klasse der Deterministischen Automaten wird mit DFA_Σ [2] bezeichnet.

[1] endlich = mit einer endlichen Menge von Zuständen
[2] Deterministic Finite Automata

1.2 Definitionen

Wenn die Zustandsübergangsfunktion δ ein Paar *(s,a)* auf *genau einem* Folgezustand abbildet, spricht man auch von einem deterministischem Automaten (*DEA*). Es existieren keine Verzweigungen im Automaten. Die Verarbeitung verläuft gradlinig, da für jedem Punkt genau ein Folgezustand definiert ist.

Da der Automat nicht verschiedene Möglichkeiten für den Übergang zu einem Folgezustand überprüfen muß und eventuell an einen früheren Verarbeitungsschritt zurückspringen muß (*backtracing*) kann ein deterministischer Automat sehr schnell ermitteln ob ein Eingabewort akzeptiert wird oder nicht.

Wörter und Alphabete

Ein endlicher Automat verarbeitet *Wörter* (endliche Zeichenfolgen), die aus atomaren Symbolen gebildet sind. Der endliche Automat hat nach dem Lesen eines Eingabewortes einen anderen Zustand erreicht. *Wenn der Automat eine endliche Menge von erreichbaren Zuständen besitzt, wird er endlicher Automat genannt.*

Die Symbole, aus denen die Eingabewörter bestehen können werden als das Alphabet Σ bezeichnet. Das allgemeine Alphabet wird in symbolischen Darstellungen definiert durch:

$$\Sigma = \{a_1, a_2, ..., a_n\}, n \geq 0$$

Die Wortbildung erfolgt durch Anneinanderreihung von einzelnen Symbole oder auch einzelner vorher definierter Wörter. Die Menge aller Wörter, die über dem Alphabet gebildet werden können, ist gekennzeichnet als: Σ^*. Hierfür gilt:

1. Jeder Buchstabe $a \in \Sigma$ ist auch ein Wort über Σ, d.h. $a \in \Sigma^*$. *Jeder Buchstabe ist/kann auch ein Wort sein.*
2. Werden bereits konstruierte Wörter hintereinander geschrieben entsteht ein neues Wort. $(v, w \in \Sigma^* \Rightarrow vw \in \Sigma^*)$
3. ϵ, das leere Wort ist ein Wort über jedem Alpahbet und daher in jedem Alphabet enthalten.

Mit Σ^+ wird die Menge aller Wörter ohne das leere Wort ϵ bezeichnet.

Sprache

Eine (formale) Sprache ist eine Untermenge aller Wörter, die aus einem Alpahbet gebildet werden können: $L \subseteq \Sigma^{[*]}$. Diese Sprache heißt *regulär* falls es einen endlichen Automaten A gibt, der sie akzeptiert. Die Klasse der von endlichen Automaten akzeptierten Sprachen über Σ wird mit DFA_Σ[3] symbolisiert. Die Klasse der Regulären Sprachen über Σ wird als REG_Σ bezeichnet.

[3]Deterministic Finite Automata

Zustände

Ein Automat besitzt zwei Speicher. Der Eingabespeicher enthält das jeweilig eingegebene Wort. Der andere Speicher ist der Zustandsspeicher, der den jeweiligen Zustand des Automaten enthält. Da nur endlich viele Zustände zugelassen werden ist der Automat ein endlicher Automat. Formal wird der endliche Automat mit dem Buchstaben S dargestellt. Jeder Automat besitzt einen definierten Anfangszustand *start*. Die möglichen Endzustände des Automaten werden durch die Menge F bezeichnet, wobei gilt: $F \subseteq S$.

Die Zustandsüberführungstabelle zeigt, welchen Zustand ein Automat ausgehend vom Ausgangszustand beim Lesen eines Eingangsbuchstabens annimmt.

	Zustand vorher		
δ	s_0	s_1	s_2
1	s_1	s_2	s_2
0	s_2	s_1	s_2

Zustandsübergänge

Der Automat geht in Abhängigkeit von seinem aktuellen Zustand und dem nächsten zu verarbeitenden Symbol in einen neuen Zustand über. Diese Verhaltensweise, das "Programm" wird durch die Zustandsüberführungsfunktion δ festgelegt. Ein Zustandsübergang wird formal ausgedrückt als:

$$\delta(s, a) = s'$$

Dieses besagt, daß der Automat wenn er sich im Zustand s befindet und das Eingabesymbol a liest, in den Zustand s' übergeht.

Die Programmanweisungen der Menge δ stellen dar wie ein Zustandsübergang zu erfolgen hat. Die Zustandsübergangsfunktion oder Programmanweisung läßt sich verallgemeinert und formal dann darstellen als:

$$\delta = \{(s, a, s') | \delta(s, a) = s'\}$$

Der Automat befindet sich im Zustand s und liest das Eingabesymbol a. Gemäß der Zustandsüberführungsfunktion δ geht er jetzt in den Zustand s' über. Das Tripel (s, a, s') ist also eine Programmzeile des Programms.

Konfigurationsübergänge

Eine Konfiguration k_a beschreibt den *aktuellen Stand der Verarbeitung* eines Wortes durch den Automaten (A). Der Konfigurationsübergang wird zum einen festgelegt durch den aktuellen Zustand des Automaten s, zum anderen durch das noch zu verarbeitende Suffix des Eingabewortes v.

Die Konfiguration beschreibt also einen Zustand beliebigen Zustand während des Programmlaufs.

Ein Übergang von einer Konfiguration $k = (s, aw)$[4] zu einer anderen $k' = (s', w)$ kann stattfinden, falls der Zustandsübergang $\delta(s, a) = s'$ existiert. Für den

[4] aw = aktuellesEingabesymbol + Rest des Eingabewortes.

Konfigurationsübergang muß also ein passender Zustandsübergang vom alten in den neuen Zustand definiert sein. Formal wird der Konfigurationsübergang wie folgt ausgedrückt:

$$(s, w) \vdash (s', w')$$

1.3 nichtdeterministische Automaten

Bei den Nichtdeterministischen Automaten ist die Zustandüberführung als Relation aufgebaut. Das heißt, daß für jeden Zustand in Verbindung mit jedem gelesenen Eingabesymbol mehrere Folgezustände existieren können. Beim Abarbeiten eines Eingabewortes muß ein Automat dieser Klasse daher an "Verzweigungen" mit Hilfe von *Trial-And-Error* überprüfen, ob ein Zustandsübergang letztendlich zu einem Endzustand führt. Falls dieser Zustandübergang nach einigen Schritten dazu führt, daß für eine Kombination aus Eingabesymbol und aktuellem Zustand kein Folgezustand definiert ist, muß der Automat wieder an die letzte Zustandsüberführung zurückspringen, an der mehrere Möglichkeiten des weiteren Vorgehens bestanden. Mit steigender Zahl von möglichen Zustandüberführungen steigt die Laufzeit dieses Automaten bei Überprüfung eines Wortes exponentiell an. Die Klasse der nichtdeterministischen Automaten wird mit NFA_Σ[5] symbolisiert.

1.4 Äquivalenz deterministischer und nichtdeterministischer Automaten

Zu jedem nichtdeterministischen Automaten läßt sich ein äquivalenter deterministischer Automat definieren, der genau dieselbe Sprache akzeptiert. Die Überführung eines nichtdeterministischen Automaten in den äquivalenten deterministischen kann mit Hilfe der *Potenzmengenkonstruktion* durchgeführt werden. Die Grundidee dieser Überführung ist, daß die nichtdeterministischen parallelen Übergänge zu einer Berechnung und somit zu einem Übergang zusammengefaßt werden. Die Zustände bestehen dann aus Mengen von Zuständen. Die Menge aller möglichen Zustände des deterministischen Automaten ist die Menge aller Teilmengen von S, der Zustandsmenge.

Aufgrund dieser Äquivalenz der beiden Klassen von Automaten akzeptiern beide auch dieselbe Klasse von Sprachen, die Klasse der regulären *Sprachen.*

1.5 Epsilonautomaten

Der ϵ- Automat ist ein Automat, in dessen Zustandsüberführungsrelation auch ϵ-Übergänge vorkommen dürfen. Der ϵ-Übergang ist dadurch gekennzeichnet, daß der Automat von einem Zustand in einen anderen wechselt, ohne daß dabei ein Eingabesymbol gelesen wurde. Der Lesekopf bleibt beim ϵ-Übergang also stehen.

Ein nichtdeterministischer Automat ist genau betrachtet ein ϵ-Automat, bei dem die ϵ-Übergänge fehlen. Daher ist die Klasse der Nichtdeterministischen

[5] Non deterministic Finite Automata

Automaten eine Untermenge der ϵ-Automaten. Daher läßt sich jeder ϵ-Automat in einen Nichtdeterministischen transformieren. Gerade bei der modularen Zusammensetzung von endlichen Automaten sind die ϵ-Übergänge sehr hilfreich. Hiermit lassen sich die einzelnen Automatenmodule, die ja jeder für sich abgeschlossene Automten sind, einfach miteinander verbinden. Beim Übergang vom einen Automaten in den anderen wird hier einfach ein ϵ-Übergang durchgeführt. Mit Hilfe dieser Möglichkeiten lassen sich daher Sprachen miteinander verknüpfen.

1.6 Verallgemeinerte Automaten

Zur Vereinfachung von Automaten mit einer Folge von deterministischen Übergängen können die verallgemeinerten Automaten genutzt werden. Hier wird der Automat dahingehend erweitert, daß er für einen Konfigurationsübergang anstatt eines Symbols ein komplettes Wort akzeptiert. Mit den verallgemeinerten Automaten läßt sich eine vereinfachte Darstellung eines Automaten realisieren, da hier ein Teil der Konfigurationsübergänge zu jeweils einem einzigen zusammengefaßt wurde. Es läßt sich ein Beweis führen, daß für jeden verallgemeinerten Automaten ein äquivalenter endlicher Automat existiert. Ein verallgemeinerter endlicher Automat wird mit GFA_Σ[6] bezeichnet.

1.7 Zusammenfassung

Endliche Automaten modellieren ein Black-Box-Modell. In Abhängigkeit ihres aktuellen Zustands und einer Eingabe gehen sie gemäßeiner festgelegten Zustandsüberführung in einen neuen Zustand über. Die Abarbeitung eines Eingabewortes, einer endlichen Folge von Eingabesymbolen, geschieht durch einen Prozeß, der durch eine Folge von Konfigurationsübergängen formal beschrieben werden kann. *Ist der Automat nach vollständiger Abarbeitung des Eingabewortes in einem Endzustand, dann akzeptiert der Automat dieses Wort.* Kann er ein Eingabewort nicht komplett abarbeiten oder ist nach seiner Abarbeitung nicht in einem Endzustand, akzeptiert er das Wort nicht. *Die Menge aller von einem endlichen Automaten aktzeptierten Wörter stellt die von ihm akzeptierte Sprache dar.* Von endlichen Automaten akzeptierte Sprachen heißen reguläre Sprachen. Jeder endliche Automat kann in einen Automaten einer anderen Klasse transformiert werden:

$$REG_\Sigma = DFA_\Sigma = NFA_\Sigma = \epsilon FA_\Sigma = GFA_\Sigma$$

1.8 Minimierung endlicher Automaten

Zu jedem endlichen Automaten läßt sich ein äquivalenter minimaler Automat mit der mindestmöglichen Zahl von Zuständen konstruieren. Zwei minimalisierte Automaten unterscheiden sich höchstens in der Bezeichnung ihrer Zustände voneinander. Alle anderen Parameter wie

[6] Gneralized Finite Automata

- Eingabesymbole
- Anzahl der Zustände
- Anzahl der Endzustände
- Zustandsüberführungen

sind *gleich*. In diesem Falle besteht eine *Strukturgleichheit* zwischen den Automaten. In diesem Falle werden die Automaten auch isomorph zueinander genannt.

1.9 Ermittlung des minimalen Automaten

Die Minimierung eines Automaten erfolgt durch die Zusammenfassung der Zustandsübergänge, die für ein Eingabesymbol (Zeile in der Zustandsübergangstabelle) einen gemeinsamen Zielzustand erreichen. Hierzu werden die einzelnen Zustandsübergänge blockweise zusammengefaßt.

Die Zustandstafel wird zunächst in zwei große Blöcke aufgeteilt, wobei ein Block möglichst derart zusammengefaßt werden sollte, daß die Folgezustände für ein Eingabesymbol einen gemeinsamen Zielzustand(-sblock) anzeigen.
Falls in dieser Aufteilung die Zielzustände eines Blocks für ein Eingabesymbol (Zeile) noch keinen gemeinsamen Zielblock erreicht haben, muß einen erneute Blockaufspaltung erfolgen. Hierbei dürfen auch Spalten zusammengefaßt werden, die nicht direkt nebeneinander liegen.

Der minimale Automat ist erreicht, wenn alle Zielzustände für einen Block auf einen gemeinsamen Zielblock verweisen. Das Zustandsdiagramm des so entstandenen Automaten wird erzeugt, indem die Blöcke jeweils einen Zustand darstellen und die Zielzustände als Verweise auf die jeweiligen Zielblöcke eingezeichnet werden. (s.a. 19)

2 Reguläre Ausdrücke

Eine reguläre Sprache kann definiert werden, indem ein endlicher Automat konstruiert wird, der diese Sprache akzeptiert. Neben dieser akzeptierenden Definition kann die Sprache auch auf *beschreibende* Art und Weise definiert werden. Hierbei wird festgelegt, aus welchen Symbolen die Sprache zusammengesetzt ist und wie diese miteinander verknüpft werden dürfen. Die Definition der Sprache erfolgt mit Hilfe der *regulären Ausdrücke*.

beschreibendes Konzept

Ein regulärer Ausdruck, der über einem Alphabet Σ gebildet wird definiert im Prinzip eine Menge von Wörtern, die sich durch die Anwendung dieses Ausdrucks ergeben. Der Ausdruck definiert somit die Sprache L über dem Alphabet Σ. Wenn für α gilt: $\alpha \in REXP_{\Sigma}$ ist $L(\alpha)$ die von α definierte Sprache.

2.1 Reguläre Ausdrücke und Sprachen

Die Interpretation des reglulären Ausdrucks zur Erzeugung der Sprache geschieht nach festgelegten Vorschriften, den (Interpretations-)Vorschriften :

$L(\Lambda) = \emptyset$ Λ definiert die leere Sprache. Der reguläre Ausdruck Λ entspricht der leeren Sprache mit dem Symbol $\emptyset$. Im regulären Ausdruck bedeutet dieses das "nichts" (wie in 'kein- oder mehrmals'). Die leere Sprache ist eine Sprache, die eigentlich nicht existiert, sie enthält nicht einmal das leere Wort ϵ.
Die $\emptyset$ verhält sich bei dem |-Operator wie die 0 bei der Addition.

$L(E) = \{\epsilon\}$ E legt die Sprache fest, die nur das leere Wort enthält. Der reguläre Ausdruck E ist der Einsoperator. Er steht für die Sprache, die *nur* das leere Wort ϵ enthält.
ϵ verhält sich bezüglich des $\circ$-Operators wie die 1 bei der Multiplikation.

$L(a) = \{a\}$ für alle $a \in \Sigma$: jeder Buchstabe aus Σ beschreibt jeweils die Sprache, die nur ihn selbst als einziges Wort enthält.

Wie oben beschrieben können Sprachen beliebig miteinander verknüpft werden. Somit können die regulären Ausdrücke zur Erzeugung der Sprachen gleichfalls mit den Mengenoperationen verknüpft werden. Für diese Operationen lassen sich die folgenden Aussagen treffen.

Sind $\alpha, \beta \in REXP_\Sigma$, dann gilt:

$L(\alpha \bullet \beta) = L(\alpha) \circ L(\beta)$: Die Konketantion regulärer Ausdrücke ist die Konketanation der durch sie definierten Sprachen.

$L(\alpha \| \beta) = L(\alpha) \cup L(\beta)$: Die Selektion (ODER- Auswahl) reguläerer Ausdrücke ist die Vereinigung der durch sie definierten Sprachen.

$L(\alpha^{\otimes}) = (L(\alpha))^*)$: Der Wiederholungsparameter wird durch das Kleene-Stern-Produkt[7] interpretiert.

Der ||- Operator (Selektions bzw. ODER Operator) weist die Besonderheit auf, daß für Ihn die *Idempotenz* gilt. Das heißt daß für jeden regulären Ausdruck gilt:

$$(\alpha|\alpha) \equiv \alpha$$

denn

$$\begin{aligned} L(\alpha|\alpha) &= L(\alpha) \cup L(\alpha) \\ &= L(\alpha) \end{aligned}$$

Bei Zahlen $\neq 0$ gilt diese nicht.

[7] Das Kleene-Stern-Produkt L^* einer Sprache L ist die Vereinigung aller ihrer Potenzen $L^n, n \geq 0$:

$$L^* = \bigcup_{n \geq 0} L^n = L^0 \cup L^1 \cup L^2 \cup L^3 ...$$

Das Kleene-Stern-Produkt von L ohne das leere Wort wird mit L^+ gekennzeichnet, d.h. $L^+ = L^* - L^0$

2.2 Anwendung regulärer Ausdrücke

Reguläre Ausdrücke werden in der Softwaretechnik zur Beschreibung der Syntax von Eingabefeldern in Dialogmasken und Formularen benutzt. Die Software, die die Richtigkeit der Eingabe überprüft, kann mit Hilfe von Scanner-Generatoren automatisiert erzeugt werden. Hier müssen nur noch die Regulären Ausdrücke erstellt werden, die dann in Quellcode der Programmiersprache (i.d.R. C) umgesetzt werden.

Die von dem Generator erzeugten Prozeduren werden auch Scanner oder *Lexical Analyzer* genannt. Ein bekannter Scanner-Generatore ist z.B. `(f)lex` unter *Unix*.

3 Grammatiken

Neben den akzeptierenden und beschreibenden Konzepten für Sprachen, wie sie mit Hilfe von Automaten und regulären Ausdrücken erfolgen (s.a. 4), ist noch das Konzept der Sprach*erzeugung* mittels Grammtiken möglich. Eine Grammatik definiert Regeln (Produktionen), die angeben wie die Worte einer Sprache gebildet werden:

Satz ⟶ Subjekt Prädikat Objekt

Subjekt ⟶ Artikel Substantiv

Artikel ⟶ der — die — das

Substantiv ⟶ Junge — Mädchen — Kind — Hund

Prädikat ⟶ Verb

Verb ⟶ bellt — bellen — beißt — jagt — sucht

Objekt ⟶ Artikel1 Substantiv

Artikel1 ⟶ den — die — das

Die Symbole links vom ⟶ werden als *Variablen* bezeichnet, die durch die rechten Seiten der *Produktionen* ersetzt werden können. Zeichenketten aus Kleinbuchstaben sind *Terminale*, die nicht weiter ersetzt werden können. Alternativen werden durch senkrechte Striche — getrennt. Eine Variable (*Satz*) ist ausgezeichnet, bei ihr beginnt der Ersetzungsprozeß. Jetzt können beliebige Sätze aus den Worten gebildet werden, indem die Regeln angewandt werden, bis keine Variablen mehr vorhanden sind. Mit diesen Regeln können allerdings *auch* Sätze gebildet werden wie:
Die Junge bellt das Mädchen

Der Satz ist syntaktisch korrekt, also aus dem Startsymbol durch die Ableitung der Grammatikregeln hergeleitet. Allerdings ist die Bedeutung, die *Semantik* eher zweifelhaft. Diese Situation gleicht der bei einer Programmiersprache, wenn einer INT-Variablen z.B. ein FLOAT-Wert zugewiesen werden soll. Die Konstruktion ist nach den den Regeln der Grammatik korrekt.

Neben den semantischen Problemen besteht bei Grammatiken das Problem der *Eindeutigkeit*. Eindeutigkeit

3.1 Typ-3-Grammatiken

Typ-3-Grammatiken erzeugen Wörter indem sie ausgehend von einem Startsymbol mit Hilfe von Ersetzungsregeln die einzelnen Buchstaben durch Symbolersetzung erzeugen. Typ-3-Grammatiken sind untereinander äquivalente erzeugende Konzepte für reguläre Sprachen, äquivalent zu endlichen Automaten und regulären Ausdrücken. Mit einer Typ-3-Grammatik kann eine *reguläre* Sprache erzeugt werden. Eine *nicht reguläre* Sprache kann hiermit nicht erzeugt werden. Das heißt, daß z.B. bei der Erzeugung nicht kontrolliert werden kann ob bei einem Wort die Anzahl der a's und b's gleich ist.

Typ-3-Grammatiken hinsichtlich der Erzeugung bzw. überprüfung einer Sprache den endlichen Automaten oder den Regulären Ausdrücken äquivalent. Die Nonterminale einer Grammatik können auch als Zustände bezeichnet werden. Man merkt sich mit einem Nonterminal den aktuellen Erzeugungszustand des abzuleitenden Wortes, von dem aus in einem "Zustandsübergang" ein neues Symbol erzeugt wird. Ein Automat, der zu dieser Grammatik äquivalent ist, wird in diesem Falle das Symbol akzeptiert.

Die allgemeine Typ-3-Grammatik ist wie folgt aufgebaut:

$$\begin{aligned} G &= (\Sigma, N, P, S) \\ mit: & \\ \Sigma &: Terminalalphabet \\ N &: Nonterminalalphabet\ (\Sigma\ diskunkt\ zu\ N) \\ P &: Produktions- \ oder\ Regelmenge \\ S &: Startsymbol \end{aligned}$$

3.2 Anwendung der Grammatik

Die Erzeugung eines Wortes mit Hilfe der Typ3-Grammatik erfolgt, indem die in der Regelmenge P definierten Regeln ausgehend vom Startsymbol angewandt werden. Diesen Vorgang bezeichnet man auch als Ableitung der Grammatik. Er wird solange durchgeführt, bis das abgeleitete Wort jeweils kein Nonterminalsymbol (N)[8] mehr enthält. Pro Ersetzungsvorgang kann nur *ein* Nonterminal durch ein Terminalsymbol ersetzt werden. ein Symbol pro Ableitung

Beispiel (s.a. **??**)
Die folgende Typ3-Grammatik soll abgeleitet werden:

$$G = (\{0,1\}, \{S, A, B\}, P, S)$$

$$mit:$$

[8]Ein Nonterminalsymbol ist ein Symbol, daß wie eine Variable durch einen anderen Ausdruck ersetzt werden kann.

$$\begin{aligned} P \quad = \{ \quad & S \to 0S, S \to 1S, \\ & S \to 111A, \\ & A \to 0A, A \to 1A, A \to \epsilon\} \end{aligned}$$

1. S wird ersetzt durch $0S$, $1S$ oder $111A$
2. A wird ersetzt durch $0A$, $1A$ oder nichts ϵ. Bei Ersetzung mit ϵ endet die Ableitung, da kein Nonterminalsymbol mehr vorhanden ist.

Im obigen Falle werden Wörter erzeugt, in denen 111 als Infix vorkommt. Die Ersetzung $S \to 0S$ bzw. $S \to 1S$ können beliebig lange kombinationen erzeugen. Es ist auf jeden Fall garantiert, daß irgendwann die Ersetzung $111A$ durchlaufen wird und somit die Kombination 111 in dem Wort vorkommt.

Die Typ-3-Grammatiken lassen sich in rechtslineare und rechtslineare Grammatiken sowie ihre verallgemeinerten Varianten unterteilen. Die rechtslinearen Grammatiken erzeugen ein Wort buchstabenweise von links nach rechts. Hier ist jede Ableitungsregel so aufgebaut, daß in jedem Ableitunsschritt rechts am schon bestehenden Wort ein neuer Buchstabe angehängt wird. Bei den linkslinearen Grammatiken ist dieses umgekehrt.

Die *verallgemeinerten* Typ-3-Grammatiken sind so aufgebaut, daß hier in jedem Ableitungsschritt nicht nur ein Terminalbuchstabe, sondern sofort ein ganzes Terminalwort erzeugt wird.

Mit einer Typ-3-Grammatik läßt sich eine Sprache der Form $L = \{L = a^n b^n | n \geq 0\}$ nicht ableiten, da diese Sprache nicht regulär ist.

4 Eigenschaften Regulärer Sprachen

Die Konzepte zur Beschreibung regulärer Sprachen sind:

- *akzeptierende Konzepte:* endliche Automaten, nichtdeterministische endliche Automaten, Automaten mit ϵ-Übergängen, verallgemeinerte endliche Automaten
  Die Sprache ist eine reguläre Sprache, wenn sie von einem (endlichen) Automaten akzeptiert wird.
- *beschreibendes Konzept:* reguläre Ausdrücke;
  Die reguläre Sprache wird durch einen regulären Ausdruck beschrieben. Dieser legt sozusagen den Aufbau der einzelnen Wörter fest.
- *erzeugende Konzepte:* Typ-3-Grammatiken (rechtlineare, linkslineare, verallgemeinerte rechtslineare,verallgemeinerte linkslineare Grammatiken)
  Die Typ-3-Grammatik gibt Regeln an, mit denen aus einem Startwert alle Wörter einer Sprache hergestellt werden können.

4.1 Abschlußeigenschaften

Die Klasse REG_Σ der regulären Sprachen über einem Alphabet ist abgeschlossen gegenüber allen gängigen Operationen. Das bedeutet, daß die Verknüpfung von zwei Sprachen mit Hilfe dieser Operationen als Ergebnis wieder eine reguläre Sprache ergibt.

Die hier gängigen Operationen auf die regulären Sprachen sind:

- Die Vereinigung von zwei regulären Sprachen ($L_1 \cup L_2 \in REG_\Sigma$),
- Die Differenz zweier regulärer Sprachen ($L_1 - L_2 \in REG_\Sigma$),
- Das Komplement einer regulären Sprache ist wieder regulär ($\Sigma^* - L \in REG_\Sigma$),
- Der Durchschnitt regulärer Sprachen ist wieder regulär ($L_1 \cap L_2 \in REG_\Sigma$),
- Die Konkatenation regulärer Sprachen ist regulär ($L_1 \circ L_2 \in REG_\Sigma$),
- das Kleene-Stern-Produkt einer regulären Sprache ist regulär ($L^* \in REG_\Sigma$)[9] interpretiert.,
- Die Spiegelung einer regulären Sprache ist regulär ($SP(L) \in REG_\Sigma$).

4.2 Das Pumping-Lemma für reguläre Sprachen

Die Frage, ob eine Sprache regulär ist, ob sie also durch einen regulären Ausdruck beschrieben, bzw. durch einenn *endlichen* Automaten erkannt werden kann, kann man mit Hilfe des Pumping-Lemmas beantworten[10]. Das Pumping-Lemma gibt eine notwendige Eigenschaft für reguläre Sprachen an. Das heißt jede reguläre Sprache *muß* das Pumping-Lemma erfüllen. Allerdings ist diese Eigenschaft nicht hinreichend, d.h. es gibt Sprachen die nicht regulär sind aber das Pumping-Lemma erfüllen.

notwendig, aber nicht hinreichend

4.3 Entscheidbarkeitsprobleme

In der Praxis des Compilerbaus ist ein Entscheidbarkeitsproblem von großer Bedeutung: das Wortproblem. Ein Quellcodeprogramm stellt ein Wort über einer Sprache, der Programmiersprache dar. Der Compiler muß entscheiden, ob dieses Programm ein gültiges Wort zu dem vorgegebenen Alphabet, daß alle möglichen syntaktisch gültigen Programme dieser Programmiersprache darstellt. Neben dem Wortproblem sind in der Praxis noch anderen Entscheidbarkeitsprobleme relevant. Für alle diese Probleme wurden passende Algorithmen entwickelt.

Wortproblem

[9] Das Kleene-Stern-Produkt L^* einer Sprache L ist die Vereinigung aller ihrer Potenzen $L^n, n \geq 0$:

$$L^* = \bigcup_{n \geq 0} L^n = L^0 \cup L^1 \cup L^2 \cup L^3 ...$$

Das Kleene-Stern-Produkt von L ohne das leere Wort wird mit L^+ gekennzeichnet, d.h. $L^+ = L^* - L^0$

[10] Beschreibung des Pumping-Lemma: s. S. 35

- das Wortproblem ($w \in L$),
- das Leerheitsproblem ($L = \emptyset$),
- das Endlichkeitsproblem ($|L| < \infty$),
- das Durchschnittsproblem ($L_1 \cap L_2 = \emptyset$),
- das Äquivalenzproblem ($L_1 = L_2$).

4.4 Nichtreguläre Sprachen und Grenzen endlicher Automaten

Eine reguläre Sprache wird dadurch definiert, daß es einen endlichen Automaten gibt, der sie akzeptiert. Dieser Automat ist durch seinen endlichen Speicher begrenzt, der ihm nur eine endliche Menge von Zuständen erlaubt. Eine nichtreguläre Sprache ist dadurch gekennzeichnet, daß sie von keinem regulären Automaten akzeptiert wird.

Eine sehr einfache Sprache, die dieser Bedingung genügt ist die Sprache:

$$L = \{a^k b^k | k \geq 0\}$$

Diese Sprache muß hat u.a. die Eigenschaft, daß der hintere Teil ihrer Wörter im Aufbau vom vorderen Teil abhängt. Für diese Sprache läßt sich zeigen, daß sie das Pumping-Lemma nicht erfüllt, sie somit nicht regulär sein kann, da das Erfüllen des Pumping-Lemma eine notwendige Eigenschaft ist.

In Programmiersprachen sind geschachtelte Blockstrukturen möglich. Diese lassen sich auf die Form $L = \{a^k b^k | k \geq 0\}$ verallgemeinern. Eine Sprache dieser Form ist aber nicht regulär, so daß Programmiersprachen nicht mit Typ-3-Grammatiken erzeugt werden können bzw. durch Compiler in Form von endlichen Automaten verarbeitet werden können.

5 Endliche Maschinen

Ein Automat besizt keine Ausgabefunktion. Der akutelle Zustand oder ein Ergebnis kann also nicht sichbar gemacht werden. Wenn ein Automat, mit einer Ausgabefunktion versehen wird, bezeichnet man ihn als Maschine. Genau wie bei den möglichen Eingaben wird die Menge der Möglichen Ausgaben in einem Alphabet zusammengefaßt. Wir sprechen dann vom Ausgabealphabet Δ. Eine Maschine ist also definiert durch:

$$\begin{aligned} M &= (\Sigma, \Delta, S, \delta, \lambda, s_0) \\ mit & \\ \Sigma &: Eingabealphabet \\ \Delta &: Ausgabealphabet \\ S &: Zustandsmenge \\ \delta &: Zustandsüberführungsfunktion \end{aligned}$$

λ : $Ausgabefunktion$
s_0 : $Startzustand$

5.1 Maschinentypen

Mealy- Maschinen

Die Mealy-Maschine ist dadurch gekennzeichnet, daß die Ausgabe den *Zustandsübergängen* zugeordnet ist. Damit hängt die Ausgabe der Mealymaschine vom *jeweiligen Eingabesymbol und vom aktuellen Zustand* ab. Da die Ausgabe mit jedem Zustandsübergang erfolgt, sind die Ein- und Ausgabewörter bei der Mealy-Maschine immer gleich lang. Nach dem Einschalten liegt bei dieser Maschine noch keine Ausgabe vor, da sie sich im Startzustand befindet und noch kein Zustandsübergang stattgefunden hat.

Mealy- Maschine

Simulation von Mealy- Maschinen durch Automaten Ein endlicher Automat ist im gewissen Sinne äquivalent zu einer Mealy-Maschine. Zu einer Mealy-Maschine läßt sich immer ein Automat entwickeln, der als Eingabewort die Kombination von Eingabe und Ausgabe der Mealy-Maschine akzeptiert.

Moore-Maschinen

Die Mealy-Maschine besitzt den Nachteil, daß sie bei Einschalten noch keine Ausgabe erzeugt, da sie sich im Startzustand befindet und noch kein Konfigurationsübergang stattgefunden hat. Um diesen Mangel zu beheben muß den Startzustand eine Ausgabe zugeordnet werden. Diese Vorgehensweise wird bei der Moore-Maschine angewandt. Hier ist die Ausgabe *nur* vom jeweiligen Zustand abhängig. Das Ausgabewort der Moore-Maschine ist *immer* um ein Symbol länger als das Eingabewort, da hier schon beim Einschalten eine Ausgabe erfolgt.

Eine Mealy-Maschine läßt sich in eine Moore-Maschine tranformieren und umgekehrt. Bei Umwandlung in die Moore-Maschine muß beachtet werden, daß eventuell neue Zustände (Zustandssplitting) eingeführt werden. Diese erzeugen je nach vorhergehenden Eingabesymbol eine andere Ausgabe. Bei der Mealy-Maschine erfolgt dieses systembedingt automatisch.

Zustandssplitting

Äquivalenz

Äquivalenz Die Mealy- und die Moore-Maschine sind nur in eingeschränktem Maße äquivalent zueinander. Aufgrund der unterschiedlichen Länge der Ausgabewörter ist eine Äquivalenz im engeren Sinne nicht gegeben. Falls die erste Ausgabe der Moore-Maschine nicht berücksichtigt wird, lassen sich die beiden Typen ineinander umwandeln. Hieraus läßt sich folgern, daß alle Mealy- berechenbaren Probleme auch Moore-berechenbar sind und umgekehrt.

Praktische Anwendungen Endliche Maschinen finden praktische Anwendungen z.B. bei der Lösung von Multiplikationen, Additionen von natürlichen

und von Dualzahlen, Paritätsbitergängzung und -prüfung sowie Teilbarkeitsteste oder den Eintrittsautomaten.

6 Zellulare Automaten

Automaten werden auch zur Modellierung von vernetzten Systemen eingesetzt, wie dieses z.B. in den Zellstrukturen des Gehirns oder anderen Nervenzellen zu finden ist. Der Zustand einer einzelnen Zelle hängt von den Zuständen ihrer direkten Nachbarzellen ab. Bei der Modellierung dieser Systeme wird jede einzelne Zelle als Automat dargestellt, dessen Zustandsänderung abhängig von den Nachbarzellen ist. Ein Eingabewort wird also aus den Zuständen der Nachbarzellen gebildet. In festgelegten Takten finden die Zustandsänderungen der Zellen *synchron* statt. Die Untersuchungen an diesem System finden in der Regeln an n-dimensionalen Gittern statt. Hierfür wird für die einzelne Zelle der sogenannte Nachbarschaftsindex aufgestellt, der in allgemeiner Beschreibung die Koordinaten der Nachbarzellen enthält, die für die Auswertung des Konfigurationsübergangs berücksichtigt werden müssen. Mit entsprechenden Algorithmen lassen sich dann Fragen untersuchen wie

n-dimensionale Gitter

- Treten Wiederholungen von Mustern und Teilmustern (nach welcher Anzahl von Perioden) auf?
- Wann wird die Konfigurationsfolge stationär?
- Können sich zellulare Automaten mit unterschiedlichen Nachbarschaftsindexen (unterschiedliche Dimensionen) gegenseitig simulieren?
- Welche Sprachklassen lassen sich mit einem zellularen Automaten simulieren?
- Welche Funktionenklassen können mit welchen Typen von zellularen Automaten berechnet werden, wenn die Automaten mit Ein- und Ausgabe versehen werden?

Die Konzepte zellularer Automaten werden in der Biologie bei der Beschreibung und Untersuchung von Zellwachstum oder Populationsmodellierung, in der Physik zur Analyse von Kristallwachstum und in der Informatik bei Entwicklung von Parallelrechnern eingesetzt.

Ein spezielles Problem ist das *Firing-Squad-Synchronization-Problem* . Im allgemein Fall wird hier untersucht, wie eine Reihe von Automaten, bei denen Nachrichten zum jeweiligen Nachbarn übermittelt werden können, so gesteuert werden können, daß alle Automaten nach einiger Zeit (=Anzahl von Takten) synchron den gleichen Zustand erreichen.

7 Petri-Netze

Die Petri-Netze dienen zur Simulation von miteinander kommunizierenden asynchron laufenden Prozessen. Ein einfaches Beispiel ist der konkurrierende Zugriff auf ein Peripheriegerät in einem Rechner mit mehreren Prozessoren.

Zum Aufbau eines Petri-Netzes wird zuerst die statische Sicht auf das System modelliert. Hier sind alle Systemzustände enthalten. Die "Laufrichtung" der Transitionen (Prozesse) wird durch Pfeile markiert werden. Zudem werden die Punkte miteinander verbunden, an denen die Prozesse miteinander kommunizieren können (sog. Semaphore). Die Kommunikation erfolgt an diesen Stellen mit Hilfe eines Tokens, welches von den Prozessen belegt, d.h. "mitgenommen" werden kann. Dieses entspricht dem Setzen eines Flags. Eine Regel bei der Modellierung ist, daß eine Transition nur *feuern* kann, wenn alle vorhergehenden Zustände mit mindestens einem Token belegt sind. Der jeweilige Prozeß nimmt das gemeinsame Token mit und signaliert hiermit die Belegung der Ressource.

Semaphore

Die Untersuchung dieses Systems im asynchron laufenden Zustand kann darüber Auskunft geben, ob das Gesamtsystem in einen Zustand laufen kann, der eine Problemlösung von außen erfordert. Beispiele für solche Zustände sind:

- Konfliktsituation: Da zwei Prozesse gleichzeitig das gleiche dürfen tritt eine Pattsituation auf. Diese muß von außen gelöst werden, indem bestimmt wird wer zuerst weiterlaufen soll.
- Verklemmung: Zwei Prozesse warten kreuzweise auf ein Ergebnis des jeweils anderen Prozesses.
- tote Transition: Hier werden Zustände im System gefunden, die ein "Hängen" eines einzelnen Prozesses feststellen können. Ein solcher Zustand kann z.B. durch eine unglückliche Modellierung des Systems auftreten, indem auf einen Zustand gewartet wird der nie eintreten kann.

Die Anwendungsgebiete von Petri-Netzen liegen in der Systemanalyse und im Entwurf. Softwarewerkzeuge (sog. CASE-Tools) können Analyse und Entwurf von Petri-Netzen unterstützen. Ein weiteres Anwendungsgebiet ist die Konzeption und Realisierung von Workflow-Managementsystemen.

CASE- Tools

8 Kontextfreie Grammatiken

Wie oben (3.1) gezeigt, kann eine Sprache der Form $L = \{a^n b^n | n \geq 0\}$ mit Hilfe einer Typ-3-Grammtik nicht erzeugt werden. Um diese Sprache zu erzeugen müssen *gleichzeitig* zwei Terminalsymbole[11] so parallel erzeugt werden[12], daß im nächsten Schritt wieder zwei Terminalsymbole parallel erzeugt werden können. Die Typ-3-Grammatik läßt höchstens die Erzeugung von *einem* Terminalsymbol pro Ersetzungsvorgang zu. Mit der Erweiterung der Möglichkeiten auf die Erzeugung von mehreren terminalen Symbolen pro Ersetzungsschritt ist eine solche Sprache definierbar.

mehrere terminale Symbole pro Ersetzung

Die Klasse der regulären Grammatiken (Typ-3-Grammatiken) ist eine Obermenge der Klasse der kontextfreien Grammatiken. Falls das Eingabealphabet Σ mehr als ein Symbol erhält ist die Klasse der kontextfreien Grammatiken eine *echte* Untermenge der regulären Grammatiken. Die kontextfreien Grammatiken zählen zu den *Typ-2-Grammatiken.*

[11] "Konstanten", nicht mehr ersetzbar

[12] In diesem Falle rechts- und linksseitig durch eine Regel wie aSb

Eine *kontextfreie Grammatik (kfG)* ist ein Viertupel:
(V,T,P,S)
mit: V = endliche Menge von *Variablen*
T = endliche Menge von *Terminalsymbolen*
P = endliche Menge von *Produktionen*
S = Startsymbol, $S \in V$

Eine Produktion definiert eine Ersetzungsregel und ist ein Paar (A,α), wobei A eine Variable, $a \in (V \cup T))^*$ ist. α *kann auch die leere Zeichenkette sein.* Die Elemente der Produktionen werden häufig als $A \longrightarrow \alpha$ beschrieben.

Da auf der linken Seite einer Produktion nur eine einzelne Variable stehen darf, *kann diese Variable* unabhängig vom Kontext ersetzt werden - *daher die Bezeichnung* kontextfreie Grammatik.

Die Klasse der *kontextfreien Grammatiken* haben für die Definition von Programmiersprachen eine sehr große Bedeutung. Diese Sprachen können durch kontextfreie Grammatiken mit Hilfe eines Formalismus beschrieben werden, der es erlaubt aus den Sprachbeschreibungen in vielen Fällen automatisch Parser zu generieren, die überprüfen können ob ein Wort zur Sprache gehört. Wenn dieses so ist, ist das Programm syntaktisch korrekt.

8.1 Vereinfachung kontextfreier Grammatiken

Die kontextfreien Grammatiken lassen sich mittels dreier Verfahren vereinfachen.

1. Eliminierung von ϵ- Regeln, so daß die Grammatiken ϵ- frei werden. Hierbei werden einige Regeln so zusammengefaßt, daß sie schon enden bevor sie auf ein ϵ- Symbol treffen.
2. Eliminierung nutzloser Symbole, die für die Erzeugung von terminalen Wörtern keine Rolle spielen.
3. Eliminierung von Kettenregeln, die keinen Beitrag zur Erzeugung von Wörtern liefern. Kettenregeln werden wo es geht zu einer Regel zusammengefaßt.

Chomsky-Normalform Eine Regel befindet sich in der Chomsky-Normalform, wenn ihre rechte Seite entweder aus genau zwei Nichtterminalen oder einem Terminalsymbol besteht. Zu jeder kontextfreien Grammatik existiert eine Grammatik in Chomsky-Normalform.

Greibach-Normalform Eine Regel in der Greibach-Normalform besteht aus einem Terminalsymbol ganz links und n- Nonterminalsymbolen. Zu jeder kontextfreien Grammatik existiert eine Grammatik in Greibach-Normalform.

8.2 Eigenschaften kontextfreier Grammatiken

Aus einer kontextfreien Grammatik kann ein Ableitungsbaum gebildet werden. Ableitungsbaum

Hierbei wird jedes Nonterminal als Verzweigung gesehen, an dessen Ästen weitere Verzweigungen oder Terminalsymbole angeordnet sind. Sie spielen eine große Rolle im Compilerbau, da sie vom Compiler als innere Repräsentation eines Programmes aufgebaut werden. Sie stellen die Grundlage zur Erzeugung des Objektcodes des Programmes dar. Aus dem Ableitungsbaum wird ein *Termbaum* entwickelt, indem die Verknüpfungssymbole auf die Verzweigungen rutschen, so daß der ganze Baum nur noch aus Terminalen besteht. Der Termbaum kann dann relativ einfach in eine Folge von Assemblerbefehlen transformiert werden.

Termbaum

Eine Grammatik kann Ableitungen enthalten, die mehrdeutig interpretiert werden können. Hier kann es bei der Auswertung Probleme geben, da Vorrangregeln[13] beachtet werden müssen. Beim Entwurf von Programmiersprachen müssen diese Probleme berücksichtigt werden. Die Grammatiken zur Definition der Syntax der Sprache muß so aufgebaut sein, daß nur Eindeutige und korrekte Ergebnisse entstehen können. Eine Grammatik mit mehrdeutigen Aussagen läßt sich in eine eindeutige Grammatik überführen.

8.3 Pumping-Lemma

Für die Klasse der kontextfreien Sprachen kann ebenso wie für die Regulären Sprachen ein Pumping-Lemma genutzt werden um zu entscheiden ob die Wörter dieser Sprache *nicht* kontextfrei sind. Allerdings kann nicht festgestellt werden ob diese Sprache kontextfrei ist.

8.4 Abschlußeigenschaften

Die Klasse der kontextfreien Sprachen ist nicht gegeüber allen gängigen Operationen abgeschlossen[14].

Abgeschlossen sind kontextfreie Sprachen gegenüber

- Vereinigung
- Konkatenation
- Kleene-Stern-Produkt
- Spiegelung

Nicht abgeschlossen sind sie gegenüber

- Druchschnitt
- Komplementbildung

[13] wie z.B. Punktrechnung vor Strichrechnung

[14] Die Anwendung der Operation auf diese Sprache erzeugt eine neue Sprache, die wiederum kontextfrei ist.

9 Erweiterte Baccus-Naur-Form

Eine Darstellung von kontextfreien Grammatiken läßt sich auch mit der erweiterten Baccus-Naur-Form (EBNF) erreichen. Eine Grammatik in Baccus-Naur-Form ist folgendermaßen aufgebaut:

$$G = (\Sigma, N, P, S) \tag{1}$$

Hier ist $S \in N$ das Startsymbol und P eine endliche Menge von erweiterten Baccus-Naur-Regeln darstellt. Die Symbolik der Regeln beinhaltet auch Symbole wie $|$, oder $\{\}_0^1$ und $\star$, die wie bei den regulären Ausdrücken interpretiert werden. Eine Ableitung bei der ein Nonterminal durch ein Terminalsymbol ersetzt wird, wird auch *Expansion* genannt. Eine Ableitung bei der ein Nonterminal durch ein anderes ersetzt wird nennt sich *Reduktion*.

Expansion
Reduktion

Die Klasse der Sprachen, die mit Erweiterten Baccus-Naur-Grammatiken erzeugt werden können ist genau die Klasse der kontextfreien Sprachen.

9.1 Syntaxdiagramme

Aus Grammatiken lassen sich Syntaxdiagramme erstellen. Diese werden in der Praxis sehr verbreitet verwendet um die Syntax von formalen Sprachen anschaulich darzustellen. Syntaxdiagramm stellen mit Hilfe von Pfeilen und Quadratischen Symbolen einen *Fluß* durch die Grammatik dar. Hierbei wird einen Alternative von Ersetzungen dargestellt indem mehrere Kästen parallel geschaltet werden. Eine Regel der Form $\{\}^*$ wird mit antiparallelen Pfeilen über dem Symbol dargestellt.

10 Reguläre Definition

Eine reguläre Definition ist eine kontextfreie Grammatik, die einen regulären Ausdruck erzeugt. Somit erzeugt eine reguläre Definition indirekt eine Sprache. Diese wird durch den von der regulären Definition erzeugten regulären Ausdruck erzeugt.

11 Kellerautomaten

Der Kellerautomat ist ein Automat mit einem Stackspeicher, in dem ein Symbol abhängig vom aktuellen Zustand und von der Eingabe abgelegt werden kann. Die formale Definition des Kellerautomaten lautet:

$$K = (\Sigma, S, \Gamma, \delta, s_0, \bot, F)$$

mit den Elementen:

$$\begin{aligned} \Sigma &= \text{Eingabealphabet} \\ S &= \text{endliche Zustandsmenge} \\ \Gamma &= \text{Kelleralphabet} \end{aligned}$$

$$
\begin{aligned}
\delta &= \text{Zustandsüberführung} \\
s_0 &= \text{Startzustand} \\
\bot &= \text{Bottomsymbol} \\
F &= \text{Endzustandsmenge}
\end{aligned}
$$

Die Zustandsüberführung ist jetzt gegenüber der beim einfachen Automaten erweitert worden, da der Keller oder Stack noch bearbeitet wird. Die Speicherung auf dem Stack erfolgt in der Weise, daß *das aktuelle Symbol vom Stack entfernt wird und ein neues Symbol gemäß Zustandsüberführung auf denselben gelegt wird.* Der *push*-Befehl eines Kellerspeichers ist also aus vorhergehendem *pop* mit nachfolgendem *push* zusammengesetzt.

Die Zustandsüberführungsfunktion des Kellerautomaten ist allgemein aufgebaut aus:

$$
\begin{aligned}
\delta &= (s_x, a_\Sigma, s_{Keller}, s_f, s_{Keller,neu}) \\
mit: & \\
s_x &= \text{aktueller Zustand des Automaten} \\
a_\Sigma &= \text{Eingabesymbol} \\
s_{Keller} &= \text{aktuelles Kellersymbol} \\
s_f &= \text{Folgezustand des Automaten} \\
s_{Keller,neu} &= \text{neues Kellersymbol}
\end{aligned}
$$

Beim Übergang in den Endzustand kann das Kellerbottomsymbol gelöscht werden. Dieses ist für die Akzeptanz eines Wortes allerdings nicht notwendig. Das gänzliche Leeren des Kellers hätte in obigem Beispiel schon ausgereicht. Das Erreichen eines Endzustandes seitens des Automaten ist in diesem Falle gar nicht erforderlich, so daß die Menge F der Endzustände die leere Menge oder ganz weggelassen werden könnte. Die Klasse der Sprachen, die durch einen Kellerautomaten mit leerem Keller akzeptiert werden ist gleich der Klasse der Sprachen, die durch einen Kellerautomaten mit Endzustand akzeptiert werden.

Transformation

Zu jedem Kellerautomaten läßt sich eine kontextfreie Grammatik konstruieren und umgekehrt. Jede kontextfreie Sprache kann durch einen Kellerautomaten mit nur einem Zustand (mit leerem Keller) akzeptiert werden. *Es gibt also auch bei den Typ-2-Sprachen äquivalente generierende (Grammatiken) und akzeptierende ((Keller-)Automaten) Konzepte.*

Ein allgemeines Verfahren, der Parser-Generator kann zu einer kontextfreien Grammatik einen Kellerautomaten konstruieren, der genau die von der Grammatik erzeugte Sprache konstruiert. Beim Akzeptieren eines Wortes mit diesem Kellerautomaten kann ein Ableitungsbaum für dieses Wort erzeugt werden. In der Praxis werden Kellerautomaten für den Compilerbau und bei der Sprachdefinition verwendet, da sie Syntaxtests in linearer Zeit ermöglichen.

11.1 deterministische Kellerautomaten

Der nichtdeterministische (Keller-) Automat kann in der praktischen Verwendung im Compilerbau einige Probleme verursachen. Beim *nichtdetermi-*

nistischen Kellerautomaten können sich für einen Zustandsübergang mehrere Möglichkeiten bieten. Der Automat muß im Durchschitt die Hälfte der Möglichkeiten durchlaufen, um festzustellen welche der Möglichkeiten überhaupt "paßt". Beim Großteil der durchlaufenden Fälle erfolgt das *Backtracking*, der Automat muß an eine zuvor gespeicherte Position zurückspringen, da die aktuell durchlaufene Möglichkeit das jeweilige Wort nicht akzeptiert. Beim nichtdeterministischen Kellerautomaten kann das Erkennen von Wörtern daher exponentiell lange dauern. Ein Eingabewort der Länge n kann 2^n Schritte für seine Überprüfung durchlaufen. Aus diesem Grunde ist man an *deterministischen* Zustandsübergängen interessiert.

Aufzählbare und nicht abzählbare Mengen

Eine Menge heißt abzählbar, falls sie *endlich* ist, oder falls es eine bijektive Abbildung $f : M \rightarrow N_0$ gibt.

Jedem Element $m \in M$ muß eindeutig eine natürliche Zahl $f_{(m)}$ zugeordnet werden können. Alle Elemente aus M besitzen eine eindeutige Nummer, können also abgezählt werden.

> Die Menge $Q = \{\frac{p}{q} | p, q \in Z, q \neq 0\}$ ist abzählbar. Hier kann jedem beliebigen Bruch eine eindeutige Zahl zugeordnet werden.

Jede Teilmenge einer abzählbaren Menge ist abzählbar. Falls eine Menge abzählbar ist, sind zwangsläufig Untermengen hiervon.

11.2 Nicht-oder überabzählbare Mengen

Eine Menge ist überabzählbar, falls sie nicht eindeutig in einer bijektiven Abbildung zugeordnet werden kann.

Jede Obermenge einer nicht abzählbaren Menge ist nicht abzählbar.

> Die Menge der reellen Zahlen ist überabzählbar. Mit Hilfe der Diagonalisierung wird dieses bewiesen. Falls die Menge der reellen Zahlen im Intervall (0,1) der Menge $I = \{x \in R | 0 < x < 1\}$ überabzählbar ist, so muß dieses auch für alle reellen Zahlen gelten.
>
> Jedes $x \in I$ läßt sich als unendliche Dezimalzahl schreiben:
>
> $$x = 0, x_1 x_2 x_3 ... \tag{2}$$
>
> Falls I abzählbar ist, lassen sich alle Elemente von I abzählen:
>
> $$\begin{aligned} x_1 &= 0, x_{11} x_{12} x_{13} ... && (3)\\ x_2 &= 0, x_{21} x_{22} x_{23} ... && (4)\\ x_3 &= 0, x_{31} x_{32} x_{33} ... && (5)\\ & && (6) \end{aligned}$$

Jede Zahl von xnm kann also eindeutig einer Zahl aus N (natürliche Zahl, Aufzählung) zugeordnet werden. Es läßt sich eine unendliche Dezimalzahl x_d konstruieren:

$$x_d = 0, a_1 a_2 a_3 ... \quad (7)$$

Für die Elemente der Dezimalzahl soll folgendes gelten:

$$a_i = \begin{cases} 1, & falls \quad x_{ii} \neq 1 \\ 2, & falls \quad x_{ii} = 1 \end{cases} \text{, für alle i} \quad (8)$$

Nach der Regel ist die i-te Dezimalziffer von x_d 1 falls die i-te Ziffer der i-ten Zahl von (3) ungleich 1 ist. Falls sie ungleich 1 ist, ist sie gleich 2. Der Dezimalanteil der Zahl x_d besteht also nur aus Einsen und Zweien, die durch die *Diagonale* der Abzählung 3 bestimmt ist.

x_d ist eine belibige Zahl aus (3), so daß für $x_d = x_1$ die Regel (8) gelten muß. Dieses ist insofern ein Wiederspruch, da $a_i = x_{11} = 1$ falls $x_{11} \neq 1$. Bei $x_{ii} = 1$ muß $x_{ii} = 2$ sein, was ebenfalls einen Widerspruch ergibt.

12 Turingautomaten

Ein Turingautomat ist durch den wahlfreien Zugriff auf den Speicher gekennzeichnet. Als Arbeitsspeicher wird hier das Eingabeband genommen. Der Turingautomat ist definiert durch:

$$\begin{aligned} TA &= (\Sigma, S, \Gamma, delta, s_0, \#, F) \\ wobei & \end{aligned}$$

$$\begin{aligned} \Sigma &= Eingabealphabet \\ S &= \text{endliche Zustandsmenge} \\ \Gamma &= \text{Arbeitsalphabet (auch Bandalphabet)} \\ \delta &= \text{Zustandsüberführung oder Turingprogramm} \\ s_0 &= Startzustand \\ \# &= Blanksymbol \\ F &= Endzustandsmenge \end{aligned}$$

Wenn die Zustandsüberführung (Turingprogramm, δ) als Funtion dargestellt werden kann, heißt *TA deterministisch*. Falls es eine Relation (1:n) ist, heißt *TA nichtdeterministisch*. Das Turingprogramm wird geschrieben als *Anweisung*:

$$(s, a, s', b, m)$$

Die Anweisung bedeutet, daß *TA* sich vorher im Zustand s, und der Schreib-Lesekopf *unter dem Symbol* a. *TA* geht jetzt in den Zustand s' über und überschreibt dabei das Symbol a mit dem Symbol b und führt dann die Bewegung m aus. Die Bewegung m kann l (links), r (rechts) oder - (keine Bewegung) sein.

Die Zustandsüberführung δ beschreibt also das Bearbeiten des Arbeitsbandes. *Es gilt die Vereinbarung, daß jede Anweisung durch ein oder mehrere Blanks (#) begrenzt ist.*

Jeder Turingautomat kann durch einen Kellerautomaten mit zwei Kellern simuliert werden.

Konstruktion für $a^n b^n c^n$

Ein Automat, der die Sprache

$$L = \{a^n b^n c^n | n \geq 1\}$$

akzeptiert, muß als Turingautomat ausgebildet werden, da L nicht kontextfrei ist. Diese Sprache wird nicht mehr von einem Kellerautomaten akzeptiert, da dieser mit Hilfe des Stacks zwar überprüfen kann, ob die Anzahl der a's gleich der Anzahl der b's ist, allerdings ist eine weitere Überprüfung auf die Anzahl der c's nicht möglich.

12.1 Linear beschränkte Automaten

Einem linear beschränkten Autmaten steht ein genau abgegrenzer Bereich des Arbeitsspeichers zur Verfügung. Dieses ist genau der Arbeitsspeicher, der für das Eingabewort zur Verfügung steht. Zur Begrenzung des Arbeitsspeichers wird das Symbol & als rechter Begrenzer eingeführt. Ein Eingabewort (w) wird also in der Form #w& auf das Band geschrieben. Hier ist # das Bandanfang-Symbol und & das Bandendesymbol.

> *Bei einem linear beschräntem Automaten darf das Programm diese Begrenzer nicht verändern und den rechten Begrenzer nicht überschreiten.*

Das Programm darf also lesend auf das Band links vom Bandanfangssybol zugreifen, aber nicht über das Wortendesymbol (&) hinaus laufen.

Der linear-beschränkte nichtdeterministische Turingautomat wird auch als *LBA* (linear bounded automaton) bezeichnet.

Ein linear beschränkter Automat ist (quasi) äquivalent zu einer kontextsensitiven (Typ-1) Grammatik. Die Aussage "quasi" beruht darauf, daß Sprachen, die von linear beschränkten Automaten akzeptiert werden, daß leere Wort enthalten können. Eine kontextsensitive Sprache kann dieses nicht enthalten, da dieses nicht von monotonen Regeln erzeugt werden kann.

Aus der obigen Aussage folgt, daß der Turingautomat vollständig äquivalent zur Typ-0-Grammatik ist.

Das LBA-Problem

Jeder nichtdeterministische Turingautomat kann durch einen deterministischen simuliert werden. Allerdings weiß man bis heute nicht, ob ein nichtdeterministischer linear beschränkter Turingautomat immer in einen deterministischen

Turingautomaten transformieren kann, der auch linear beschränkt ist. Dieses offene Problem ist das LBA-Problem der theoretischen Informatik.

12.2 Turing-Berechenbarkeit

Endliche Maschinen (Mealy-, Moore-) ordnen Eingabewörtern Ausgabewörter zu; sie berechnen damit eine Funktion. Ein Turing-Automat mit einer Ausgabefunktion wird zu einer Turing-Maschine. Mit Hilfe dieser Maschine läßt sich u.U. eine Funktion berechnen – diese ist dann Turing-Berechenbar.

Mit Hilfe der Turing-Maschine läßt sich die allgemeine mathematische Berechenbarkeit formal präzisieren. Das Turingband entspricht dem Speicher oder einem Blatt Papier für Notizen, Schreib- und Löschwerkzeuge entsprechen dem Schreib-/Lesekopf. Die Ausgangsdaten (gegebene Parameter) werden jeweils mit Hilfe eines bestimmten *endlichen* Verfahrens (Algorithmus, Programm) manipuliert. Hält das Programm schließlich an, steht das Ergebnis auf dem Blatt bzw. im Speicher.

12.3 Turing-Berechenbarkeit und Algorithmus

Ein Algorithmus ist ein formales Verfahren zur Lösung eines (Wort-) Problems. Dieses kann ein mathematisches Problem sein, daß mit Hilfe einer Maschine gelöst werden soll. Es gilt:

> *Eine Funktion (über einem Alphabet Σ bzw. über N_0^k) ist algorithmisch berechenbar, falls es einen Algorithmus (eine Turingmaschine) gibt, der sie berechnet.*

12.4 Universelle Turingmaschinen

Die universelle Turingmaschine (UTM) ist ein theoretisches Konzept für die Existenz universeller Rechner. Sie kann jede andere Turingmaschine einschließlich sich selbst auf deren Eingaben simulieren. Das heißt, daß die UTM als Eingabe zum einen die Turingmaschine erhält, die sie simuliert und zum anderen das Programm, daß die simulierte Turingmaschine ausführen soll.

Die Existenz dieser universellen Turingmaschine is die Grundlage für die Existenz universeller Rechner. Die CPU mit ihren Maschinenanweisungen läßt sich als universelle Turingmaschine auffassen. Sie kann andere Turingmaschinen simulieren, die z.B. eine mathematische Aufgabe lösen sollen. Das Programm, daß auf der CPU läuft ist die Turingmaschine, die die Aufgabe löst. Die Eingabeparameter des Programms sind die Eingabe für die simulierte Turingmaschine.

13 Weitere Berechenbarkeitsbegriffe

Neben der Turing-Berechenbarkeit können noch weitere Berechenbarkeitsbegriffe herangezogen werden. Diese gehen allerdings hinsichtlich ihrer Konzeption schon in die Prozeduralen Programmiersprachen hinein.

loop Loop-Berechnungen erfolgen durch vorher festgelegtes Durchlaufen einer Schleife mit Berechungsanweisungen. Ein Loop-Programm terminiert aus diesem Grunde immer und ist somit total.

while Die While-Berechnung unterscheidet sich dahingehend von der Loop-Berechnung, daß hier erst im Laufe der Berechnung ermittelt werden kann, wie oft die Berechnung wiederholt wird. While-Berechenbarkeit ist äquivalent zur Turing-Berechenbarkeit und zur Goto-Berechnung. Daher können diese Formen ineinander umgewandelt werden.

goto Die Goto-Berechung erfolgt mit Hilfe von festgelegten Sprüngen zu Teilberechnungen. Sie ist äquivalent zur While- und zur Turing-Berechenbarkeit und kann in diese umgewandelt werden und umgekehrt.

Aufgrund der Äquivalenz von Goto- und While-Berechenbarkeit ist es möglich ein beliebiges Programm in einer prozeduralen Programmiersprache zu erstellen, daß ohne Goto-Sprünge auskommt.

Loop-berechenbare Funktionen sind *total*, das heißt sie terminieren immer und sind somit immer lösbar. Die Klasse der Loop-berechenbaren Funktionen ist eine Untermenge der While-, Goto- und Turing-Funktionen.

Aufgrund dieser Tatsachen wird die Churchsche These, nach der diese formalen Berechenbarkeitsbegriffe mit unserem intuitiven Verständnis von Berechenbarkeit übereinstimmen, allgemein akzeptiert.

14 Entscheidbarkeit

Bei der Berechenbarkeit von Mengen interessiert nur, ob ein spezielles Objekt zur Menge gehört oder nicht. Im Zusammenhang mit Mengen wird dann von Entscheidbarkeit gesprochen. Beim Berechnen einer Funktion wird mit einem Automaten das Akzeptieren einer Sprache simuliert.

14.1 Semi-Entscheidbarkeit

Eine Menge ist über einem Alphabet in Verbindung mit einer Grammatik entscheidbar, falls eindeutig festgestellt werden kann, ob die ein Subset des Alphabets darstellende Menge ein Wort im Sinne der Grammatik darstellt oder nicht.

L ist entscheidbar, falls gilt:

$$X_L : \Sigma^* \rightarrow \{0,1\}$$

$$X_L(w) = \begin{cases} 1, & falls \quad w \in L \\ 0, & falls \quad w \notin L \end{cases}$$

Dagegen ist L semi-entscheidbar für:

$$X'_L : \Sigma^* \rightarrow \{0,1\}$$

$$X'_L(w) = \begin{cases} 1, & falls \quad w \in L \\ undefiniert, & falls \quad w \notin L \end{cases}$$

Die Voraussetzung für die Entscheidbarkeit ist also, daß auf jeden Fall festgelegt ist, ob ein akzeptiert werden kann oder nicht. Für die semi-Entscheidbarkeit genügt der Beweis, daß ein Wort akzeptiert wird.

Reduzierbarkeit von Mengen Bei Feststellung der Entscheidbarkeit von konkreten Mengen ist es oft hilfreich, falls die Frage nach der Entscheidbarkeit einer Menge L_1 auf die Entscheidbarkeit einer anderen Menge L_2 zurückgeführt werden kann. Hieraus kann dann auf die Entscheidbarkeit von L_1 geschlossen werden.

semi-Entscheidbarkeit

Das Postsche Korrespondenzproblem Das Postsche Korrespondenzproblem ist insofern hilfreich, als das andere (unentscheidbare) Probleme auf dieses Problem reduziert werden können und damit bewiesen werden kann, daß diese unentscheidbar sind.

Unentscheidbare und semi-entscheidbare Probleme

unentscheidbar

Korrektheitsproblem Das Korrektheitsproblem ist ein nichtentscheidbares Problem. Es besagt, daß es nicht möglich ist, automatisiert festzustellen ob ein beliebiges Programm ein Problem korrekt löst.

unentscheidbar

Das Äquivalenzproblem Das Äquivalenzproblem ist der Beweis ob zwei Versionen eines Programmes dasselbe Problem berechnen. Dieser Beweis würde es ermöglichen mit Hilfe von weiteren Untersuchungen dasjenige Programm zu ermitteln, welches nach anderen Kriterien (Benutzerfreundlichkeit) besser zur Problemlösung geeignet wäre.

15 Komplexizität

Auch prinzipiell lösbare Probleme können sich in der realen Welt als unlösbar herausstellen. Falls die Laufzeit des Algorithmus zur Lösung des Problems exponentiell mit der Größe der Eingabewörter anwächst kann die Lösung des Problems in einer endlichen Zeitspanne nicht mehr zu erreichen sein.

Die Darstellung der Komplexizität erfolgt mit Hilfe der O-Notation. Diese legt die Laufzeiten in Abhängigkeit von der Eingabe wie folgt fest:

$O(1)$ konstant, also unabhängig von der Eingabe,

$O(log\ N$ logarithmisch,

$O(n)$ linear,

$O(n\ log\ n)$ n-log-n,

$O(n^2)$ quadratisch,

$O(n^3)$ kubisch,

$O(n^k)$ exponentiell.

Die P-NP-Frage P ist die Klasse der Probleme, die mit deterministischen Turingmaschinen in Polynomzeit ($O(n^k)$) berechenbar sind. NP ist die Klasse aller Probleme, die mit nichtdeterministischen Turingmaschinen in Polynomzeit berechenbar sind. Bei der Transformation von nichtdeterministischen Turingmaschinen in äquivalente deterministische Turingmaschinen werden die Laufzeiten exponentiell und damit praktisch unakzeptabel.

Viele Probleme, die so nicht in akzeptabler Zeit gelöst werden können wie z.B. das *Travelling Salesman* werden in der Praxis durch Heuristiken, die auf Erfahrungen beruhen und suboptimal sind, zufriedenstellend gelöst.

Teil II
Zusammenfassung

Die Sprachen bzw. die sie erzeugenden Grammatiken lassen sich hierarchisch ordnen. Diese Aufstellung wird als *Chomsky*-Hierarchie bezeichnet.

16 Die Chomsky-Hierarchie

16.1 Klassifizierung der Grammatiken

Die Grammatiken lassen sich in *kontextfreie* und kontextsensitive Grammaktiken, sowie in Grammtiken vom Typ-0, Typ-1, Typ-2 und Typ-3 einordnen. Dieses Arten von Grammatiken bilden eine Hierarchie, bei der die Typ-0-Grammatiken die größte Menge darstellen. Die weiteren Typen sind jeweils Untermengen der verhergehenden Typen. Dieses liegt darin begründet, daß der Aufbau der Regelmenge ausgehend von den Typ-0 Grammatiken zu den Typ-3 Grammatiken immer weiter eingeschränkt wird, so daß die mit Ihnen erzeugbarenen Mengen an Sprachen auch immer kleiner werden.

Typ-3-Grammatiken

Eine Regel einer Typ-3-Grammatik kann pro Ableitungsschritt höchstens ein Terminalsymbol mit einer links- oder rechtsseitigen Ableigung erzeugen. Aus diesem Grunde ist die Ableitung einer Sprache der Form $L = \{a^n b^n | n \geq 0\}$ nicht möglich, da die Anzahl der as und bs bei der Ableitung der beiden Grammatiken gleich sein muß. Dieses läßt sich nur erreichen, indem pro Ableitungsschritt zwei Terminalsymbole erzeugt werden.

Typ-2-Grammatiken (kontextfreie Grammatiken)

Eine Ableitung bei der pro Schritt zwei Terminalsymbole erzeugt werden ist mit der Typ-2-Grammatik möglich. Aus diese

Typ-2-Grammatiken sind kontextfreie Grammatiken. Die Anwendung der einzelnen Regeln erfolgt vollkommen unabhängig vom Kontext des schon erzeugten Wortes. Die linke Seite einer Regel einer kontextfreien Grammatik besteht nur aus einem Nichtterminal, daß dann abgeleitet wird. Die Ableitungsregeln dürfen allerdings

Typ-1-Grammatiken (kontextsensitive Grammatiken)

Die Typ-1-Grammatiken werden auch konstextsensitive Grammatiken genannt. Hier ist der Einsatz der einzelnen Regeln vom Kontext abhängig.

Die linken Seiten der Regeln können aus Wörtern aus nichtterminalen und terminalen Symbolen bestehen, wobei allerdings mindestens 1 Nichtterminal enthalten sein muß. Die Länge der rechten Regelseiten darf nicht kleiner sein als die Linke Seite. Ein Ableitungsschritt erzeugt ein Wort, dessen Länge größer oder gleich der Länge des abzuleitenden Wortes ist. Aus diesem Grunde heißen kontextsensitive Grammatiken monoton.

Monotonie

Die Sprache $L = \{a^n b^n c^n\} | n \geq 1$ läßt sich mit einer kontextfreien Grammatik *nicht* erzeugen, da mit den kontextfreien Regeln zum einen die Anzahl und Reihenfolge der Buchstaben nicht kontrolliert werden kann. Zudem ist der Einsatz der Terminierungsregeln vom Kontext abhängig, da diese nur abhängig vom Kontext eingesetzt werden dürfen. Eine solche Grammatik heißt auch kontextsensitiv.

Typ-0-Grammatiken

Die Typ-0-Grammatiken entstehen aus den Typ-1-Grammatiken wenn die Monotonieeigenschaft aufgehoben wird. Bei Ableitung einer entsprechenden Typ-0-Regel kann die Länge des abgeleitenden Wortes in einem Ableitungsschritt kleiner sein als die des abzuleitenden Wortes.

Mit den Typ-0-Grammatiken wird die Klasse der sogenannten rekursiv-aufzählbaren Sprachen erzeugt. Mit Hilfe einer Typ-0-Grammatik lassen sich fast alle denkbaren Sprachen erzeugen. Nur Sprachen, die überabzählbar sind, also keine einfache bijektive Abbildung (Wörter lassen sich 1 zu 1 auf der Menge der Natürlichen Zahlen abbilden und können somit eindeutig durchnummeriert werden) zulassen lassen sich hiermit nicht erzeugen.

rekursiv aufzählbar

Die Klasse der überabzählbaren Sprachen lassen sich mit keiner Grammatik erzeugen und werden auch von keinem Automaten akzeptiert. Diese Sprachen werden als 2^{Σ^*} Sprachen bezeichnet.

16.2 Hierarchie der Grammatiken

Aus den einzelnen Grammatiken läßt sich die folgende Hierarchie mit den aufgeführten Eigenschaften aufstellen:

Typ	Kontext-	monotonie	links	rechts
Typ-0	kontextsensitiv	nicht monoton	T + nT	t
Typ-1	kontextsensitiv	monoton	T + nT	tSt
Typ-2	kontextfrei	monoton	nT	tSt
Typ-3	kontextfrei	monoton	nT	tS oder St

Wobei:

T	Terminalsymbol
nT	Nichtterminalsymbol
t	Terminalsymbol
S	Nichtterminalsymbol

16.3 Hierarchie der Sprachen

Die Chomsky-Hierarchie ordnet die Sprachen nach ihren Mengenbeziehungen untereinander ein:

$$REG_\Sigma \subset kfS_\Sigma \subset ksS_\Sigma \subset RE_\Sigma \subset 2^{\Sigma^*}$$

oder

$$TYP3_\Sigma \subset TYP2_\Sigma \subset TYP1_\Sigma \subset TYP0_\Sigma \subset 2^{\Sigma^*}$$

17 Zuordnung von Sprachen und Automaten

Die Grammatiken stellen erzeugende Konzepte für Sprachen dar, während Automaten akzeptierende Konzepte sind. Im vorherigen Kapitel wurden die Typen der Grammatiken den einzelnen Sprachtypen zugeordnet. Auch die einzelnen Automatentypen akzeptieren verschiedene Sprachtypen.

Automat	Sprachtyp	Beispiel
endliche Automaten	TYP-3 Sprachen	a^n
Kellerautomaten	TYP-2 Sprachen	$a^n b^n$
linear beschr. Turingautomaten	TYP-1 Sprachen	$a^n b^n c^n$
Turingautomaten	TYP-0 Sprachen	

Einfache endliche Automaten sind nur in der Lage die einfachen TYP-3 Sprachen zu erkennen. Das Wortproblem ist für sie nur entscheidbar, wenn keine Forderung an die Sybole eines Wortes der Form

$$a^n b^n$$

vorliegt, wie es bei TYP-2 Sprachen vorkommt. In einem solchen Falle benötigt der Automat einen zusätzlichen Speicher, in dem er die Anzahl der schon eingelesenen Symbole a ablegt, um diese später mit der Zahl der b Symbole vergleichen zu können.
Um dieses Wortproblem entscheiden zu können benötigt der Automat einen einfachen Stackspeicher wie er beim Kellerautomaten vorhanden ist. Dieser ist im Beispiel so konstruiert, daß er jedes eingelesene Symbol a auf den Stack legt

um es beim Einlesen eines Symbols b wieder vom Stack zu entfernen. Wenn der Stack geleert ist, stimmt die Anzahl der Symbole überein.

Eine TYP-1 Sprache verlangt, daß Wörter in der Form $a^n b^n c^n$ vorkommen können. Dieses Wortproblem ist vom Kellerautomaten nicht entscheidbar, da er nur über einen Specher mit sequentiellem Zugriff verfügt und somit nicht für den Vergleich der Anzahl von Buchstaben nicht je nach Eingabesymbol wahlfrei zugreifen kann. Die Erweiterung des Kellerautomaten, der Turingautomat ist in der Lage dieses Problem zu lösen. Er ist aufgrund seines Speicherbandes in der Lage einen wahlfreien Zugriff zum Vergleich beliebiger Symbole durchzuführen.

18 Entscheidbarkeit des Wortproblems

Das Wortproblem ist für alle Typen von Sprachen außer den den Typ-0 Sprachen entscheidbar. Je nach Automaten- bzw. Sprachtyp wird jedoch die Komplexizität größer. Das heißt, daß die Laufzeit bis zur Akzeptanz eines Wortes größer wird. Bei TYP-3- und DPDA-S liegt hier noch ein linearer Zusammenhang zwischen Wortlänge und Laufzeit bzw. Anzahl der durchzuführenden Operationen vor. Bei den TYP-2-Sprachen mit gegebener Grammatik in Chomsky-Normalform steigt die Laufzeit schon in der 3. Potenz an um bei TYP-1-Grammatiken exponentiell mit der Wortlänge zu steigen. TYP-0-Sprachen schließlich sind hinsichtlich des Wortproblems unlösbar.

Teil III

Anhang

19 Minimaler Automat

1. Die Zustandsmenge S des Automaten wird in zwei disjunkte[15] Teilmengen zerlegt. Diese bilden insgesamt die Partition $\Pi 1$:

$$\begin{aligned} S_{11} &= F \\ S_{12} &= \{S - F\} \\ \Pi_1 &= \{S_{11}, S_{12}\} \\ \Pi_1 &= \{\{s_4, s_5\}, \{s_0, s_1, s_2, s_3\}\} \end{aligned}$$

   Jetzt werden die Teilmengen jede für sich dahingehend untersucht, ob sich die Folgezustände je Eingabesymbol sich alle in dem selben Block befinden (Untersuchung, ob die Folgezustandszeile des Blocks nur Folgezustände in einem Block aufweist).

[15] gegensätzlich einander ausschließend

	S_{11}				S_{12}	
	s_4	s_5	s_0	s_1	s_2	s_3
0	S_{12}	S_{12}	S_{12}	S_{11}	S_{12}	S_{11}
1	S_{11}	S_{11}	S_{12}	S_{11}	S_{12}	S_{11}

Für den Block S_{11} ist die Bedingung erfüllt. Hier liegen die Folgezustände für jedes Eingabewort jeweils im selben Block. Er wird im folgenden als Block S_{21} bezeichnet.
Bei Block S_{11} ist diese Bedingung nicht erfüllt. Hier muß eine weitere Unterteilung stattfinden, daß sich die Partition Π_{i+1} ergibt als:

$$\begin{aligned} \Pi_2 &= \{S_{21}, S_{22}, S_{23}\} \\ \Pi_2 &= \{\{s_4, s_5\}, \{s_0, s_2\}, \{s_1, s_3\}\} \end{aligned}$$

Die neu erzeugten Blöcke wurden hinsichtlich der Minimierung zu s_0, s_2 und s_1, s_3 zusammengefaßt. Die neue Zustandstabelle ergibt sich damit zu:

	S_{21}		S_{22}		S_{23}	
	s_4	s_5	s_0	s_2	s_1	s_3
0	S_{23}	S_{23}	S_{23}	S_{22}	S_{21}	S_{21}
1	S_{21}	S_{21}	S_{22}	S_{22}	S_{21}	S_{21}

Der Block S_{22} mit den Zuständen s_0 und s_2 hat für das Eingabesymbol 0 noch nicht das Ziel der Folgezustände im gleichen Block erreicht. Daher muß dieser Block noch einmal aufgeteilt werden:

$$\begin{aligned} \Pi_3 &= \{S_{31}, S_{32}, S_{33}, S_{34}\} \\ \Pi_3 &= \{\{s_4, s_5\}, \{s_1, s_3\}, \{s_0\}, \{s_2\}\} \end{aligned}$$

Hier ergibt sich die Zustandstabelle:

	S_{31}		S_{32}		S_{33}	S_{34}
	s_4	s_5	s_1	s_3	s_0	s_2
0	S_{32}	S_{32}	S_{31}	S_{31}	S_{32}	S_{33}
1	S_{31}	S_{31}	S_{31}	S_{31}	S_{34}	S_{33}

Jetzt liegt für jeden Block der Folgezustand für ein Eingangssymbol im selben Block. Der minimale Automat ist hiermit erreicht. Die Definition des minimalen Automaten lautet:

$$A_{min} = (\{0,1\}, \{s_0\}, \{s_2\}, \{s_1, s_3\}, \{s_4, s_5\}, \delta_{min}, \{s_0\}, \{s_4, s_5\})$$

20 Herleiten einer Grammatik

Es soll eine Grammatik hergeleitet werden, die alle Bitfolgen erzeugt, in denen genau einmal das Infix 111 enthalten ist.

Die allgemeine Form der zu erzeugenden Sprache ist wie folgt Beschrieben:

$$L = \{u111v | u, v \in \{0,1\}^*\}$$

Ein Wort der Sprache ist besteht also aus einem Präfix u, welches eine beliebige Bitfolge darstellt, die höchstens zwei aufeinanderfolgende Einsen enthält (also das Teilwort 111 nicht enthält) und auch nicht auf 1 oder 11 enden darf.
Das Postfix v ist eine Bitfolge, die auch höchstens zwei aufeinanderfolgende Einsen enthalten darf. Das Postfix kann auch ganz entfallen, also $v = \epsilon$.

Nun können Regeln aufgestellt werden, die die einzelnen Muster enthalten.

1. Die Regel $S \to 0S$ erzeugt beliebig viele Nullen.
2. Die Regel $S \to 1A$ erzeugt eine 1.
3. Die Regel $A \to 0S$ erzeugt eine 0 hinter der 1 und bewirkt ein Beginn von vorn.
4. Die Regel $A \to 1B$ erzeugt eine zweite 1 in Folge nach der Regel. 2
5. Die Regel $B \to 0S$ erzeugt nach der zweiten Eins in Folge aus Regel 4 eine 0 und bewirkt einen Neubeginn bei 1 oder 2.
6. Die Regel $B \to 1C$ erzeugt die dritte 1 in Folge, hiernach darf zwangsläufig nur noch eine 0 kommen. Zudem dürfen nur noch zwei Einsen in Folge auftreten.
7. Die Regel $C \to 0D$ erzeugt eine 0 nach der dritten Eins in Folge aus 6.
8. Die Regel $C \to \epsilon$ terminiert die Ableitung nach den drei Einsen in Folge. Das Postfix v ist dann das leere Wort.
9. Die Regel $D \to 0D$ erzeugt beliebig viele Nullen.
10. Die Regel $D \to 1E$ erzeugt wiederum eine Eins nach der Null aus 6.
11. Die Regel $E \to 0D$ erzeugt eine 0 nach der Eins aus 10 mit einem Rücksprung.
12. Die Regel $E \to 1F$ erzeugt die zweite 1 in Folge im Postfix v.
13. Die Regel $F \to 0D$ erzeugt eine Null nach den zwei Einsen in Folge mit einem Rücksprung an den Beginn des Postfix v.
14. Die Regel $D \to \epsilon$ führt zur Terminierung.
15. Die Regel $E \to \epsilon$ führt zur Terminierung.
16. Die Regel $F \to \epsilon$ führt zur Terminierung.

Es ergibt sich also die Grammatik

$$G \quad = \quad (\{0,1\}, \{S, A, B, C, D, E, F\}, P, S)$$

mit der Regelmenge:

$$
\begin{aligned}
P \quad = \quad & \{ \\
& S \to 0S, \\
& S \to 1A, \\
& A \to 0S, \\
& A \to 1B, \\
& B \to 0S, \\
& B \to 1C, \\
& C \to 0D, \\
& C \to \epsilon, \\
& D \to 0D, \\
& D \to 1E, \\
& E \to 0D, \\
& E \to 1F, \\
& F \to 0D, \\
& D \to \epsilon, \\
& E \to \epsilon, \\
& F \to \epsilon \\
& \}
\end{aligned}
$$

21 Beschreibung des Pumping-Lemma

Wenn L eine reguläre Sprache ist, existiert ein *DEA* (deterministischer endlicher Automat) $M = (\Sigma, S, \delta, s_0, F)$, der L akzeptiert.

Die Anzahl der Zustände in der Zustandsmenge S sei n. Wenn es nun ein Wort w in L gibt, daß n oder mehr Zeichen hat, muß auf dem Weg vom Anfangszustand *s_0* zu einem Endzustand mindestens ein Zustand *mehrfach* angenommen werden, da es ja nur n verschiedene Zustände gibt.

Um zu zeigen, daß eine bestimmte Sprache L *nicht* regulär ist, kann man so vorgehen:

1. Wähle n aber nicht beliebig.
2. Wähle $z \in L$, so daß $|z| \geq n$.
3. Teile z auf in $z = uvw, |uv| \leq n, |v| \geq 1$.
4. Zeige, daß für jedes uvw ein i existiert, so daß $uv^i w \notin L$.
5. Wenn das klappt, kann L nicht regulär sein.

Die Definition besagt, daß in einer regulären Sprache alle Wörter deren Länge größer oder gleich einer bestimmten Zahl ist, in drei Teile aufgeteilt werden können, wobei alle Wörter die durch beliebig Wiederholen ("Aufpumpen") des mittleren Teils entstehen, ebenfalls Wörter dieser Sprache sind.

Es können also neue Wörter gebildet werden, indem der Mittelteil eines Wortes beliebig wiederholt wird.

22 Ableitung einer Typ-3-Grammatik

Die Typ-3-Grammatik besitzt den Aufbau:

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

mit

Σ : Terminalalphabet (Menge der Buchstaben der Sprache)

N : Nonterminalalphabet ("Variablen", diskunkt zu N)

P : Produktions- oder Regelmenge

S : *Startsymbol*

Die Ableitung der Regelmenge erfolgt schrittweise. Ein Ableitungsschritt wird wie folgt durchgeführt:

> Ein Nichtterminal (Variable) wird durch ein Wort über $\Sigma \cup N$ (Vereinigung von Terminalalphabet, also den "Buchstaben" der Sprache vereinigt mit der Menge der Nichtterminale) ersetzt. Ein Nonterminal kann also durch eine Kombination von Buchstaben und Variablen ersetzt werden. Ist ein Wort $\omega_1 A \omega_2$ mit $A \in N$ (A ist ein Nonterminal, also eine Variable) und $\omega_1, \omega_3 \in (\Sigma \cup N)^*$ (ω_1, ω_3 sind kombiniert aus Terminal und Nonterminal) bereits abgeleitet und es existierte eine Regel $A \to \omega_2, \omega_2 \in (\Sigma \cup N)^*$, dann gilt
>
> $$\omega_1 A \omega_3 \Rightarrow \omega_1 \omega_2 \omega_3$$
>
> Die Regel besagt, daß ein Nonterminalsymbol zu einer Kombination aus Terminal- und Nonterminalsymbol abgeleitet werden kann.

Eine Typ-3-Grammatik ist genau wie eine Typ-2-Grammatik eine *kontextfreie Grammatik*. Die Ableitung der Regeln, die linksseitig nur aus einem Nonterminal (Variable) bestehen, erfolgt unabhängig vom Kontext des bisher abgeleiteten (Teil-) Wortes, also unabhängig der rechts- und linksseitigen Teilwörter. Mit einer Typ-3-Grammatik läßt sich *keine* Sprache der Form $L = \{a^n b^n | n \geq 0\}$ erzeugen. Dieses liegt darin begründet, daß bei Ableitung einer Typ-3-Regel höchstens *ein* Terminalsymbol erzeugt werden kann. Um die Sprache L zu erzeugen ist es jedoch notwendig, daß die Anzahl der a's und b's gleich ist. Dieses ist nur möglich wenn eine Regel definiert wird, die beide Terminalsymbole gleichzeitig erzeugt:

$$S \to aSb$$

Bei den Typ-3-Grammatiken ist aber nur eine linksseitige oder eine rechtsseitige Ableitung möglich, so daß obiges Konstrukt fehlschlagen muß. Eine Regel der obigen Form darf nur bei der *Typ-2-Grammatik* aufgestellt werden.

Die Regel zur Erzeugung endet, wenn kein Nonterminalsymbol mehr ersetzt werden kann. In der Regel wird hier die Regel $S \to \epsilon$ eingesetzt.

23 Herleiten des Ableitungsbaums

Mit Hilfe des sogenannten Ableitungs- oder Parsebaums lassen sich die die Ableitungen kontextfreier Grammatiken veranschaulichen. Hierbei wird die Wurzel des Ableitungsbaums (t_G) mit S markiert. Die inneren Knoten sind mit Nichtterminalen ("Verben") markiert. Die Blätter sind mit Terminalsymbolen aus $N \cup \Sigma \cup \{epsilon\}$ markiert.

Die kontextfreie Grammatik $G = (\{a, b\}, \{S, A\}, P, S)$ mit der Regelmenge

$$\begin{aligned} P &= \{S \to aAS|a \\ &\quad a \to SbA|SS|ba\} \end{aligned}$$

und der unten aufgeführten Linksableitung (bei der in jedem Ableitungsschritt das am weitesten links stehende Nonterminal (beliebig bei Alternativen) ersetzt wird) soll als Ableitungsbaum dargestellt werden. Die Linksableitung der Grammatik wird wie folgt durchgeführt.
[16]

$$\begin{aligned} S &\Rightarrow aAS \\ &\Rightarrow aSbAS \\ &\Rightarrow aabAS \\ &\Rightarrow aabbaS \\ &\Rightarrow aabbaa \end{aligned}$$

Ausgehend von der Wurzel S werden jetzt die ersten *drei* Äste für die erste Ableitung eingezeichnet *a,A,S*. Der Ast zum a endet im Blatt *a*. Die anderen beiden Äste enthalten die Nonterminalsymbole A und S. Sie sind folglich *innere* Knoten, von denen noch weitere Äste ausgehen.

Der Knoten S wird jetzt zu a abgeleitet und endet somit im Blatt *a*. Der Knoten A wird zu SbA abgeleitet. Von ihm gehen daher drei Äste aus, von denen einer *(b)* im Blatt endet. Der neue Knoten S wird zu a abgeleitet und endet im entsprechenden Blatt. Der Knoten A wird zu (den Blättern) b und a abgeleitet.
[17]

24 Erweiterte-Baccus-Naur Form

$$G_{integer} = (\{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{int, vz, zf, z1, z0\}, P, int)$$

[18] mit den Regeln:

$$\begin{aligned} P &= \{ \\ &\quad int ::= vz, zf, \end{aligned}$$

[16] Hier wird das S zuerst durch die Möglichkeit aAS ersetzt. Theoretisch wäre auch eine Ableitung nach a möglich gewesen. In diesem Falle wäre das gebildete Wort nur ein *a*.
[17] s. Aufs 4, S. 14
[18] $G = (\Sigma, N, P, S)$, mit Startsymbol S

$$
\begin{aligned}
&vz ::= \{(+|-)_0^1\},\\
&zf ::= (0|z1\{z0\}^*),\\
&z1 ::= (1|2|...|9),\\
&z0 ::= (0|z1)\\
&\}
\end{aligned}
$$

Eine Ableitung kann wie folgt durchgeführt werden:

$$
\begin{aligned}
int &\Rightarrow vzzf \text{ (Expansion)}\\
&\Rightarrow \{(+|-)\}_0^1 zf \text{ (Expansion)}\\
&\Rightarrow zf \text{ (Reduktion } \{(+|-)\}_0^1 \rightarrow \epsilon)\\
&\Rightarrow (0|z1\{z0\}^*) \text{ (Expansion)}\\
&\Rightarrow z1\{z0\}^* \text{ (Reduktion: Auswahl einer Alternative)}
\end{aligned}
$$

Umwandlung in eine kontextfeie Grammatik

Eine Regel in der Backus-Naur-Form läßt sich in eine Regel einer kontextfreien Grammatik umwandeln.

$$A ::= (\alpha_1, \alpha_2, ..., \alpha_k) \quad wird \quad A \rightarrow \alpha_1\alpha_2...\alpha_k$$

$$
\begin{aligned}
A ::= (\alpha_1|\alpha_2|...|\alpha_k) \quad wird \quad & A \rightarrow \alpha_1,\\
& A \rightarrow \alpha_2,\\
& A \rightarrow \alpha_k\\
& \\
& oder\\
& A \rightarrow \alpha_1|\alpha_2|...|\alpha_k
\end{aligned}
$$

$$A ::= \alpha\{\beta\}^*\gamma \quad wird \quad A \rightarrow \alpha B\gamma, B \rightarrow \beta B, B \rightarrow \epsilon$$

$$A ::= \alpha\{\beta\}_0^1\gamma \quad wird \quad A \rightarrow \alpha\gamma, A \rightarrow \alpha\beta\gamma$$

25 Beispiel eines Kellerautomaten

$$L = \{a^n b^n | n \geq 0\}$$

Es soll ein Kellerautomat konstruiert werden, der die Sprache L akzeptiert. Die Sprachdefinition sagt aus, daß die Anzahl der a's und b's gleich groß ist. Um dieses zu überprüfen, bietet sich der Kellerautomat an. Da die a's zuerst gelesen werden muß so etwas wie eine Zählung der gelesenen a's erfolgen. Mit Hilfe des Kellerautomaten läßt sich dieses lösen indem jedes gelesene a auf den Stack

gepusht wird. Wenn die b's gelesen werden wird pro gelesenem b ein a vom Stack entfernt (pop). Sind alle b's gelesen und alle a's vom Stack entfernt, wird das Eingabewort akzeptiert.

Der folgende Kellerautomat akzeptiert die oben beschriebene Sprache L mit Endzustand:

$$\begin{aligned} K &= (\Sigma, S, \Gamma, \delta, s_0, \bot, F) \\ K &= (\{a,b\}, \{s_0, s_1, s_f\}, \{a, \bot\}, \delta, s_0, \{s_f\}) \end{aligned}$$

s_f ist der Endzustand des Automaten.

Die Zustandsüberführung δ lautet:

$$\begin{aligned} \delta = \{ \quad & (s_0, \epsilon, \bot, s_f, \epsilon) && (9) \\ & (s_0, a, \bot, s_0, a\ \bot), && (10) \\ & (s_0, a, a, s_0, aa), && (11) \\ & (s_0, b, a, s_1, \epsilon), && (12) \\ & (s_1, b, a, s_1, \epsilon), && (13) \\ & (s_1, \epsilon, \bot, s_f, \epsilon)\} && (14) \end{aligned}$$

Das letzte Symbol der Zustandsüberführung ist das Symbol, welches das aktuelle Kellersymbol ersetzt.

Die einzelnen Zustandsüberführungen erfolgen wie folgt:

(1) akzeptiert das leere Wort. Der Automat befindet sich im Ausgangszustand s_0, der Stack ist leer $\bot$. Nach Lesen des leeren Wortes geht der Automat direkt in den Endzustand s_f über und hat somit das leere Wort akzeptiert. (da $L = \{a^n b^n | n \geq 0\}$)

(2) legt das *erste* eingelesene a auf den Keller. Der Automat befindet sich noch im Ausgangszustand, der Keller ist noch nicht belegt. Man sieht, daß das Kellerbottomsymbol durch durch die Kombination $a\ \bot$ ersetzt wird, also ein *pop* auf den Keller gefolgt von einem *push* $a\ \bot$ ausgeführt wird.

(3) legte alle folgende a's auf den Keller. Der Automat befindet sich im Zustand s_0, das Eingabesymbol ist wie das oberste Kellersymbol ein a. Der Automat bleibt im Zustand s_0 und ersetzt das aktuelle Kellersymbol a durch aa.

(4) ist der Zustandsübergang beim Einlesen vom ersten b. Der Automat befindet sich im Zustand s_0, das Kellersymbol ist das a. Der Automat wechselt in den Zustand s_1. Das aktuelle Kellersymbol wird durch das leere Wort (ϵ) ersetzt (das oberste Symbol also gelöscht). Im Zustand s_1 können nur noch die Zustandsübergänge (5) und (6) wirksam werden.

(5) dieser Übergang wird bei Einlesen der nächsten b's durchlaufen. Der Keller zeigt jeweils a, aktueller Zustand ist immer s_1. Bei jedem dieser Zustandsübergänge wird ein a vom Keller gepopt.

(6) ist der Zustand der durchlaufen wird, falls das Eingabewort abgearbeitet ist (die Eingabe liefert nur noch das leere Wort). Der Keller hat den Boden ($\perp$) erreicht. Der diesem Zustand folgende ist der Endzustand s_f. Das Eingabewort wurde vom Automaten akzeptiert. Diese letzte Schritt mit dem Einlesen des leeren Wortes vom Eingabeband ist notwendig, weil immer nur das oberste Symbol des Kellers überprüft werden kann. Daher muß das Bottomsymbol ausgelesen werden. Falls der Automat in dieser Position stehen geblieben ist und das Eingabewort noch nicht komplett abgearbeitet ist, akzeptiert der Kellerautomat das Eingabewort nicht. In diesem Falle würde das Eingabewort mehr b's als a's enthalten.

26 Abzählbarkeit

Es soll gezeigt werden, ob die Potenzmenge P(N) von N, das ist die Menge aller Teilmengen von N, überabzählbar ist.

> Die Potenzmenge *P(M)* ist die Menge aller Teilmengen der Menge M.
>
> $$P(M) = \{M' | M' \subseteq M\}$$
>
> Neben der Menge M selbst gehört auch die leere Menge $\emptyset$ zur Potenzmenge von M.

Diese Menge ist abzählbar wenn gilt:

$$P(N) = \{M_1, M_2, M_3, ...\}$$

Die Potenzmenge ist die Menge der Teilmengen von M.

Die Menge der natürlichen Zahlen ist abzählbar:

$$N = \{1, 2, 3, ...\}$$

Wir definieren einen weitere Menge M durch:

$$i \in M \text{ genau dann, wenn } i \notin M_i$$

Das Element i gehört zur Menge M, wenn es nicht in der Menge M_i enthalten ist.

Für eine beliebige Zahl k gilt nun:

- Ist $k \in M$, dann ist $k \in M_k$, da $M = M_k$. Hieraus folgt aber, daß $k \notin M$.
- Ist $k \notin M$, dann ist $k \notin M_k$, da $M = M_k$. Hieraus folgt, daß $k \in M$.

Es gilt also:

$$k \in M \text{ genau dann, wenn } k \notin M$$

Dieses ist offensichtlich ein Wiederspruch, so daß die ursprüngliche Annahme, daß die Potenzmenge von N abzählbar ist falsch ist. *P(N)* ist also überabzählbar.

27 Beispiele der Turing-Berechenbarkeit

Vorbemerkung

Die Codierung von (natürlichen) Zahlen in der Turingmaschine soll mit Hilfe eines Strichcodes erfolgen. Hierzu wird für jede natürliche Zahl n die Anzahl n Strichsymbole — auf das Band geschrieben bzw. gelesen. Wir definieren daher eine Eingabefunktion $\alpha : N_0 \rightarrow \{|\}^*$, die diese Umwandlung vornimmt:

$$\alpha : \alpha(n) = |^n$$

Die Ausgabe wird durch die Ausgabefunktion $\beta : \{|\}^* \rightarrow N_0$ vorgenommen:

$$\beta : \beta(|^n) = n$$

Zur Bearbeitung einer mehrstelligen Eingabe wird die Eingabecodierung so abgewandelt, daß die einzelnen Komponenten durch *0* getrennt werden: $\alpha_k : N_0^k \rightarrow \{|, 0\}^*$:

$$\alpha_k(n_1, n_2, ..., n_k) = |^{n_1}0|^{n_2}0...|^{n_k}$$

Eine Eingabe von (3,2) wird also als |||0|| auf das Band geschrieben.

1. $\omega : N_0 \rightarrow N_0$, definiert durch $\omega(n) = undefiniert$
   Diese Funktion ist Turing-Berechenbar, da die Turingmaschine:

   $$\begin{aligned} TM &= (\{|\}, \{s_0\}, \{|, \#\}, \delta, s_0, \#, \emptyset) \\ mit & \\ \delta &= \{(s_0, a, s_0, a, r) | a \in \{|, \#\}\} \end{aligned}$$

   ω berechnet.
   Für jede Eingabe bewegt TM den Schreibkopf nur nach rechts und hält nie an. Die Maschine liefert also keine Ausgabe.

2. $suc : N_0 \rightarrow N_0$, definiert durch $suc(n) = n + 1$
   Auch diese Funktion ist Turing-Berechenbar, da die Turingmaschine:

   $$\begin{aligned} TM &= (\{|\}, \{s_0, s_1, s_f\}, \{|, \#\}, \delta, s_0, \#, s_f) \\ mit & \\ \delta &= \{(s_0, |, s_1, |, l), (s_1, \#, s_f, |, -)\} \end{aligned}$$

   suc berechnet. Diese Maschine fügt links an die Eingabe einen Strich (—) an und geht in den Endzustand.