Vorlesung Grundlagen der Theoretischen Informatik

Markus Lohrey

Universität Siegen

Sommersemester 2015

Organisatorisches zur Vorlesung

Informationen finden Sie unter

`http://www.eti.uni-siegen.de/fb12/ti/lehre/ss15/grundlagen_der_theoretischen_informatik/grundlagen_der_theoretischen_informatik.html?m=e&lang=de`

z. B.

- Aktuelle Version der Folien
- Übungsblätter
- Aktuelle Informationen

Literaturempfehlung: Uwe Schöning: Theoretische Informatik – kurz gefasst, Spektrum Akademischer Verlag

Die **Übungen** werden organisiert von: Danny Hucke, Daniel König und Moses Ganardi.

Tutorials: Wir noch organisiert

Prfungsvorbereitungskurs: Wird es wohl auch wieder geben

Mengentheoretische Grundlagen

Naive Definition (Mengen, Elemente, $\in$, $\notin$)

Eine Menge ist die Zusammenfassung von bestimmten unterschiedlichen Objekten (die Elemente der Menge) zu einem neuen Ganzen.

Wir schreiben $x \in M$, falls das Objekt x zur Menge M gehört.
Wir schreiben $x \notin M$, falls das Objekt x nicht zur Menge M gehört.

Eine Menge, welche nur aus endlich vielen Objekten besteht (eine endliche Menge), kann durch explizite Auflistung dieser Elemente spezifiziert werden.

Beispiel: $M = \{2, 3, 5, 7\}$.

Hierbei spielt die Reihnfolge der Auflistung keine Rolle:
$\{2, 3, 5, 7\} = \{7, 5, 3, 2\}$.

Auch Mehrfachauflistungen spielen keine Rolle:
$\{2, 3, 5, 7\} = \{2, 2, 2, 3, 3, 5, 7\}$.

Mengentheoretische Grundlagen

Eine besonders wichtige Menge ist die leere Menge $\emptyset = \{\}$, die keinerlei Elemente enthält.

In der Mathematik hat man es häufig auch mit unendlichen Mengen zu tun (Mengen, die aus unendlich vielen Objekten bestehen).

Solche Mengen können durch Angabe einer Eigenschaft, welche die Elemente der Menge auszeichnet, spezifiziert werden.

Beispiele:

- $\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$ (Menge der natürlichen Zahlen)
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (Menge der ganzen Zahlen)
- $P = \{n \in \mathbb{N} \mid n \geq 2, n \text{ ist nur durch 1 und } n \text{ teilbar}\}$ (Menge der Primzahlen)

Definition ($\subseteq$, Potenzmenge, $\cap$, $\cup$, $\setminus$, disjunkt)

Seien A und B zwei Mengen.

- $A \subseteq B$ bedeutet, dass jedes Element von A auch zu B gehört (A ist eine Teilmenge von B); formal:

$$\forall a : a \in A \rightarrow a \in B$$

- $2^A = \{B \mid B \subseteq A\}$ (Potenzmenge von A)
- $A \cap B = \{c \mid c \in A \text{ und } c \in B\}$ (Schnitt von A und B)
- $A \cup B = \{c \mid c \in A \text{ oder } c \in B\}$ (Vereinigung von A und B)
- $A \setminus B = \{c \in A \mid c \notin B\}$ (Differenz von A und B)
- Zwei Mengen A und B sind disjunkt, falls $A \cap B = \emptyset$ gilt.

Definition (beliebige Vereinigung und Schnitt)

Sei I eine Menge und für jedes $i \in I$ sei A_i wiederum eine Menge. Dann definieren wir:

$$\bigcup_{i \in I} A_i = \{a \mid \exists j \in I : a \in A_j\}$$

$$\bigcap_{i \in I} A_i = \{a \mid \forall j \in I : a \in A_j\}$$

Beispiele:

$$\bigcup_{a \in A} \{a\} = A \text{ für jede Menge } A$$

$$\bigcap_{n \in \mathbb{N}} \{m \in \mathbb{N} \mid m \geq n\} = \emptyset$$

Definition (Kartesisches Produkt)

Für zwei Mengen A und B ist

$$A \times B = \{(a, b) \mid a \in A \text{ und } b \in B\}$$

das kartesische Produkt von A und B (Menge aller Paare aus einem Element von A und einem Element von B).

Allgemeiner: Für Mengen $A_1, \ldots, A_n$ $(n \geq 2)$ sei

$$\begin{aligned} \prod_{i=1}^{n} A_i &= A_1 \times A_2 \times \cdots \times A_n \\ &= \{(a_1, \ldots, a_n) \mid \text{ für alle } 1 \leq i \leq n \text{ gilt } a_i \in A_i\} \end{aligned}$$

Falls $A_1 = A_2 = \cdots A_n = A$ schreiben wir auch A^n für diese Menge.

Beispiele und einige einfache Aussagen:

- $\{1,2,3\} \times \{4,5\} = \{(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)\}$
- Für alle Mengen A, B, und C gilt:

$$(A \cup B) \times C = (A \times C) \cup (B \times C)$$
$$A \times (B \cup C) = (A \times B) \cup (A \times C)$$
$$(A \cap B) \times C = (A \times C) \cap (B \times C)$$
$$A \times (B \cap C) = (A \times B) \cap (A \times C)$$

Vollständige Induktion

Um eine Aussage $P(n)$ für jede natürliche Zahl $n \in \mathbb{N}$ zu beweisen, genügt es, folgendes zu zeigen:

1. $P(0)$ gilt (Induktionsanfang).
2. Für jede natürliche Zahl $n \in \mathbb{N}$ gilt: Wenn $P(n)$ gilt, dann gilt auch $P(n+1)$ (Induktionsschritt).

Dieses Beweisprinzip nennt man das Prinzip der vollständigen Induktion.

Beispiel: Wir beweisen mittels vollständiger Induktion, dass für alle natürlichen Zahlen n gilt:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Vollständige Induktion

Induktionsanfang: Es gilt $\sum_{i=1}^{0} i = 0 = \frac{0 \cdot 1}{2}$.

Induktionsschritt: Angenommen es gilt

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Dann gilt auch

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= (\sum_{i=1}^{n} i) + n + 1 \\
&= \frac{n(n+1)}{2} + n + 1 \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2}
\end{aligned}$$

Vollständige Induktion

Mittels des Prinzips der Induktion kann man auch Objekte definieren.

Angenommen, wir wollen für jede natürliche Zahl $n \in \mathbb{N}$ ein Objekt A_n definieren.

Dies kann man wie folgt machen:

1. Definiere A_0.
2. Gib eine allgemeine Vorschrift an, wie das Objekt A_{n+1} aus den (bereits konstruierten) Objekten $A_0, A_1, \ldots, A_n$ konstruiert werden kann.

Wörter

Definition (Alphabet, Wörter)

Ein Alphabet ist eine endliche nicht-leere Menge.

Ein Wort über dem Alphabet Σ ist eine endliche Zeichenkette der Form $a_1 a_2 \cdots a_n$ mit $a_i \in \Sigma$ für $1 \leq i \leq n$. Die Länge dieses Worts ist n.

Für ein Wort w schreiben wir auch $|w|$ für die Länges des Wortes w.

Für $n = 0$ erhalten wir das leere Wort (das Wort der Länge 0), welches mit ε bezeichnet wird.

Mit Σ^* bezeichnen wir die Menge aller Wörter über dem Alphabet Σ.

Die Menge aller nicht-leeren Wörter ist $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Beispiel: Sei $\Sigma = \{a, b, c\}$. Dann sind mögliche Wörter aus Σ^*:

$$\varepsilon, a, b, aa, ab, bc, bbbab, \ldots$$

Wörter

Konventionen: Wörter aus Σ^* werden mit Kleinbuchstaben (aus der hinteren Hälfte des Alphabets) bezeichnet: u, v, w, x, y, z, ...

Definition (Konkatenation von Wörtern)

Für Wörter $u = a_1 \cdots a_m$ und $v = b_1 \cdots b_m$ mit $a_1, \ldots, a_m, b_1, \ldots, b_m \in \Sigma$ ist das Wort

$$u \circ v = a_1 \cdots a_m b_1 \cdots b_m.$$

die Konkatenation (oder Hintereinanderschreibung) der Wörter u und v. Anstatt $u \circ v$ schreiben wir meistens nur uv.

Offensichtlich gilt für alle Wörter $u, v, w \in \Sigma^*$:

- $(u \circ v) \circ w = u \circ (v \circ w)$ oder kurz $(uv)w = u(vw)$
- $\varepsilon \circ u = u \circ \varepsilon$

Erinnerung aus DMI: $(\Sigma^*, \circ)$ ist also ein Monoid, man nennt es auch das von Σ erzeugte freie Monoid.

Sprachen

Definition (Sprache)

Sei Σ ein Alphabet.
Eine (formale) Sprache L über Σ ist eine beliebige Teilmenge von Σ^*, d.h. $L \subseteq \Sigma^*$.

Beispiel: Sei $\Sigma = \{(,), +, -, *, /, a\}$. Dann können wir die Sprache *EXPR* der korrekt geklammerten Ausdrücke definieren. Es gilt beispielsweise:

- $(a - a) * a + a/(a + a) - a \in EXPR$
- $(((a))) \in EXPR$
- $((a+) - a(\notin EXPR$

Grammatiken (Einführung)

Grammatiken in der Informatik sind – ähnlich wie Grammatiken für natürliche Sprachen – ein Mittel, um alle syntaktisch korrekten Sätze (hier: Wörter) einer Sprache zu erzeugen.

Beispiel: Eine vereinfachte Grammatik zur Erzeugung natürlichsprachiger Sätze:

$$
\begin{aligned}
\langle \text{Satz} \rangle &\rightarrow \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle \\
\langle \text{Subjekt} \rangle &\rightarrow \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle \\
\langle \text{Artikel} \rangle &\rightarrow \varepsilon \\
\langle \text{Artikel} \rangle &\rightarrow \text{der} \\
\langle \text{Artikel} \rangle &\rightarrow \text{die} \\
\langle \text{Artikel} \rangle &\rightarrow \text{das} \\
\langle \text{Attribut} \rangle &\rightarrow \varepsilon
\end{aligned}
$$

Grammatiken (Einführung)

$$
\begin{aligned}
\langle \text{Attribut} \rangle &\rightarrow \langle \text{Adjektiv} \rangle \\
\langle \text{Attribut} \rangle &\rightarrow \langle \text{Adjektiv} \rangle \langle \text{Attribut} \rangle \\
\langle \text{Adjektiv} \rangle &\rightarrow \text{kleine} \\
\langle \text{Adjektiv} \rangle &\rightarrow \text{bissige} \\
\langle \text{Adjektiv} \rangle &\rightarrow \text{große} \\
\langle \text{Substantiv} \rangle &\rightarrow \text{Hund} \\
\langle \text{Substantiv} \rangle &\rightarrow \text{Katze} \\
\langle \text{Prädikat} \rangle &\rightarrow \text{jagt} \\
\langle \text{Objekt} \rangle &\rightarrow \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle
\end{aligned}
$$

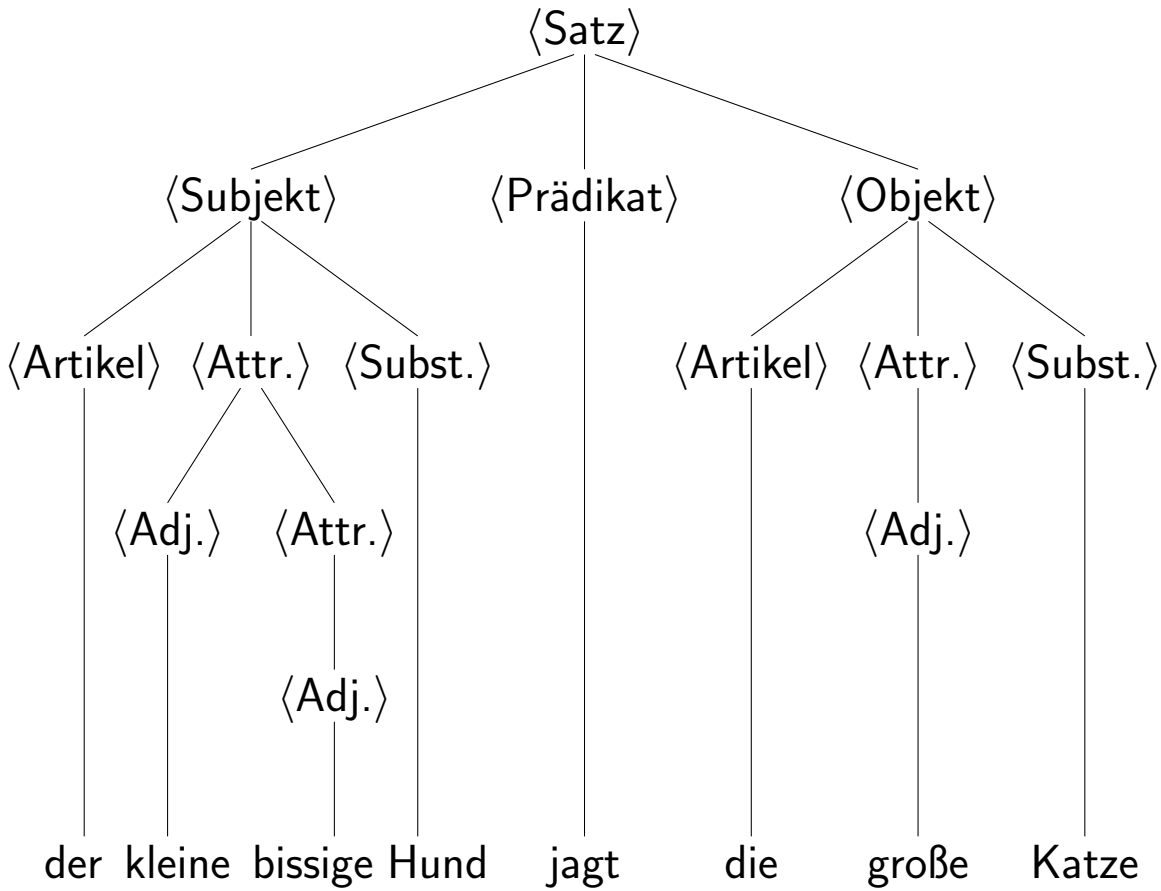
- In spitzen Klammern: Variable, Nicht-Terminale
- Ohne spitze Klammern: Terminale

Grammatiken (Einführung)

Gehört folgender Satz zu der Sprache, die von der Grammatik erzeugt wird?

der kleine bissige Hund jagt die große Katze

Grammatiken (Einführung)



Dieser Baum ist der "Beweis" dafür, dass der Satz in der Sprache vorkommt. Man nennt ihn einen Syntaxbaum.

Grammatiken (Einführung)

Mit Hilfe dieser (endlichen) Grammatik ist es möglich, unendlich viele Sätze zu erzeugen:

der Hund jagt die kleine kleine kleine . . . Katze

Das heißt, die zu der Grammatik gehörende Sprache (man sagt auch: die von der Grammatik erzeugte Sprache) ist unendlich.

Grammatiken (Definition)

Grammatiken besitzen Produktionen der Form

$$\textit{linke Seite} \rightarrow \textit{rechte Seite}$$

Sowohl auf der linken, als auch auf der rechten Seite können zwei Typen von Symbolen vorkommen:

- Nicht-Terminale (die Variablen, aus denen noch weitere Wortbestandteile abgeleitet werden sollen)
- Terminale (die "eigentlichen" Symbole)

Im vorherigen Beispiel: auf der linken Seite befindet sich immer genau ein Nicht-Terminal; man spricht von einer kontextfreien Grammatik.

Es gibt aber auch allgemeinere Grammatiken.
Es gibt sogar Grammatiken, die auf Bäumen und Graphen statt auf Wörtern arbeiten. Diese werden in der Vorlesung jedoch nicht behandelt.

Grammatiken (Definition)

Definition (Grammatik, Satzform)

Eine Grammatik G ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt:

- V ist ein Alphabet (Menge der Nicht-Terminalen oder Variablen).
- Σ ist ein Alphabet (Menge der Terminal(symbol)e) mit $V \cap \Sigma = \emptyset$, d.h., kein Zeichen ist gleichzeitig Terminal und Nicht-Terminal.
- $P \subseteq ((V \cup \Sigma)^+ \setminus \Sigma^*) \times (V \cup \Sigma)^*$ ist eine endliche Menge von Produktionen (Produktionen).
- $S \in V$ ist die Startvariable (Axiom).

Ein Wort aus $(V \cup \Sigma)^*$ nennt man auch eine Satzform.

Grammatiken (Definition)

Eine Produktion aus P ist also ein Paar (ℓ, r) von Wörtern über $V \cup \Sigma$, das zumeist als $\ell \longrightarrow r$ geschrieben wird. Dabei gilt:

- Sowohl ℓ als auch r bestehen aus Variablen und Terminalsymbolen.
- ℓ darf nicht nur aus Terminalen bestehen. Eine Regel muss also immer zumindest ein Nicht-Terminal ersetzen.

Konventionen:

- Variablen (Elemente aus V) werden mit Großbuchstaben bezeichnet: A, B, C, ..., S, T, ...
- Terminalsymbole (Elemente aus Σ) werden mit Kleinbuchstaben dargestellt: a, b, c, ...

Grammatiken (Beispiel)

Beispiel-Grammatik

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

- $V = \{S, B, C\}$
- $\Sigma = \{a, b, c\}$
- $P = \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab,$
  $bB \to bb, bC \to bc, cC \to cc\}$

Grammatiken (Ableitungen)

Wie werden die Produktionen eingesetzt, um Wörter aus der Startvariablen S zu erzeugen?

Definition (Ableitung)

Sei $G = (V, \Sigma, P, S)$ eine Grammatik und seien $u, v \in (V \cup \Sigma)^*$. Es gilt:

$$u \Rightarrow_G v \ (u \text{ geht unter } G \text{ unmittelbar über in } v),$$

falls eine Produktion $(\ell \to r) \in P$ und Wörter $x, y \in (V \cup \Sigma)^*$ existieren mit

$$u = x\ell y \qquad v = xry.$$

Man kann $\Rightarrow_G$ als binäre Relation auf $(V \cup \Sigma)^*$, d.h. als Teilmenge von $(V \cup \Sigma)^* \times (V \cup \Sigma)^*$ auffassen:

$$\Rightarrow_G = \{(u, v) \mid \exists (\ell \to r) \in P \ \exists x, y \in (V \cup \Sigma)^* : u = x\ell y, v = xry\}$$

Grammatiken (Ableitungen)

Statt $u \Rightarrow_G v$ schreibt man auch $u \Rightarrow v$, wenn klar ist, um welche Grammatik es sich handelt.

Definition (Ableitung)

Eine Folge von Wörtern $w_0, w_1, w_2, \ldots, w_n$ mit $w_0 = S$ und

$$w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$$

heißt Ableitung von w_n (aus S). Dabei darf w_n sowohl Terminale als auch Variablen enthalten, ist also eine Satzform.

Definition (die von einer Grammatik erzeugte Sprache)

Die von einer Grammatik $G = (V, \Sigma, P, S)$ erzeugte (dargestellte, definierte) Sprache ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}.$$

Dabei ist $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$, d.h. $u \Rightarrow_G^* v$ genau dann, wenn $n \geq 0$ und Satzformen $u_0, u_1, \ldots u_n \in (V \cup \Sigma)^*$ existieren mit: $u_0 = u$, $u_n = v$ und $u_i \Rightarrow_G u_{i+1}$ für alle $0 \leq i \leq n-1$.

In anderen Worten: Die von G erzeugte Sprache $L(G)$ besteht genau aus den Satzformen, die in beliebig vielen Schritten aus S abgeleitet werden können und nur aus Terminalen bestehen.

Die vorherige Beispielgrammatik G erzeugt die Sprache

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}.$$

Dabei ist $a^n = \underbrace{a \ldots a}_{n\text{-mal}}$.

Die Behauptung, dass G wirklich diese Sprache erzeugt, ist nicht offensichtlich.

Bemerkung: Ableiten ist kein deterministischer, sondern ein nichtdeterministischer Prozess. Für ein $u \in (V \cup \Sigma)^*$ kann es entweder gar kein, ein oder mehrere v geben mit $u \Rightarrow_G v$.

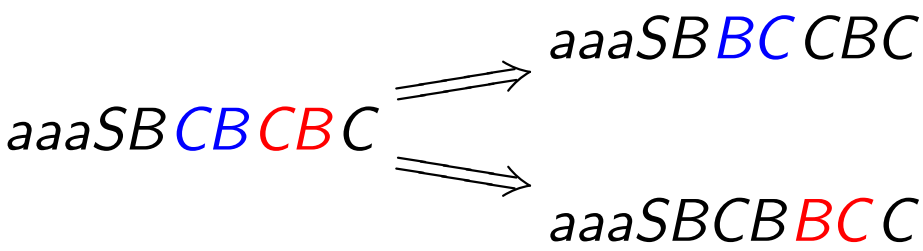
In anderen Worten: $\Rightarrow_G$ ist keine Funktion.

Dieser Nichtdeterminismus kann durch zwei verschiedene Effekte verursacht werden ...

Grammatiken und Sprachen

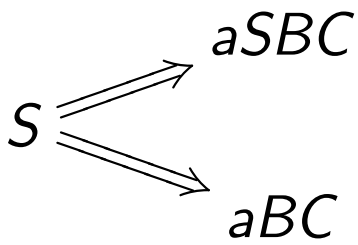
- Eine Regel ist an zwei verschiedenen Stellen anwendbar.

  Beispiel-Grammatik:



- Zwei verschiedene Produktionen sind anwendbar (entweder an der gleichen Stelle – wie unten abgebildet – oder an verschiedenen Stellen):

  Beispiel-Grammatik:



Weitere Bemerkungen:

- Es kann beliebig lange Ableitungen geben, die nie zu einem Wort aus Terminalsymbolen führen:

$$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaSBCBCBC \Rightarrow \ldots$$

- Manchmal können Ableitungen in einer Sackgasse enden, d.h., obwohl noch Variablen in einer Satzform vorkommen, ist keine Regel mehr anwendbar.

$$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow aabcBC \not\Rightarrow$$

Chomsky-Hierarchie

Typ 0 – Chomsky-0

Jede Grammatik ist vom Typ 0 (keine Einschränkung der Produktionen).

Typ 1 – Chomsky-1

Eine Grammatik $G = (V, \Sigma, P, S)$ ist vom Typ 1 (oder monoton, kontextsensitiv), falls $|\ell| \leq |r|$ für alle Produktionen $(\ell \to r) \in P$ gilt.

Typ 2 – Chomsky-2

Eine Typ-1-Grammatik $G = (V, \Sigma, P, S)$ ist vom Typ 2 (oder kontextfrei), wenn $\ell \in V$ für alle Produktionen $(\ell \to r) \in P$.

Chomsky-Hierarchie

Typ 3 – Chomsky-3

Eine Typ-2-Grammatik $G = (V, \Sigma, P, S)$ ist vom Typ 3 (oder regulär), falls zusätzlich für alle Produktionen $(\ell \to r) \in P$ gilt: $r \in \Sigma \cup \Sigma V$. D.h., die rechten Seiten von Produktionen sind entweder einzelne Terminale oder ein Terminal gefolgt von einer Variablen.

Typ-i-Sprache

Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ i ($i \in \{0, 1, 2, 3\}$), falls es eine Typ-i-Grammatik G gibt mit $L(G) = L$.

Solche Sprachen nennt man dann auch semi-entscheidbar bzw. rekursiv aufzählbar (Typ 0), kontextsensitiv (Typ 1), kontextfrei (Typ 2) oder regulär (Typ 3).

Bemerkungen:

- Woher kommt der Name "kontextsensitiv"?

  Bei kontextfreien Grammatiken gibt es nur Produktionen der Form $A \longrightarrow x$, wobei $A \in V$ und $x \in (\Sigma \cup V)^*$. Das bedeutet: A kann – unabhängig vom Kontext – durch x ersetzt werden.

  Bei den mächtigeren kontextsensitiven Grammatiken sind dagegen Produktionen der Form $uAv \longrightarrow uxv$ möglich, mit der Bedeutung: A kann nur in bestimmten Kontexten durch x ersetzt werden.

Chomsky-Hierarchie

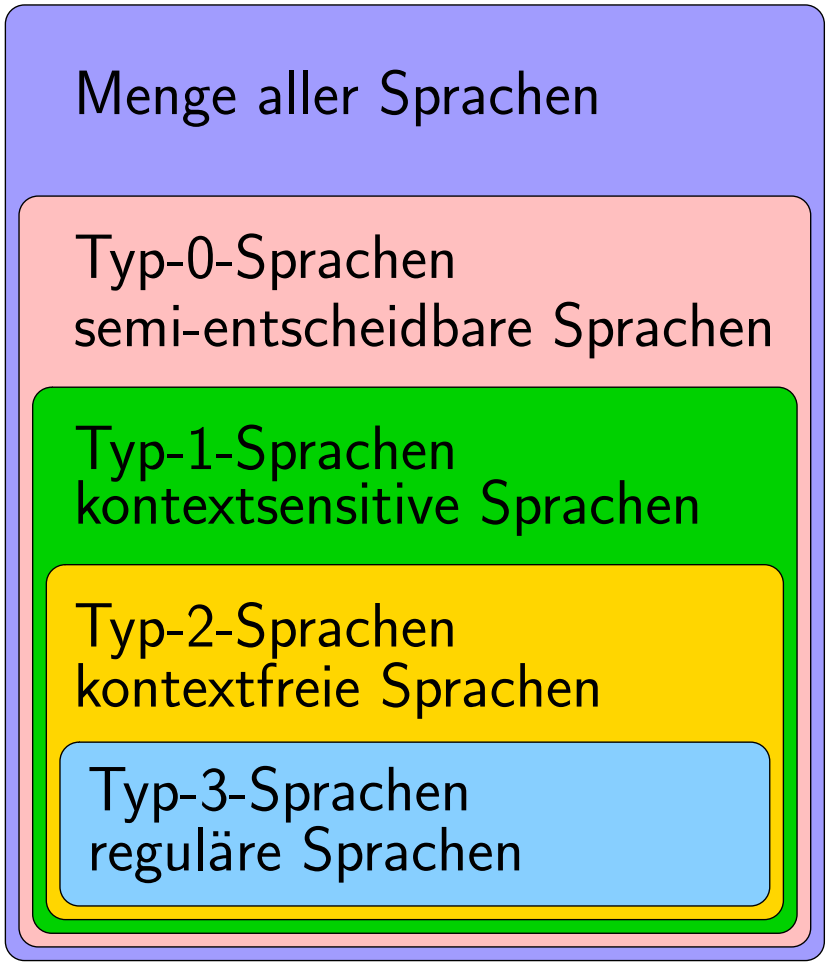
- ε-Sonderregelung: Bei Typ-1-Grammatiken (und damit auch bei regulären und kontextfreien Grammatiken) sind Produktionen der Form $\ell \to \varepsilon$ zunächst nicht zugelassen, wegen $|\ell| > 0$ und $|\ell| \leq |r|$ für alle $(\ell \to r) \in P$. Das bedeutet aber: das leere Wort ε kann nicht abgeleitet werden!

  Wir modifizieren daher die Grammatik-Definition für Typ-1 (und Typ-2, Typ-3) Grammatiken leicht und erlauben $S \to \varepsilon$, falls S das Startsymbol ist und auf keiner rechten Seite vorkommt.

Chomsky-Hierarchie

Jede Typ-i-Grammatik ist eine Typ-$(i-1)$-Grammatik (für $i \in \{1, 2, 3\}$) $\rightsquigarrow$ die entsprechenden Mengen von Sprachen sind ineinander enthalten.

Außerdem: die Inklusionen sind echt, d.h., es gibt für jedes i eine Typ-$(i-1)$-Sprache, die keine Typ-i-Sprache ist (z. B. eine kontextfreie Sprache, die nicht regulär ist). Das werden wir später zeigen.



Wortproblem

Definition (Wortproblem)

Sei $G = (V, \Sigma, P, S)$ eine Grammatik (von beliebigem Typ). Das Wortproblem für $L(G)$ ist das folgende Entscheidungsproblem:
EINGABE: Ein Wort $w \in \Sigma^*$.
FRAGE: Gilt $w \in L(G)$?

Satz (Entscheidbarkeit des Wortproblems für Typ 1)

Es gibt einen Algorithmus, der als Eingabe eine Typ-1-Grammatik $G = (V, \Sigma, P, S)$ und ein Wort $w \in \Sigma^*$ bekommt, und nach endlicher Zeit "Ja" (bzw. "Nein") ausgibt, falls $w \in L(G)$ (bzw. $w \notin L(G)$) gilt.

Man sagt auch: Das Wortproblem ist entscheidbar für Typ-1-Sprachen.

Wortproblem

Beweis:

Definiere für jedes $n \geq 1$ und $m \geq 0$ eine Menge $T_n^m \subseteq \{w \in (V \cup \Sigma)^* \mid |w| \leq n\}$ durch Induktion wie folgt:

$$\begin{aligned} T_n^0 &= \{S\} \\ T_n^{m+1} &= T_n^m \cup \{w \in (V \cup \Sigma)^* \mid |w| \leq n, \exists v \in T_n^m : v \Rightarrow_G w\} \end{aligned}$$

Durch Induktion über $m \geq 0$ folgt sofort für jedes Wort $w \in (V \cup \Sigma)^*$ mit $|w| \leq n$:

$$w \in T_n^m \iff w \text{ ist aus } S \text{ in höchstens } m \text{ Schritten ableitbar.}$$

Bemerkung: Für eine Typ-0-Grammatik G ist dies i.A. falsch.

Da $T_n^0 \subseteq T_n^1 \subseteq T_n^2 \subseteq \cdots$ und $|T_n^m| \leq \sum_{i=0}^{n} (|V| + |\Sigma|)^i$ für alle $m \geq 0$ gilt, muss ein $k \geq 0$ mit

$$T_n^k = T_n^{k+1} = T_n^{k+2} = \cdots$$

existieren.

Wortproblem

Algorithmus, um $w \in L(G)$ zu entscheiden:

input $G = (V, \Sigma, P, S)$ vom Typ 1, $w \in \Sigma^*$
$n := |w|$;
$T := \{S\}$;
repeat
    $U := T$;
    $T := U \cup \{w \in (V \cup \Sigma)^* \mid |w| \leq n, \exists v \in U : v \Rightarrow_G w\}$;
    if $w \in T$ **then return** „w gehört zu $L(G)$“
until $T = U$
return „w gehört nicht zu $L(G)$“

Bemerkung: Dieser Algorithmus ist nicht sehr effizient, da die Anzahl der Durchläufe durch die **repeat**-Schleife $\sum_{i=0}^{n}(|V| + |\Sigma|)^i \geq (|V| + |\Sigma|)^{|w|}$ sein kann.
Wir werden noch sehen, dass dies wahrscheinlich unvermeidbar ist.

Syntaxbäume und Eindeutigkeit

Wir betrachten folgende Beispiel-Grammatik zur Erzeugung von korrekt geklammerten arithmetischen Ausdrücken:

$$G = (\{E, T, F\}, \{(,), a, +, *\}, P, E)$$

mit folgender Produktionenmenge P (in abkürzender Backus-Naur-Form):

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow a \mid (E) \end{aligned}$$

Syntaxbäume und Eindeutigkeit

Für die meisten Wörter der von G erzeugten Sprache gibt es mehrere mögliche Ableitungen:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow a * F \Rightarrow a * (E)$$
$$\Rightarrow a * (E + T) \Rightarrow a * (T + T) \Rightarrow a * (F + T)$$
$$\Rightarrow a * (a + T) \Rightarrow a * (a + F) \Rightarrow a * (a + a)$$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow T * (E + T)$$
$$\Rightarrow T * (E + F) \Rightarrow T * (E + a) \Rightarrow T * (T + a)$$
$$\Rightarrow T * (F + a) \Rightarrow T * (a + a) \Rightarrow F * (a + a) \Rightarrow a * (a + a)$$

Die erste Ableitung ist eine sogenannte Linksableitung (in jedem Schritt wird das am weitesten links stehende Nicht-Terminal ersetzt), die zweite eine Rechtsableitung (in jedem Schritt wird das am weitesten rechts stehende Nicht-Terminal ersetzt).

Syntaxbäume und Eindeutigkeit

Wir bilden nun aus beiden Ableitungen den Syntaxbaum, indem wir

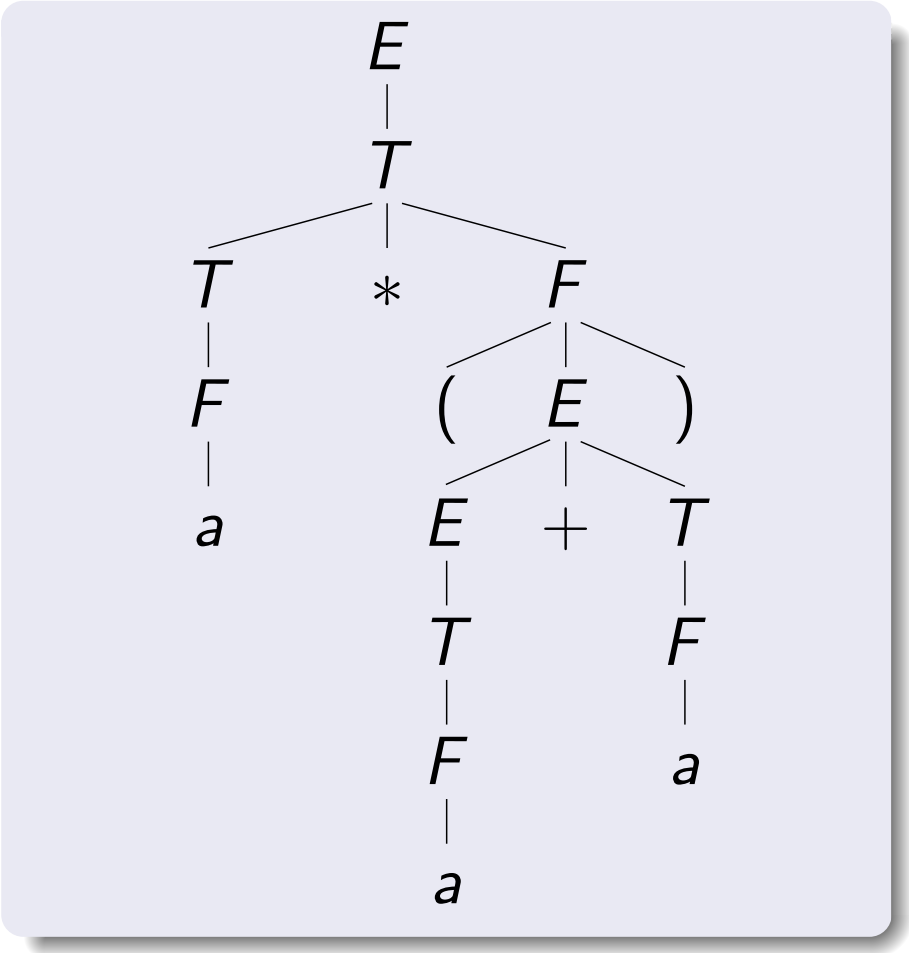
- Die Wurzel des Baums mit der Startvariablen der Grammatik beschriften.
- Bei jeder Anwendung einer Produktion $A \longrightarrow z$ zu A genau $|z|$ Kinder hinzufügen, die mit den Zeichen von z beschriftet sind.

Syntaxbäume lassen sich für alle Ableitungen von kontextfreien Grammatiken aufbauen.

Syntaxbäume und Eindeutigkeit

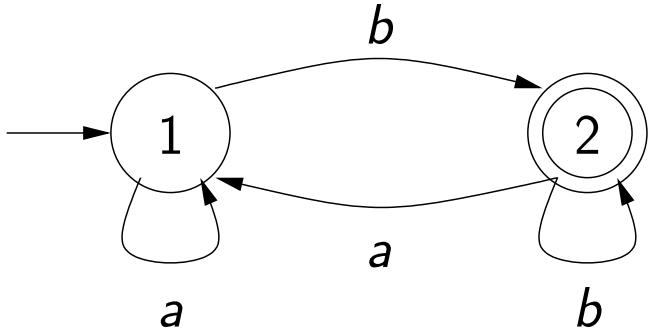
Dabei erhalten wir in beiden Fällen den gleichen Syntaxbaum.

Man sagt, eine Grammatik ist eindeutig, wenn es für jedes Wort in der erzeugten Sprache genau einen Syntaxbaum gibt
$\Longleftrightarrow$ es gibt für jedes Wort genau eine Linksableitung
$\Longleftrightarrow$ es gibt für jedes Wort genau eine Rechtsableitung.



Endliche Automaten

In diesem Abschnitt beschäftigen wir uns mit regulären Sprachen, aber zunächst unter einem anderen Blickwinkel. Statt Typ-3-Grammatiken betrachten wir zustandsbasierte Automatenmodelle, die man auch als "Spracherzeuger" bzw. "Sprachakzeptierer" betrachten kann.



Deterministische endliche Automaten

Definition (Deterministischer endlicher Automat)

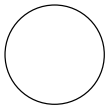
Ein (deterministischer) endlicher Automat M ist ein 5-Tupel $M = (Z, \Sigma, \delta, z_0, E)$, wobei:

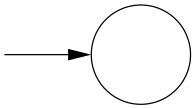
- Z eine endliche Menge von Zuständen ist,
- Σ das endliche Eingabealphabet (mit $Z \cap \Sigma = \emptyset$) ist,
- $z_0 \in Z$ der Startzustand ist,
- $E \subseteq Z$ die Menge der Endzustände ist und
- $\delta\colon Z \times \Sigma \to Z$ die Überführungsfunktion (oder Übergangsfunktion) ist.

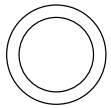
Abkürzung: DFA (deterministic finite automaton)

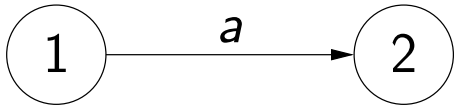
Deterministische endliche Automaten

Graphische Notation:

Zustand: 

Startzustand: 

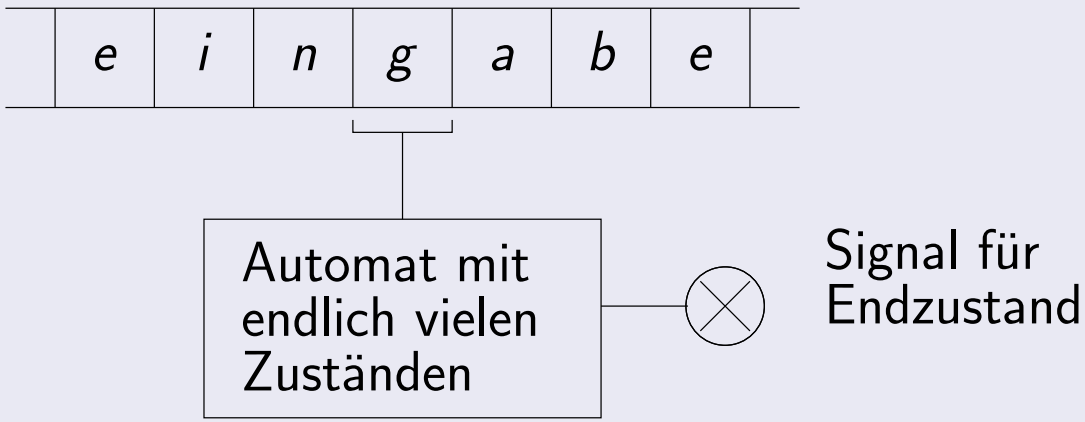
Endzustand: 

Übergang $\delta(1, a) = 2$: 

Deterministische endliche Automaten

Woher kommt der Name "endlicher Automat"?

Vorstellung von einer Maschine, die sich in endlich vielen Zuständen befinden kann, die eine Eingabe liest und die signalisiert, sobald die Eingabe akzeptiert ist.



Analogie Fahrkartenautomat:

Ein Fahrkartenautomat kann sich in folgenden Zuständen befinden:

- Keine Eingabe
- Fahrtziel ausgewählt
- Geld eingegeben
- Fahrkarte wurde ausgegeben

Das ist natürlich nur die halbe Wahrheit, da ein Fahrkartenautomat mitzählen muss, wieviel Geld bereits eingeworfen wurde. Eine Modellierung mit nur endlich vielen Zuständen ist daher stark vereinfacht.

Deterministische endliche Automaten

Die bisherige Übergangsfunktion δ eines DFA liest nur ein Zeichen auf einmal ein. Wir verallgemeinern sie daher zu einer Übergangsfunktion $\widehat{\delta}$, die die Übergänge für ganze Wörter ermittelt.

Definition (Mehr-Schritt-Übergänge eines DFA)

Zu einem gegebenen DFA $M = (Z, \Sigma, \delta, z_0, E)$ definieren wir eine Funktion $\widehat{\delta}\colon Z \times \Sigma^* \to Z$ induktiv wie folgt, wobei $z \in Z$, $x \in \Sigma^*$ und $a \in \Sigma$:

$$\begin{aligned} \widehat{\delta}(z, \varepsilon) &= z \\ \widehat{\delta}(z, ax) &= \widehat{\delta}(\delta(z, a), x) \end{aligned}$$

Definition (von einem DFA akzeptierte Sprache)

Die von einem DFA $M = (Z, \Sigma, \delta, z_0, E)$ akzeptierte Sprache ist

$$T(M) = \{x \in \Sigma^* \mid \widehat{\delta}(z_0, x) \in E\}.$$

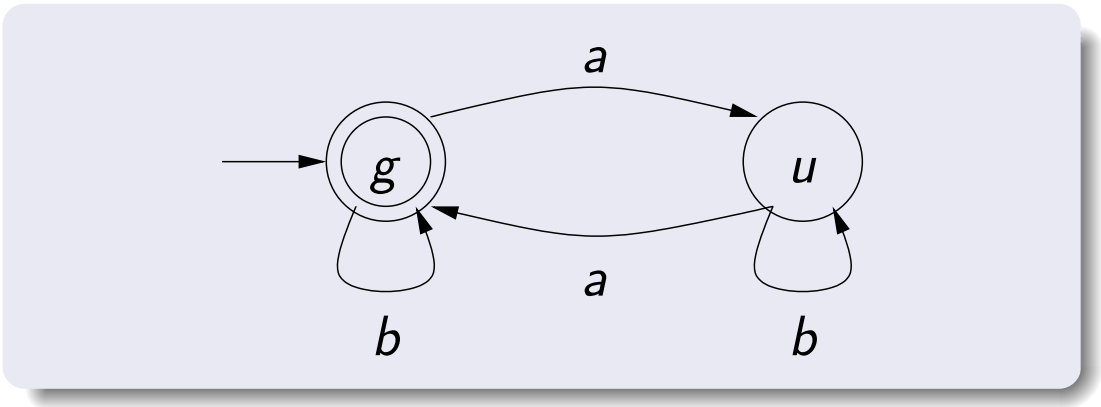
In anderen Worten:
Die Sprache kann man dadurch erhalten, indem man allen Pfaden vom Anfangszustand zu einem Endzustand folgt und dabei alle Zeichen auf den Übergängen aufsammelt.

Deterministische endliche Automaten

Beispiel 1: Wir suchen einen DFA, der folgende Sprache L akzeptiert:

$$L = \{w \in \{a, b\}^* \mid \#_a(w) \text{ gerade}\}.$$

Dabei ist $\#_a(w)$ die Anzahl der a's in w.

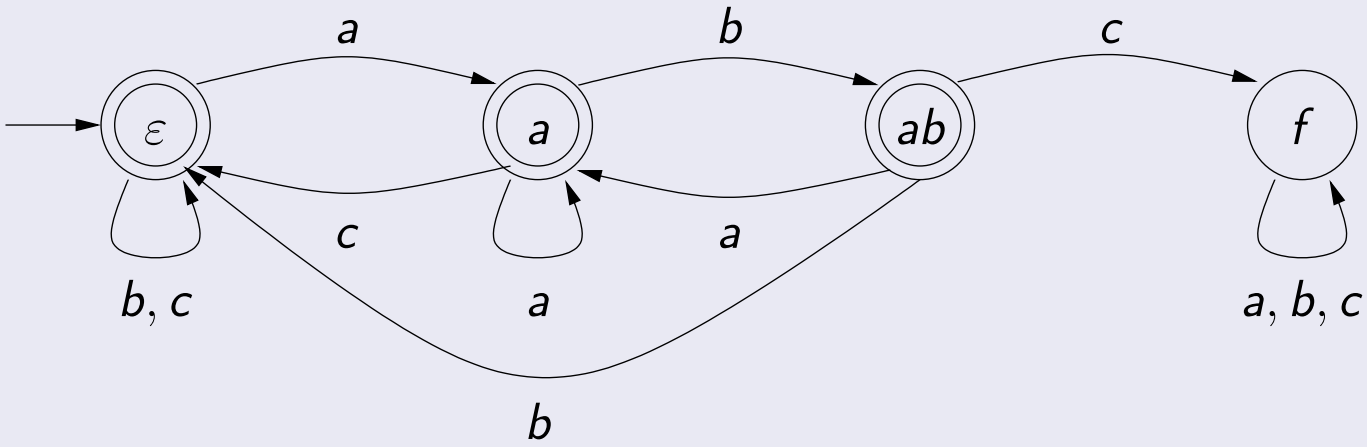


Bedeutung der Zustände:
g – gerade Anzahl a's
u – ungerade Anzahl a's

Deterministische endliche Automaten

Beispiel 2: Wir suchen einen DFA M mit

$$T(M) = \{w \in \{a, b, c\}^* \mid \text{das Teilwort } abc \text{ kommt in } w \text{ nicht vor}\}.$$



Bedeutung der Zustände:

- ε: kein Präfix von abc gelesen
- a: letztes gelesenes Zeichen war ein a
- ab: zuletzt ab gelesen
- f abc kam im bereits gelesenen Wort vor
  (Fangzustannd, Fehlerzustand)

Satz (DFAs $\longrightarrow$ reguläre Grammatik)

Jede von einem DFA akzeptierte Sprache ist regulär.

Bemerkung: Es gilt auch die umgekehrte Aussage: jede reguläre Sprache kann von einem DFA akzeptiert werden (dazu später mehr.)

Beweis:

Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA.

Zunächst modifizieren wie M so, dass $\delta(z, a) \neq z_0$ für alle $z \in Z$ und $a \in \Sigma$.

Sei hierzu $z_0' \notin Z$ und $Z' = Z \cup \{z_0'\}$.

Sei $M' = (Z', \Sigma, \delta', z_0, E')$, wobei gilt:

$$\delta'(z, a) = \begin{cases} \delta(z, a) & \text{falls } z \in Z \text{ und } \delta(z, a) \neq z_0 \\ z_0' & \text{falls } z \in Z \text{ und } \delta(z, a) = z_0 \end{cases}$$

$$\delta'(z_0', a) = \begin{cases} \delta(z_0, a) & \text{falls } \delta(z_0, a) \neq z_0 \\ z_0' & \text{falls } \delta(z_0, a) = z_0 \end{cases}$$

$$E' = \begin{cases} E & \text{falls } z_0 \notin E \\ E \cup \{z_0'\} & \text{falls } z_0 \in E \end{cases}$$

Dann gilt $\delta'(z, a) \neq z_0$ für alle $z \in Z'$ und $a \in \Sigma$ und $T(M') = T(M)$.

Deterministische endliche Automaten

Wir schreiben nun wieder Z, δ, E für Z', δ', E'.

Wir definieren nun eine Typ-3 Grammatik $G = (V, \Sigma, P, S)$ mit $L(G) = T(M)$ wie folgt:

$$\begin{aligned} V &= Z \\ S &= z_0 \\ P &= \{z \to a\ \delta(z,a) \mid z \in Z, a \in \Sigma\} \cup \\ &\quad \{z \to a \mid z \in Z, a \in \Sigma, \delta(z,a) \in E\} \cup \\ &\quad \{z_0 \to \varepsilon\} \text{ falls } z_0 \in E \end{aligned}$$

Beachte: ε-Sonderregelung ist erfüllt.

Behauptung 1: Für alle $z, z' \in Z$ und $w \in \Sigma^*$ gilt:

$$z \Rightarrow_G^* wz' \quad \Longleftrightarrow \quad \widehat{\delta}(z,w) = z'.$$

Behauptung 1 zeigt man durch Induktion über $|w|$.

Behauptung 2: Für alle $w \in \Sigma^*$ gilt: $w \in L(G) \iff w \in T(M)$.

1. Fall $w = \varepsilon$.

Es gilt:

$$\varepsilon \in L(G) \iff (z_0 \to \varepsilon) \in P \iff z_0 \in E \iff \varepsilon \in T(M)$$

2. Fall: $w \neq \varepsilon$.

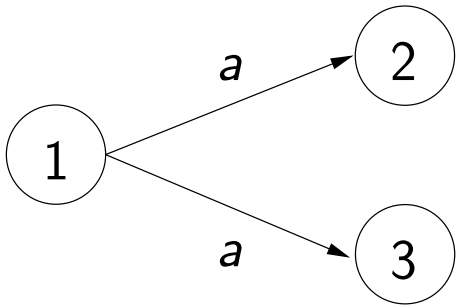
Sei $w = va$ mit $a \in \Sigma$ und $v \in \Sigma^*$. Es gilt:

$$\begin{aligned} va \in L(G) &\iff \exists z \in Z : z_0 \Rightarrow_G^* vz \Rightarrow_G va \\ &\overset{\text{Beh. 1}}{\iff} \exists z \in Z : \widehat{\delta}(z_0, v) = z, \widehat{\delta}(z, a) \in E \\ &\iff \widehat{\delta}(z_0, va) \in E \\ &\iff va \in T(M) \end{aligned}$$

Nichtdeterministische endliche Automaten

Im Gegensatz zu Grammatiken gibt es bei DFAs keine nichtdeterministischen Effekte. Das heißt, sobald das nächste Zeichen eingelesen wurde, ist klar, welcher Zustand der Folgezustand ist.

Aber: In vielen Fällen ist es natürlicher, wenn man auch nichtdeterministische Übergänge zuläßt. Das führt auch oft zu kleineren Automaten.



Nichtdeterministische endliche Automaten

Definition (Nichtdeterministischer endlicher Automat)

Ein nichtdeterministischer endlicher Automat M ist ein 5-Tupel $M = (Z, \Sigma, \delta, S, E)$, wobei:

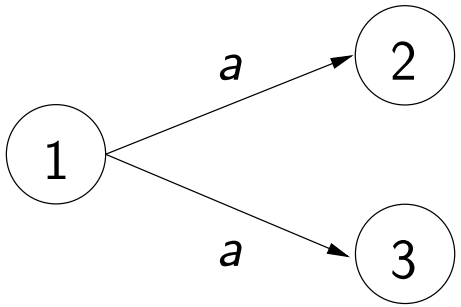
- Z ist eine endliche Menge von Zuständen,
- Σ ist das endliche Eingabealphabet (mit $Z \cap \Sigma = \emptyset$),
- $S \subseteq Z$ ist die Menge der Startzustände,
- $E \subseteq Z$ ist die Menge der Endzustände und
- $\delta\colon Z \times \Sigma \to 2^Z$ ist die Überführungsfunktion (oder Übergangsfunktion).

Abkürzung: NFA (nondeterministic finite automaton)

Nichtdeterministische endliche Automaten

Zur Erinnerung: $2^Z = \{A \mid A \subseteq Z\}$ ist die Potenzmenge von Z.

Beispiel: $\delta(1, a) = \{2, 3\}$



Nichtdeterministische endliche Automaten

Die Übergangsfunktion δ kann wieder zu einer Mehr-Schritt-Übergangsfunktion erweitert werden:

Definition (Mehr-Schritt-Übergänge eines NFA)

Zu einem gegebenen NFA $M = (Z, \Sigma, \delta, S, E)$ definieren wir eine Funktion

$$\widehat{\delta}\colon 2^Z \times \Sigma^* \to 2^Z$$

induktiv wie folgt, wobei $Y \subseteq Z$, $x \in \Sigma^*$ und $a \in \Sigma$:

$$\begin{aligned} \widehat{\delta}(Y, \varepsilon) &= Y \\ \widehat{\delta}(Y, ax) &= \widehat{\delta}\Big(\bigcup_{z \in Y} \delta(z, a),\ x\Big) \end{aligned}$$

Definition (von einem NFA akzeptierte Sprache)

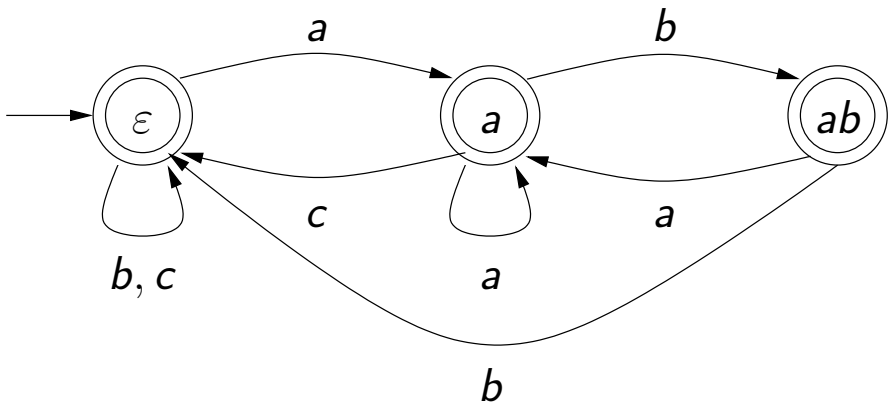
Die von einem NFA M akzeptierte Sprache ist

$$T(M) = \{x \in \Sigma^* \mid \widehat{\delta}(S, x) \cap E \neq \emptyset\}.$$

In anderen Worten: ein Wort x wird akzeptiert, genau dann wenn es einen Pfad von einem Anfangszustand zu einem Endzustand gibt, dessen Übergänge mit den Zeichen von x markiert sind (es könnte auch mehrere solche Pfade geben).

Nichtdeterministische endliche Automaten

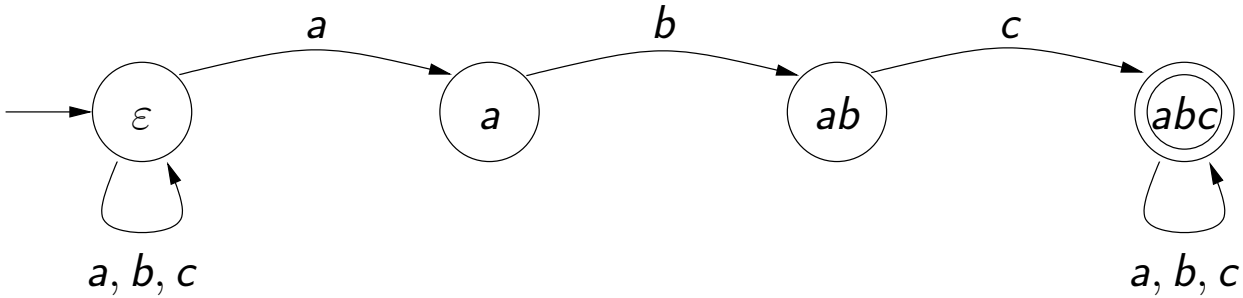
Beispiel 1: bei nicht-deterministischen Automaten darf auch $\delta(z, a) = \emptyset$ für ein $a \in \Sigma$ gelten, das heißt, es muss nicht für jedes Alphabetsymbol immer einen Übergang geben und der Fangzustand kann weggelassen werden.



Beispiel 2: gesucht ist ein NFA, der die Sprache

$$L = \{w \in \{a, b, c\}^* \mid \text{das Teilwort } abc \text{ kommt in } w \text{ vor}\}$$

akzeptiert.



Dieser Automat entscheidet zu einem bestimmten Zeitpunkt nicht-deterministisch, dass jetzt das Teilwort *abc* beginnt.

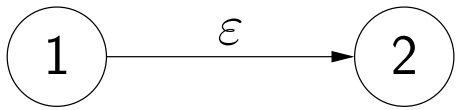
Nichtdeterministische endliche Automaten

Andere Interpretation: jedes Mal, wenn eine nicht-deterministische Verzweigung möglich ist, werden mehrere "Paralleluniversen" erzeugt, in denen verschiedene Kopien der Maschine die verschiedenen möglichen Pfade erkunden. Das Wort wird akzeptiert, wenn es in einem dieser Paralleluniversen akzeptiert wird.

Nichtdeterministische endliche Automaten

Es gibt auch nichtdeterministische Automaten mit sogenannten ε-Kanten (spontante Übergänge, bei denen kein Alphabetsymbol eingelesen wird). Diese werden jedoch in der Vorlesung im allgemeinen nicht benutzt.

Beispiel für eine ε-Kante:

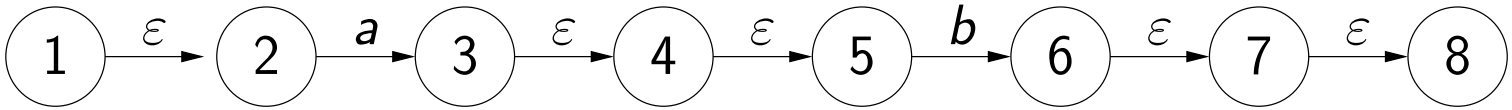


Neue Übergangsfunktion: $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \to 2^Z$

Im obigen Beispiel: $\delta(1, \varepsilon) = \{2\}$.

Nichtdeterministische endliche Automaten

Neue Mehr-Schritt-Übergangsfunktion: $\widehat{\delta}\colon 2^Z \times \Sigma^* \to 2^Z$. Dabei dürfen zwischen dem Einlesen der Zeichen beliebig viele ε-Übergänge gemacht werden.



$$\widehat{\delta}(\{1\}, ab) = \{6, 7, 8\}$$

Äquivalenz von NFAs mit und ohne ε-Übergänge

Jeder NFA mit ε-Übergängen kann in einen NFA ohne ε-Übergänge umgewandelt werden, ohne die akzeptierte Sprache zu ändern und ohne die Anzahl der Zustände zu erhöhen.

(Ohne Beweis.)

Satz (NFAs $\rightarrow$ DFAs; Rabin, Scott)

Jede von einem NFA akzeptierbare Sprache kann auch von einem DFA akzeptiert werden.

Beweis:

Idee: Wir lassen die verschiedenen "Paralleluniversen" von einem Automaten simulieren. Dieser merkt sich, in welchen Zuständen er sich gerade befindet.

Das heißt, die Zustände dieses Automaten sind Mengen von Zuständen des ursprünglichen Automaten. Man nennt diese Konstruktion daher auch Potenzmengenkonstruktion.

NFAs, DFAs und reguläre Grammatiken

Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA.

Definiere den DFA

$$M' = (2^Z, \Sigma, \gamma, S, F)$$

wobei

$$\begin{aligned} \gamma(Y, a) &= \bigcup_{z \in Y} \delta(z, a) \text{ für } Y \subseteq Z, a \in \Sigma \\ F &= \{Y \subseteq Z \mid Y \cap E \neq \emptyset\} \end{aligned}$$

Durch Induktion über die Länge des Wortes $w \in \Sigma^*$ zeigen wir für alle $Y \subseteq Z$:

$$\widehat{\gamma}(Y, w) = \widehat{\delta}(Y, w)$$

NFAs, DFAs und reguläre Grammatiken

Induktionsanfang: $\widehat{\gamma}(Y, \varepsilon) = Y = \widehat{\delta}(Y, \varepsilon)$

Induktionsschritt: Sei $w = ax$ mit $a \in \Sigma$ und $x \in \Sigma^*$. Dann gilt:

$$\begin{aligned}
\widehat{\gamma}(Y, ax) &= \widehat{\gamma}(\gamma(Y, a), x) \\
&\overset{\text{IA}}{=} \widehat{\delta}(\gamma(Y, a), x) \\
&= \widehat{\delta}\left(\bigcup_{z \in Y} \delta(z, a), x\right) \\
&= \widehat{\delta}(Y, ax)
\end{aligned}$$

Also gilt für jedes Wort $w \in \Sigma^*$:

$$\begin{aligned}
w \in T(M') &\iff \widehat{\gamma}(S, w) \in F \\
&\iff \widehat{\delta}(S, w) \cap E \neq \emptyset \\
&\iff w \in T(M)
\end{aligned}$$

Beispiel: Für $k \geq 1$ sei

$$L_k = \{w \in \{0,1\}^* \mid \ |w| \geq k, \text{das } k\text{-letzte Zeichen von } w \text{ ist } 0\}.$$

(A) Es gibt einen NFA M mit $k+1$ Zuständen und $T(M) = L_k$.

(B) Es gibt keinen DFA M mit weniger als 2^k Zuständen und $T(M) = L_k$.

Beweis von (B): Angenommen, $M = (Z, \Sigma, \delta, z_0, E)$ wäre ein DFA mit weniger als 2^k Zuständen und $T(M) = L_k$.

Dann gibt es Wörter $w_1, w_2 \in \{0,1\}^k$ mit $w_1 \neq w_2$ und $\widehat{\delta}(z_0, w_1) = \widehat{\delta}(z_0, w_2)$ (denn es gibt nur 2^k viele Wörter in $\{0,1\}^k$).

Sei $i \in \{1, \ldots, k\}$ die erste Position, an der sich w_1 und w_2 unterscheiden.

Sei $w \in \{0,1\}^{i-1}$ beliebig.

Dann existieren Wörter $v, v' \in \{0,1\}^{k-i}$ und $u \in \{0,1\}^{i-1}$ mit (o.B.d.A.)

$$w_1 w = u0vw \quad \text{und} \quad w_2 w = u1v'w.$$

Wegen $|vw| = |v'w| = k - i + i - 1 = k - 1$ gilt

$$w_1 w \in L_k \quad \text{und} \quad w_2 w \notin L_k.$$

Aber:

$$\widehat{\delta}(z_0, w_1 w) = \widehat{\delta}(\widehat{\delta}(z_0, w_1), w) = \widehat{\delta}(\widehat{\delta}(z_0, w_2), w) = \widehat{\delta}(z_0, w_2 w),$$

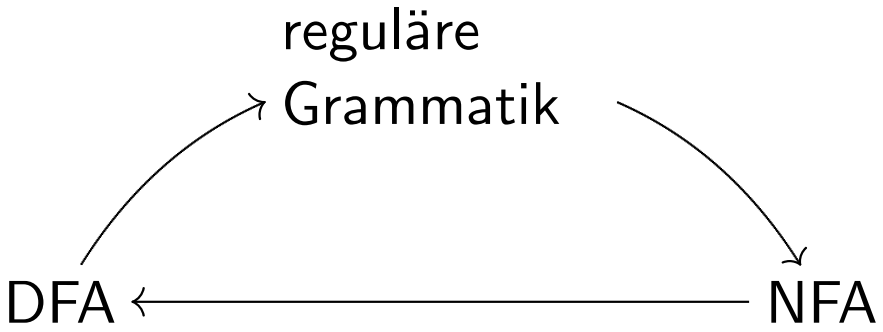
d.h. $w_1 w \in L_k \Leftrightarrow w_2 w \in L_k$. Widerspruch!

NFAs, DFAs und reguläre Grammatiken

Wir können nun

- NFAs in DFAs umwandeln
- DFAs in reguläre Grammatiken umwandeln

Es fehlt noch die Richtung "reguläre Grammatik $\rightarrow$ NFA", dann haben wir die Äquivalenz aller dieser Formalismen gezeigt.



Satz (Reguläre Grammatiken $\longrightarrow$ NFAs)

Zu jeder regulären Grammatik G gibt es einen NFA M mit $L(G) = T(M)$.

Beweis:

Sei $G = (V, \Sigma, P, S)$ eine reguläre Grammatik.

Wir definieren den NFA $M = (Z, \Sigma, \delta, S', E)$, wobei:

$$Z = V \cup \{X\} \quad \text{mit } X \notin V$$

$$\delta(A, a) = \{B \mid (A \to aB) \in P\} \cup \{X \mid (A \to a) \in P\} \text{ für } A \in V, a \in \Sigma$$

$$\delta(X, a) = \emptyset \text{ für } a \in \Sigma$$

$$S' = \{S\}$$

$$E = \begin{cases} \{S, X\} & \text{falls } (S \to \varepsilon) \in P \\ \{X\} & \text{falls } (S \to \varepsilon) \notin P \end{cases}$$

NFAs, DFAs und reguläre Grammatiken

Wegen der Konstruktion gilt

$$\varepsilon \in L(G) \iff (S \to \varepsilon) \in P \iff S' \cap E \neq \emptyset \iff \varepsilon \in T(M).$$

Wir müssen also noch für alle Wörter $w \in \Sigma^+$ zeigen:
$w \in L(G) \iff w \in T(M)$.

Behauptung: Für alle $w \in \Sigma^*$ und alle $A, B \in V$ gilt:

$$A \Rightarrow_G^* wB \iff B \in \widehat{\delta}(\{A\}, w)$$

Wir zeigen diese Behauptung durch Induktion über $|w|$.

IA: $w = \varepsilon$. Es gilt:

$$A \Rightarrow_G^* B \iff A = B \iff B \in \{A\} = \widehat{\delta}(\{A\}, \varepsilon)$$

IS: Sei $w = av$ ($a \in \Sigma$, $v \in \Sigma^*$) und gelte die Behauptung bereits für das Wort v.

$$\begin{aligned} A \Rightarrow_G^* avB &\iff \exists C \in V : (A \to aC) \in P \text{ und } C \Rightarrow_G^* vB \\ &\iff \exists C \in V : C \in \delta(A, a) \text{ und } B \in \widehat{\delta}(\{C\}, v) \\ &\iff B \in \widehat{\delta}(\{A\}, av) \end{aligned}$$

Dies zeigt die Behauptung.

Sei nun $w \in \Sigma^+$, etwa $w = va$ mit $a \in \Sigma$.

Dann gilt:

$$\begin{aligned} va \in L(G) &\iff \exists A \in V : S \Rightarrow_G^* vA \text{ und } (A \to a) \in P \\ &\overset{\text{Beh.}}{\iff} \exists A \in V : A \in \widehat{\delta}(\{S\}, v) \text{ und } X \in \delta(A, a) \\ &\iff X \in \widehat{\delta}(\{S\}, va) \\ &\iff va \in T(M) \end{aligned}$$

Beachte für die letzte Äquivalenz: Entweder

- X ist der einzige Endzustand von M (falls $(S \to \varepsilon) \notin P$) oder
- S ist der zweite Endzustand (falls $(S \to \varepsilon) \in P$), ist dann jedoch nicht Ziel einer Transition von M (d.h. $S \notin \delta(A, a)$ für alle $A \in Z$, $a \in \Sigma$), weil S nicht auf der rechten Seite einer Produktion aus P vorkommen darf (ε-Sonderregelung). □

NFAs, DFAs und reguläre Grammatiken

Zwischenzusammenfassung

Wir haben verschiedene Modelle zur Beschreibung regulärer Sprachen kennengelernt:

- Reguläre Grammatiken: Schaffen die Verbindung zur Chomsky-Hierarchie. Werden zur Erzeugung von Sprachen eingesetzt. Sind weniger gut geeignet, um zu entscheiden, ob ein bestimmtes Wort zur Sprache gehört.
- NFAs: Erlauben oft kleine, kompakte Darstellungen von Sprachen. Sind, wegen ihres Nichtdeterminismus, genauso wie Grammatiken weniger gut für die Lösung des Wortproblems geeignet. Besitzen aber eine intuitive graphische Notation.
- DFAs: Können gegenüber äquivalenten NFAs exponentiell größer sein. Sobald jedoch ein DFA vorliegt, erlaubt dieser eine effiziente Lösung des Wortproblems (einfach den Übergängen des Automaten nachlaufen und überprüfen, ob ein Endzustand erreicht wird).

Reguläre Ausdrücke

Alle Modelle benötigen jedoch relativ viel Schreibaufwand und Platz für die Notation. Gesucht wird also eine kompaktere Repräsentation.
Dies sind reguläre Ausdrücke.

Definition (reguläre Ausdrücke)

Die Menge $\mathrm{Reg}(\Sigma)$ der regulären Ausdrücke über dem Alphabet Σ ist die kleinste Menge mit folgenden Eigenschaften:

- $\emptyset \in \mathrm{Reg}(\Sigma)$, $\varepsilon \in \mathrm{Reg}(\Sigma)$, $\Sigma \subseteq \mathrm{Reg}(\Sigma)$.
- Wenn $\alpha, \beta \in \mathrm{Reg}(\Sigma)$, dann auch $\alpha\beta, (\alpha|\beta), (\alpha)^* \in \mathrm{Reg}(\Sigma)$.

Bemerkungen:

- Statt $(\alpha|\beta)$ wird oft auch $(\alpha + \beta)$ geschrieben.
- überflüssige Klammern lassen wir häufig weg.
  Z. B. $(a|b)^*$ anstatt $((a|b))^*$.

Reguläre Ausdrücke

Nach der Festlegung der Syntax regulärer Ausdrücke, müssen wir auch deren Bedeutung festlegen, d.h., welcher reguläre Ausdruck steht für welche Sprache?

Definition (Sprache eines regulären Ausdrucks)

- $L(\emptyset) = \emptyset, \quad L(\varepsilon) = \{\varepsilon\}, \quad L(a) = \{a\}$ für $a \in \Sigma$.
- $L(\alpha\beta) = L(\alpha)L(\beta)$, wobei $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ für zwei Sprachen L_1, L_2.
- $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$
- $L((\alpha)^*) = (L(\alpha))^*$, wobei $L^* = \{w_1 \cdots w_n \mid n \in \mathbb{N}_0, w_i \in L\}$ für eine Sprache L

Bemerkungen zum $*$-Operator: $L^* = \{w_1 \cdots w_n \mid n \in \mathbb{N}_0, w_i \in L\}$

- Dieser Operator wird oft Kleenesche Hülle genannt. Nur durch ihn kann man unendliche Sprachen erzeugen.
- L^* enthält immer das leere Wort ε (siehe Definition).
- Beispiel für die Anwendung des $*$-Operators:

  Sei $L = \{a, bb, cc\}$

  $\rightsquigarrow$

  $L^* = \{\varepsilon, a, bb, cc, aa, abb, acc, bba, bbbb, bbcc, cca, ccbb, cccc, \ldots\}$

  Alle Kombinationen beliebiger Länge sind möglich.

Reguläre Ausdrücke

Beispiele für reguläre Ausdrücke über dem Alphabet $\Sigma = \{a, b\}$.

Beispiel 1: Sprache aller Wörter, die mit a beginnen und mit bb enden

$$\alpha = a(a|b)^* bb$$

Beispiel 2: Sprache aller Wörter, die das Teilwort aba enthalten.

$$\alpha = (a|b)^* aba(a|b)^*$$

Beispiel 3: Sprache aller Wörter, die gerade viele a's enthalten.

$$\alpha = (b^* ab^* a)^* b^* \qquad \text{oder} \qquad \alpha = (b \mid ab^* a)^*$$

Satz (reguläre Ausdrücke $\longrightarrow$ NFAs)

Zu jedem regulären Ausdruck γ gibt es einen NFA M mit $L(\gamma) = T(M)$.

Beweis: Induktion über den Aufbau von γ

IA: Für $\gamma = \emptyset$, $\gamma = \varepsilon$, $\gamma = a$ $(a \in \Sigma)$ gibt es offensichtlich entsprechende NFAs.

IS: Sei nun $\gamma = \alpha\beta$. Dann gibt es NFAs

$$\begin{aligned} M_\alpha &= (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha) \\ M_\beta &= (Z_\beta, \Sigma, \delta_\beta, S_\beta, E_\beta) \end{aligned}$$

mit $T(M_\alpha) = L(\alpha)$ und $T(M_\beta) = L(\beta)$.

Wir können annehmen, dass $Z_\alpha \cap Z_\beta = \emptyset$.

Reguläre Ausdrücke

Wir verknüpfen nun M_α und M_β sequentiell zu einem NFA M:

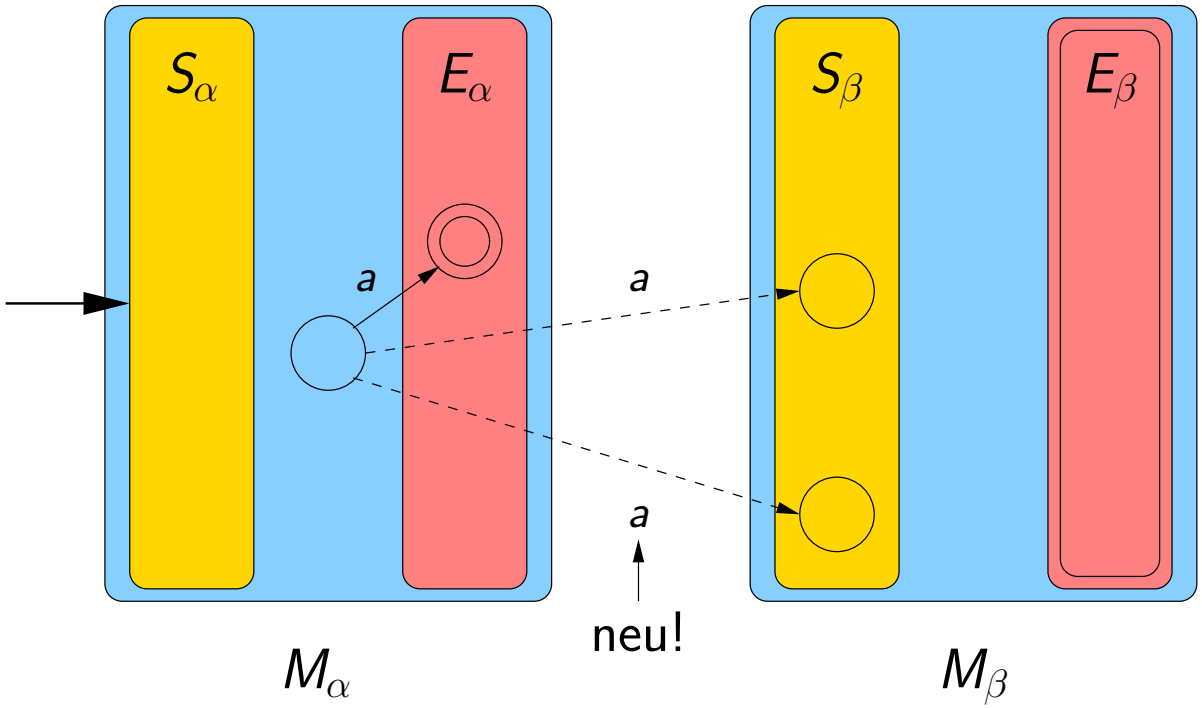
- M hat als Zustände die Vereinigung beider Zustandsmengen, die gleichen Startzustände wie M_α und die gleichen Endzustände wie M_β. Falls $\varepsilon \in L(\alpha)$, so sind auch die Startzustände von M_β Startzustände von M.
- Alle Übergänge von M_α bzw. M_β bleiben erhalten. Alle Zustände, die einen Pfeil zu einem Endzustand von M_α haben, erhalten zusätzlich genauso beschriftete Pfeile zu allen Startzuständen von M_β.

Formal: $M = (Z_\alpha \cup Z_\beta, \Sigma, \delta, S, E_\beta)$, wobei

$$S = \begin{cases} S_\alpha & \text{falls } \varepsilon \notin L(\alpha) \\ S_\alpha \cup S_\beta & \text{falls } \varepsilon \in L(\alpha) \end{cases}$$

$$\delta(z,a) = \begin{cases} \delta_\beta(z,a) & \text{für } z \in Z_\beta \\ \delta_\alpha(z,a) & \text{für } z \in Z_\alpha \text{ mit } \delta_\alpha(z,a) \cap E_\alpha = \emptyset \\ \delta_\alpha(z,a) \cup S_\beta & \text{für } z \in Z_\alpha \text{ mit } \delta_\alpha(z,a) \cap E_\alpha \neq \emptyset \end{cases}$$

Reguläre Ausdrücke



Es gilt $T(M) = T(M_\alpha)T(M_\beta) = L(\alpha)L(\beta) = L(\alpha\beta) = L(\gamma)$

Reguläre Ausdrücke

Sei nun $\gamma = (\alpha \mid \beta)$. Dann gibt es NFAs

$$\begin{aligned} M_\alpha &= (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha) \\ M_\beta &= (Z_\beta, \Sigma, \delta_\beta, S_\beta, E_\beta) \end{aligned}$$

mit $T(M_\alpha) = L(\alpha)$ und $T(M_\beta) = L(\beta)$.

Wir können annehmen, dass $Z_\alpha \cap Z_\beta = \emptyset$.

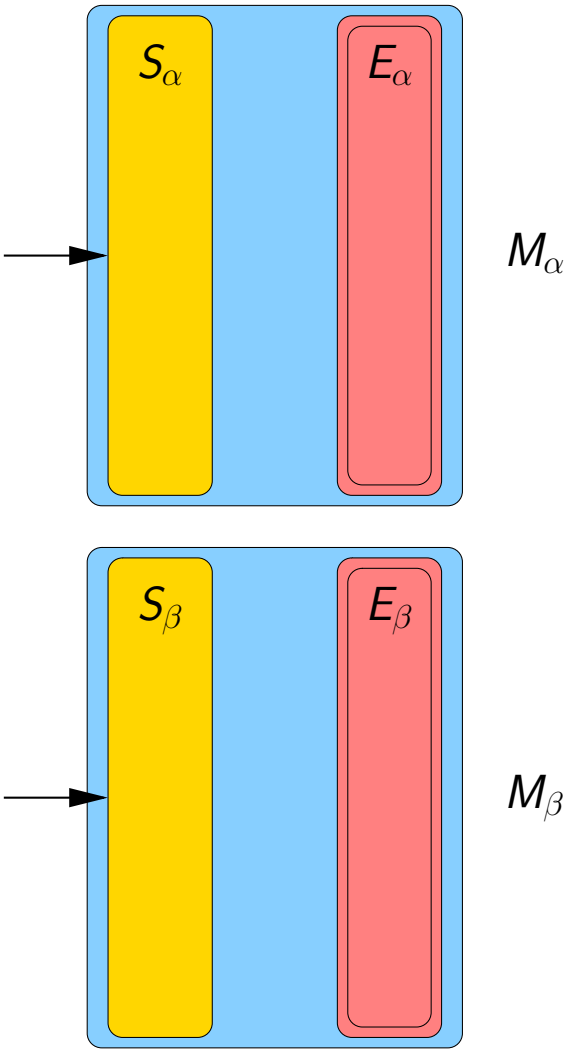
Wir bauen nun aus diesen zwei NFAs einen Vereinigungs-NFA M:

- M hat als Zustände die Vereinigung beider Zustandsmengen. Ebenso ergeben sich die Startzustände als Vereinigung der Startzustandsmengen und die Endzustände als Vereinigung der Endzustandsmengen.
- Alle Übergänge von M_α bzw. M_β bleiben erhalten.

Formal: $M = (Z_\alpha \cup Z_\beta, \Sigma, \delta, S_\alpha \cup S_\beta, E_\alpha \cup E_\beta)$, wobei

$$\delta(z, a) = \begin{cases} \delta_\alpha(z, a) & \text{für } z \in Z_\alpha \\ \delta_\beta(z, a) & \text{für } z \in Z_\beta \end{cases}$$

Reguläre Ausdrücke



$$\begin{aligned} \text{Es gilt } T(M) &= T(M_\alpha) \cup T(M_\beta) \\ &= L(\alpha) \cup L(\beta) \\ &= L(\alpha \mid \beta) \\ &= L(\gamma) \end{aligned}$$

Reguläre Ausdrücke

Sei nun $\gamma = (\alpha)^*$. Dann gibt es einen NFA

$$M_\alpha = (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha)$$

mit $T(M_\alpha) = L(\alpha)$.

Wir bauen aus diesem NFA nun wie folgt einen NFA M:

- Falls $\varepsilon \notin T(M_\alpha)$, so gibt es einen zusätzlichen Zustand, der sowohl Start- als auch Endzustand ist (damit auch das leere Wort erkannt wird).
- Die anderen Zustände, Start- und Endzustände sowie Übergänge bleiben erhalten.
- Alle Zustände, die einen Pfeil zu einem Endzustand von M_α haben, erhalten zusätzlich genauso beschriftete Pfeile zu allen Startzuständen von M_α (Rückkopplung).

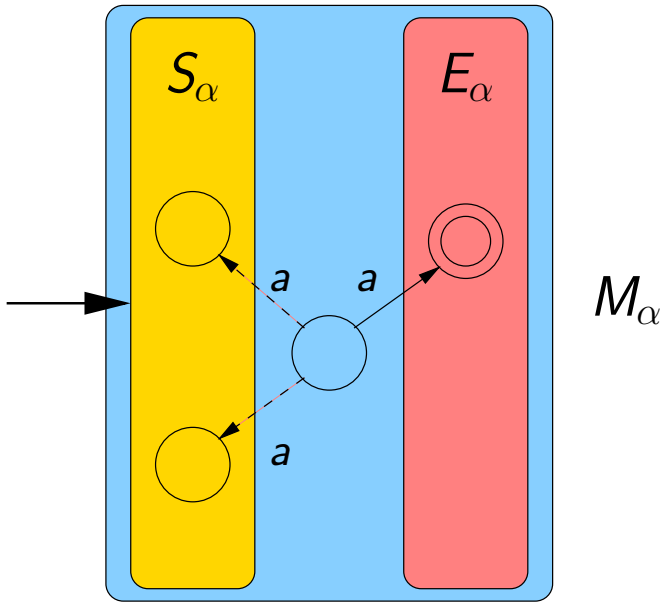
Reguläre Ausdrücke

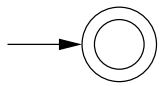
Formal: $M = (Z, \Sigma, \delta, S, E)$, wobei:

$$
\begin{aligned}
Z &= \begin{cases} Z_\alpha & \text{falls } \varepsilon \in L(\alpha) \\ Z_\alpha \cup \{s_0\} & \text{falls } \varepsilon \notin L(\alpha) \end{cases} \\
S &= \begin{cases} S_\alpha & \text{falls } \varepsilon \in L(\alpha) \\ S_\alpha \cup \{s_0\} & \text{falls } \varepsilon \notin L(\alpha) \end{cases} \\
E &= \begin{cases} E_\alpha & \text{falls } \varepsilon \in L(\alpha) \\ E_\alpha \cup \{s_0\} & \text{falls } \varepsilon \notin L(\alpha) \end{cases} \\
\delta(z,a) &= \begin{cases} \delta_\alpha(z,a) & \text{für } z \in Z_\alpha \text{ mit } \delta_\alpha(z,a) \cap E_\alpha = \emptyset \\ \delta_\alpha(z,a) \cup S_\alpha & \text{für } z \in Z_\alpha \text{ mit } \delta_\alpha(z,a) \cap E_\alpha \neq \emptyset \end{cases}
\end{aligned}
$$

Hierbei gilt $s_0 \notin Z_\alpha$.

Reguläre Ausdrücke





evtl. zusätzl. Zustand

Es gilt $T(M) = (T(M_\alpha))^* = (L(\alpha))^* = L(\alpha^*) = L(\gamma)$.

Reguläre Ausdrücke

Satz (DFAs $\rightarrow$ Reguläre Ausdrücke)

Zu jedem DFA M gibt es einen regulären Ausdruck γ mit $T(M) = L(\gamma)$.

Beweis:

Sei $M = (\{z_1, \ldots, z_n\}, \Sigma, \delta, z_1, E)$ ein DFA.

Wir konstruieren einen regulären Ausdruck γ mit $T(M) = L(\gamma)$.

Für ein Wort $w \in \Sigma^*$ sei

$$\text{Pref}(w) = \{u \in \Sigma^* \mid \exists v : w = uv, \varepsilon \neq u \neq w\}$$

die Menge aller nicht-leeren echten Präfixe von w.

Für $i, j \in \{1, \ldots, n\}$ und $k \in \{0, \ldots, n\}$ sei

$$L_{i,j}^k = \{w \in \Sigma^* \mid \widehat{\delta}(z_i, w) = z_j, \forall u \in \text{Pref}(w) : \widehat{\delta}(z_i, u) \in \{z_1, \ldots, z_k\}\}.$$

Reguläre Ausdrücke

Intuitiv: Ein Wort w gehört zu $L_{i,j}^k$ genau dann, wenn w den Zustand z_i in den Zustand z_j überführt, und dabei kein Zwischenzustand (ausser ganz am Anfang und ganz am Ende) aus $\{z_{k+1}, \ldots, z_n\}$ vorkommt.

Wir konstruieren für alle $i, j \in \{1, \ldots, n\}$ und $k \in \{0, \ldots, n\}$ reguläre Ausdrücke $\gamma_{i,j}^k$ mit $L(\gamma_{i,j}^k) = L_{i,j}^k$.

Falls $E = \{z_{i_1}, z_{i_2}, \ldots, z_{i_m}\}$, ergibt sich dann

$$L(\gamma_{1,i_1}^n \mid \gamma_{1,i_2}^n \mid \cdots \mid \gamma_{1,i_m}^n) = T(M).$$

Konstruktion von $\gamma_{i,j}^k$ durch Induktion über $k \in \{0, \ldots, n\}$.

IA: $k = 0$. Es gilt:

$$L_{i,j}^0 = \begin{cases} \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_j\} & \text{falls } i = j \\ \{a \in \Sigma \mid \delta(z_i, a) = z_j\} & \text{falls } i \neq j \end{cases}$$

Einen regulären Ausdruck $\gamma_{i,j}^0$ mit $L(\gamma_{i,j}^0) = L_{i,j}^0$ können wir leicht angeben.

IS: Sei $0 \leq k < n$ und seien die regulären Ausdrücke $\gamma_{p,q}^k$ für alle $p, q \in \{1, \ldots, n\}$ bereits konstruiert.

Sei $i, j \in \{1, \ldots, n\}$.

Behauptung: $L_{i,j}^{k+1} = L_{i,j}^k \cup L_{i,k+1}^k (L_{k+1,k+1}^k)^* L_{k+1,j}^k$

Begründung:

$\subseteq$: Sei $w \in L_{i,j}^{k+1}$ und sei $\ell \geq 0$ so, dass der Zustand z_{k+1} auf dem eindeutigen mit w beschrifteten Pfad von z_i nach z_j genau ℓ mal als echter Zwischenzustand auftaucht.

1.Fall: $\ell = 0$, d.h. z_{k+1} kommt garnicht als echter Zwischenzustand vor.

$\rightsquigarrow \quad w \in L_{i,j}^k$

2.Fall: $\ell > 0$.

$\rightsquigarrow$ w kann als $w = w_0 w_1 \cdots w_{\ell-1} w_\ell$ geschrieben werden, wobei:

$$\begin{aligned} \widehat{\delta}(z_i, w_0) &= z_{k+1} \\ \widehat{\delta}(z_{k+1}, w_p) &= z_{k+1} \text{ für } 1 \leq p \leq \ell - 1 \\ \widehat{\delta}(z_{k+1}, w_\ell) &= z_j \end{aligned}$$

$\rightsquigarrow$ $w_0 \in L^k_{i,k+1}$, $w_p \in L^k_{k+1,k+1} (1 \leq p \leq \ell - 1)$, $w_\ell \in L^k_{k+1,j}$

$\rightsquigarrow$ $w = w_0 (w_1 \cdots w_{\ell-1}) w_\ell \in L^k_{i,k+1} (L^k_{k+1,k+1})^* L^k_{k+1,j}$

$\supseteq$: $L^k_{i,j} \subseteq L^{k+1}_{i,j}$ ist offensichtlich.

Falls $w \in L^k_{i,k+1} (L^k_{k+1,k+1})^* L^k_{k+1,j}$, existiert ein $\ell \geq 1$ und eine Faktorisierung $w = w_0 w_1 \cdots w_{\ell-1} w_\ell$ mit

$$w_0 \in L^k_{i,k+1}, \; w_1, \ldots, w_{\ell-1} \in L^k_{k+1,k+1}, \; w_\ell \in L^k_{k+1,j}.$$

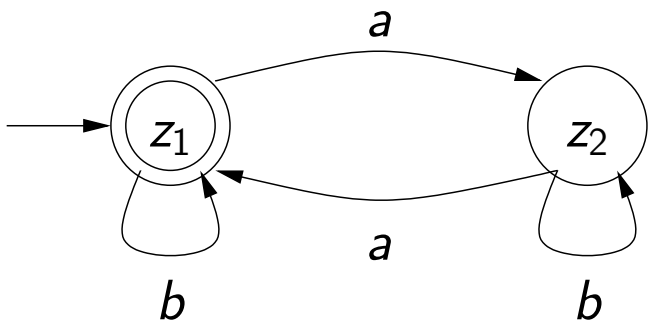
Hieraus ergibt sich leicht $w \in L^{k+1}_{i,j}$. Dies zeigt die Behauptung.

Da alle regulären Ausdrücke $\gamma^k_{p,q}$ bereits konstruiert sind, können wir setzen:

$$\gamma^{k+1}_{i,j} = \gamma^k_{i,j} \mid \gamma^k_{i,k+1}(\gamma^k_{k+1,k+1})^*\gamma^k_{k+1,j}$$

□

Beispiel: Betrachte den folgenden DFA:



Damit ergibt sich (bei Durchführung offensichtlicher Vereinfachungen):

$$\gamma^0_{1,1} = \varepsilon | b \qquad \gamma^0_{1,2} = a \qquad \gamma^0_{2,1} = a \qquad \gamma^0_{2,2} = \varepsilon | b$$

$$\gamma^1_{1,1} = \gamma^0_{1,1} \,|\, \gamma^0_{1,1}(\gamma^0_{1,1})^* \gamma^0_{1,1} = \varepsilon \,|\, b \,|\, (\varepsilon|b)(\varepsilon|b)^*(\varepsilon|b) = b^*$$
$$\gamma^1_{1,2} = \gamma^0_{1,2} \,|\, \gamma^0_{1,1}(\gamma^0_{1,1})^* \gamma^0_{1,2} = a \,|\, (\varepsilon|b)(\varepsilon|b)^* a = b^* a$$
$$\gamma^1_{2,1} = \gamma^0_{2,1} \,|\, \gamma^0_{2,1}(\gamma^0_{1,1})^* \gamma^0_{1,1} = a \,|\, a(\varepsilon|b)(\varepsilon|b)^* = ab^*$$
$$\gamma^1_{2,2} = \gamma^0_{2,2} \,|\, \gamma^0_{2,1}(\gamma^0_{1,1})^* \gamma^0_{1,2} = \varepsilon \,|\, b \,|\, a(\varepsilon|b)^* a = \varepsilon \,|\, b \,|\, ab^* a$$

$$\gamma^2_{1,1} = \gamma^1_{1,1} \,|\, \gamma^1_{1,2}(\gamma^1_{2,2})^* \gamma^1_{2,1} = b^* \,|\, b^* a(\varepsilon|b|ab^* a)^* ab^*$$

Reguläre Ausdrücke

Wozu sind reguläre Ausdrücke in der Praxis nützlich?

- Suchen und Ersetzen in Editoren
  (Ausprobieren mit `vi`, `emacs`, ...)
- Pattern-Matching und Verarbeitung großer Texte und Datenmengen, z.B., beim Data-Mining
  (Tools: Stream-Editor `sed`, `awk`, ...)
- Übersetzung von Programmiersprachen:
  Lexikalische Analyse – Umwandlung einer Folge von Zeichen (das Programm) in eine Folge von Tokens, in der bereits die Schlüsselwörter, Bezeichner, Daten, etc. identifiziert sind.
  (Tools: `lex`, `flex`, ...)

Abschlusseigenschaften

Definition (Abgeschlossenheit)

Gegeben sei eine Menge M und ein binärer Operator $\otimes$: $M \times M \to M$. Man sagt, eine Menge $M' \subseteq M$ ist unter $\otimes$ abgeschlossen, wenn für zwei beliebige Elemente $m_1, m_2 \in M'$ gilt: $m_1 \otimes m_2 \in M'$.

Wir betrachten hier Abschlusseigenschaften für die Menge aller regulären Sprachen (d.h. wir setzen $M =$ Menge aller Sprachen und $M' =$ Menge aller regulären Sprachen)

Die interessante Frage ist:

> Falls L_1, L_2 regulär sind, sind dann auch $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 L_2$, $\overline{L_1} = \Sigma^* \backslash L_1$ (Komplement) und L_1^* regulär?

Kurze Antwort: Die regulären Sprachen sind unter allen diesen Operationen abgeschlossen.

Abschlusseigenschaften

Warum sind Abschlusseigenschaften interessant?

Sie sind vor allem dann interessant, wenn sie konstruktiv verwirklicht werden können, das heißt, wenn man – gegeben Automaten für L_1 und L_2 – auch einen Automaten beispielsweise für den Schnitt von L_1 und L_2 konstruieren kann.

Damit hat man dann mit Automaten eine Datenstruktur für unendliche Sprachen, die man maschinell weiterverarbeiten kann.

Satz (Abschluss unter Vereinigung)

Wenn L_1 und L_2 reguläre Sprachen sind, dann ist auch $L_1 \cup L_2$ regulär.

Beweis:

Den Automaten für $L_1 \cup L_2$ kann man mit der selben Methode bauen wie den Automaten für $L(\alpha|\beta)$ bei der Umwandlung von regulären Ausdrücken in NFAs. □

Abschlusseigenschaften

Satz (Abschluss unter Komplement)

Wenn $L \subseteq \Sigma^*$ eine reguläre Sprache ist, dann ist auch $\overline{L} = \Sigma^* \backslash L$ regulär.

Bemerkung: bei Bildung des Komplements muss immer festgelegt werden, bezüglich welcher Obermenge das Komplement gebildet werden soll. Hier ist das die Menge Σ^* aller Wörter über dem Alphabet Σ, das gerade betrachtet wird.

Beweis:

Aus einem DFA $M = (Z, \Sigma, \delta, z_0, E)$ für L gewinnt man leicht einen DFA M' für $\overline{L}$ indem man die End- und Nicht-Endzustände vertauscht. D.h. $M' = (Z, \Sigma, \delta, z_0, Z \backslash E)$.

Dann gilt:
$w \in \overline{L} \iff w \notin T(M) \iff \widehat{\delta}(z_0, w) \notin E \iff \widehat{\delta}(z_0, w) \in Z \backslash E \iff w \in T(M')$. □

Satz (Abschluss unter Produkt/Konkatenation)

Wenn L_1 und L_2 reguläre Sprachen sind, dann ist auch $L_1 L_2$ regulär.

Beweis:

Den Automaten für $L_1 L_2$ kann man mit der selben Methode bauen wie den Automaten für $L(\alpha\beta)$ bei der Umwandlung von regulären Ausdrücken in NFAs. □

Satz (Abschluss unter der Stern-Operation)

Wenn L eine reguläre Sprache ist, dann ist auch L^* regulär.

Beweis:

Den Automaten für L^* kann man mit der selben Methode bauen wie den Automaten für $L((\alpha)^*)$ bei der Umwandlung von regulären Ausdrücken in NFAs. □

Satz (Abschluss unter Schnitt)

Wenn L_1 und L_2 reguläre Sprachen sind, dann ist auch $L_1 \cap L_2$ regulär.

Beweis 1:

Es gilt $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ und wir wissen bereits, dass reguläre Sprachen unter Komplement und Vereinigung abgeschlossen sind. □

Beweis 2:

Es gibt noch eine andere direktere Konstruktion. Dabei werden die zwei Automaten für L_1 und L_2 miteinander synchronisiert und quasi "parallelgeschaltet". Dies erfolgt durch das Bilden des Kreuzprodukts.

Seien $M_1 = (Z_1, \Sigma, \delta_1, S_1, E_1)$, $M_2 = (Z_2, \Sigma, \delta_2, S_2, E_2)$ NFAs mit $T(M_1) = L_1$ und $T(M_2) = L_2$. Dann akzeptiert der folgende NFA M die Sprache $L_1 \cap L_2$:

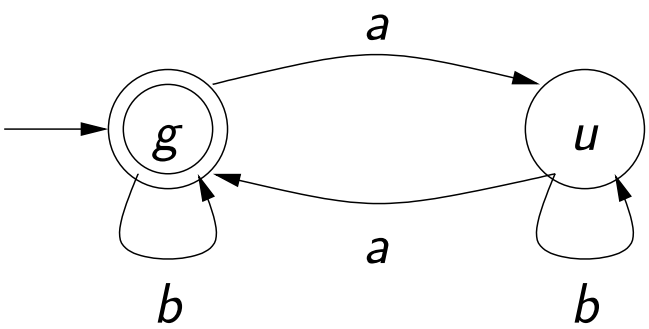
$$M = (Z_1 \times Z_2, \Sigma, \delta, S_1 \times S_2, E_1 \times E_2),$$

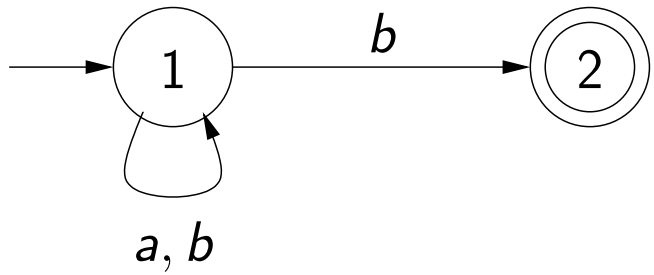
wobei $\delta((z_1, z_2), a) = \{(z_1', z_2') \mid z_1' \in \delta_1(z_1, a), z_2' \in \delta_2(z_2, a))\}$.

M akzeptiert ein Wort w genau dann, wenn sowohl M_1 als auch M_2 das Wort w akzeptieren. □

Beispiel für ein Kreuzprodukt:

Bilde das Kreuzprodukt der folgenden zwei Automaten:





Ausblick

Weitere wichtige Fragen

- Wie kann man zeigen, dass eine Sprache nicht regulär ist?

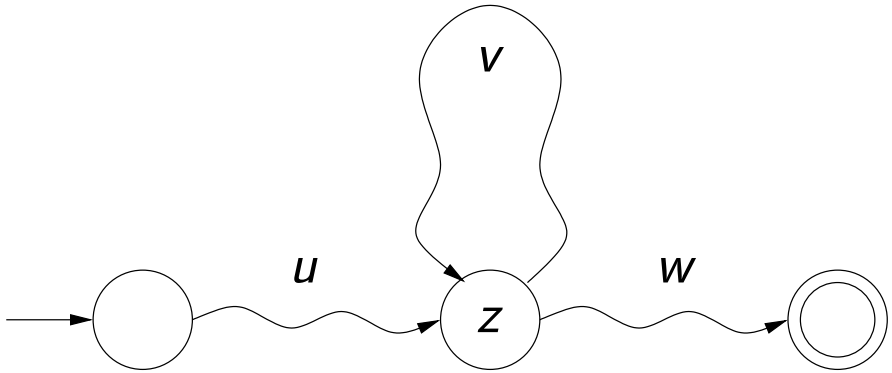
  Beispiel: Die Sprache $\{a^n b^n c^n \mid n \geq 1\}$, die als Beispiel auftauchte, scheint nicht regulär zu sein. Wie kann man das zeigen?
- Wenn eine Sprache regulär ist, wie groß ist dann der kleinste Automat, der die Sprache akzeptiert? Gibt es überhaupt den kleinsten Automaten?

Das Pumping-Lemma

Wie beweist man, dass eine Sprache L nicht regulär ist?

Idee: Man versucht auszunutzen, dass eine reguläre Sprache von einem Automat mit endlich vielen Zuständen akzeptiert werden muss. Das bedeutet auch: wenn ein Wort $x \in L$ ausreichend lang ist, so besucht man damit beim Durchlauf durch den Automaten mindestens einen Zustand z zweimal.

Das Pumping-Lemma



Die dadurch entstehende Schleife kann nun mehrfach (oder gar nicht) durchlaufen werden, dadurch wird das Wort $x = uvw$ "aufgepumpt" und man stellt fest, dass uw, uv^2w, uv^3w, ... auch in L liegen müssen.

Bemerkung: Es gilt $v^i = \underbrace{v \dots v}_{i\text{-mal}}$.

Das Pumping-Lemma

Außerdem kann man für u, v, w folgende Eigenschaften verlangen, wobei n die Anzahl der Zustände des Automaten ist.

1. $|v| \geq 1$: Die Schleife ist auf jeden Fall nicht trivial, d.h. sie enthält mindestens einen Übergang.
2. $|uv| \leq n =$ Anzahl der Zustände des NFA: Spätestens nach n Alphabetsymbolen wird der Zustand z das zweite Mal erreicht.

Das Pumping-Lemma

Satz (Pumping-Lemma, *uvw*-Theorem)

Sei L eine reguläre Sprache. Dann gibt es eine Zahl n, so dass sich alle Wörter $x \in L$ mit $|x| \geq n$ zerlegen lassen in $x = uvw$, so das folgende Eigenschaften erfüllt sind:

1. $|v| \geq 1$,
2. $|uv| \leq n$ und
3. für alle $i \geq 0$ gilt $uv^i w \in L$.

Dabei ist n die Anzahl der Zustände eines Automaten, der L erkennt. Dieses Lemma spricht jedoch nicht über Automaten, sondern nur über die Eigenschaften der Sprache. Daher ist es dazu geeignet, Aussagen über Nicht-Regularität zu machen.

Das Pumping-Lemma

Beweis des Pumping-Lemmas:

Sei L eine reguläre Sprache.

Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA mit $L = T(M)$, sei $n = |Z|$.

Sei nun x ein beliebiges Wort mit $x \in L = T(M)$ und $|x| \geq n$, d.h. $x = a_1 a_2 \cdots a_m$ mit $m \geq n$ und $a_1, a_2, \ldots, a_m \in \Sigma$.

Da $x \in T(M)$, existieren Zustände $z_0, z_1, \ldots, z_m \in Z$ mit

$$z_0 \in S, \quad z_j \in \delta(z_{j-1}, a_j) \text{ für } 1 \leq j \leq m, \quad z_m \in E.$$

Wegen $|Z| = n$ existieren $0 \leq j < k \leq n$ mit $z_j = z_k$ (Schubfachprinzip).

Sei $u = a_1 \cdots a_j$, $v = a_{j+1} \cdots a_k$ und $w = a_{k+1} \cdots a_m$.

Dann gilt:

- $|v| = k - (j+1) + 1 = k - j > 0$ und $|uv| = k \leq n$
- für alle $i \geq 0$: $z_m \in \widehat{\delta}(\{z_0\}, uv^i w)$ und damit $uv^i w \in T(M) = L$,

Das Pumping-Lemma

Wie kann man das Pumping-Lemma nutzen, um zu zeigen, dass eine Sprache nicht regulär ist?

Aussage des Pumping-Lemmas mit logischen Operatoren:

L regulär $\rightarrow$
$\exists n \;\; \forall x \in L$ mit $|x| \geq n \;\; \exists u, v, w$ mit $|v| \geq 1, |uv| \leq n, x = uvw$ und
$\forall i : uv^i w \in L$

Das ist logisch äquivalent zu

$\forall n \quad \exists x \in L$ mit $|x| \geq n, \quad \forall u, v, w$ mit $|v| \geq 1, |uv| \leq n$ und $x = uvw$
$\exists i : uv^i w \notin L$
$\rightarrow L$ ist nicht regulär

$$A \rightarrow B \equiv \neg B \rightarrow \neg A \text{ und } \neg \forall x \exists y F \equiv \exists x \forall y \neg F.$$

Pumping-Lemma

"Kochrezept" für das Pumping-Lemma

Gegeben sei eine Sprache L.

Beispiel: $\{a^k b^k \mid k \geq 0\}$

Wir wollen zeigen, dass sie nicht regulär ist.

1. Nehme eine beliebige Zahl n an. Diese Zahl darf nicht frei gewählt werden.
2. Wähle ein geeignetes Wort $x \in L$ mit $|x| \geq n$. Damit das Wort auch wirklich mindestens die Länge n hat, empfiehlt es sich, dass n (beispielsweise als Exponent) im Wort auftaucht.

   Beispiel: $x = a^n b^n$

Pumping-Lemma

"Kochrezept" für das Pumping-Lemma

3. Betrachte nun alle möglichen Zerlegungen $x = uvw$ mit den Einschränkungen $|v| \geq 1$ und $|uv| \leq n$.

   Beispiel: Aus $|v| \geq 1$ und $|uv| \leq n$ folgt, dass $j \geq 0$ und $\ell \geq 1$ existieren mit:
   $u = a^j$, $v = a^\ell$ und $w = a^m b^n$ mit $j + \ell + m = n$

4. Wähle für jede dieser Zerlegungen ein i (das kann jedes Mal ein anderes i sein), so dass $uv^i w \notin L$.
   In vielen Fällen sind $i = 0$ und $i = 2$ eine gute Wahl.

   Beispiel: wähle $i = 2$, dann gilt $uv^2w = a^{j+2\ell+m} b^n \notin L$, da $j + 2\ell + m = n + \ell \neq n$ wegen $\ell \geq 1$.

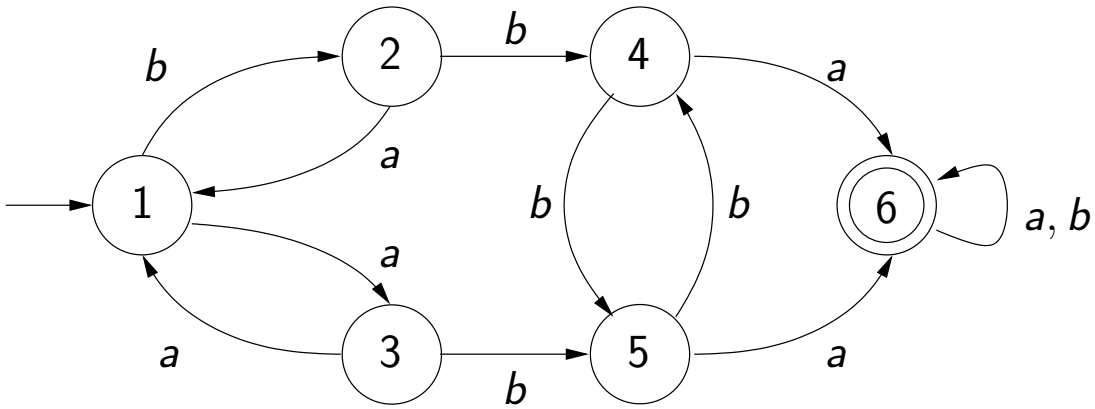
Äquivalenzrelationen und Minimalautomat

Wir beschäftigen uns nun mit folgenden Fragen:

- Gibt es zu jeder Sprache immer den kleinsten deterministischen/nicht-deterministischen Automat?
- Kann man direkt aus der Sprache die Anzahl der Zustände des minimalen Automaten ablesen?
- Wie bestimmt man den minimalen Automat?

Äquivalenzrelationen und Minimalautomat

Betrachte den folgenden DFA M:



Feststellung: für die Zustände 4, 5 gilt

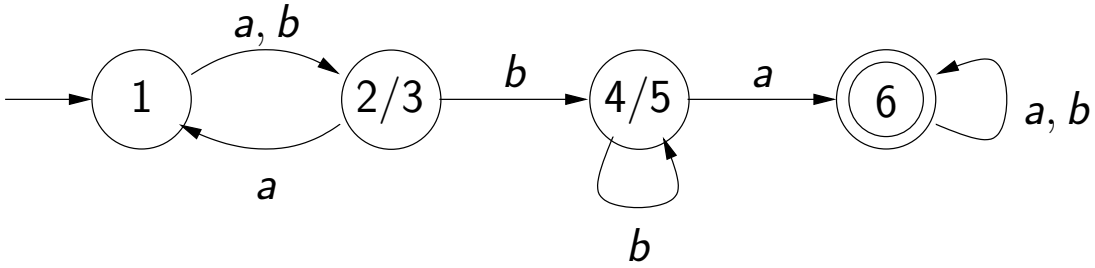
- mit einem Wort, das ein a enthält, landet man von dort aus immer im Zustand 6 (Endzustand)
- mit einem Wort, das kein a enthält, landet man von dort aus immer im Zustand 4 bzw. 5 (kein Endzustand)

Daraus folgt: 4 und 5 sind erkennungsäquivalent und können zu einem Zustand verschmolzen werden.

Äquivalenzrelationen und Minimalautomat

Ebenso: die Zustände 2 und 3 sind erkennungsäquivalent

Entstehender Automat M':



Jetzt sind keine Zustände mehr erkennungsäquivalent und sie können daher nicht weiter verschmolzen werden.

$\rightsquigarrow$ Der Automat M' ist minimal für diese Sprache.

Äquivalenzrelationen und Minimalautomat

Definition (Erkennungsäquivalenz)

Gegeben sei ein DFA $M = (Z, \Sigma, \delta, q_0, E)$.

Zwei Zustände $z_1, z_2 \in Z$ heißen erkennungsäquivalent genau dann, wenn für jedes Wort $w \in \Sigma^*$ gilt:

$$\widehat{\delta}(z_1, w) \in E \iff \widehat{\delta}(z_2, w) \in E.$$

Äquivalenzrelationen und Minimalautomat

Jedem Wort $x \in \Sigma^*$ kann man in einem deterministischen Automaten einen eindeutigen Zustand $z = \widehat{\delta}(z_0, x)$ zuordnen. Daher kann die Definition der Erkennungsäquivalenz auf Wörter aus Σ^* und Sprachen (anstatt Automaten) ausgedehnt werden.

Definition (Myhill-Nerode-Äquivalenz)

Gegeben sei eine Sprache L und Wörter $x, y \in \Sigma^*$.
Wir definieren eine Äquivalenzrelation R_L mit $x \, R_L \, y$ genau dann wenn

$$\forall w \in \Sigma^*(xw \in L \iff yw \in L).$$

Das ist gleichbedeutend damit, dass $\widehat{\delta}(z_0, x)$ und $\widehat{\delta}(z_0, y)$ erkennungsäquivalent sind, und zwar für einen beliebigen DFA M, der L akzeptiert.

Äquivalenzrelationen und Minimalautomat

Beispiel 1 für Myhill-Nerode-Äquivalenz: Gegeben sei die Sprache

$$L = \{w \in \{a, b\}^* \mid \#_a(w) \text{ gerade}\}.$$

Es gibt folgende Äquivalenzklassen für R_L:

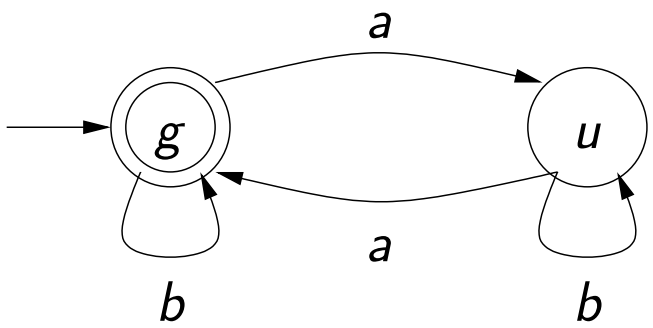
- $[\varepsilon] = \{w \in \{a, b\}^* \mid \#_a(w) \text{ gerade}\} = L$
  (Äquivalenzklasse von ε)
- $[a] = \{w \in \{a, b\}^* \mid \#_a(w) \text{ ungerade}\} = \{a, b\}^* \setminus L$
  (Äquivalenzklasse von a)

Die Wörter ε und aa sind äquivalent, denn:

- Wird an beide ein Wort mit gerade vielen a's angehängt, so bleiben sie beide in der Sprache.
- Wird an beide ein Wort mit ungerade vielen a's angehängt, so fallen sie beide aus der Sprache heraus.

Äquivalenzrelationen und Minimalautomat

DFA für $\{w \in \{a, b\}^* \mid \#_a(w) \text{ gerade}\}$:



Äquivalenzrelationen und Minimalautomat

Beispiel 2 für Myhill-Nerode-Äquivalenz: Gegeben sei die Sprache

$$L = \{w \in \{a, b, c\}^* \mid \text{ das Teilwort } abc \text{ kommt in } w \text{ nicht vor}\}.$$

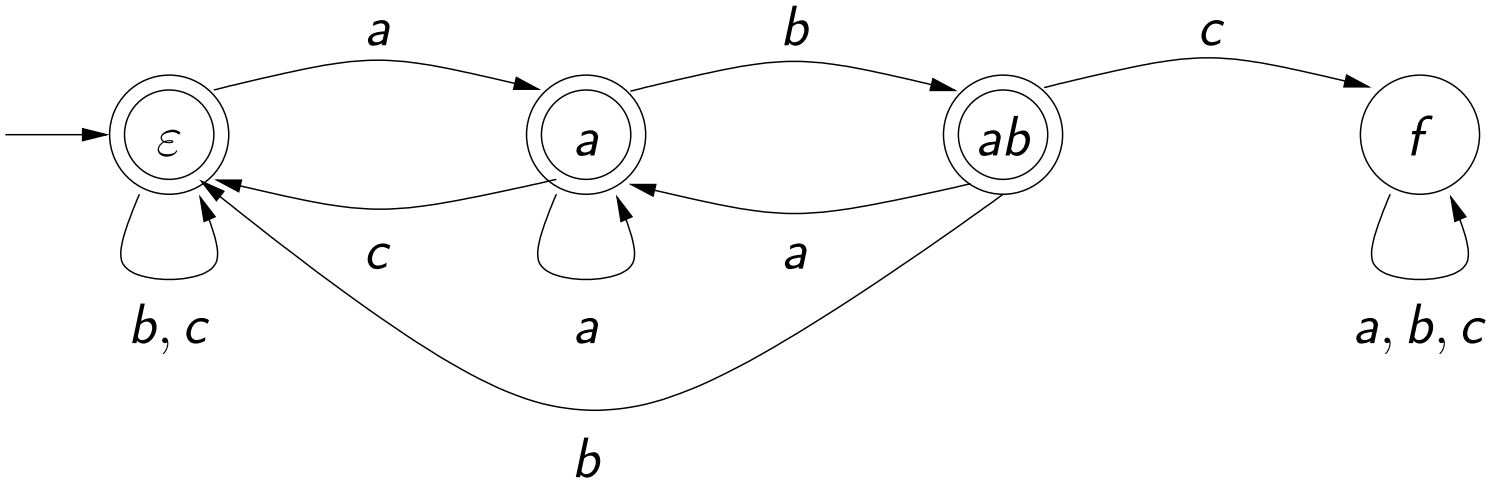
Es gibt folgende Äquivalenzklassen für R_L:

- $[\varepsilon] = \{w \in \{a, b, c\}^* \mid w$ endet nicht auf a oder ab und enthält abc nicht $\}$
- $[a] = \{w \in \{a, b, c\}^* \mid w$ endet auf a und enthält abc nicht $\}$
- $[ab] = \{w \in \{a, b, c\}^* \mid w$ endet auf ab und enthält abc nicht $\}$
- $[abc] = \{w \in \{a, b, c\}^* \mid \ w$ enthält $abc\}$ (Fangzustand)

Die Wörter a und ab sind nicht äquivalent, denn wird an beide ein c angehängt, so ist ac noch in L, abc ist es aber nicht.

Äquivalenzrelationen und Minimalautomat

DFA für $\{w \in \{a, b, c\}^* \mid$ das Teilwort abc kommt in w nicht vor$\}$:



Äquivalenzrelationen und Minimalautomat

Sei $R \subseteq A \times A$ eine Äquivalenzrelation.

Für $x \in A$ ist $[x] = \{y \in A \mid x\, R\, y\}$ die Äquivalenzklasse von x.

Der Index $\text{index}(R)$ von R ist die Anzahl der Äquivalenzklassen von R:

$$\text{index}(R) = |\{[x] \mid x \in A\}| \in \mathbb{N} \cup \{\infty\}$$

Satz von Myhill-Nerode

Sei L eine Sprache. L ist regulär $\iff \text{index}(R_L) < \infty$

Beweis:

$\Longrightarrow$: Sei L regulär.

Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA mit $T(M) = L$.

Definiere eine Äquivalenzrelation R_M auf Σ^* wie folgt:

$$x\, R_M\, y \quad \iff \quad \widehat{\delta}(z_0, x) = \widehat{\delta}(z_0, y).$$

Äquivalenzrelationen und Minimalautomat

Beachte:

- R_M ist in der Tat eine Äquivalenzrelation.
- $\text{index}(R_M) \leq |Z|$

Behauptung: $\forall x, y \in \Sigma^*(x \, R_M \, y \implies x \, R_L \, y)$, d.h. $R_M \subseteq R_L$.

Beweis der Behauptung:

$$
\begin{aligned}
x \, R_M \, y \quad &\iff \quad \widehat{\delta}(z_0, x) = \widehat{\delta}(z_0, y) \\
&\iff \quad \forall w \in \Sigma^* : \widehat{\delta}(z_0, xw) = \widehat{\delta}(z_0, yw) \\
&\implies \quad \forall w \in \Sigma^* : xw \in T(M) = L \leftrightarrow yw \in T(M) = L \\
&\iff \quad x \, R_L \, y
\end{aligned}
$$

Aus der Behauptung folgt $\text{index}(R_L) \leq \text{index}(R_M) \leq |Z| < \infty$.

Äquivalenzrelationen und Minimalautomat

$\Longleftarrow$: Sei $\text{index}(R_L) < \infty$.

Sei $[x_1], \ldots, [x_n]$ eine Auflistung aller Äquivalenzklassen von R_L.

Beachte:

- $\Sigma^* = [x_1] \cup \cdots \cup [x_n]$.
- Wenn $[x] = [y]$, dann $[xa] = [ya]$ für alle $a \in \Sigma$

Wir definieren nun den DFA

$$M_L = (\{[x_1], \ldots, [x_n]\}, \Sigma, \delta, [\varepsilon], \{[w] \mid w \in L\}),$$

wobei $\delta([x_i], a) = [x_i a]$ für alle $1 \leq i \leq n$ und $a \in \Sigma$.

Offensichtlich gilt für alle $x \in \Sigma^*$: $\widehat{\delta}([\varepsilon], x) = [x]$.

Behauptung: $T(M_L) = L$ (dies zeigt dann, dass L regulär ist).

Beweis der Behauptung:

$$
\begin{aligned}
x \in T(M_L) &\iff \widehat{\delta}([\varepsilon], x) \in \{[w] \mid w \in L\} \\
&\iff [x] \in \{[w] \mid w \in L\} \\
&\iff \exists w \in L : [x] = [w] \\
&\iff \exists w \in L : x \, R_L \, w \\
&\iff x \in L
\end{aligned}
$$

□

Äquivalenzrelationen und Minimalautomat

Mit dem Satz von Myhill-Nerode kann man auch zeigen, dass eine Sprache L nicht regulär ist.

Dazu muss man nur unendlich viele Wörter aus Σ^* finden, die in verschiedenen R_L-Äquivalenzklassen liegen.

Beispiel 3 für Myhill-Nerode-Äquivalenz:

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

Betrachte die Wörter a, aa, aaa, $\ldots$, a^i, $\ldots$

Es gilt: $\neg(a^i \, R_L \, a^j)$ für $i \neq j$, denn $a^i b^i \in L$ und $a^j b^i \notin L$.

Also hat R_L unendlich viele Äquivalenzklassen und L ist nicht regulär.

Äquivalenzrelationen und Minimalautomat

Betrachten wir nochmals den DFA M_L, der im Beweis des Satzes von Myhill-Nerode (Richtung "$\Longleftarrow$") konstruiert wurde.

Satz

Sei L regulär und sei M ein beliebiger DFA mit $T(M) = L$.

1. M hat mindestens soviele Zustände, wie M_L, d.h. M_L ist ein minimaler DFA für L.
2. Falls Anzahl der Zustände von M = Anzahl der Zustände von M_L, so sind M und M_L bis auf Umbennenung der Zustände identisch.

Beweis:

Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA mit $T(M) = L$.

Im Beweis von "$\Longrightarrow$" haben wir gesehen: $\text{index}(R_L) \leq |Z|$.

Äquivalenzrelationen und Minimalautomat

Ausserdem ist die Anzahl der Zustände von $M_L = \text{index}(R_L)$.

Dies zeigt (1).

Sei nun $|Z| =$ Anzahl der Zustände von $M_L = \text{index}(R_L)$.

Wegen $|Z| = \text{index}(R_L) \leq \text{index}(R_M) \leq |Z|$ gilt
$\text{index}(R_L) = \text{index}(R_M) < \infty$.

Mit $R_M \subseteq R_L$ folgt $R_M = R_L = R_{M_L}$.

Also sind M und M_L bis auf Umbennenung der Zustände identisch. □

Bemerkung: Es gibt also für eine reguläre Sprache bis auf Umbenennung von Zuständen nur einen minimalen DFA.

Äquivalenzrelationen und Minimalautomat

Problemstellung: Konstruiere den minimalen Automaten M_L aus einem nicht unbedingt minimalen DFA $M = (Z, \Sigma, \delta, z_0, E)$ mit $T(M) = L$.

Zunächst können wir voraussetzen, dass jeder Zustand $z \in Z$ vom Anfangszustand z_0 erreicht werden kann, d.h. $\exists x \in \Sigma^* : \widehat{\delta}(z_0, x) = z$.

Beachte: Dies impliziert $|Z| = \text{index}(R_M)$.

Nun gilt:

$$
\begin{aligned}
M \text{ nicht minimal} &\iff R_M \subsetneq R_L \\
&\iff \exists x, y \in \Sigma^* : (x, y) \in R_L \ \wedge \ (x, y) \notin R_M \\
&\iff \exists x, y \in \Sigma^* : \forall w \in \Sigma^* : xw \in L \leftrightarrow yw \in L \\
&\qquad\qquad \wedge \ \widehat{\delta}(z_0, x) \neq \widehat{\delta}(z_0, y) \\
&\iff \exists x, y \in \Sigma^* : \forall w \in \Sigma^* : xw \in T(M) \leftrightarrow yw \in T(M) \\
&\qquad\qquad \wedge \ \widehat{\delta}(z_0, x) \neq \widehat{\delta}(z_0, y) \\
&\iff \exists z_1, z_2 \in Z : z_1, z_2 \text{ sind erkennungsäquivalent,} \\
&\qquad\qquad z_1 \neq z_2
\end{aligned}
$$

Äquivalenzrelationen und Minimalautomat

Lösung: wir verschmelzen in M alle erkennungsäquivalenten Zustände.

Um herauszufinden, welche Zustände erkennungsäquivalent sind, markieren wir alle Zustandspaare $\{z, z'\}$, die nicht erkennungsäquivalent sind.

Zunächst sind sicherlich alle Paare $\{z, z'\}$ mit $z \in E$ und $z' \notin E$ nicht erkennungsäquivalent, diese Paare markieren wir zu Beginn.

Angenommen für ein Paar $\{z, z'\}$ existiert ein $a \in \Sigma$, so dass $\{\delta(z, a), \delta(z', a)\}$ nicht erkennungsäquivalent sind.
Dann sind auch $\{z, z'\}$ nicht erkennungsäquivalent.

Diese Beobachtung erlaubt uns, weitere Paare als nicht erkennungsäquivalent zu markieren.

Äquivalenzrelationen und Minimalautomat

Algorithmus Minimalautomat

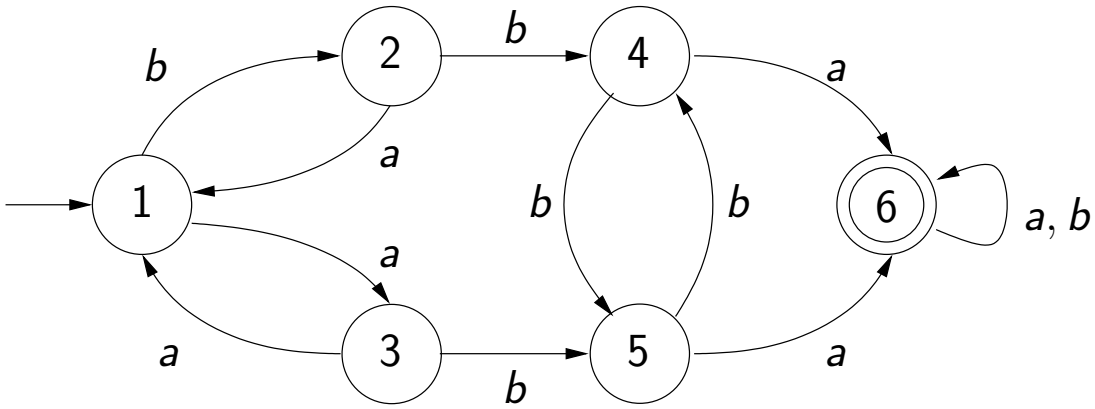
Eingabe: DFA M (Zustände, die vom Startzustand aus nicht erreichbar sind, sind bereits entfernt.)

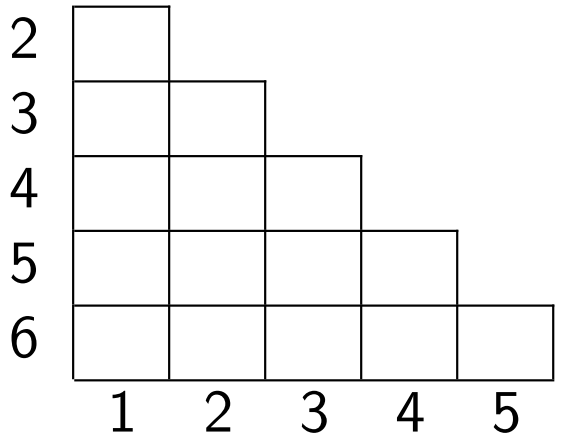
Ausgabe: Mengen von erkennungsäquivalenten Zuständen

1. Stelle eine Tabelle aller Zustandspaare $\{z, z'\}$ mit $z \neq z'$ auf.
2. Markiere alle Paare $\{z, z'\}$ mit $z \in E$ und $z' \notin E$.
3. Für jedes noch unmarkierte Paar $\{z, z'\}$ und jedes $a \in \Sigma$ teste, ob $\{\delta(z, a), \delta(z', a)\}$ bereits markiert ist. Wenn ja: markiere auch $\{z, z'\}$.
4. Wiederhole den vorherigen Schritt, bis sich keine Änderung in der Tabelle mehr ergibt.
5. Für alle jetzt noch unmarkierten Paare $\{z, z'\}$ gilt: z und z' sind erkennungsäquivalent.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

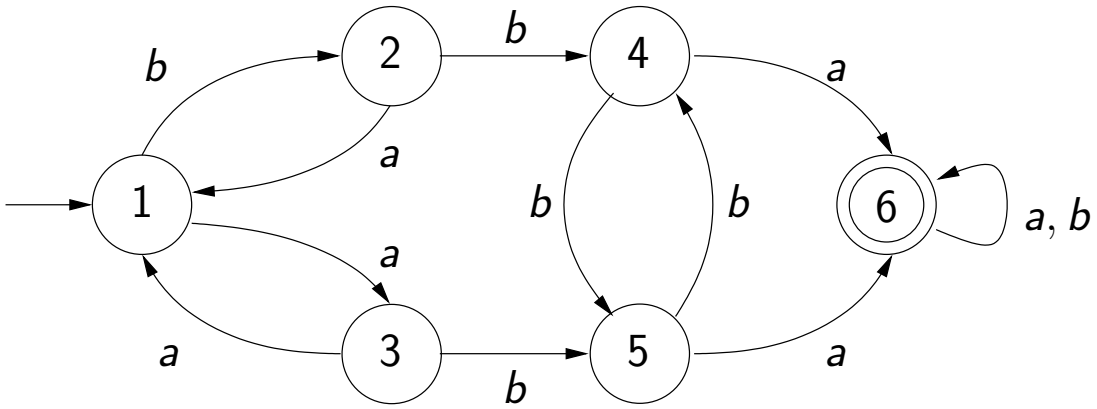




Erstelle eine Tabelle aller Zustandspaare.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

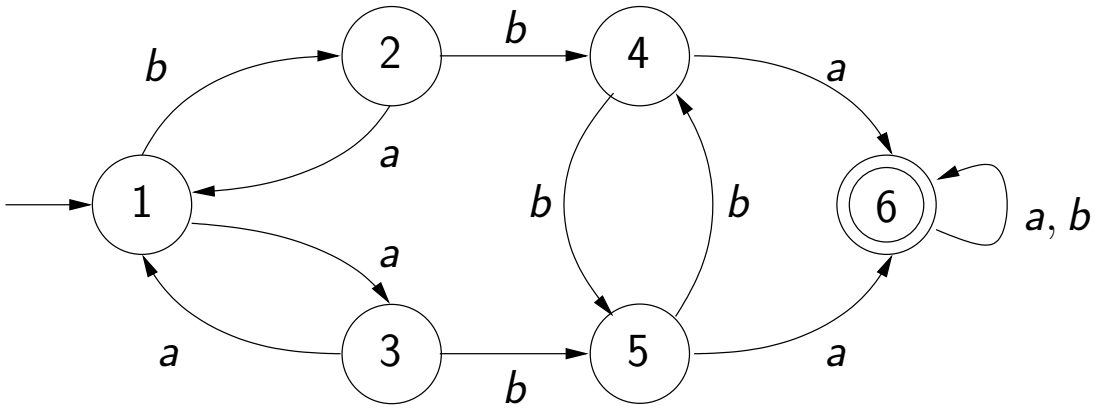


2					
3					
4					
5					
6	1	1	1	1	1
	1	2	3	4	5

(1) Markiere Paare von Endzuständen und Nicht-Endzuständen.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

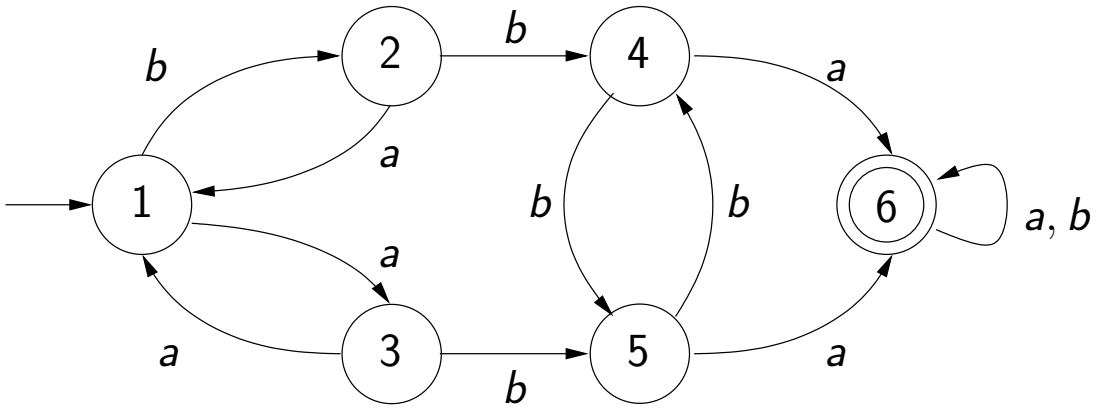


2					
3					
4		2			
5					
6	1	1	1	1	1
	1	2	3	4	5

(2) Markiere $\{2, 4\}$ wegen $\delta(2, a) = 1$, $\delta(4, a) = 6$ und $\{1, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

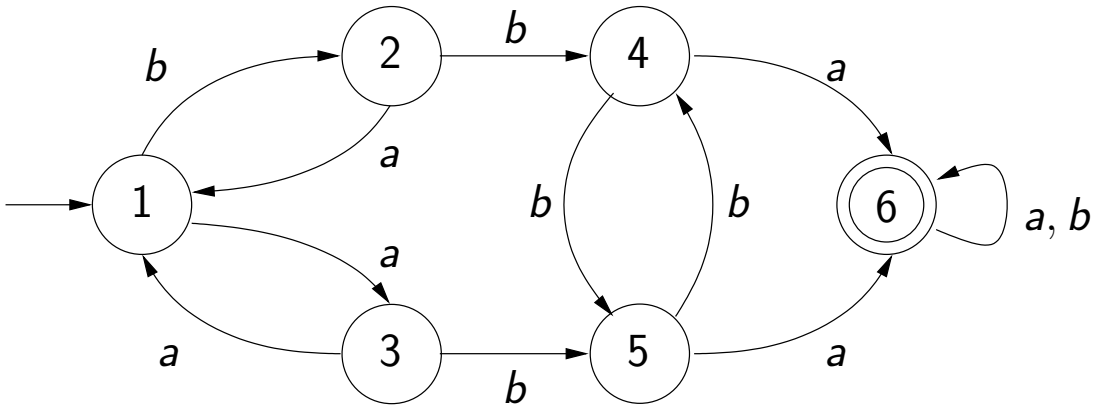


2					
3					
4		2			
5			3		
6	1	1	1	1	1
	1	2	3	4	5

(3) Markiere $\{3, 5\}$ wegen $\delta(3, a) = 1$, $\delta(5, a) = 6$ und $\{1, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

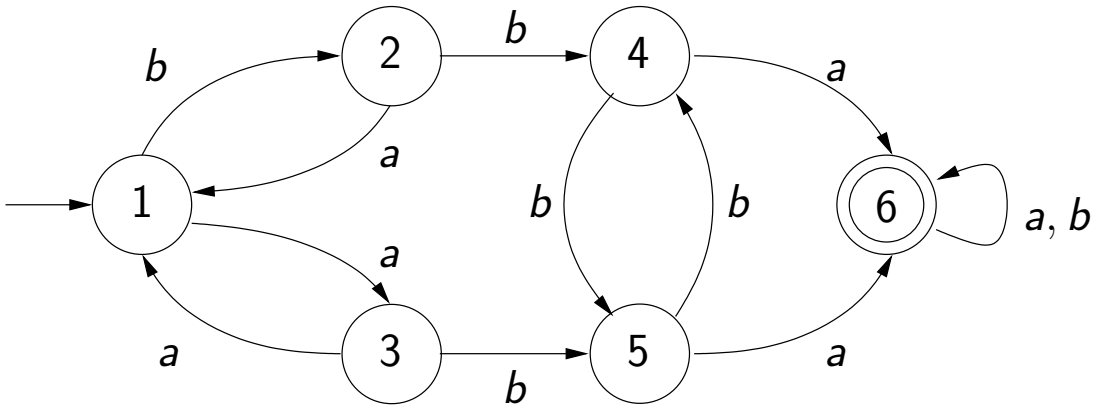


	1	2	3	4	5
2					
3					
4		2			
5		4	3		
6	1	1	1	1	1

(4) Markiere $\{2, 5\}$ wegen $\delta(2, a) = 1$, $\delta(5, a) = 6$ und $\{1, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

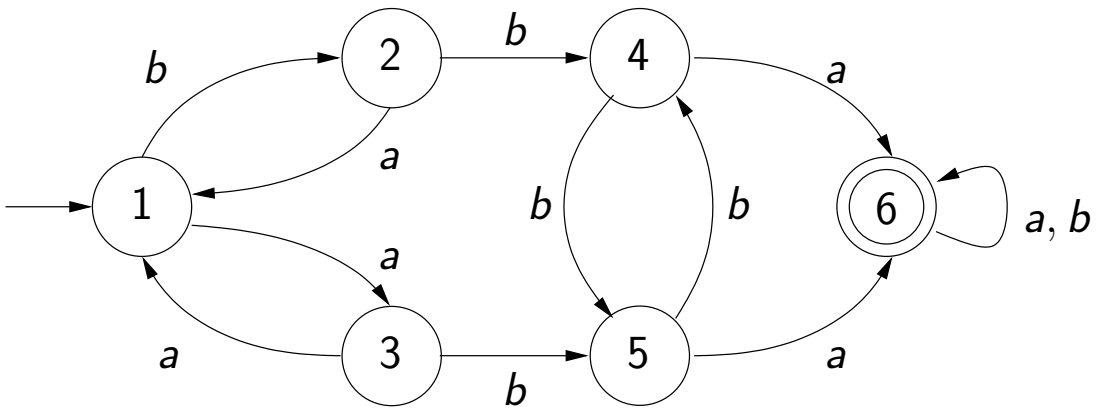


	1	2	3	4	5
2					
3					
4		2	5		
5		4	3		
6	1	1	1	1	1

(5) Markiere $\{3, 4\}$ wegen $\delta(3, a) = 1$, $\delta(4, a) = 6$ und $\{1, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

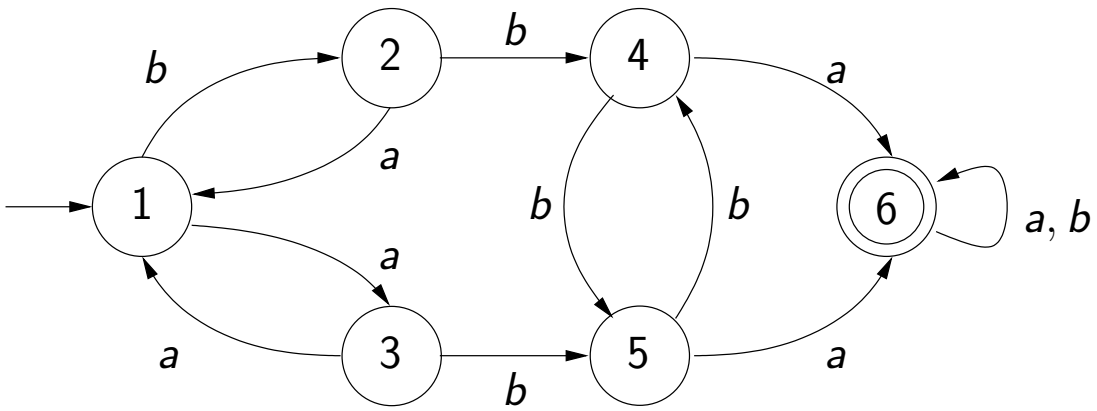


	1	2	3	4	5
2					
3					
4		2	5		
5	6	4	3		
6	1	1	1	1	1

(6) Markiere $\{1, 5\}$ wegen $\delta(1, a) = 3$, $\delta(5, a) = 6$ und $\{3, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

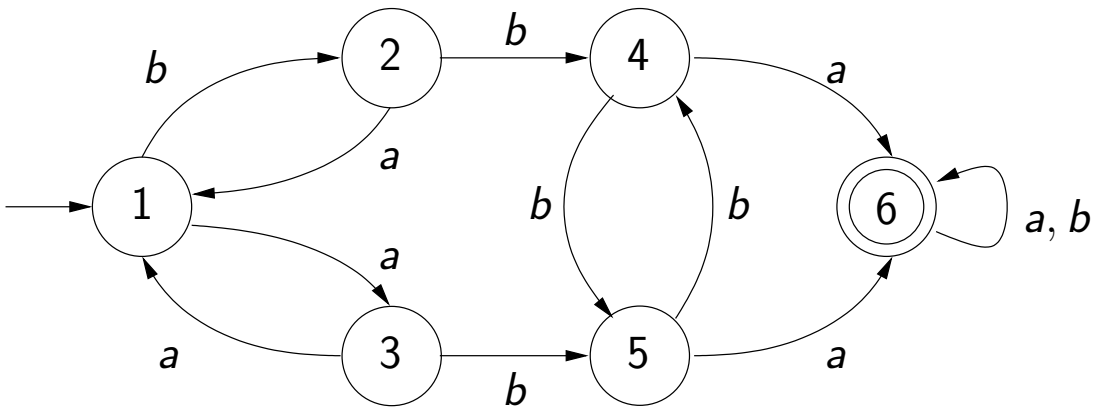


2					
3					
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

(7) Markiere $\{1, 4\}$ wegen $\delta(1, a) = 3$, $\delta(4, a) = 6$ und $\{3, 6\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

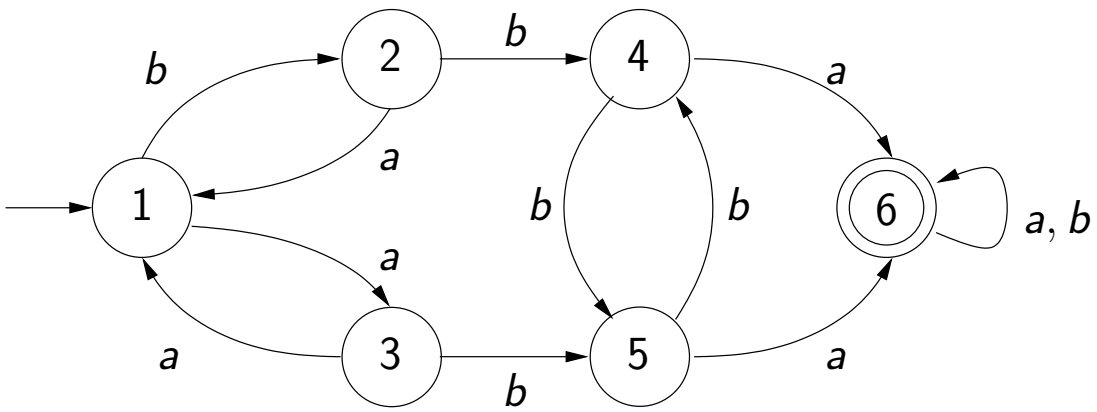


2					
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

(8) Markiere $\{1, 3\}$ wegen $\delta(1, b) = 2$, $\delta(3, b) = 5$ und $\{2, 5\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:

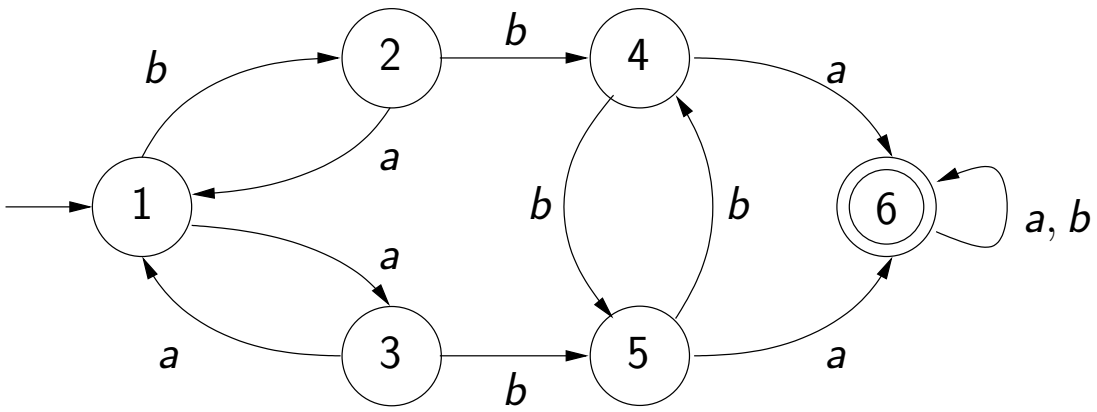


2	9				
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

(9) Markiere $\{1, 2\}$ wegen $\delta(1, b) = 2$, $\delta(2, b) = 4$ und $\{2, 4\}$ markiert.

Äquivalenzrelationen und Minimalautomat

Beispiel für Durchführung des Minimierungsalgorithmus:



2	9				
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

Die verbleibenden Zustandspaare $\{2,3\}$ und $\{4,5\}$ können nicht mehr markiert werden. $\rightsquigarrow$ Sie sind erkennungsäquivalent.

Äquivalenzrelationen und Minimalautomat

Satz (Korrektheit des Minimierungsalgorithmus)

Für einen gegebenen DFA $M = (Z, \Sigma, \delta, z_0, E)$ markiert der Minimierungsalgorithmus ein Paar $\{z, z'\}$ $(z, z' \in Z,\ z \neq z')$ genau dann, wenn z und z' nicht erkennungsäquivalent sind.

Beweis:

(A) Falls $\{z, z'\}$ markiert wird, so sind z und z' nicht erkennungsäquivalent.

Beweis durch Induktion über den Zeitpunkt, zu dem $\{z, z'\}$ markiert wird.

IA: $\{z, z'\}$ wird zu Beginn markiert, weil $z \in E$ und $z' \notin E$.

$\rightsquigarrow$ z und z' sind nicht erkennungsäquivalent.

IS: $\{z, z'\}$ wird irgendwann markiert, weil ein $a \in \Sigma$ existiert, so dass $\{\delta(z, a), \delta(z', a)\}$ zu einem früheren Zeitpunkt markiert wurden.

IH $\rightsquigarrow$ $\delta(z, a)$ und $\delta(z', a)$ sind nicht erkennungsäquivalent.

Äquivalenzrelationen und Minimalautomat

$\rightsquigarrow$ z und z' sind nicht erkennungsäquivalent.

(B) Wenn z und z' nicht erkennungsäquivalent sind, dann wir $\{z, z'\}$ irgendwann markiert.

Seien z und z' nicht erkennungsäquivalent.

Sei $\lambda(z, z')$ die Länge eines kürzesten Wortes w mit $\widehat{\delta}(z, w) \in E$, $\widehat{\delta}(z', w) \notin E$ (oder umgekehrt).

Wir zeigen durch Induktion über $\lambda(z, z')$, dass $\{z, z'\}$ markiert wird.

IA: $\lambda(z, z') = 0$

$\rightsquigarrow$ $z \in E$ und $z' \notin E$

$\rightsquigarrow$ $\{z, z'\}$ wird zu Beginn markiert.

IS: Sei $\lambda(z, z') > 0$.

$\rightsquigarrow$ Es gibt ein Wort au ($a \in \Sigma$ und $u \in \Sigma^*$) mit $|au| = \lambda(z, z')$, so dass

$$\widehat{\delta}(z, au) = \widehat{\delta}(\delta(z, a), u) \in E, \quad \widehat{\delta}(z', au) = \widehat{\delta}(\delta(z', a), u) \notin E$$

(oder umgekehrt).

Dann sind auch $\delta(z, a)$ und $\delta(z', a)$ nicht erkennungsäquivalent und $\lambda(\delta(z, a), \delta(z', a)) \leq |u| < \lambda(z, z')$

IH $\rightsquigarrow$ $\{\delta(z, a), \delta(z', a)\}$ wird irgendwann markiert.

$\rightsquigarrow$ $\{z, z'\}$ wird markiert. □

Äquivalenzrelationen und Minimalautomat

Hinweise für die Durchführung des Minimierungsalgorithmus:

- Die Tabelle möglichst so aufstellen, dass jedes Paar nur genau einmal vorkommt! Also bei Zustandsmenge $\{1, \ldots, n\}$:

  $2, \ldots, n$ vertikal und $1, \ldots, n-1$ horizontal notieren.

- Bitte angeben, welche Zustände in welcher Reihenfolge und warum markiert wurden!

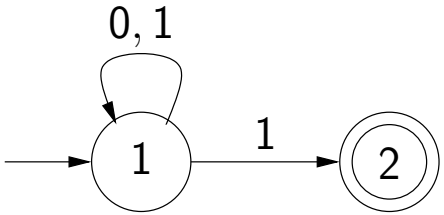
  Im Buch von Schöning werden immer nur Sternchen ($*$) verwendet, aber daraus werden bei der Korrektur die Reihenfolge und die Gründe für die Markierung nicht ersichtlich.

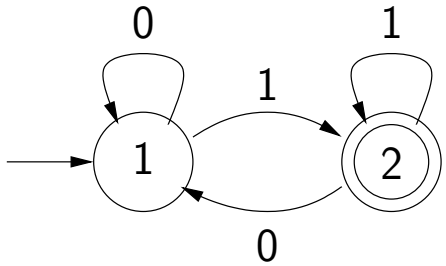
Äquivalenzrelationen und Minimalautomat

Für nicht-deterministische Automaten kann man folgende Aussagen treffen:

- Es gibt nicht *den* minimalen NFA, sondern es kann mehrere geben.

  Folgende zwei minimale NFAs erkennen $L = ((0|1)^*1)$ und haben zwei Zustände (mit nur einem Zustand kann L nicht erkannt werden).





- Gegeben ein DFA M. Dann hat ein minimaler NFA, der $T(M)$ erkennt, immer höchstens so viel Zustände wie M, denn M selbst ist schon ein NFA.

  Außerdem: der minimale NFA kann exponentiell kleiner sein als der minimale DFA.
  Siehe $L_k = \{x \in \{0,1\}^* \mid |x| \geq k, \text{das } k\text{-letzte Zeichen von } x \text{ ist } 0\}$.

Entscheidbarkeit

Wir diskutieren nun, ob es Verfahren gibt, um die folgenden Fragestellungen bzw. Probleme für reguläre Sprachen zu entscheiden. Dabei nehmen wir an, dass reguläre Sprachen als DFAs, NFAs, Grammatiken oder reguläre Ausdrücke gegeben sind.

Probleme

- Wortproblem: Gilt $w \in L$ für eine gegebene reguläre Sprache L und $w \in \Sigma^*$?
- Leerheitsproblem: Gilt $L = \emptyset$ für eine gegebene reguläre Sprache L?
- Endlichkeitsproblem: Ist eine gegebene reguläre Sprache L endlich?
- Schnittproblem: Gilt $L_1 \cap L_2 = \emptyset$ für gegebene reguläre L_1, L_2?
- Inklusionsproblem: Gilt $L_1 \subseteq L_2$ für gegebene reguläre L_1, L_2?
- Äquivalenzproblem: Gilt $L_1 = L_2$ für gegebene reguläre L_1, L_2?

Entscheidbarkeit

Wortproblem:

Sei $L \subseteq \Sigma^*$ eine reguläre Sprache, gegeben durch einen DFA $M = (Z, \Sigma, \delta, z_0, E)$ mit $T(M) = L$, und sei $w \in \Sigma^*$.

Frage: $w \in L$?

Lösung:

Sei $w = a_1 a_2 \cdots a_n$ mit $a_i \in \Sigma$.

Verfolge die Zustandsübergänge von M, die durch die Symbole $a_1, \ldots, a_n$ vorgegeben sind:

$z := z_0$
for $i := 1$ **to** n **do**
  $z := \delta(z, a_i)$
endfor
if $z \in E$ **then return**(JA) **else return**(NEIN)

Leerheitsproblem:

Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA.

Frage: $T(M) \neq \emptyset$?

Lösung:

Sei $G = (Z, \rightarrow)$ der gerichtete Graph mit

$$z \rightarrow z' \quad \Longleftrightarrow \quad \exists a \in \Sigma : z' \in \delta(z, a).$$

Dann gilt: $T(M) \neq \emptyset$, genau dann, wenn es in dem Graphen G einen (evtl. leeren) Pfad von einem Knoten aus S zu einem Knoten aus E gibt.

Dies kann z. B. mit Tiefen- oder Breitensuche entschieden werden.

Entscheidbarkeit

Endlichkeitsproblem:

Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA.

Frage: Ist $T(M)$ endlich?

Lösung:

Sei G wie auf der vorherigen Folie definiert.

Dann gilt: $T(M)$ ist unendlich, genau dann, wenn es $z \in Z$, $z_0 \in S$ und $z_1 \in E$ gibt mit: $z_0 \rightarrow^* z \rightarrow^+ z \rightarrow^* z_1$, d.h. z liegt auf einem Zyklus, ist von einem Startzustand aus erreichbar, und von z kann ein Endzustand erreicht werden.

Dies kann wieder durch Tiefen- oder Breitensuche entschieden werden.

Schnittproblem:

Seien M_1 und M_2 NFAs.

Frage: Gilt $T(M_1) \cap T(M_2) = \emptyset$?

Lösung:

Konstruiere aus M_1 und M_2 den Produktautomaten M ($\rightsquigarrow$ $T(M) = T(M_1) \cap T(M_2)$), siehe Folie 103.

Teste, ob $T(M) = \emptyset$ gilt.

Inklusionsproblem:

Seien M_1 und M_2 NFAs.

Frage: Gilt $T(M_1) \subseteq T(M_2)$?

Lösung: Aus M_1 und M_2 können wir einen NFA M mit $T(M) = \overline{T(M_2)} \cap T(M_1)$ konstruieren.

Es gilt: $T(M_1) \subseteq T(M_2)$ genau dann, wenn $T(M) = \emptyset$.

Äquivalenzproblem:

Seien M_1 und M_2 NFAs.

Frage: Gilt $T(M_1) = T(M_2)$?

Lösung 1:

Es gilt: $T(M_1) = T(M_2)$ genau dann, wenn $T(M_1) \subseteq T(M_2)$ und $T(M_2) \subseteq T(M_1)$.

Lösung 2:

Bestimme zu M_i ($i \in \{1, 2\}$) einen äquivalenten minimalen DFA N_i.

Dann gilt: $T(M_1) = T(M_2) \Leftrightarrow T(N_1) = T(N_2) \Leftrightarrow N_1$ und N_2 sind isomorph (d.h. können durch Umbenennung der Zustände ineinander überführt werden).

Entscheidbarkeit

Effizienzbetrachtungen:

Je nachdem, in welcher Darstellung eine reguläre Sprache L gegeben ist, kann die Komplexität der oben beschriebenen Verfahren sehr unterschiedlich ausfallen.

Beispiel: Äquivalenzproblem $L_1 = L_2$:

- L_1, L_2 gegeben als DFAs

  $\rightsquigarrow$ Komplexität $O(n^2)$

- L_1, L_2 gegeben als Grammatiken, reguläre Ausdrücke oder NFAs

  $\rightsquigarrow$ Komplexität NP-hart

  Das bedeutet unter anderem: Es ist nicht bekannt, ob dieses Problem in polynomieller Zeit lösbar ist.

  Mehr zur Komplexitätsklasse NP und verwandten Fragestellungen $\rightsquigarrow$ Master Vorlesung Komplexitätstheorie

Kontextfreie Sprachen

Wir behandeln nun die kontextfreien oder Typ-2-Sprachen.

Wiederholung: Produktionen kontextfreier Grammatiken

Bei kontextfreien Grammatiken haben alle Produktionen die Form $A \to w$, wobei $A \in V$ (d.h., A ist eine Variable) und $w \in (V \cup \Sigma)^+$.

Ausnahme (ε-Sonderregelung): $S \to \varepsilon$, dann darf das Startsymbol S nicht auf der rechten Seite einer Produktion vorkommen.

Betrachtete Beispielgrammatiken:

- Grammatik, die korrekt geklammerte arithmetische Ausdrücke erzeugt
- Grammatik, die Sätze der natürlichen Sprache erzeugt

Ein weiteres Beispiel: die Sprache $L = \{a^k b^k \mid k \geq 0\}$ ist kontextfrei.

Produktionen: $S \to \varepsilon \mid T$, $T \to ab \mid aTb$

Anwendungen kontextfreier Sprachen

Hauptanwendung: Beschreibung der Syntax von Programmiersprachen

Viele der hier besprochenen Techniken sind daher interessant für den Einsatz im Compilerbau.

Bemerkung: eine Grammatik, die eine natürliche Sprache beschreibt, kann trotz mancher kontextfreier Bestandteile nicht kontextfrei sein, da bei Sprache viele subtile Kontextabhängigkeiten berücksichtigt werden müssen.

Bisher ist es auch noch niemandem gelungen eine vollständige Grammatik aller korrekten natürlichsprachigen Sätze zu bilden.
Frage: Was ist überhaupt ein korrekter Satz?

Kontextfreie Sprachen

Inhalt des Abschnitts "Kontextfreie Sprachen"

- Normalformen – wichtig für die Anwendung bestimmter Verfahren/Techniken ist es, eine Grammatik in eine bestimmte Normalform zu bringen.
- Pumping-Lemma für kontextfreie Sprachen
- Abschlusseigenschaften – die kontextfreien Sprachen verhalten sich hier nicht ganz so gutartig wie die regulären Sprachen.
- Wortproblem – und der Algorithmus, um das Wortproblem zu lösen (CYK-Algorithmus)
- Kellerautomaten – das Automatenmodell zu kontextfreien Sprachen

Normalformen

Wir beschäftigen uns zunächst noch einmal mit der "ε-Sonderregelung":

Die Definition für kontextfreie Grammatiken (mit ε-Sonderregelung) fordert, dass S auf keiner rechten Seite auftauchen darf, wenn $S \to \varepsilon$ als Produktion vorkommt. Außerdem dürfen keine weiteren Produktionen der Form $A \to \varepsilon$ auftauchen.

Was passiert, wenn man diese Bedingungen aufhebt und beliebige Regeln der Form $A \to \varepsilon$ erlaubt? Kann es dann passieren, dass man eine nicht-kontextfreie Sprache erzeugt?

Antwort: nein

Satz (ε-freie Grammatiken)

Gegeben sei eine Grammatik $G = (V, \Sigma, P, S)$, deren Produktionen alle von der Form $A \to w$ für $A \in V$, $w \in (V \cup \Sigma)^*$ sind.
Dann gibt es eine Grammatik $G' = (V, \Sigma, P', S)$, deren Produktionen alle von der Form $A \to w$ für $A \in V$, $w \in (V \cup \Sigma)^+$ sind, so dass $L(G') = L(G) \setminus \{\varepsilon\}$.

Man darf also ε-Produktionen beliebig verwenden. Sie verändern nichts an der Ausdrucksmächtigkeit kontextfreier Grammatiken.

Beweis:

Sei $V_\varepsilon = \{A \in V \mid A \Rightarrow_G^* \varepsilon\}$.

Die Menge V_ε können wir leicht berechnen!

Normalformen

Für ein Wort $w \in (V \cup \Sigma)^+$ definieren wir die Menge von Wörtern $F(w) \subseteq (V \cup \Sigma)^+$ wie folgt:

Sei $w = w_0 A_1 w_1 A_2 \cdots w_{n-1} A_n w_n$ wobei $n \geq 0$, $A_1, \ldots, A_n \in V_\varepsilon$ und in dem Wort $w_0 w_1 \cdots w_n$ keine Variable aus V_ε vorkommt. Dann sei

$$F(w) = \{w_0 u_1 w_1 u_2 \cdots w_{n-1} u_n w_n \mid \forall 1 \leq i \leq n : u_i \in \{A_i, \varepsilon\}\} \setminus \{\varepsilon\}.$$

Wir können nun die Produktionsmenge P' der ε-freien Grammatik G' wie folgt definieren:

$$P' = \{A \to w' \mid \exists w : (A \to w) \in P \text{ und } w' \in F(w)\}.$$

Behauptung: $L(G') = L(G) \setminus \{\varepsilon\}$

Normalformen

$L(G') \subseteq L(G) \setminus \{\varepsilon\}$:

Nach Konstruktion von G' gilt $\varepsilon \notin L(G')$.

Ausserdem gilt für jede Produktion $(A \to w) \in P'$ von G':

$$A \Rightarrow_G^* w.$$

Dies impliziert $L(G') \subseteq L(G)$.

$L(G) \setminus \{\varepsilon\} \subseteq L(G')$:

Mittels Induktion über die Länge von Ableitungen zeigen wir für alle Nichtterminale $A \in N$ und $w \in \Sigma^+$:

$$A \Rightarrow_G^* w \quad \textit{impliziert} \quad A \Rightarrow_{G'}^* w.$$

Gelte also $A \Rightarrow_G^* w$.

Normalformen

Wenn $(A \to w) \in P$, dann $(A \to w) \in P'$ und somit $A \Rightarrow_{G'}^* w$.

Angenommen die Ableitung $A \Rightarrow_G^* w$ hat Länge mindestens 2.

$\rightsquigarrow$ Es gibt eine Produktion $(A \to w_0 A_1 w_1 A_2 w_2 \cdots A_n w_n) \in P$ und kürzere Ableitungen $A_i \Rightarrow_G^* u_i$ $(1 \leq i \leq n)$, so dass $w = w_0 u_1 w_1 u_2 w_2 \cdots u_n w_n$.

Sei $J = \{i \mid 1 \leq i \leq n, u_i = \varepsilon\}$.

Sei w' das Wort, dass aus $w_0 A_1 w_1 A_2 w_2 \cdots A_n w_n$ entsteht, indem alle A_i mit $i \in J$ durch ε ersetzt werden.

$\rightsquigarrow (A \to w') \in P'$.

Ausserdem gilt nach Induktion: $A_i \Rightarrow_{G'}^* u_i$ für alle $i \in \{1, \ldots, n\} \setminus J$.

$\rightsquigarrow A \Rightarrow_{G'}^* w$ □

Normalformen

Wir betrachten nun eine weitere wichtige Normalform:

Definition (Chomsky-Normalform)

Eine kontextfreie Grammatik G mit $\varepsilon \notin L(G)$ heißt in Chomsky-Normalform (kurz CNF), falls alle Produktionen eine der folgenden zwei Formen haben:

$$A \to BC \qquad A \to a$$

Dabei sind $A, B, C \in V$ Variablen und $a \in \Sigma$ ein Terminalsymbol.

Satz (Umwandlung in Chomsky-Normalform)

Zu jeder kontextfreien Grammatik G mit $\varepsilon \notin L(G)$ gibt es eine Grammatik G' in Chomsky-Normalform mit $L(G) = L(G')$.

Beweis:

Schritt 1:

Auf Grund des Satzes "ε-freie Grammatiken" können wir davon ausgehen, dass G keine Produktionen der Form $A \longrightarrow \varepsilon$ hat.

Schritt 2:

Wir führen für jedes Terminalsymbol $a \in \Sigma$ eine neue Variable $A_a \notin V$ zusammen mit der Produktion $A_a \longrightarrow a$ ein.

Dann können wir jedes Vorkommen von a in einer rechten Seite $\neq a$ durch A_a ersetzen.

$\rightsquigarrow$ alle Produktionen haben dann die Form $A \longrightarrow a$ oder $A \longrightarrow A_1 \cdots A_n$ mit $a \in \Sigma$, $n \geq 1$ und Variablen $A_1, \ldots, A_n$.

Schritt 3: Elimination von Kettenregeln.

Wir eliminieren nun alle Produktionen der Form $A \to B$ für Variablen A, B (Kettenregeln) wie folgt:

Für jede Variable A fügen wir die Produktion $A \to \alpha$ hinzu, falls α keine Variable ist und eine Variable B mit $A \Rightarrow^* B \to \alpha$ existiert.

Dannach können wir alle Kettenregeln weglassen.

$\rightsquigarrow$ alle Produktionen haben die Form $A \to a$ oder $A \to A_1 \cdots A_n$ mit $a \in \Sigma$, $n \geq 2$ und Variablen $A_1, \ldots, A_n$.

Schritt 4: Elimination von Produktionen der Form $A \to A_1 \cdots A_n$ mit $n \geq 3$.

Sei $A \to A_1 \cdots A_n$ eine Produktion mit $n \geq 3$.

Wir führen neue Variablen $B_2, \ldots, B_{n-1}$ ein und ersetzen die Produktion $A \to A_1 \cdots A_n$ durch die folgenden Produktionen:

$$A \to A_1 B_2, \quad B_i \to A_i B_{i+1} \ (2 \leq i \leq n-2), \quad B_{n-1} \to A_{n-1} A_n$$

□

Normalformen

Beispiel: Sei

$$G = (\{S, A\}, \{a, b, c\}, P, S)$$

mit folgender Produktionenmenge P:

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow S \mid aaSc \mid \varepsilon \end{aligned}$$

Wir wandeln G in CNF um.

Schritt 1: Wir machen G ε-frei.

Dies ergibt die Produktionen

$$\begin{aligned} S &\rightarrow aAb \mid ab \\ A &\rightarrow S \mid aaSc \end{aligned}$$

Schritt 2: Dies ergibt die Produktionen

$$
\begin{aligned}
S &\rightarrow A_a A A_b \mid A_a A_b \\
A &\rightarrow S \mid A_a A_a S A_c \\
A_a &\rightarrow a \\
A_b &\rightarrow b \\
A_c &\rightarrow c
\end{aligned}
$$

Normalformen

Schritt 3: Elimination von Kettenregeln.

Die einzige Kettenregel unserer Grammatik ist $A \to S$. Deren Elimination liefert die folgenden Produktionen:

$$
\begin{aligned}
S &\to A_a A A_b \mid A_a A_b \\
A &\to A_a A A_b \mid A_a A_b \mid A_a A_a S A_c \\
A_a &\to a \\
A_b &\to b \\
A_c &\to c
\end{aligned}
$$

Schritt 4: Elimination von Regeln der Gestalt $A \longrightarrow A_1 \cdots A_n$ mit $n \geq 3$.

$$
\begin{array}{rcl}
S & \longrightarrow & A_a B \mid A_a A_b \\
A & \longrightarrow & A_a B \mid A_a A_b \mid A_a C \\
B & \longrightarrow & A A_b \\
C & \longrightarrow & A_a S A_c \\
A_a & \longrightarrow & a \\
A_b & \longrightarrow & b \\
A_c & \longrightarrow & c
\end{array}
$$

Schritt 4: Elimination von Regeln der Gestalt $A \to A_1 \cdots A_n$ mit $n \geq 3$.

$$
\begin{array}{lcl}
S & \to & A_a B \mid A_a A_b \\
A & \to & A_a B \mid A_a A_b \mid A_a C \\
B & \to & A A_b \\
C & \to & A_a D \\
D & \to & S A_c \\
A_a & \to & a \\
A_b & \to & b \\
A_c & \to & c
\end{array}
$$

Normalformen

Definition (Greibach-Normalform)

Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit $\varepsilon \notin L(G)$ ist in Greibach-Normalform, falls alle Produktionen aus P folgende Form haben:

$$A \to aB_1B_2 \dots B_k \qquad \text{mit } k \geq 0$$

Dabei sind $A, B_1, \cdots, B_k \in V$ Variablen und $a \in \Sigma$ ein Alphabetsymbol.

Die Greibach-Normalform garantiert, dass bei jedem Ableitungsschritt genau ein Alphabetsymbol entsteht.
Sie ist nützlich, um zu zeigen, dass Kellerautomaten (d.h., Automaten für kontextfreie Sprachen) keine ε-Übergänge brauchen.

Satz (Umwandlung in Greibach-Normalform)

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

Beweis: Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik mit $\varepsilon \notin L(G)$.

Vorüberlegung:

Angenommen in P gibt es für eine Variable A die folgenden Produktionen:

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_k \mid \beta_1 \mid \cdots \mid \beta_\ell.$$

Hierbei sind $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_\ell \in (V \cup \Sigma)^*$ und $\beta_1, \ldots, \beta_\ell$ beginnen nicht mit A.

Dann kann man mit diesen Produktionen genau die gleichen Satzformen erzeugen wie mit

$$A \to \beta_1 \mid \cdots \mid \beta_\ell \mid \beta_1 B \mid \cdots \mid \beta_\ell B$$
$$B \to \alpha_1 \mid \cdots \mid \alpha_k \mid \alpha_1 B \mid \cdots \mid \alpha_k B.$$

Mit beiden Regelsätzen lassen sich alle Satzformen aus

$$(\beta_1 \mid \cdots \mid \beta_\ell)(\alpha_1 \mid \cdots \mid \alpha_k)^*$$

erzeugen.

Normalformen

Sei nun $A_1, \ldots, A_m$ eine beliebige Aufzählung aller Variablen von G.

Schritt 1: Mit dem Algorithmus auf der nächsten Folie formen wir G in eine äquivalente kontextfreie Grammatik um, in der für alle Produktionen der Form $A_i \to \alpha$ gilt:

$$\alpha = a\beta \text{ mit } a \in \Sigma, \beta \in V^* \text{ oder } \alpha = A_j\beta \text{ mit } j > i, \beta \in V^*$$

O.B.d.A. können wir davon ausgehen, dass G in Chomsky-Normalform ist.

Normalformen

for $i := 1$ **to** m **do**
  for $j := 1$ **to** $i - 1$ **do**
    for all $(A_i \to A_j\alpha) \in P$ **do**
      Seien $A_j \to \beta_1 \mid \cdots \mid \beta_n$ alle Regeln mit linker Seite $= A_j$.
      $P := (P \cup \{A_i \to \beta_1\alpha \mid \cdots \mid \beta_n\alpha\}) \setminus \{A_i \to A_j\alpha\}$
    endfor
  endfor
  if es gibt Produktionen der Form $A_i \to A_i\alpha$ **then**
    Wende die Transformation aus der Vorüberlegung auf A_i an
    (dabei wird eine neue Variable B_i eingeführt).
  endif
endfor

Normalformen

Nach Schritt 1 sind insbesondere alle Produktionen mit linker Seite $= A_m$ von der Form $A_m \to a\alpha$ mit $a \in \Sigma, \alpha \in V^*$.

Schritt 2: Der folgende Algorithmus erzwingt, dass alle Produktionen mit linker Seite A_i rechts mit einem Terminalsymbol beginnen.

for $i := m-1$ **downto** 1 **do**
    forall $(A_i \to A_j\alpha) \in P$ mit $j > i$ **do**
        Seien $A_j \to \beta_1 \mid \cdots \mid \beta_n$ alle Regeln mit linker Seite $= A_j$.
        $P := (P \cup \{A_i \to \beta_1\alpha \mid \cdots \mid \beta_n\alpha\}) \setminus \{A_i \to A_j\alpha\}$
    endfor
endfor

Nach Schritt 2 sind alle Produktionen mit linker Seite $= A_i$ $(1 \leq i \leq m)$ in Greibach-Normalform.

Aber: Die in Schritt 1 eingeführten Produktionen für die neuen Variablen B_i könnten nicht in Greibach-Normalform sein.

Normalformen

Sei $B_i \to A_j\alpha$ eine solche Regel, die die Greibach-Normalform verletzt.

Seien $A_j \to \beta_1 \mid \cdots \mid \beta_k$ alle Produktionen mit linker Seite $= A_j$.

Dann beginnen $\beta_1, \ldots, \beta_k$ mit Terminalsymbolen.

Ersetze $B_i \to A_j\alpha$ durch $B_i \to \beta_1\alpha \mid \cdots \mid \beta_k\alpha$.

Nun ist die Grammatik in Greibach-Normalform. □

Beispiel: Sei G die Grammatik in CNF mit folgenden Produktionen:

$$\begin{aligned} A_1 &\to A_2 A_3 \\ A_2 &\to A_3 A_1 \mid b \\ A_3 &\to A_1 A_2 \mid a. \end{aligned}$$

In Schritt 1 wird nur die Produktion $A_3 \to A_1 A_2$ im Durchlauf $i = 3$ wie folgt ersetzt:

- Bei $j = 1$: $A_3 \to A_2 A_3 A_2$
- Bei $j = 2$: $A_3 \to A_3 A_1 A_3 A_2 \mid b A_3 A_2$

Nun wird eine neue Variable B_3 eingeführt, und die Produktionen

$$A_3 \to A_3 A_1 A_3 A_2 \mid b A_3 A_2 \mid a$$

werden ersetzt durch

$$\begin{aligned} A_3 &\to b A_3 A_2 B_3 \mid a B_3 \mid b A_3 A_2 \mid a \\ B_3 &\to A_1 A_3 A_2 B_3 \mid A_1 A_3 A_2. \end{aligned}$$

Normalformen

Wir haben also nach Schritt 1 folgende Grammatik vorliegen:

$$\begin{aligned} A_1 &\to A_2 A_3 \\ A_2 &\to A_3 A_1 \mid b \\ A_3 &\to b A_3 A_2 B_3 \mid a B_3 \mid b A_3 A_2 \mid a \\ B_3 &\to A_1 A_3 A_2 B_3 \mid A_1 A_3 A_2. \end{aligned}$$

Beachte: Alle Produktionen für A_3 beginnen in der Tat mit einem Terminalsymbol auf der rechten Seite.

Nach Schritt 2, Durchlauf $i = 2$:

$$\begin{aligned} A_1 &\to A_2 A_3 \\ A_2 &\to b A_3 A_2 B_3 A_1 \mid a B_3 A_1 \mid b A_3 A_2 A_1 \mid a A_1 \mid b \\ A_3 &\to b A_3 A_2 B_3 \mid a B_3 \mid b A_3 A_2 \mid a \\ B_3 &\to A_1 A_3 A_2 B_3 \mid A_1 A_3 A_2 \end{aligned}$$

Nach Schritt 2, Durchlauf $i = 1$:

$$
\begin{aligned}
A_1 &\to bA_3A_2B_3A_1A_3 \mid aB_3A_1A_3 \mid bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3 \\
A_2 &\to bA_3A_2B_3A_1 \mid aB_3A_1 \mid bA_3A_2A_1 \mid aA_1 \mid b \\
A_3 &\to bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\
B_3 &\to A_1A_3A_2B_3 \mid A_1A_3A_2
\end{aligned}
$$

Nun muss noch in den rechten Seiten der B_3-Produktionen A_1 durch die rechten Seiten von A_1 ersetzt werden:

Normalformen

$$
\begin{aligned}
A_1 &\to bA_3A_2B_3A_1A_3 \mid aB_3A_1A_3 \mid bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3 \\
A_2 &\to bA_3A_2B_3A_1 \mid aB_3A_1 \mid bA_3A_2A_1 \mid aA_1 \mid b \\
A_3 &\to bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\
B_3 &\to bA_3A_2B_3A_1A_3A_3A_2B_3 \mid aB_3A_1A_3A_3A_2B_3 \mid bA_3A_2A_1A_3A_3A_2B_3 \mid \\
&\quad aA_1A_3A_3A_2B_3 \mid bA_3A_3A_2B_3 \mid bA_3A_2B_3A_1A_3A_3A_2 \mid \\
&\quad aB_3A_1A_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2 \mid aA_1A_3A_3A_2 \mid bA_3A_3A_2
\end{aligned}
$$

Normalformen

Bemerkung zum leeren Wort ε: Mit Grammatiken in Chomsky-Normalform bzw. Greibach-Normalform lassen sich nur kontextfreie Sprachen L mit $\varepsilon \notin L$ erzeugen.

Hat man nun eine kontextfreie Grammatik G mit $\varepsilon \in L(G)$ vorliegen, so kann man wie folgt vorgehen:

- Konstruiere aus G eine kontextfreie Grammatik G' mit $L(G') = L(G) \setminus \{\varepsilon\}$ (siehe Satz "ε-freie Grammatiken").
- Wandle G' in eine Grammatik G'' in Chomsky-Normalform bzw. Greibach-Normalform um.

  Sei S das Startsymbol von G'' und seien $S \to \alpha_1 \mid \cdots \mid \alpha_n$ alle Produktionen in G'' mit linker Seite $= S$.
- Nimm ein neues Startsymbol S' und füge zu G'' die Produktionen $S' \to \varepsilon \mid \alpha_1 \mid \cdots \mid \alpha_n$ hinzu.

  Für die resultierende Grammatik H gilt $L(G) = L(H)$, und bis auf die Produktion $S' \to \varepsilon$ sind alle Produktionen in H in Chomsky-Normalform bzw. Greibach-Normalform.

Pumping-Lemma

Weitgehend analog zu regulären Sprachen kann man nun ein Pumping-Lemma für kontextfreie Sprachen zeigen.

Die für reguläre Sprachen und endliche Automaten geltende Aussage

> Jedes ausreichend lange Wort durchläuft einen Zustand des Automaten zweimal.

wird dabei ersetzt durch

> Auf einem Pfad des Syntaxbaums, der die Ableitung eines ausreichend langen Wortes durch eine kontextfreie Grammatik darstellt, kommt eine Variable mindestens zweimal vor.

Pumping-Lemma

Was bedeutet hier "ausreichend langes Wort"?

Die Beantwortung dieser Frage hängt davon ab, in welcher Form sich die Grammatik befindet.

Wir nehmen an, sie befinde sich in Chomsky-Normalform.

Dann gilt: Syntaxbäume sind (bis auf die unterste Schicht der Blätter) immer Binärbäume (aufgrund der Produktionen der Form $A \to BC$).

Für Binärbäume gilt nun:

Lemma (Länge von Pfaden in Binärbäumen)

Sei B ein Binärbaum (d.h., jeder Knoten in B hat entweder null oder zwei Kinder) mit mindestens 2^k Blättern.
Dann hat B einen von der Wurzel ausgehenden Pfad, der aus mindestens k Kanten und $k+1$ Knoten besteht.

Normalformen

Beweis: Induktion über k.

IA: $k = 0$.

Sei B ein Binärbaum mit mindestens $2^0 = 1$ Blättern.

Dann hat B einen Pfad, der aus mindestens einem Knoten besteht.

IS: $k \geq 0$.

Sei B ein Binärbaum mit mindestens $2^{k+1} = 2^k + 2^k$ Blättern.

Seien v_1 und v_2 die beiden Kinder der Wurzel, und seien B_1 und B_2 die Binärbäume mit Wurzel v_1 bzw. v_2.

Dann muss B_1 oder B_2 mindestens 2^k Blätter haben.

O.B.d.A. habe B_1 mindestens 2^k viele Blätter.

IH $\rightsquigarrow$ in B_1 gibt es einen Pfad mit mindestens k Kanten und $k+1$ Knoten.

$\rightsquigarrow$ in B gibt es einen Pfad, mit mindestens $k+1$ Kanten und $k+2$ Knoten.

Pumping-Lemma

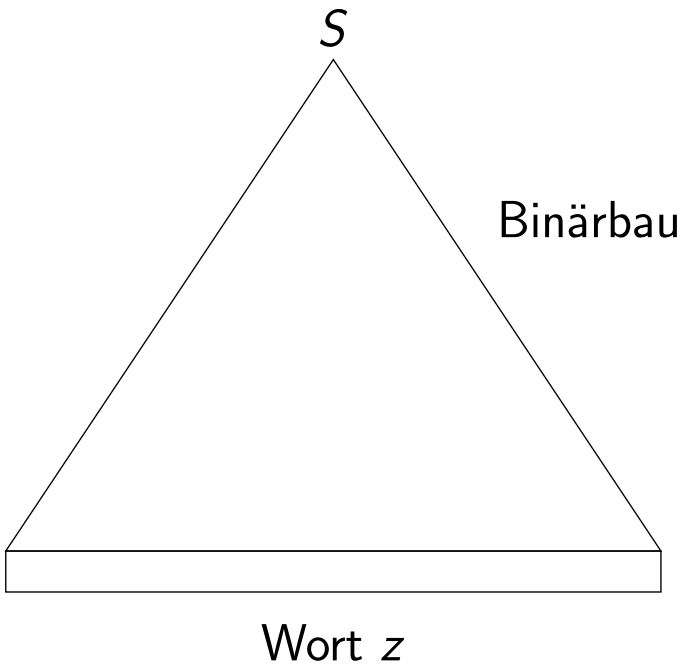
Sei nun $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $k = |V|$ vielen Variablen.

Sei weiter $z \in L(G)$.

- Wenn $|z| \geq 2^k$, dann hat jeder Syntaxbaum für z offensichtlich mindestens 2^k Blätter.
- Das Lemma impliziert, dass der obere Teil eines Syntaxbaums (bei dem die Blätter abgeschnitten sind) einen Pfad mit mindestens $k + 1 > |V|$ vielen Knoten hat.
- Auf diesem Pfad, der nur innere Knoten enthält, muss eine Variable – nennen wir sie A – mindestens zweimal vorkommen.

Pumping-Lemma

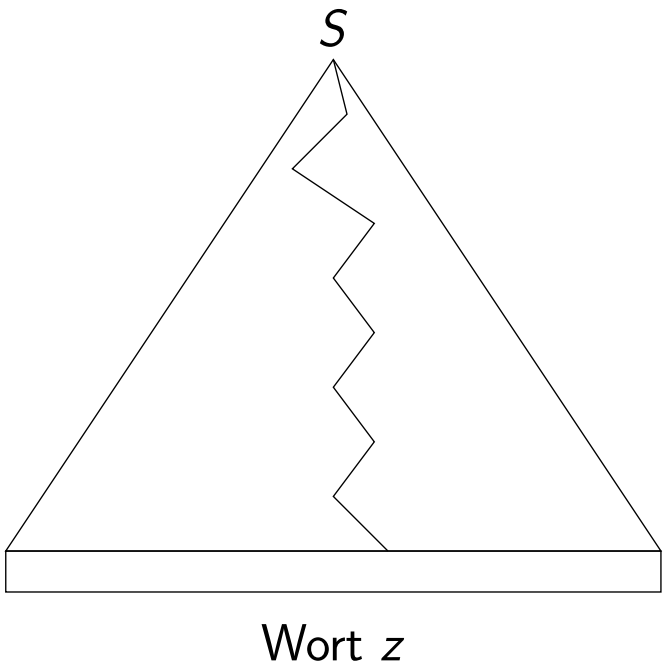
Syntaxbaum für ein Wort z mit $|z| \geq n = 2^k$
n ist hier die "Konstante des Pumping-Lemmas".



Ebene der Blätter
(letzter Ableitungsschritt)

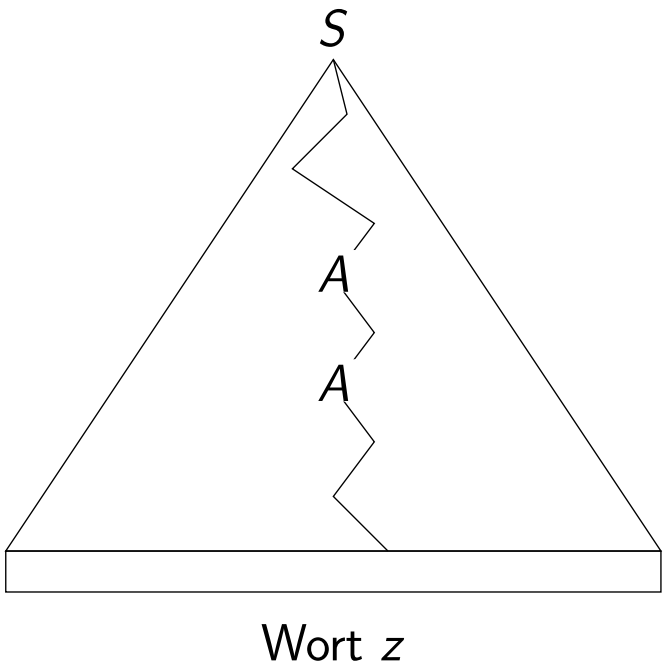
Pumping-Lemma

Es gibt einen Pfad mit mindestens $k + 1$ inneren Knoten.



Pumping-Lemma

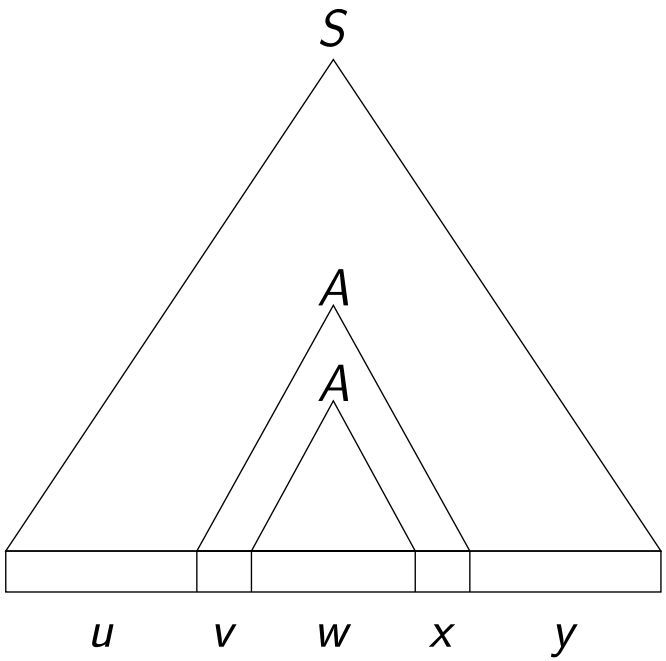
Auf diesem Pfad gibt es eine Variable, die zweimal auftaucht, etwa A.



Pumping-Lemma

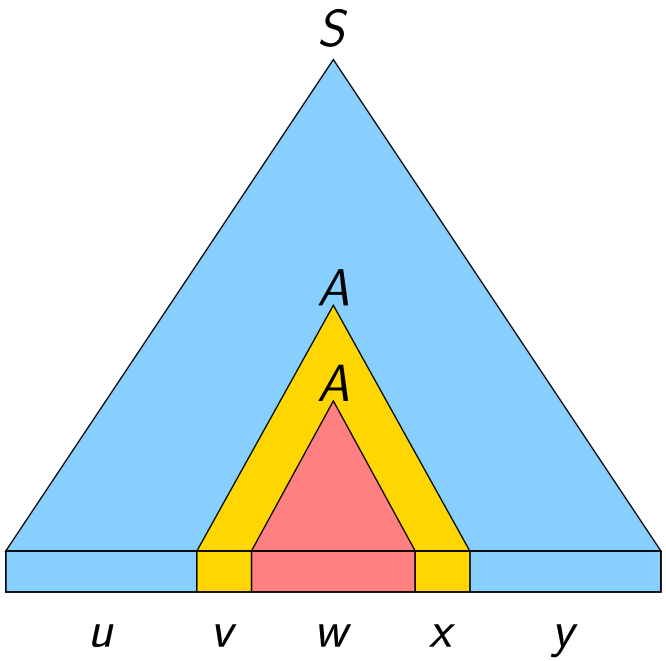
Das Wort z wird nun in fünf Teilwörter u, v, w, x, y aufgespalten:

- w wird aus dem unteren A abgeleitet: $A \Rightarrow^* w$
- vwx wird aus dem oberen A abgeleitet: $A \Rightarrow^* vAx \Rightarrow^* vwx$



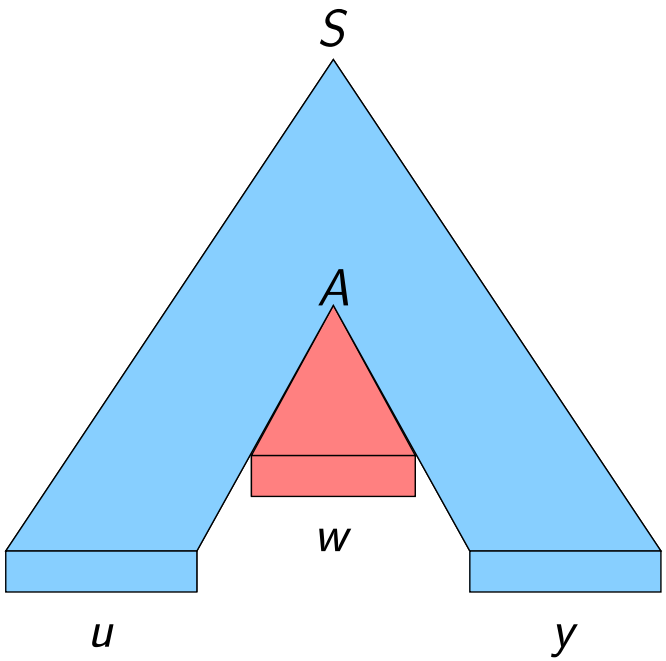
Pumping-Lemma

Damit erhält man drei ineinander enthaltene Teil-Syntaxbäume, die man neu zusammenstecken kann.



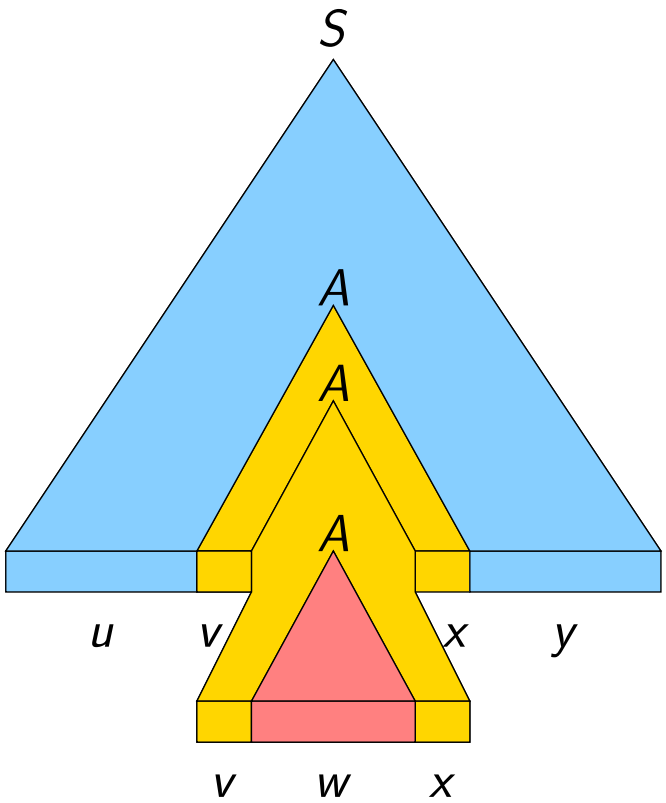
Pumping-Lemma

Durch Weglassen des mittleren Teilbaums erhält man einen Syntaxbaum für uwy. Damit gilt: $uwy \in L(G)$.



Pumping-Lemma

Durch Verdoppeln des mittleren Teilbaums erhält man einen Syntaxbaum für uv^2wx^2y. Damit gilt: $uv^2wx^2y \in L(G)$.



Pumping-Lemma

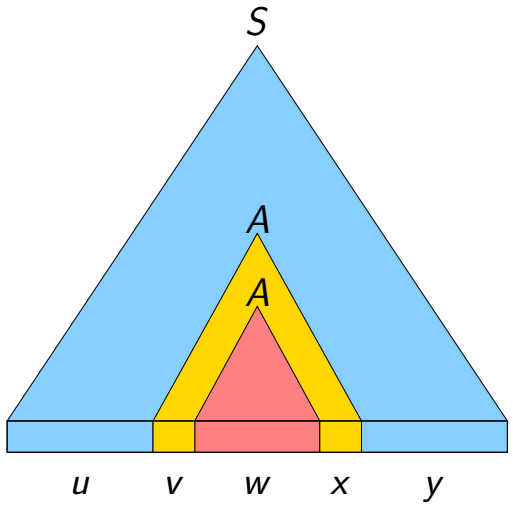
Außerdem kann man für v, w, x folgende Eigenschaften verlangen:

$|vwx| \leq n = 2^k$:

Wir können annehmen, dass wir das am weitesten unten liegende Doppelvorkommen einer Variable gewählt haben, d.h., das Doppelvorkommen mit der größten Tiefe.

Das kann dadurch erreicht werden, dass ein Pfad maximaler Länge von unten nach oben verfolgt wird, bis ein Doppelvorkommen entdeckt wird.

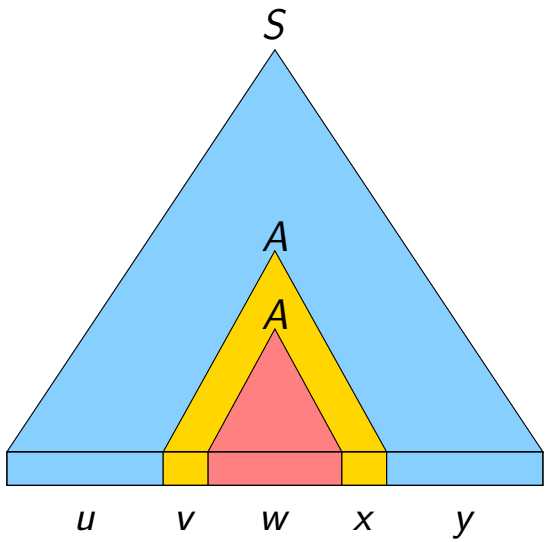
Demnach ist der Abstand des oberen A zur Blattebene höchstens k und der darunter hängende Binärbaum hat höchstens 2^k Blätter.



Pumping-Lemma

$|vx| \geq 1$:

Seien B, C die beiden Kinder des oberen A. Dann geht das untere A entweder aus B oder C hervor. Die jeweils andere Variable muss – da die Grammatik in Chomsy-Normalform ist – ein nicht-leeres Wort ableiten. Und dieses Wort ist ein Teilwort von v oder von x.



Das Pumping-Lemma

Wir haben somit den folgenden Satz bewiesen:

Satz (Pumping-Lemma, *uvwxy*-Theorem)

Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl n, so dass sich alle Wörter $z \in L$ mit $|z| \geq n$ zerlegen lassen als $z = uvwxy$, so dass folgende Eigenschaften erfüllt sind:

1. $|vx| \geq 1$,
2. $|vwx| \leq n$,
3. für alle $i \geq 0$ gilt: $uv^i wx^i y \in L$.

Dabei geht $n = 2^k$ aus der Anzahl k der Variablen einer kontextfreien Grammatik in CNF für L hervor.

Pumping-Lemma

Anwendung des Pumping-Lemmas:

Wir zeigen, dass die Sprache $L = \{a^m b^m c^m \mid m \geq 1\}$ nicht kontextfrei ist.

1. Wir nehmen eine beliebige Zahl n an.
2. Wir wählen ein Wort $z \in L$ mit $|z| \geq n$. In diesem Fall eignet sich $z = a^n b^n c^n$.
3. Wir betrachten nun alle möglichen Zerlegungen $z = uvwxy$ mit den Einschränkungen $|vx| \geq 1$ und $|vwx| \leq n$.

   Wegen $|vwx| \leq n$ gilt, dass vx nicht aus a's, b'c und c's bestehen kann, denn es kann sich nicht über den gesamten b-Block erstrecken.
4. Wir wählen für alle diese möglichen Zerlegungen $i = 2$ und betrachten uv^2wx^2y. Wegen der obigen Überlegungen sind nun ein oder zwei Alphabetsymbole gepumpt worden, mindestens eines jedoch nicht.

   Damit ist klar, dass uv^2wx^2y nicht in L liegen kann, denn jedes Wort in L hat gleich viele a's, b's und c's.

Pumping-Lemma

Man kann auch für folgende Sprachen zeigen, dass sie nicht kontextfrei sind:

$$\begin{aligned} L_1 &= \{0^p \mid p \text{ ist Primzahl}\} \\ L_2 &= \{0^n \mid n \text{ ist Quadratzahl}\} \\ L_3 &= \{0^{2^n} \mid n \geq 0\} \end{aligned}$$

Die Sprachen L_1, L_2, L_3 sind alle unär, d.h., sie sind Sprachen über einem einelementigen Alphabet: $L_1, L_2, L_3 \subseteq \Sigma^*$ mit $|\Sigma| = 1$

Für unäre Sprachen gilt sogar folgender Satz (ohne Beweis).

Satz (unäre kontextfreie Sprachen)

Jede kontextfreie Sprache über einem einelementigen Alphabet ist bereits regulär.

Abschlusseigenschaften

Abgeschlossenheit

Die kontextfreien Sprachen sind abgeschlossen unter:

- Vereinigung (L_1, L_2 kontextfrei $\Rightarrow$ $L_1 \cup L_2$ kontextfrei)
- Produkt/Konkatenation (L_1, L_2 kontextfrei $\Rightarrow$ $L_1 L_2$ kontextfrei)
- Stern-Operation (L kontextfrei $\Rightarrow$ L^* kontextfrei)

Die kontextfreien Sprachen sind nicht abgeschlossen unter:

- Schnitt
- Komplement

Abschluss unter Vereinigung

Wenn L_1 und L_2 kontextfreie Sprachen sind, dann ist auch $L_1 \cup L_2$ kontextfrei.

Begründung: Seien

$$G_1 = (V_1, \Sigma, P_1, S_1), \quad G_2 = (V_2, \Sigma, P_2, S_2)$$

kontextfreie Grammatiken, o.B.d.A. gelte $V_1 \cap V_2 = \emptyset$.

Dann ist

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$$

eine kontextfreie Grammatik mit $L(G) = L(G_1) \cup L(G_2)$.

Abschluss unter Produkt/Konkatenation

Wenn L_1 und L_2 kontextfreie Sprachen sind, dann ist auch $L_1 L_2$ kontextfrei.

Begründung: Seien

$$G_1 = (V_1, \Sigma, P_1, S_1), \quad G_2 = (V_2, \Sigma, P_2, S_2)$$

kontextfreie Grammatiken, o.B.d.A. gelte $V_1 \cap V_2 = \emptyset$.

Dann ist

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

eine kontextfreie Grammatik mit $L(G) = L(G_1)L(G_2)$.

Abschlusseigenschaften

Abschluss unter der Stern-Operation

Wenn L eine kontextfreie Sprache ist, dann ist auch L^* kontextfrei.

Begründung: Sei

$$G_1 = (V_1, \Sigma, P_1, S_1)$$

eine kontextfreie Grammatik.

Dann ist

$$G = (V_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to \varepsilon, S \to S_1 S\}, S)$$

eine kontextfreie Grammatik mit $L(G) = L(G_1)^*$.

Abschlusseigenschaften

Kein Abschluss unter Schnitt

Es gibt kontextfreie Sprachen L_1 und L_2, so dass $L_1 \cap L_2$ nicht kontextfrei ist.

Gegenbeispiel: Die Sprachen

$$\begin{aligned} L_1 &= \{a^j b^k c^k \mid j \geq 0, k \geq 0\} \\ L_2 &= \{a^k b^k c^j \mid j \geq 0, k \geq 0\} \end{aligned}$$

sind beide kontextfrei (einfach).

Für ihren Schnitt gilt jedoch

$$L_1 \cap L_2 = \{a^k b^k c^k \mid k \geq 0\},$$

und diese Sprache ist – wie mit dem Pumping-Lemma gezeigt wurde – nicht kontextfrei.

Kein Abschluss unter Komplement

Es gibt eine kontextfreie Sprache L, so dass $\overline{L} = \Sigma^* \setminus L$ nicht kontextfrei ist.

Begründung:

Nehmen wir an, die kontextfreien Sprachen wären unter Komplement abgeschlossen und seien L_1 und L_2 kontextfrei. Wegen

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

wäre dann auch $L_1 \cap L_2$ kontextfrei.

Dies ist ein Widerspruch zum Gegenbeispiel auf der vorherigen Folie.

Der CYK-Algorithmus

Wir kennen bereits ein Verfahren, mit dem man das Wortproblem für G lösen kann, wobei G eine Typ-1-, Typ-2- oder Typ-3-Grammatik sein kann. Im wesentlichen: Aufzählen aller Wörter bis zu einer bestimmten Länge.

Da dieses Verfahren jedoch exponentielle Laufzeit (in der Länge des Wortes) haben kann, betrachten wir hier ein effizienteres Verfahren für kontextfreie Grammatiken: den CYK-Algorithmus
(entwickelt von Cocke, Younger, Kasami).

Voraussetzung: die Grammatik ist in Chomsky-Normalform, alle Produktionen haben also die Form $A \to a$ oder $A \to BC$.

Der CYK-Algorithmus

Idee: Gegeben sei ein Wort $x \in \Sigma^*$. Wir wollen feststellen, aus welchen Variablen es abgeleitet werden kann.

- **Möglichkeit 1:** $x = a \in \Sigma$, d.h., x besteht aus einem einzigen Alphabetsymbol.

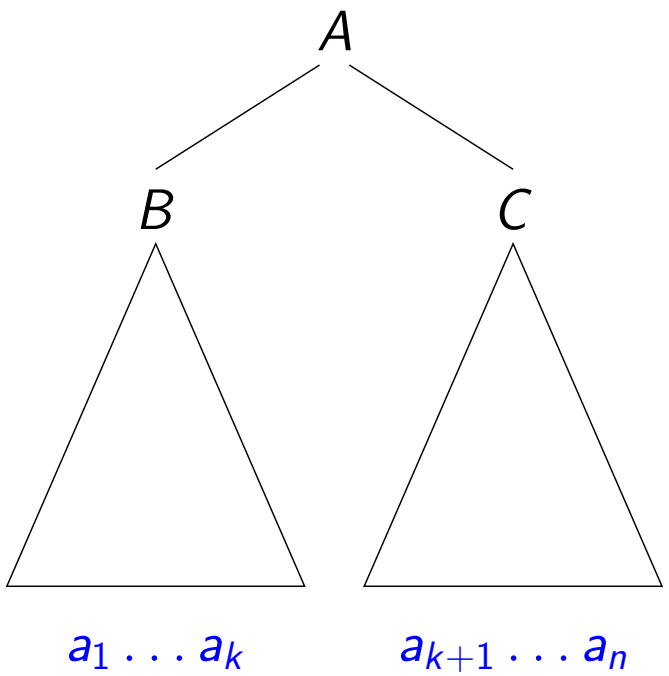
  Dann kann x nur aus Variablen A abgeleitet werden, für die es eine Produktion $A \to a$ gibt.

- **Möglichkeit 2:** $x = a_1 \cdots a_n$ mit $n \geq 2$.

  In diesem Fall gilt: Zunächst muss eine Produktion $A \to BC$ angewandt werden, dann muss ein Teil $a_1 \cdots a_k$ des Wortes aus B und der andere Teil $a_{k+1} \cdots a_n$ aus C abgeleitet werden $(1 \leq k < n)$.

Der CYK-Algorithmus

Möglichkeit 2 läßt sich schematisch folgendermaßen darstellen:



Der CYK-Algorithmus

Es ist jedoch nicht klar, wo das Wort x geteilt werden muss, d.h., wie groß die Position k ist!

Daher: Probiere alle möglichen k's durch. Das heißt:

Gegeben ein Wort $x = a_1 \cdots a_n$.

Für alle k mit $1 \leq k < n$ mache Folgendes:

- Bestimme alle Variablen V_1, aus denen sich $a_1 \cdots a_k$ ableiten lässt.
- Bestimme alle Variablen V_2, aus denen sich $a_{k+1} \cdots a_n$ ableiten lässt.
- Stelle fest, ob es Variablen A, B, C gibt mit $(A \to BC) \in P$, $B \in V_1$ und $C \in V_2$.
  In diesem Fall lässt sich x aus A ableiten .

Der CYK-Algorithmus

Um Mehraufwand zu vermeiden, verwenden wir die Methode der dynamischen Programmierung, das heißt:

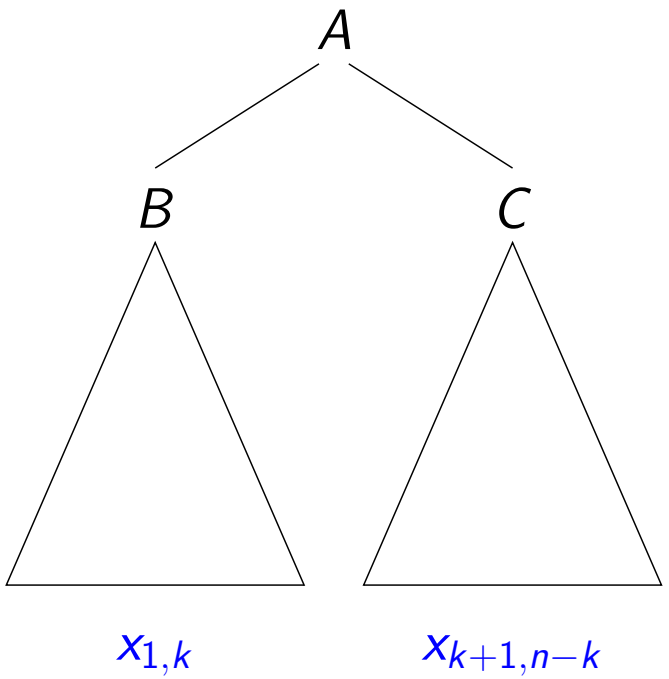
- berechne zuerst alle Variablen, aus denen sich Teilwörter der Länge 1 ableiten lassen,
- berechne dann alle Variablen, aus denen sich Teilwörter der Länge 2 ableiten lassen,
- $\vdots$
- zuletzt berechne alle Variablen, aus denen sich x ableiten läßt. Falls sich die Startvariable S unter diesen Variablen befindet, so liegt x in der von der Grammatik erzeugten Sprache.

Der CYK-Algorithmus

Notation: Wir bezeichnen mit $x_{i,j}$ das Teilwort von x, das an der Stelle i beginnt und die Länge j hat.

$$x = a_1 \cdots a_n \quad \rightsquigarrow \quad x_{i,j} = a_i \cdots a_{i+j-1}$$

Damit sieht das obige Bild folgendermaßen aus:



Der CYK-Algorithmus

Wir bezeichnen mit $T_{i,j}$ die Menge aller Variablen, aus denen sich $x_{i,j}$ ableiten lässt:

$$T_{i,j} = \{A \in V \mid A \Rightarrow_G^* x_{i,j}\}$$

Für $j \geq 2$ lässt sich $T_{i,j}$ aus den Mengen $T_{\ell,k}$ mit $k < j$ folgendermaßen bestimmen:

$$T_{i,j} = \{A \mid \exists (A \to BC) \in P \; \exists 1 \leq k < j : B \in T_{i,k} \text{ und } C \in T_{i+k,j-k}\}$$

Der CYK-Algorithmus

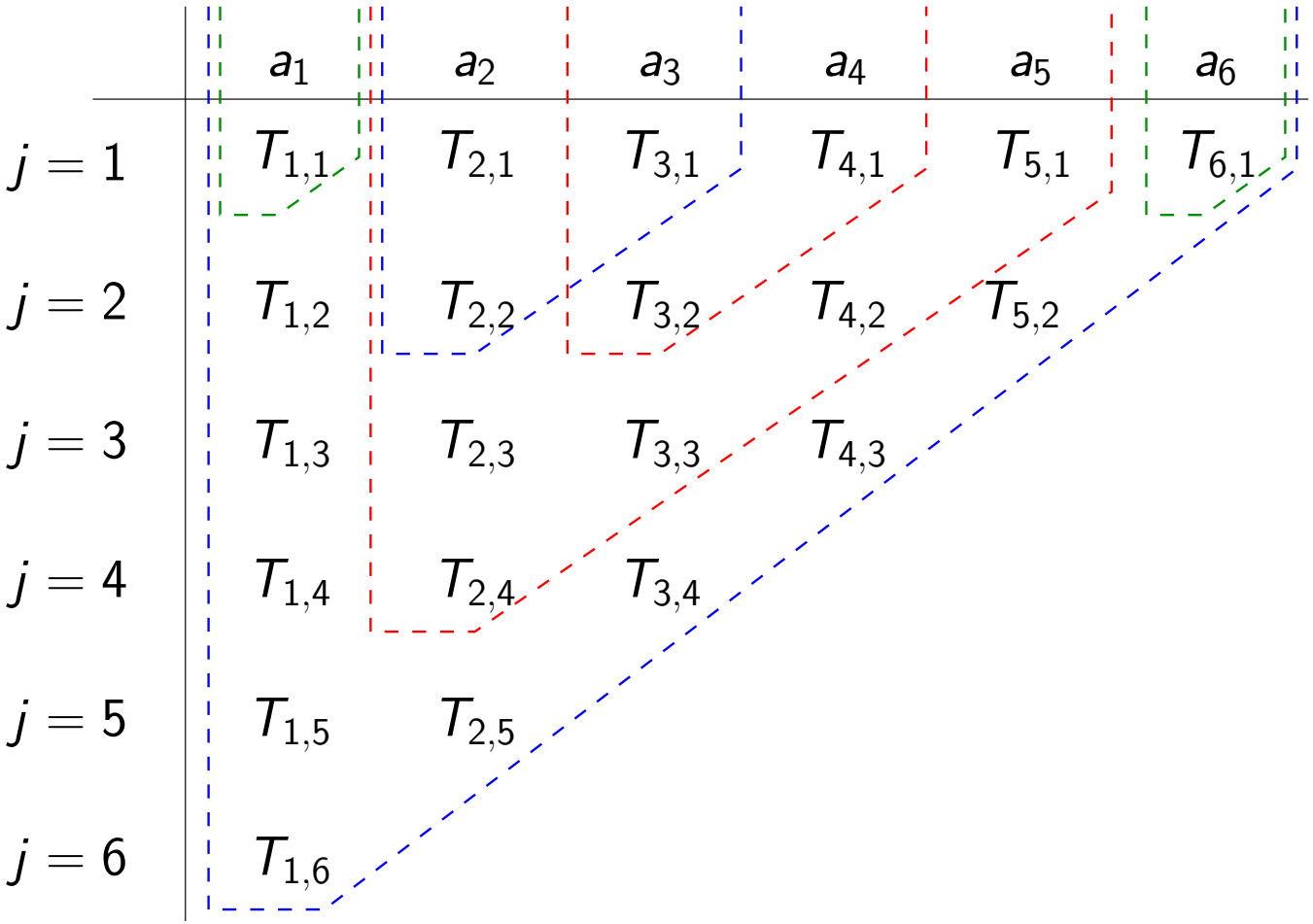
Praktische Ausführung des CYK-Algorithmus:

Wir tragen die Variablenmengen $T_{i,j}$ in folgende Tabelle ein:

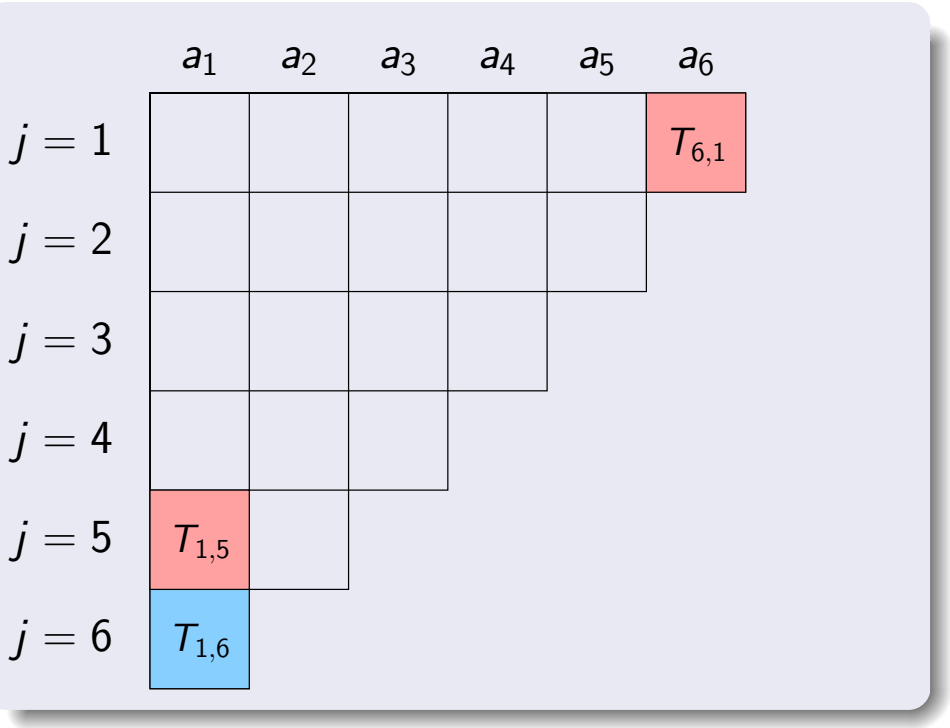
	a_1	a_2	$\ldots$		a_{n-1}	a_n
$j = 1$	$T_{1,1}$	$T_{2,1}$	$\ldots$	$\ldots$	$T_{n-1,1}$	$T_{n,1}$
$j = 2$	$T_{1,2}$	$T_{2,2}$	$\ldots$	$\ldots$	$T_{n-1,2}$	
	$\ldots$	$\ldots$	$\ldots$	$\ldots$		
$\ldots$	$\ldots$	$\ldots$	$\ldots$			
$j = n-1$	$T_{1,n-1}$	$T_{2,n-1}$				
$j = n$	$T_{1,n}$					

Der CYK-Algorithmus

Folgendermaßen lässt sich veranschaulichen, welche Variablenmenge welches Teilwort ableitet:



Der CYK-Algorithmus

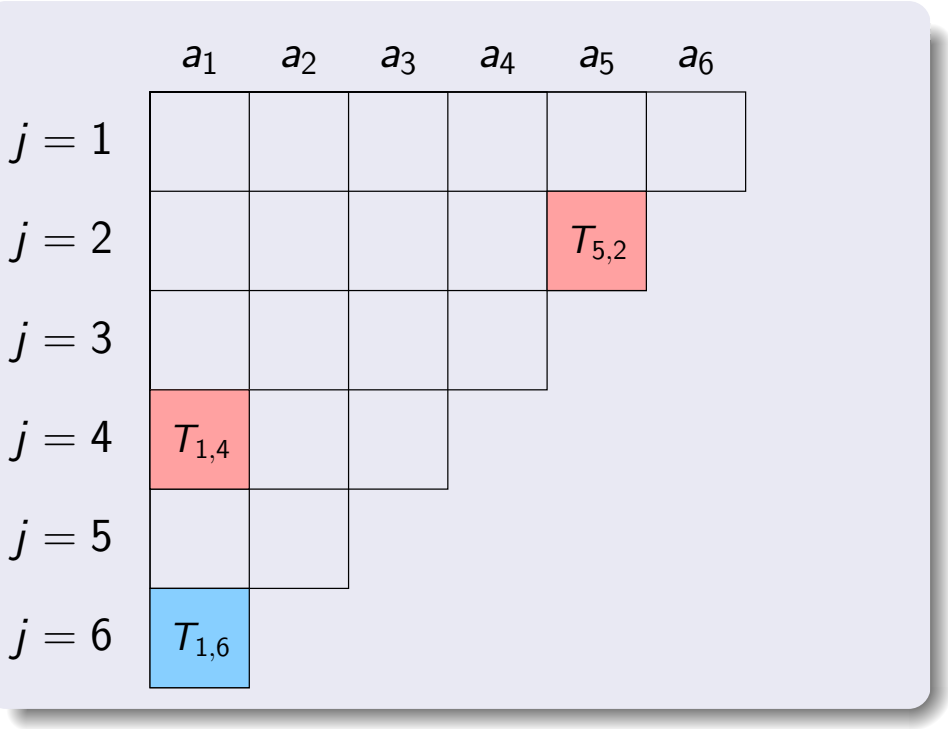


$x = a_1 a_2 a_3 a_4 a_5 \mid a_6$

$(A \to BC) \in P$,

$B \in T_{1,5}$, $C \in T_{6,1} \Rightarrow A \in T_{1,6}$

Der CYK-Algorithmus

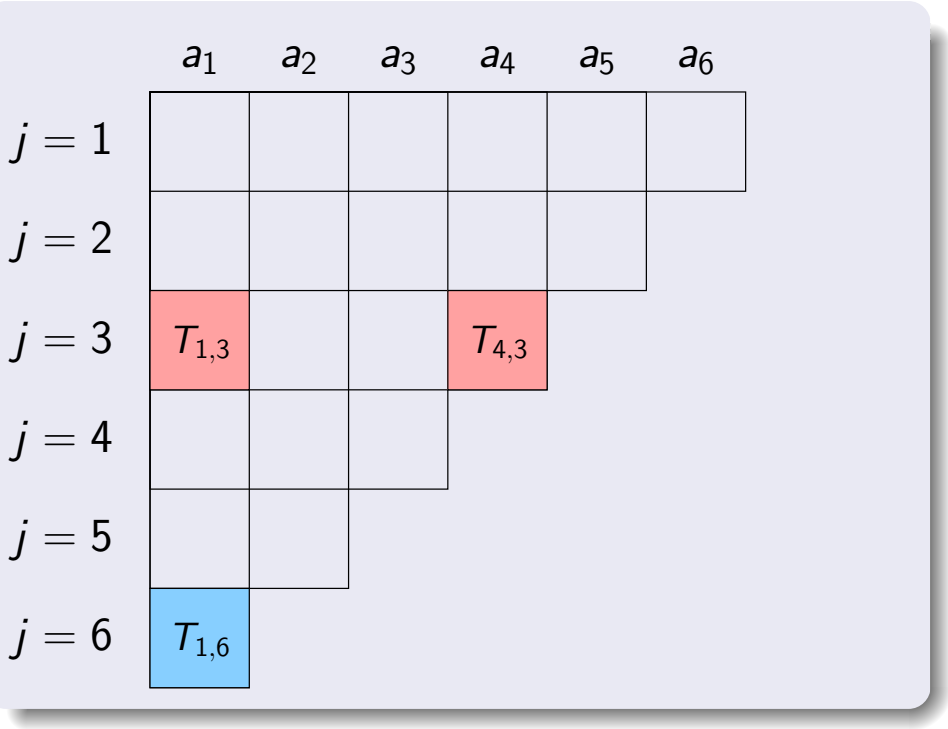


$x = a_1 a_2 a_3 a_4 \mid a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,4}$, $C \in T_{5,2} \Rightarrow A \in T_{1,6}$

Der CYK-Algorithmus

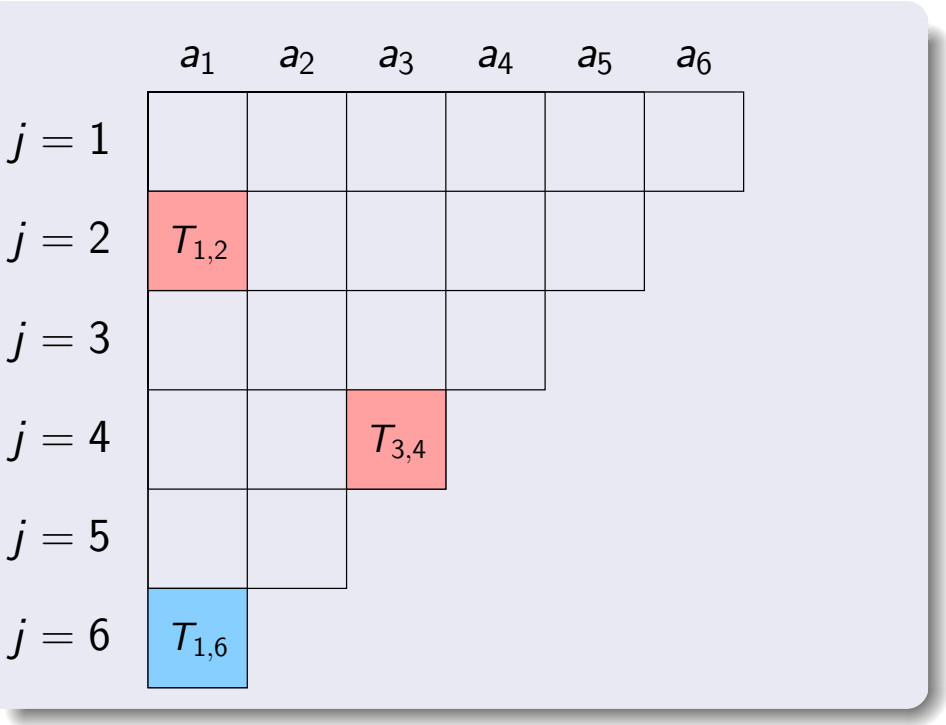


$x = a_1 a_2 a_3 \,|\, a_4 a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,3}$, $C \in T_{4,3} \Rightarrow A \in T_{1,6}$

Der CYK-Algorithmus

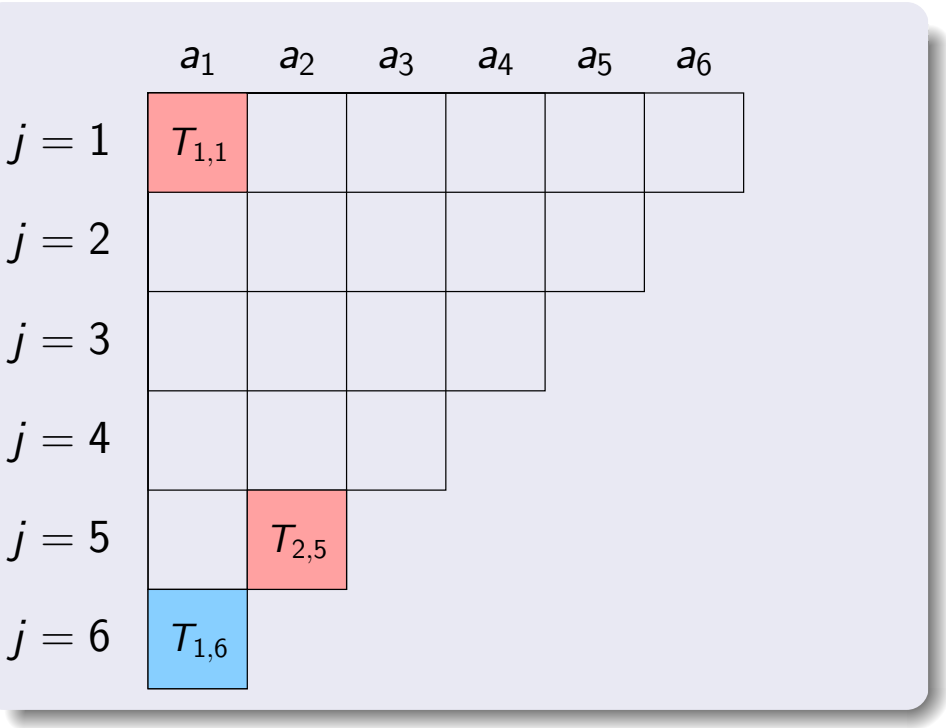


$x = a_1 a_2 \,|\, a_3 a_4 a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,2}$, $C \in T_{3,4} \Rightarrow A \in T_{1,6}$

Der CYK-Algorithmus



$x = a_1 \mid a_2 a_3 a_4 a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,1}$, $C \in T_{2,5} \Rightarrow A \in T_{1,6}$

Der CYK-Algorithmus

Beispiel 1: Betrachte eine Grammatik für die Sprache $L = \{a^k b^k c^j \mid k, j > 0\}$ mit folgenden Produktionen:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow ab \mid aAb \\ B &\rightarrow c \mid cB \end{aligned}$$

Frage: Sei $x = aaabbbcc$. Gilt $x \in L$?

Der CYK-Algorithmus

Beispiel 2: Betrachte eine Grammatik mit folgenden Produktionen:

$$
\begin{aligned}
S &\to AD \mid FG \\
D &\to SE \mid BC \\
E &\to BC \\
F &\to AF \mid a \\
G &\to BG \mid CG \mid b \\
A &\to a \\
B &\to b \\
C &\to c
\end{aligned}
$$

Frage: Sei $x = aabcbc$. Gilt $x \in L$?

Der CYK-Algorithmus

	a	a	b	c	b	c
$j = 1$	A, F	A, F	B, G	C	B, G	C
$j = 2$	F	S	D, E	G	D, E	
$j = 3$	S	S	G			
$j = 4$		S				
$j = 5$	S	D				
$j = 6$	S					

Der CYK-Algorithmus

Komplexität des CYK-Algorithmus

Sei $n = |x|$ die Länge des Wortes, das untersucht wird. Die Größe der Grammatik wird als konstant angesehen. Dann gilt:

- $O(n^2)$ Tabellenfelder müssen ausgefüllt werden.
- Für das Ausfüllen jedes Tabellenfeldes müssen bis zu $O(n)$ andere Felder betrachtet werden.

  (Für $T_{1,n}$ müssen beispielsweise die Felder $T_{1,n-1}$, $T_{n,1}$ und $T_{1,n-2}$, $T_{n-1,2}$ und ... und $T_{1,1}$, $T_{2,n-1}$ betrachtet werden. Insgesamt $n-1$ Paare von Feldern.)

Daher ergibt sich insgesamt als Zeitkomplexität: $O(n^3)$.

Die Zeitkomplexität ist noch polynomiell, aber für das Parsen großer Programme eigentlich nicht mehr geeignet.

Kellerautomaten

Was ist ein geeignetes Automatenmodell für kontextfreie Sprachen?

Analog zu regulären Sprachen suchen wir hier ein Automatenmodell für kontextfreie Sprachen.

Antwort: Kellerautomaten, d.h., Automaten, die mit einem zusätzlichen Keller ausgestattet sind.

Nutzen eines solchen Automatenmodells

Manche Konstruktionen und Verfahren lassen sich besser mit Hilfe des Automatenmodells durchführen (anstatt auf Grammatiken).

- Wortproblem: Wir werden herausfinden, dass das Wortproblem unter bestimmten Umständen effizienter als in Zeit $O(n^3)$ gelöst werden kann.
- Abschlusseigenschaften: Abschluss der kontextfreien Sprachen unter Schnitt mit regulären Sprachen lässt sich gut mit Kellerautomaten zeigen.

Kellerautomaten

Wir betrachten die Sprache

$$L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid a_i \in \Delta\}$$

mit $\Sigma = \Delta \cup \{\$\}$.

Ein endlicher Automat kann diese Sprache deshalb nicht erkennen, weil er sich keine beliebig langen Wörter der Form $a_1 a_2 \cdots a_n$ "merken" kann.

Er müsste sich aber solche Wörter merken, um die Übereinstimmung mit dem Wortteil nach dem \$ zu überprüfen.

Kellerautomaten

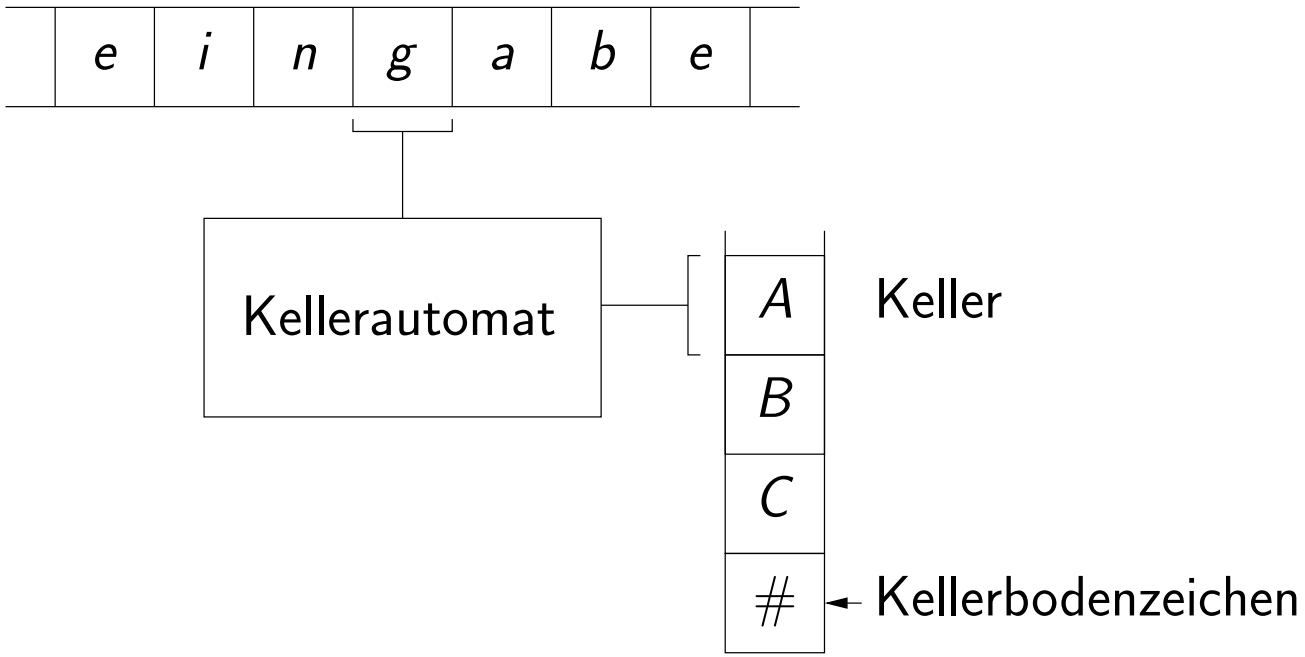
Um ein Automatenmodell für kontextfreie Sprachen zu erhalten,

- führen wir daher einen Keller oder Pushdown-Speicher ein, auf dem sich eine beliebig lange Sequenz von Zeichen befinden darf.
- Beim Einlesen eines neuen Zeichens darf das oberste Zeichen des Kellers gelesen und folgendermaßen verändert werden:
  - Entweder bleibt der Keller unverändert oder
  - das oberste Zeichen des Kellers wird entfernt und durch eine (evtl. leere) Sequenz von Zeichen ersetzt.

  An anderen Stellen darf der Keller nicht gelesen oder verändert werden.

Kellerautomaten

Schematische Darstellung eines Kellerautomaten:



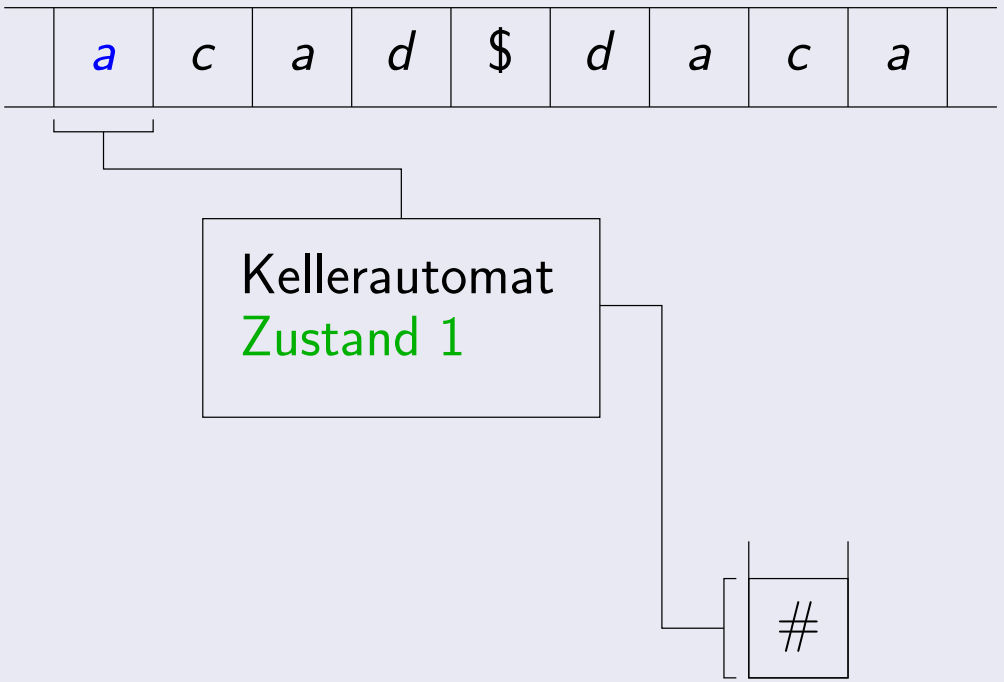
Kellerautomaten

Sei $\Delta = \{a, b, c, d\}$ und $L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid a_i \in \Delta\}$.

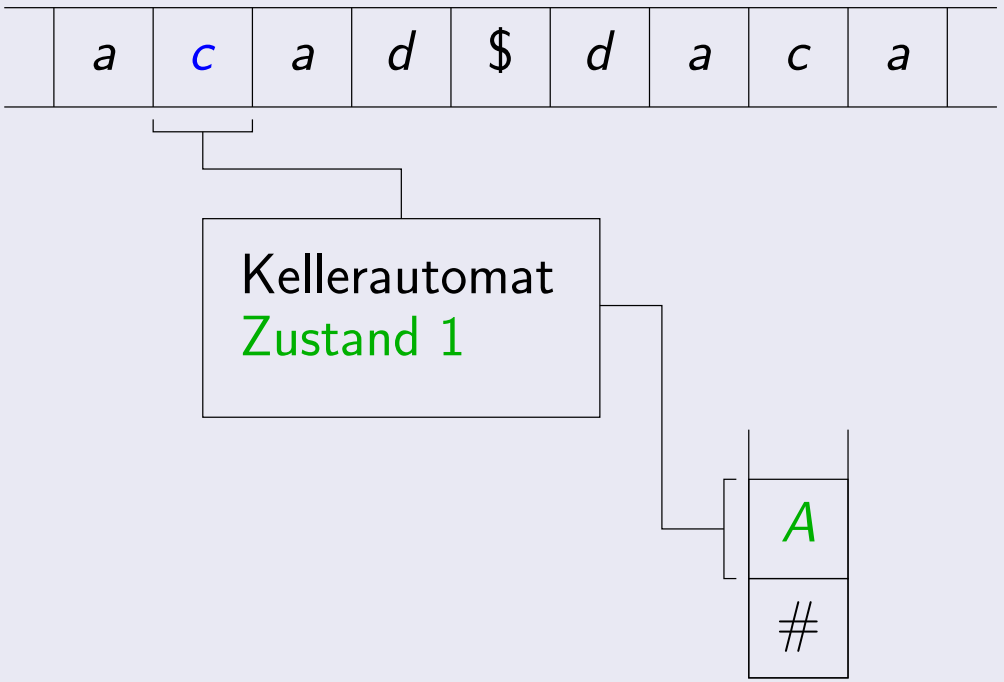
Ein Kellerautomat erkennt diese Sprache folgendermaßen:

- Ein Wort w wird von links nach rechts eingelesen.
- Solange \$ noch nicht erreicht ist, wird jedes eingelesene Symbol als Großbuchstabe auf den Keller gelegt ($a \rightsquigarrow A$, $b \rightsquigarrow B$, ...).
- Wenn \$ eingelesen wird, bleibt der Keller unverändert.
- Anschließend wird für jedes neu eingelesene Zeichen überprüft, ob der passende Großbuchstabe auf dem Keller liegt. Dieser wird dann entfernt.
- Falls irgendwann keine Übereinstimmung festgestellt wird, blockiert der Kellerautomat.
- Falls immer Übereinstimmung herrscht, wird schließlich auch das Kellerbodenzeichen # entfernt, und der Automat akzeptiert mit leerem Keller.

Simulation

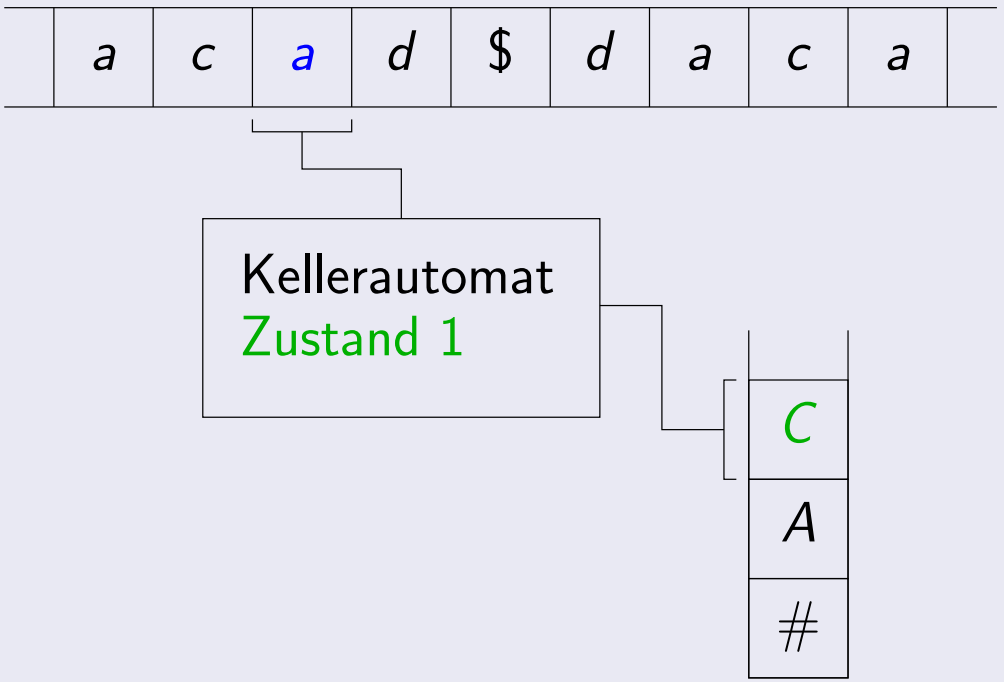


Simulation

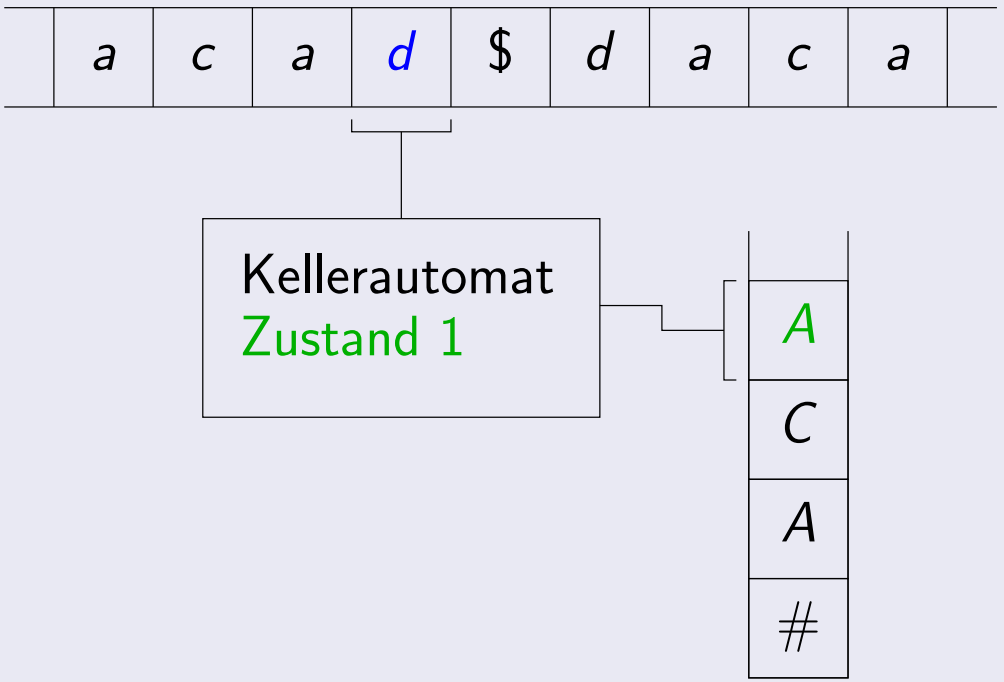


Kellerautomaten

Simulation

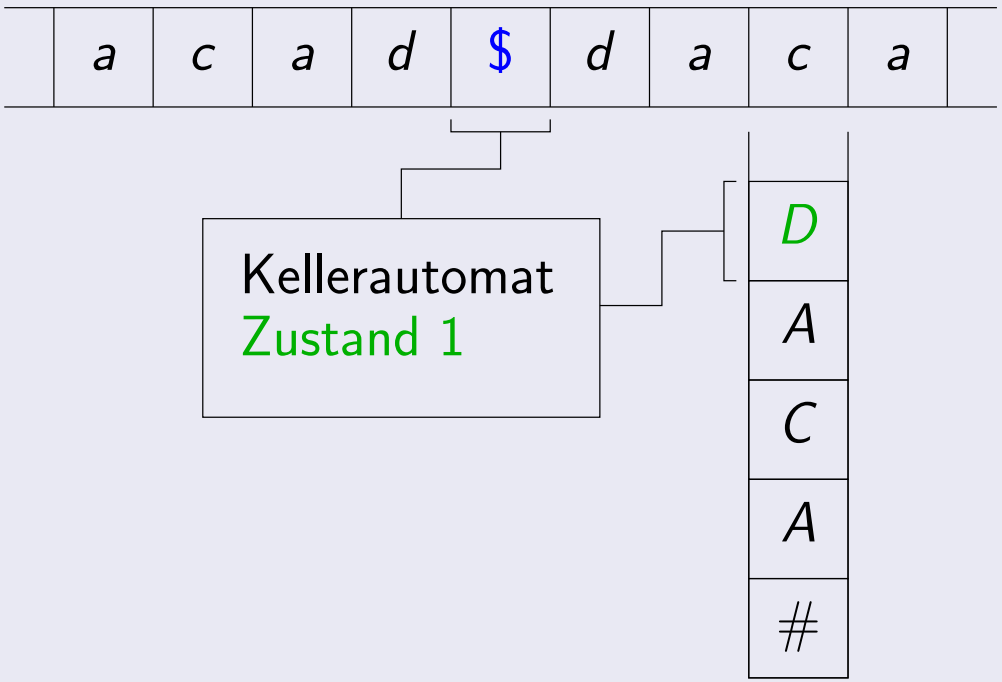


Simulation

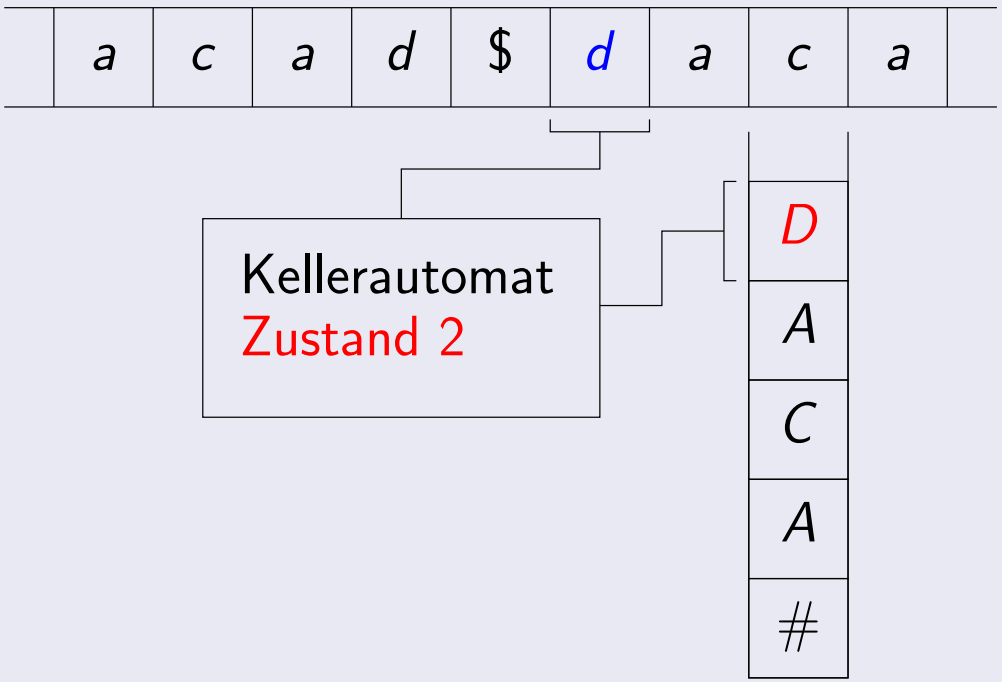


Kellerautomaten

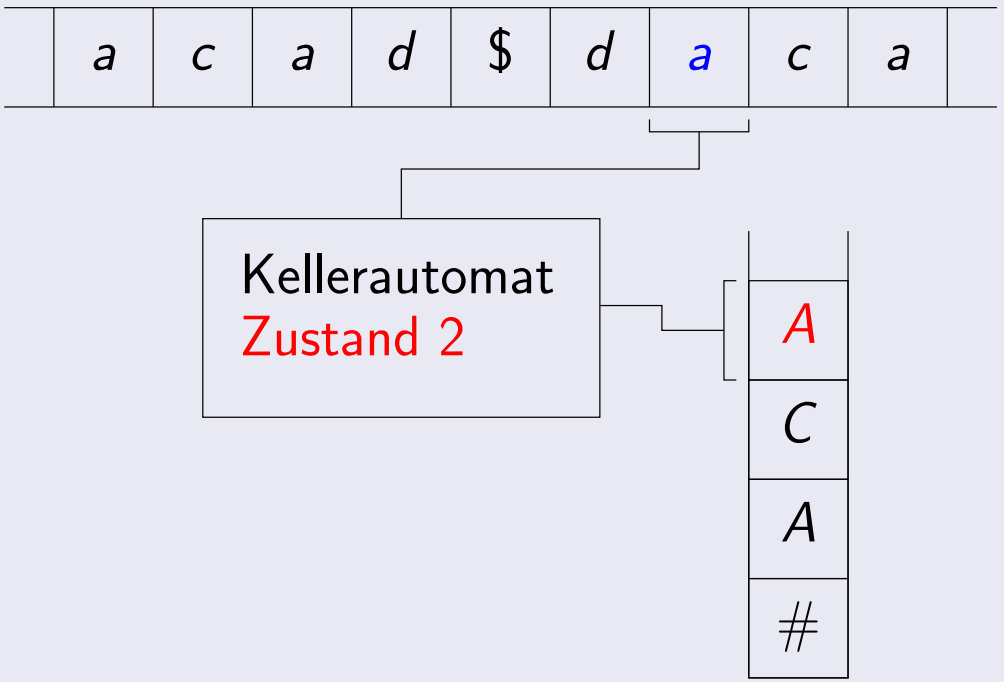
Simulation



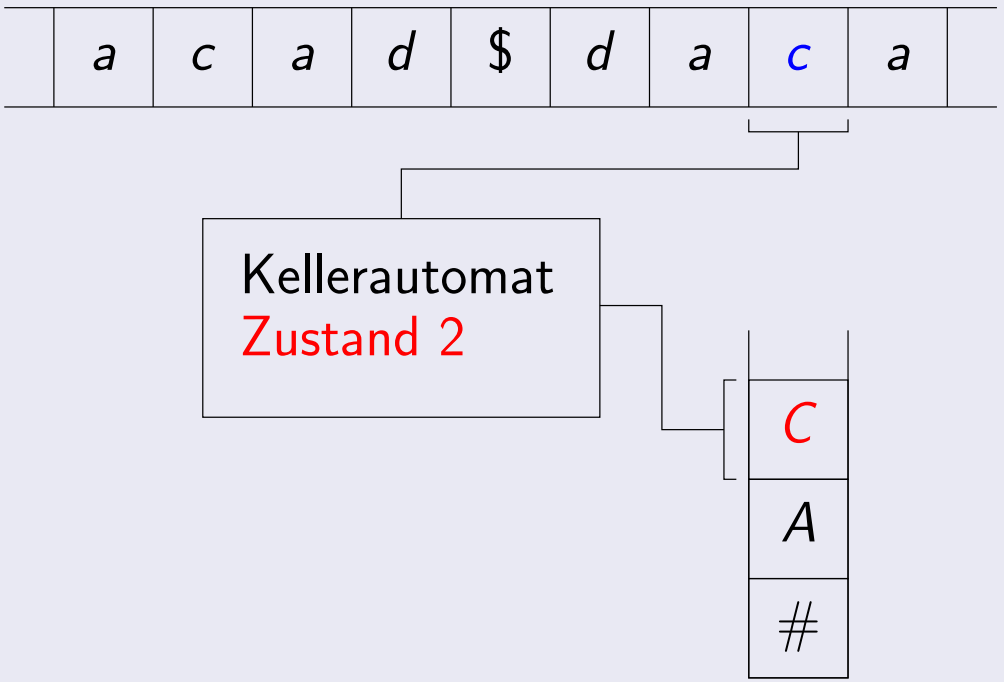
Simulation



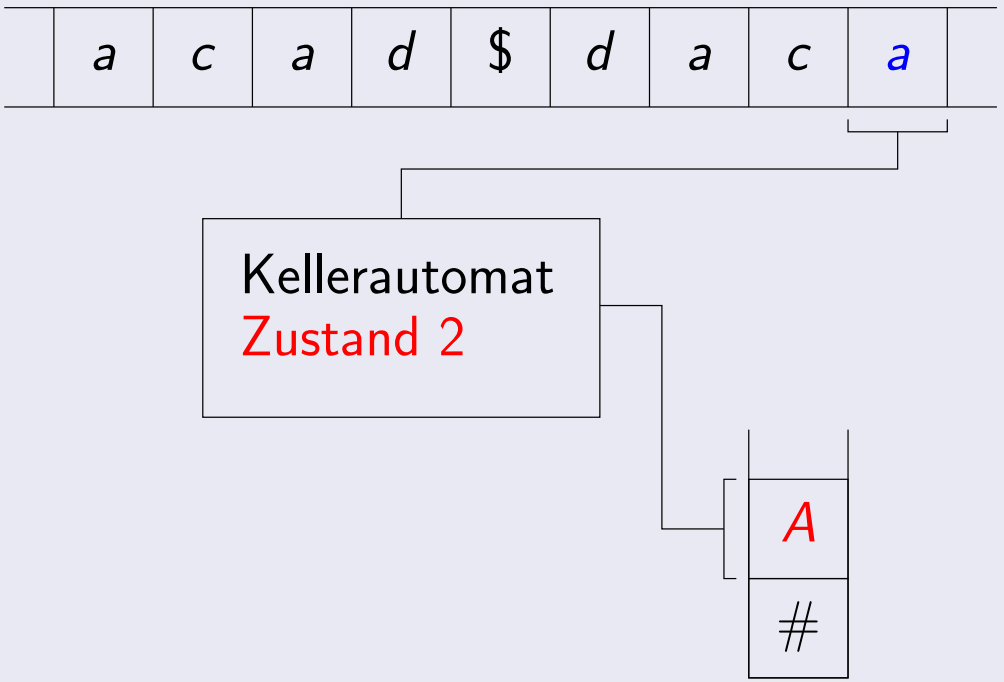
Simulation



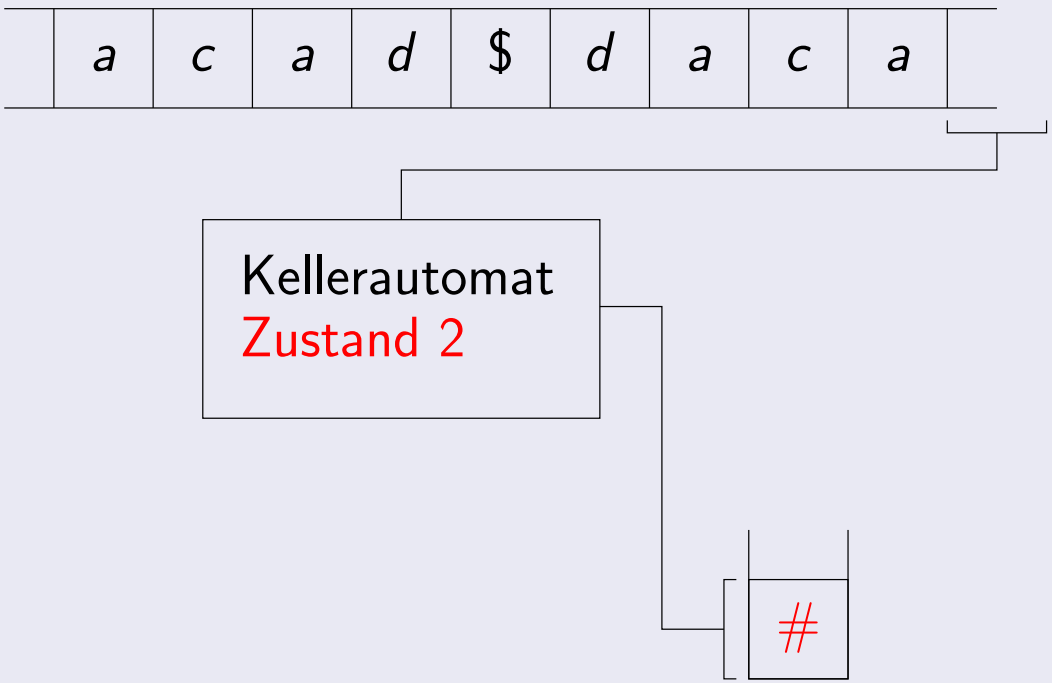
Simulation



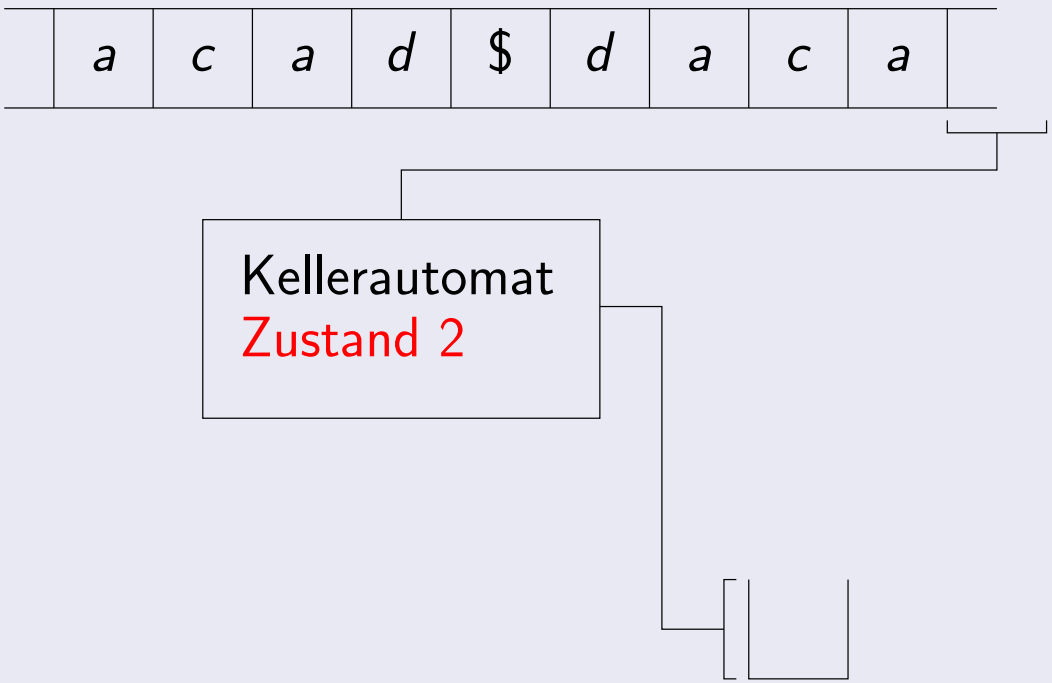
Simulation



Simulation



Simulation



Kellerautomaten

Definition (Kellerautomat)

Ein nichtdeterministischer Kellerautomat M ist ein 6-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$, wobei

- Z die endliche Menge der Zustände,
- Σ das endliche Eingabealphabet (mit $Z \cap \Sigma = \emptyset$),
- Γ das endliche Kelleralphabet,
- $z_0 \in Z$ der Startzustand,
- $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_e(Z \times \Gamma^*)$ die Überführungsfunktion und
- $\# \in \Gamma$ das unterste Kellerzeichen oder Kellerbodenzeichen ist.

Bemerkungen:

- $\mathcal{P}_e(Z \times \Gamma^*)$ bezeichnet die Menge aller endlichen Teilmengen von $Z \times \Gamma^*$.
- **Abkürzung:** KA oder PDA (pushdown automaton).

Kellerautomaten

- Wir betrachten die Überführungsfunktion

$$\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_e(Z \times \Gamma^*)$$

Falls $(z', B_1 \cdots B_k) \in \delta(z, a, A)$, so bedeutet das:

- wenn im Zustand z das Eingabesymbol a gelesen wird und das Zeichen A als oberstes auf dem Keller liegt, dann
- wird A vom Keller entfernt und durch $B_1 \cdots B_k$ ersetzt (B_1 liegt zuoberst) und der Automat geht in den Zustand z' über.

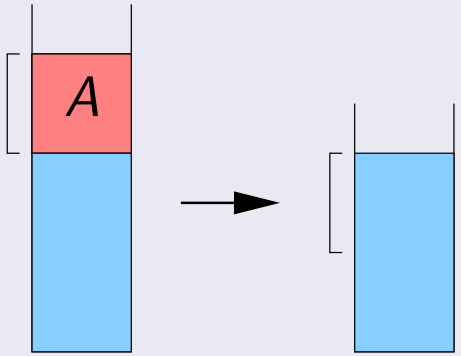
Es kann auch $a = \varepsilon$ gelten. In diesem Fall wird kein Eingabesymbol eingelesen.

Kellerautomaten

Wir betrachten verschiedene Fälle von Werten der Überführungsfunktion δ:

$(z', \varepsilon) \in \delta(z, a, A)$

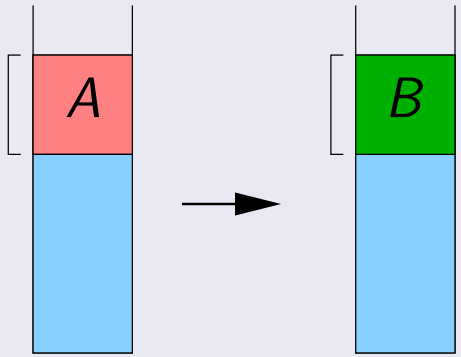
- Zeichen a wird gelesen.
- Zustand ändert sich von z nach z'.
- Symbol A wird vom Keller entfernt:



Kellerautomaten

$(z', B) \in \delta(z, a, A)$

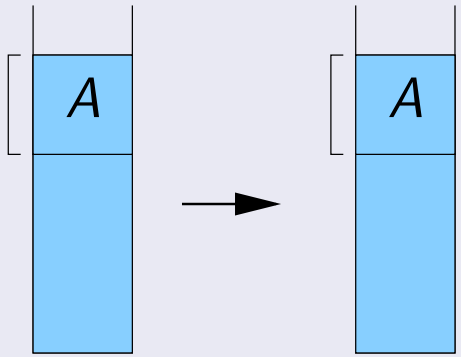
- Zeichen a wird gelesen.
- Zustand ändert sich von z nach z'.
- Symbol A auf dem Keller wird durch B ersetzt:



Kellerautomaten

$(z', A) \in \delta(z, a, A)$

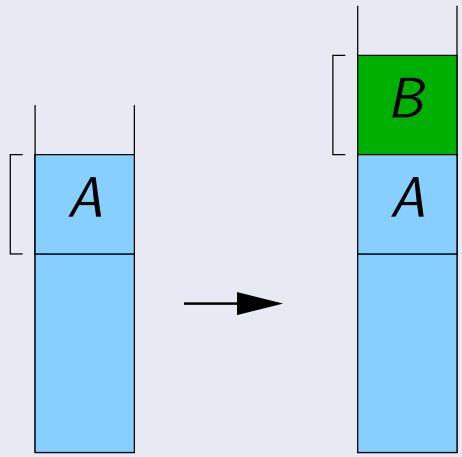
- Zeichen a wird gelesen.
- Zustand ändert sich von z nach z'.
- Symbol A bleibt auf dem Keller:



Kellerautomaten

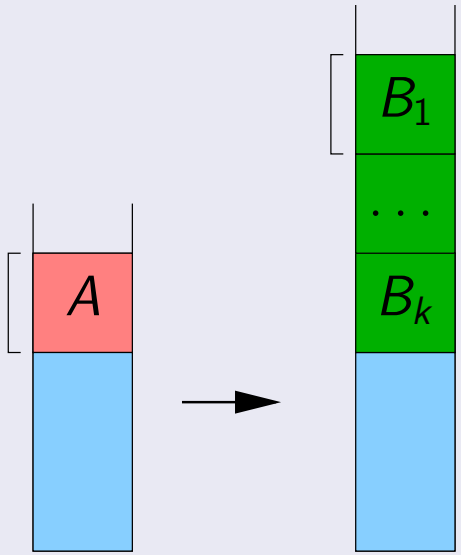
$(z', BA) \in \delta(z, a, A)$

- Zeichen a wird gelesen.
- Zustand ändert sich von z nach z'.
- Symbol B wird neu auf den Keller gelegt:



$(z', B_1 \cdots B_k) \in \delta(z, a, A)$

- Zeichen a wird gelesen.
- Zustand ändert sich von z nach z'.
- Symbol A wird durch mehrere neue Symbole ersetzt:



Kellerautomaten

- Zu Beginn einer jeden Berechnung enthält der Keller genau das Kellerbodenzeichen #.
- Der Keller ist nicht beschränkt und kann beliebig wachsen. Es gibt unendlich viele mögliche Kellerinhalte, das unterscheidet Kellerautomaten von endlichen Automaten.
- Die von uns betrachteten Kellerautomaten akzeptieren immer mit leerem Keller (in diesem Fall gibt es auch keine Übergangsmöglichkeiten mehr). Es gibt aber auch andere Varianten von Kellerautomaten, die mit Endzustand akzeptieren.

Beispiel:

PDA für $L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid n \geq 0, a_1, \ldots, a_n \in \{a, b\}\}$:

$$M = (\{z_1, z_2\}, \{a, b, \$\}, \{\#, A, B\}, \delta, z_1, \#),$$

wobei δ folgendermaßen definiert ist (wir schreiben $(z, a, A) \to (z', x)$, falls $(z', x) \in \delta(z, a, A)$).

$$\begin{array}{lll}
(z_1, a, \#) \to (z_1, A\#) & (z_1, a, A) \to (z_1, AA) & (z_1, a, B) \to (z_1, AB) \\
(z_1, b, \#) \to (z_1, B\#) & (z_1, b, A) \to (z_1, BA) & (z_1, b, B) \to (z_1, BB) \\
(z_1, \$, \#) \to (z_2, \#) & (z_1, \$, A) \to (z_2, A) & (z_1, \$, B) \to (z_2, B) \\
(z_2, a, A) \to (z_2, \varepsilon) & (z_2, b, B) \to (z_2, \varepsilon) & (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}$$

Definition (Konfiguration eines PDA)

Eine Konfiguration eines PDA ist ein Tripel $k \in Z \times \Sigma^* \times \Gamma^*$.

Bedeutung der Komponenten von $k = (z, w, \gamma) \in Z \times \Sigma^* \times \Gamma^*$:

- $z \in Z$ ist der aktuelle Zustand des PDA.
- $w \in \Sigma^*$ ist der noch zu lesende Teil der Eingabe.
- $\gamma \in \Gamma^*$ ist der aktuelle Kellerinhalt. Dabei steht das oberste Kellerzeichen ganz links.

Kellerautomaten

Übergänge zwischen Konfigurationen ergeben sich aus der Überführungsfunktion δ:

Definition (Konfigurationsübergänge eines PDA)

Es gilt

$$(z, aw, A\gamma) \vdash (z', w, B_1 \cdots B_k \gamma),$$

falls $(z', B_1 \cdots B_k) \in \delta(z, a, A)$, und es gilt

$$(z, w, A\gamma) \vdash (z', w, B_1 \cdots B_k \gamma),$$

falls $(z', B_1 \cdots B_k) \in \delta(z, \varepsilon, A)$.

Hierbei ist $\gamma \in \Gamma^*$ eine beliebige Folge von Kellersymbolen, $w \in \Sigma^*$, $a \in \Sigma$, und $z, z' \in Z$.

Im ersten Fall wird ein Zeichen der Eingabe gelesen, im zweiten jedoch nicht.

Kellerautomaten

Wir definieren $\vdash^*$ als die reflexive and transitive Hülle von $\vdash$.

Damit kann jetzt die von einem PDA akzeptierte Sprache definiert werden:

Definition (Akzeptierte Sprache eines PDA)

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ ein PDA. Dann ist die von M akzeptierte Sprache:

$$N(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash^* (z, \varepsilon, \varepsilon) \text{ für ein } z \in Z\}.$$

Das heißt die akzeptierte Sprache enthält alle Wörter, mit Hilfe derer es möglich ist, den Keller vollständig zu leeren.

Da Kellerautomaten jedoch nicht-deterministisch sind, kann es auch Berechnungen für dieses Wort geben, die den Keller nicht leeren.

Ein weiteres **Beispiel**: ein PDA für die Sprache

$$L = \{a_1 a_2 \cdots a_n a_n \cdots a_2 a_1 \mid n \geq 0, a_1, \ldots, a_n \in \{a, b\}\}.$$

Idee: anstatt auf das Zeichen \$ zu warten, kann sich der Automat nicht-deterministisch entscheiden, in den Zustand z_2 (= Keller abbauen) überzugehen, sobald das aktuelle Zeichen auf dem Band mit dem Zeichen auf dem Keller übereinstimmt (oder wenn der Keller leer ist).

Kellerautomaten

Veränderte Überführungsfunktion δ (3. Zeile ist geändert):

$$
\begin{array}{lll}
(z_1, a, \#) \to (z_1, A\#) & (z_1, a, A) \to (z_1, AA) & (z_1, a, B) \to (z_1, AB) \\
(z_1, b, \#) \to (z_1, B\#) & (z_1, b, A) \to (z_1, BA) & (z_1, b, B) \to (z_1, BB) \\
(z_1, \varepsilon, \#) \to (z_2, \#) & (z_1, a, A) \to (z_2, \varepsilon) & (z_1, b, B) \to (z_2, \varepsilon) \\
(z_2, a, A) \to (z_2, \varepsilon) & (z_2, b, B) \to (z_2, \varepsilon) & (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}
$$

Anmerkung: dieser Kellerautomat ist (im Gegensatz zum vorherigen) nicht-deterministisch, d.h., eine Konfiguration kann mehrere mögliche Nachfolger haben. (Und möglicherweise enden einige Konfigurationsfolgen als Sackgassen und führen nicht dazu, dass der Keller geleert wird.)

Beispiel: Kellerautomat erhält die Eingabe *aabbaa*.

Kellerautomaten

Wir müssen nun noch zeigen, dass man mit Kellerautomaten wirklich genau die kontextfreien Sprachen akzeptieren kann.

Satz (kontextfreie Grammatiken $\longrightarrow$ Kellerautomaten)

Zu jeder kontextfreien Grammatik G gibt es einen PDA M mit $L(G) = N(M)$.

Kellerautomaten

Beweisidee:

1. Wir können ohne Beschränkung der Allgemeinheit davon ausgehen, dass G in Greibach-Normalform ist.
2. Wir simulieren eine Ableitung von G, indem wir den Keller zur Abspeicherung derjenigen Variablen, für die noch etwas abgeleitet werden muss, verwenden.
3. Eine Produktion $A \longrightarrow aA_1 \cdots A_n$ wird wie folgt simuliert:

   Falls a das nächste Eingabesymbol ist und auf dem Keller oben A liegt, kann A durch $A_1 \cdots A_n$ ersetzt werden.
4. Wenn die komplette Eingabe gelesen wurde, und der Keller gleichzeitig leer ist, dann wurde eine komplette Ableitung für das Eingabewort simuliert.

Kellerautomaten

Formaler: Zunächst nehmen wir an, dass $\varepsilon \notin L(G)$.

Dann können wir o.B.d.A. davon ausgehen, dass $G = (V, \Sigma, P, S)$ in Greibach-Normalform ist.

Wir definieren den PDA

$$M = (\{z\}, \Sigma, V, \delta, z, S)$$

mit folgender Überführungsfunktion: Für $A \in V$ und $a \in \Sigma$ sei

$$\delta(z, a, A) = \{(z, A_1 \cdots A_m) \mid (A \to aA_1 \cdots A_m) \in P\}$$

Beachte:

- M hat nur einen Zustand (z).
- M hat keine ε-Transitionen.
- Das Startsymbol S von G ist das Kellerbodenzeichen.
- Da G in Greibach-Normalform ist, sind alle Produktionen in P von der Form $A \to aA_1 \cdots A_m$ mit $m \geq 0$, $A, A_1, \ldots, A_m \in V$ und $a \in \Sigma$.

Falls $\varepsilon \in L(G)$ gilt, können wir o.B.d.A. davon ausgehen, dass bis auf die Produktion $S \to \varepsilon$ alle Produktionen von G in Greibach-Normalform sind.

Wir fügen dann zu dem auf der vorherigen Folie definierten PDA noch die Transition $\delta(z, \varepsilon, S) = (z, \varepsilon)$ (die einzige ε-Transition) hinzu.

Dann gilt wie gewünscht $L(G) = N(M)$. □

Alternative Konstruktion:

Wir können auch direkt aus einer beliebigen kontextfreien Grammatik $G = (V, \Sigma, P, S)$ einen PDA M mit $L(G) = N(M)$ konstruieren.

Definiere den PDA $M = (\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$ mit einem Zustand z und Kelleralphabet $V \cup \Sigma$.

Überführungsfunktion δ:

$$\delta(z, \varepsilon, A) = \{(z, \alpha) \mid (A \to \alpha) \in P\} \text{ für } A \in V$$
$$\delta(z, a, a) = \{(z, \varepsilon)\} \text{ für } a \in \Sigma$$

Mit Produktionen vom ersten Typ werden Ableitungsschritt auf dem Keller ohne Lesen der Eingabe simuliert.

Mit Produktionen vom zweiten Typ wird ein Symbol der Eingabe mit dem Keller verglichen.

Beachte: M hat ε-Produktionen.

Kellerautomaten

Wir betrachten folgende kontextfreie Grammatik mit dem zweielementigen Alphabet $\Sigma = \{[,]\}$, die korrekte Klammerstrukturen erzeugt:

$$S \to [S]S \mid \varepsilon$$

Aufgabe: wandle diese Grammatik in einen Kellerautomaten um und akzeptiere damit das Wort $[\,[\,]\,]\,[\,]$.

Kellerautomaten

Nun geht es darum zu zeigen, dass es zu jedem Kellerautomaten eine entsprechende kontextfreie Grammatik gibt.

Das ist die schwierigere Richtung.

Satz (Kellerautomaten $\longrightarrow$ kontextfreie Grammatiken)

Zu jedem Kellerautomaten M gibt es eine kontextfreie Grammatik G mit $N(M) = L(G)$.

Beweisidee:

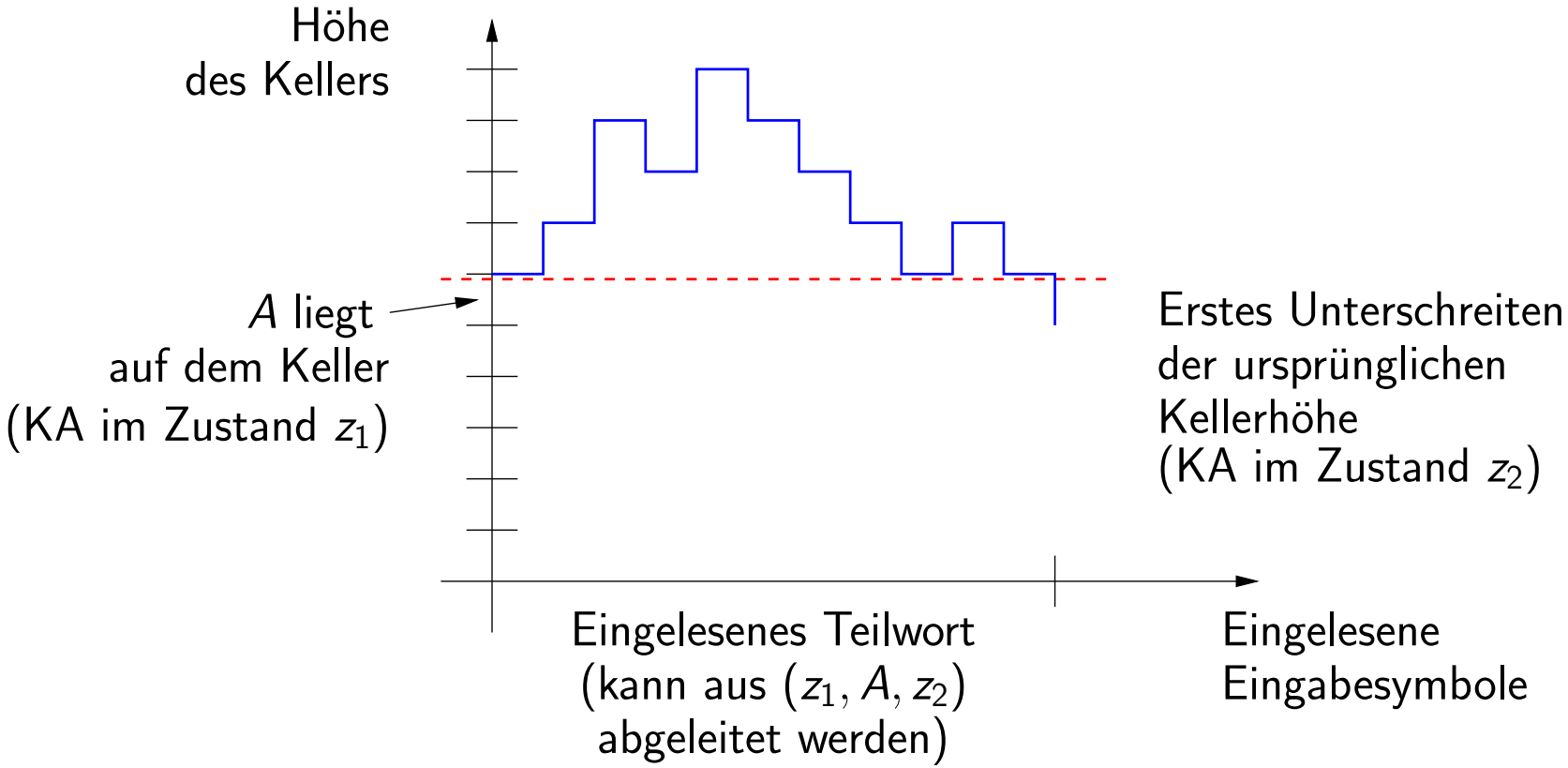
1. Wir wollen beschreiben, welche Wörter man durch Abbauen eines bestimmten Kellersymbols akzeptieren kann. Die vom Automaten akzeptierte Sprache besteht nämlich aus allen Wörtern, die man durch Abbauen von # erzeugen kann.

   "Abbauen" bedeutet: zwischendurch dürfen weitere Symbole auf den Keller gelegt werden, aber zuletzt muss der Keller um dieses eine Symbol kürzer geworden sein.

2. Die zu erstellende kontextfreie Grammatik besitzt Variablen der Form (z_1, A, z_2) mit der Bedeutung:

   Aus (z_1, A, z_2) kann man genau die Wörter ableiten, die der Kellerautomat einliest, wenn er im Zustand z_1 startet, A vom Keller abbaut und im Zustand z_2 aufhört.

Kellerautomaten



Zwischendurch kann A durch ein anderes Symbol ersetzt werden. Die ursprüngliche Kellerhöhe wird jedoch nicht unterschritten.

Formale Bedeutung der Symbole (z_1, A, z_2):

$$(z_1, A, z_2) \Rightarrow^* x \quad \Longleftrightarrow \quad (z_1, x, A) \vdash^* (z_2, \varepsilon, \varepsilon)$$

Gegeben sei ein Kellerautomat $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$.

Wir definieren eine Grammatik $G = (V, \Sigma, P, S)$ wie folgt (siehe nächste Folie):

Kellerautomaten

- Variablen: $V = \{S\} \cup Z \times \Gamma \times Z$
  (Eigene Startvariable und Variablen der Form (z_1, A, z_2))
- Produktionen haben folgende Form:

$$S \rightarrow (z_0, \#, z) \qquad \text{für alle } z \in Z$$

(Entfernen des Kellerbodenzeichens)

$$(z, A, z') \rightarrow a \qquad \text{falls } (z', \varepsilon) \in \delta(z, a, A)$$

(Symbol A kann – bei Einlesen von a – sofort entfernt werden)

$$(z, A, z') \rightarrow a\,(z_1, B_1, z_2)(z_2, B_2, z_3) \cdots (z_k, B_k, z')$$

für alle $(z_1, B_1 \cdots B_k) \in \delta(z, a, A)$, $z_2, \ldots, z_k \in Z$

(Symbol A wird durch $B_1 \ldots B_k$ ersetzt, diese müssen über Zwischenzustände $z_1, \ldots, z_k$ entfernt werden.)

Kellerautomaten

Beispiel: Wir betrachten den Kellerautomaten

$$M = (\{z_1, z_2\}, \{a, b\}, \{A, \#\}, \delta, z_1, \#)$$

mit folgender Überführungsfunktion δ:

$$\begin{array}{lcl} (z_1, \varepsilon, \#) & \to & (z_2, \varepsilon) \\ (z_1, a, \#) & \to & (z_1, AA) \\ (z_1, a, A) & \to & (z_1, AAA) \\ (z_1, b, A) & \to & (z_2, \varepsilon) \\ (z_2, b, A) & \to & (z_2, \varepsilon) \end{array}$$

Es gilt: $N(M) = \{a^n b^{2n} \mid n \geq 0\}$.

Aufgabe: Umwandlung von M in eine kontextfreie Grammatik.

Kellerautomaten

Bemerkung zu den Umwandlungen
"Kontextfreie Grammatik $\leftrightarrow$ Kellerautomat":

Zu jedem Kellerautomaten M gibt es immer einen äquivalenten Kellerautomaten M' mit nur einem Zustand und ohne ε-Transitionen (falls $\varepsilon \notin N(M)$).

1. Wandele M zunächst in eine kontextfreie Grammatik G um.
2. Wandle dann G in eine kontextfreie Grammatik G' in Greibach-Normalform um.
3. Wandle schließlich G' in einen Kellerautomaten M' um.

Es wird ausgenutzt, dass bei der Umwandlung einer Grammatik (in Greibach-Normalform) in einen Kellerautomaten immer nur Automaten mit einem Zustand und ohne ε-Transitionen konstruiert werden.

Deterministisch kontextfreie Sprachen

Wir betrachten nun eine Unterklasse von Kellerautomaten, die dazu verwendet werden können, Sprachen deterministisch und damit effizient zu erkennen.

Definition (deterministischer Kellerautomat)

Ein deterministischer Kellerautomat M ist ein 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$, wobei

- $(Z, \Sigma, \Gamma, \delta, z_0, \#)$ ein Kellerautomat ist,
- $E \subseteq Z$ eine Menge von Endzuständen ist, und
- die Überführungsfunktion $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_e(Z \times \Gamma^*)$ in folgendem Sinne deterministisch ist:

  Für alle $z \in Z$, $a \in \Sigma$ und $A \in \Gamma$ gilt:

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

Deterministisch kontextfreie Sprachen

Unterschiede zwischen Kellerautomaten und deterministischen Kellerautomaten:

- Deterministische Kellerautomaten haben eine Menge von Endzuständen und akzeptieren mit Endzustand – und nicht mit leerem Keller.

  Bei deterministischen Kellerautomaten ist dies ein Unterschied, für nicht-deterministische Kellerautomaten sind beide Akzeptanzmöglichkeiten gleichwertig (siehe Folie 232).

- Für jeden Zustand z und jedes Kellersymbol A gilt:
  - entweder gibt es höchstens einen ε-Übergang
  - oder es gibt für jedes Alphabetsymbol höchstens einen Übergang.

Deterministisch kontextfreie Sprachen

Konfigurationen und Übergänge zwischen Konfiguration bleiben jedoch gleich definiert.

Konfigurationsfolgen werden jedoch zu linearen Ketten, d.h., es gibt immer höchstens eine Folgekonfiguration.

Diese Tatsache wird dann für die effiziente Lösung des Wortproblems ausgenutzt.

Deterministisch kontextfreie Sprachen

Definition (akzeptierte Sprache bei det. Kellerautomaten)

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$ ein deterministischer PDA. Dann ist die von M akzeptierte Sprache:

$$D(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash^* (z, \varepsilon, \gamma) \text{ für ein } z \in E, \gamma \in \Gamma^*\}.$$

Vergleiche mit der Definition für nicht-deterministische Kellerautomaten!

Bei deterministischen Kellerautomaten ist folgendes anders:

- Der erreichte Zustand z muss ein Endzustand sein.
- Es darf ein Kellerinhalt γ übrigbleiben.

Deterministisch kontextfreie Sprachen

Definition (deterministisch kontextfreie Sprachen)

Eine Sprache heißt deterministisch kontextfrei genau dann, wenn sie von einem deterministischen PDA akzeptiert wird.

Beispiele:

- Die Sprache $L = \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \Delta\}$ ist deterministisch kontextfrei (siehe den entsprechenden PDA).
- Die Sprache $L = \{a_1 a_2 \dots a_n a_n \dots a_2 a_1 \mid a_i \in \Delta\}$ ist jedoch nicht deterministisch kontextfrei (ohne Beweis).

Deterministisch kontextfreie Sprachen

Beachte: A priori folgt aus der Definition von deterministisch kontextfreien Sprachen nicht sofort, dass deterministisch kontextfreie Sprachen auch kontextfrei sind (Akzeptanz mit Endzuständen versus leeren Keller).

Dies ist aber der Fall: Aus einem deterministischen PDA $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$ konstruieren wir einen (nicht-deterministischen) PDA $M' = (Z \cup \{z_0', z_f\}, \Sigma, \Gamma \cup \{\#'\}, \delta', z_0', \#')$, wobei gilt:

$$\delta'(z_0', \varepsilon, \#') = \{(z_0, \#\#')\}$$

$$\delta'(z, a, A) = \begin{cases} \delta(z, a, A) & \text{falls } (z \in Z \setminus E \text{ oder } a \in \Sigma),\ A \in \Gamma \\ \delta(z, a, A) \cup \{(z_f, \varepsilon)\} & \text{falls } z \in E, a = \varepsilon, A \in \Gamma \end{cases}$$

$$\delta(z, \varepsilon, \#') = \{(z_f, \varepsilon)\} \text{ falls } z \in E$$

$$\delta(z_f, \varepsilon, A) = \{(z_f, \varepsilon)\} \text{ falls } A \in \Gamma \cup \{\#'\}$$

Dann gilt: $N(M') = D(M)$.

Deterministisch kontextfreie Sprachen

Die Konstruktion auf der vorherigen Folie zeigt auch, wie man einen (nicht-deterministischen) PDA, der mit Endzuständen akzeptiert, in einen (nicht-deterministischen) PDA, der mit leeren Keller akzeptiert, umwandelt.

Umgekehrt kann man einen (nicht-deterministischen) PDA $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$, der mit leeren Keller akzeptiert, in einen (nicht-deterministischen) PDA, der mit Endzuständen akzeptiert, wie folgt umwandeln:

Sei $M' = (Z \cup \{z_0', z_f\}, \Sigma, \Gamma \cup \{\#'\}, \delta', z_0', \#', \{z_f\})$, wobei gilt:

$$\delta'(z_0', \varepsilon, \#') = \{(z_0, \#\#')\}$$
$$\delta'(z, a, A) = \delta(z, a, A) \text{ falls } z \in Z,\ a \in \Sigma \cup \{\varepsilon\},\ A \in \Gamma$$
$$\delta(z, \varepsilon, \#') = \{(z_f, \varepsilon)\} \text{ für alle } z \in Z$$

Dann gilt: $N(M') = N(M)$.

Deterministisch kontextfreie Sprachen

Weitere Bemerkungen:

- Effizienz: Mit Hilfe von deterministischen Kellerautomaten hat man jetzt ein Verfahren zur Lösung des Wortproblems, das die Komplexität $O(n)$ hat. Hierbei ist n die Länge des Eingabewortes.

  Dazu lässt man einfach den Automaten auf dem Wort arbeiten und überprüft ob man in einen Endzustand gelangt.

- Deterministisch kontextfreie Grammatiken: Da die Syntax von Sprachen einfacher mit Hilfe von Grammatiken als mit Hilfe von Kellerautomaten definiert werden kann, ist es notwendig, die zu deterministischen Kellerautomaten passende Klasse von deterministisch kontextfreien Grammatiken zu definieren.

  Da dies nicht ganz einfach ist, gibt es hierzu mehrere Ansätze. Der bekannteste davon sind die sogenannten $LR(k)$-Grammatiken (siehe Compilerbau und Syntaxanalyse).

Deterministisch kontextfreie Sprachen

Die Abschlusseigenschaften bei deterministisch kontextfreien Sprachen sehen etwas anders aus als bei kontextfreien Sprachen.

Satz (Abschluss unter Komplement)

Wenn L eine deterministisch kontextfreie Sprache ist, dann ist auch $\overline{L} = \Sigma^* \backslash L$ deterministisch kontextfrei.

Wir verzichten hier auf den recht technischen Beweis.

Deterministisch kontextfreie Sprachen

Kein Abschluss unter Schnitt

Es gibt deterministisch kontextfreie Sprachen L_1 und L_2, so dass $L_1 \cap L_2$ nicht deterministisch kontextfrei ist.

Begründung:

Die Beispiel-Sprachen aus dem Argument, dass die kontextfreien Sprachen unter Schnitt nicht abgeschlossen sind, sind sogar deterministisch kontextfrei, ihr Schnitt jedoch noch nicht einmal kontextfrei:

$$\begin{aligned} L_1 &= \{a^j b^k c^k \mid j \geq 0, k \geq 0\} \\ L_2 &= \{a^k b^k c^j \mid j \geq 0, k \geq 0\} \end{aligned}$$

Deterministisch kontextfreie Sprachen

Kein Abschluss unter Vereinigung

Es gibt deterministisch kontextfreie Sprachen L_1 und L_2, so dass $L_1 \cup L_2$ nicht deterministisch kontextfrei ist.

Begründung:

Aus dem Abschluss unter Vereinigung und Komplement würde auch der Abschluss unter Schnitt folgen (wegen $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$).

Deterministisch kontextfreie Sprachen

Man hat jedoch sehr wohl Abschluss unter Schnitt mit regulären Sprachen:

Satz (Abschluss unter Schnitt mit regulären Sprachen)

Sei L eine deterministisch kontextfreie Sprache und R eine reguläre Sprache. Dann ist $L \cap R$ eine deterministisch kontextfreie Sprache.

Beweisidee: (analog der Kreuzprodukt-Konstruktion für NFAs)

Sei $M = (Z_1, \Sigma, \Gamma, \delta_1, z_0^1, \#, E_1)$ ein deterministischer PDA für L.

Sei $A = (Z_2, \Sigma, \delta_2, z_0^2, E_2)$ ein DFA für R.

Konstruktion eines deterministischen PDA M' für $L \cap R$:

$$M' = (Z_1 \times Z_2, \Sigma, \Gamma, \delta', (z_0^1, z_0^2), \#, E_1 \times E_2).$$

Hierbei ist die Überführungsfunktion δ wie folgt definiert ist:

Deterministisch kontextfreie Sprachen

$$\begin{aligned}
\delta((z_1, z_2), a, A) &= \{((z_1', z_2'), B_1 \cdots B_k) \mid (z_1', B_1 \cdots B_k) \in \delta_1(z_1, a, A), \\
&\qquad\qquad \delta_2(z_2, a) = z_2', a \in \Sigma\} \\
\delta((z_1, z_2), \varepsilon, A) &= \{((z_1', z_2), B_1 \cdots B_k) \mid (z_1', B_1 \cdots B_k) \in \delta_1(z_1, \varepsilon, A)\}
\end{aligned}$$

Beachte: Die so definierte Überführungsfunktion erfüllt die in der Definition von deterministischen PDAs gestellten Voraussetzungen. □

Nochmal Abschlusseigenschaften

Mit der gleichen Technik und unter Ausnutzung der Tatsache, dass für allgemeine (nicht-deterministische) Kellerautomaten die Akzeptanz mit leerem Keller analog zur Akzeptanz mit Endzustand ist (siehe Folie 218), lässt sich auch folgendes zeigen:

Satz (Abschluss unter Schnitt mit regulären Sprachen II)

Sei L eine kontextfreie Sprache und R eine reguläre Sprache. Dann ist $L \cap R$ eine kontextfreie Sprache.

Entscheidbarkeit

Wir betrachten nun noch Probleme für kontextfreie Sprachen und stellen fest, ob sie entscheidbar sind, d.h., ob es entsprechende Verfahren zu ihrer Lösung gibt.

Wortproblem für eine kontextfreie Sprache L

Gegeben $w \in \Sigma^*$.
Gilt $w \in L$?

Ist die kontextfreie Sprache L durch eine kontextfreie Grammatik in Chomsky-Normalform gegeben, so kann das Wortproblem mit dem CYK-Algorithmus in $O(|w|^3)$ Zeit gelöst werden.

Ist L deterministisch kontextfrei und durch einen deterministischen PDA gegeben, so kann das Wortproblem für L sogar in Zeit $O(n)$ gelöst werden.

Leerheitsproblem für kontextfreie Sprachen

Gegeben eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$.
Gilt $L(G) = \emptyset$?

Bestimme die Menge

$$W = \{A \in V \mid \exists w \in \Sigma^* : A \Rightarrow_G^* w\}$$

aller produktiven Variablen:

$W := \{A \in V \mid \exists w \in \Sigma^* : (A \to w) \in P\}$
$W' := \emptyset$
while $W' \neq W$ do
  $W' := W$
  $W := W \cup \{A \in V \mid \exists w \in (\Sigma \cup W)^* : (A \to w) \in P\}$
endwhile

Dann gilt: $L(G) \neq \emptyset \iff S \in W$.

Endlichkeitsproblem für kontextfreie Sprachen

Gegeben eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$.
Ist $L(G)$ endlich?

Ohne Beschränkung der Allgemeinheit können wir davon ausgehen, dass G in Chomsky-Normalform ist.

Wir definieren einen Graphen (W, E) auf der Menge der produktiven Variablen (siehe vorherige Folie) mit folgender Kantenrelation:

$$E = \{(A, B) \in W \times W \mid \exists C \in W : (A \to BC) \in P \text{ oder } (A \to CB) \in P\}$$

Behauptung: $|L(G)| = \infty \iff \exists A \in W : (S, A) \in E^*$ und $(A, A) \in E^+$.
Beachte: $(B, C) \in E^*$ (bzw. $(B, C) \in E^+$) bedeutet hierbei, dass es einen Pfad (bzw. einen nicht leeren Pfad) von B nach C in der Relation E gibt.

Entscheidbarkeit

"$\Leftarrow$": Sei $A \in W$ so, dass $(S, A) \in E^*$ und $(A, A) \in E^+$.

Dann existieren in G Ableitungen der Form:

$$S \Rightarrow_G^* uAy, \; A \Rightarrow_G^+ vAx, \; A \Rightarrow_G^* w$$

mit $u, v, w, x, y \in \Sigma^*$.

$\rightsquigarrow$ $S \Rightarrow_G^* uv^i wx^i y \in \Sigma^*$ für alle $i \geq 0$.

Da ausserdem in der Ableitung $A \Rightarrow_G^+ vAx$ mindestens ein Ableitungsschritt gemacht wird, und G in Chomsky-Normalform ist, muss $vx \neq \varepsilon$ gelten.

$\rightsquigarrow$ $\{uv^i wx^i y \mid i \geq 0\}$ ist unendlich.

Entscheidbarkeit

"$\Rightarrow$": Sei $L(G)$ unendlich.

Sei n die Konstante aus dem Pumping-Lemma ($= 2^{|V|}$) und sei $z \in L(G)$ mit $|z| \geq n$.

Im Beweis des Pumping-Lemmas haben wir gesehen, dass eine Variable A existiert mit Ableitungen $S \Rightarrow_G^* uAy$, $A \Rightarrow_G^* vAx$, und $A \Rightarrow_G^* w$, wobei $z = uvwxy$.

$\rightsquigarrow \quad A \in W$

Die Ableitungen $S \Rightarrow^* uAy$ und $A \Rightarrow^* vAx$ (genauer, der Pfad im Syntaxbaum von der Wurzel S bis zum zweiten Vorkommen von A) zeigen, dass $(S, A) \in E^*$ und $(A, A) \in E^+$ gilt. $\square$

Beispiel:

Sei G die Grammatik in Chomsky-Normalform mit den folgenden Produktionen:

$$\begin{aligned} S &\to AC \\ A &\to BC \\ B &\to CA \mid b \\ C &\to a \end{aligned}$$

In diesem Fall ist $W = \{S, A, B, C\}$, d.h. alle Variablen sind produktiv: Nach Durchlauf i durch die **while-Schleife** (Folie 253) erhalten wir

1. für $i = 0$: $W = \{B, C\}$
2. für $i = 1$: $W = \{A, B, C\}$
3. für $i = 2$: $W = \{S, A, B, C\}$

Also gilt $L(G) \neq \emptyset$.

Beispiel (Fortsetzung):

Die Kantenrelation E ist

$$E = \{(S, A), (S, C), (A, B), (A, C), (B, C), (B, A)\}.$$

Es gilt z.B. $S \xrightarrow{E} A \xrightarrow{E} B \xrightarrow{E} A$.

Also ist $L(G)$ unendlich.

Unentscheidbarkeit bei kontextfreien Sprachen

Folgende Probleme sind für kontextfreie Sprachen nicht entscheidbar, d.h. man kann zeigen, dass es kein entsprechendes Verfahren gibt:

- Äquivalenzproblem: Gegeben zwei kontextfreie Sprachen L_1, L_2. Gilt $L_1 = L_2$?
- Schnittproblem: Gegeben zwei kontextfreie Sprachen L_1, L_2. Gilt $L_1 \cap L_2 = \emptyset$?

Bemerkung: Im weiteren Verlauf der Vorlesung wird es darum gehen, wie man solche Unentscheidbarkeitsresultate zeigen kann.

Entscheidbarkeit

Das Schnittproblem ist jedoch entscheidbar, wenn von einer der beiden Sprachen L_1, L_2 bekannt ist, dass sie regulär ist, und sie als endlicher Automat gegeben ist.

Entscheidungsverfahren:

1. In diesem Fall kann ein Kellerautomat M konstruiert werden (Konstruktion siehe weiter oben), der $L_1 \cap L_2$ akzeptiert.
2. Der Kellerautomat M kann dann in eine kontextfreie Grammatik G umgewandelt werden.
3. Durch Bestimmung der produktiven Variablen von G kann dann ermittelt werden, ob S nicht-produktiv ist und damit, ob $L_1 \cap L_2$ leer ist.

Entscheidbarkeit

Entscheidbarkeit bei deterministisch kontextfreien Sprachen

Folgende Probleme sind für deterministisch kontextfreie Sprachen (repräsentiert durch einen deterministischen Kellerautomaten) entscheidbar:

- Wortproblem für eine deterministisch kontextfreie Sprache L: Gegeben $w \in \Sigma^*$. Gilt $w \in L$?

  Mit einem deterministischen Kellerautomaten in $O(|w|)$ Zeit.

- Leerheitsproblem: Gegeben eine deterministisch kontextfreie Sprache L. Gilt $L = \emptyset$?

  Siehe das entsprechende Entscheidungsverfahren für kontextfreie Sprachen.

Entscheidbarkeit

Entscheidbarkeit bei deterministisch kontextfreien Sprachen

- Endlichkeitsproblem: Gegeben eine kontextfreie Sprache L. Ist L endlich?

  Siehe das entsprechende Entscheidungsverfahren für kontextfreie Sprachen.

- Äquivalenzproblem: Gegeben zwei deterministisch kontextfreie Sprachen L_1, L_2. Gilt $L_1 = L_2$?

  War lange offen und die Entscheidbarkeit wurde erst 1997 von Géraud Sénizergues gezeigt.

Unentscheidbarkeit bei deterministisch kontextfreien Sprachen

Folgende Probleme sind für deterministisch kontextfreie Sprachen nicht entscheidbar, d.h. man kann zeigen, dass es kein entsprechendes Verfahren gibt:

- Schnittproblem: Gegeben zwei deterministisch kontextfreie Sprachen L_1, L_2. Gilt $L_1 \cap L_2 = \emptyset$?

  Wie bei kontextfreien Sprachen ist dieses Problem jedoch entscheidbar, wenn eine der beiden Sprachen regulär ist.
- Inklusionsproblem: Gegeben zwei deterministisch kontextfreie Sprachen L_1, L_2. Gilt $L_1 \subseteq L_2$?

Bisheriger Inhalt

Bisheriger Inhalt in der Vorlesung **GTI**:
die untersten beiden Stufen der Chomsky-Hierarchie

Reguläre Sprachen (Chomsky Typ 3)

reguläre Grammatiken, (deterministische und nichtdeterministische) endliche Automaten, reguläre Ausdrücke, Pumping-Lemma, Minimalautomat, Abschlusseigenschaften, Entscheidbarkeitsresultate

Kontextfreie Sprachen (Chomsky Typ 2)

kontextfreie Grammatiken, Normalformen, Pumping-Lemma, Abschlusseigenschaften, CYK-Algorithmus, Kellerautomaten, deterministisch kontextfreie Sprachen, Entscheidbarkeitsresultate

Zukünftiger Inhalt

Und was kommt jetzt?

- Wir beschäftigen uns mit den verbleibenden beiden Stufen der Chomsky-Hierarchie: Chomsky Typ 1 und Chomsky Typ 0.
- Die Theorie der Typ-0-Sprachen ist im wesentlichen Berechenbarkeitstheorie: Welche Sprachen sind überhaupt mit (informatischen) Maschinen akzeptierbar? Welche Funktionen sind überhaupt berechenbar?

  Bei Berechenbarkeitstheorie: Fokus liegt etwas mehr auf berechenbaren Funktionen
- Und als letztes Kapitel: Komplexitätstheorie

  Welche Sprachen kann man akzeptieren, wenn die Resourcen (Zeit, Platz) beschränkt sind? Beispielsweise: was kann man alles in polynomieller Zeit erreichen?

Chomsky-Hierarchie

Grammatik (Wiederholung)

Eine Grammatik G ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt:

- V ist eine endliche Menge von Nicht-Terminalen bzw. Variablen
- Σ ist das (endliche) Alphabet bzw. die Menge der Terminal(symbol)e.
- P ist eine endliche Menge von Regeln bzw. Produktionen mit $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$.
- $S \in V$ ist die Startvariable bzw. das Axiom.

Die von einer Grammatik erzeugte Sprache (Wiederholung)

Die von einer Grammatik $G = (V, \Sigma, S, P)$ erzeugte Sprache ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}.$$

(Menge aller Wörter aus Alphabetsymbolen, die aus der Startvariable S ableitbar sind.)

Beispiel: Die Grammatik $G = (\{S, A, B\}, \{a, b\}, S, P)$ mit der Produktionenmenge

$$P = \{S \to ABS \mid \varepsilon, \quad AB \to BA, \quad BA \to AB, \quad A \to a, \quad B \to b\}$$

erzeugt die Sprache

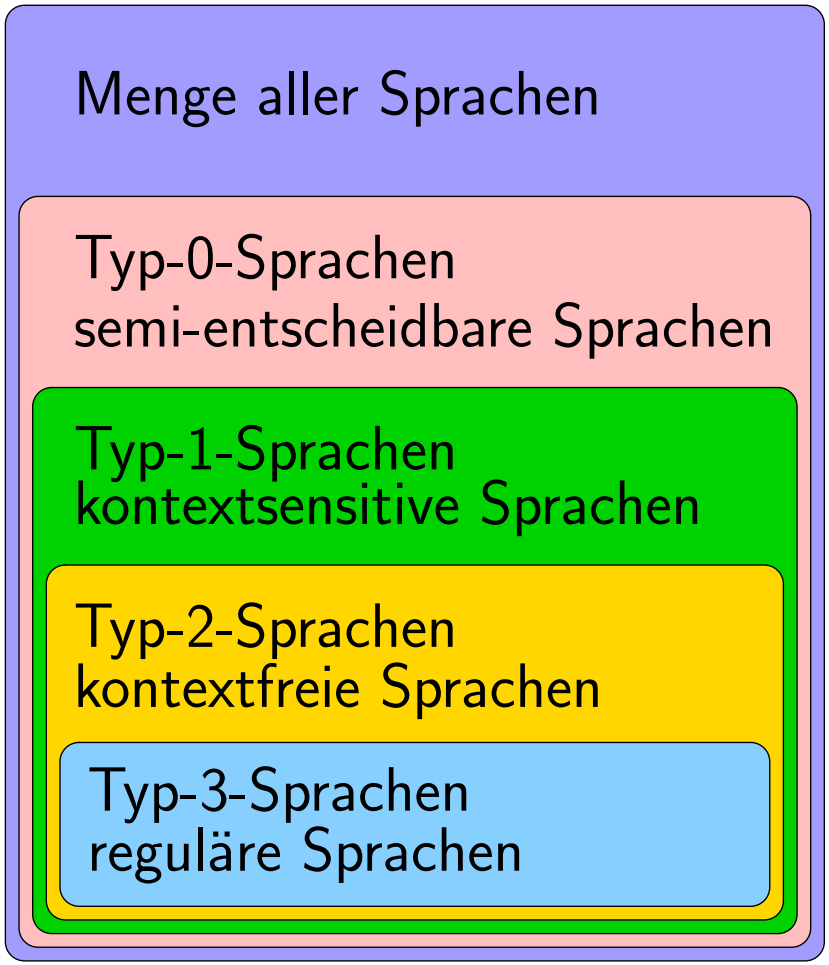
$$L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}.$$

Hier ist $\#_a(w)$ die Anzahl der a's im Wort w.

Chomsky-Hierarchie

- Typ-0-Grammatiken: keine Einschränkung
- Typ-1-Grammatiken: $|\text{linke Seite}| \leq |\text{rechte Seite}|$

  (Wortproblem ist hier noch entscheidbar.)



Berechnungsmodelle

- Was bedeutet es überhaupt im allgemeinen, dass eine Funktion berechenbar oder eine Sprache akzeptierbar ist?
- Wie kann man ein generelles Berechnungsmodell definieren?
- Was ist das Maschinenmodell für Chomsky-0-Sprachen?

Berechnungsmodelle

Im Laufe der Zeit sind mehrere Berechnungsmodelle entwickelt worden, die jedoch alle zueinander äquivalent sind:

- Turing-Maschinen (benannt nach Alan Turing)
- WHILE-Programme, GOTO-Programme
- μ-rekursive Funktionen

Turing-Maschinen mit entsprechender Beschränkung dienen auch als Maschinenmodell für Chomsky-1-Sprachen

Unentscheidbarkeit

Gibt es Sprachen, für die das Wortproblem ($w \in L$?) nicht entscheidbar ist?

Antwort: ja

Kann man Funktionen definieren, die nicht berechenbar sind? Das heißt, es gibt keine Turing-Maschine/kein WHILE-Programm/keine μ-rekursive Funktion, die diese Funktion berechnet?

Antwort: ja

Außerdem: Wie zeigt man diese negativen Resultate?

Unentscheidbarkeit

Die "Mutter" aller unentscheidbaren Probleme: das Halteproblem

Gegeben sei eine Turing-Maschine (oder einfach ein Programm) M und eine Eingabe $w \in \Sigma^*$.

Frage: Terminiert M bei Eingabe w?

Das Halteproblem ist unentscheidbar, das heißt, es gibt kein Verfahren, das – gegeben M und w – immer entscheidet, ob M mit Eingabe w terminiert oder nicht.

Wir werden die Unentscheidbarkeit des Halteproblems im Rahmen der Vorlesung zeigen. (Der Beweis ist überraschend kurz und elegant.)

Unentscheidbarkeit

Intuition: Warum ist das Halteproblem unentscheidbar?

Man könnte doch einfach M mit Eingabe w laufenlassen und nachsehen was passiert ....

- Wenn M nach einer endlichen Anzahl von Schritten von terminiert, dann hat man Gewissheit.
- Aber: wenn M nach 10.000 Schritten noch nicht terminiert hat, was dann? Nochmal 10.000 Schritte weiterlaufen lassen?

Unentscheidbarkeit

Neben dem Halteproblem gibt es noch viele andere wichtige unentscheidbare Probleme, z.B., das Schnittproblem für kontextfreie Sprachen.

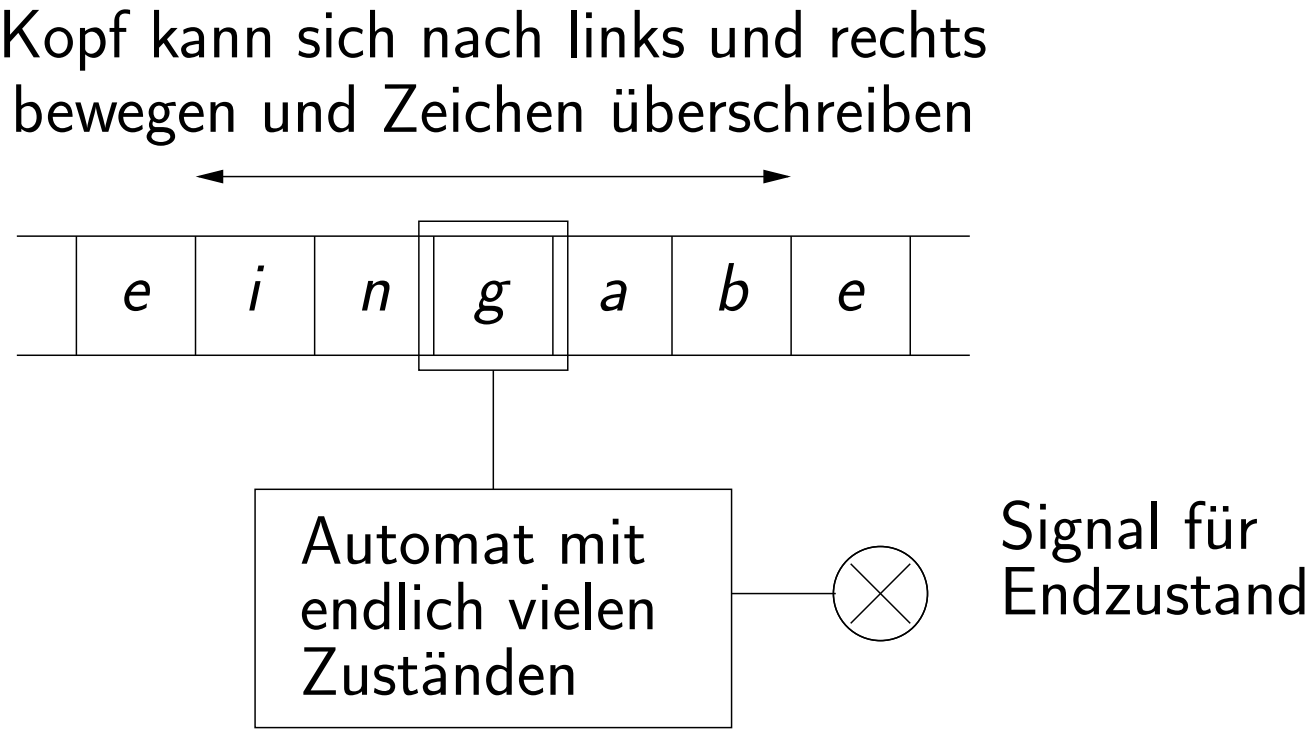
Gegeben zwei kontextfreie Grammatiken für Sprachen L_1, L_2.

Gilt $L_1 \cap L_2 = \emptyset$?

Beweistechnik: Man zeigt die Unentscheidbarkeit solcher Probleme meistens durch einen Widerspruchsbeweis. "Wenn dieses Problem entscheidbar wäre, dann wäre auch das Halteproblem entscheidbar."

Dazu kodiert man das Halteproblem so um, dass daraus eine Instanz eines solchen Problems wird.

Schematische Darstellung einer Turing-Maschine:



Turingmaschinen

Eigenschaften von Turing-Maschinen:

- Wie endliche Automaten lesen Turing-Maschinen eine Eingabe von einem Band und haben endlich viele Zustände.
- Unterschied zu endlichen Automaten: der Lesekopf kann sich nach links und rechts bewegen und auch Zeichen überschreiben.
- Falls nur Zeichen der Eingabe überschrieben werden: Turing-Maschine heißt linear beschränkt (Maschinenmodell für Chomsky-1-Sprachen).
- Falls der Lesekopf auch über den linken und rechten Rand hinauslaufen und dort schreiben kann: allgemeine Turingmaschine mit unbeschränktem Band (Maschinenmodell für Chomsky-0-Sprachen).

Turingmaschinen

Turing-Maschinen und Computer:

- Das Konzept der Turing-Maschine wurde von Alan Turing 1936 erfunden, noch bevor die ersten echten Computer gebaut wurden.
- Es ist nicht nur aus historischen Gründen interessant, sondern auch, weil es ein sehr einfaches Berechnungsmodell darstellt.

  Wenn man zeigen will, dass etwas *nicht* berechenbar ist, dann ist es viel besser, dies mit einem möglichst einfachen Berechnungsmodell zu tun. (Natürlich sollte man vorher sicherstellen, dass dieses Berechnungsmodell äquivalent zu komplexeren Modellen ist.)
- Analogie zu einem heutigen Computer:
  - Kontrolle mit endlich vielen Zuständen $\rightsquigarrow$ Programm
  - (Eingabe-)Band $\rightsquigarrow$ Speicher

Turingmaschinen

Beispiel 1: Turing-Maschine, die eine Binärzahl auf dem Band um eins inkrementiert.

Idee:

- Kopf der Turing-Maschine steht zunächst auf dem am weitesten links befindlichen (höchstwertigen) Bit der Binärzahl.
- Kopf nach rechts laufen lassen, bis ein Leerzeichen gefunden wird.
- Dann wieder nach links laufen und jede 1 durch 0 ersetzen, solange bis eine 0 oder ein Leerzeichen auftaucht.
- Dieses Zeichen dann durch 1 ersetzen, bis zum Zahlanfang laufen und in einen Endzustand übergehen.

Simulation

□: Leerzeichen auf dem Band

Simulation

□: Leerzeichen auf dem Band

Simulation

□: Leerzeichen auf dem Band

Simulation

□: Leerzeichen auf dem Band

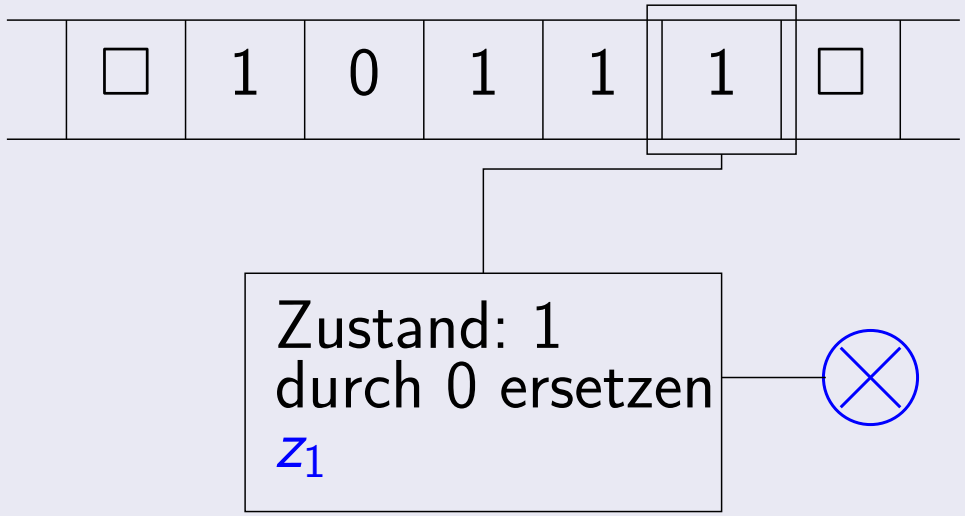
Simulation

□: Leerzeichen auf dem Band

Simulation

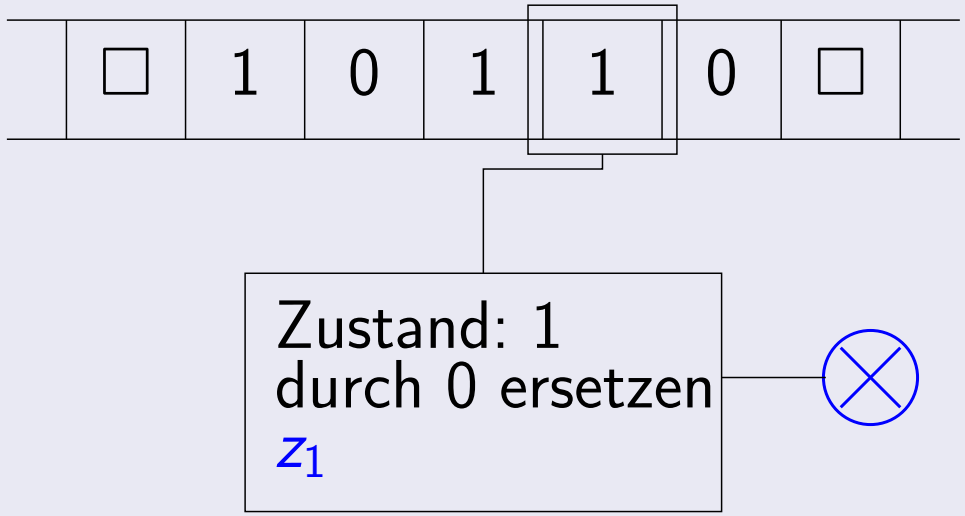
□: Leerzeichen auf dem Band

Simulation



□: Leerzeichen auf dem Band

Simulation



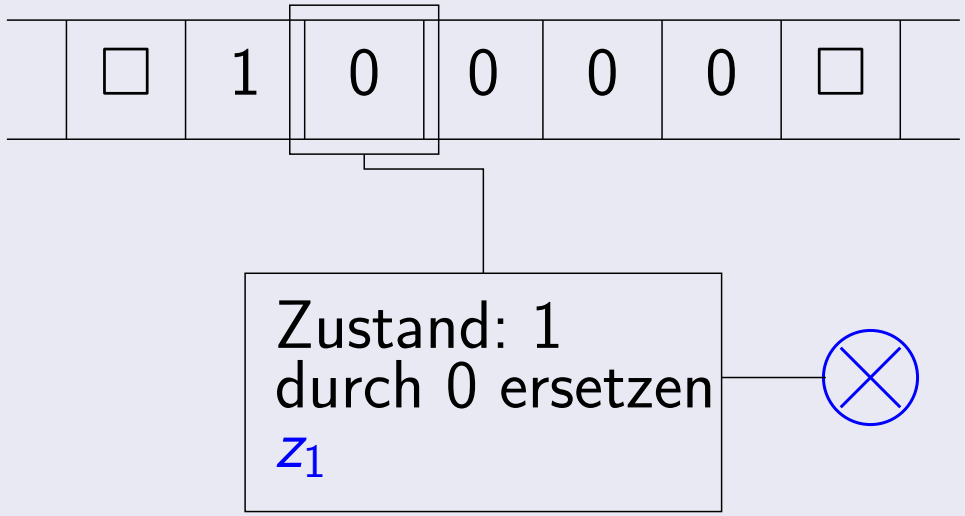
□: Leerzeichen auf dem Band

Turingmaschinen

Simulation

□: Leerzeichen auf dem Band

Simulation



□: Leerzeichen auf dem Band

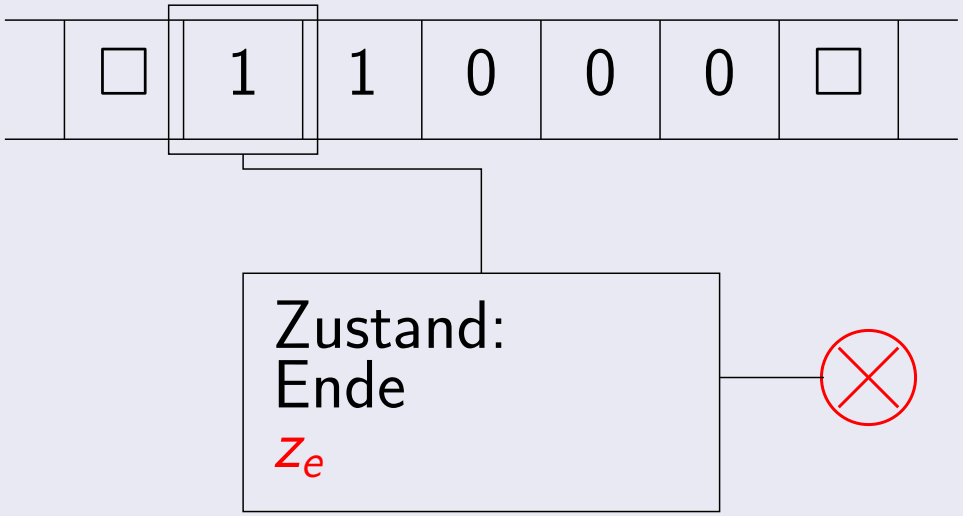
Simulation

□: Leerzeichen auf dem Band

Simulation

□: Leerzeichen auf dem Band

Simulation



□: Leerzeichen auf dem Band

Turingmaschinen

Turingmaschine (Definition)

Eine deterministische Turingmaschine M ist ein 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$, wobei

- Z die endliche Menge der Zustände,
- Σ das endliche Eingabealphabet,
- Γ mit $\Gamma \supset \Sigma$ das endliche Arbeitsalphabet oder Bandalphabet,
- $z_0 \in Z$ der Startzustand,
- $E \subseteq Z$ die Menge der Endzustände,
- $\delta\colon (Z \setminus E) \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ die Überführungsfunktion, *und*
- $\square \in \Gamma \setminus \Sigma$ das Leerzeichen oder Blank ist.

Abkürzung: TM

Bedeutung der Überführungsfunktion:
sei $\delta(z, a) = (z', b, x)$ mit $x \in \{L, R, N\}$.

Falls die Turingmaschine im Zustand z auf dem Symbol a steht, so

- wechselt sie in den Zustand z',
- überschreibt a durch b *und*
- führt folgende Kopfbewegung aus.
  - Kopf einen Schritt nach links, falls $x = L$.
  - Kopf bleibt stehen, falls $x = N$.
  - Kopf einen Schritt nach rechts, falls $x = R$.

Turingmaschinen

Neben deterministischen Turingmaschinen gibt es auch nichtdeterministische Turingmaschinen.

Überführungsfunktion für nichtdeterministische Turingmaschinen:

$$\delta\colon (Z \setminus E) \times \Gamma \to 2^{Z \times \Gamma \times \{L,R,N\}}.$$

Jedem Zustand und Bandsymbol wird eine (eventuell leere) Menge von möglichen Aktionen zugeordnet.

Turingmaschinen

Beispiel: Turingmaschine zur Inkrementierung einer Binärzahl

$$M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \square\}, \delta, z_0, \square, \{z_e\}) \qquad \text{mit}$$

Überführungsfunktion: Zahlende finden

$$\begin{aligned} \delta(z_0, 0) &= (z_0, 0, R) \\ \delta(z_0, 1) &= (z_0, 1, R) \\ \delta(z_0, \square) &= (z_1, \square, L) \end{aligned}$$

Überführungsfunktion: 1 durch 0 ersetzen

$$\begin{aligned} \delta(z_1, 0) &= (z_2, 1, L) \\ \delta(z_1, 1) &= (z_1, 0, L) \\ \delta(z_1, \square) &= (z_e, 1, N) \end{aligned}$$

Überführungsfunktion: zurück zum Zahlanfang

$$\begin{aligned} \delta(z_2, 0) &= (z_2, 0, L) \\ \delta(z_2, 1) &= (z_2, 1, L) \\ \delta(z_2, \square) &= (z_e, \square, R) \end{aligned}$$

Beispiel 2: Turing-Maschinen zur Spracherkennung

Wir suchen eine Turing-Maschine, die die Sprache $L = \{0^{2^n} \mid n \geq 0\}$ (nicht kontextfrei) erkennt.

Idee:

- Kopf steht zunächst am Beginn der Folge von Nullen.
- Anfang und Ende der Folge von Nullen markieren. Links neben der Folge von Nullen die Binärzahl 0 aufs Band schreiben.
- Nullen nacheinander durch ein anderes Zeichen (#) ersetzen. Nach jeder Ersetzung nach links zum Zähler laufen und diesen um eins inkrementieren.
- Sobald alle Nullen verschwunden sind (Endmarker ist erreicht), überprüfen, ob der Zähler die Form $10 \cdots 0$ hat.

Turingmaschinen

Wie bei anderen Maschinenmodellen gibt es auch bei Turingmaschinen den Begriff einer Konfiguration, d.h., einer Momentaufnahme einer Turingmaschinen-Berechnung

Konfiguration (Definition)

Eine Konfiguration einer Turingmaschine ist gegeben durch ein Wort

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

Bedeutung: $k = \alpha z \beta$

- $\alpha \in \Gamma^*$ ist der bereits besuchte Teil des Bandes vor dem Kopf.
- $\beta \in \Gamma^+$ ist der Abschnitt des Bandes ab dem Kopf, der bereits besucht wurde oder im Bereich des Eingabewortes liegt. Der Kopf steht auf dem ersten Zeichen von β.
- $z \in Z$ ist der aktuelle Zustand.

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Keine Bewegung

Es gilt: $a_1 \cdots a_m z\, b_1\, b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z'}\, c\, b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, N) \quad (m \geq 0,\ n \geq 1)$.

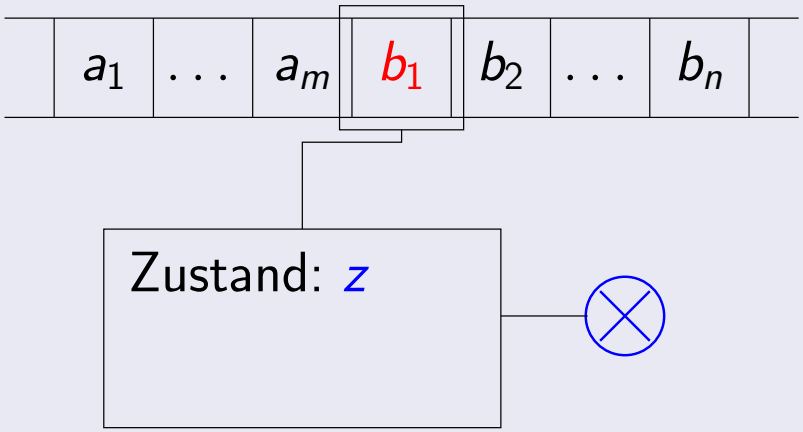
Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Keine Bewegung

Es gilt: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z'} c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, N) \quad (m \geq 0,\ n \geq 1)$.



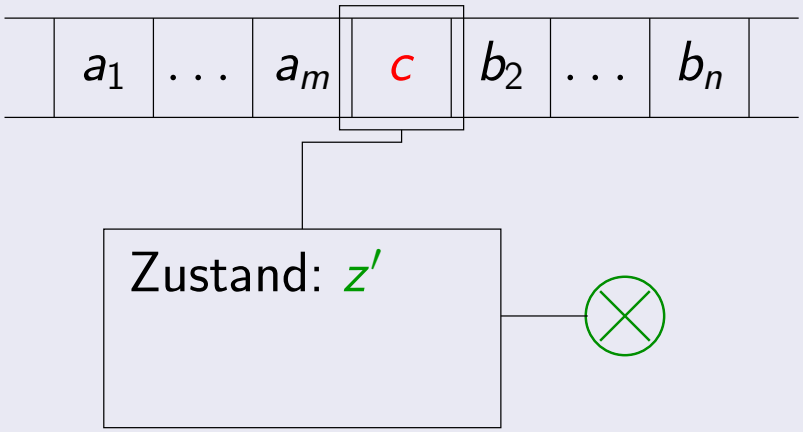
Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Keine Bewegung

Es gilt: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z'} c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, N) \quad (m \geq 0,\ n \geq 1)$.



Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach links

Es gilt: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} \mathbf{z'} a_m c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, L) \quad (m \geq 1,\ n \geq 1)$.

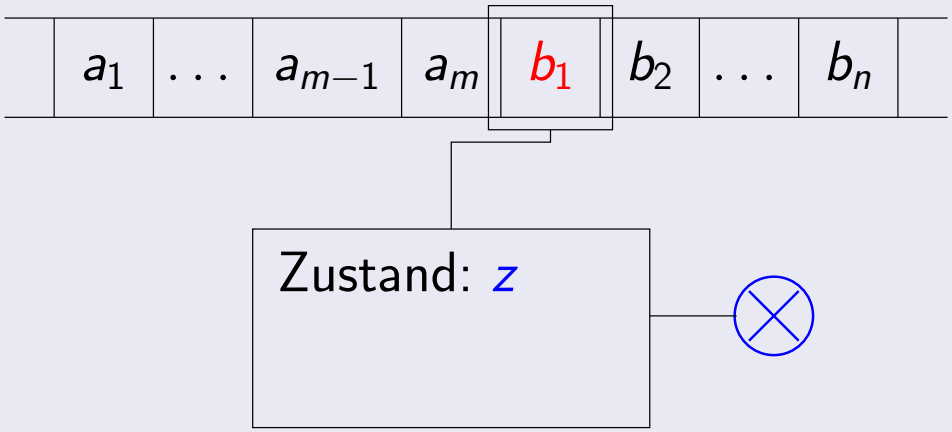
Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach links

Es gilt: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} z' a_m c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (z', c, L) \quad (m \geq 1,\ n \geq 1)$.



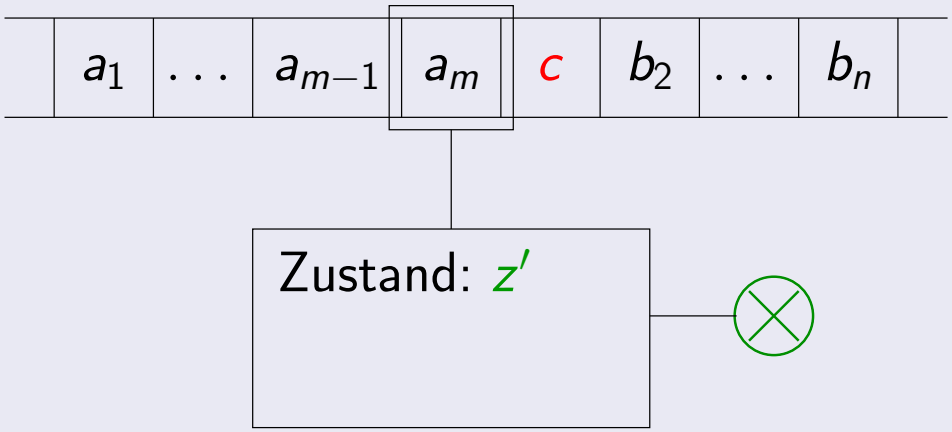
Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach links

Es gilt: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} z' a_m c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (z', c, L) \quad (m \geq 1,\ n \geq 1)$.



Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach rechts

Es gilt: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z'} b_2 \cdots b_n$,

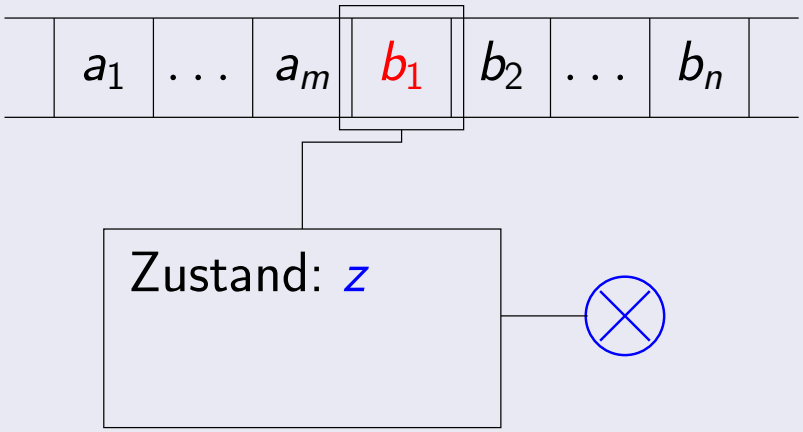
falls $\delta(z, b_1) = (\mathbf{z'}, c, R) \quad (m \geq 0,\ n \geq 2)$.

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach rechts

Es gilt: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z'} b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, R) \quad (m \geq 0,\ n \geq 2)$.



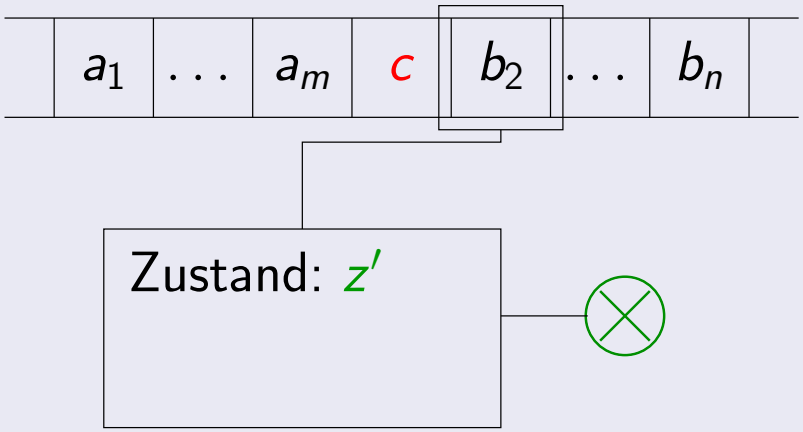
Turingmaschinen

Definition einer Übergangsrelation $\vdash_M$, die beschreibt, welche Konfigurationsübergänge möglich sind.

Schritt nach rechts

Es gilt: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z'} b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, R) \quad (m \geq 0,\ n \geq 2)$.



Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Linkes Bandende

Es gilt: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z}' \square c b_2 \cdots b_n$,

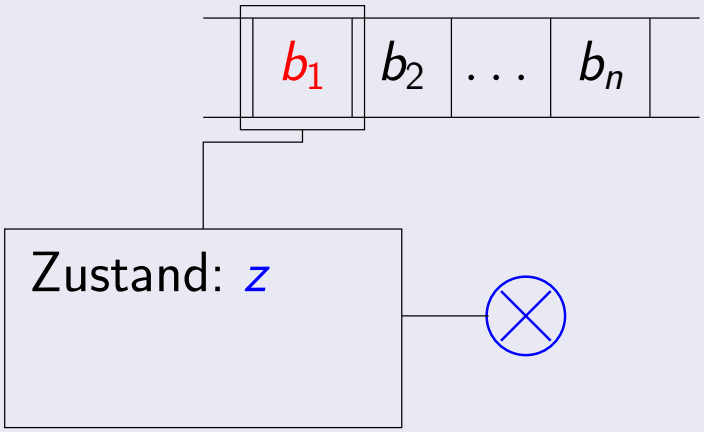
falls $\delta(z, b_1) = (\mathbf{z}', c, L)$.

Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Linkes Bandende

Es gilt: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z'} \square c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, L)$.

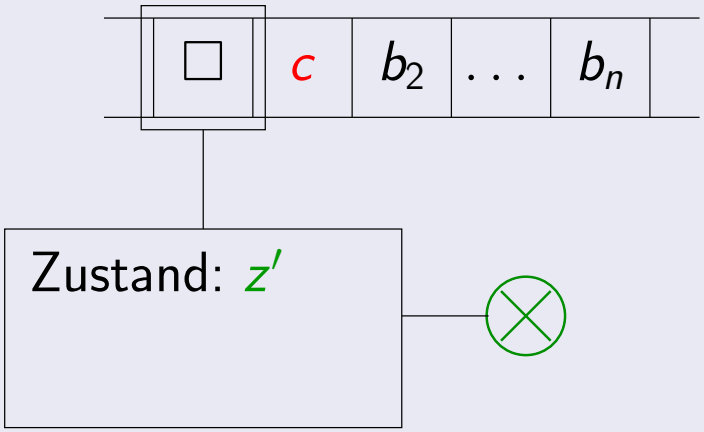


Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Linkes Bandende

Es gilt: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z'} \square c b_2 \cdots b_n$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, L)$.



Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Rechtes Bandende

Es gilt: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z}' \square$,

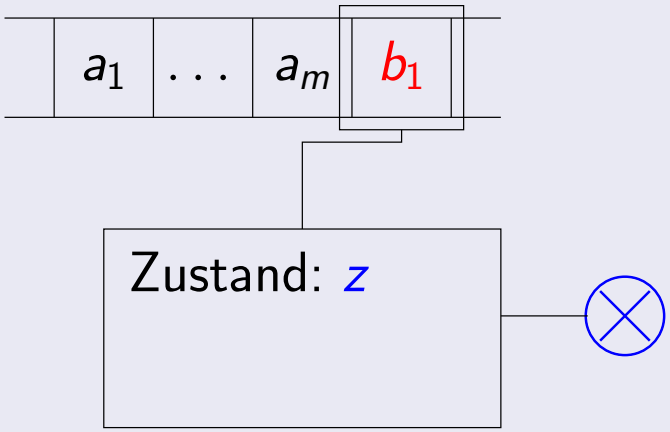
falls $\delta(z, b_1) = (\mathbf{z}', c, R)$.

Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Rechtes Bandende

Es gilt: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z}' \square$,

falls $\delta(z, b_1) = (\mathbf{z}', c, R)$.

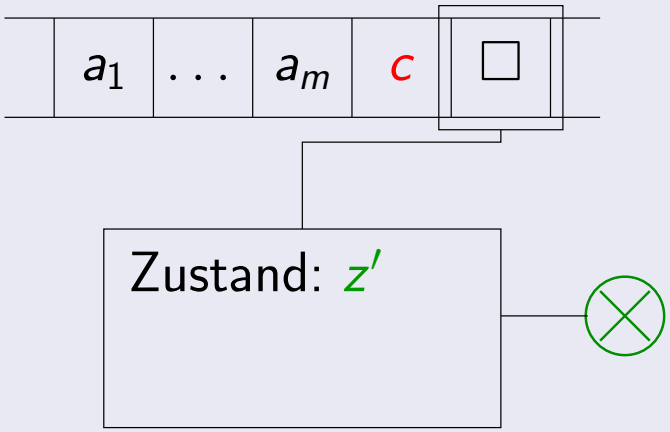


Sonderfälle: Bandende erreicht $\rightsquigarrow$ zusätzliches Leerzeichen muss hinzugefügt werden

Rechtes Bandende

Es gilt: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z'} \square$,

falls $\delta(z, b_1) = (\mathbf{z'}, c, R)$.



Akzeptierte Sprache (Definition)

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ eine Turingmaschine. Dann ist die von M akzeptierte Sprache:

$$T(M) = \{x \in \Sigma^+ \mid \exists k \in \Gamma^* E \Gamma^+ : z_0 x \vdash_M^* k\} \cup \{\varepsilon \mid \exists k \in \Gamma^* E \Gamma^+ : z_0 \square \vdash_M^* k\}.$$

Akzeptierte Sprache: Alle Eingabe-Wörter, mit denen die Turing-Maschine in einen Endzustand gelangen kann. Dabei startet die Turing-Maschine im Anfangszustand z_0, der Kopf befindet sich auf dem ersten Zeichen des Eingabe-Wortes. Falls dieses nicht existiert (Eingabe x ist das leere Wort) liest der Kopf ein Blank $\square$.

Turingmaschinen

Für **nicht-deterministische Turingmaschinen** müssen die Definitionen folgendermaßen angepaßt werden:

- Falls sich die Turingmaschine im Zustand z befindet und das Zeichen b auf dem Band steht, sind alle Konfigurationsübergänge möglich, die durch die Menge $\delta(z, b)$ beschrieben werden.
- Ein Wort ist akzeptiert, wenn es eine mögliche Folge von Konfigurationen gibt, die zu einem Endzustand führt, auch wenn andere Folgen in Sackgassen geraten oder unendlich lang sind, ohne dabei je einen Endzustand zu erreichen.

Linear beschränkte Automaten

Wir definieren nun ein Maschinenmodell für Chomsky-1-Sprachen (erzeugt durch monotone Grammatiken): linear beschränkte Automaten, die niemals außerhalb der Eingabe arbeiten dürfen.

Linear beschränkte Automaten

Ein (nicht)deterministischer linear beschränkter Automat (LBA) ist ein Tupel $A = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$, welches den gleichen Eigenschaften wie eine (nicht)deterministische Turingmaschine genügt, nur dass (i) A nicht das Blanksymbol $\square$ durch ein Nicht-Blanksymbol überschreiben darf und (ii) A nicht ein Nicht-Blanksymbol durch $\square$ überschreiben darf.

Die Relation $\vdash_A$ ist wie für eine TM definiert, nur dass wir die Sonderfälle für das linke und rechte Bandende (Folie 291 & 292) weglassen.

Die von dem LBA A akzeptierte Sprache ist

$$T(A) = \{w \in \Sigma^* \mid \exists k \in \Gamma^* E \Gamma^+ : z_0 w \square \vdash_A^* k\}$$

Chomsky-1-Sprachen

Bemerkung: Das abschließende Blanksymbol $\square$ erlaubt A, das rechte Bandende zu erkennen. Es dient sozusagen als rechtes Begrenzungssymbol.

Satz 1 (Kuroda)

Eine Sprache L wird von einem nichtdeterministischen LBA erkannt, genau dann, wenn es eine Typ-1-Grammatik G mit $L = L(G)$ gibt.

Beweis:

Sei zunächst $G = (V, \Sigma, P, S)$ eine Typ-1-Grammatik, d.h. es gilt $|\ell| \leq |r|$ für alle $(\ell, r) \in P$ (einzige Ausnahme: $S \to \varepsilon$, siehe ε-Sonderregelung).

Sei $w \in \Sigma^*$ eine Eingabe.

Wir simulieren nun eine Ableitung $S \Rightarrow_G^* w$ rückwärts mittels eines nichtdeterministischen LBA A.

Chomsky-1-Sprachen

Hierzu geht A wie folgt vor ($B[i]$ ist im folgenden das i-te Zeichen auf dem Band):

1. A bewegt den Kopf zum linkesten Symbol auf dem Band, "rät" nichtdeterministisch eine Regel $(\ell, r) \in P$ und merkt sich diese im Zustand.
2. Nun läuft der Kopf von A nach rechts zu einer nichtdeterministisch geratenen Position i.
3. Falls $B[i] \cdots B[i + |r| - 1] = r$ gilt, schreibt A auf den Bandabschnitt $B[i] \cdots B[i + |\ell| - 1]$ das Wort ℓ. Ansonsten gehe wieder zu (1).
4. Falls $|\ell| < |r|$ gilt, muss M nun jedes Zeichen auf dem Band ab Position $i + |r|$ um genau $|r| - |\ell|$ viele Positionen nach links verschieben.
5. A akzeptiert, falls das aktuelle Band mit $S\square$ oder $S\widetilde{\square}$ beginnt ($\widetilde{\square}$ ist eine Kopie von $\square$), ansonsten gehe wieder zu (1).

Falls $S \to \varepsilon$ eine Produktion in P ist (d.h. $\varepsilon \in L(G)$), kann A bei Lesen von $\square$ direkt aus dem Anfangszustand in einen Endzustand übergehen.

Chomsky-1-Sprachen

Für diesen LBA A gilt $L(G) = T(A)$.

Wir zeigen nun die andere Richtung.

Das folgende Lemma wird hilfreich sein.

Lemma 2

Sei $G = (V, \Sigma \cup \{r\}, P, S)$ eine Typ-1-Grammatik mit $r \notin \Sigma$ und $L(G) \subseteq \Sigma^* r$. Dann existiert eine Typ-1-Grammatik G' mit

$$L(G') = \{w \in \Sigma^* \mid wr \in L(G)\}.$$

Beweis:

O.B.d.A. können wir annehmen, dass gilt:

- Für jede Produktion $(u, v) \in P$ gilt $0 \leq |v| - |u| \leq 1$
- S kommt nicht in einer rechten Seite vor.

Chomsky-1-Sprachen

Wir definieren eine neue Variablenmenge V' durch

$$V' = V \cup \{r\} \cup \{(ab] \mid a, b \in V \cup \Sigma \cup \{r\}\}.$$

Die neue Produktionsmenge P' enthält die folgenden Produktionen:

- $S \to \varepsilon$ falls $r \in L(G)$,
- $S \to (ab]$ falls $a, b \in V \cup \Sigma \cup \{r\}$ und $S \Rightarrow_G^* ab$,
- $x(ab] \to y(cd]$ falls $(xab \to ycd) \in P$, $a, b, c, d \in V \cup \Sigma \cup \{r\}$
- $x(ab] \to y(cb]$ falls $(xa \to yc) \in P$, $a, b, c \in V \cup \Sigma \cup \{r\}$
- $(ab] \to (ac]$ falls $(b \to c) \in P$, $a, b, c \in V \cup \Sigma \cup \{r\}$
- $(ab] \to ax(cd]$ falls $(b \to xcd) \in P$, $a, b, c, d \in V \cup \Sigma \cup \{r\}$
- alle Produktionen aus P und
- $(ar] \to a$ für $a \in \Sigma$

Dann ist $G' = (V', \Sigma, P', S)$ die gesuchte Grammatik. □

Chomsky-1-Sprachen

Völlig analog zeigt man:

Lemma 3

Sei $G = (V, \Sigma \cup \{\ell\}, P, S)$ eine Typ-1-Grammatik mit $\ell \notin \Sigma$ und $L(G) \subseteq \ell\Sigma^*$. Dann existiert eine Typ-1-Grammatik G' mit

$$L(G') = \{w \in \Sigma^* \mid \ell w \in L(G)\}.$$

und durch Anwendung beider Lemmata:

Lemma 4

Sei $G = (V, \Sigma \cup \{\ell, r\}, P, S)$ eine Typ-1-Grammatik mit $\ell, r \notin \Sigma$ und $L(G) \subseteq \ell\Sigma^* r$. Dann existiert eine Typ-1-Grammatik G' mit

$$L(G') = \{w \in \Sigma^* \mid \ell w r \in L(G)\}.$$

Chomsky-1-Sprachen

Nun zurück zum Beweis von Satz 1. Sei $A = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ ein LBA.

Aufgrund von Lemma 4 genügt es, eine Typ-1-Grammatik für die Sprache $\{\$w\Box \mid w \in T(A)\}$ anzugeben.

Hierzu simulieren wir A rückwärts mittels der Typ-1-Grammatik $G = (V, \Sigma \cup \{\$, \Box\}, P, S)$ mit $V = \{S, A, B\} \cup (\Gamma \setminus (\Sigma \cup \{\Box\})) \cup (Z \times \Gamma)$ und der folgenden Produktionsmenge P:

$$
\begin{array}{rcll}
S & \to & \$A & \\
A & \to & aA \mid (z,a)B \mid (z,\Box) & (a \in \Gamma \setminus \{\Box\}, z \in E) \\
B & \to & aB \mid \Box & (a \in \Gamma \setminus \{\Box\}) \\
(z',a') & \to & (z,a) & \text{falls } (z',a',N) \in \delta(z,a) \\
a'(z',b) & \to & (z,a)b & \text{falls } (z',a',R) \in \delta(z,a), b \in \Gamma \\
(z',b)a' & \to & b(z,a) & \text{falls } (z',a',L) \in \delta(z,a), b \in \Gamma \\
\$(z_0,a) & \to & \$a & (a \in \Sigma) \\
\$(z_0,\Box) & \to & \$\Box &
\end{array}
$$

Chomsky-0-Sprachen

Satz 5 (Turingmaschinen und Chomsky-0-Sprachen)

Eine Sprache L wird von einer nichtdeterministischen Turingmaschine erkannt, genau dann, wenn es eine Typ-0-Grammatik G mit $L = L(G)$ gibt.

Beweisidee: durch Modifikation des Beweises von Satz 1:

Grammatiken $\rightarrow$ Turingmaschinen: hier muss bei der Simulation der Grammatik auf dem Turingmaschinen-Band bei verkürzenden Regeln (linke Seite ist länger als rechte Seite) der Bandinhalt auseinandergeschoben werden.

Turingmaschinen $\rightarrow$ Grammatiken: hier muss dafür gesorgt werden, dass die Grammatik bei Simulation der Turingmaschine links und rechts Leerzeichen erzeugen kann und diese nach erfolgreicher Berechnung auch wieder löscht.

Chomsky-0-Sprachen

Formal: Wir simulieren eine TM $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ mittels der Typ-0-Grammatik $G = (\{S, A, B, \$_1, \$_2\} \cup (\Gamma \setminus \Sigma) \cup (Z \times \Gamma),\ \Sigma, P, S)$ mit der folgenden Produktionsmenge P:

$$
\begin{array}{rcll}
S & \to & \$_1 A & \\
A & \to & aA \mid (z,a)B & (a \in \Gamma, z \in E) \\
B & \to & aB \mid \$_2 & (a \in \Gamma) \\
(z',a') & \to & (z,a) & \text{falls } (z',a',N) \in \delta(z,a) \\
a'(z',b) & \to & (z,a)b & \text{falls } (z',a',R) \in \delta(z,a), b \in \Gamma \\
(z',b)a' & \to & b(z,a) & \text{falls } (z',a',L) \in \delta(z,a), b \in \Gamma \\
\$_1\square & \to & \$_1 & \\
\$_1(z_0,a) & \to & a & (a \in \Sigma) \\
\square\$_2 & \to & \$_2 & \\
a\$_2 & \to & a & (a \in \Sigma) \\
\$_1(z_0,\square)\$_2 & \to & \varepsilon &
\end{array}
$$

Ergebnisse für Chomsky-1- und Chomsky-0-Sprachen

Satz 6 (Abschluss unter Komplement von Typ-1-Sprachen, Immerman, Szelepcsényi)

Wenn L eine Typ-1-Sprache ist, dann ist auch $\overline{L} = \Sigma^* \setminus L$ eine Typ-1-Sprache.

(Hier ohne Beweis, siehe Vorlesung **Strukturelle Komplexitätstheorie im Wintersemester**)

Satz 7 (Nicht-Abschluss unter Komplement von Typ-0-Sprachen)

Es gibt eine Typ-0-Sprache $L \subseteq \Sigma^*$, so dass $\overline{L} = \Sigma^* \setminus L$ keine Typ-0-Sprache ist.

Begründung und Beispiele später im Kapitel Berechenbarkeitstheorie

Ergebnisse für Chomsky-1- und Chomsky-0-Sprachen

Satz 8 (Determinismus und Nichtdeterminismus bei Turingmaschinen)

Zu jeder nichtdeterministischen Turingmaschine gibt es eine deterministische Turingmaschine, die dieselbe Sprache akzeptiert.

Beweis:

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ eine nichtdeterministische TM, d. h.

$$\delta \colon (Z \setminus E) \times \Gamma \to 2^{Z \times \Gamma \times \{L,R,N\}}.$$

Idee: Wir konstruieren eine deterministische Turingmaschine, die bei Eingabe $x \in \Sigma^*$ systematisch nach einer erfolgreichen Berechnung von M sucht.

Sei $\# \notin Z \cup \Gamma$ ein neues Symbol.

Ergebnisse für Chomsky-1- und Chomsky-0-Sprachen

Eine erfolgreiche Berechnung von M auf Eingabe x ist ein Wort der Form

$$k_0 \# k_1 \# \cdots k_{m-1} \# k_m$$

mit folgenden Eigenschaften:

(1) $k_0, k_1, \ldots, k_m \in \Gamma^* Z \Gamma^+$

(2) $k_0 = z_0 x \square$.

(3) $\forall 0 \leq i < m : k_i \vdash_M k_{i+1}$

(4) $k_m \in \Gamma^* E \Gamma^+$

Offensichtlich gilt $x \in L(M)$ genau dann, wenn eine erfolgreiche Berechnung von M auf Eingabe x existiert.

Eine deterministische Turingmaschine M' kann bei Eingabe von x und $w \in (Z \cup \Gamma \cup \{\#\})^*$ feststellen, ob w eine erfolgreiche Berechnung von M auf Eingabe x ist (geht sogar mit einem deterministischen LBA).

Hierzu muss M' lediglich die vier Eigenschaften (1)–(4) überprüfen.

Ergebnisse für Chomsky-1- und Chomsky-0-Sprachen

Nun muss man nur noch eine deterministische Turingmaschine M'' konstruieren, die systematisch der Reihe nach alle Wörter $w \in (Z \cup \Gamma \cup \{\#\})^*$ durchgeht, und jedesmal (mittels M') überprüft, ob w eine erfolgreiche Berechnung von M auf Eingabe x ist.

"Systematisch der Reihe nach" kann hier z. B. mittels einer längenlexikographischen Ordnung formalisiert werden.

Sei zunächst $\prec$ eine beliebige lineare Ordnung auf dem Alphabet $\Omega = Z \cup \Gamma \cup \{\#\}$.

Die zu $\prec$ gehörende längenlexikographische Ordnung $\prec_{\ell\ell}$ auf Ω^* ist wie folgt definiert:

Für $u, v \in \Omega^*$ gilt $u \prec_{\ell\ell} v$ genau dann, wenn

- $|u| < |v|$ oder
- $|u| = |v|$ und es gibt $x, y, z \in \Omega^*, a, b \in \Omega$ mit $u = xay$, $v = xbz$, $a \prec b$.

Ergebnisse für Chomsky-1- und Chomsky-0-Sprachen

Grobstruktur der deterministischen Turingmaschine M'':

1. Initialisiere hinter der Eingabe x auf dem Band ein Wort $w \in (Z \cup \Gamma \cup \{\#\})^*$ mit ε
2. Überprüfe mittels M' ob w eine erfolgreiche Berechnung von M auf Eingabe x ist.
   Falls ja, gehe in einen Endzustand über, sonst gehe zu (3)
3. Inkrementiere w, d. h. überschreibe w mit dem längenlexikographisch nächsten Wort (formal wird w durch das kleinste Wort w' mit $w \prec_{\ell\ell} w'$ ersetzt).
4. Gehe zu (2).

□

Determinismus und Nichtdeterminismus bei linear beschränkten Automaten (erstes LBA-Problem)

Für linear beschränkte Automaten ist es *nicht* bekannt, ob die deterministischen und die nichtdeterministischen Maschinenmodelle gleich ausdrucksmächtig sind.

Berechenbarkeit: Motivation

Nach der Beantwortung der Frage, welche Sprachen maschinell akzeptierbar sind, beschäftigen wir uns mit der Frage, welche Funktionen berechenbar sind.

Wir betrachten folgende Typen von Funktionen:

- (mehrstellige) Funktionen auf natürlichen Zahlen (die Null ist eingeschlossen):

$$f : \mathbb{N}^k \to \mathbb{N}$$

- Funktionen auf Wörtern:

$$f : \Sigma^* \to \Sigma^*$$

Erlaubt sind auch partielle Funktionen, die nicht notwendigerweise überall definiert sind.

Berechenbarkeit: Motivation

Intuitiver Berechenbarkeitsbegriff

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ soll als berechenbar angesehen werden, wenn es ein Rechenverfahren/einen Algorithmus/ein Programm gibt, das f berechnet, d.h.

- das Verfahren erhält $(n_1, \ldots, n_k)$ als Eingabe,
- terminiert nach endlich vielen Schritten, falls die Funktion auf dieser Eingabe definiert ist
- und gibt $f(n_1, \ldots, n_k)$ aus.

Falls die Funktion auf $(n_1, \ldots, n_k)$ nicht definiert ist, so soll das Verfahren nicht stoppen (z.B., durch eine unendliche Schleife).

Berechenbarkeit: Motivation

Die Äquivalenz vieler Berechnungsmodelle (das wird noch gezeigt) und das intuitive Verständnis des Begriff der Berechenbarkeit führen zu folgender (nicht beweisbaren) These.

Churchsche These

Die durch die formale Definition der Turingmaschinen-Berechenbarkeit (äquivalent: WHILE-Berechenbarkeit, GOTO-Berechenbarkeit, μ-Rekursivität) erfasste Klasse von Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

Bemerkungen: Ein Berechnungsmodell, das äquivalent zu Turingmaschinen ist, nennt man auch Turing-mächtig. Der entsprechende Berechenbarkeitsbegriff heißt Turing-Berechenbarkeit.

Fast alle Programmiersprachen sind Turing-mächtig.

Berechenbarkeit: Motivation

Nicht-berechenbare Funktionen

Es gibt Funktionen der Form $f : \mathbb{N} \to \mathbb{N}$, die nicht berechenbar sind.

Beweisidee: wir wählen ein beliebiges Berechnungsmodell und stellen nur eine Anforderung:

> Programme bzw. Maschinen in diesem Berechnungsmodell, können als Wörter über einem endlichen Alphabet kodiert werden.

Dann gilt: es gibt höchstens abzählbar viele Maschinen/Programme.

Berechenbarkeit: Motivation

Aber: es gibt überabzählbar viele (totale) Funktionen.

Wir zeigen dies durch einen Widerspruchsbeweis: angenommen die Menge aller Funktionen $\mathcal{F}$ auf natürlichen Zahlen ist abzählbar. Das heißt, es gibt eine surjektive Abbildung $F : \mathbb{N} \to \mathcal{F}$.

Wir konstruieren die Funktion $g : \mathbb{N} \to \mathbb{N}$ mit

$$g(n) = f_n(n) + 1, \qquad \text{wobei } f_n = F(n).$$

Da F surjektiv ist, muss es eine natürliche Zahl i geben mit $F(i) = g$. Für dieses i gilt dann: $g(i) = f_i(i)$. Aber das ist ein Widerspruch zur Definition von g mit $g(i) = f_i(i) + 1$.

Berechenbarkeit: Motivation

Veranschaulichung: wir stellen F dadurch dar, indem wir zu jedem n die Funktion f_n als Folge $f_n(0)$, $f_n(1)$, $f_n(2)$, ... notieren.

Zum Beispiel:

n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	...
0	7	20	33	0	12	
1	12	33	94	2	17	
2	99	101	16	11	22	
3	2	0	14	99	42	
4	17	5	77	7	11	
...						

Berechenbarkeit: Motivation

Veranschaulichung: wir stellen F dadurch dar, indem wir zu jedem n die Funktion f_n als Folge $f_n(0)$, $f_n(1)$, $f_n(2)$, ... notieren.

Alle Zahlen auf der Diagonale verwenden ...

n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	...
0	7	20	33	0	12	
1	12	33	94	2	17	
2	99	101	16	11	22	
3	2	0	14	99	42	
4	17	5	77	7	11	
...						

Berechenbarkeit: Motivation

Veranschaulichung: wir stellen F dadurch dar, indem wir zu jedem n die Funktion f_n als Folge $f_n(0)$, $f_n(1)$, $f_n(2)$, ... notieren.

... und um eins erhöhen. Dadurch erhält man g.

n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	...
0	8	20	33	0	12	
1	12	34	94	2	17	
2	99	101	17	11	22	
3	2	0	14	100	42	
4	17	5	77	7	12	
...						

Berechenbarkeit: Motivation

Veranschaulichung: wir stellen F dadurch dar, indem wir zu jedem n die Funktion f_n als Folge $f_n(0)$, $f_n(1)$, $f_n(2)$, ... notieren.

Die Funktion g kann aber aufgrund dieser Konstruktion mit keiner der anderen Funktionen übereinstimmen.

n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	...
0	8	20	33	0	12	
1	12	34	94	2	17	
2	99	101	17	11	22	
3	2	0	14	100	42	
4	17	5	77	7	12	
...						

Berechenbarkeit: Motivation

Veranschaulichung: wir stellen F dadurch dar, indem wir zu jedem n die Funktion f_n als Folge $f_n(0)$, $f_n(1)$, $f_n(2)$, ... notieren.

Die Funktion g kann aber aufgrund dieser Konstruktion mit keiner der anderen Funktionen übereinstimmen.

n	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	...
0	8	20	33	0	12	
1	12	34	94	2	17	
2	99	101	17	11	22	
3	2	0	14	100	42	
4	17	5	77	7	12	
...						

Diese Art von "selbstbezüglichen" Beweisen nennt man aufgrund ihrer Veranschaulichung durch solche Diagramme oft Diagonalisierungsbeweise.

Turing-Berechenbarkeit

Nach dem intuitiven Berechenbarkeitsbegriff beschäftigen wir uns nun mit dem formalen Berechenbarkeitsbegriff, zunächst basierend auf Turingmaschinen.

Wir wissen bereits, was es bedeutet, dass eine Turingmaschine eine Sprache akzeptiert. Nun definieren wir, was es bedeutet, dass eine Turingmaschine eine Funktion berechnet.

Für eine Zahl $n \in \mathbb{N}$ sei $bin(n)$ die Binärdarstellung von n:

- $bin(n) \in 1\{0,1\}^* \cup \{0\}$
- Falls $bin(n) = b_k b_{k-1} \cdots b_0$, so gilt $n = \sum_{i=0}^{k} b_i 2^i$.

Turing-berechenbare Funktionen auf natürlichen Zahlen

Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist Turing-berechenbar, falls es eine deterministische Turingmaschine $M = (Z, \{0, 1, \#\}, \Gamma, \delta, z_0, \Box, E)$ gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$ gilt:

- Falls $f(n_1, \ldots, n_k)$ undefiniert ist, so terminiert M auf der Startkonfiguration $z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\#$ nicht, d. h. es gibt keine Konfiguration $c \in \Gamma^* E \Gamma^+$ mit

$$z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\# \vdash_M^* c.$$

- Falls $f(n_1, \ldots, n_k)$ definiert ist, und $f(n_1, \ldots, n_k) = m$, dann existiert ein Endzustand $z_e \in E$ mit

$$z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\# \vdash_M^* \Box \cdots \Box z_e bin(m) \Box \cdots \Box.$$

Turing-Berechenbarkeit

Intuition:

- Wenn man die Zahlen $n_1, \ldots, n_k$ in Binärdarstellung – voneinander getrennt durch $\#$ – aufs Band schreibt, so terminiert M genau dann, wenn $f(n_1, \ldots, n_k)$ definiert ist.
- Falls $f(n_1, \ldots, n_k) = m$, so berechnet M aus $n_1, \ldots, n_k$ die Zahl $f(n_1, \ldots, n_k)$ (möglicherweise umgeben von Leerzeichen) und geht in einen Endzustand über.

Turing-Berechenbarkeit

Turing-berechenbare Funktionen auf Wörtern

Eine partielle Funktion $f : \Sigma^* \to \Sigma^*$ ist Turing-berechenbar, falls es eine deterministische Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

- Falls $f(x)$ undefiniert ist, so terminiert M auf der Startkonfiguration z_0x nicht, d. h. es gibt kein $c \in \Gamma^* E \Gamma^+$ mit $z_0x \vdash_M^* c$.
- Falls $f(x)$ definiert ist, und $f(x) = y$, dann existiert ein Endzustand $z_e \in E$ mit $z_0x \vdash_M^* \Box \cdots \Box z_e y \Box \cdots \Box$.

Intuition:

- Wenn man das Wort x aufs Band schreibt, so terminiert M genau dann, wenn $f(x)$ definiert ist.
- Wenn $f(x) = y$, so berechnet M aus x das Wort y (möglicherweise umgeben von Leerzeichen) und geht in einen Endzustand über.

Turing-Berechenbarkeit

Beispiele für Turing-berechenbare Funktionen:

Beispiel 1: die Nachfolgerfunktion $n \mapsto n+1$ ist Turing-berechenbar (siehe die im vorherigen Kapitel angegebene Turingmaschine, die auf Binärdarstellungen arbeitet).

Beispiel 2: Sei Ω die überall undefinierte Funktion. Diese ist auch Turing-berechenbar, etwa durch eine Turingmaschine, die keinen Endzustand hat. Beispielsweise durch eine Turingmaschine mit der Übergangsregel

$$\delta(z_0, a) = (z_0, a, N) \qquad \text{für alle } a \in \Gamma.$$

Beispiel 3: gegeben sei eine Typ-0-Sprache $L \subseteq \Sigma^*$. Wir betrachten die sogenannte "halbe" charakteristische Funktion von L:

$$\begin{aligned} \chi_L \colon \Sigma^* &\to \{1\} \\ \chi_L(w) &= \begin{cases} 1 & \text{falls } w \in L \\ \text{undefiniert} & \text{sonst} \end{cases} \end{aligned}$$

Turing-Berechenbarkeit

Idee für eine Turingmaschine M, die χ_L berechnet:

- Wir verwenden die Transformation "Grammatik $\longrightarrow$ Turingmaschine", und erhalten eine Turingmaschine M' mit $T(M') = L$.
- Die Maschine M' ist nicht-deterministisch. Wegen der Äquivalenz von deterministischen und nicht-deterministischen Turingmaschinen kann man M' in eine deterministische Turingmaschine M'' mit $T(M'') = T(M') = L$ umwandeln.
- Aus M'' erhalten wir leicht eine deterministische Turingmaschine M, die sich wie M'' verhält, außer: Wenn M'' in einen Endzustand übergehen sollte (und damit akzeptiert), dann überschreibt M den kompletten Bandinhalt mit 1 und geht in einen Endzustand über.
- Beachte: Wenn die Eingabe x nicht zu L gehört, wird M nicht terminieren.

Mehrband-Turingmaschine

Wir führen jetzt mehrere neue Berechnungsmodelle ein und zeigen, dass sie alle äquivalent zu Turingmaschinen sind. Das erste davon ist die sogenannte Mehrband-Turingmaschine.

Mehrband-Turingmaschine

- Eine Mehrband-Turingmaschine besitzt k ($k \geq 1$) Bänder mit k unabhängigen Köpfen, aber nur einen Zustand.
- Aussehen der Übergangsfunktion:

$$\delta \colon (Z \setminus E) \times \Gamma^k \to Z \times \Gamma^k \times \{L, R, N\}^k$$

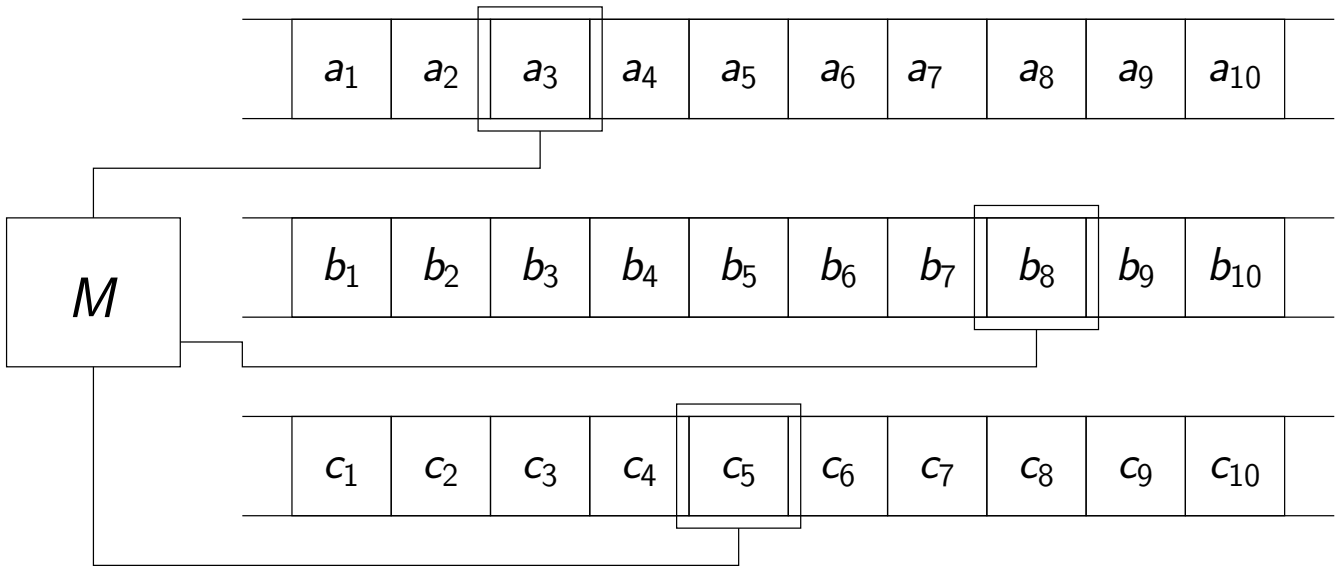
  (ein Zustand, k Bandsymbole, k Bewegungen)
- Die Ein- und Ausgabe stehen jeweils auf dem ersten Band. Zu Beginn sind die restlichen Bänder leer.

Mehrband-Turingmaschine

Satz 9 (Mehrband-Turingmaschinen $\longrightarrow$ Turingmaschinen)

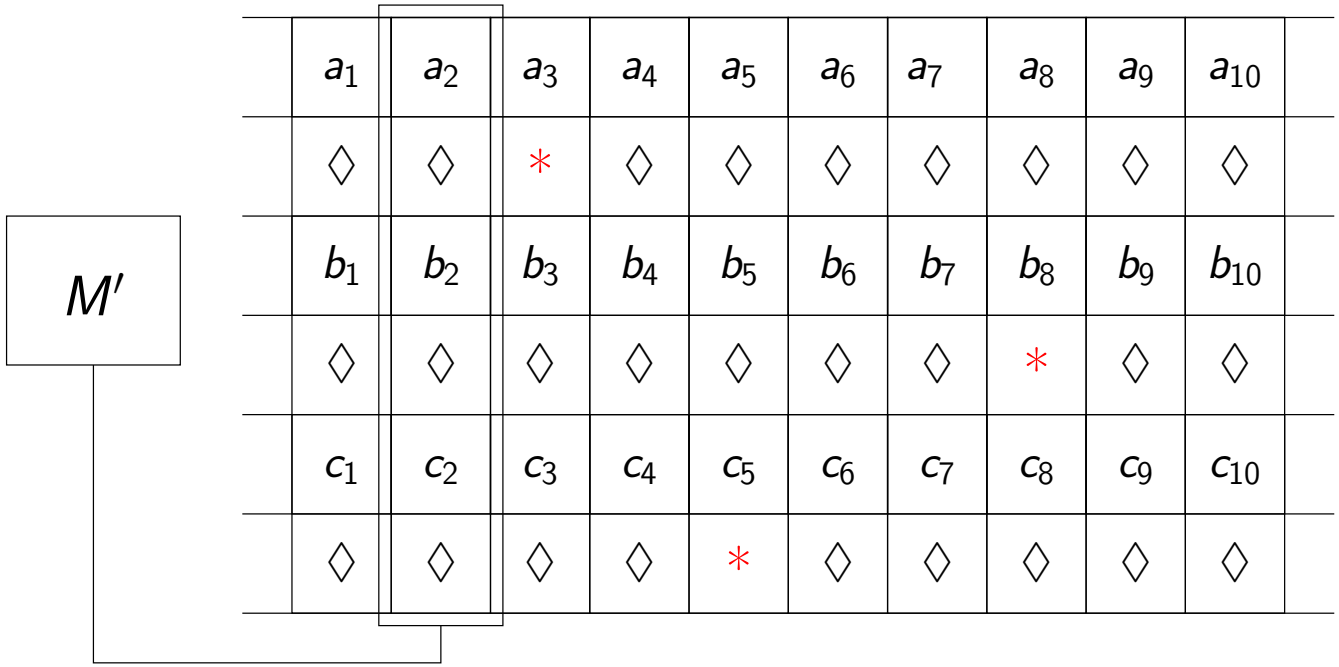
Zu jeder Mehrband-Turingmaschine M gibt es eine (Einband-)Turingmaschine M', die dieselbe Sprache akzeptiert bzw. dieselbe Funktion berechnet.

Beweisidee: wir beginnen mit der Darstellung einer typischen Konfiguration einer Mehrband-Turingmaschine



Mehrband-Turingmaschine

Simulation mittels Einband-Turingmaschine durch Erweiterung des Alphabets: Wir fassen die übereinanderliegenden Bandeinträge zu einem Feld zusammen.



Mehrband-Turingmaschine

Neues Bandalphabet:

$$\Gamma' = \Gamma \cup (\Gamma \cup \{*, \Diamond\})^{2k}$$

Bedeutung:

- Mit Hilfe von Symbolen von Γ wird die Eingabe dargestellt. Diese wird dann in einem ersten Durchlauf in die "Mehrband-Kodierung" umgewandelt.
- Ein Alphabetsymbol der Form $(a, *, b, \Diamond, c, *, \dots) \in (\Gamma \cup \{*, \Diamond\})^{2k}$ hat die Bedeutung:

  Entsprechende Felder sind mit $a, b, c, \dots$ belegt,

  1. und 3. Kopf sind anwesend. ($*$: Kopf anwesend, $\Diamond$: Kopf nicht anwesend)

Mehrband-Turingmaschine

Problem: Eine Einband-Turingmaschine hat nur *einen* Kopf und der kann nur an *einer* Stelle stehen $\rightsquigarrow$ Simulation eines Übergangs der Mehrband-Turingmaschine in mehreren Schritten.

Simulation:

- Zu Beginn der Simulation eines Schritts steht der Kopf der Einband-Turingmaschine M' links von allen $*$-Markierungen.
- Dann läuft der Kopf nach rechts, überschreitet alle $*$-Markierungen und merkt sich die jeweils anzuwendenden Fälle der δ-Funktion. (Dazu benötigt man viele Zustände.)
- Dann läuft der Kopf wieder zurück nach links und führt alle notwendigen Änderungen aus. □

Loop-, While-, Goto-Berechenbarkeit

Wir betrachten nun ein weiteres Berechnungsmodell, das im wesentlichen eine einfache Programmiersprache mit verschiedenen Konstrukten darstellt.

- Diese Programme haben Variablen, die mit natürlichen Zahlen belegt sind. Diesen Variablen dürfen arithmetische Ausdrücke (mit Konstanten, Variablen und Operatoren $+$, $-$) zugewiesen werden.
- Außerdem enthalten die Programme verschiedene Schleifenkonstrukte.

Loop-, While-, Goto-Berechenbarkeit

Insbesondere betrachten wir folgende Typen von Programmen:

Loop-Programme

Enthalten nur Loop- bzw. For-Schleifen, bei denen bereits bei Eintritt feststeht, wie oft sie durchlaufen werden.

While-Programme

Enthalten nur While-Schleifen mit einer immer wieder zu evaluierenden Abbruchbedingung.

Goto-Programme

Enthalten Gotos (unbedingte Sprünge) und If-Then-Else-Anweisungen.

Wir interessieren uns vor allem für die Funktionen, die von solchen Programmen berechnet werden.

LOOP-Programme

Syntaktische Komponenten für LOOP-Programme

- Variablen: $x_1, x_2, x_3, \ldots$
- Konstanten: $0, 1, 2 \ldots$
- Trennsymbole: ; und $:=$
- Operatorsymbole: $+$ und $-$
- Schlüsselwörter: LOOP, DO, END

Induktive Syntax-Definition von LOOP-Programmen

Ein LOOP-Programm ist entweder von der Form

- $x_i := x_j + c$ oder $x_i := x_j - c$ mit $c \in \mathbb{N}$ und $i, j \geq 1$ (Wertzuweisung) oder
- $P_1; P_2$, wobei P_1 und P_2 bereits LOOP-Programme sind (sequentielle Komposition) oder
- LOOP x_i DO P END, wobei P ein LOOP-Programm ist und $i \geq 1$.

LOOP-Programme

Informelle Beschreibung der Semantik

- Ein LOOP-Programm, das eine k-stellige Funktion $f : \mathbb{N}^k \to \mathbb{N}$ berechnen soll, startet mit dem Eingabewerten $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$. Alle anderen Variablen haben den Startwert 0. Das Ergebnis $f(n_1, \ldots, n_k)$ liegt bei Termination in x_1.
- Interpretation der Wertzuweisungen:
  - $x_i := x_j + c$ – wie üblich
  - $x_i := x_j - c$ – modifizierte Subtraktion, falls $c > x_j$, so ist das Resultat gleich 0
- Sequentielle Komposition $P_1; P_2$: erst P_1, dann P_2 ausführen.
- LOOP x_i DO P END: das Programm P wird so oft ausgeführt, wie die Variable x_i zu Beginn angibt.

Formale Beschreibung der Semantik

Für jedes LOOP-Programm P, in dem keine Variable x_i mit $i > k$ vorkommt, definieren wir zunächst eine Funktion $[P]_k : \mathbb{N}^k \to \mathbb{N}^k$ wie folgt durch Induktion über den Aufbau von P:

- $[x_i := x_j + c]_k(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ genau dann, wenn (i) $m_\ell = n_\ell$ für $\ell \neq i$ und (ii) $m_i = n_j + c$.
- $[x_i := x_j - c]_k(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ genau dann, wenn (i) $m_\ell = n_\ell$ für $\ell \neq i$ und (ii) $m_i = \max\{0, n_j - c\}$.
- $[P_1; P_2]_k(n_1, \ldots, n_k) = [P_2]_k([P_1]_k(n_1, \ldots, n_k))$
- $[\text{LOOP } x_i \text{ DO } P \text{ END}]_k(n_1, \ldots, n_k) = [P]_k^{n_i}(n_1, \ldots, n_k)$

LOOP-Programme

Sei im folgenden $\pi_i(n_1, \ldots, n_k) = n_i$ (Projektionsfunktion).

LOOP-Berechenbarkeit (Definition)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt LOOP-berechenbar, falls es ein $\ell \geq k$ und ein LOOP-Programm P, in dem nur die Variablen $x_1, \ldots, x_\ell$ vorkommen, gibt mit:

$$\forall n_1, \ldots, n_k \in \mathbb{N} : f(n_1, \ldots, n_k) = \pi_1([P]_\ell(n_1, \ldots, n_k, 0, \ldots, 0)).$$

Alle LOOP-Programme stoppen nach endlicher Zeit
$\rightsquigarrow$ alle LOOP-berechenbaren Funktionen sind total.

Wir werden später zeigen, dass es aber sogar totale Turing-berechenbare Funktionen gibt, die nicht LOOP-berechenbar sind.

$\rightsquigarrow$ LOOP-Berechenbarkeit ist schwächer als Turing-Berechenbarkeit!

LOOP-Programme

LOOP-Programme können aber gewisse Programmkonstrukte simulieren, die in der Syntax nicht enthalten sind.

If-Then-Else

Simulation von IF $x_1 = 0$ THEN A END

$x_2 := 1$; LOOP x_1 DO $x_2 := 0$ END; LOOP x_2 DO A END

Addition

Simulation von $x_i := x_j + x_k$ (sei $i \neq k$)

$x_i := x_j$; LOOP x_k DO $x_i := x_i + 1$ END

Multiplikation

Simulation von $x_i := x_j \cdot x_k$ (sei $k \neq i \neq j$)

$x_i := 0$; LOOP x_k DO $x_i := x_i + x_j$ END

Wir werden im folgenden solche Konstrukte auch in While-Programmen verwenden. Wir nehmen dann an, dass sie – wie oben – geeignet simuliert werden.

Analog: Ganzzahlige Division (x DIV y) und Divisionsrest (x MOD y).

WHILE-Programme

Wir erweitern nun die Syntax von LOOP-Programmen zu WHILE-Programmen, indem wir neben LOOP-Schleifen noch ein weiteres Schleifenkonstrukt erlauben.

Syntax von WHILE-Programmen

Wenn P ein WHILE-Programm ist und $i \geq 1$ gilt, so ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

ein WHILE-Programm.

Alle in LOOP-Programmen erlaubten Konstrukte (siehe Folie 328) sind auch in WHILE-Programmen erlaubt.

Intuition: Programm P wird so lange ausgeführt bis der Wert von x_i gleich 0 ist.

Semantik von WHILE-Programmen

Wie bei LOOP-Programmen definieren wir zunächst für jedes WHILE-Programm P, in dem keine Variable x_i mit $i > k$ vorkommt, eine partielle Abbildung $[P]_k : \mathbb{N}^k \to \mathbb{N}^k$ induktiv.

Für die in LOOP-Programmen verfügbaren Konstrukte übernehmen wir die Definitionen von Folie 329.

Sei nun $P = \text{WHILE } x_i \neq 0 \text{ DO } A \text{ END}$ $(i \leq k)$ und $(n_1, \ldots, n_k) \in \mathbb{N}^k$.

Falls eine Zahl τ mit $\pi_i([A]_k^\tau(n_1, \ldots, n_k)) = 0$ existiert, so sei t die kleinste Zahl mit dieser Eigenschaft. Ansonsten sei t undefiniert.

Dann sei

$$[P]_k(n_1, \ldots, n_k) = \begin{cases} [A]_k^t(n_1, \ldots, n_k) & \text{falls } t \text{ definiert ist} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

WHILE-Programme

Beachte: Eine LOOP-Schleife

LOOP x DO P END

kann simuliert werden durch

$y := x$;
WHILE $y \neq 0$ DO $y := y - 1; P$ END

Wichtig: dabei ist y eine neue Variable, die insbesondere in P nicht vorkommt.

WHILE-Berechenbarkeit (Definition)

Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt WHILE-berechenbar, falls es ein $\ell \geq k$ und ein WHILE-Programm P, in dem nur die Variablen $x_1, \ldots, x_\ell$ vorkommen, gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$ gilt:

- $f(n_1, \ldots, n_k)$ definiert $\iff$ $[P]_\ell(n_1, \ldots, n_k, 0, \ldots, 0)$ definiert
- Falls $f(n_1, \ldots, n_k)$ definiert ist, gilt $f(n_1, \ldots, n_k) = \pi_1([P]_\ell(n_1, \ldots, n_k, 0, \ldots, 0))$.

Satz 10 (WHILE-Programme $\rightarrow$ Turingmaschinen)

Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Anders ausgedrückt: Turingmaschinen können WHILE-Programme simulieren.

Beweisidee:

- Wir verwenden eine Mehrband-Turingmaschine, bei der auf jedem Band eine andere Variable in Binärdarstellung gespeichert wird.
  k Variablen $\rightsquigarrow$ k Bänder
- $x_i := x_j + c$ kann von der Turingmaschine durchgeführt werden, indem die Inkrementierungsfunktion $(+1)$ c-mal ausgeführt wird.
- $x_i := x_j - c$ funktioniert ähnlich.

WHILE-Programme

- Sequentielle Komposition $P_1; P_2$:

  Wir bestimmen Turingmaschinen M_1, M_2 für P_1, P_2.

  Diese machen wir wie folgt zu einer Turingmaschine für $P_1; P_2$:

  - Vereinigung der Zustandsmengen, Bandalphabete und Übergangsfunktionen
  - Anfangszustand ist Anfangszustand von M_1. Endzustände sind Endzustände von M_2.
  - Statt in einen Endzustand von M_1 wird ein Übergang in den Anfangszustand von M_2 gemacht.

    (Vergleiche mit der Konkatenationskonstruktion für endliche Automaten.)

WHILE-Programme

- WHILE-Schleife WHILE $x_i \neq 0$ DO P END:

  Bestimme zunächst eine Turingmaschine M für P.

  Modifiziere M wie folgt:

  Im neuen Anfangszustand wird zunächst überprüft, ob 0 auf dem i-ten Band steht.

  - Falls ja: Übergang in Endzustand
  - Falls nein: M wird ausgeführt

  Statt Übergang in Endzustand: Übergang in den neuen Anfangszustand.

GOTO-Programme

Syntax von GOTO-Programmen

Mögliche Anweisungen für GOTO-Programme:

Wertzuweisung: $x_i := x_j + c$ bzw. $x_i := x_j - c$ (mit $c \in \mathbb{N}$)

Unbedingter Sprung: GOTO M_i

Bedingter Sprung: IF $x_i = c$ THEN GOTO M_i

Stopanweisung: HALT

Ein GOTO-Programm besteht aus einer Folge von Anweisungen A_i, vor denen sich jeweils eine (Sprung-)Marke M_i befindet.

$$M_1 : A_1; M_2 : A_2; \ldots; M_k : A_k$$

(Wenn Marken nicht angesprungen werden, werden wir sie manchmal einfach weglassen.)

GOTO-Programme

Intuitive Semantik von GOTO-Programmen

- IF-Anweisungen werden wie üblich interpretiert.
- GOTO *M* springt an die entsprechende Marke des Programms.
- HALT-Anweisungen beenden GOTO-Programme. (Die letzte Anweisung eines Programms sollte ein HALT oder ein unbedingter Sprung sein.)

Dies ist keine wirklich formale Semantikdefinition. Schreiben Sie als Übung eine formale Semantikdefinition (analog zu WHILE-Programmen) auf.

Wie WHILE-Programme können auch GOTO-Programme in unendliche Schleifen geraten.

GOTO-berechenbare Funktionen sind analog zu WHILE-berechenbaren Funktionen definiert.

Satz 11 (WHILE-Programme → GOTO-Programme)

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden, d. h. jede WHILE-berechenbare Funktion ist GOTO-berechenbar.

Beweis:

Eine WHILE-Schleife

WHILE $x \neq 0$ DO P END

kann simuliert werden durch

M_1: IF $x = 0$ THEN GOTO M_2;
P;
GOTO M_1;
M_2: ...

□

GOTO-Programme

Auch die nicht ganz so offensichtliche Umkehrung gilt:

Satz 12 (GOTO-Programme → WHILE-Programme)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden, d. h. jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Das ist einer der Gründe dafür, warum in modernen Programmiersprachen im allgemeinen keine GOTOS verwendet werden.

Weitere Gründe: Spaghetti-Code bei Verwendung von GOTOS, siehe auch Edsger W. Dijkstra: "Go To Statement Considered Harmful" (1968), `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`.

Beweis (Goto-Programme $\rightarrow$ While-Programme):

Sei $P = (M_1 : A_1; M_2 : A_2; \ldots M_k : A_k)$ ein GOTO-Programm.

Wir simulieren P durch das folgende WHILE-Programm Q mit nur einer WHILE-Schleife:

count := 1;
WHILE count $\neq 0$ DO
    IF count = 1 THEN A'_1 END;
    IF count = 2 THEN A'_2 END;
        $\vdots$
    IF count = k THEN A'_k END
END

GOTO-Programme

Hierbei ist A'_i das folgende WHILE-Programm.

$$A'_i = \begin{cases} x_j := x_\ell \pm c;\ \text{count} := \text{count} + 1 & \text{falls } A_i = (x_j := x_\ell \pm c) \\ \text{count} := n & \text{falls } A_i = (\text{GOTO } M_n) \\ \text{IF } x_j = c \text{ THEN count} := n & \text{falls } A_i = (\text{IF } x_j = c \text{ THEN} \\ \quad \text{ELSE count} := \text{count} + 1 \text{ END} & \quad \text{GOTO } M_n) \\ \text{count} := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

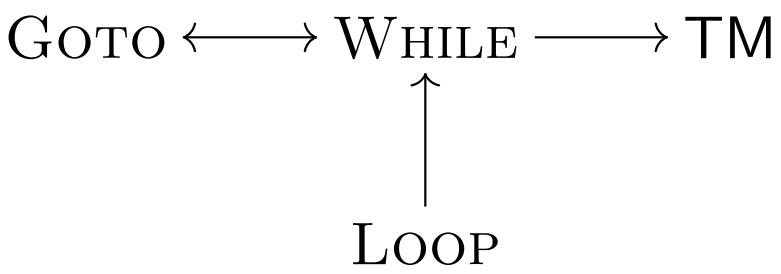
□

Die Simulation von GOTO-Programmen durch WHILE-Programme verwendet nur *eine* WHILE-Schleife (falls man IF THEN ELSE als elementares Konstrukt erlaubt).

Das bedeutet: Ein WHILE-Programm kann durch Umwandlung in ein GOTO-Programm und Zurückumwandlung in ein WHILE-Programm in ein äquivalentes WHILE-Programm mit einer WHILE-Schleife umgewandelt werden (Kleenesche Normalform für WHILE-Programme).

Goto-Programme

Welche Transformationen haben wir bisher durchgeführt?



Um die Äquivalenz von Goto-, While- und Turing-Berechenbarkeit zu zeigen, fehlt uns noch die Richtung

$$\text{TM} \rightarrow \text{Goto}$$

Satz 13 (TM $\rightarrow$ GOTO-Programme)

Jede Turingmaschine kann durch ein GOTO-Programm simuliert werden. Das heißt, jede Turing-berechenbare Funktion ist GOTO-berechenbar.

Beweis:

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, E, \square)$ eine deterministische Turingmaschine, welche eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ berechnet.

O.B.d.A. sei $\Gamma = \{0, \ldots, m-1\}$, $\square = 0$ und $Z = \{0, \ldots, n-1\}$.

Für $a_1 a_2 \cdots a_p \in \Gamma^*$ sei

$$(a_1 a_2 \cdots a_p)_m = \sum_{i=1}^{p} a_i \cdot m^{i-1}$$

der Wert von $a_1 a_2 \cdots a_p$ in der Darstellung zur Basis m (niederwertigste Ziffer ganz links).

GOTO-Programme

Eine Konfiguration $a_1 \cdots a_p z b_1 \cdots b_q$ wird durch das Tripel

$$((a_p \ldots a_1)_m, z, (b_1 \cdots b_q)_m) \in \mathbb{N} \times \{0, \ldots, n-1\} \times \mathbb{N}$$

repräsentiert.

Solch ein Tripel wird im folgenden mit den drei Variablen x, z, y gespeichert.

Simulation von Turingmaschinenoperationen:

Kopf liest Zeichen: $a := y \text{ MOD } m$

Zeichen b aufs Band schreiben: $y := (y \text{ DIV } m) \cdot m + b$

Kopf nach links: $y := y \cdot m + (x \text{ MOD } m);\ x := x \text{ DIV } m$

Kopf nach rechts: $x := x \cdot m + (y \text{ MOD } m);\ y := y \text{ DIV } m$

Sei nun $(n_1, \ldots, n_k) \in \mathbb{N}^k$ eine Eingabetupel.

GOTO-Programme

Das zu erstellende GOTO-Programm besteht aus folgenden drei Programmteilen:

Teil 1: $x := 0$; $z = z_0$; $y := (\text{bin}(n_1)\#\text{bin}(n_2)\#\cdots\#\text{bin}(n_k))_m$.

In $\text{bin}(n_i)$ sei hier die Binärziffer 0 (bzw. 1) durch das Symbol $a \in \Gamma \setminus \{\square\}$ (bzw. $b \in \Gamma \setminus \{\square\}$) repräsentiert. Dies ist notwendig, da $\square$ bereits mit der 0 identifiziert wurde.

Teil 2: Die Turingmaschinenberechnung wird durch Manipulation von x, z, y simuliert bis schließlich $z \in E$ gilt.

Teil 3: Die in y gespeicherte Zahl wird in den eigentlichen Ausgabewert überführt: Hat y den Wert $(a_1 a_2 \cdots a_n)_m$ mit $a_1, \ldots, a_n \in \{a, b\}$, so muss die eindeutige Zahl n mit $\text{bin}(n) = a_1 a_2 \cdots a_n$ berechnet werden.

Teil 1 und Teil 3 bestehen aus einfachen numerischen Transformationen (Übung).

Bemerkung: Nur der 2. Teil hängt von der Überführungsfunktion δ ab.

Schema für Teil 2:

M_2: IF $(z \in E)$ THEN GOTO M_3;
$a := y$ MOD m; (Zeichen einlesen)
IF $(z = 0)$ AND $(a = 0)$ THEN GOTO $M_{0,0}$;
IF $(z = 0)$ AND $(a = 1)$ THEN GOTO $M_{0,1}$;
$\vdots$

$M_{0,0}$: $P_{0,0}$; (Aktion $\delta(0,0)$ ausführen)
GOTO M_2;

$M_{0,1}$: $P_{0,1}$; (Aktion $\delta(0,1)$ ausführen)
GOTO M_2;
$\vdots$

M_3: Führe Teil 3 aus

Im Programmteil $P_{i,j}$ wird die durch $\delta(i,j)$ beschriebene Aktion so wie auf Folie 348 simuliert. □

Primitiv rekursive und μ-rekursive Funktionen

LOOP-, WHILE- und GOTO-Programme sind vereinfachte imperative Programme und stehen für imperative Programmiersprachen, bei denen Programme als Folgen von Befehlen aufgefaßt werden.

Parallel dazu gibt es jedoch auch funktionale Programme, deren Hauptbestandteil die Definition rekursiver Funktionen ist. Es gibt auch Berechnungsbegriffe, die sich eher an funktionalen Programmen orientieren.

Zum Beispiel:

- λ-Kalkül (Alonzo Church, 1932)
- μ-rekursive und primitiv rekursive Funktionen (werden hier in der Vorlesung betrachtet)

Primitiv rekursive und μ-rekursive Funktionen

Wir definieren nun Klassen von Funktionen der Form $f : \mathbb{N}^k \to \mathbb{N}$.

Primitiv rekursive Funktionen (Definition)

Die Klasse der primitiv rekursive Funktionen ist induktiv wie folgt definiert:

- Alle konstanten Funktionen der Form $k_m : \mathbb{N} \to \mathbb{N}$ mit $k_m(n) = m$ (für ein festes $m \in \mathbb{N}$) sind primitiv rekursiv.
- Alle Projektionen der Form $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ mit $\pi_i^k(n_1, \ldots, n_k) = n_i$ sind primitiv rekursiv.
- Die Nachfolgerfunktion $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = n + 1$ ist primitiv rekursiv.
- Wenn $g : \mathbb{N}^k \to \mathbb{N}$ und $f_1, \ldots, f_k : \mathbb{N}^r \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch die Abbildung $f : \mathbb{N}^r \to \mathbb{N}$ mit

$$f(n_1, \ldots, n_r) = g(f_1(n_1, \ldots, n_r), \ldots, f_k(n_1, \ldots, n_r))$$

primitiv rekursiv (Einsetzung/Komposition)

Primitiv rekursive und μ-rekursive Funktionen

Primitiv rekursive Funktionen (Definition, Fortsetzung)

- Jede Funktion f, die durch primitive Rekursion aus primitiv rekursiven Funktionen entsteht ist primitiv rekursiv.

  Das heißt $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ muss folgende Gleichungen erfüllen (für primitiv rekursive Funktionen $g\colon \mathbb{N}^k \to \mathbb{N}$, $h\colon \mathbb{N}^{k+2} \to \mathbb{N}$):

$$\begin{aligned} f(0, n_1, \ldots, n_k) &= g(n_1, \ldots, n_k) \\ f(n+1, n_1, \ldots, n_k) &= h(f(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{aligned}$$

Anschaulich: bei primitiver Rekursion wird die Definition von $f(n+1, \ldots)$ zurückgeführt auf $f(n, \ldots)$. Das bedeutet, dass primitive Rekursion immer terminiert.

$\rightsquigarrow$ Berechnungsmodell analog zu LOOP-Programmen.

Primitiv rekursive und μ-rekursive Funktionen

Beispiele für primitiv rekursive Funktionen:

Additionsfunktion

Die Funktion $add : \mathbb{N}^2 \to \mathbb{N}$ mit $add(n, m) = n + m$ ist primitiv rekursiv.

$$\begin{aligned} add(0, m) &= m \\ add(n + 1, m) &= s(add(n, m)) \end{aligned}$$

Multiplikationsfunktion

Die Funktion $mult : \mathbb{N}^2 \to \mathbb{N}$ mit $mult(n, m) = n \cdot m$ ist primitiv rekursiv.

$$\begin{aligned} mult(0, m) &= 0 \\ mult(n + 1, m) &= add(mult(n, m), m) \end{aligned}$$

Primitiv rekursive und μ-rekursive Funktionen

Dekrementierung

Die Funktion $dec : \mathbb{N} \to \mathbb{N}$ mit $dec(n) = n - 1$ ist primitiv rekursiv.

$$\begin{aligned} dec(0) &= 0 \\ dec(n+1) &= n \end{aligned}$$

Subtraktion

Die Funktion $sub : \mathbb{N}^2 \to \mathbb{N}$ mit $sub(n, m) = \max\{0, n - m\}$ ist primitiv rekursiv.

$$\begin{aligned} sub(n, 0) &= n \\ sub(n, m+1) &= dec(sub(n, m)) \end{aligned}$$

Primitiv rekursive und μ-rekursive Funktionen

Die einstellige Funktion $n \mapsto \binom{n}{2}$ is primitiv rekursiv.

$$\binom{0}{2} = 0$$
$$\binom{n+1}{2} = \binom{n}{2} + n$$

Durch Komposition folgt, dass auch die Abbildung $c : \mathbb{N}^2 \to \mathbb{N}$ mit

$$c(n, m) = n + \binom{n+m+1}{2}$$

primitiv rekursiv ist.

Primitiv rekursive und μ-rekursive Funktionen

Die Abbildung c ist eine Bijektion von $\mathbb{N}^2$ nach $\mathbb{N}$ (Paarungsfunktion).

	$n=0$	1	2	3	4
$m=0$	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Beachte: $c(n,m) = n + \sum_{i=1}^{n+m} i$.

Die Funktion c kann verwendet werden, um beliebige k-Tupel ($k \geq 2$) von natürlichen Zahlen durch eine Zahl zu kodieren:

$$\langle n_1, n_2, \ldots, n_k \rangle = c(n_1, c(n_2, \ldots, c(n_{k-1}, n_k) \ldots))$$

Die Abbildung $(n_1, n_2, \ldots, n_k) \mapsto \langle n_1, n_2, \ldots, n_k \rangle$ ist dann auch primitiv rekursiv (für jedes feste k).

Primitiv rekursive und μ-rekursive Funktionen

Es seien $\ell : \mathbb{N} \to \mathbb{N}$ und $r : \mathbb{N} \to \mathbb{N}$ die eindeutigen Funktionen mit:

$$\forall n, m \in \mathbb{N} : \ell(c(n, m)) = n \text{ und } r(c(n, m)) = m$$

Wir werden nun zeigen, dass die Funktionen ℓ und r ebenfalls primitiv rekursiv sind. Mit diesen lassen sich dann auch primitiv rekursive Dekodierfunktionen für kodierte k-Tupel definieren:

$$\begin{aligned}
d_1(n) &= \ell(n) \\
d_2(n) &= \ell(r(n)) \\
&\vdots \\
d_{k-1}(n) &= \ell(\underbrace{r(r(\cdots r(}_{k-2 \text{ mal}} n) \cdots)) \\
d_k(n) &= \underbrace{r(r(\cdots r(}_{k-1 \text{ mal}} n) \cdots))
\end{aligned}$$

Dann gilt: $d_i(\langle n_1, n_2, \ldots, n_k \rangle) = n_i$.

Primitiv rekursive und μ-rekursive Funktionen

Sei $P : \mathbb{N}^{k+1} \to \{0,1\}$ ein Prädikat. Dann wird die Funktion $q : \mathbb{N}^{k+1} \to \mathbb{N}$ mit

$$q(n, n_1, \ldots, n_k) = \begin{cases} 0 \text{ falls } P(x, n_1, \ldots, n_k) = 0 \text{ für alle } x \in \{0, \ldots, n\} \\ \max\{x \leq n \mid P(x, n_1, \ldots, n_k) = 1\} \text{ sonst} \end{cases}$$

durch Anwendung des beschränkten max-Operators auf P definiert.

Wenn P primitiv rekursiv ist, dann ist auch q primitiv rekursiv.

$$\begin{aligned} q(0, n_1, \ldots, n_k) &= 0 \\ q(n+1, n_1, \ldots, n_k) &= \begin{cases} n+1 & \text{falls } P(n+1, n_1, \ldots, n_k) = 1 \\ q(n, n_1, \ldots, n_k) & \text{sonst} \end{cases} \\ &= q(n, n_1, \ldots, n_k) + \\ &\quad P(n+1, n_1, \ldots, n_k) * (n+1 - q(n, n_1, \ldots, n_k)) \end{aligned}$$

Primitiv rekursive und μ-rekursive Funktionen

Sei $P : \mathbb{N}^{k+1} \to \{0,1\}$ ein Prädikat. Dann wird das Prädikat $Q : \mathbb{N}^{k+1} \to \{0,1\}$ mit

$$Q(n, n_1, \ldots, n_k) = \begin{cases} 1 & \text{falls } \exists x \leq n : P(x, n_1, \ldots, n_k) = 1 \\ 0 & \text{sonst} \end{cases}$$

durch Anwendung des beschränkten Existenzquantors auf P definiert.

Wenn P primitiv rekursiv ist, dann ist auch Q primitiv rekursiv.

$$\begin{aligned} Q(0, n_1, \ldots, n_k) &= P(0, n_1, \ldots, n_k) \\ Q(n+1, n_1, \ldots, n_k) &= P(n+1, n_1, \ldots, n_k) + Q(n, n_1, \ldots, n_k) \\ &\quad - P(n+1, n_1, \ldots, n_k) * Q(n, n_1, \ldots, n_k) \end{aligned}$$

Primitiv rekursive und μ-rekursive Funktionen

Nun können wir zeigen, dass die Umkehrfunktionen ℓ und r von $c : \mathbb{N}^2 \to \mathbb{N}$ primitiv rekursiv sind.

Das Prädikat $C : \mathbb{N}^3 \to \{0, 1\}$ mit "$C(x, y, z) = 1$ g.d.w. $c(x, y) = z$" ist primitiv rekursiv:

$$C(x, y, z) = \Big(1 - (c(x, y) - z)\Big) * \Big(1 - (z - c(x, y))\Big).$$

Deshalb sind auch die Funktionen ℓ' und r' mit

$$\begin{aligned} \ell'(k, m, n) &= \max\{x \leq k \mid \exists y \leq m : C(x, y, n) = 1\} \\ r'(k, m, n) &= \max\{y \leq k \mid \exists x \leq m : C(x, y, n) = 1\} \end{aligned}$$

primitiv rekursiv.

Schließlich gilt:

$$\ell(n) = \ell'(n, n, n) \text{ und } r(n) = r'(n, n, n).$$

Primitiv rekursive und μ-rekursive Funktionen

Satz 14 (Primitiv rekursive Funktionen $\leftrightarrow$ LOOP-berechenbare Funktionen)

Die Klasse der primitiv rekursiven Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein.

Beweis:

Sei zunächst $f : \mathbb{N}^r \to \mathbb{N}$ LOOP-berechenbar.

Also gibt es ein LOOP-Programm P, in dem nur die Variablen $x_1, \ldots, x_k$ ($k \geq r$) vorkommen, mit

$$\forall n_1, \ldots, n_r \in \mathbb{N} : f(n_1, \ldots, n_r) = \pi_1([P]_k(n_1, \ldots, n_r, 0, \ldots, 0)).$$

Durch Induktion über den Aufbau von P definieren wir eine primitiv rekursive Funktion $g_P : \mathbb{N} \to \mathbb{N}$ mit

$$\forall n_1, \ldots, n_k \in \mathbb{N} : g_P(\langle n_1, \ldots, n_k \rangle) = \langle [P]_k(n_1, \ldots, n_k) \rangle.$$

Wegen $f(n_1, \ldots, n_r) = d_1(g_P(\langle n_1, \ldots, n_r, 0, \ldots, 0 \rangle))$ ist dann auch f primitiv rekursiv.

Primitiv rekursive und μ-rekursive Funktionen

Fall 1. $P = (x_i := x_j \pm c)$:

$$g_P(n) = \langle d_1(n), \ldots, d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \ldots, d_k(n) \rangle.$$

Fall 2. $P = (Q; R)$:

$$g_P(n) = g_R(g_Q(n)).$$

Fall 3. $P = (\textsc{Loop}\ x_i\ \textsc{Do}\ Q\ \textsc{End})$:

Definiere zunächst die primitiv rekursive Funktion h durch

$$\begin{aligned} h(0, m) &= m \\ h(n+1, m) &= g_Q(h(n, m)) \end{aligned}$$

Dann gilt also $h(n, m) = g_Q^n(m)$.

Es gilt somit

$$g_P(x) = h(d_i(x), x).$$

Primitiv rekursive und μ-rekursive Funktionen

Sei nun $f : \mathbb{N}^r \to \mathbb{N}$ primitiv rekursiv.

Durch Induktion über den Aufbau von f zeigen wir, dass f LOOP-berechenbar ist.

Fall 1: f ist eine der Basisfunktionen (konstante Funktionen, Projektionen, Nachfolgerfunktion).

Dann ist f offensichtlich LOOP-berechenbar.

Fall 2: Es gibt primitiv rekursive Funktionen $g, f_1, \ldots, f_k$ mit

$$f(n_1, \ldots, n_r) = g(f_1(n_1, \ldots, n_r), \ldots, f_k(n_1, \ldots, n_r)).$$

Nach Induktion sind $g, f_1, \ldots, f_k$ LOOP-berechenbar.

Seien $G, F_1, \ldots, F_k$ LOOP-Programme zur Berechnung von $g, f_1, \ldots, f_k$.

Dann berechnet das folgende LOOP-Programm f:

Primitiv rekursive und μ-rekursive Funktionen

$y_1 := x_1; \ldots; y_r := x_r;$
$F_1;$
$z_1 := x_1;$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_2;$
$z_2 := x_1;$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_3;$
$z_3 := x_1;$
$\vdots$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_k;$
$z_k := x_1;$
$x_1 := z_1; \ldots; x_k := z_k;$
G

Primitiv rekursive und μ-rekursive Funktionen

Fall 3: f entsteht aus g und h durch primitive Rekursion.

Es gibt also primitiv rekursive Funktionen $g\colon \mathbb{N}^{r-1} \to \mathbb{N}$ und $h\colon \mathbb{N}^{r+1} \to \mathbb{N}$ mit

$$\begin{aligned} f(0, n_2, \ldots, n_r) &= g(n_2, \ldots, n_r) \\ f(n_1 + 1, n_2, \ldots, n_r) &= h(f(n_1, n_2, \ldots, n_r), n_1, n_2, \ldots, n_r) \end{aligned}$$

Die Funktion f lässt sich dann durch das folgende (Pseudocode-) LOOP-Programm berechnen:

$y := g(x_2, \ldots, x_r);\ k := 0;$
LOOP x_1 DO
    $y := h(y, k, x_2, \ldots, x_r); k := k + 1$
END

Unter Verwendung von LOOP-Programmen für g und h sowie Zwischenspeicherung ähnlich zu Fall 2 kann dieser Pseudocode in ein LOOP-Programm für f umgesetzt werden. □

Primitiv rekursive und μ-rekursive Funktionen

Wir definieren nun ein weitere Klasse, die gleichmächtig zu WHILE-Programmen, GOTO-Programmen und Turingmaschinen ist.

μ-rekursive Funktionen (Definition)

Die μ-rekursiven Funktionen verwenden die gleichen Basisfunktionen (konstante Funktionen, Projektionen, Nachfolgerfunktion) und Operatoren (Einsetzung, primitive Rekursion) wie primitiv rekursive Funktionen.

Zusätzlich darf noch der μ-Operator verwendet werden.

Der μ-Operator verwandelt eine Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$ in eine Funktion $\mu f : \mathbb{N}^k \to \mathbb{N}$ mit

$$\begin{aligned} \mu f(x_1, \ldots, x_k) \quad = \quad & \min\{n \mid f(n, x_1, \ldots, x_k) = 0 \text{ und} \\ & \forall m < n : f(m, x_1, \ldots, x_k) \text{ definiert}\} \end{aligned}$$

Dabei gilt $\min \emptyset = \textit{undefiniert}$.

Primitiv rekursive und μ-rekursive Funktionen

Intuitive Berechnung von $\mu f(x_1, \ldots, x_k)$:

- Berechne $f(0, x_1, \ldots, x_k)$, $f(1, x_1, \ldots, x_k)$, $\ldots$
- Falls für ein n gilt $f(n, x_1, \ldots, x_k) = 0$, so gib n als Funktionswert zurück.
- Falls $f(m, x_1, \ldots, x_k)$ undefiniert ist (ohne, dass vorher einmal der Funktionswert gleich 0 war), oder der Funktionswert 0 nie erreicht wird: die intuitive Berechnung terminiert nicht.
  In diesem Fall gilt auch $\mu f(x_1, \ldots, x_k) = \min \emptyset = \textit{undefiniert}$.

Analogie zu WHILE-Programmen: es ist nicht klar, ob die Abbruchbedingung jemals erfüllt wird.

Primitiv rekursive und μ-rekursive Funktionen

Durch den μ-Operator können nun auch echt partielle Funktionen erzeugt werden.

Überall undefinierte Funktion

Die Funktion $\Omega\colon \mathbb{N} \to \mathbb{N}$ mit $\Omega(n) = \textit{undefiniert}$ für alle $n \in \mathbb{N}$ ist μ-rekursiv.

Verwende die 2-stellige Funktion $f\colon \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = 1$ für alle x, y. Es gilt $f = k \circ \pi_1^2$, wobei k die konstante 1-stellige Funktion ist, die alles auf 1 abbildet.

Dann gilt $\Omega = \mu f$.

Primitiv rekursive und μ-rekursive Funktionen

Weiteres Beispiel:

Wurzelfunktion

Die Funktion $sqrt : \mathbb{N} \to \mathbb{N}$ mit $sqrt(n) = \lceil \sqrt{n} \rceil$ ist μ-rekursiv.

(Dabei rundet $\lceil \ldots \rceil$ eine reelle Zahl auf die nächstgrößere (oder gleiche) ganze Zahl auf.)

Sei $f(m, n) = n - m \cdot m$. (beachte: die Multiplikationsfunktion und Subtraktionsfunktion sind primitiv rekursiv).

Dann gilt $sqrt = \mu f$.

Diese Funktion ist jedoch auch primitiv rekursiv. Intuition: bei Berechnung von $sqrt(n)$ sind immer höchstens n Iterationen notwendig.)

Primitiv rekursive und μ-rekursive Funktionen

Satz 15

Die Klasse der μ-rekursiven Funktionen stimmt genau mit der Klasse der WHILE-(GOTO-, Turing-)berechenbaren Funktionen überein.

Beweis:

Wir zeigen, dass die Klasse der μ-rekursiven Funktionen mit der Klasse der WHILE-berechenbaren Funktionen übereinstimmt.

Hierfür genügt es den Beweis für "Primitiv rekursive Funktionen $\leftrightarrow$ LOOP-berechenbare Funktionen" um den μ-Operator sowie die WHILE-Schleife zu erweitern.

Sei zunächst $P = (\text{WHILE } x_i \neq 0 \text{ DO } Q \text{ End})$ ein WHILE-Programm.

Wir müssen zeigen, dass die auf Folie 361 definierte Funktion $g_P : \mathbb{N} \to \mathbb{N}$ μ-rekursiv ist.

Nach Induktion ist dies für g_Q bereits der Fall.

Primitiv rekursive und μ-rekursive Funktionen

Wie im Beweis von " LOOP-berechenbare Funktionen $\rightarrow$ Primitiv rekursive Funktionen" (Fall 3) können wir eine μ-rekursive Funktion $h : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $h(n, m) = g_Q^n(m)$ definieren.

Sei $k : \mathbb{N}^2 \rightarrow \mathbb{N}$ definiert durch $k(n, m) = d_i(h(n, m))$.

Dann gilt $g_P(x) = h((\mu k)(x), x)$.

Sei nun $f = \mu g : \mathbb{N}^r \rightarrow \mathbb{N}$ für eine μ-rekursive Funktion $g : \mathbb{N}^{r+1} \rightarrow \mathbb{N}$.

Dann kann f durch das folgende (Pseudocode-) WHILE-Programm berechnet werden:

$y := 0;\ z := g(0, x_1, \ldots, x_r);$
WHILE $z \neq 0$ DO
    $y := y + 1;\ z := g(y, x_1, \ldots, x_r);$
END;
$x_1 := y$

Primitiv rekursive und μ-rekursive Funktionen

Wir werden nun zeigen, dass es totale Turing-berechenbare/μ-rekursive Funktionen gibt, die nicht primitiv rekursiv sind.

Ein klassisches Beispiel hierfür ist die sogenannte Ackermannfunktion.

Ackermannfunktion $a\colon \mathbb{N}^2 \to \mathbb{N}$ (Ackermann 1928)

$$a(0,y) = y+1 \tag{1}$$
$$a(x,0) = a(x-1,1), \text{ falls } x > 0 \tag{2}$$
$$a(x,y) = a(x-1,a(x,y-1)), \text{ falls } x,y > 0 \tag{3}$$

Lemma 16

Die Ackermannfunktion ist eine totale Funktion.

Beweis: Durch Induktion über das erste Argument x.

Seien $x, y \in \mathbb{N}$.

Falls $x = 0$, gilt $a(x, y) \overset{(1)}{=} y + 1$.

Falls $x > 0$, gilt

$$\begin{aligned} a(x,y) &\overset{(3)}{=} a(x-1, a(x, y-1)) \\ &\overset{(3)}{=} a(x-1, a(x-1, a(x, y-2))) = \cdots \\ &= \underbrace{a(x-1, a(x-1, \ldots a(x-1}_{(y+1)\text{-mal}}, 1) \ldots)). \end{aligned}$$

Da nach Induktion alle Werte $a(x-1, z)$ (für $z \in \mathbb{N}$) definiert sind, ist auch $a(x, y)$ definiert. □

Primitiv rekursive und μ-rekursive Funktionen

Wertetabelle der Ackermannfunktion für kleine Werte:

$y =$	0	1	2	3	4	...	$a(x, y)$
$x = 0$	1	2	3	4	5	...	$y + 1$
$x = 1$	2	3	4	5	6	...	$y + 2$
$x = 2$	3	5	7	9	11	...	$2y + 3$
$x = 3$	5	13	29	61	125	...	$2^{y+3} - 3$
$x = 4$	13	65533	$> 10^{19727}$				$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3 \text{ Zweier}} - 3$
...							

Satz 17

Die Ackermannfunktion ist WHILE-berechenbar, aber nicht primitiv rekursiv bzw. nicht LOOP-berechenbar.

Primitiv rekursive und μ-rekursive Funktionen

Beweis:

Wir zeigen zunächst, dass die Ackermannfunktion WHILE-berechenbar ist (die Ackermannfunktion ist sicherlich berechenbar im intuitiven Sinne).

Hierfür ist es sinnvoll, Stacks (Keller) von natürlichen Zahlen einzuführen.

Eine Folge $(n_0, n_1, \ldots, n_k)$ von natürlichen Zahlen kann durch die Kodierungsfunktion $(n_0, n_1, \ldots, n_k) \mapsto \langle n_0, n_1, \ldots, n_k \rangle$ (siehe Folie 95) in einer Zahl abgespeichert werden.

Sei stack eine Integer-Variable, die einen Stack von natürlichen Zahlen repräsentiert. Wir definieren die folgenden Operationen:

- $\text{INIT}(\text{stack})$: $\text{stack} := 0$
- $\text{PUSH}(n, \text{stack})$ für $n \in \mathbb{N}$: $\text{stack} := c(n+1, \text{stack})$
- $x := \text{POP}(\text{stack})$: $x := \ell(\text{stack}) - 1;\ \text{stack} := r(\text{stack})$

Ausserdem verwenden wir $\text{size}(\text{stack}) \neq 1$ als Abkürzung für $r(\text{stack}) \neq 0$.

Primitiv rekursive und μ-rekursive Funktionen

Beachte: Bei einer PUSH-Operation wird das auf den Keller zu legende Argument um 1 inkrementiert, bei einer POP-Operation wird es dann wieder dekrementiert.

Dies ist notwendig, um diese Argumente von dem untersten Kellersymbol 0 zu unterscheiden.

Würden wir diese Inkrementierung nicht vornehmen, so würden alle Keller der Form $(0, 0, \ldots, 0)$ durch die Zahl 0 kodiert werden.

Mit diesen Operationen kann die Ackermannfunktion durch das folgende WHILE-Programm berechnet werden:

Primitiv rekursive und μ-rekursive Funktionen

```
INIT(stack);
PUSH(x_1, stack);
PUSH(x_2, stack);
WHILE size(stack) ≠ 1 DO
    y := POP(stack);
    x := POP(stack);
    IF x = 0 THEN PUSH(y + 1, stack);
    ELSEIF y = 0 THEN PUSH(x − 1, stack); PUSH(1, stack);
    ELSE PUSH(x − 1, stack); PUSH(x, stack); PUSH(y − 1, stack);
    END (if)
END (while)
x_1 := POP(stack);
```

Wir zeigen nun, dass die Ackermannfunktion stärker als jede primitiv rekursive Funktion wächst.

Hierfür beweisen wir eine Reihe von Lemmata.

Lemma 18

$\forall x, y \in \mathbb{N} : y < a(x, y)$

Beweis: Induktion über x.

IA: $x = 0$.

Es gilt $y < y + 1 \stackrel{(1)}{=} a(0, y)$ für alle $y \in \mathbb{N}$.

IS: Gelte $\forall y \in \mathbb{N} : y < a(x, y)$ (IH 1).

Wir zeigen nun durch Induktion über $y \in \mathbb{N}$, dass $\forall y \in \mathbb{N} : y < a(x+1, y)$.

IA: $y = 0$.

Es gilt $1 < a(x, 1)$ (nach (IH 1)) und damit $0 < 1 < a(x, 1) \stackrel{(2)}{=} a(x + 1, 0)$.

IS: Gelte $y < a(x + 1, y)$ (IH 2).

Wir zeigen nun $y + 1 < a(x + 1, y + 1)$.

Zunächst gilt nach (IH 1): $a(x+1,y) < a(x,a(x+1,y))$
(ersetze y in (IH 1) durch $a(x+1,y)$).

Also gilt: $y+1 \overset{\text{(IH 2)}}{\leq} a(x+1,y) < a(x,a(x+1,y)) \overset{(3)}{=} a(x+1,y+1)$. □

Lemma 19

$\forall x,y \in \mathbb{N} : a(x,y) < a(x,y+1)$

Beweis:

Für $x = 0$ gilt $a(0,y) \overset{(1)}{=} y+1 < y+2 \overset{(1)}{=} a(0,y+1)$ für alle $y \in \mathbb{N}$.

Für $x > 0$ gilt $a(x,y) < a(x-1,a(x,y))$ nach Lemma 18
(ersetze in Lemma 18 x durch $x-1$ und y durch $a(x,y)$).

Dies ergibt $a(x,y) < a(x-1,a(x,y)) \overset{(3)}{=} a(x,y+1)$. □

Lemma 20

$\forall x, y \in \mathbb{N} : a(x, y+1) \leq a(x+1, y)$

Beweis: Induktion über y.

IA: $y = 0$.

Es gilt $a(x, 1) \overset{(2)}{=} a(x+1, 0)$ für alle $x \in \mathbb{N}$.

IS: Gelte $\forall x \in \mathbb{N} : a(x, y+1) \leq a(x+1, y)$ (IH).

Wir zeigen $a(x, y+2) \leq a(x+1, y+1)$ für alle $x \in \mathbb{N}$.

Nach Lemma 18 gilt $y + 1 < a(x, y+1)$.

Also gilt $y + 2 \leq a(x, y+1) \overset{(IH)}{\leq} a(x+1, y)$.

Lemma 19 impliziert $a(x, y+2) \leq a(x, a(x+1, y))$.

Also ergibt sich $a(x, y+2) \leq a(x, a(x+1, y)) \overset{(3)}{=} a(x+1, y+1)$. □

Lemma 21

$\forall x, y \in \mathbb{N} : a(x, y) < a(x + 1, y)$

Beweis:

Lemma 19 $\rightsquigarrow$ $a(x, y) < a(x, y + 1)$.

Lemma 20 $\rightsquigarrow$ $a(x, y + 1) \leq a(x + 1, y)$. □

Aus Lemma 19 und Lemma 21 folgt, dass die Funktion a monoton ist:

Wenn $x \leq x'$ und $y \leq y'$ dann $a(x, y) \leq a(x', y')$. Gilt ausserdem noch $x < x'$ oder $y < y'$ so folgt $a(x, y) < a(x', y')$.

Primitiv rekursive und μ-rekursive Funktionen

Sei nun P ein LOOP Programm, in dem keine Variable x_i mit $i > k$ vorkommt.

Für ein Tupel $(n_1, \ldots, n_k) \in \mathbb{N}^k$ schreiben wir im folgenden

$$\sum(n_1, \ldots, n_k) = n_1 + \cdots + n_k.$$

Dann definieren wir

$$f_P(n) = \max\{\sum[P]_k(n_1, \ldots, n_k) \mid n_1, \ldots, n_k \in \mathbb{N}, \sum(n_1, \ldots, n_k) \leq n\}.$$

Lemma 22

Für jedes LOOP Programm P gibt es eine Zahl ℓ, so dass für alle $n \in \mathbb{N}$ gilt: $f_P(n) < a(\ell, n)$.

Beweis: Induktion über den Aufbau von P.

Primitiv rekursive und μ-rekursive Funktionen

O.B.d.A. erfülle P die folgenden Eigenschaften:

- Für jedes Teilprogramm $x_i := x_j \pm c$ in P gilt $c = 1$.

  Z. B. kann $x_i := x_j + 2$ ersetzt werden durch $x_i := x_j + 1; x_i := x_i + 1$.

- Für jedes Teilprogramm $\text{LOOP } x_i \text{ DO } Q \text{ END}$ in P gilt:
  x_i kommt in Q nicht vor.

  Sollte x_i in Q vorkommen, so ersetzen wir $\text{LOOP } x_i \text{ DO } Q \text{ END}$ durch $y := x_i;\ \text{LOOP } y \text{ DO } Q \text{ END}$, wobei y eine neue Variable ist.

Fall 1: $P = (x_i := x_j \pm 1)$

Dann gilt $f_P(n) \leq 2n + 1$.

Mit Induktion über y kann man leicht zeigen:
$a(1, y) = y + 2$ und $a(2, y) = 2y + 3$ (Übung).

Also gilt $f_P(n) < a(2, n)$.

Fall 2: $P = (P_1; P_2)$.

Nach Induktionsvoraussetzung gibt es Zahlen ℓ_1 und ℓ_2 mit

$$\forall n \in \mathbb{N} : f_{P_1}(n) < a(\ell_1, n) \text{ und } f_{P_2}(n) < a(\ell_2, n). \tag{4}$$

Wir zeigen zunächst $f_P(n) \leq f_{P_2}(f_{P_1}(n))$.

Nach Definition von $f_P(n)$ gibt es ein Tupel $(n_1, \ldots, n_k) \in \mathbb{N}^k$ mit $\sum(n_1, \ldots, n_k) \leq n$ und

$$f_P(n) = \sum [P]_k(n_1, \ldots, n_k) = \sum [P_2]_k([P_1]_k(n_1, \ldots, n_k)).$$

Nun gilt nach Definition von $f_{P_1}(n)$: $\sum [P_1]_k(n_1, \ldots, n_k) \leq f_{P_1}(n)$.

Mit der Definition von $f_{P_2}(n)$ folgt schließlich:

$$f_P(n) = \sum [P_2]_k([P_1]_k(n_1, \ldots, n_k)) \leq f_{P_2}(f_{P_1}(n)).$$

Mit $\ell_3 := \max\{\ell_1 - 1, \ell_2\}$ folgt nun:

$$\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(\ell_2, a(\ell_1, n)) && \text{((4) und Monotonie von } a\text{)} \\
&\leq a(\ell_3, a(\ell_3 + 1, n)) && \text{(Monotonie von } a\text{)} \\
&= a(\ell_3 + 1, n + 1) && \text{(Definition von } a\text{)} \\
&\leq a(\ell_3 + 2, n) && \text{(Lemma 20)}
\end{aligned}$$

Wir können somit $\ell = \ell_3 + 2$ in Lemma 22 wählen.

Fall 3: $P = (\textsc{Loop}\ x_i\ \textsc{Do}\ Q\ \textsc{End})$

Nach Induktionsvoraussetzung gibt es eine Zahl ℓ_1 mit

$$\forall n \in \mathbb{N} : f_Q(n) < a(\ell_1, n). \tag{5}$$

Beachte: x_i kommt in Q nicht vor.

Primitiv rekursive und μ-rekursive Funktionen

Wähle $m, n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_k \leq n$ so, dass gilt:

$$\begin{aligned} f_P(n) &= \max\{\sum[P]_k(n_1, \ldots, n_k) \mid n_1, \ldots, n_k \in \mathbb{N}, \sum(n_1, \ldots, n_k) \leq n\} \\ &= \sum[P]_k(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k), \end{aligned}$$

wobei $n_1 + \cdots + n_{i-1} + n_{i+1} + \cdots + n_k + m \leq n$.

Falls $m = 0$ gilt, so ergibt sich

$$\begin{aligned} f_P(n) &= \sum[P]_k(n_1, \ldots, n_{i-1}, 0, n_{i+1}, \ldots, n_k) \\ &= \sum(n_1, \ldots, n_{i-1}, 0, n_{i+1}, \ldots, n_k) \\ &\leq n \\ &< n + 1 \\ &= a(0, n). \end{aligned}$$

Primitiv rekursive und μ-rekursive Funktionen

Falls $m \geq 1$, so erhalten wir (da x_i in Q nicht vorkommt):

$$
\begin{aligned}
f_P(n) &= \sum [P]_k(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k) \\
&= \sum [Q]_k^m(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k) \\
&\leq \underbrace{f_Q(f_Q(\cdots f_Q}_{m\text{-mal}}(n-m)\cdots)) + m \\
&< a(\ell_1, \underbrace{f_Q(f_Q(\cdots f_Q}_{m-1\text{-mal}}(n-m)\cdots))) + m \\
&\vdots \\
&< \underbrace{a(\ell_1, a(\ell_1, \cdots a(\ell_1}_{m\text{-mal}}, n-m)\cdots)) + m
\end{aligned}
$$

Primitiv rekursive und μ-rekursive Funktionen

Da in dieser Abschätzung m-mal "$<$" vorkommt, erhalten wir:

$$
\begin{aligned}
f_P(n) &\leq \underbrace{a(\ell_1, a(\ell_1, \cdots a(\ell_1,}_{m\text{-mal}} n - m) \cdots)) \\
&< \underbrace{a(\ell_1, \cdots a(\ell_1,}_{m-1\text{-mal}} a(\ell_1 + 1, n - m)) \cdots) && \text{(Monotonie von } a\text{, } m \geq 1) \\
&= a(\ell_1 + 1, n - 1) && \text{(Definition von } a) \\
&< a(\ell_1 + 1, n) && \text{(Monotonie von } a)
\end{aligned}
$$

Wir können somit $\ell = \ell_1 + 1$ in Lemma 22 wählen. □

Primitiv rekursive und μ-rekursive Funktionen

Wir können nun den Beweis von Satz 17 endlich beenden.

Angenommen die Ackermannfunktion a wäre LOOP-berechenbar.

Dann ist auch die Funktion $g : \mathbb{N} \to \mathbb{N}$ mit $g(n) = a(n, n)$ LOOP-berechenbar.

Sei P ein LOOP-Programm mit

$$\forall n \in \mathbb{N} : g(n) = \pi_0([P]_k(n, 0, \ldots, 0)).$$

Nach Lemma 22 existiert eine Konstante ℓ mit

$$\forall n \in \mathbb{N} : f_P(n) < a(\ell, n).$$

Für $n = \ell$ folgt dann

$$g(\ell) = \pi_0([P]_k(\ell, 0, \ldots, 0)) \leq f_P(\ell) < a(\ell, \ell) = g(\ell).$$

Dies ist ein Widerspruch. □

Überblick

- Zunächst einmal definieren wir formal den Begriff Entscheidbarkeit.
  Was bedeutet es überhaupt, dass ein Problem entscheidbar ist?
- Dann kommen wir zum Begriff Semi-Entscheidbarkeit.
  Hier wird erlaubt, dass das Entscheidungsverfahren bei einer negativen Antwort nicht terminiert und keine Antwort liefert.
- Anschließend geht es um negative Resultate.
  Wie kann man zeigen, dass ein Problem nicht entscheidbar ist?

Entscheidbarkeit

Entscheidbarkeit (Definition)

Eine Sprache $A \subseteq \Sigma^*$ heißt entscheidbar, falls die charakteristische Funktion von A, d.h. die Funktion $\chi_A \colon \Sigma^* \to \{0,1\}$ mit

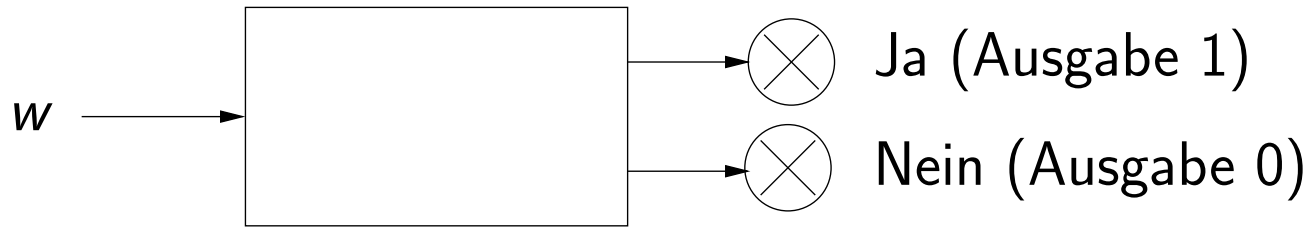
$$\chi_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ 0 & \text{falls } w \notin A \end{cases}$$

berechenbar ist.

Eine Sprache, die nicht entscheidbar ist, heißt unentscheidbar.

Entscheidbarkeit

Darstellung der Entscheidbarkeit an einem Maschinenmodell:



Bei jeder Eingabe rechnet die Maschine endliche Zeit und gibt dann entweder “Ja” oder “Nein” aus.

Entscheidbarkeit

Bei Semi-Entscheidbarkeit erlaubt man, dass die charakteristische Funktion im negativen Fall undefiniert ist, d.h., keine Antwort zurückkommt. Bei konkreten Berechnungsmodellen bedeutet das dann Nicht-Terminierung.

Semi-Entscheidbarkeit (Definition)

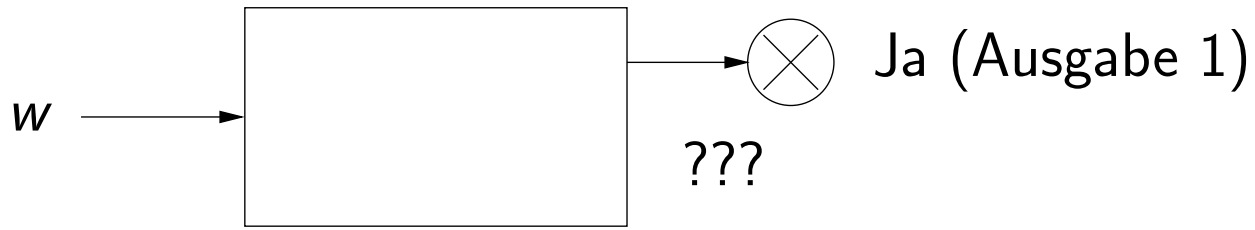
Eine Sprache $A \subseteq \Sigma^*$ heißt semi-entscheidbar, falls die "halbe" charakteristische Funktion von A, d.h. die Funktion $\chi'_A \colon \Sigma^* \to \{1\}$ mit

$$\chi'_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ \textit{undefiniert} & \text{falls } w \notin A \end{cases}$$

berechenbar ist.

Entscheidbarkeit

Darstellung der Semi-Entscheidbarkeit an einem Maschinenmodell:



Bei jeder Eingabe rechnet die Maschine und gibt im Fall $w \in A$ nach endlicher Zeit "Ja" aus. Falls $w \notin A$ gilt, so terminiert die Maschine nie.

Das heißt, man kann sich nie sicher sein, ob nicht doch irgendwann "Ja" ausgegeben wird, da die Antwortzeit der Maschine nicht beschränkt ist.

Satz 23 (Semi-Entscheidbarkeit und Chomsky-0-Sprachen)

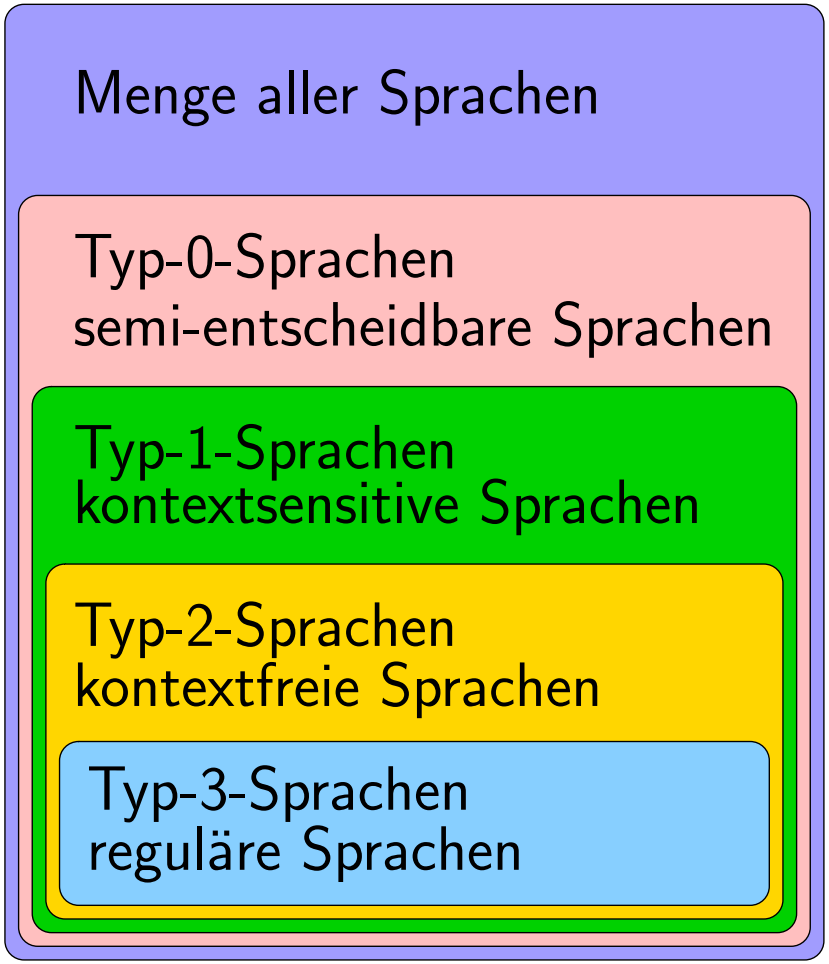
Eine Sprache A ist semi-entscheidbar genau dann, wenn sie vom Typ 0 ist.

Beweis: Die Chomsky-0-Sprachen sind nach dem Satz "Turingmaschinen und Chomsky-0-Sprachen" (Folie 301) genau die Sprachen, die von einer Turing-Maschine akzeptiert werden.

Eine Turing-Maschine, die die halbe charakteristische Funktion χ'_A berechnet, akzeptiert auch die Sprache A, da sie nach Schreiben der 1 in einen Endzustand übergeht.

Eine Turing-Maschine, die eine Sprache A akzeptiert, kann leicht in eine Turing-Maschine umgewandelt werden, die die "halbe" charakteristische Funktion χ'_A berechnet, indem sie bei jeden Übergang in einen Endzustand in die Lage versetzt wird, das Band zu löschen und eine 1 zu schreiben. □

Zur Erinnerung: die Chomsky-Hierarchie



Bemerkungen:

- Im Zusammenhang mit Fragestellungen der Entscheidbarkeit werden Sprachen oft auch als Probleme bezeichnet.
- Auch wenn charakteristische Funktionen Wörter als Argumente haben, kann man sie leicht als Funktionen über natürlichen Zahlen auffassen und so mit WHILE- bzw. GOTO-Programmen berechnen. Jedes Wort aus Σ^* kann als Zahl zur Basis b aufgefasst werden, wobei $b \geq |\Sigma|$. (Siehe auch die Umwandlung "Turing-Maschinen $\rightarrow$ GOTO-Programme".)
- Daher werden wir als Probleme im folgenden auch Teilmengen von $\mathbb{N}$ bzw. $\mathbb{N}^k$ betrachten.

Entscheidbarkeit

Typische Beispiele für Probleme:

Beispiel 1: das Wortproblem

Gegeben sei eine feste Chomsky-Grammatik G und das Problem sei $A = L(G)$. Wir wissen bereits, dass A entscheidbar ist, falls G eine Chomsky-1-Grammatik ist. Außerdem gibt es Grammatiken, für die $L(G)$ nicht entscheidbar ist (noch ohne Beweis).

Beispiel 2: das allgemeine Wortproblem

Das allgemeine Wortproblem ist die Menge

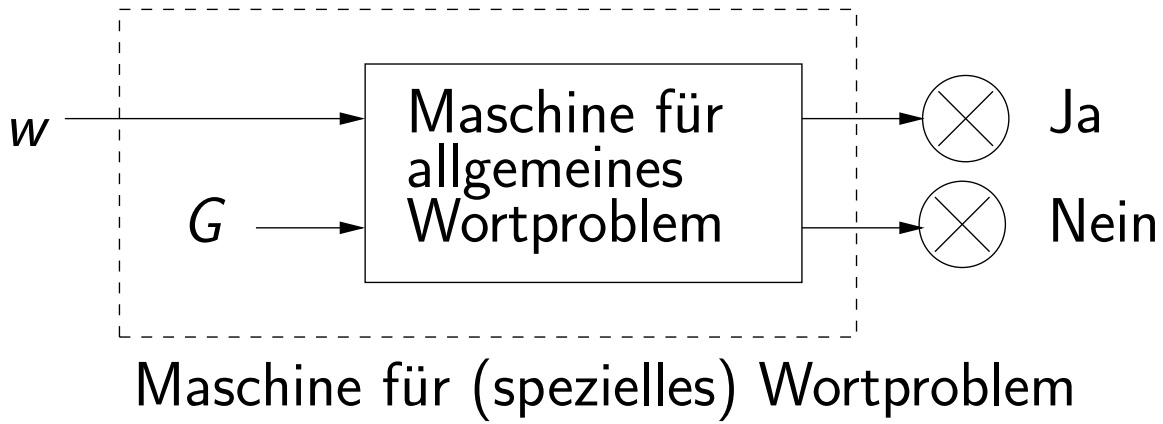
$$A = \{(w, G) \mid w \in L(G),\ G \text{ Chomsky-Grammatik über dem Alphabet } \Sigma\},$$

wobei die Paare (w, G) geeignet als Zeichenketten kodiert werden müssen.

Satz 24 (Unentscheidbarkeit des allgemeinen Wortproblems)

Das allgemeine Wortproblem ist unentscheidbar.

Beweis: Sei G eine Grammatik, für die das Wortproblem $L(G)$ unentscheidbar ist. Falls das allgemeine Wortproblem A entscheidbar wäre, so wäre auch $L(G)$ entscheidbar. Für ein gegebenes Wort w müßte man dann nur überprüfen, ob $(w, G) \in A$ gilt. $\rightsquigarrow$ Widerspruch!



Das heißt, aus einer Maschine zur Lösung des allgemeinen Wortproblems könnte man eine Maschine zur Lösung des (speziellen) Wortproblems bauen. Da es letztere aber nicht für alle Grammatiken G gibt, kann es auch erstere nicht geben.

Argumentationen dieser Art ("wenn es ein Verfahren für A gibt, dann kann man daraus ein Verfahren für B konstruieren") bezeichnet man als Reduktionen. Wir werden sie im folgenden häufiger anwenden.

Beispiel 3: das Schnittproblem

Das Schnittproblem für kontextfreie Grammatiken ist die Menge

$$A = \{(G_1, G_2) \mid G_1, G_2 \text{ kontextfreie Grammatiken}, \\ L(G_1) \cap L(G_2) \neq \emptyset\}.$$

Das Schnittproblem ist unentscheidbar (noch ohne Beweis).

Satz 25 (Entscheidbarkeit und Semi-Entscheidbarkeit)

Eine Sprache A ist entscheidbar, genau dann, wenn sowohl A als auch $\overline{A}$ (das Komplement von A) semi-entscheidbar sind.

Beweis:

Sei zunächst A entscheidbar.

Dann ist also die charakterische Funktion χ_A berechenbar mittels einer Turing-Maschine M.

Die folgende Turing-Maschine M_A berechnet die halbe charakteristische Funktion χ'_A:

- M_A simuliert die Maschine M.
- Falls M mit der Ausgabe 0 terminiert, geht jedoch M_A in eine Endlosschleife über.

Entscheidbarkeit

Die folgende Turing-Maschine $M_{\overline{A}}$ berechnet die halbe charakteristische Funktion $\chi'_{\overline{A}}$:

- $M_{\overline{A}}$ simuliert die Maschine M.
- Falls M mit der Ausgabe 1 terminiert, geht jedoch $M_{\overline{A}}$ in eine Endlosschleife über.
- Falls M mit der Ausgabe 0 terminiert, so terminiert $M_{\overline{A}}$ mit der Ausgabe 1.

Also sind sowohl A als auch $\overline{A}$ semi-entscheidbar.

Sei nun sowohl A als auch $\overline{A}$ semi-entscheidbar.

Sei M_A (bzw. $M_{\overline{A}}$) eine Turing-Maschine, die die halbe charakteristische Funktion χ'_A (bzw. $\chi'_{\overline{A}}$) berechnet.

Der folgende Algorithmus berechnet dann die charakteristische Funktion von A:

INPUT w
$t := 1$;
WHILE true DO
  IF M_A terminiert bei Eingabe w nach t Schritten THEN
    OUTPUT(1)
  ELSEIF $M_{\overline{A}}$ terminiert bei Eingabe w nach t Schritten THEN
    OUTPUT(0)
  END
  $t := t + 1$
END

Beachte: Die WHILE-Schleife wird stets nach endlich vielen Schritten terminieren, da entweder $w \in A$ oder $w \notin A$ gilt.

Also gibt es eine Zahl t, so dass entweder M_A oder $M_{\overline{A}}$ bei Eingabe w nach t Schritten terminiert. □

Rekursive Aufzählbarkeit (Definition)

Eine Sprache $A \subseteq \Sigma^*$ heißt rekursiv aufzählbar, falls $A = \emptyset$ oder es gibt eine totale und berechenbare Funktion $f : \mathbb{N} \to \Sigma^*$ mit

$$A = \{f(n) \mid n \in \mathbb{N}\} = \{f(1), f(2), f(3), \ldots\}.$$

Bemerkungen:

- Sprechweise: die Funktion f zählt die Sprache A auf.
- Äquivalente Definition von rekursiver Aufzählbarkeit: es gibt eine totale, berechenbare und *surjektive* Funktion $f : \mathbb{N} \to A$.
- Der mathematische Begriff der Abzählbarkeit ist ganz ähnlich definiert. Nur fordert man dort nicht, dass f berechenbar ist. Daraus folgt:

  A rekursiv aufzählbar $\Rightarrow$ A abzählbar.

Satz 26 (Rekursive Aufzählbarkeit und Semi-Entscheidbarkeit)

Eine Sprache A ist rekursiv aufzählbar, genau dann, wenn sie semi-entscheidbar ist.

Beweis:

Rekursive Aufzählbarkeit $\longrightarrow$ Semi-Entscheidbarkeit:

Gegeben: rekursiv aufzählbare Sprache $L \subseteq \Sigma^*$, beschrieben durch eine berechenbare und totale Funktion f.

Gesucht: TM, die bestimmt, ob ein Wort $w \in \Sigma^*$ in L liegt.

Lösung: Berechne $f(1), f(2), f(3), \ldots$, solange bis $w = f(i)$ für ein i gilt. In diesem Fall gibt die TM 1 aus, ansonsten terminiert sie nicht.

Entscheidbarkeit

Semi-Entscheidbarkeit $\rightarrow$ Rekursive Aufzählbarkeit:

Gegeben: semi-entscheidbare Sprache $L \subseteq \Sigma^*$, beschrieben durch eine deterministische TM $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ mit $L = T(M)$.

Gesucht: TM M', die eine Funktion f mit $\{f(1), f(2), f(3), \ldots\} = L$ berechnet.

O.B.d.A. sei L unendlich (sonst kann f als schließlich konstant und somit berechenbar gewählt werden).

Erinnern Sie sich, wie wir im Beweis des Satzes "Determinismus und Nichtdeterminismus bei Turingmaschinen" (Folie 304) erfolgreiche Berechnungen von M durch Wörter über dem Alphabet $\Omega = Z \cup \Gamma \cup \{\#\}$ beschrieben haben.

Da M deterministisch ist, gibt es zu jedem Wort $w \in T(M)$ genau eine erfolgreiche Berechnung $c(w)$.

M' arbeitet nun wie folgt:

INPUT: $i \in \mathbb{N}$
$c := \varepsilon$,
$j := 0$;
WHILE true DO
  $c :=$ das längenlexikographisch nach c kommende Wort aus Ω^*;
  IF c ist eine erfolgreiche Berechnung THEN
    $j := j + 1$
    IF $j = i$ THEN OUTPUT w falls $c = c(w)$
  END
END

Begründung der Korrektheit:

- Sei $c(w_1), c(w_2), \ldots$ eine längenlexikographische Auflistung aller erfolgreichen Berechnungen von M.
- Dann ist die von M' berechnete Funktion: $i \mapsto w_i$. □

Entscheidbarkeit

Aus den Sätzen von Folie 301, 395 und 406 folgt, dass die folgenden Aussagen für eine Sprache A äquivalent sind.

Semi-Entscheidbarkeit und äquivalente Begriffe

- A ist semi-entscheidbar, d. h. χ'_A ist (Turing, WHILE-, GOTO-)berechenbar.
- A ist rekursiv aufzählbar
- A ist vom Typ 0.
- $A = T(M)$ für eine Turing-Maschine M.

Halteproblem/Kodierung von Turing-Maschinen

Unser Ziel ist es nun zu zeigen, dass das sogenannte Halteproblem unentscheidbar ist.

Halteproblem (informell)

- **Eingabe:** deterministische Turing-Maschine M mit Eingabe w.
- **Ausgabe:** Hält M auf w?

Dazu werden wir jedoch zunächst genauer definieren, wie eine Turing-Maschine kodiert werden kann, um als Eingabe einer berechenbaren (charakteristischen) Funktion verwendet zu werden.

Halteproblem/Kodierung von Turing-Maschinen

Ziel: Kodierung einer deterministischen Turing-Maschine $M = (Z, \{0,1\}, \Gamma, \delta, z_0, \Box, E)$ durch ein Wort über dem Alphabet $\{0,1\}$.
O.B.d.A. können wir annehmen:

$$\begin{aligned} \Gamma &= \{1, \ldots, m\} \text{ wobei } \Box = 1, \\ Z &= \{1, \ldots, n\} \text{ wobei } z_0 = 1 \text{ und} \\ E &= \{k+1, \ldots, m\} \end{aligned}$$

Der wichtigste Teil von M ist die Überführungsfunktion δ. Diese kann folgendermaßen kodiert werden:
Für alle $1 \leq i \leq k$ und $1 \leq j \leq m$ definieren wir das Wort $w_{i,j}$ wie folgt:
Sei $\delta(i,j) = (i', j', y)$. Dann ist

$$w_{i,j} = 1^i 0 1^j 0 1^{i'} 0 1^{j'} 0 1^{\text{code}(y)} 0 \in \{0,1\}^*.$$

Dabei ist $\text{code}(y) = \begin{cases} 1 & \text{falls } y = L \\ 2 & \text{falls } y = R \\ 3 & \text{falls } y = N \end{cases}$

Halteproblem/Kodierung von Turing-Maschinen

Dann kann die deterministischen Turing-Maschine M durch das folgende Wort kodiert werden:

$$\text{code}(M) = 1^n 0 1^m 0 1^k 0 \prod_{1 \leq i \leq k} \prod_{1 \leq j \leq m} w_{i,j}$$

Bemerkungen: Viele der Festlegungen bei dieser Kodierung sind hochgradig willkürlich. Es gibt viele andere Möglichkeiten, Turing-Maschinen zu kodieren. Wichtig ist hier nur, dass es eine mögliche Kodierung gibt.

Dekodierung: Sei $w \in \{0,1\}^*$ und $\widehat{M}$ eine beliebige aber feste deterministische Turing-Maschine mit Eingabealphabet $\{0,1\}$. Dann sei

$$M_w = \begin{cases} M & \text{falls code}(M) = w \\ \widehat{M} & \text{falls kein Turing-Maschine } M \text{ mit code}(M) = w \text{ existiert} \end{cases}$$

Halteproblem/Kodierung von Turing-Maschinen

Damit können wir nun zwei verschiedene Varianten des Halteproblems definieren.

Halteproblem

Das (allgemeine) Halteproblem ist die Sprache

$$\begin{aligned} H &= \{w\#x \mid w, x \in \{0,1\}^*, M_w \text{ angesetzt auf } x \text{ hält}\} \\ &= \{w\#x \mid w, x \in \{0,1\}^*, x \in T(M_w)\} \end{aligned}$$

Spezielles Halteproblem

Das spezielle Halteproblem ist die Sprache

$$K = \{w \in \{0,1\}^* \mid M_w \text{ angesetzt auf } w \text{ hält}\}.$$

Man beachte die Selbstbezüglichkeit! Hier wird eine Turing-Maschine auf ihre eigene Kodierung angesetzt.

Universelle Turing-Maschine

Einige der folgenden Ergebnisse beruhen darauf, dass es eine deterministische Turing-Maschine gibt, die eine andere Turing-Maschine simulieren kann, wenn deren Kodierung gegeben ist. So eine Turing-Maschine heißt auch universelle Turing-Maschine.

Universelle Turing-Maschine

Eine deterministische Turing-Maschine U heißt universelle Turing-Maschine, wenn sie sich bei Eingabe von $w\#x$ wie folgt verhält:

- Wenn M_w bei Eingabe x nicht hält, so hält U bei Eingabe $w\#x$ auch nicht.
- Wenn M_w bei Eingabe x mit der Ausgabe y hält, so hält U bei Eingabe $w\#x$ ebenfalls mit y.

Im folgenden sei U eine universelle Turing-Maschine.

Unentscheidbarkeit des Halteproblems

Satz 27 (Unentscheidbarkeit des Halteproblems)

Das spezielle Halteproblem K ist nicht entscheidbar.

Beweis:

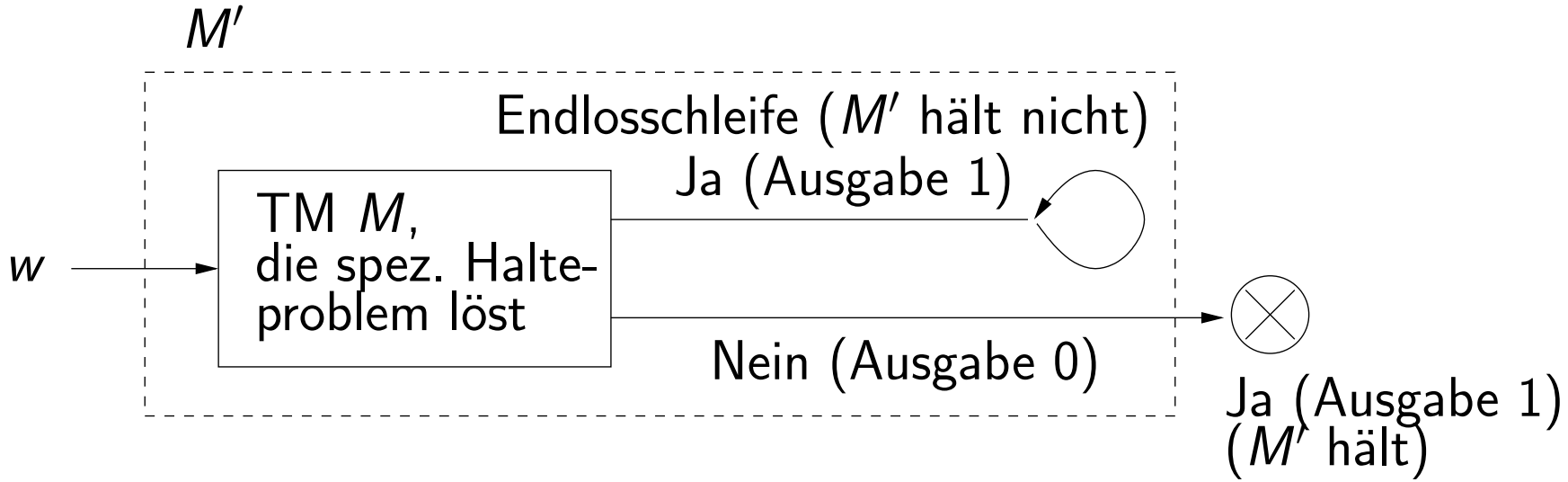
Angenommen das spezielle Halteproblem K ist entscheidbar, d. h. die charakteristische Funktion $\chi_K : \{0,1\}^* \to \{0,1\}$ wird durch eine Turing-Maschine M berechnet.

Dann können wir eine Turing-Maschine M' konstruieren, die sich wie folgt verhält:

INPUT $w \in \{0,1\}^*$
Simuliere M auf Eingabe w
IF $\chi_K(w) = 0$ THEN OUTPUT(1)
ELSE Gehe in Endlosschleife über
END

Unentscheidbarkeit des Halteproblems

Veranschaulichung der Turing-Maschine M':



Sei $w' \in \{0,1\}^*$ so, dass $M' = M_{w'}$.

Dann erhalten wir den folgenden Widerspruch:

$$
\begin{aligned}
M' = M_{w'} \text{ hält bei Eingabe } w' &\iff M \text{ gibt 0 bei Eingabe } w' \text{ aus} \\
&\iff \chi_K(w') = 0 \\
&\iff w' \notin K \\
&\iff M_{w'} \text{ hält bei Eingabe } w' \text{ nicht}
\end{aligned}
$$

Unentscheidbarkeit des Halteproblems

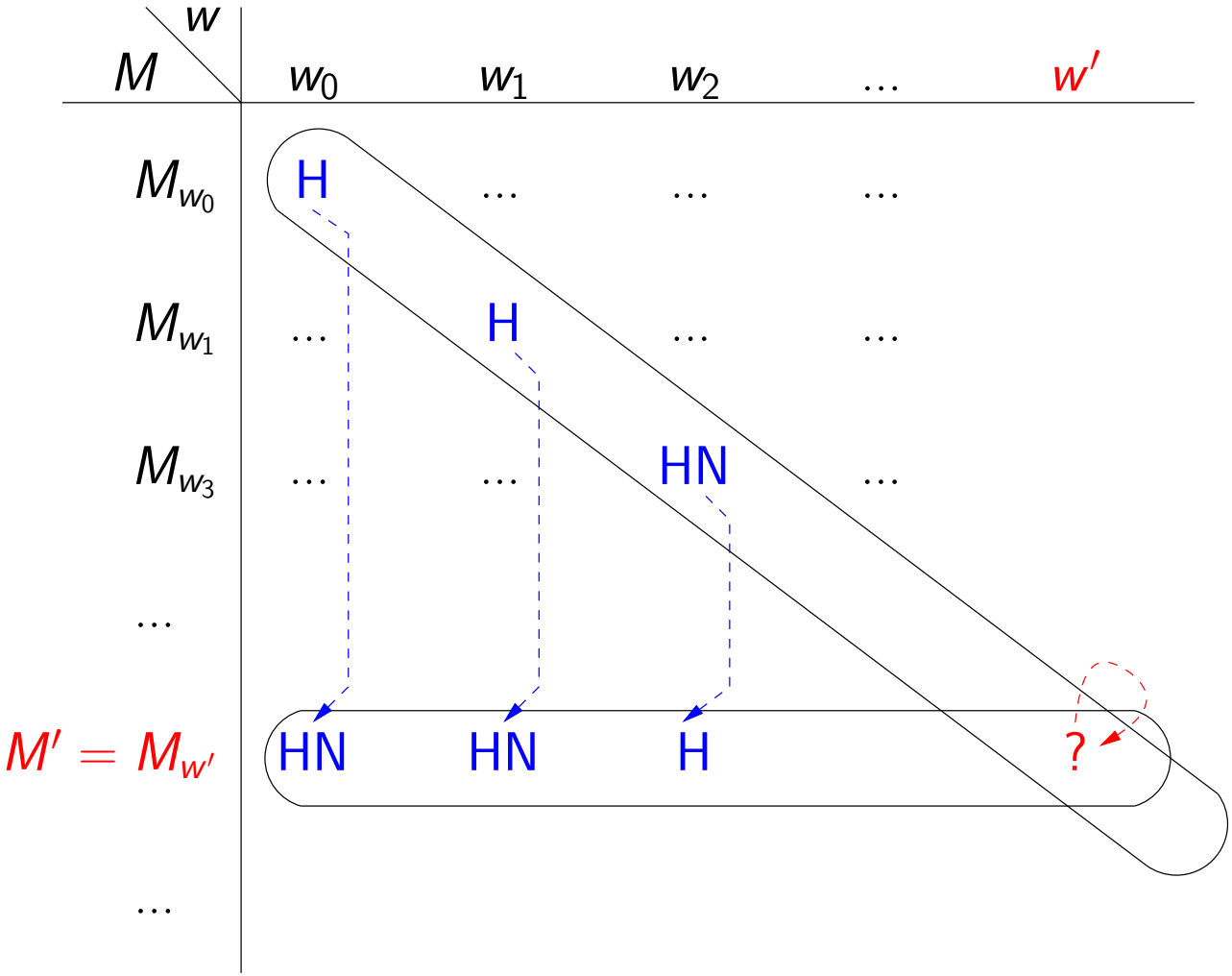
Es handelt sich hierbei um einen Diagonalisierungsbeweis:

Sei $w_0, w_1, w_2, \ldots$ eine Aufzählung aller Wörter aus $\{0,1\}^*$, beispielsweise $w_0 = \varepsilon$, $w_1 = 0$, $w_2 = 1$, $w_3 = 00$, $\ldots$

Wir tragen in eine Tabelle folgendes ein: "Wie verhält sich M_{w_i} auf w_j?"

Es gibt zwei Möglichkeiten: H (M_{w_i} hält) oder HN (M_{w_i} hält nicht).

Unentscheidbarkeit des Halteproblems



Unentscheidbarkeit des Halteproblems

Die konstruierte Turing-Maschine M' und ein w' mit $M' = M_{w'}$ sind ebenfalls in die Tabelle eingetragen.

Aufgrund der Konstruktion von M': die Felder in der Diagonalen bedingen die Felder in der Zeile von M'.

Problem: nichts passt an die Stelle des Fragezeichens!

$\rightsquigarrow$ es kann keine Turing-Maschine geben, die das Halteproblem löst. $\square$

Unentscheidbarkeit des Halteproblems

Satz 28 (Semi-Entscheidbarkeit des speziellen Halteproblems)

Das spezielle Halteproblem K ist semi-entscheidbar.

Beweis:

Die "halbe" charakteristische Funktion $\chi'_K : \{0,1\}^* \to \{1\}$ kann wie folgt berechnet werden:

- Bei Eingabe w starten wir die universelle Turing-Maschine U mit der Eingabe $w\#w$.
- Falls U bei Eingabe $w\#w$ terminiert, wird die produzierte Ausgabe durch 1 überschrieben. □

Reduktionen

Wir haben nun die Unentscheidbarkeit eines Problems, des speziellen Halteproblems, nachgewiesen.

Daraus sollen weitere Unentscheidbarkeitsresultate gewonnen werden.

Dies erfolgt mit Argumentationen folgender Art:

1. Wenn man Problem B lösen könnte, dann könnte man auch A lösen. (Reduktionsschritt)
2. Daraus folgt, dass B schwieriger bzw. allgemeiner ist als A ($A \leq B$).
3. Wir wissen jedoch bereits, dass A unentscheidbar ist.
4. Also muss das schwierigere Problem B auch unentscheidbar sein.

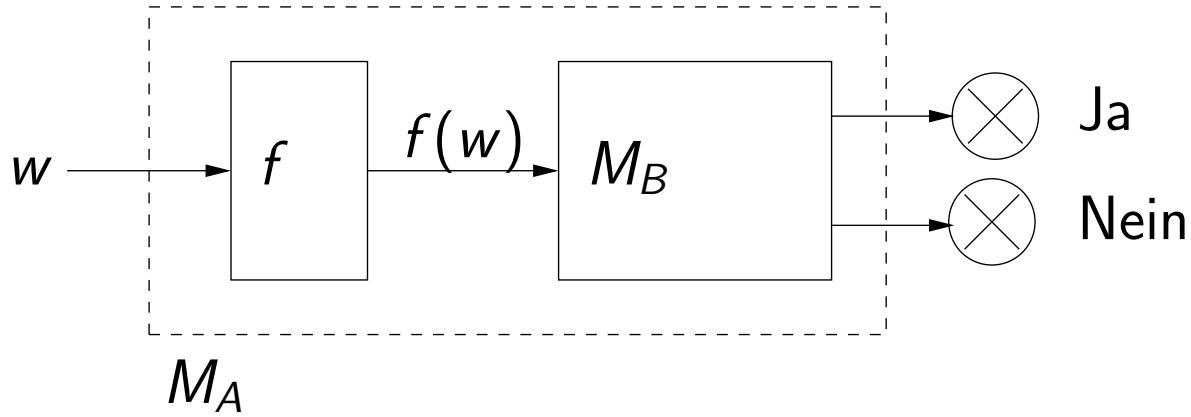
Reduktionen

Reduktion/Reduzierbarkeit (Definition)

Gegeben seien Sprachen $A \subseteq \Sigma^*$, $B \subseteq \Gamma^*$. Dann heißt A auf B reduzierbar (in Zeichen $A \leq B$), falls es eine totale und berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

$$x \in A \iff f(x) \in B.$$

Anschaulich: $A \leq B$, falls man aus einer Maschine M_B für B und einer Funktion f eine Maschine M_A für A bauen kann. Das heißt, M_B wird nach einer Vorverarbeitung der Eingabe durch f als Unterprozedur aufgerufen.



Reduktionen

Die folgende Aussage ist dann offensichtlich:

Lemma 29 (Reduktionen und Entscheidbarkeit)

Es sei $A \leq B$.

- Falls B entscheidbar ist, dann ist auch A entscheidbar.
- Falls A unentscheidbar ist, dann ist auch B unentscheidbar.

Kochrezept um die Unentscheidbarkeit eines Problems B zu zeigen

- Finde ein geeignetes Problem A, von dem bekannt ist, dass es unentscheidbar ist.

  Bisher kennen wir nur das spezielle Halteproblem K, wir werden allerdings bald weitere geeignete Probleme kennenlernen.
- Finde eine geeignete Funktion f, mit Hilfe derer A auf B reduziert werden kann und beweise, dass sie korrekt ist.
- Dann folgt daraus unmittelbar, dass B unentscheidbar ist.

Unentscheidbarkeit des Halteproblems

Satz 30 (Unentscheidbarkeit des Halteproblems)

Das (allgemeine) Halteproblem H ist nicht entscheidbar.

Beweis:

Sei die berechenbare Funktion f definiert durch $f(w) = w\#w$ für $w \in \{0,1\}^*$.

Dann gilt für alle $w \in \{0,1\}^*$:

$$w \in K \iff w\#w \in H \iff f(w) \in H.$$

Also gilt $K \leq H$.

Da K nach Satz 28 unentscheidbar ist, muss auch H unentscheidbar sein. □

Unentscheidbarkeit des Halteproblems

Halteproblem auf leerem Band (Definition)

Das Halteproblem auf leerem Band ist die Sprache

$$H_0 = \{w \in \{0,1\}^* \mid M_w \text{ hält angesetzt auf ein leeres Band}\}.$$

Satz 31 (Unentscheidbarkeit des Halteproblems auf leerem Band)

Das Halteproblem auf leerem Band H_0 ist nicht entscheidbar.

Beweis:

Wir zeigen $H \leq H_0$.

Jedem Wort $w\#x$ mit $w, x \in \{0,1\}^*$ ordnen wir eine Turing-Maschine $M(w\#x)$ zu, die, wenn auf dem leeren Band gestartet, wie folgt arbeitet:

1. Schreibe x auf das Band.
2. Simuliere dann die Maschine M_w.

Unentscheidbarkeit des Halteproblems

Es ist egal, wie die Maschine $M(w\#x)$ auf einem nicht-leeren Band arbeitet.

Wir definieren nun die Funktion $f : \{0,1,\#\}^* \to \{0,1\}^*$ durch die Vorschrift

$$M(w\#x) = M_{f(w\#x)} \text{ für alle } w, x \in \{0,1\}^*.$$

Für Wörter der Form $y \in \{0,1,\#\}^* \setminus \{0,1\}^*\#\{0,1\}^*$ sei $f(y)$ der Code einer Turing-Maschine M, die nicht auf dem leeren Band hält.

Die Funktion f ist dann offensichtlich berechenbar und es gilt:

$$\begin{aligned} w\#x \in H &\iff M_w \text{ hält bei Eingabe } x \\ &\iff M(w\#x) \text{ hält auf dem leeren Band} \\ &\iff M_{f(w\#x)} \text{ hält auf dem leeren Band} \\ &\iff f(w\#x) \in H_0 \end{aligned}$$

Unentscheidbarkeit des Halteproblems

Außerdem gilt für alle $y \in \{0,1,\#\}^* \setminus \{0,1\}^*\#\{0,1\}^*$:

$$y \notin H \quad \text{und } f(y) \notin H_0.$$

Also vermittelt f in der Tat eine Reduktion von H auf H_0. □

Satz von Rice

Das nächste Resultat zeigt, dass es unentscheidbar ist, ob die Funktion, die von einer Turing-Maschine M berechnet wird, eine bestimmte Eigenschaft $\mathcal{S}$ hat.

Das bedeutet, es gibt keine Methode, mit der man für alle Turing-Maschinen verläßliche Aussagen über die von ihnen berechneten Funktionen machen kann.

Satz 32 (Satz von Rice)

Sei $\mathcal{R}$ die Klasse aller Turing-berechenbaren Funktionen und sei $\mathcal{S}$ eine beliebige Teilmenge hiervon mit $\mathcal{S} \neq \emptyset$ und $\mathcal{S} \neq \mathcal{R}$.
Dann ist die Sprache

$$C(\mathcal{S}) = \{w \in \{0,1\}^* \mid \text{die von } M_w \text{ berechnete Funktion liegt in } \mathcal{S}\}$$

unentscheidbar.

Satz von Rice

Beweis:

Sei Ω die überall undefinierte Funktion.

Es gilt entweder $\Omega \in \mathcal{S}$ oder $\Omega \notin \mathcal{S}$.

Fall 1: $\Omega \in \mathcal{S}$

Da $\mathcal{S} \neq \mathcal{R}$ gilt, gibt es eine Funktion $q \in \mathcal{R} \setminus \mathcal{S}$.

Sei Q eine Turing-Maschine, welche q berechnet.

Wir ordnen nun jedem Wort $w \in \{0,1\}^*$ eine Turing-Maschine $M(w)$ zu, die sich bei einer Eingabe $y \in \{0,1\}^*$ wie folgt verhält.

1. $M(w)$ ignoriert die Eingabe y zunächst und simuliert M_w auf dem leeren Band.
2. Falls diese Simulation schließlich hält, so simuliert $M(w)$ die Maschine Q auf y.

Satz von Rice

Dann gilt für die von $M(w)$ berechnete Funktion g:

$$g = \begin{cases} \Omega & \text{falls } M_w \text{ auf dem leeren Band nicht hält, d. h. } w \notin H_0 \\ q & \text{sonst, d. h. } w \in H_0 \end{cases}$$

Die totale Funktion $f : \{0,1\}^* \to \{0,1\}^*$ mit

$$f(w) = \text{der Code der Maschine } M(w)$$

ist offensichtlich berechenbar.

Beachte: Es gilt $M(w) = M_{f(w)}$.

Wir erhalten:

$$\begin{aligned} w \in H_0 &\implies g = q \\ &\implies \text{die von } M_{f(w)} \text{ berechnete Funktion liegt nicht in } \mathcal{S} \\ &\implies f(w) \notin C(\mathcal{S}) \end{aligned}$$

Satz von Rice

Umgekehrt gilt:

$$\begin{aligned} w \notin H_0 &\implies g = \Omega \\ &\implies \text{die von } M_{f(w)} \text{ berechnete Funktion liegt in } \mathcal{S} \\ &\implies f(w) \in C(\mathcal{S}) \end{aligned}$$

Es gilt also $w \in \overline{H_0} \iff f(w) \in C(\mathcal{S})$, d. h. $\overline{H_0} \leq C(\mathcal{S})$.

Da H_0 nach Satz 31 unentscheidbar ist, ist auch $\overline{H_0}$ und somit $C(\mathcal{S})$ unentscheidbar.

Fall 2: $\Omega \notin \mathcal{S}$

Da $\mathcal{S} \neq \emptyset$ gilt, gibt es eine Funktion $q \in \mathcal{S}$.

Sei Q eine Turing-Maschine, welche q berechnet.

Für $w \in \{0,1\}^*$ seien die Maschine $M(w)$ sowie die berechenbare totale Funktion $f : \{0,1\}^* \to \{0,1\}^*$ exakt wie in Fall 1 definiert.

Satz von Rice

Wir erhalten diesmal:

$$\begin{aligned} w \in H_0 &\implies g = q \\ &\implies \text{die von } M_{f(w)} \text{ berechnete Funktion liegt in } \mathcal{S} \\ &\implies f(w) \in C(\mathcal{S}) \end{aligned}$$

Umgekehrt gilt:

$$\begin{aligned} w \notin H_0 &\implies g = \Omega \\ &\implies \text{die von } M_{f(w)} \text{ berechnete Funktion liegt nicht in } \mathcal{S} \\ &\implies f(w) \notin C(\mathcal{S}) \end{aligned}$$

Hieraus folgt die Unentscheidbarkeit von $C(\mathcal{S})$ wie in Fall 1. □

Satz von Rice

Konsequenzen aus dem Satz von Rice

Folgende Probleme sind unentscheidbar:

- Konstante Funktion: $\{w \mid M_w \text{ berechnet eine konstante Funktion}\}$
- Identität: $\{w \mid M_w \text{ berechnet die Identitätsfunktion}\}$
- Totale Funktion: $\{w \mid M_w \text{ berechnet eine totale Funktion}\}$
- Überall undefinierte Funktion: $\{w \mid M_w \text{ berechnet } \Omega\}$

Der Satz von Rice erlaubt es Unentscheidbarkeitsresultat für die Eigenschaften der von einer Turing-Maschine berechneten Funktion zu zeigen, nicht jedoch für andere Eigenschaften einer Turing-Maschine (wie beispielsweise die Anzahl ihrer Zustände oder das Bandalphabet).

Konsequenz des Satzes von Rice auf die Verifikation von Programmen: Kein Programm kann automatisch die Korrektheit von Software überprüfen.

Satz von Rice

Der Satz von Rice und seine Varianten gelten natürlich auch für andere universelle Berechnungsmodelle.

Satz 33 (Halteproblem für GOTO-/WHILE-Programme)

Für ein gegebenes GOTO-/WHILE-Programm und Anfangswerte für die Variablen ist es nicht entscheidbar, ob das Programm auf dieser Eingabe hält.

Beweis:

Das Halteproblem für Turing-Maschinen ist auf dieses Problem reduzierbar. Dazu muss nur die Turing-Maschine in das entsprechende GOTO-/WHILE-Programm und die Eingabe der Maschine in die entsprechenden Variablenbelegungen übersetzt werden.

Siehe Transformation "Turing-Maschine $\rightarrow$ GOTO-Programm" als Reduktionsfunktion f. □

Satz von Rice

Es ist bereits folgendes Problem unentscheidbar:

Satz 34 (Halteproblem für GOTO-Programme mit zwei Variablen)

Für ein gegebenes GOTO-Programm mit zwei Variablen, die beide mit 0 vorbelegt sind, ist es nicht entscheidbar, ob das Programm hält.

(Ohne Beweis)

Für GOTO-Programme mit nur einer Variablen ist das Halteproblem übrigens entscheidbar, denn eine Variable kann durch einen Kellerautomaten simuliert werden.

Nicht-berechenbare Funktionen

Ähnlich zur Unentscheidbarkeit von Problemen kann auch gezeigt werden, dass bestimmte Funktionen nicht berechenbar sind. Ein Beispiel dafür ist die sogenannte Busy-Beaver-Funktion.

Busy Beaver

Wir betrachten alle Turing-Maschinen mit dem zweielementigen Bandalphabet $\Gamma = \{1, \square\}$ und n Zuständen. Von diesen Maschinen halten einige auf dem leeren Band, andere terminieren nicht.
Der Wert der Busy-Beaver-Funktion Σ an der Stelle n ist die maximale Anzahl an Einsen, die von einer Maschine mit n Zuständen geschrieben werden kann, die auf dem leeren Band terminiert.

Nicht-berechenbare Funktionen

Von der Busy-Beaver-Funktion $\Sigma\colon \mathbb{N} \to \mathbb{N}$ ist folgendes bekannt:

- Sie ist nicht berechenbar.
- Über die Funktionswerte weiß man folgendes:

n	$\Sigma(n)$
1	1
2	4
3	6
4	13
5	≥ 4098
6	$\geq 1,29 * 10^{865}$

Bereits der Funktionswert an der Stelle $n = 5$ ist bisher noch nicht genau ermittelt worden.

Unentscheidbare Probleme

Wir verwenden nun die Reduktions-Beweistechnik und zeigen die Unentscheidbarkeit folgender Probleme:

- Postsches Korrespondenzproblem PCP

  PCP: ein kombinatorisches Problem auf Wörtern, wichtiges (Hilfs-)Problem, um damit die Unentscheidbarkeit anderer Probleme zu zeigen
- Schnittproblem für kontextfreie Grammatiken

Postsches Korrespondenzproblem

Wir betrachten nun ein wichtiges unentscheidbares Problem, das dazu benutzt wird, die Unentscheidbarkeit vieler anderer Probleme zu zeigen:

Postsches Korrespondenzproblem (PCP)

- **Eingabe:** Eine endliche Liste von Wortpaaren $I = ((x_1, y_1), \ldots, (x_k, y_k))$ mit $x_i, y_i \in \Sigma^+$.

  Dabei ist Σ ein beliebiges Alphabet.
- **Frage:** Gibt es eine Folge von Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$, $n \geq 1$ mit $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$?

Eine Folge $(i_1, \ldots, i_n)$ mit $n \geq 1$, $i_1, \ldots, i_n \in \{1, \ldots, k\}$ und $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$ nennen wir eine Lösung für die PCP-Eingabe $I = ((x_1, y_1), \ldots, (x_k, y_k))$ und $x_{i_1} \cdots x_{i_n}$ ist das Lösungswort.

Postsches Korrespondenzproblem

Beispiel 1: Die folgende PCP-Eingabe ist lösbar:

$$\begin{array}{llllll} x_1 & = & 0 & x_2 & = & 1 & x_3 & = & 0101 \\ y_1 & = & 010 & y_2 & = & 101 & y_3 & = & 01 \end{array}$$

Eine mögliche Lösung: $(3, 3, 1, 2)$:

$$\begin{array}{l} 01 \quad 01 \mid 010 \quad 1 \mid 0 \mid 1 \\ 01 \mid 01 \mid 010 \mid 1 \quad 0 \quad 1 \end{array}$$

Eine weitere (kürzere) Lösung ist: $(3, 1)$

Postsches Korrespondenzproblem

Beispiel 2: Die folgende PCP-Eingabe ist lösbar:

$$\begin{array}{llllllllllll} x_1 & = & 001 & x_2 & = & 01 & x_3 & = & 01 & x_4 & = & 10 \\ y_1 & = & 0 & y_2 & = & 011 & y_3 & = & 101 & y_4 & = & 001 \end{array}$$

Eine kürzeste Lösung besteht bereits aus 66 Indizes:
$(2, 4, 3, 4, 4, 2, 1, 2, 4, 3, 4, 3, 4, 4, 3, 4, 4, 2, 1, 4, 4, 2, 1, 3, 4, 1, 1, 3,$
$4, 4, 4, 2, 1, 2, 1, 1, 1, 3, 4, 3, 4, 1, 2, 1, 4, 4, 2, 1, 4, 1, 1, 3, 4, 1, 1, 3,$
$1, 1, 3, 1, 2, 1, 4, 1, 1, 3)$.

An der Komplexität dieser Lösung kann man bereits die Schwierigkeit des Problems ablesen.

Postsches Korrespondenzproblem

Satz 35 (Semi-Entscheidbarkeit des PCP)

Das Postsche Korrespondenzproblem ist semi-entscheidbar.

Beweis:

Probiere erst alle Indexfolgen der Länge 1 aus, dann alle Indexfolgen der Länge 2, ...

Falls irgendwann eine passende Indexfolge gefunden wird, so gib 1 aus. □

Postsches Korrespondenzproblem

Der erste Schritt des Unentscheidbarkeitsbeweises ist es, das folgende modifizierte Problem zu betrachten.

Modifiziertes PCP (MPCP)

- **Eingabe:** I wie beim PCP.
- **Frage:** Gibt es eine Lösung $(i_1, \ldots, i_n)$ für I mit $i_1 = 1$?

Wir beweisen nun zwei Reduktions-Lemmata, aus denen die Unentscheidbarkeit des Postschen Korrespondenzproblems folgt:

Lemma 36 (MPCP auf PCP reduzierbar)

$\text{MPCP} \leq \text{PCP}$

Postsches Korrespondenzproblem

Beweis:

Sei $I = ((x_1, y_1), \ldots, (x_k, y_k))$ mit $x_i, y_i \in \Sigma^+$ eine endliche Folge von Wortpaaren.

Seien $\#$ und \$ zwei neue Symbole.

Für ein Wort $w = a_1 a_2 \cdots a_n$ mit $a_1, \ldots, a_n \in \Sigma$ und $n \geq 1$ definieren wir die Wörter ${}^{\#}w$, $w^{\#}$, ${}^{\#}w^{\#}$ wie folgt:

$$\begin{aligned} {}^{\#}w &= \#a_1\#a_2\#\cdots\#a_n \\ w^{\#} &= a_1\#a_2\#\cdots\#a_n\# \\ {}^{\#}w^{\#} &= \#a_1\#a_2\#\cdots\#a_n\# \end{aligned}$$

Wir ordnen nun der Liste I die Liste

$$f(I) = (({}^{\#}x_1^{\#}, {}^{\#}y_1), ({}x_1^{\#}, {}^{\#}y_1), \ldots, ({}x_k^{\#}, {}^{\#}y_k), (\$, \#\$))$$

zu. Diese besteht aus $k + 2$ Paaren.

Die Funktion f ist offensichtlich berechenbar.

Postsches Korrespondenzproblem

Wir behaupten, dass f eine Reduktion von MPCP nach PCP vermittelt.

Sei zunächst $(i_1, i_2, \ldots, i_n)$ eine Lösung für I mit $i_1 = 1$ $(n \geq 1)$.

Dann ist $(1, i_2 + 1, \ldots, i_n + 1, k + 2)$ eine Lösung für $f(I)$.

Sei nun $(i_1, \ldots, i_n)$ eine kürzeste Lösung von $f(I)$.

Dann muss $i_1 = 1$, $i_2, \ldots, i_{n-1} \in \{2, \ldots, k + 1\}$ und $i_n = k + 2$ gelten.

Dann ist $(1, i_2 - 1, \ldots, i_{n-1} - 1)$ eine Lösung für I. □

Postsches Korrespondenzproblem

Lemma 37 (Halteproblem auf MPCP reduzierbar)

$H \leq \text{MPCP}$

Beweis:

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ eine deterministische Turing-Maschine (gegeben durch ihre Kodierung) und $w \in \Sigma^*$.

Wir konstruieren hieraus eine MPCP-Eingabe

$$I(M, w) = ((x_1, y_1), \ldots, (x_k, y_k)),$$

welche lösbar ist genau dann, wenn M auf Eingabe w hält.

$I(M, w)$ wird über dem Alphabet $Z \cup \Gamma \cup \{\$, \#\}$ definiert (o.B.d.A. sei $Z \cap \Gamma = \emptyset$).

Postsches Korrespondenzproblem

Dann sieht $I(M, w)$ wie folgt aus:

- 1. Wortpaar:
  $(\$, \$\square\square z_0 w\square\#)$
- Kopierpaare:
  (a, a) für alle $a \in \Gamma \cup \{\#\}$
- Transitionspaare:
  $(za, z'c)$ falls $\delta(z, a) = (z', c, N)$
  (za, cz') falls $\delta(z, a) = (z', c, R)$
  $(bza, z'bc)$ falls $\delta(z, a) = (z', c, L)$ und $b \in \Gamma$
- Paar um Blanks am Rand hinzuzufügen:
  $(\#, \square\#)$ und $(\#, \#\square)$
- Löschpaare:
  (az_e, z_e) und $(z_e a, z_e)$ für alle $z_e \in E$ und $a \in \Gamma$
- Abschlusspaar:
  $(z_e\#\#, \#)$ für alle $z_e \in E$

Postsches Korrespondenzproblem

Behauptung: M hält auf Eingabe w genau dann, wenn $I(M, w)$ lösbar ist.

Angenommen M hält auf Eingabe w.

Dann gibt es eine Folge von Konfigurationen $k_0, k_1, \ldots, k_t \in \Gamma^+ Z \Gamma^+$ mit:

- $k_0 = \square z_0 w \square$
- $k_i \vdash_M k_{i+1}$ für alle $0 \leq i \leq t-1$
- $k_t \in \Gamma^+ E \Gamma^+$

Dann erhalten wir eine Lösung von $I(M, w)$, wobei das Lösungswort wie folgt aussieht:

$$\$k_0 \# k_1 \# \cdots \# k_t \# k_t' \# k_t'' \# k_t''' \cdots \# z_e \# \#.$$

Hierbei entstehen $k_t', k_t'', k_t''', \ldots$ aus k_t, indem jeweils das Symbol links oder rechts von z_e gelöscht wird.

Postsches Korrespondenzproblem

Angenommen $I(M, w)$ hat nun eine Lösung $(1, i_2, \ldots, i_t)$, welche also mit $(\#, \#\square\square z_0 w \square \#)$ beginnt.

Solange in der "partiellen Lösung" $(x_1 x_{i_2} \cdots x_{i_n}, y_1 y_{i_2} \cdots y_{i_n})$ in $y_1 y_{i_2} \cdots y_{i_n}$ (dem längeren Wort) noch kein Endzustand $z_e \in E$ vorkommt, muss mittels der Kopierpaare, Transitionspaare und dem Paar zum Hinzufügen von Blanks eine Berechnung von M auf Eingabe w korrekt simuliert werden.

Da wir aber eine endliche Lösung $(1, i_2, \ldots, i_t)$ haben, muss für ein $m \leq t$ ein Endzustand $z_e \in E$ in $y_1 y_{i_1} \cdots y_{i_m}$ vorkommen.

Also hält M bei Eingabe w. $\square$

Postsches Korrespondenzproblem

Satz 38 (PCP unentscheidbar)

Das Postsche Korrespondenzproblem ist unentscheidbar.

Beweis: Die Behauptung folgt direkt aus den beiden vorherigen Lemmata:

Aus $H \leq \text{MPCP} \leq \text{PCP}$ folgt $H \leq \text{PCP}$. (durch Komposition der Reduktionsabbildungen).

Da außerdem das allgemeine Halteproblem H nicht entscheidbar ist, ist auch PCP nicht entscheidbar. □

Postsches Korrespondenzproblem

Bemerkungen:

Sei $\text{PCP}_{m,n}$ die Einschränkung des PCPs auf Eingaben der Form $((x_1, y_1), \ldots, (x_k, y_k))$ mit $k \leq m$, $x_1, y_1, \ldots, x_k, y_k \in \{a_1, \ldots, a_n\}^+$
(d. h. n-elementiges Alphabet und nur maximal m Wortpaare)

- Bereits $\text{PCP}_{5,2}$ ist unentscheidbar.
  (Turlough Neary 2015, `http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=4948`
- $\text{PCP}_{m,1}$ und $\text{PCP}_{2,n}$ sind entscheidbar (für m und n beliebig).
- Es ist unbekannt, ob $\text{PCP}_{k,2}$ für $k \in \{3, 4\}$ entscheidbar ist.

Schnittproblem für kontextfreie Grammatiken

Wir werden nun das PCP dazu nutzen, um die Unentscheidbarkeit des Schnittproblems für kontextfreie Grammatiken zu zeigen.

Schnittproblem für kontextfreie Grammatiken

- **Eingabe:** zwei kontextfreie Grammatiken G_1, G_2.
- **Frage:** Gilt $L(G_1) \cap L(G_2) \neq \emptyset$, d.h., es gibt ein Wort, das sowohl von G_1 als auch von G_2 erzeugt wird?

Satz 39 (Schnittproblem unentscheidbar)

Das Schnittproblem für kontextfreie Grammatiken ist unentscheidbar.

Schnittproblem für kontextfreie Grammatiken

Beweis:

Aufgrund von Satz 38 (PCP unentscheidbar) genügt es zu zeigen, dass PCP auf das Schnittproblem für kontextfreie Grammatiken reduzierbar ist.

Sei hierzu $I = ((x_1, y_1), \ldots, (x_k, y_k))$ eine beliebige PCP-Eingabe mit $x_1, y_1, \ldots, x_k, y_k \in \Sigma^+$.

Sei $\Gamma = \Sigma \cup \{\$, 1, \ldots, k\}$.

Wir definieren nun zwei kontextfreie Grammatiken G_1 und G_2 über dem Terminalalphabet Γ.

Produktionen von G_1 (S ist das Startsymbol):

$$\begin{array}{lcl} S & \to & A\,\$\,B \\ A & \to & 1\,A\,x_1 \mid 1\,x_1 \mid \cdots \mid k\,A\,x_k \mid k\,x_k \\ B & \to & y_1^{\text{rev}}\,B\,1 \mid y_1^{\text{rev}}\,1 \mid \cdots \mid y_k^{\text{rev}}\,B\,k \mid y_k^{\text{rev}}\,k \end{array}$$

Schnittproblem für kontextfreie Grammatiken

Hierbei ist w^{rev} das Wort w von recht nach links gelesen.

Dann gilt:

$$L(G_1) = \{i_n \cdots i_1 x_{i_1} \cdots x_{i_n} \$ (y_{j_1} \cdots y_{j_m})^{\text{rev}} j_1 \cdots j_m \mid n, m \geq 1, 1 \leq i_1, \ldots, i_n, j_1, \ldots, j_m \leq k\}.$$

Produktionen von G_2 (S ist das Startsymbol):

$$\begin{array}{lcl} S & \to & 1S1 \mid \cdots kSk \mid T \\ T & \to & aTa \text{ für alle } a \in \Sigma \mid \$ \end{array}$$

Dann gilt: $L(G_2) = \{uv\$v^{\text{rev}}u^{\text{rev}} \mid u \in \{1, \ldots, k\}^*, v \in \Sigma^*\}$.

Also gilt: I lösbar $\iff$ $L(G_1) \cap L(G_2) \neq \emptyset$. □

Schnittproblem für kontextfreie Grammatiken

In dem vorherigen Beweis entsprechen Lösungen der PCP-Eingabe I genau den Wörtern in $L(G_1) \cap L(G_2)$.

Nun gilt für jede PCP-Eingabe J: J lösbar genau dann, wenn J unendlich viele Lösungen hat.

Also gilt: I lösbar genau dann, wenn $L(G_1) \cap L(G_2)$ unendlich ist.

Wir erhalten:

Satz 40

Es ist unentscheidbar, ob für gegebene kontextfreie Grammatiken G_1 und G_2 der Schnitt $L(G_1) \cap L(G_2)$ unendlich ist.

Schnittproblem für kontextfreie Grammatiken

Es ist einfach, eine kontextfreie Grammatik G_2' für die Sprache $\Gamma^* \setminus L(G_2)$ anzugeben (Übung).

Sei nun G_3 eine kontextfreie Grammatik für $L(G_1) \cup L(G_2')$.

Dann gilt:

$$\begin{aligned} L(G_1) \cap L(G_2) = \emptyset \quad &\iff \quad L(G_1) \subseteq L(G_2') \\ &\iff \quad L(G_1) \cup L(G_2') = L(G_2') \\ &\iff \quad L(G_3) = L(G_2') \end{aligned}$$

Wir erhalten:

Satz 41

Es ist unentscheidbar, ob für gegebene kontextfreie Grammatiken G_1 und G_2 gilt:

- $L(G_1) \subseteq L(G_2)$
- $L(G_1) = L(G_2)$

Schnittproblem für kontextfreie Grammatiken

Schließlich kann man von den Grammatiken G_1, G_2 und G_2' zeigen, dass sie deterministisch kontextfreie Sprachen erzeugen, und man kann aus G_1, G_2, G_2' äquivalente deterministische Kellerautomten A_1, A_2, A_2' konstruieren.

Also ergibt sich:

Satz 42

Es ist unentscheidbar, ob für gegebene deterministische Kellerautomaten A_1 und A_2 gilt:

- $T(A_1) \cap T(A_2) \neq \emptyset$
- $T(A_1) \cap T(A_2)$ ist unendlich.
- $T(A_1) \subseteq T(A_2)$

Schnittproblem für kontextfreie Grammatiken

Bemerkung 1: Die auf Folie 195 konstruierte Sprache $L(G_1) \cup L(G_2')$ ist nicht notwendigerweise deterministisch kontextfrei (die Klasse der deterministisch kontextfreien Sprachen ist nicht unter Vereinigung abgeschlossen).

In der Tat ist es entscheidbar, ob $T(A_1) = T(A_2)$ für zwei gegebene deterministische Kellerautomaten A_1 und A_2 gilt (Senizergues 1997).

Bemerkung 2: Das Schnittproblem für kontextfreie Sprachen ist semi-entscheidbar:

Allgemeiner: Die Menge $\{(u, v) \mid u, v \in \{0, 1\}^*, T(M_u) \cap T(M_v) \neq \emptyset\}$ ist semi-entscheidbar, d.h. das Schnittproblem für Typ-0-Sprachen ist semi-entscheidbar:

Die Sprachen $T(M_u)$ und $T(M_v)$ sind rekursiv aufzählbar.

Zähle "parallel" die Sprachen $T(M_u)$ und $T(M_v)$ auf.

Leerheit von kontextsenisitiven Sprachen

Terminiere mit Ausgabe 1 falls irgendwann ein Wort w in beiden Aufzählungen auftaucht.

Konsequenz: Das Komplement des Schnittproblems ist nicht semi-entscheidbar. Ansonsten wäre es nämlich entscheidbar (Satz 25).

Satz 43 (Leerheit von Typ-1-Grammatiken unentscheidbar)

Es ist unentscheidbar, ob für eine gegebene Typ-1-Grammatik G (oder alternativ einen linear beschränkten Automaten) gilt: $L(G) \neq \emptyset$.

Beweis:

Wir reduzieren das Schnittproblem für kontextfreie Grammatiken auf das Leerheitsproblem für Typ-1-Grammatiken.

Mit Satz 39 beweist dies den Satz.

Seien G_1 und G_2 zwei kontextfreie Grammatiken.

Diese sind insbesondere vom Typ-1.

Leerheit von kontextsenisitiven Sprachen

Da die Typ-1-Sprachen effektiv unter Schnitt abgeschlossen sind, können wir aus G_1 und G_2 eine Typ-1-Grammatik G mit $L(G) = L(G_1) \cap L(G_2)$ konstruieren.

Etwas detaillierter: Konstruiere aus G_1 und G_2 zwei linear beschränkte Automaten A_1 und A_2 mit $L(G_1) = T(A_1)$ und $L(G_2) = T(A_2)$ (siehe Konstruktion im Beweis von Satz 1).

Aus A_1 und A_2 kann man dann leicht einen linear beschränkten Automaten A mit $T(A) = T(A_1) \cap T(A_2)$ konstruieren.

A kann dann wieder mittels der Konstruktion im Beweis von Satz 1 in eine äquivalente Typ-1-Grammatik umgewandelt werden. □

Logik-Probleme

Prädikatenlogik (1. Stufe) (siehe Vorlesung *Logik*): man darf über Elemente des Universums quantifizieren ($\forall x$, $\exists x$), nicht jedoch über Mengen oder Relationen.

Satz von Church (1936)

Es ist unentscheidbar, ob eine gegebene Formel der Prädikatenlogik unerfüllbar ist, d. h. das folgende Problem ist unentscheidbar:

EINGABE: Eine prädikatenlogische Formel φ

FRAGE: Hat φ kein Modell?

Es folgt sofort, dass auch die Frage, ob eine prädikatenlogische Formel φ gültig ist (d. h. in allen passenden Strukturen gilt) unentscheidbar ist:

$$\varphi \text{ gültig} \iff \neg\varphi \text{ unerfüllbar}$$

Aber es gilt noch:

Semi-Entscheidbarkeit der gültigen Formeln

Die Menge aller gültigen prädikatenlogischen Formeln ist semi-entscheidbar.

Z. B. liefert der prädikatenlogische Resolutionskalkül (siehe 1. Semester) ein Semi-Entscheidungsverfahren für unerfüllbare Formeln (und damit für gültige Formeln).

Hieraus folgt dann wieder, dass die Menge aller erfüllbaren Formeln nicht semi-entscheidbar (und damit nicht rekursiv aufzählbar) ist.

Logik-Probleme

Ist die folgende Aussage über die natürlichen Zahlen wahr?

$$\forall x \Big((x > 2 \ \wedge \ \exists y \ (x = y + y)) \ \rightarrow$$
$$\exists p \ \exists q \ (x = p + q \ \wedge \ p > 1 \ \wedge \ q > 1 \ \wedge$$
$$\forall u \ \forall v \ ((p = u \cdot v \ \vee \ q = u \cdot v) \ \rightarrow \ (u = 1 \ \vee \ v = 1))) \Big)$$

Dies ist ein seit über 250 Jahren offenes Problem (Goldbachsche Vermutung: Jede gerade Zahl größer als 2 ist Summe von zwei Primzahlen).

Erster Gödelscher Unvollständigkeitssatz (1931)

Die Menge aller prädikatenlogischen Aussagen, die in der Struktur $(\mathbb{N}, +, \cdot)$ wahr sind, ist nicht semi-entscheidbar (und damit nicht rekursiv aufzählbar).

Logik-Probleme

Der erste Gödelsche Unvollständigkeitssatz wird auch häufig so formuliert: Es gibt keinen endlichen Beweiskalkül, mit dem sich alle wahren arithmetischen Aussagen herleiten lassen (denn ein solcher Beweiskalkül könnte zur Aufzählung aller wahren arithmetischen Aussagen benutzt werden).

Aus dem ersten Gödelschen Unvollständigkeitssatz folgt auch, dass das Erfüllbarkeitsproblem für Logiken höherer Stufe (z. B. Prädikatenlogik 2. Stufe, bei der Quantifikation über Relationen erlaubt ist), nicht mehr semi-entscheidbar ist. Also haben solche Logiken keinen endlichen Beweiskalkül (wie etwa der Resolutionskalkül für Prädikatenlogik).

Denn in diesen Logiken lassen sich die natürlichen Zahlen axiomatisieren (Peano-Axiome) und damit beliebige arithmetische Aussagen formulieren. Dies ist mit Prädikatenlogik 1. Stufe nicht ohne weiteres möglich, da man darin das Induktionsaxiom nicht ausdrücken kann.

Logik-Probleme

Bereits kleine Fragmente der Arithmetik liefern unentscheidbare Probleme. Ein besonders prominentes Beispiel ist:

Hilbert's 10. Problem ist unentscheidbar (Matiyasevich 1970)

Das folgende Problem ist unentscheidbar:

EINGABE: Ein Polynom $p(x_1, \ldots, x_n)$ (in mehreren Variablen) mit Koeffizienten aus $\mathbb{Z}$.

FRAGE: Existieren $a_1, \ldots, a_n \in \mathbb{Z}$ mit $p(a_1, \ldots, a_n) = 0$?

Deterministische Zeitklassen

Sei $f : \mathbb{N} \to \mathbb{N}$ eine monotone Funktion. Die Klasse DTIME(f) besteht aus allen Sprachen L, für die es eine deterministische Turingmaschine M gibt mit:

- M berechnet die charakteristische Funktion von L.
- Für jede Eingabe $w \in \Sigma^*$ erreicht M von der Startkonfiguration $z_0 w \Box$ aus nach höchstens $f(|w|)$ Rechenschritten einen Endzustand (und gibt 0 oder 1 aus, je nachdem ob $w \notin L$ oder $w \in L$ gilt).

Poly bezeichet die Menge aller durch ein Polynom mit Koeffizienten aus $\mathbb{N}$ beschriebenen Funktionen auf $\mathbb{N}$ (z. B. $n, 2n, n^2 + 3n, n^{10000}$).

Die Klasse P

$$\mathrm{P} = \bigcup_{f \in \mathrm{Poly}} \mathrm{DTIME}(f)$$

Bemerkungen:

- P wird häufig als die Menge aller effizient lösbaren Probleme betrachtet.
- Die Klasse P ist relativ robust gegen Änderungen des Berechnungsmodells. So ändert sich z. B. die Klasse P nicht, wenn wir anstatt normalen Turingmaschinen Mehrband-Turingmaschinen erlauben würden (was auch etwas realistischer wäre, da man bei Verwendung von Einband-Turingmaschinen sehr viel Information kopieren muss).
- Definiert man P über WHILE- oder GOTO-Programme, so muss man den Zeitaufwand für eine Zuweisung (z. B. $x_i := x_j + 1$) als die aktuelle Anzahl der Bits von x_j (also etwa $\log(x_j)$) bemessen: logarithmisches Kostenmaß.

- Würde man das uniforme Kostenmaß (d. h. eine Zuweisung wird als ein Schritt gezählt) verwenden, so wäre z. B. der folgende Algorithmus polynomial:

  INPUT n;
  $x := 2$;
  LOOP n DO $x := x * x$ END;
  OUTPUT(x)

  Dieser Algorithmus berechnet die Zahl 2^{2^n}, und um diese Zahl aufzuschreiben benötigt man schon 2^n viele Bits.

  Bei diesem Beispiel haben wir aber etwas betrogen, da wir Multiplikation als elementare Operation verwenden. Ersetzt man $x := x * x$ durch ein (Standard-) Loop-Programm, so wird die Rechenzeit des obigen Algorithmus exponentiell.

Nichtdeterministische Zeitklassen

Sei $f : \mathbb{N} \to \mathbb{N}$ eine monotone Funktion. Die Klasse NTIME(f) besteht aus allen Sprachen L, für die es eine nichtdeterministische Turingmaschine M gibt mit:

- Für jede Eingabe $w \in \Sigma^*$ erreicht M von der Startkonfiguration $z_0 w \square$ aus auf jedem Berechnungspfad nach höchstens $f(|w|)$ Rechenschritten einen Endzustand und gibt 0 oder 1.
- Es gilt: $w \in L$ genau dann, wenn M auf mindestens einem Berechnungspfad eine 1 ausgibt.

Die Klasse NP

$$\mathrm{NP} = \bigcup_{f \in \mathrm{Poly}} \mathrm{NTIME}(f)$$

Bemerkungen:

- Offensichtlich gilt $P \subseteq NP$.
- Ob $P = NP$ gilt, gilt als die wichtigste offene Frage in der Theoretischen Informatik. Es wird im Allgemeinen vermutet, dass $P \neq NP$ gilt.
- Warum ist die Frage $P = NP$ so interessant?
  Von einer Vielzahl von Problemen ist bekannt, dass sie in NP liegen, man weiß jedoch nicht, ob sie in P liegen.
  Man kennt sogar eine große Klasse von Problemen (die NP-vollständigen Probleme, mehr dazu gleich) von denen folgendes bekannt ist: Gehört eines dieser Probleme zu P, so folgt $P = NP$.
- Es ist nicht schwer zu sehen, dass alle Sprachen in NP LOOP-entscheidbar sind (d. h. die charakteristischen Funktionen von Sprachen aus NP sind LOOP-berechenbar).

Beispiel: SAT

Wir betrachten aussagenlogische Formeln, wie z. B.
$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee C)$
(dies ist eine Formel in KNF).

Solche Formeln lassen sich z. B. durch Wörter über dem Alphabet $\{a, \vee, \wedge, \neg,), (\}$ kodieren (atomare Formeln werden durch Wörter der Form a^n kodiert).

Zur Erinnerung (Vorlesung Logik, 1. Semester): Ein aussagenlogische Formel F ist erfüllbar, falls F ein Modell hat, d. h. es gibt eine Belegung der atomaren Formeln mit $0, 1$, so dass die gesamte Formel sich zu 1 auswertet.

Das Problem SAT

EINGABE: Eine aussagenlogische Formel F

FRAGE: Ist F erfüllbar?

Satz 44

SAT $\in$ NP

Beweis: Sei F eine aussagenlogische Formel, in der die atomaren Formeln $A_1, \ldots, A_n$ vorkommen.

Eine nichtdeterministische Turingmaschine "rät" nun in einer ersten Phase eine Belegung $\mathcal{B} : \{A_1, \ldots, A_n\} \to \{0, 1\}$:

- Im ersten Schritt verzweigt sich die Turingmaschine (d. h. es gibt zwei Folgekonfigurationen).
  Im ersten Zweig schreibt die Turingmaschine $A_1 0$ auf das Band, im zweiten Zweig schreibt sie $A_1 1$ auf das Band.
- Im zweiten Schritt verzweigt sich die Turingmaschine wieder.
  Im ersten Zweig schreibt sie (hinter $A_1 b$ mit $b \in \{0, 1\}$) $A_2 0$ auf das Band, im zweiten Zweig schreibt sie $A_2 1$ auf das Band.

  $\vdots$

Nach n Schritten steht in jedem der 2^n Berechnungszweige ein Wort der Form $A_1 b_1 A_2 b_2 \cdots A_n b_n$ mit $b_1, \ldots, b_n \in \{0, 1\}$ auf dem Band.

Dieses Wort kodiert die Belegung $\mathcal{B}$ mit $\mathcal{B}(A_i) = b_i$ für $1 \leq i \leq n$.

In einer zweiten Phase kann die Turingmaschine nun den Wert $\mathcal{B}(F)$ deterministisch ausrechnen, indem die Formel F einmal von links nach rechts durchlaufen wird und jedesmal, wenn eine atomare Formel A_i gesehen wird, der Wert $\mathcal{B}(A_i) = b_i$ aus dem gespeicherten "Belegungswort" ermitelt wird.

Dies benötigt höchstens $|F|^2$ Schritte, die Turingmaschine rechnet also auf jedem Berechnungspfad nur $O(|F|^2)$ Schritte.

Die Maschine gibt am Ende 1 aus, genau dann, wenn $\mathcal{B}(F) = 1$.

Also gibt es einen Berechnungspfad, auf dem die Maschine 1 ausgibt, genau dann, wenn F erfüllbar ist. □

Reduktionen so wie wir sie im Abschnitt über (Un)Entscheidbarkeit kennengelernt haben, sind für entscheidbare Probleme nicht sehr aussagekräftig:

Fakt

Seien $A, B \subseteq \Sigma^*$ entscheidbare Sprachen mit $\emptyset \neq B \neq \Sigma^*$. Dann gilt $A \leq B$.

Wähle hierzu zwei Elemente $x \in B$ und $y \in \Sigma^* \setminus B$.

Definiere die Funktion $f : \Sigma^* \to \Sigma^*$ durch

$$f(w) = \begin{cases} x \text{ falls } w \in A \\ y \text{ falls } w \notin A \end{cases}$$

Da A entscheidbar ist, ist f berechenbar, und es gilt $w \in A \iff f(w) \in B$, d. h. $A \leq B$.

Polynomiale Reduzierbarkeit

Eine Funktion $f : \Sigma^* \to \Gamma^*$ ist polynomial berechenbar, falls eine deterministische Turingmaschine M und ein Polynom $p(n)$ existiert, so dass für alle $w \in \Sigma^*$ gilt:

Wenn M mit der Eingabe w gestartet wird, hält M nach höchstens $p(|w|)$ vielen Schritten mit der Ausgabe $f(w)$ auf dem Arbeitsband an.

Eine Sprache $A \subseteq \Sigma^*$ ist polynomial reduzierbar auf eine Sprache $B \subseteq \Gamma^*$ (kurz $A \leq_p B$), falls eine polynomial berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ existiert mit

$$\forall w \in \Sigma^* : w \in A \iff f(w) \in B.$$

Lemma 45

Wenn $A \leq_p B$ und $B \in \mathrm{P}$ (bzw. $B \in \mathrm{NP}$), dann gilt $A \in \mathrm{P}$ (bzw. $A \in \mathrm{NP}$).

Beweis:

Sei zunächst $A \leq_p B$ und $B \in \mathrm{P}$.

Dann existieren Polynome $p(n)$ und $q(n)$ sowie Turingmaschinen M und N mit folgenden Eigenschaften:

- M berechnet aus einer Eingabe $w \in \Sigma^*$ in Zeit $p(|w|)$ ein Wort $f(w)$ mit: $w \in A \iff f(w) \in B$.

  Beachte: Da die Maschine M in $p(|w|)$ Schritten nur eine Ausgabe der Länge höchstens $p(|w|)$ erzeugen kann, gilt $|f(w)| \leq p(|w|)$.

- N akzeptiert die Sprache B in Zeit $q(n)$.

Ein Turingmaschine für die Sprache A arbeitet dann bei einer Eingabe w wie folgt:

1. Berechne $f(w)$ (Zeitbedarf: $p(|w|)$).
2. Simuliere die Maschine N auf $f(w)$ (Zeitbedarf: $q(p(|w|))$).

Der gesamte Zeitbedarf ist also $p(|w|) + q(p(|w|))$, was wieder ein Polynom ist.

Die Aussage für die Klasse NP kann genauso bewiesen werden. □

NP-Vollständigkeit

Eine Sprache A ist NP-hart, falls für alle $B \in \text{NP}$ gilt: $B \leq_p A$
(A ist mindestens so schwer wie jedes Problem in NP).

Eine Sprache A ist NP-vollständig, falls sie zu NP gehört und NP-hart ist.

Intuition: NP-vollständige Sprachen sind die schwierigsten Sprachen in NP.

Noch wissen wir garnicht, ob es überhaupt NP-vollständige Sprachen gibt.
Dies werden wir bald zeigen zeigen.

Zunächst aber noch ein einfaches Resultat:

Lemma 46

Wenn A NP-vollständig ist, dann gilt: $\mathrm{P} = \mathrm{NP} \iff A \in \mathrm{P}$.

Beweis:

$\Rightarrow$: Sei $\mathrm{P} = \mathrm{NP}$.

Da A NP-vollständig ist, folgt $A \in \mathrm{NP} = \mathrm{P}$.

$\Leftarrow$: Sei $A \in \mathrm{P}$ und sei $B \in \mathrm{NP}$ beliebig.

Da A NP-vollständig ist, folgt $B \leq_p A \in \mathrm{P}$.

Lemma 45 impliziert $B \in \mathrm{P}$.

Also gilt $\mathrm{NP} \subseteq \mathrm{P}$ und damit $\mathrm{NP} = \mathrm{P}$. □

Satz 47 (Satz von Cook)

SAT ist NP-vollständig.

Beweis:

Wir müssen noch zeigen, dass SAT NP-hart ist.

Sei hierfür $L \in \mathrm{NP}$, $L \subseteq \Sigma^*$.

Zu $w \in \Sigma^*$ konstruieren wir eine aussagenlogische Formel $f(w)$ mit:
$w \in L \iff f(w)$ ist erfüllbar.

Die Abbildung f wird polynomial berechenbar sein.

Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ eine nichtdeterministische Turingmaschine für L, die bei einer Eingabe der Länge n auf jedem Berechnungspfad nach $\leq p(n)$ Schritten terminiert ($p(n)$ ist ein Polynom).

Sei $w = w_1 w_2 \cdots w_n \in \Sigma^*$ eine Eingabe der Länge n.

Wir stellen o.B.d.A. folgende Forderungen an M:

1. Der Kopf von M wandert nie links von der Position, wo er zu Beginn steht (kann durch spezielle Markierung erreicht werden).
2. M terminiert genau dann mit Ausgabe 1, wenn M in den speziellen Zustand $z_1 \in E$ übergeht. D. h. der Zustand z_1 signalisiert Akzeptanz der Eingabe.
3. Alle Tupel der Form (z_1, a, z_1, a, N) (mit $a \in \Gamma$) gehören zu δ.
4. $(z, a, z', a', D), (z, b, z'', b', D') \in \delta \implies a = b, a' = b', D = D'$

   Nur hinsichtlich des Folgezustands z' haben wir also eine nichtdeterministische Wahl (siehe nächste Folie).

Eigenschaft (4) kann wie folgt erzwungen werden.

Zunächst können wir die Sprache L durch $\$L$ für ein neues Symbol \$ ersetzen, da $L \leq_p \$L$ gilt (d. h. aus $\$L \leq_p$ SAT folgt wieder $L \leq_p$ SAT).

Wir wissen also, dass alle positiven Eingaben mit einem \$ beginnen müssen.

Definiere nun die Zustandsmenge und die Transitionsrelation wie folgt um:

$$\begin{aligned} Z' =& \{(z[a, a', D] \mid z \in Z,\ a, a' \in \Gamma,\ D \in \{L, R, N\}\} \cup \{z_0[\$, \$, R]\} \\ \delta' =& \{(z[a, a', D], a, z'[b, b', D'], a', D) \mid (z, a, z', a', D) \in \delta, \\ & \qquad b, b' \in \Gamma, D' \in \{L, R, N\}\} \cup \\ & \{(z_0[\$, \$, R], \$, z_0[a, a', D], \$, R) \mid a, a' \in \Gamma,\ D \in \{L, R, N\}\} \end{aligned}$$

Der neue Anfangszustand ist $z_0[\$, \$, R]$.

Jede von der Startkonfiguration erreichbare Konfiguration kann durch ein Wort aus

$$\text{Conf} = \{\Box uzv\Box \mid z \in Z;\ u, v \in \Gamma^*;\ |uv| = p(n)\}$$

beschrieben werden.

Die Startkonfiguration ist $\Box z_0 w \Box^{p(n)+1-n}$.

Wegen Punkt (2) und (3) akzeptiert M auf einem bestimmten Berechnungspfad die Eingabe w genau dann, wenn sich M nach $p(n)$ vielen Schritten im Zustand z_1 befindet.

Notation: Für ein $\alpha \in \text{Conf}$ schreiben wir

$$\alpha = \alpha[-1]\alpha[0]\cdots\alpha[p(n)]\alpha[p(n)+1]$$

wobei $\alpha[-1] = \Box$, $\alpha[0], \ldots, \alpha[p(n)] \in Z \cup \Gamma$, $\alpha[p(n)+1] = \Box$.

Definiere die Menge der 4-Tupel

$$\Delta = \{(a, b, c, b) \mid a, b, c \in \Gamma\}$$
$$\cup \{(c, b, z, z'), (b, z, a, b), (z, a, d, a') \mid (z, a, z', a', L) \in \delta, c, b, d \in \Gamma\}$$
$$\cup \{(c, b, z, b), (b, z, a, z'), (z, a, d, a') \mid (z, a, z', a', N) \in \delta, c, b, d \in \Gamma\}$$
$$\cup \{(c, b, z, b), (b, z, a, a'), (z, a, d, z') \mid (z, a, z', a', R) \in \delta, c, b, d \in \Gamma\}$$

Idee der Δ-Tupel: Wenn drei aufeinanderfolgende Positionen $i-1$, i, $i+1$ einer Konfiguration α die Symbole $x, y, z \in Z \cup \Gamma$ enthalten, dann muss für jede Folgekonfiguration α' von α ein Tupel (x, y, z, y') existieren, in der Position i das Symbol y' enthält.

Wegen Punkt (4) gilt dann für alle $\alpha, \alpha' \in \square(Z \cup \Gamma)^*\square$ mit $|\alpha| = |\alpha'|$:

$$\alpha, \alpha' \in \text{Conf und } \alpha \vdash_M \alpha'$$
$$\Longleftrightarrow$$
$$\alpha \in \text{Conf und } \forall i \in \{0, \ldots, p(n)\} : (\alpha[i-1], \alpha[i], \alpha[i+1], \alpha'[i]) \in \Delta.$$

Beispiel:

Falls $(z, a, z', a', L) \in \delta$ ist folgende lokale Bandänderung für alle $b \in \Gamma$ möglich:

Position			i–1	i	i+1		
α	=	$\cdots$	$\cdots$	b	z	a	$\cdots$	$\cdots$
α'	=	$\cdots$	$\cdots$	z'	b	a'	$\cdots$	$\cdots$

Falls $(z, a, z', a', R) \in \delta$ ist folgende lokale Bandänderung für alle $b \in \Gamma$ möglich:

Position			i–1	i	i+1		
α	=	$\cdots$	$\cdots$	b	z	a	$\cdots$	$\cdots$
α'	=	$\cdots$	$\cdots$	b	a'	z'	$\cdots$	$\cdots$

Eine Rechnung von M können wir nun als Matrix beschreiben:

$$\begin{array}{lcccccc}
\alpha_0 & = & \square & \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,p(n)} & \square \\
\alpha_1 & = & \square & \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,p(n)} & \square \\
 & & & & \vdots & & & \\
\alpha_{p(n)} & = & \square & \alpha_{p(n),0} & \alpha_{p(n),1} & \cdots & \alpha_{p(n),p(n)} & \square
\end{array}$$

Für jedes Tripel (a, i, t) $(a \in Z \cup \Gamma,\ -1 \leq i \leq p(n)+1,\ 0 \leq t \leq p(n))$ sei $x(a, i, t)$ eine aussagenlogische Variable (atomare Formel).

Interpretation: $x(a, i, t) =$ **true** genau dann, wenn zum Zeitpunkt t das i-te Zeichen der aktuellen Konfiguration ein a ist.

An den Positionen -1 und $p(n)+1$ steht immer $\Box$:

$$G(n) \quad = \quad \bigwedge_{0 \leq t \leq p(n)} \Big(x(\Box, -1, t) \wedge x(\Box, p(n)+1, t) \Big)$$

Für jedes Paar (i, t) ist genau eine Variable $x(a, i, t)$ wahr (zu jedem Zeitpunkt kann auf einem Bandfeld nur ein Zeichen stehen):

$$X(n) \quad = \quad \bigwedge_{\substack{0 \leq t \leq p(n) \\ -1 \leq i \leq p(n)+1}} \left(\bigvee_{a \in Z \cup \Gamma} \left(x(a, i, t) \wedge \bigwedge_{b \neq a} \neg x(b, i, t) \right) \right)$$

Zum Zeitpunkt $t = 0$ ist die Konfiguration gleich $\Box z_0 w \Box^{p(n)+1-n}$

$$S(w) \quad = \quad \left(x(z_0, 0, 0) \wedge \bigwedge_{i=1}^{n} x(w_i, i, 0) \wedge \bigwedge_{i=n+1}^{p(n)} x(\Box, i, 0) \right)$$

Die Berechnung respektiert die lokale Relation Δ:

$$D(n) \quad = \quad \bigwedge_{\substack{0 \leq i \leq p(n) \\ 0 \leq t < p(n)}} \quad \bigvee_{(a,b,c,d) \in \Delta} \left(\begin{array}{lll} x(a, i-1, t) & \wedge & x(b, i, t) \wedge \\ x(c, i+1, t) & \wedge & x(d, i, t+1) \end{array} \right)$$

Sei schließlich

$$R(w) = G(n) \wedge X(n) \wedge S(w) \wedge D(n).$$

Es ergibt sich eine natürliche Bijektion zwischen der Menge der $R(w)$ erfüllenden Belegungen und der Menge derjenigen Rechnungen von M auf die Eingabe w, die aus $p(n)$ Rechenschritten bestehen.

Für $f(w) = R(w) \wedge \bigvee_{i=0}^{p(n)} x(z_1, i, p(n))$ gilt somit:

$$f(w) \text{ erfüllbar} \quad \Longleftrightarrow \quad w \in L.$$

Zahl der Variablen von $f(w) \in \mathcal{O}(p(n)^2)$

Länge von $f(w) \in \mathcal{O}(p(n)^2 \log p(n))$

Der Faktor $\mathcal{O}(\log p(n))$ ist notwendig, da zum Aufschreiben der Indizes $\log p(n)$ viele Bits benötigt werden. □

Von vielen weiteren Problemen A kann die NP-Vollständigkeit durch eine Reduktion SAT $\leq_p A$ gezeigt werden.

Hier sind einige Beispiele für weitere NP-vollständige Probleme (ohne Beweis; siehe Schöning für Beweise).

3-KNF-SAT

EINGABE: Eine aussagenlogische Formel φ in KNF, bei der jede Klausel aus höchstens 3 Literalen (atomare Formeln oder negierte atomare Formeln) besteht.
FRAGE: Ist φ erfüllbar?

Bemerkung: 2-KNF-SAT $\in$ P

CLIQUE

EINGABE: Ein ungerichteter Graph $G = (V, E)$ (siehe Vorlesung *Diskrete Strukturen*) und eine Zahl k (unär kodiert)
FRAGE: Hat G eine Clique der Größe k, d. h. existiert eine Menge $U \subseteq V$ mit $|U| \geq k$ und für alle $u, v \in U$ mit $u \neq v$: $\{u, v\} \in E$?

VERTEX-COVER

EINGABE: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl k (unär kodiert)
FRAGE: Hat G eine Knotenüberdeckung der Größe k, d. h. existiert eine Menge $U \subseteq V$ mit $|U| \leq k$ und für alle $\{u, v\} \in E$ gilt $U \cap \{u, v\} \neq \emptyset$?

3-FÄRBBARKEIT

EINGABE: Ein ungerichteter Graph $G = (V, E)$
FRAGE: Ist die Färbungszahl von G höchstens 3.

HAMILTON-CIRCUIT

EINGABE: Ein ungerichteter Graph $G = (V, E)$
FRAGE: Hat G einen Hamiltonkreis (siehe Vorlesung *Diskrete Strukturen*)?

SUBSETSUM

EINGABE: Binär-kodierte Zahlen $t, w_1, \ldots, w_n$
FRAGE: Gibt es eine Teilmenge $U \subseteq \{w_1, \ldots w_n\}$ mit $t = \sum_{w \in U} w$?

Bemerkung: Dieses Problem gehört zu P, falls man die Zahlen $t, w_1, \ldots, w_n$ unär kodiert.