Theoretische Grundlagen der Informatik

— Skript zur Vorlesung —

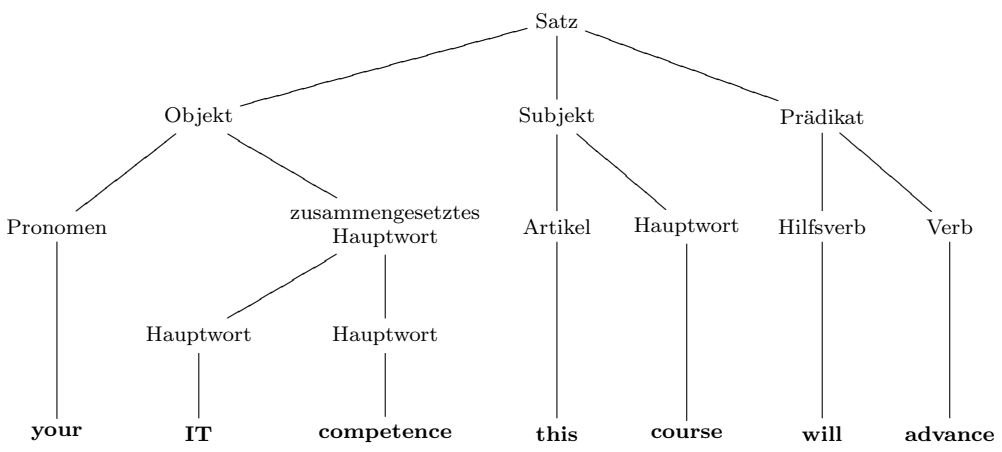
Ulrik Brandes

Sommersemester 2006

(Version vom 25. Oktober 2006)

Prolog

Die theoretischen Grundlagen der Informatik gelten gemeinhin als hartes Brot. Dies mag zum Teil daran liegen, dass die Notwendigkeit der formalen Argumentation und der weitreichende Nutzen der daraus gewonnenen Erkenntnisse nicht unmittelbar ersichtlich wird. In dieser Veranstaltung werden daher immer wieder Ausblicke auf Anwendungen der behandelten Konzepte und Resultate eingestreut, um die immense praktische Relevanz zumindest erahnbar zu machen. In jedem Fall werden die hier geleisteten Investitionen sich in den vertiefenden Veranstaltungen des Studiums bezahlt machen.



Yoda

Natürliche Sprache

- ist komplex
- ist mehrdeutig
- setzt Kontextwissen voraus
- verändert sich laufend

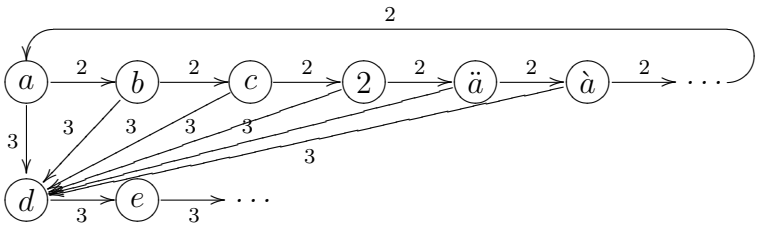
und ist daher problematisch für die formale Behandlung bzw. maschinelle Verarbeitung. In Informationssystemen werden formale Sprachen verwendet, die so gewählt werden, dass sie

- so einfach (zu handhaben) wie möglich aber
- so ausdrucksstark wie nötig sind.

Beispiele: Maschinencode, JAVA-Quelltext, -Bytecode, XML.

0.1 Beispiel (SMS – short message service)

- *Eingabe ist auch schon Kommunikation (nämlich mit dem Gerät)*
- *Benutzte Zeichen: Druck auf Tasten (nicht: Ziffern, Buchstaben, etc.)*
- *Mobiltelefon „erkennt“ zulässige Folgen von Tastendrücken und verarbeitet diese*



Eingaben bestehen also aus Folgen von Tastendrücken, Ausgaben aus alphanumerischen Zeichen und weiteren Aktionen (Löschen, Wählen, Senden, etc.). Das Gerät befindet sich dabei immer in irgendeinem Zustand, von dem die Wirkung einer Taste abhängt, die das Gerät wiederum in einen anderen Zustand überführt.

Die formale Sichtweise auf diesen Prozess ist hilfreich, z.B. um Spezifikation, Implementation, Verifikation und Dokumentation effizient und verlässlich zu gestalten.

0.2 Beispiel (Meisterschaftsproblem)
Nach dem drittletzten Spieltag der Fussball-Bundesliga Saison 1964/65 sah die Tabellenspitze wie folgt aus:

1.	*Werder Bremen*	*37 Punkte*
2.	*1. FC Köln*	*36 Punkte*
3.	*Borussia Dortmund*	*35 Punkte*
4.	*TSV 1860 München*	*33 Punkte*

Alle anderen Mannschaften hatten weniger als 33 Punkte und damit keine Chance mehr auf die Meisterschaft, da nach der 2-Punkt-Regel gespielt wurde.

Frage: *Hatte der TSV 1860 München noch eine Chance auf die Meisterschaft?*

Nein. Das lag an den zu diesem Zeitpunkt noch zu spielenden Partien

vorletzter Spieltag:		***letzter Spieltag:***	
Werder Bremen	*- Borussia Dortmund*	*1. FC Nürnberg*	*- Werder Bremen*
1. FC Köln	*- 1. FC Nürnberg*	*Borussia Dortmund*	*- 1. FC Köln*
TSV 1860 München	*- Hamburger SV*	*Meidericher SV*	*- TSV 1860 München*

und kann auf verschiedene Weisen bewiesen werden (beachte, dass München nur noch höchstens 37 Punkte erreichen konnte):

1. *Bremen, Köln und Dortmund holen an den letzten beiden Spieltagen zusammen mindestens 4 Punkte, da in zwei Spielen keine anderen Mannschaften beteiligt sind. Zum Schluss haben sie damit zusammen mindestens* $37 + 36 + 35 + 4 = 112$ *Punkte, also muss eine Mannschaft mindestens* $\lceil \frac{112}{3} \rceil = 38$ *Punkte gehabt haben.*

2. *Nur wenn die Bremer beide Spiele verlieren, haben sie weniger als 38 Punkte. In diesem Fall hat aber Dortmund mit dem Sieg gegen Bremen schon mal mindestens 37 Punkte und müsste gegen die Kölner verlieren, die dann selbst 38 Punkte hätten.*

Je nach Tabellensituation und noch ausstehenden Spielen kann die Antwort erheblich schwerer fallen. Denn man kann zwar immer alle Möglichkeiten durchgehen, da ein Spiel 3 mögliche Ausgänge hat, sind das bei n *noch ausstehenden Spielen mit Beteiligung der interessierenden Mannschaften aber* 3^n *Fälle.*

$3^6 = 729$
$3^{10} = 59\,049$
$3^{16} = 43\,046\,721$

Gibt es eine allgemeine Vorgehensweise, um immer zu kurzen Argumentationen wie im obigen Beispiel zu kommen? Anders gefragt: Ist das Meisterschaftsproblem effizient lösbar? Und welchen Einfluss haben die 3-Punkte-Regel und die Methode, mit der vom Deutschen Fussball-Bund (DFB) der Spielplan erstellt wird?

Der Vorlesungsstoff umfasst zwei wesentliche Bereiche:

1. Formale Sprachen und Automatentheorie
   - grundlegende Formalismen für Aufgaben in der Informationsverarbeitung
   - Beschreiben und Erkennen unterschiedlich komplexer „Sprachen“
2. Komplexitätstheorie
   - grundsätzliche Fragen der Berechenbarkeit („Gibt es ‘Probleme’, die – unabhängig von den zur Verfügung stehenden Hilfsmitteln wie Rechner, Speicherplatz, Geschwindigkeit, etc. – unlösbar sind?“)
   - Klassifikation lösbarer Probleme in unterschiedliche Schwierigkeitsgrade („Ist ein ‘Problem’ grundsätzlich schwieriger als ein anderes?“)
     → Wer wird Millionär?: 7 Probleme für das 3. Jahrtausend
     http://www.claymath.org/prizeproblems/ ($\mathcal{P} \stackrel{?}{=} \mathcal{NP}$)

Inhaltsverzeichnis

Kapitel 1

Einführung: Formale Sprachen

Formale Sprachen sind *das* grundlegende Konzept zur Repräsentation von Informationen.

1.1 Definition

Symbole: atomare Informations-einheiten

- *Ein (endliches) Alphabet $\Sigma = \{a, b, \ldots\}$ ist eine (endliche) Menge von Symbolen (Zeichen).*
- *Eine endliche Folge $w = a_1 \cdots a_n$ von Symbolen $a_i \in \Sigma$ heißt Wort (Zeichenkette) über Σ. Die Zahl ihrer Folgenglieder heißt die Länge des Wortes, in Zeichen: $|w| = |a_1 \cdots a_n| = n$. Die Zahl der Vorkommen eines bestimmten Symbols $a \in \Sigma$ wird mit $|w|_a$ bezeichnet. Das Wort der Länge 0 wird mit ε bezeichnet und heißt auch das leere Wort. Mit Σ^* wird die Menge aller Wörter über einem Alphabet Σ bezeichnet.*

1.2 Beispiel

Sei $\Sigma = \{0, 1\}$. Dann ist

$$\Sigma^* = \left\{ \underbrace{\varepsilon,}_{\text{Länge 0}} \quad \underbrace{0, 1,}_{\text{Länge 1}} \quad \underbrace{00, 01, 10, 11,}_{\text{Länge 2}} \; 000, \ldots \right\}$$

die Menge der Binärwörter.

1.3 Definition (Formale Sprachen)

Eine Teilmenge $L \subseteq \Sigma^*$ der Wörter über einem Alphabet Σ heißt (formale) Sprache über Σ.

1.4 Beispiel

a) Die Menge aller Bytes (Binärwörter der Länge 8).

b) $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}, \ddot{\mathtt{a}}, \ddot{\mathtt{o}}, \ddot{\mathtt{u}}, \mathtt{ß}, \mathtt{A}, \ldots, \mathtt{Z}, \ddot{\mathtt{A}}, \ddot{\mathtt{O}}, \ddot{\mathtt{U}}\}$

$$\begin{aligned} L &= \{w \in \Sigma^* : w \text{ ist ein Wort der deutschen Sprache}\} \\ &= \{\mathtt{Aal}, \mathtt{Aas}, \mathtt{aasen}, \ldots\} \end{aligned}$$

c) $\Sigma = \{\mathtt{0}, \ldots, \mathtt{9}\}$

$$\begin{aligned} L &= \{w \in \Sigma^* : w \text{ ist eine Primzahl}\} \\ &= \{\mathtt{2}, \mathtt{3}, \mathtt{5}, \mathtt{7}, \mathtt{11}, \mathtt{13}, \ldots\} \end{aligned}$$

d) Sei Σ die Menge der Unicode-Zeichen. Dann ist die Menge der syntaktisch korrekten JAVA*-Programme eine Sprache über Σ.*

1.5 Bemerkung

Jede Sprache über Σ ist ein Element der Potenzmenge von Σ^*.

1.6 Definition

Aus zwei Wörtern $w_1, w_2 \in \Sigma^*$ erhält man durch Hintereinanderschreiben ein neues Wort $w = w_1 w_2$. Die zugehörige Abbildung $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ heißt Konkatenation (wobei $\cdot(w_1, w_2) = w_1 \cdot w_2 = w_1 w_2 = w$).

Lässt sich ein Wort $w \in \Sigma^*$ als $w = u \cdot v \cdot x$ (für $u, v, x \in \Sigma^*$) schreiben, dann heißen

u	*Präfix*
v	*Teilwort* oder *Infix*
x	*Postfix* oder *Suffix*

von w.

1.7 Beispiel

Das Wort $\mathtt{SMS} \in \mathtt{ASCII}^*$ *hat*

Präfixe	$P = \{\varepsilon, \mathtt{S}, \mathtt{SM}, \mathtt{SMS}\}$
Suffixe	$S = \{\varepsilon, \mathtt{S}, \mathtt{MS}, \mathtt{SMS}\}$
Teilwörter	$P \cup S \cup \{\mathtt{M}\}$

1.8 Bemerkung

Eine Sprache heißt präfixfrei, falls kein Wort der Sprache Präfix eines anderen Wortes der Sprache ist. Präfixfreiheit ist eine wichtige Eigenschaft für die Decodierung, d.h. die Rückübersetzung codierter Information. (Z.B. sind alle Sprachen präfixfrei, deren Wörter die selbe Länge haben.)

vgl. z.B. Huffman-Code

Aus gegebenen Sprachen $L, L_1, L_2 \subseteq \Sigma^*$ lassen sich weitere Sprachen wie folgt zusammensetzen:

$$L_1 \cdot L_2 := \{w_1 \cdot w_2 \,:\, w_1 \in L_1, w_2 \in L_2\} \qquad \text{(Produktsprache)}$$

$$L^0 := \{\varepsilon\} \qquad \text{(0-faches Produkt)}$$

$$L^k := L \cdot L^{k-1} = \{w_1 \cdot \ldots \cdot w_k \,:\, w_i \in L\} \qquad \text{(k-faches Produkt, } k \in \mathbb{N}\text{)}$$

$\mathbb{N} := \{1, 2, 3, \ldots\}$
$\mathbb{N}_0 := \{0\} \cup \mathbb{N}$

$$L^* := \bigcup_{k \in \mathbb{N}_0} L^k \qquad \text{(Kleene'scher Abschluss)}$$

$$L^+ := \bigcup_{k \in \mathbb{N}} L^k \qquad \text{(Positiver Abschluss)}$$

$$\overline{L} := \Sigma^* \setminus L \qquad \text{(Komplementsprache)}$$

$$L_1/L_2 := \{w \in \Sigma^* \,:\, w \cdot z \in L_1 \text{ für ein } z \in L_2\}$$
(Quotientensprache, „L_1 modulo L_2“)

1.9 Bemerkung

Die schon eingeführte Menge Σ^* aller Wörter über dem Alphabet Σ ist der Kleene'sche Abschluss von Σ.

1.10 Beispiel

Seien $L_1 = \{0, 10\}$ und $L_2 = \{\varepsilon, 0\}$ über $\Sigma = \{0, 1\}$ definiert. Dann sind

$$\begin{aligned} L_1 \cdot L_2 &= \{0, 10, 00, 100\} \\ L_1^* &= \Big\{ \underbrace{\varepsilon,}_{L_1^0} \underbrace{0, 10,}_{L_1^1} \underbrace{00, 100, 010, 1010,}_{L_1^2} \ldots \Big\} \\ L_1/L_2 &= \{\varepsilon, 1, 0, 10\} \end{aligned}$$

Kapitel 2

Reguläre Sprachen

2.1 Reguläre Ausdrücke

Für die symbolische Verarbeitung von Informationen mit Hilfe von Rechnern ist es zweckmäßig, wenn formale Sprachen selbst durch Zeichenketten repräsentiert werden können.

2.1 Definition

- *Die Menge der regulären Ausdrücke über einem Alphabet Σ ist induktiv definiert:*

  Klammern zur besseren Lesbarkeit erlaubt

  - $\boldsymbol{\emptyset}$, $\boldsymbol{\varepsilon}$ *sind reguläre Ausdrücke*
  - $\boldsymbol{a}$ *ist ein regulärer Ausdruck, falls* $a \in \Sigma$
  - *sind* α, β *reguläre Ausdrücke, dann auch* $\alpha + \beta$, $\alpha \cdot \beta$, α^+ *und* α^*

- *Die durch einen regulären Ausdruck beschriebene Sprache ist definiert durch:*

$$\begin{aligned} L(\boldsymbol{\emptyset}) &= \emptyset \\ L(\boldsymbol{\varepsilon}) &= \{\varepsilon\} \\ L(\boldsymbol{a}) &= \{a\} \text{ für } a \in \Sigma \end{aligned}$$

$$\begin{aligned} L(\alpha + \beta) &= L(\alpha) \cup L(\beta) \\ L(\alpha \cdot \beta) &= L(\alpha) \cdot L(\beta) \\ L(\alpha^+) &= L(\alpha)^+ \\ L(\alpha^*) &= L(\alpha)^* \end{aligned}$$

- *Eine Sprache heißt regulär genau dann, wenn sie durch einen regulären Ausdruck beschrieben werden kann.*

2.2 Bemerkung
Der Einfachheit halber schreiben wir häufig nur α statt $L(\alpha)$ und entsprechend $w \in \alpha$ statt $w \in L(\alpha)$.

2.3 Beispiel

a) *Die Sprache $L \subseteq \{0,1\}^*$ aller Binärwörter, die als vorletztes Zeichen eine 0 haben, wird durch den regulären Ausdruck $(0+1)^*0(0+1)$ beschrieben.*

b) *$(0+1)^*10(0+1)^*$ beschreibt alle Binärwörter, die das Teilwort 10 enthalten. Die Komplementärmenge dazu beschreibt 0^*1^*, denn*

i) *$w \in 0^*1^* \implies 10$ nicht Teilwort von w*

ii) *enthält w nicht das Teilwort 10, dann folgt nach der ersten 1 in w keine 0 mehr $\implies w \in 0^*1^*$*

c) *$(0+1)^*101(0+1)^*$ beschreibt alle Binärwörter, die das Teilwort 101 enthalten. Gibt es auch hier einen regulären Ausdruck, der die Komplementsprache $L = \overline{(0+1)^*101(0+1)^*}$ beschreibt?*

Betrachte ein $w \in L$, das eine beliebige Anzahl k von Teilwörtern 10 habe. Ist $k > 0$, lässt sich w schreiben als

$$w = w_1 \cdot 10 \cdot w_2 \cdot 10 \cdot \ldots \cdot w_k \cdot 10 \cdot w_{k+1} = \left(\prod_{i=1}^{k} (w_i \cdot 10) \right) w_{k+1}$$

wobei die w_i nicht 10 enthalten. Da 101 in w nicht vorkommt, beginnen alle $w_i, i = 2, \ldots, k$, mit einer 0. Lediglich w_1 kann mit einer 1 beginnen und w_1 und w_{k+1} können auch leer sein. Also gilt

$$w = \left(\prod_{i=1}^{k-1} (v_i \cdot 100) \right) v_k 10 w_{k+1}$$

mit $v_i \in 0^*1^*$, $i = 1, \ldots, k$, und $w_{k+1} \in \varepsilon + 0(0^*1^*)$. Falls w genau $k > 0$ Vorkommen von 10 enthält, folgt daher siehe b)

$$w \in \left(\prod_{i=1}^{k-1} 0^*1^*100 \right) 0^*1^*10(\varepsilon + 00^*1^*)$$

und bei genau $k = 0$ Vorkommen ist wieder mit b)

$$w \in 0^*1^* \ .$$

Insgesamt erhalten wir $L = 0^*1^* + (0^*1^*100)^*0^*1^*10(\varepsilon + 00^*1^*)$.

2.4 Beispiel (Perl regular expressions)

Varianten regulärer Ausdrücke sind als sogenannte wildcards *aus Kommandozeilenumgebungen und Texteditoren (Suchen & Ersetzen) bekannt. Perl ist eine Skriptsprache, die eine sehr komfortable Spezifikation von regulären Ausdrücken unterstützt.*

UNIX: `man perlre`

Sei $\Sigma =$ `ASCII`*, dann steht jedes alphanumerische Zeichen für sich selbst (wie oben). Ausnahmen sind die Zeichen*

`+, ?, ., *, ^, $, (, ), [, ], {, }, \, |` ,

die zur Kombination der Ausdrücke benötigt werden. Ihre Sonderbedeutung kann durch ein vorangestelltes `\` *umgangen werden.* `+` *und* `*` *stehen für den positiven bzw. Kleene'schen Abschluss,* `|` *steht für Alternativen (oben:* $+$*) und das Produkt (oben:* $\cdot$*) ist implizit.*

Darüber hinaus sind zahlreiche Abkürzungen definiert, so steht `.` *z.B. für ein beliebiges Zeichen,* `\s` *für einen Leerraum (*whitespace*) und* `(...)?` *ist kurz für die Alternative „leeres Wort oder der Ausdruck in der Klammer".*

Die Sprache der Binärwörter ohne Teilwort `101` *ist daher beschrieben durch die Perl regular expression*

```
(0*1* | (0*1*100)*0*1*10(00*1*)?) .
```

Die durch reguläre Ausdrücke beschriebenen regulären Sprachen sind damit alle Sprachen, die aus den Sprachen

$$\emptyset \text{ und } \{a\}, a \in \Sigma \qquad \text{(Verankerung)}$$

durch endlich viele Operationen der Form

$$\left.\begin{array}{rr} L_1 \cdot L_2 & \text{(Produkt)} \\ L_1 \cup L_2 & \text{(Vereinigung)} \\ \text{oder} \quad L^* & \text{(Kleene'scher Abschluss)} \end{array}\right\} \qquad \text{(Induktion)}$$

hervorgehen.

$\{\varepsilon\} = \emptyset^*$

Es drängen sich einige Fragen auf:

- Sind alle Sprachen regulär?
- Sind die regulären Sprachen abgeschlossen unter Komplementbildung, Durchschnitt, Quotientenbildung?
- Kann man (effizient) entscheiden, ob ein gegebenes Wort in einer gegebenen (regulären) Sprache enthalten ist („Wortproblem“)? Wie?

Suchen & Ersetzen mit reg. Ausdr.

2.5 Satz
Jede endliche Sprache ist regulär.

■ **Beweis:** Selbst. □ Übung

2.2 Endliche Automaten

Wir betrachten ein sehr einfaches Maschinenmodell, von dem sich zeigen wird, dass es ausreicht, um das Wortproblem für reguläre Sprachen (effizient) zu lösen.

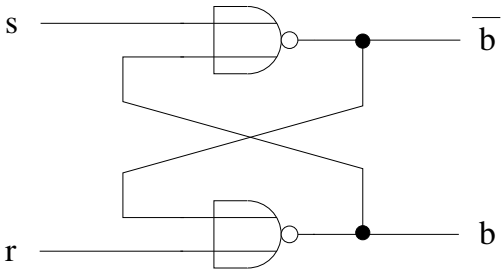
2.6 Definition
Ein deterministischer endlicher Automat (DEA) $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ besteht aus:

- *einer endlichen Menge Q von Zuständen*
- *einem Alphabet Σ (auch: Eingabesymbole)*
- *einer Übergangsfunktion $\delta : Q \times \Sigma \to Q$ (auch: Transitionsfunktion)*
- *einem Startzustand $s \in Q$*
- *einer Menge von Endzuständen $F \subseteq Q$*

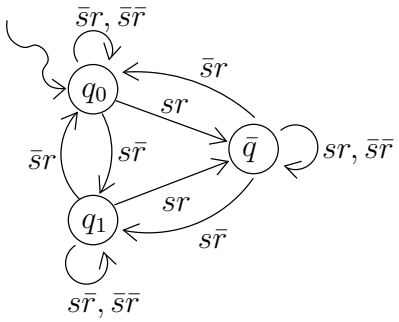
2.7 Bemerkung
$\mathcal{A}$ heisst:

- *endlich, weil Q endlich ist* endl. Speicher
- *deterministisch, weil δ eine Funktion ist (somit ist jeder Übergang eindeutig festgelegt)*

2.8 Beispiel (Flip-Flop)



„Implementation“



„Spezifikation“

2.9 Bemerkung

Zur einfacheren Beschreibung wird die Übergangsfunktion δ häufig nur für interessierende Konfigurationen $(q, a) \in Q \times \Sigma$ angegeben.

In solchen Fällen wird $\mathcal{A}$ implizit durch einen neuen Zustand $\bar{q} \notin Q$ (den Fehlerzustand) vervollständigt. Ist $\delta(q, a)$ nicht angegeben, so werden $\delta(q, a) = \bar{q}$ und generell $\delta(\bar{q}, a) = \bar{q}$ für alle $a \in \Sigma$ vereinbart. Der Fehlerzustand $\bar{q}$ ist dann natürlich kein Endzustand.

Die Übergangsfunktion lässt sich auf $Q \times \Sigma^*$ erweitern, indem wir

$$\begin{aligned} \delta(q, \varepsilon) &= q \\ \text{und} \quad \delta(q, aw) &= \delta(\delta(q, a), w) \text{ für } w \in \Sigma^* \end{aligned}$$

vereinbaren.

2.10 Definition

Ein DEA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ akzeptiert (erkennt) ein Wort $w \in \Sigma^*$, falls $\delta(s, w) \in F$. Die von $\mathcal{A}$ erkannte Sprache ist die Menge $L(\mathcal{A}) := \{w \in \Sigma^* : \delta(s, w) \in F\}$ der akzeptierten Wörter.

2.11 Bemerkung

Zwei mögliche Sichtweisen:

- *Erkennen/Testen von Eingaben*
- *Generieren von Wörtern einer Sprache (benötigt keinen Fehlerzustand)*

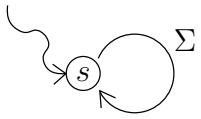
Wir wollen nun den Zusammenhang mit den regulärem Sprachen herstellen. Der folgende Satz besagt, dass deterministische endliche Automaten ausreichend mächtig sind, um das Wortproblem für reguläre Sprachen effizient zu entscheiden.

2.12 Satz
Zu jeder regulären Sprache L gibt es einen deterministischen endlichen Automaten $\mathcal{A}$ mit $L = L(\mathcal{A})$.

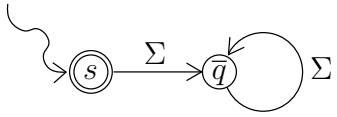
■ **Beweis:** Sei α ein regulärer Ausdruck, der L beschreibt. Wir führen den Beweis induktiv über die Anzahl n der Vorkommen von $+$, $\cdot$ und $*$ in α.

$n = 0$**:** (Induktionsanfang)
α ist entweder $\boldsymbol{\emptyset}$, $\boldsymbol{\varepsilon}$, oder $\boldsymbol{a}$ für ein $a \in \Sigma$. Daher wird L durch einen der folgenden DEAn akzeptiert:

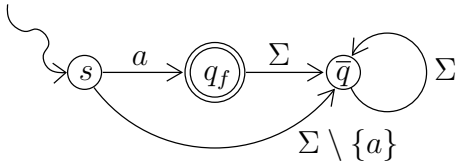
$L = \emptyset$:



$L = \varepsilon$:



$L = a$:

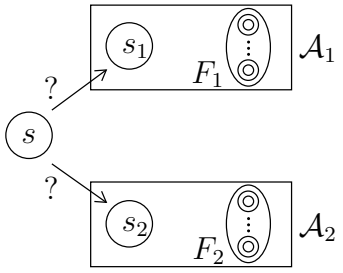


$n > 0$**:** (Induktionsschluss)
α ist von der Form $\alpha_1 + \alpha_2$, $\alpha_1 \cdot \alpha_2$ oder ${\alpha_1}^*$ und wir nehmen an, dass es Automaten $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ und $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ mit $L(\mathcal{A}_1) = \alpha_1$ und $L(\mathcal{A}_2) = \alpha_2$ gibt (Induktionsannahme).

Ein DEA $\mathcal{A}$ zur Erkennung von L bräuchte sich also nur wie $\mathcal{A}_1$ zu verhalten, falls $w \in \alpha_1$ und wie $\mathcal{A}_2$, falls $w \in \alpha_2$. Sei daher $\mathcal{A} = (\{s\} \dot\cup Q_1 \dot\cup Q_2, \Sigma, \delta, s, F_1 \cup F_2)$, wobei die Zustände so umbenannt seien,

dass $Q_1 \cap Q_2 = \emptyset$ und

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1 \\ \delta_2(q,a) & \text{falls } q \in Q_2 \\ ??? & \text{falls } q = s \end{cases}$$



Und jetzt?

An dieser Stelle stockt die Beweisführung, denn wir müssten schon im Startzustand „wissen“, welcher Teilautomat das Wort erkennt, und dürfen beim Übergang in dessen Startzustand auch kein Zeichen der Eingabe verbrauchen.

Wir führen daher zunächst ein handlicheres Maschinenmodell ein, mit dem wir den Beweis dann später forführen können... □

2.13 Definition
Ein nichtdeterministischer endlicher Automat (NEA) $\mathcal{A} = (Q\,\Sigma, \delta, s, F)$ besteht aus

- *einer endlichen Zustandsmenge Q*
- *einem Alphabet Σ*
- *einer Übergangsfunktion $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$*

Potenzmenge

- *einem Startzustand s*
- *einer Endzustandsmenge $F \subseteq Q$*

2.14 Bemerkung
Ein NEA unterscheidet sich von einem DEA also durch die Übergangsfunktion, die es erlaubt, „spontane“ ε-Übergänge (ohne Verbrauch eines Eingabesymbols) festzulegen, und außerdem die Wahl zwischen mehreren zulässigen Folgezuständen offen lässt.

Auch für NEAn nehmen wir eine implizite Vervollständigung an (wobei wir statt eines Fehlerzustands $\delta(q,a) = \emptyset$ bzw. $\delta(q,\varepsilon) = q$ für nicht angegebene

Konfigurationen $(q,a),(q,\varepsilon) \in Q \times (\Sigma \cup \{\varepsilon\})$ vereinbaren) und erweitern δ auf $Q \times \Sigma^*$, allerdings ist dabei der Nichtdeterminismus des Folgezustandes zu berücksichtigen:

$$\delta(q,aw) = \bigcup_{q' \in \delta(q,a)} \delta(q',w) \qquad \text{für alle } q \in Q, a \in \Sigma \cup \{\varepsilon\} \ .$$

2.15 Definition
Ein NEA $\mathcal{A} = (Q,\Sigma,\delta,s,F)$ erkennt (akzeptiert) ein Wort $w \in \Sigma^*$, falls $\delta(s,w) \cap F \neq \emptyset$. Die von $\mathcal{A}$ erkannte (akzeptierte) Sprache ist

$$L(\mathcal{A}) := \{w \in \Sigma^* \ : \ \delta(s,w) \cap F \neq \emptyset\}$$

Nichtdeterministische endliche Automaten können genau das, was uns im obigen Beweis gefehlt hat. Damit erhalten wir zumindest schon mal den folgenden Satz.

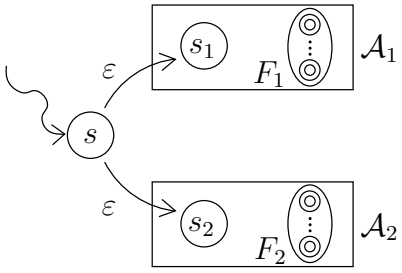
2.16 Satz
Zu jeder regulären Sprache L gibt es einen nichtdeterministischen endlichen Automaten $\mathcal{A}$ mit $L(\mathcal{A}) = L$.

■ **Beweis:** Wir setzen den Beweis an wie in Satz 2.12. Da es zu jedem DEA einen entsprechenden NEA gibt, können wir gleich mit dem Induktionsschluss weitermachen. Seien also α ein regulärer Ausdruck für L von der Form $\alpha_1+\alpha_2$, $\alpha_1 \cdot \alpha_2$ oder α_1^*, und $\mathcal{A}_i = (Q_i,\Sigma,\delta_i,s_i,F_i)$ NEAn mit $L(\mathcal{A}_i) = \alpha_i$, $i = 1,2$.

1. Fall: $\alpha = \alpha_1 + \alpha_2$

Wir konstruieren $\mathcal{A} = (Q,\Sigma,\delta,s,F)$ mit $Q = Q_1 \dot\cup Q_2 \dot\cup \{s\}$ (evtl. Umbenennung der Zustände), $F = F_1 \cup F_2$ und

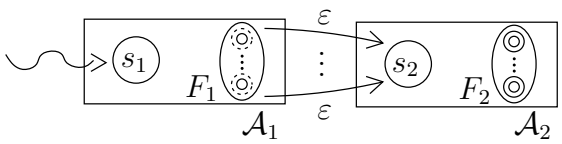
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1 \\ \delta_2(q,a) & \text{falls } q \in Q_2 \\ \{s_1,s_2\} & \text{falls } q = s, a = \varepsilon \\ \emptyset & \text{falls } q = s, a \neq \varepsilon \end{cases}$$



2. Fall: $\alpha = \alpha_1 \cdot \alpha_2$

Wir konstruieren $\mathcal{A} = (Q, \Sigma, \delta, s_1, F_2)$ durch $Q = Q_1 \dot\cup Q_2$ (evtl. umbenennen) und

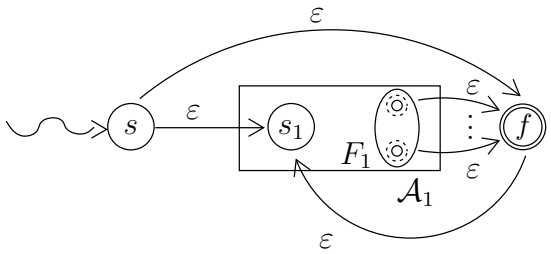
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1 \setminus F_1 \text{ oder } q \in Q_1, a \neq \varepsilon \\ \delta_2(q,a) & \text{falls } q \in Q_2 \\ \delta_1(q,a) \cup \{s_2\} & \text{falls } q \in F_1 \text{ und } a = \varepsilon \end{cases}$$



3. Fall: $\alpha = \alpha_1^*$

Wir konstruieren $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ durch $Q = Q_1 \dot\cup \{s, f\}$ (evtl. umbenennen), $F = \{f\}$ und

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1 \setminus F_1 \text{ oder } q \in Q_1, a \neq \varepsilon \\ \delta_1(q,a) \cup \{f\} & \text{falls } q \in F_1, a = \varepsilon \\ \{s_1, f\} & \text{falls } q = s, a = \varepsilon \\ \{s_1\} & \text{falls } q = f, a = \varepsilon \\ \emptyset & \text{sonst} \end{cases}$$



In allen drei Fällen gilt $L(\mathcal{A}) = \alpha$. $\square$

2.17 Fragen

1. *Sind nichtdeterministische endliche Automaten mächtiger als deterministische?*

2. *Erkennen (nicht)deterministische endliche Automaten mehr als nur reguläre Sprachen?*

Wir zeigen zunächst, dass NEAn nicht mächtiger sind als DEAn, und dann, dass jede von einem DEA akzeptierte Sprache regulär ist.

2.18 Definition
Zwei endliche Automaten heissen äquivalent, falls sie dieselbe Sprache akzeptieren.

Wir werden benutzen, dass ε-Übergänge zwar bequem sind, man aber auch ohne sie auskommt.

2.19 Definition
Für einen nichtdeterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ und einen Zustand $q \in Q$ heisst die Menge

$$E(q) = \{q' \in Q : q' \text{ ist von } q \text{ durch } \varepsilon\text{-Übergänge erreichbar}\}$$

der ε-Abschluss (die ε-Hülle) von q.

2.20 Bemerkung
Es gilt $E(q) \subseteq Q$ (also auch $E(q) \in 2^Q$) und $q \in E(q)$. Die ε-Hülle einer Zustandsmenge $P \subseteq Q$ wird mit $E(P) = \bigcup_{q \in P} E(q)$ bezeichnet.

2.21 Satz
Zu jedem nichtdeterministischen endlichen Automaten existiert ein äquivalenter deterministischer endlicher Automat.

■ **Beweis:** (Potenzmengenkonstruktion)
Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ ein beliebiger NEA. Wir konstruieren einen DEA $\widetilde{\mathcal{A}} = (\widetilde{Q}, \Sigma, \widetilde{\delta}, \widetilde{s}, \widetilde{F})$ mit wesentlich mehr Zuständen, um die Wahlmöglichkeiten des NEA zu simulieren. Dazu definieren wir

$\widetilde{Q} := 2^Q$ d.h. für $\widetilde{q} \in \widetilde{Q}$ ist $\widetilde{q} \subseteq Q$.
(Idee: der NEA ist in einem der Zustände aus $\widetilde{q}$)

$\widetilde{s} := E(s)$ (Idee: ohne ein Zeichen der Eingabe zu lesen kann der NEA bereits in jeden Zustand des ε-Abschlusses von s übergehen)

$\widetilde{F} := \left\{\widetilde{q} \in \widetilde{Q} \,:\, \widetilde{q} \cap F \neq \emptyset\right\}$
(Idee: der NEA kann sich in einem Endzustand befinden)

Wir müssen nun noch $\widetilde{\delta}$ so definieren, dass ε-Übergänge und Nichtdeterminismus korrekt simuliert werden. Da $\widetilde{\mathcal{A}}$ ein DEA ist, hat die Übergangsfunktion die Signatur $\widetilde{\delta} : \widetilde{Q} \times \Sigma \to \widetilde{Q}$, d.h. $\widetilde{\delta} : 2^Q \times \Sigma \to 2^Q$.

Ist $\mathcal{A}$ in einem Zustand $q \in Q$ und ist das nächste Eingabesymbol $a \in \Sigma$, dann kann $\mathcal{A}$ zunächst in irgendeinen Zustand aus $E(q)$ übergehen, dann (nichtdeterministisch) in irgendeinen durch Lesen von a erreichbaren Zustand wechseln und schließlich wieder in irgendeinen Zustand aus dessen ε-Abschluss übergehen. Wir definieren deshalb

$$\widetilde{\delta}(\widetilde{q}, a) := \bigcup_{q \in \widetilde{q}} E\Big(\bigcup_{q' \in E(q)} \delta(q', a)\Big) \quad \text{für alle } \widetilde{q} \in \widetilde{Q},\ a \in \Sigma$$

und erweitern wie üblich auf $\widetilde{Q} \times \Sigma^* \to \widetilde{Q}$.

Schließlich zeigen wir durch Induktion über $|w|$, dass $\widetilde{\delta}(\widetilde{s}, w) = \delta(s, w)$ für alle $w \in \Sigma^*$, woraus dann

$$w \in L(\widetilde{\mathcal{A}}) \iff \widetilde{\delta}(\widetilde{s}, w) \in \widetilde{F} \iff \delta(s, w) \cap F \neq \emptyset \iff w \in L(\mathcal{A})$$

und damit $L(\widetilde{\mathcal{A}}) = L(\mathcal{A})$ folgt.

$\underline{|w| = 0}$**:** (Induktionsanfang)
Es muss dann $w = \varepsilon$ sein und damit $\widetilde{\delta}(\widetilde{s}, w) = \widetilde{s} = E(s) = \delta(s, w)$.

$\underline{|w| > 0}$**:** (Induktionsschluss)
Sei $w = va$ mit $a \in \Sigma$, $v \in \Sigma^*$. Wir können annehmen, dass $\widetilde{\delta}(\widetilde{s}, v') =$

$\delta(s, v')$ für alle $v' \in \Sigma^*$ mit $|v'| < |w|$ (Induktionsannahme).

$$\begin{aligned}
\widetilde{\delta}(\widetilde{s}, w) &= \widetilde{\delta}\left(\widetilde{\delta}(\widetilde{s}, v), a\right) \\
&= \widetilde{\delta}\left(\delta(s, v), a\right) \\
&= \bigcup_{q \in \delta(s,v)} E\left(\bigcup_{q' \in E(q)} \delta(q', a)\right) = \delta(s, w).
\end{aligned}$$

□

2.22 Bemerkung

- *Mit dem obigen Satz haben wir insbesondere den abgebrochenen Beweis von Satz 2.12 vervollständigt.*
- *Die Frage, ob ein NEA ein Wort akzeptiert, kann durch Ausprobieren aller möglichen Übergänge beantwortet werden. Die Potenzmengenkonstruktion ist ein Beispiel für den Platz-Zeit-Spagat, da die Alternativen in den Zuständen repräsentiert sind – eine verringerte Rechenzeit also durch erhöhten Platzbedarf erkauft wird.*
- *Zu einem NEA mit $n = |Q|$ Zuständen liefert die Potenzmengenkonstruktion einen DEA mit 2^n Zuständen.*

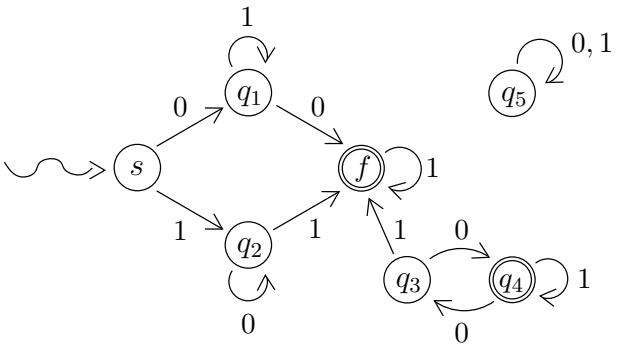
2.23 Frage

Lässt sich die Anzahl der Zustände eines deterministischen endlichen Automaten verringern? Wie?

2.24 Definition

Ein Zustand q eines endlichen Automaten heißt überflüssig, wenn es kein Wort $w \in \Sigma^*$ mit $\delta(s, w) = q$ gibt, d.h. q vom Startzustand aus nicht erreichbar ist.

2.25 Beispiel



$L(\mathcal{A}) = L(\mathbf{01}^*\mathbf{0} + \mathbf{10}^*\mathbf{1})\mathbf{1}^*)$

überflüssig: q_3, q_4, q_5

2.26 Satz

Die Menge der nicht-überflüssigen Zustände eines deterministischen endlichen Automaten kann in $\mathcal{O}(|Q| \cdot |\Sigma|)$ *Zeit bestimmt werden.*

■ **Beweis:** Ein endlicher Automat definiert einen gerichteten Graphen, in dem die Knoten den Zuständen entsprechen und die Kanten den Zustandsübergängen gemäß δ. Eine beim Startzustand beginnende Tiefensuche liefert die erreichbaren Zustände, alle so nicht erreichten sind überflüssig. Die Laufzeit ist daher durch die Anzahl Kanten des Graphen beschränkt. □

2.27 Beispiel (Gitterautomat)

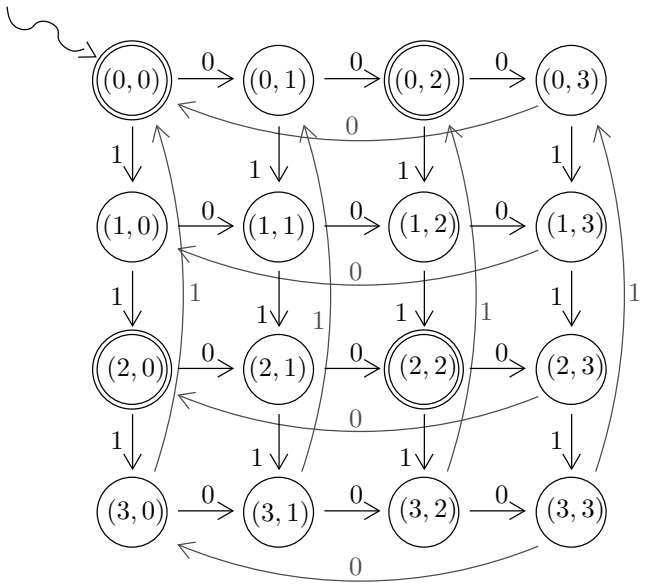
Sei $L = \{w \in \{0,1\}^* : |w|_0 \equiv |w|_1 \equiv 0 \mod 2\}$ *die Sprache aller Binärwörter, in denen jeweils eine gerade Anzahl von Nullen und Einsen vorkommt. Der Automat* $\mathcal{A} = \left(\{0,\ldots,3\}^2, \{0,1\}, \delta, (0,0), F\right)$ *mit*

$$\begin{aligned} \delta((i,j),0) &= (i, j+1 \mod 4) \\ \delta((i,j),1) &= (i+1 \mod 4, j) \end{aligned}$$

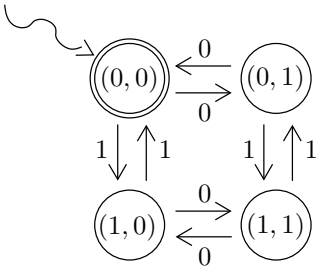
und

$$F = \{(i,j) \in Q : i \equiv j \equiv 0 \mod 2\}$$

akzeptiert L, *denn* $\delta((0,0),w) = (|w|_0 \mod 4, |w|_1 \mod 4)$.



Obwohl $\mathcal{A}$ keine überflüssigen Zustände hat, gibt es einen Automaten $\mathcal{B}$ mit weniger Zuständen und $L = L(\mathcal{A}) = L(\mathcal{B})$:



2.28 Definition
Zwei Zustände p, q eines deterministischen endlichen Automaten heißen äquivalent, $p \equiv q$, falls

$$\delta(p, w) \in F \iff \delta(q, w) \in F$$

für alle Wörter $w \in \Sigma^*$ gilt.

2.29 Bemerkung

- *$\equiv$ ist eine Äquivalenzrelation.*
- *Die Restklasse der zu $q \in Q$ äquivalenten Zustände wird mit $[q]$ bezeichnet.*

Da äquivalente Zustände dasselbe „Akzeptanzverhalten“ haben, brauchen wir sie nicht zu unterscheiden.

2.30 Definition
Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ ein deterministischer endlicher Automat. Der Äquivalenzklassenautomat $\mathcal{A}^{\equiv} = (Q^{\equiv}, \Sigma, \delta^{\equiv}, s^{\equiv}, F^{\equiv})$ zu $\mathcal{A}$ ist definiert durch:

$$\begin{aligned} Q^{\equiv} &:= \{[q] : q \in Q\} \\ \delta^{\equiv}([q], a) &:= [\delta(q, a)] \\ s^{\equiv} &:= [s] \\ F^{\equiv} &:= \{[q] : q \in F\} \end{aligned}$$

2.31 Satz
Der Äquivalenzklassenautomat $\mathcal{A}^{\equiv}$ zu einem deterministischen endlichen Automaten $\mathcal{A}$ ist ein deterministischer endlicher Automat.

■ **Beweis:** Wir brauchen nur zu zeigen, dass $F^{\equiv}$ und $\delta^{\equiv}$ wohldefiniert sind.

a) Ein Endzustand kann nur zu einem Endzustand äquivalent sein:
Aus $p \equiv q$ folgt $\delta(p, \varepsilon) \in F \iff \delta(q, \varepsilon) \in F$. Weil aber $\delta(f, \varepsilon) \in F \iff f \in F$ gilt entweder $p, q \in F$ oder $p, q \notin F$, $F^{\equiv}$ ist also wohldefiniert.

b) δ überführt äquivalente Zustände (beim Lesen desselben Symbols) in äquivalente Zustände:
Für $p \equiv q$ ist $\delta(p, w) \in F \iff \delta(q, w) \in F$ für alle $w \in \Sigma^*$. Also gilt insbesondere $\delta(p, aw) \in F \iff \delta(q, aw) \in F$ für alle $a \in \Sigma$ und $w \in \Sigma^*$. Das bedeutet aber, dass $\delta(p, a) \equiv \delta(q, a)$ und damit $\delta^{\equiv}$ wohldefiniert ist.

□

2.32 Satz
Für einen deterministischen endlichen Automaten $\mathcal{A}$ gilt $L(\mathcal{A}^{\equiv}) = L(\mathcal{A})$, d.h. der Äquivalenzklassenautomat erkennt dieselbe Sprache.

■ **Beweis:** Seien $w \in \Sigma^*$ und $s = q_0, q_1, \ldots, q_n$ die von $\mathcal{A}$ bei der Abarbeitung von w durchlaufenen Zustände ($n = |w|$). Bei Abarbeitung von w durchläuft $\mathcal{A}^{\equiv}$ die Zustände $s^{\equiv} = [q_0], [q_1], \ldots, [q_n]$. Es gilt $w \in L(\mathcal{A}) \iff q_n \in F$ und $w \in L(\mathcal{A}^{\equiv}) \iff [q_n] \in F^{\equiv}$. Nach Definition von $\mathcal{A}^{\equiv}$ ist aber $q_n \in F \iff [q_n] \in F^{\equiv}$. □

2.33 Frage

Wie kann $\mathcal{A}^{\equiv}$ zu $\mathcal{A}$ konstruiert werden?

Interessant, da i.a. weniger Zustände

Für die Äquivalenz $p \equiv q$ zweier Zustände ist zu zeigen, dass

$$\delta(p,w) \in F \iff \delta(q,w) \in F \qquad \underline{\text{für alle } w \in \Sigma^*} \ .$$

Es ist daher einfacher, umgekehrt vorzugehen.

2.34 Definition

Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ ein deterministischer endlicher Automat. Ein Wort $w \in \Sigma^*$ heißt Zeuge für die Nicht-Äquivalenz zweier Zustände $p, q \in Q$, falls

$$\begin{aligned} &\delta(p,w) \in F,\ \delta(q,w) \notin F \quad \textit{oder} \\ &\delta(p,w) \notin F,\ \delta(q,w) \in F \ . \end{aligned}$$

Das Wort w heißt kürzester Zeuge für $p \not\equiv q$, falls

$$\delta(p,w') \in F \iff \delta(q,w') \in F \qquad \textit{für alle } w' \in \Sigma^* \textit{ mit } |w'| < |w| \ .$$

2.35 Lemma

Gibt es keinen kürzesten Zeugen der Länge n, dann gibt es auch keinen kürzesten Zeugen einer Länge $m > n$.

■ **Beweis:** Angenommen, es gibt keinen kürzesten Zeugen der Länge n, aber einen der Länge $m > n$. Sei $w = av$, $|w| = m$ ein solcher kürzester Zeuge für $p \not\equiv q$. Dann ist v kürzester Zeuge für $\delta(p,a) \not\equiv \delta(q,a)$ mit $|v| = m - 1$, denn gäbe es einen kürzeren Zeugen v' für $\delta(p,a) \not\equiv \delta(q,a)$, dann wäre av' auch ein kürzerer Zeuge für $p \not\equiv q$ als $av = w$.

Wendet man diesen Ansatz induktiv auf v an, erhält man im Widerspruch zur Annahme einen kürzesten Zeugen der Länge n. □

Wir brauchen also nur die Wörter aus Σ^* sortiert nach nicht-absteigender Länge auf ihre Eignung als Zeuge zu testen. Sobald es für eine kleinste Länge kein Wort gibt, dass Zeuge für die Nichtäquivalenz irgendeines Zustandspaares ist, haben wir alle benötigten Zeugen gefunden.

2.36 Beispiel (Fortsetzung von 2.27)

- *ε ist (kürzester) Zeuge für $p \not\equiv q$ mit*
  $p \in \{00, 02, 20, 22\} = F$ und
  $q \in \{01, 03, 10, 11, 12, 13, 21, 23, 30, 31, 32, 33\} = Q \setminus F$

- 0 *ist (kürzester) Zeuge für* $p \not\equiv q$ *mit* $p \in \{01, 03, 21, 23\}$ *und* $q \in \{10, 11, 12, 13, 30, 31, 32, 33\}$
- 1 *ist (kürzester) Zeuge für* $p \not\equiv q$ *mit* $p \in \{10, 12, 30, 32\}$ *und* $q \in \{11, 13, 31, 33\}$
- $00, 01, 10, 11$ *sind keine kürzesten Zeugen*

$\curvearrowright$ *die Äquivalenzklassen der Zustände sind* $[00], [01], [10], [11]$, *der kleinere Automat* $\mathcal{B}$ *aus Beispiel 2.27 war also der zu* $\mathcal{A}$ *gehörige Äquivalenzklassenautomat.*

2.37 Frage
Ist $\mathcal{A}^{\equiv}$ *immer der äquivalente DEA mit minimaler Zustandsanzahl?*

2.38 Definition
Eine Äquivalenzrelation $\approx$ *über* Σ^* *(also* $\approx \subseteq \Sigma^* \times \Sigma^*$*) heißt rechtsinvariant, falls für alle* $w, w' \in \Sigma^*$ *mit* $w \approx w'$ *gilt* $wv \approx w'v$ *für alle* $v \in \Sigma^*$.

Die Anzahl $\operatorname{ind}(\approx)$ *der Restklassen von* $\approx$ *heißt Index von* $\approx$.

2.39 Definition
Die Nerode-Relation $\approx_L$ *einer Sprache* $L \subseteq \Sigma^*$ *ist definiert durch*

$$(w \approx_L w') \iff (\text{für alle } v \in \Sigma^* \text{ ist } wv \in L \iff w'v \in L)$$

2.40 Bemerkung
Die Nerode-Relation $\approx_L$ *ist offensichtlich eine Äquivalenzrelation und rechtsinvariant:*

$$\begin{aligned} w \approx_L w' &\implies (wv \in L \iff w'v \in L \text{ für alle } v \in \Sigma^*) \\ &\implies (wvx \in L \iff w'vx \in L \text{ für alle } v, x \in \Sigma^*) \\ &\implies wv \approx_L w'v \text{ für alle } v \in \Sigma^*. \end{aligned}$$

2.41 Satz (Satz von Nerode)
Die folgenden Aussagen sind äquivalent:

(1) $L \subseteq \Sigma^*$ *wird von einem endlichen Automaten erkannt*

(2) $L \subseteq \Sigma^*$ *ist die Vereinigung einiger Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation mit endlichem Index*

(3) Die Nerode-Relation $\approx_L$ zu $L \subseteq \Sigma^*$ hat endlichen Index

■ **Beweis:**

(1) $\Longrightarrow$ (2)**:** Sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ der L akzeptierende endliche Automat, und sei $\approx_{\mathcal{A}}$ definiert durch

$$w \approx_{\mathcal{A}} w' \iff \delta(s, w) = \delta(s, w')$$

für alle $w, w' \in \Sigma^*$. Dann ist $\approx_{\mathcal{A}}$ eine rechtsinvariante Äquivalenzrelation mit endlichem Index (da $\text{ind}(\approx_{\mathcal{A}})$ gerade die Anzahl der nicht überflüssigen Zustände von $\mathcal{A}$ ist).
L ist die Vereinigung derjenigen Äquivalenzklassen von $\approx_{\mathcal{A}}$, die zu Endzuständen gehören.

(2) $\Longrightarrow$ (3)**:** Sei $\approx$ die rechtsinvariante Äquivalenzrelation zu L mit endlichem Index. Wir zeigen $w \approx w' \implies w \approx_L w'$ (d.h. die Nerode-Relation $\approx_L$ ist eine Vergröberung von $\approx$), woraus $\text{ind}(\approx_L) < \text{ind}(\approx)$ folgt.
Sei also $w \approx w'$. Da $\approx$ rechtsinvariant ist, gilt $wv \approx w'v$ für alle $v \in \Sigma^*$. Nach Voraussetzung ist jede Äquivalenzklasse von $\approx$ ganz oder gar nicht in L, so dass $wv, w'v \in L$ oder $wv, w'v \notin L$ und damit $w \approx_L w'$.

(3) $\Longrightarrow$ (1)**:** Wir konstruieren zu $\approx_L$ einen deterministischen endlichen Automat, der L akzeptiert. Sei dazu $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit

- $Q := \{[w]_{\approx_L} : w \in \Sigma^*\}$
  (Menge der Restklassen bezüglich $\approx_L$, wegen des endlichen Indexes ist auch Q endlich.)
- $s := [\varepsilon]_{\approx_L}$
- $F := \{[w]_{\approx_L} : w \in L\}$
- $\delta([w]_{\approx_L}, a) := [wa]_{\approx_L}$
  (δ ist wohldefiniert, da $[w]_{\approx_L} = [w']_{\approx_L} \implies w \approx_L w'$ und wegen Rechtsinvarianz auch $wa \approx_L w'a$, so dass $[wa]_{\approx_L} = [w'a]_{\approx_L}$)

Nach Konstruktion ist $\delta(s, w) = \delta([\varepsilon]_{\approx_L}, w) = [\varepsilon w]_{\approx_L} = [w]_{\approx_L} \in F \iff w \in L$.

□

2.42 Korollar
Der (wie im letzten Beweisschritt konstruierte) Automat der Nerode-Relation einer Sprache $L \subseteq \Sigma^*$ ist zustandsminimal.

■ **Beweis:** Sei $\mathcal{A}$ der Nerode-Automat zu L und $\mathcal{A}' = (Q', \Sigma, \delta', s', F')$ ein anderer deterministischer endlicher Automat mit $L(\mathcal{A}') = L$.

Aus (1) $\Longrightarrow$ (2) folgt, dass es eine rechtsinvariante Äquivalenzrelation $\approx_{\mathcal{A}'}$ mit $\text{ind}(\approx_{\mathcal{A}'}) \leq |Q'|$ gibt. Wegen (2) $\Longrightarrow$ (3) ist $\text{ind}(\approx_L) \leq \text{ind}(\approx_{\mathcal{A}'})$. Und mit (3) $\Longrightarrow$ (1) folgt $|Q| = \text{ind}(\approx_L) \leq \text{ind}(\approx_{\mathcal{A}'}) \leq |Q'|$. □

2.43 Satz
Ist $\mathcal{A}$ ein deterministischer endlicher Automat ohne überflüssige Zustände, dann ist der Äquivalenzklassenautomat $\mathcal{A}^{\equiv}$ zustandsminimal.

■ **Beweis:** Offensichtlich hat $\mathcal{A}^{\equiv}$ keine überflüssigen Zustände. Wegen Korollar 2.42 genügt es zu zeigen, dass $|Q^{\equiv}| = \text{ind}(\approx_L)$ für $L = L(\mathcal{A}) = L(\mathcal{A}^{\equiv})$.
Für alle $w, w' \in \Sigma^*$ gilt aber

$$\begin{aligned} w \approx_L w' &\Longrightarrow (wv \in L \iff w'v \in L \text{ für alle } v \in \Sigma^*) \\ &\Longrightarrow (\delta(s, wv) \in F \iff \delta(s, w'v) \in F \text{ für alle } v \in \Sigma^*) \\ &\Longrightarrow \Big(\delta\big(\delta(s, w), v\big) \in F \iff \delta\big(\delta(s, w'), v\big) \in F \text{ für alle } v \in \Sigma^*\Big) \\ &\Longrightarrow \delta(s, w) \equiv \delta(s, w') \end{aligned}$$

sodass $|Q^{\equiv}| \leq \text{ind}(\approx_L)$. □

2.44 Satz
Die von einem endlichen Automaten $\mathcal{A}$ erkannte Sprache ist regulär.

■ **Beweis:** O.B.d.A. sei $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit $Q = \{q_1, \ldots, q_n\}$ und $s = q_1$ deterministisch. Für $i, j \in \{1, \ldots, n\}$ und $k \in \{0, \ldots, n\}$ sei

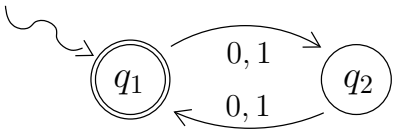
$$L_{ij}^k := \left\{ w \in \Sigma^* \;:\; \begin{array}{l} \delta(q_i, w) = q_j \text{ und keiner der Zustände } q_{k+1}, \ldots, q_n \\ \text{wird zwischendurch benutzt} \end{array} \right\}$$

Dann ist $L = L(\mathcal{A}) = \bigcup_{q_j \in F} L_{1j}^n$ und L ist regulär, wenn alle L_{1j}^n regulär sind.
Wir zeigen per Induktion über k, dass alle L_{ij}^k regulär sind:

$$\begin{aligned} \underline{k = 0:} \quad & L_{ij}^k \subseteq \Sigma \cup \{\varepsilon\} \text{ und damit regulär.} \\ \underline{(k-1) \to k:} \quad & L_{ij}^k = L_{ij}^{k-1} \cup L_{ik}^{k-1} \cdot \big(L_{kk}^{k-1}\big)^* \cdot L_{kj}^{k-1} \end{aligned}$$

□

2.45 Beispiel



Welche Sprache akzeptiert dieser Automat?

dynamische Programmierung

$$L^0_{11} = L^0_{22} = \{\varepsilon\} = \varepsilon\,, \qquad L^0_{12} = L^0_{21} = \{0,1\} = 0 + 1$$

$$\begin{aligned}
L^1_{11} &= L^0_{11} \cup L^0_{11} \cdot \left(L^0_{11}\right)^* \cdot L^0_{11} && = \varepsilon \\
L^1_{22} &= L^0_{22} \cup L^0_{21} \cdot \left(L^0_{11}\right)^* \cdot L^0_{12} && = \varepsilon + (0+1) \cdot \varepsilon^* \cdot (0+1) = \varepsilon + (0+1)(0+1) \\
L^1_{12} &= L^0_{12} \cup L^0_{11} \cdot \left(L^0_{11}\right)^* \cdot L^0_{12} && = (0+1) + \varepsilon \cdot \varepsilon^* \cdot (0+1) = 0 + 1 \\
L^1_{21} &= L^0_{21} \cup L^0_{21} \cdot \left(L^0_{11}\right)^* \cdot L^0_{11} && = (0+1) + (0+1) \cdot \varepsilon^* \cdot \varepsilon = 0 + 1
\end{aligned}$$

$$\begin{aligned}
L = L^2_{11} = L^1_{11} \cup L^1_{12} \cdot \left(L^1_{22}\right)^* \cdot L^1_{21} \quad &= \varepsilon + (0+1) \cdot \left(\varepsilon + (0+1)(0+1)\right)^* \cdot (0+1) \\
&= \left((0+1)(0+1)\right)^*
\end{aligned}$$

2.46 Korollar (Satz von Kleene)
Eine Sprache ist regulär genau dann, wenn sie von einem endlichen Automaten akzeptiert wird.

Wir haben reguläre Sprachen damit beschrieben über:

- Mengen für formale Behandlung
- reguläre Ausdrücke für symbolische Behandlung
- Automaten
  - nichtdeterministische für Spezifikation
  - deterministische für Implementation.

Es bleibt die Frage, ob es nicht-reguläre Sprachen gibt.

2.47 Beispiel

Sei $L \subseteq \{(,)\}^*$ die Sprache der korrekten Klammerausdrücke, also z.B. $(()())$, $(()()(()))) \in L$ und $((()$, $(()))()(\notin L$.

Ein Klammerausdruck $w \in \{(,)\}^*$ ist genau dann korrekt, wenn $|w|_(= |w|_)$ ÜBUNG
und $|v|_(\geq |v|_)$ für jedes Präfix v von w.

Ein L erkennender Automat müsste sich also „merken“ können, wieviele $($ und $)$ er bereits abgearbeitet hat. Da diese Anzahl beliebig groß sein kann, müsste der Automat über unendlich viele Zustände verfügen. L ist also eine Sprache, die nicht regulär ist.

2.48 Satz (Pumping-Lemma für reguläre Sprachen)

Sei L eine reguläre Sprache. Dann existiert eine Zahl $n \in \mathbb{N}$, sodass sich jedes Wort $w \in L$ mit $|w| \geq n$ darstellen lässt als

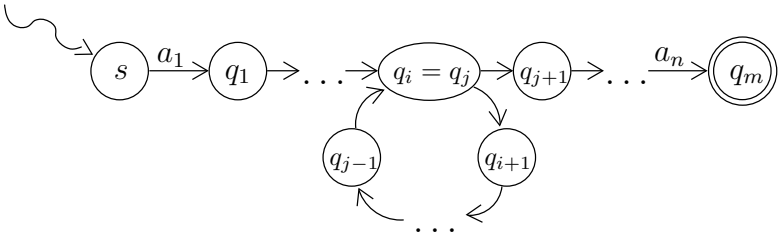
$$w = uvx \qquad \text{mit } |uv| \leq n, v \neq \varepsilon$$

und

$$uv^i x \in L \qquad \text{für alle } i \in \mathbb{N}_0 \ .$$

■ **Beweis:** Zu L gibt es einen det. endl. Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit $L = L(\mathcal{A})$. Wir setzen $n := |Q|+1$ und betrachten ein $w \in L$ mit $|w| \geq n$, etwa $w = a_1 \dots a_m$ mit $a_i \in \Sigma, i = 1, \dots, m \geq n$.

Bei der Abarbeitung von w durchlaufe $\mathcal{A}$ die Zustände $s = q_0, q_1, \dots, q_m$, d.h. $\delta(s, a_1 \dots a_i) = q_i, i = 0, \dots, m$ und $q_m \in F$. Wegen $m \geq n$ sind die durchlaufenen Zustände nicht alle verschieden. Also gibt es i, j mit $i < j \leq n$ und $q_i = q_j$.



Für die Zerlegung

$$w = \underbrace{a_1 \dots a_i}_{u} \cdot \underbrace{(a_{i+1} \dots a_j)}_{v} \cdot \underbrace{a_{j+1} \dots a_m}_{x}$$

gilt daher $|uv| \leq n$ (da $j \leq n$), $v \neq \varepsilon$ (da $i > j$) und $uv^i x \in L$ für alle $i \in \mathbb{N}_0$ (da der Zykel $q_i, \dots, q_j$ beliebig oft durchlaufen werden kann). □

2.49 Bemerkung
Die Bedingung im Pumping-Lemma ist notwendig, aber nicht hinreichend für die Regularität einer Sprache.

2.50 Beispiele

1. *Sei $L = 0^*1^*$ wieder die Sprache der Binärwörter ohne das Teilwort 10. Betrachte $n = 1$ und $w = uvx \in L$ mit $u = \varepsilon$. Dann ist v der erste Buchstabe von w und damit $uv^ix = v^ix \in L$ für alle $i \in \mathbb{N}_0$*

2. *Sei $L = \left\{0^k1^k \,:\, k \geq 0\right\} \subseteq \{0,1\}^*$. Diese Sprache ist nicht regulär, denn für ein beliebiges $n > 0$ sei $w = 0^n1^n$. Dann ist $|w| > n$ und für jede Darstellung $w = uvx$ mit $|uv| \leq n$ und $v \neq \varepsilon$ ist $v = 0^k$ für ein $k > 1$. Daraus folgt aber, dass $uv^0x = 0^{n-k}1^n \notin L$.*

3. *Um zu zeigen, dass die korrekten Klammerausdrücke keine reguläre Sprache bilden, wähle entsprechend $w = (^n)^n$.*

4. *Sei $\Sigma = \{0\}$ und $L = \left\{0^{k^2} \,:\, k \geq 0\right\} \subseteq \Sigma^*$ die Sprache aller Wörter über Σ mit quadratischer Länge. Auch L ist nicht regulär, denn für ein beliebiges $n > 1$ wähle $w = 0^{n^2}$, also $|w| > n$ (für $n = 1$ und $w = w^{(n+1)^2} = 0^4$ ist die Argumentation analog).*

   Für jede Zerlegung $w = uvx$ mit $|uv| \leq n$ und $v \neq \varepsilon$ gilt $1 \leq |v| \leq n$. Wegen $n^2 < n^2 + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$ ist dann $uv^2x = 0^{n^2+|v|} \notin L$.

5. *Sei $\Sigma = \{0,1\}$ und*

$$L = \left\{w \in \Sigma^* \,:\, w = 1^k, k > 0 \text{ oder } w = 0^j1^{k^2},\ j > 0, k \geq 0\right\}$$

   Dann erfüllt L die Bedingung des Pumping-Lemmas: Sei $n = 1$ und $w \in L$ mit $|w| > 1$. Wir stellen w durch $w = uvx$ mit $u = \varepsilon$ und $|v| = 1$ dar. Also ist v das erste Symbol von w und es gilt

   - *falls $w = 1^k$, so ist auch $uv^ix = 1^i1^{k-1} = 1^{k+i-1} \in L$* [da $k \geq 2$]
   - *falls $w = 0^j1^{k^2}$, so ist auch $uv^ix = 0^{j+i-1}1^{k^2} \in L$.*
     [Beachte, dass für $i = 0$, $j = 1$ gilt $uv^ix = 1^{k^2} \in L$.]

   Wegen 4. ist aber zu vermuten, dass L nicht regulär ist.

2.51 Satz (Verschärftes Pumping-Lemma für reguläre Sprachen)
Sei L eine reguläre Sprache. Dann existiert eine Zahl $n \in \mathbb{N}$, so dass für jedes Wort $w \in L$ mit $|w| \geq n$ und jede Darstellung $w = tyz$ mit $|y| = n$ eine Darstellung

$$y = uvx \qquad \text{mit } v \neq \varepsilon$$

existiert, für die $tuv^ixz \in L$ für alle $i \in \mathbb{N}_0$.

■ **Beweis:** Zu L gibt es einen deterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ mit $L = L(\mathcal{A})$. Wir setzen $n := |Q| + 1$ und betrachten ein $w = tyz \in L$ mit $|y| = n$.

Bei der Abarbeitung von y durchlaufe $\mathcal{A}$ die Zustände $q_0, \ldots, q_n$. Wie im Beweis des Pumping-Lemmas sind die q_i nicht alle verschieden und es existiert eine Zerlegung $y = uvx$, sodass $v \neq \varepsilon$ die Folge der Symbole auf einem Zykel in $\mathcal{A}$ ist. Dieser Zykel kann wieder beliebig oft durchlaufen werden und damit gilt $tuv^ixz \in L$ für alle $i \in \mathbb{N}_0$. □

2.52 Bemerkung
Mit dem verschärften Pumping-Lemma kann die Nicht-Regularität einer größeren Klasse von Sprachen gezeigt werden, es ist aber trotzdem keine äquivalente Bedingung für Regularität.

2.53 Beispiel (Fortsetzung von Beispiel 2.50/5.)
Zu $n > 1$ sei $w = 01^{n^2} = tyz$ mit $y = 1^n$. Dann folgt die Nichtregularität wie in 4.

2.3 Exkurs: Pattern-Matching

Gegeben:	Text $t = t_1 \cdots t_n \in \Sigma^*$
	Muster $p = p_1 \cdots p_m \in \Sigma^*$
	typischerweise $n \gg m$
Gesucht:	alle Vorkommen von p in t
	[naive Lösung hat *worst-case* Laufzeit $\Theta(n \cdot m)$]

Die Suffix-Funktion $\sigma_p : \Sigma^* \to \{0, \ldots, m\}$ ist definiert durch

$$\sigma_p(w) = \max\{l \in \mathbb{N}_0 \,:\, w = vp_1 \cdots p_l\}$$

d.h. $\sigma_p(w)$ ist die Länge des längsten Präfixes von p, das ein Suffix von w ist.

Konstruiere nun einen deterministischen endlichen Automaten $\mathcal{A}_p = (Q, \Sigma, \delta_p, s, F)$ durch

$$\begin{aligned} Q &= \{0, \ldots, m\} \\ s &= 0 \\ F &= \{m\} \\ \delta_p(i, a) &= \sigma_p(p_1 \ldots p_i a) \end{aligned}$$

Die Übergangsfunktion kann in $\mathcal{O}(m^3|\Sigma|)$ berechnet werden. ÜBUNG

Wir zeigen nun, dass $\mathcal{A}_p$ bei Eingabe von t nach Lesen von k Zeichen im Zustand $\delta_p(0, t_1 \ldots t_k) = \sigma_p(t_1 \ldots t_k)$ ist. Der Zustand zeigt damit das längste Anfangsstück des Musters an, mit dem der bisher gelesene Text endet.

2.54 Lemma
Für alle $w \in \Sigma^*$ und $a \in \Sigma$ gilt $\sigma_p(w) = i \implies \sigma_p(wa) = \sigma_p(p_1 \ldots p_i a)$.

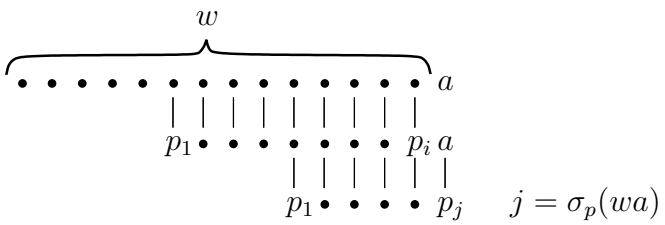
■ **Beweis:** Sei $w = w_1 \ldots w_k$, dann folgt aus $\sigma_p(w) = i$ zunächst

$$\begin{aligned} w_k &= p_i \\ w_{k-1} &= p_{i-1} \\ &\vdots \\ w_{k-i+1} &= p_1 \end{aligned}$$

Für $j = \sigma_p(wa) = \max\{l : wa = vp_1 \dots p_l\}$ folgt damit

$$\begin{aligned} p_j &= a \\ p_{j-1} &= w_k && = p_i \\ p_{j-2} &= w_{k-1} && = p_{i-1} \\ &\vdots && \vdots \\ p_1 &= w_{k-j+2} && = p_{i-j+2} \ , \end{aligned}$$

also $\sigma_p(wa) = j = \max\{l : p_1 \dots p_i a = vp_1 \dots p_l\} = \sigma_p(p_1 \dots p_i a)$.



□

2.55 Satz
$\delta_p(0, t_1 \dots t_k) = \sigma_p(t_1 \dots t_k)$ *für alle* $k = 0, \dots, n$.

■ **Beweis:** Induktion über die Länge des Eingabewortes (d.h. des gelesenen Textpräfixes). Zunächst gilt $\delta_p(0, \varepsilon) = 0 = \sigma_p(\varepsilon)$. Für $k > 0$ setze $i = \delta_p(0, t_1 \dots t_{k-1}) \underset{\text{I.V.}}{=} \sigma_p(t_1 \dots t_{k-1})$, dann gilt

$$\begin{aligned} \delta(0, t_1 \dots t_k) &= && \delta_p(i, t_k) \\ &\underset{\text{Def. } \delta_p}{=} && \sigma_p(p_1 \dots p_i t_k) \\ &\underset{\text{Lemma 2.54}}{=} && \sigma_p(t_1 \dots t_{k-1} t_k) \ . \end{aligned}$$

□

Mit $\mathcal{A}_p$ wird t nun wie folgt durchsucht:

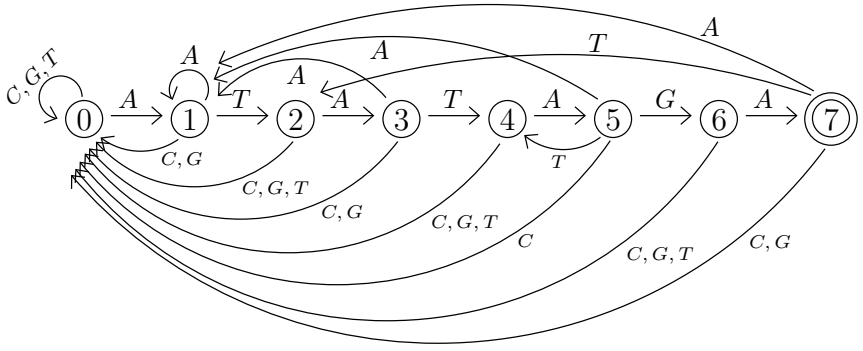
Algorithmus 1: Pattern Matching mit endlichen Automaten

Eingabe : Text $t = t_1 \cdots t_k \in \Sigma^*$
Muster $p = p_1 \cdots p_m \in \Sigma^*$

konstruiere $\mathcal{A}_p = (\{0, \ldots, m\}, \Sigma, \delta_p, 0, \{m\})$
$i \leftarrow 0$
for $k = 1, \ldots, n$ **do**
  $i \leftarrow \delta_p(i, t_k)$
  if $i = m$ **then**
    gib aus „Das Muster kommt ab der Stelle $k - m + 1$ vor“

Die Gesamtlaufzeit wurde damit reduziert auf $\mathcal{O}(\underbrace{m^3|\Sigma|}_{\substack{\text{Konstruktion} \\ \text{von } \mathcal{A}_p}} + \underbrace{n}_{\substack{\text{Text-} \\ \text{durchlauf}}})$.

2.56 Beispiel
Seien $p =$ `ATATAGA` *und* $\Sigma = \{$`A`, `C`, `G`, `T`$\}$



Kapitel 3

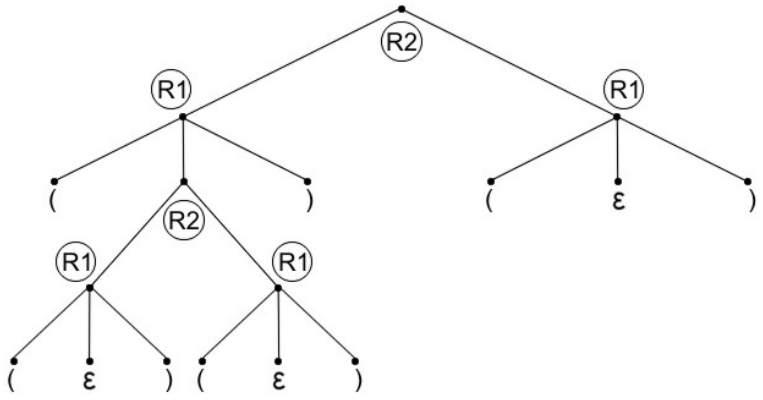
Kontextfreie Sprachen

Induktive Charakterisierung der Sprache L der korrekten Klammerausdrücke: s. Übung

- $\varepsilon \in L$ (R0)
- $(w) \in L$, falls $w \in L$ (R1)
- $vw \in L$, falls $v, w \in L$ (R2)

Für beliebige $w \in \{(,)\}^*$ kann $w \in L$ nachgewiesen werden durch Angabe der Regeln, nach denen w aufgebaut ist.

3.1 Beispiel
$w = (()())()$



Liest man diesen Nachweis umgekehrt, so wird ein noch unbekanntes Wort an der Wurzel entsprechend (R2) durch zwei unbekannte Wörter ersetzt, die dann wiederum durch andere Wörter ersetzt werden, bis an den Blättern ε oder Symbole des Alphabets die Ersetzung abschließen.

3.1 Kontextfreie Grammatiken

3.2 Definition
Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ besteht aus

- *einer endlichen Menge von Variablen (Nichtterminalsymbolen) V*
- *einem Alphabet (Terminalsymbolen) Σ*
- *einer endlich Menge von Ableitungsregeln (Produktionen) $P \subseteq V \times (V \cup \Sigma)^*$*
- *einem Startsymbol $S \in V$.*

Jedes $\alpha \in (V \cup \Sigma)^*$ heißt Satzform. Für Produktionen $(A, \alpha) \in P$ schreiben wir $A \to \alpha$, und für alternative Produktionen $A \to \alpha$, $A \to \beta$ auch kurz $A \to \alpha \,|\, \beta$.

3.3 Bemerkung
Anwenden einer Ableitungsregel $A \to \alpha$ bedeutet, in einer Satzform ein Vorkommen von A durch α zu ersetzen: $\beta_1 A \beta_2 \to \beta_1 \alpha \beta_2$ (bzw. $\beta_1 A \beta_2 \underset{G}{\to} \beta_1 \alpha \beta_2$).

Dies geschieht ohne Berücksichtigung des Kontextes β_1, β_2 von A. Geht eine Satzform β aus einer Satzform α durch Anwenden einer endlichen Anzahl von Ableitungsregeln hervor, schreiben wir $\alpha \overset{*}{\to} \beta$ (bzw. $\alpha \underset{G}{\overset{*}{\to}} \beta$).

3.4 Definition
Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ erzeugt ein Wort $w \in \Sigma^*$ (w kann in G abgeleitet werden), falls $S \overset{*}{\to} w$.
Die Menge $L(G) := \left\{ w \in \Sigma^* : S \overset{*}{\to} w \right\}$ ist die von G erzeugte Sprache.
Eine Sprache heißt kontextfrei, wenn sie von einer kontextfreien Grammatik erzeugt wird.

3.5 Beispiele

1. *Die Sprache der korrekten Klammerausdrücke wird von der Grammatik* $G = (\{S\}, \{(,)\}, P, S)$ *mit*

$$\begin{array}{lll} P: & S \to \varepsilon & (R0) \\ & S \to SS & (R1) \\ & S \to (S) & (R2) \end{array}$$

erzeugt. Z.B. ist $(()())() \in L(G)$, *da* $S \to SS \to (S)S \to (SS)S \to ((S)S)S \to (()S)S \to (()(S))S \to (()())S \to (()())(S) \to (()())()$.

2. *Für* $w = a_1 \cdots a_n$ *sei* $w^R := a_n a_{n-1} \cdots a_1$. *Die Sprache* $L = \left\{w \in \Sigma^* : w = w^R\right\}$ *der Palindrome über einem beliebigen Alphabet* Σ *ist nicht regulär. Es ist aber* $\overline{L} = L(G)$ *für* $G = (\{S\}, \Sigma, P, S)$ *mit* s. Übung

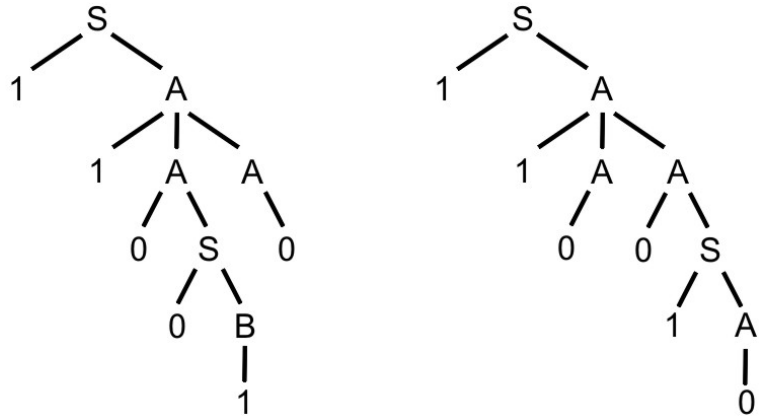
$$\begin{array}{lll} P: & S \to \varepsilon & \\ & S \to a & \text{für alle } a \in \Sigma \\ & S \to aSa & \text{für alle } a \in \Sigma \end{array}$$

3. *Sei* $L = \{w \in \{0,1\}^* : |w|_0 = |w|_1 > 0\}$ *die Sprache der nichtleeren Binärwörter mit gleich vielen 0'en und 1'en. Eine Grammatik, die* L *erzeugt, muss sich in jeder während einer Ableitung auftretenden Satzform „merken", wieviele 0'en oder 1'en zusätzlich benötigt werden, um das Verhältnis später wieder auszugleichen:* $G = (V, \Sigma, P, S)$ *mit*

Reihenfolge der Ableitungsschritte egal

$$V = \{A, B, S\} \text{ und } P: \quad \begin{array}{l} S \to 0B \mid 1A \\ A \to 0 \mid 0S \mid 1AA \\ B \to 1 \mid 1S \mid 0BB \end{array}$$

Ein Syntaxbaum ist die graphische Darstellung der Ableitung eines Wortes, z.B. für `110010` und die Grammatik aus Beispiel 3.5(3.):



3.6 Bemerkung

- *In Syntaxbäumen*
  - *ist die Wurzel mit dem Startsymbol markiert*
  - *ist jeder innere Knoten ein Nichtterminalsymbol und jedes Blatt ein Terminalsymbol*
  - *bilden die Kinder eines Nichtterminalsymbols die Satzform einer passenden rechten Regelseite*
- *Zu jeder Ableitung gehört genau ein Syntaxbaum, aber zu einem Syntaxbaum gehören mehrere Ableitungen, insbesondere eine Linksableitung (bei der jeweils das linkeste Nichtterminalsymbol als nächstes ersetzt wird) und eine Rechtsableitung (entsprechend).*
- *Zu einem von einer kontextfreien Grammatik erzeugten Wort kann es verschiedene Ableitungen mit verschiedenen Syntaxbäumen geben.*

3.7 Definition

Eine kontextfreie Grammatik G heißt eindeutig, falls es für jedes Wort $w \in L(G)$ genau einen Syntaxbaum gibt; andernfalls heißt G mehrdeutig.

Eine kontextfreie Sprache L heißt eindeutig, falls es eine eindeutige kontextfreie Grammatik gibt, die L erzeugt; andernfalls heißt L inhärent mehrdeutig.

Wir werden auf diese Begriffe zurückkommen, wollen jedoch zunächst das Wortproblem für kontextfreie Sprachen angehen.

3.8 Definition
Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ ist in Chomsky-Normalform, wenn alle Produktionen von der Form

$$A \to BC \quad \text{oder} \quad A \to a$$

mit $A, B, C \in V$ und $a \in \Sigma$ sind.

3.9 Bemerkung
Für Sprachen L, die das leere Wort enthalten, erhalten Grammatiken in Chomsky-Normalform, die $L \setminus \{\varepsilon\}$ erzeugen, ein neues Startsymbol S' und neue Produktionen

$$S' \to \varepsilon \mid S \ .$$

3.10 Satz
Jede kontextfreie Grammatik G kann in eine kontextfreie Grammatik G' in Chomsky-Normalform mit $L(G') = L(G) \setminus \{\varepsilon\}$ transformiert werden.

■ **Beweis:** G wird in 4 Schritten in G' transformiert:

1. Schritt: (Terminale treten nur allein auf der rechten Seite auf)
Führe neue Nichtterminale S_a für alle $a \in \Sigma$ ein. Danach ersetze alle Vorkommen von $a \in \Sigma$ auf der rechten Regelseite durch S_a und führe neue Produktionen $S_a \to a$ ein.

2. Schritt: (alle rechten Regelseiten haben höchstens zwei Symbole)
Für jede Produktion $P_i : A \to B_1 \dots B_m$ mit $m > 2$ führe neue Nichtterminale C^i_j, $j = 1, \dots, m-2$, ein und ersetze P_i durch

$$\begin{aligned} A &\to B_1 C^i_1 \\ C^i_j &\to B_{j+1} C^i_{j+1}, \quad j = 1, \dots, m-3 \\ C^i_{m-2} &\to B_{m-1} B_m \ . \end{aligned}$$

3. Schritt: (ε-Elimination)
Berechne zunächst die Menge V' der Nichtterminale, aus denen das

leere Wort abgeleitet werden kann, d.h. $A \in V' \iff A \stackrel{*}{\to} \varepsilon$. Dazu seien

$$\begin{aligned} V_0 &= \emptyset \\ V_1 &= \{A \in V : A \to \varepsilon \text{ in } P\} \\ V_{i+1} &= V_i \cup \{A \in V : A \to B \text{ oder } A \to BC \text{ mit } B, C \in V_i\} \quad \text{für alle } i \geq 1 \ . \end{aligned}$$

Für das kleinste $i \in \mathbb{N}_0$ mit $V_{i+1} = V_i$ gilt schon $V_i = V'$. Füge nun für alle Regeln $A \to BC$ mit $B \in V'$ (bzw. $C \in V'$) Regeln $A \to C$ (bzw. $A \to B$) hinzu und streiche alle Regeln der Form $A \to \varepsilon$.

4. Schritt: (Elimination von Kettenregeln $A \to B$)
Ausgehend von rechten Regelseiten der Form $a \in \Sigma$ und BC mit $B, C \in V$ berechne rückwärts alle Ableitungen der Form

$$A \to A_1 \to A_2 \to \ldots \to A_k \begin{array}{l} \nearrow a \\ \searrow BC \end{array}$$

ohne Wiederholungen. Füge für jede der obigen Ableitungen $A \to a$ bzw. $A \to BC$ ein und streiche alle Produktionen der Form $A \to B$.

Bis auf den Schritt 3 wird die erzeugte Sprache nicht verändert. Im 3. Schritt geht höchstens ε verloren. □

3.11 Satz (Cocke, Younger, Kasami)
Für eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform kann das Wortproblem $w \in L(G)$ für $w \in \Sigma^*$ in Zeit $\mathcal{O}(|P| \cdot |w|^3)$ gelöst werden.

■ **Beweis:** Seien $w = a_1 \ldots a_n$ mit $a_i \in \Sigma$, $w_{i,j} = a_i \ldots a_j$ und $V_{i,j} = \left\{A \in V : A \stackrel{*}{\to} w_{i,j}\right\}$.

Dann ist $w \in L(G) \iff S \in V_{1,n}$. Wir konstruieren die $V_{i,j}$ nach wachsender Differenz der Anfangs- und Endposition $k = j - i$ des abgeleiteten Teilwortes $w_{i,j}$.

$\underline{k = 0}$**:**
$V_{i,i} = \left\{A \in V : A \stackrel{*}{\to} w_{i,i}\right\} = \{A \in V : A \to a_i \text{ in } P\}$, da G in Chomsky-NF. Die $V_{i,i}$ können daher in der Zeit $\mathcal{O}(|P| + |w|)$ durch Prüfen aller Regeln bestimmt werden.

$k > 0$**:**

Wir suchen alle A mit $A \overset{*}{\to} w_{i,j}$. Da $k = j - i > 0$ und G in Chomsky-NF, muss ein erster Ableitungsschritt von der Form $A \to BC$ sein. Wegen der ε-Freiheit von G gilt $A \overset{*}{\to} w_{i,j}$ daher genau dann, wenn $B \overset{*}{\to} w_{i,l}$ und $C \overset{*}{\to} w_{l+1,j}$, d.h. $B \in V_{i,l}$, $C \in V_{l+1,j}$, für ein $l \in \{i, \ldots, j-1\}$.

Werden alle $V_{i,j}$ als Arrays der Länge $|V|$ gespeichert (sodass $A \in V_{i,j}$ in konstanter Zeit überprüft werden kann), so ist der Aufwand in $\mathcal{O}(|P| \cdot |w|^3)$, da $\binom{|w|}{2}$ viele Mengen in jeweils $\mathcal{O}(|P| \cdot |w|)$ konstruiert werden.

□

Aus dem Beweis erhalten wir den folgenden Algorithmus zur Lösung des Wortproblems für kontextfreie Sprachen.

Algorithmus 2: CYK-Algorithmus

Eingabe : kontextfreie Grammatik G in Chomsky-Normalform
Wort $w = a_1 \cdots a_n$

```
for i = 1, ..., n do V_{i,i} ← {A ∈ V : A → a_i in P}
for k = 1, ..., n − 1 do
    for i = 1, ..., n − k do
        j ← i + k
        V_{i,j} ← ∅
        for l = i, ..., j − 1 do
            foreach A → BC in P do
                if B ∈ V_{i,l} und C ∈ V_{l+1,j} then
                    V_{i,j} ← V_{i,j} ∪ {A}
if S ∈ V_{1,n} then gib aus: „w ∈ L(G)“ else gib aus: „w ∉ L(G)“
```

3.12 Beispiel

in Chomsky-Normalform

$$G: \quad \begin{aligned} &S \to SA \,|\, \mathtt{a} \\ &A \to BS \\ &B \to BB \,|\, BS \,|\, \mathtt{b} \,|\, \mathtt{c} \end{aligned}$$

Frage: $\mathtt{abacba} \in L(G)$*?*

Von der Diagonalen her konstruieren

i \ j	1	2	3	4	5	6
1	S	$\emptyset$	S	$\emptyset$	$\emptyset$	S
2		B	A, B	B	B	A, B
3			S	$\emptyset$	$\emptyset$	S
4				B	B	A, B
5					B	A, B
6						S

$s \in V_{1,n} \implies w \in L(G)$.

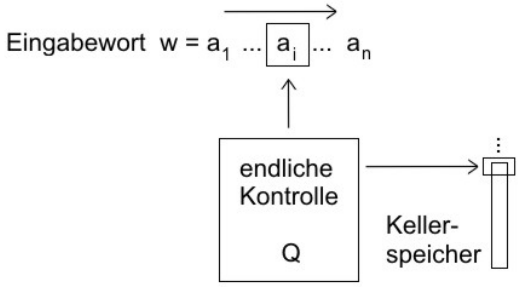
3.2 Kellerautomaten

Für praktische Zwecke ist der CYK-Algorithmus zu aufwändig. Wir betrachten daher auch für kontextfreie Sprachen ein passendes Maschinenmodell.

3.13 Definition
Ein Kellerautomat (KA) $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ *besteht aus*

- *einer endlichen Zustandsmenge* Q
- *einem (Eingabe-)Alphabet* Σ
- *einem (Keller-)Alphabet* Γ
- *einer Übergangsrelation* $\delta \subseteq \big(Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma\big) \times (\Gamma^* \times Q)$
- *einem Startzustand* $s \in Q$
- *einem Kellerstartsymbol* $S \in \Gamma$
- *einer Endzustandsmenge* $F \subseteq Q$

Wie endliche Automaten liest ein Kellerautomat ein Eingabewort zeichenweise und geht dabei von einem Zustand in einen anderen über. Der zusätzliche Keller erlaubt die Ablage von Zwischeninformationen:



Die Schreibweise $(q, a, Z, \gamma, q') \in \delta$ besagt, dass im Zustand q mit nächstem Eingabesymbol $a \neq \varepsilon$ und oberstem Kellersymbol Z der Automat in den Zustand q' übergehen darf und dabei das Z durch γ ersetzt. Analog geschieht dies für $a = \varepsilon$, wobei dann kein Zeichen der Eingabe verbraucht wird.

Für $|\gamma| > 1$ spricht man von einem push-Schritt, für $\gamma = \varepsilon$ von einem pop-Schritt.

Übergänge beschreiben wir durch Konfigurationen $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$ mit $(q, av, Z\gamma) \to (q', v, \gamma'\gamma)$ falls $(q, a, Z, \gamma', q') \in \delta$ und verwenden wieder $\overset{*}{\to}$ für eine Folge von beliebig vielen Übergängen (entsprechend auch $\underset{A}{\to}$, $\overset{*}{\underset{A}{\to}}$).

3.14 Definition
Ein Kellerautomat $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ akzeptiert ein Wort $w \in \Sigma^*$, falls $(s, w, S) \overset{*}{\to} (q, \varepsilon, \gamma)$ mit $q \in F, \gamma \in \Gamma^*$. Die Menge

$$L(\mathcal{A}) := \left\{ w \in \Sigma^* \,:\, (s, w, S) \overset{*}{\to} (q, \varepsilon, \gamma),\ q \in F, \gamma \in \Gamma^* \right\}$$

heißt die von $\mathcal{A}$ akzeptierte Sprache.

3.15 Beispiel
Wir konstruieren einen Kellerautomaten, der die Sprache $L = \left\{ w \in \Sigma^* \,:\, w = vv^R,\ v \in \Sigma^* \right\}$ der Palindrome gerade Länge über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert.

Dazu legen wir auf dem Keller die erste Worthälfte ab und bauen ihn beim Lesen der zweiten Worthälfte wieder ab.

Der Kellerautomat besteht aus $Q = \{s, q_1, q_2, q_f\}$, $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{S\}$, $F = \{q_f\}$ *und*

$$
\begin{array}{rllrl}
\delta: & s & a & S & aS & q_1 \\
 & s & b & S & bS & q_1 \\
 & s & \varepsilon & S & S & q_f \\
 & & & & & \\
 & q_1 & a & b & ab & q_1 \\
 & q_1 & b & a & ba & q_1 \\
 & q_1 & a & a & aa & q_1 \\
\textbf{!}\rightarrow & q_1 & a & a & \varepsilon & q_2 \\
 & q_1 & b & b & bb & q_1 \\
\textbf{!}\rightarrow & q_1 & b & b & \varepsilon & q_2 \\
 & & & & & \\
 & q_2 & a & a & \varepsilon & q_2 \\
 & q_2 & b & b & \varepsilon & q_2 \\
 & q_2 & \varepsilon & S & \varepsilon & q_f
\end{array}
$$

Beachte, dass der Übergang vom Auf- zum Abbau des Kellers nichtdeterministisch *erfolgt.*

Ein Kellerautomat kommt auch ohne Endzustände aus:

3.16 Definition
Sei $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, \emptyset)$ *ein Kellerautomat ohne ausgezeichnete Endzustände. Ein Wort* $w \in \Sigma^*$ *wird von* $\mathcal{A}$ *mit leerem Keller akzeptiert, falls* $(s, w, S) \xrightarrow{*} (q, \varepsilon, \varepsilon)$ *für irgendein* $q \in Q$.

Die Menge

$$N(\mathcal{A}) := \left\{ w \in \Sigma^* \,:\, (s, w, S) \xrightarrow{*} (q, \varepsilon, \varepsilon) \right\}$$

heißt die von $\mathcal{A}$ *mit leerem Keller akzeptierte Sprache.*

3.17 Satz (Äquivalenz der Akzeptanzarten)
a) *Zu jedem KA $\mathcal{A}$ existiert ein KA $\mathcal{B}$ mit $L(\mathcal{A}) = N(\mathcal{B})$.*

b) *Zu jedem KA $\mathcal{A}$ existiert ein KA $\mathcal{B}$ mit $N(\mathcal{A}) = L(\mathcal{B})$.*

■ **Beweis:**

a) Zu $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ konstruiere KA $\mathcal{B}$, der wie $\mathcal{A}$ arbeitet und von den Endzuständen aus den Keller leert (falls $\mathcal{A}$ das Wort nicht akzeptiert, darf der Keller nicht leer werden). Ergänze daher $\mathcal{A}$ um

- einen neuen Anfangszustand s'
- einen Zustand $\hat{q}$ zum Leeren des Kellers
- ein neues Kellerstartsymbol S'

sowie Transitionen

s'	ε	S'	SS'	s	
q	ε	Z	ε	$\hat{q}$	für alle $q \in F$, $Z \in \Gamma \cup \{S'\}$
$\hat{q}$	ε	Z	ε	$\hat{q}$	für alle $Z \in \Gamma \cup \{S'\}$.

b) Zu $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, \emptyset)$ konstruiere KA $\mathcal{B}$, der wie $\mathcal{A}$ arbeitet und genau bei geleertem Keller in einen Endzustand wechselt. Ergänze daher $\mathcal{A}$ um

- einen neuen Anfangszustand s'
- einen neuen und dann einzigen Endzustand q_f
- ein neues Kellerstartsymbol S'

sowie Transitionen

s'	ε	S'	SS'	s	
q	ε	S'	ε	q_f	für alle $q \in Q$.

□

3.18 Satz

Eine Sprache ist kontextfrei genau dann, wenn sie von einem Kellerautomaten akzeptiert wird.

■ **Beweis:**

„ $\Longrightarrow$ “**:** Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Wir zeigen, dass es einen Kellerautomaten $\mathcal{A}$ mit $N(\mathcal{A}) = L(G)$ gibt, d.h. $\mathcal{A}$ akzeptiert ein Wort mit leerem Keller genau dann, wenn es in G ableitbar ist.

Der Automat simuliert im Keller eine Linksableitung, indem Nichtterminalsymbole an der Kellerspitze durch die Satzform einer rechten Regelseite ersetzt und Terminalsymbole mit der Eingabe abgeglichen werden. Sei daher $\mathcal{A} = (\{s\}, \Sigma, \Gamma, \delta, s, S, \emptyset)$ mit $\Gamma = \Sigma \dot\cup V$ und

$$\begin{array}{lccccll} \delta: & s & \varepsilon & A & \gamma & s & \text{für alle } A \to \gamma \text{ in } P \\ & s & a & a & \varepsilon & s & \text{für alle } a \in \Sigma\ . \end{array}$$

Wir zeigen, dass für alle $u \in \Sigma^*$ und $\alpha \in \Gamma^*$, die nicht mit einem Terminal beginnen (d.h. $\alpha \in \{\varepsilon\} \cup V \cdot \Gamma^*$), gilt

$$\boxed{S \overset{*}{\underset{G}{\to}} u\alpha \text{ mit Linksableitungen } \iff (s, u, S) \overset{*}{\underset{\mathcal{A}}{\to}} (s, \varepsilon, \alpha)\ .}$$

Daraus folgt dann für $\alpha = \varepsilon$, dass $\mathcal{A}$ genau die von der Grammatik erzeugte Sprache erkennt.

„ $\Longleftarrow$ “**:** Induktion über die Länge k der Berechnung in $\mathcal{A}$.

$\underline{k = 0}$**:** Dann sind $u = \varepsilon$, $\alpha = S$ und $S \overset{*}{\underset{A}{\to}} S$ mit Linksableitungen.

$\underline{k > 0}$**:** Es gelte $(s, u, S) \overset{k}{\underset{\mathcal{A}}{\to}} (s, \varepsilon, \alpha)$ und wir betrachten den letzten Berechnungsschritt:

1. Fall: (Transition der Form $(s, \varepsilon, A, \gamma, s) \in \delta$)
Dann gilt $(s, u, S) \overset{k-1}{\underset{\mathcal{A}}{\to}} (s, \varepsilon, A\alpha') \underset{\mathcal{A}}{\to} (s, \varepsilon, \gamma\alpha')$ mit $\alpha = \gamma\alpha'$, und aus der Induktionsvoraussetzung folgt $S \overset{*}{\underset{G}{\to}} uA\alpha' \underset{G}{\to} u\gamma\alpha' = u\alpha$ mit Linksableitungen.

2. Fall: (Transition der Form $(s, a, a, \varepsilon, s) \in \delta$)
Dann gilt $(s, u'a, S) \overset{k-1}{\underset{\mathcal{A}}{\to}} (s, a, a\alpha) \underset{\mathcal{A}}{\to} (s, \varepsilon, \alpha)$ mit $u = u'a$,

und aus der Induktionsvoraussetzung folgt $S \xrightarrow[\mathcal{A}]{*} u'a\alpha = u\alpha$ mit Linksableitungen.

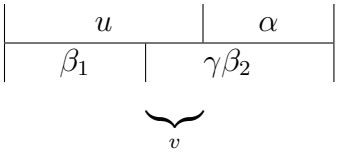
„$\Longrightarrow$“**:** Induktion über die Anzahl k der Linksableitungen in G.

$k = 0$**:** Dann sind $u = \varepsilon$, $\alpha = S$ und $(s, u, S) \xrightarrow[A]{*} (s, \varepsilon, \alpha)$.

$k > 0$**:** Es gelte $S \xrightarrow[G]{k} u\alpha$ mit Linksableitungen und deren letzte sei $A \to \gamma$. Dann gilt $S \xrightarrow[G]{k-1} \beta_1 A \beta_2 \underset{G}{\to} \beta_1\gamma\beta_2 = u\alpha$ mit $\beta_1 \in \Sigma^*$, da Linksableitungen vorausgesetzt wurden. Mit der Induktionsvoraussetzung gilt also

$$(s, \beta_1, S) \xrightarrow[\mathcal{A}]{*} (s, \varepsilon, A\beta_2) \underset{\mathcal{A}}{\to} (s, \varepsilon, \gamma\beta_2) \ . \qquad (\star)$$

Weil α nicht mit einem Terminal beginnt und A das linkeste Nichtterminal in $\beta_1 A \beta_2$ ist, gilt $|\beta_1| \leq |u|$. Es muss daher ein $v \in \Sigma^*$ mit $\beta_1 v = u$ und $v\alpha = \gamma\beta_2$ geben.



Durch Anwendung von Transitionen $(s, a, a, \varepsilon, s) \in \delta$, $a \in \Sigma$, erhalten wir

$$(s, u, S) = (s, \beta_1 v, S) \underbrace{\xrightarrow[\mathcal{A}]{*}}_{(\star)} (s, v, \gamma\beta_2) \xrightarrow[\mathcal{A}]{*} (s, \varepsilon, \alpha) \ .$$

„$\Longleftarrow$“**:** Zu einem Kellerautomaten $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, \emptyset)$ konstruieren wir eine kontextfreie Grammatik $G = (V, \Sigma, P, S')$ mit $L(G) = N(\mathcal{A})$. Die Variablen

$$V = S' \cup \{[pZq] \ : \ p, q \in Q, Z \in \Gamma\}$$

der Grammatik stehen für mögliche Folgen von Berechnungsschritten des Automaten. Die Idee dahinter ist, dass $[pZq] \xrightarrow[G]{*} u$ genau dann gelten soll, wenn $\mathcal{A}$ beim Übergang von p nach q die Eingabe u verbraucht

und das Symbol Z von der Kellerspitze entfernt (ohne die Zeichen darunter anzufassen). Wir nehmen daher folgende Produktionen auf:

$$\begin{array}{rrcll} P: & S' & \to & [sSq] & \text{für alle } q \in Q \\ & [pZq] & \to & a & \text{für alle } (p,a,Z,\varepsilon,q) \in \delta \\ & [pZq] & \to & a[p_0Z_1p_1][p_1Z_2p_2]\cdots[p_{n-1}Z_nq] & \\ & & & & \text{für alle } (p,a,Z,Z_1\cdots Z_n,p_0) \in \delta \\ & & & & \text{mit } n>0 \text{ und } p_1,\ldots,p_{n-1} \in Q \end{array}$$

Beachte, dass $a \in \Sigma \cup \{\varepsilon\}$.

Wir zeigen, dass für alle $p,q \in Q$, $u \in \Sigma^*$ und $Z \in \Gamma$ gilt

$$\boxed{(p,u,Z) \xrightarrow[\mathcal{A}]{*} (q,\varepsilon,\varepsilon) \iff [pZq] \xrightarrow[G]{*} u}$$

woraus für $p=s$, $Z=S$ und $u=w$ die Behauptung folgt, weil dann $(s,w,S) \xrightarrow[\mathcal{A}]{*} (q,\varepsilon,\varepsilon) \iff S' \underset{G}{\to} [sSq] \xrightarrow[G]{*} w$.

„ $\Longrightarrow$ “**:** Induktion über die Länge $k \geq 1$ der Berechnung in $\mathcal{A}$.

$\underline{k=1}$**:** Dann ist $u = a \in \Sigma \cup \{\varepsilon\}$ und der Berechnungsschritt war von der Form $(p,a,Z,\varepsilon,q) \in \delta$. Nach Konstruktion ist $[pZq] \to a$ in P.

$\underline{k>1}$**:** Sei $u = au'$ mit $a \in \Sigma \cup \{\varepsilon\}$ und der erste Berechnungsschritt sei $(p,a,Z,Z_1\cdots Z_n,p_0) \in \delta$, dann gilt

$$(p,au',Z) \underset{\mathcal{A}}{\to} (p_0,u',Z_1\cdots Z_n) \overset{k-1}{\underset{\mathcal{A}}{\to}} (q,\varepsilon,\varepsilon)\ .$$

Wähle nun $u_1,\ldots,u_n \in \Sigma^*$ mit $u' = u_1\cdots u_n$ gemäß dem Abbau der $Z_1,\ldots,Z_n$ im Keller, d.h. $u_1\cdots u_i$ führt über vom Zustand p_0 mit Kellerspitze $Z_1\cdots Z_n$ zum Zustand p_i mit Kellerspitze $Z_{i+1}\cdots Z_n$. Die Zahl der für die Übergänge

$$(p_{i-1},u_i,Z_i) \xrightarrow[\mathcal{A}]{*} (p_i,\varepsilon,\varepsilon) \qquad \text{für } i=1,\ldots,n \text{ und } p_n = q$$

benötigten Berechnungsschritte ist jeweils kleiner als k, sodass sich mit der Induktionsvoraussetzung

$$[p_{i-1}Z_ip_i] \xrightarrow[G]{*} u_i$$

ergibt. Nach Konstruktion gibt es zum ersten Ableitungsschritt die Produktion $[pZq] \to a[p_0Z_1p_1]\dots[p_{n-1}Z_nq]$, sodass wir insgesamt

$$[pZq] \underset{G}{\to} a[p_0Z_1p_1]\cdots[p_{n-1}Z_nq] \overset{*}{\underset{G}{\to}} au_1\cdots u_n = au' = u$$

erhalten.

„$\Longleftarrow$“: Induktion über die Anzahl $k \geq 1$ der Ableitungen in G.

$k = 1$: Die Ableitung muss von der Form $[pZq] \to a$ sein. Es existiert daher eine Transition $(p, a, Z, \varepsilon, q) \in \delta$ und somit $(p, a, Z) \underset{\mathcal{A}}{\to} (q, \varepsilon, \varepsilon)$.

$k > 1$: Betrachte die Ableitung

$$[pZq] \underset{G}{\to} a[p_0Z_1p_1]\dots[p_{n-1}Z_np_n] \overset{k-1}{\underset{G}{\to}} au_1\cdots u_n \ .$$

Nach Konstruktion muss $(p, a, Z, Z_1 \dots Z_n, p_0) \in \delta$ sein und aus der Induktionsvoraussetzung folgt $(p_{i-1}, u_i, Z_i) \overset{*}{\underset{\mathcal{A}}{\to}} (p_i, \varepsilon, \varepsilon)$, sodass insgesamt

$$(p, u, Z) = (p, au_1 \dots u_n, Z) \overset{*}{\underset{\mathcal{A}}{\to}} (p_n, \varepsilon, \varepsilon) = (q, \varepsilon, \varepsilon) \ .$$

□

3.19 Beispiel

Wir nutzen die Konstruktion aus der Hinrichtung des obigen Beweises, um einen Kellerautomaten zu konstruieren, der die Sprache der Palindrome gerader Länge mit leerem Keller akzeptiert.

$$\begin{array}{lll} G: & S \to \varepsilon & \\ & S \to aSa & \text{für alle } a \in \Sigma \end{array}$$

führt auf

$$\begin{array}{lllllll} \mathcal{A}: & s & \varepsilon & S & \varepsilon & s & \\ & s & \varepsilon & S & aSa & s & \text{für alle } a \in \Sigma \\ & s & a & a & \varepsilon & s & \text{für alle } a \in \Sigma. \end{array}$$

3.20 Satz (Pumping-Lemma für kontextfreie Sprachen)
Sei L eine kontextfreie Sprache. Dann existiert eine Zahl $n \in \mathbb{N}$ so, dass sich jedes Wort $w \in L$ mit $|w| \geq n$ darstellen läßt als

$$w = tuvxz \qquad \text{mit } |ux| \geq 1,\ |uvx| \leq n$$

und

$$tu^ivx^iz \in L \qquad \text{für alle } i \in \mathbb{N}_0.$$

Wir beweisen nur die verschärfte Form.

3.21 Satz (Ogdens Lemma: verschärftes PL für kfS)
Sei L eine kontextfreie Sprache. Dann existiert eine Zahl $n \in \mathbb{N}$ so, dass für jedes Wort $w \in L$ mit $|w| \geq n$ gilt: Werden in w mindestens n Buchstaben markiert, dann läßt sich w darstellen als

$$w = tuvxz \qquad \begin{array}{l}\text{mit mindestens einer Markierung in } ux \\ \text{und höchstens } n \text{ Markierung in } uvx\end{array}$$

und

$$tu^ivx^iz \in L \qquad \text{für alle } i \in \mathbb{N}_0.$$

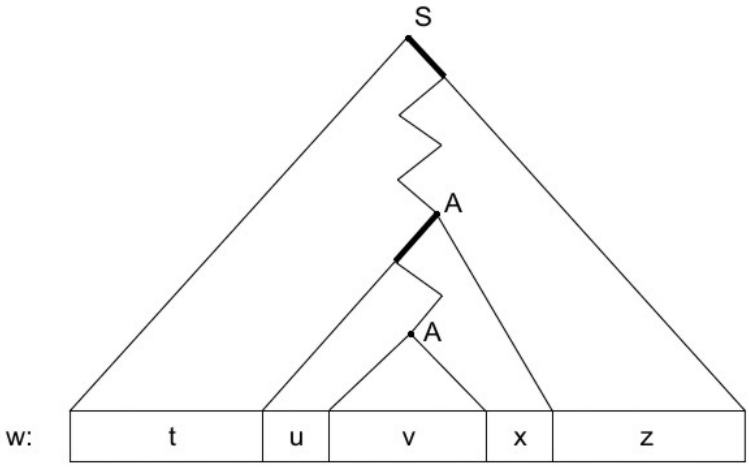
■ **Beweis:** Zu L gibt es eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform mit $L(G) = L \setminus \{\varepsilon\}$. Wir setzen $n := 2^{|V|+1}$ und betrachten ein $w \in L$ mit $|w| \geq n$ und mindestens n markierten Buchstaben, sowie einen zugehörigen Syntaxbaum.

Der Syntaxbaum hat $|w|$ Blätter, alle inneren Knoten haben höchstens zwei Kinder und die inneren Knoten mit nur einem Kind sind genau die Eltern der Blätter (da G in Chomsky-NF ist).

Wir wählen im Baum einen Weg von der Wurzel zu einem Blatt, der immer in Richtung der größeren Anzahl der n markierten Blätter weiterläuft (bei Gleichheit beliebig). Knoten mit markierten Blättern im linken und im rechten Teilbaum heißen *Verzweigungsknoten*.

Wegen $n = 2^{|V|+1}$ liegen mindestens $|V| + 1$ Verzweigungsknoten auf dem gewählten Weg. Betrachte die letzten $|V| + 1$ davon und beachte, dass mindestens zwei mit der gleichen Variable A markiert sein müssen.

Wir erhalten die folgende Darstellung von w:



Da die mit A beschrifteten Knoten Verzweigunsknoten sind, enthält ux mindestens einen markierten Buchstaben, und da der gewählte Weg ab dem oberen A-Knoten nur noch höchstens $|V| + 1$ Verzweigunsknoten enthält, folgt aus der Auswahlregel für den jeweils nächsten Knoten, dass uvx höchstens $2^{|V|+1} = n$ markierte Buchstaben enthält.

In der Grammatik sind offensichtlich folgende Ableitungen möglich:

$$S \overset{*}{\rightarrow} tAz \tag{1}$$

$$A \overset{*}{\rightarrow} uAx \tag{2}$$

$$A \overset{*}{\rightarrow} v \tag{3}$$

Daraus erhalten wir für jedes $i \in \mathbb{N}_0$ eine Ableitung

$$S \underset{(1)}{\overset{*}{\rightarrow}} tAz \underbrace{\underset{(2)}{\overset{*}{\rightarrow}} tuAxz \underset{(2)}{\overset{*}{\rightarrow}} \ldots \underset{(2)}{\overset{*}{\rightarrow}} tu^iAx^iz}_{i\text{-mal}} \underset{(3)}{\overset{*}{\rightarrow}} tu^ivx^iz$$

sodass $tu^ivx^iz \in L(G) \subseteq L$ für alle $i \in \mathbb{N}_0$. □

3.22 Beispiel

Wie beim Pumping-Lemma für reguläre Sprachen lassen sich diese beiden Sätze zum Nachweis der Nicht-Kontextfreiheit einer Sprache einsetzen:

Sei $L = \{a^i b^j a^j b^i : i > j > 0\}$ *und zum* n *aus dem Pumping-Lemma betrachte* $w = a^{n+1}b^n a^n b^{n+1}$. *Sei ferner* $w = tuvxz$ *eine zugehörige Darstellung aus dem Pumping-Lemma mit* $ux \neq \varepsilon$ *und* $|uvx| \leq n$.

1. Fall: *(uvx ist im Mittelteil* $b^n a^n$ *enthalten)*
Dann kann durch Aufpumpen die Bedingung $i > j$ *verletzt werden.*

2. Fall: *(uvx ist in* $a^{n+1}b^n$ *oder in* $a^n b^{n+1}$ *enthalten)*
Dann kann durch Aufpumpen die Bedingung der gleichen Anzahlen verletzt werden.

3.23 Definition

Ein Kellerautomat $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ *heißt deterministisch (DKA), falls*

i) Für alle $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$ *existiert höchstens ein Übergang der Form* $(q, a, Z, \cdot, \cdot) \in \delta$.

ii) Für alle $q \in Q$, $Z \in \Gamma$ *gilt: Ist ein Übergang der Form* $(q, \varepsilon, Z, \cdot, \cdot) \in \delta$, *dann gibt es keinen Übergang der Form* $(q, a, Z, \cdot, \cdot)$ *mit* $a \in \Sigma$.

Wir werden zeigen, dass deterministische Kellerautomaten nicht alle kontextfreien Sprachen erkennen.

3.24 Lemma

Gibt es zu einer Sprache $L \subseteq \Sigma^*$ *einen deterministischen Kellerautomaten* $\mathcal{A}$ *mit* $L(\mathcal{A}) = L$, *so gibt es zu*

$$MIN(L) := \{w \in L : \text{es gibt kein echtes Präfix von } w \text{ in } L\}$$

einen deterministischen Kellerautomaten $\mathcal{B}$ *mit* $L(\mathcal{B}) = MIN(L)$.

■ **Beweis:** Der DKA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ erkenne L. Konstruiere den Automaten $\mathcal{B}$ durch Hinzufügen eines neuen Zustands q' und der Übergänge

$$q' \quad a \quad Z \quad Z \quad q' \qquad \text{für alle } a \in \Sigma,\ Z \in \Gamma$$

sowie Umwandeln aller Transition mit $q \in F$, $a \in \Sigma$, $Z \in \Gamma$ in

$$q \quad a \quad Z \quad Z \quad q' \ .$$

„Einmal in F, nie wieder in F“

□

3.25 Lemma
Sei $L = L(\mathcal{A}) \subseteq \Sigma^*$ für einen (deterministischen) Kellerautomaten $\mathcal{A}$ und $R \subseteq \Sigma^*$ regulär. Dann ist $L \cap R = L(\mathcal{B})$ für einen (deterministischen) Kellerautomaten $\mathcal{B}$.

■ **Beweis:** Sei $L = L(\mathcal{A})$ für den (D)KA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, S, F)$ und $R = L(\mathcal{A}')$ für den DEA $\mathcal{A}' = (Q', \Sigma, \delta', s', F')$. Wir schalten $\mathcal{A}$ und $\mathcal{A}'$ wie folgt zu einem (D)KA $\mathcal{B} = (Q \times Q', \Sigma, \Gamma, \hat{\delta}, (s, s'), S, F \times F')$ zusammen:

$$\big((p,p'),a,Z,\gamma,(q,q')\big) \in \hat{\delta} \quad :\Longleftrightarrow \quad \begin{array}{l}(p,a,Z,\gamma,q) \in \delta \text{ und } \delta'(p',a) = q' \\ \text{für } a \in \Sigma \\ \text{bzw. } a = \varepsilon \text{ und } p' = q' \ .\end{array}$$

Mit Induktion über die Länge k der Berechnung folgt

$$((s,s'),w,S) \xrightarrow[\mathcal{B}]{k} ((q,q'),\varepsilon,\gamma) \iff (s,w,S) \xrightarrow[\mathcal{A}]{k} (q,\varepsilon,\gamma) \text{ und } \delta'(s',w) = q' \ ,$$

und damit $L(\mathcal{B}) = L(\mathcal{A}) \cap L(\mathcal{A}') = L \cap R$. □

3.26 Satz
Es gibt keinen deterministischen Kellerautomaten, der die kontextfreie Sprache

$$L = \big\{w \in \{a,b\}^* \,:\, w = vv^R, v \in \{a,b\}^*\big\}$$

der Palindrome gerader Länge über $\{a,b\}$ akzeptiert.

■ **Beweis:** Angenommen, es existiert ein DKA $\mathcal{A}$ mit $L(\mathcal{A}) = L$. Nach Lemma 3.24 und Lemma 3.25 existiert dann zu

$$L' := \mathrm{MIN}(L) \cap (\mathbf{ab})^+(\mathbf{ba})^+(\mathbf{ab})^+(\mathbf{ba})^+$$

ein DKA $\mathcal{A}'$ mit $L(\mathcal{A}') = L'$. Da alle Wörter in L' Palindrome gerader Länge ohne echtes Präfix dieser Art sind, gilt:

$$L' = \big\{(ab)^i(ba)^j(ab)^j(ba)^i \,:\, i > j > 0\big\}$$

Mit dem Pumping-Lemma lässt sich dann aber wie in Beispiel 3.22 zeigen, dass L' nicht kontextfrei ist. Dies ist ein Widerspruch! □

Exkurs: Compilerbau

Compiler übersetzen Programme einer höheren Programmiersprache in Maschinencode, also Wörter über dem ASCII- oder Unicode-Alphabet in Wörter über einem Prozessor-spezifischen Alphabet aus Instruktionen und Daten. Aspekte von (höheren) Programmiersprachen:

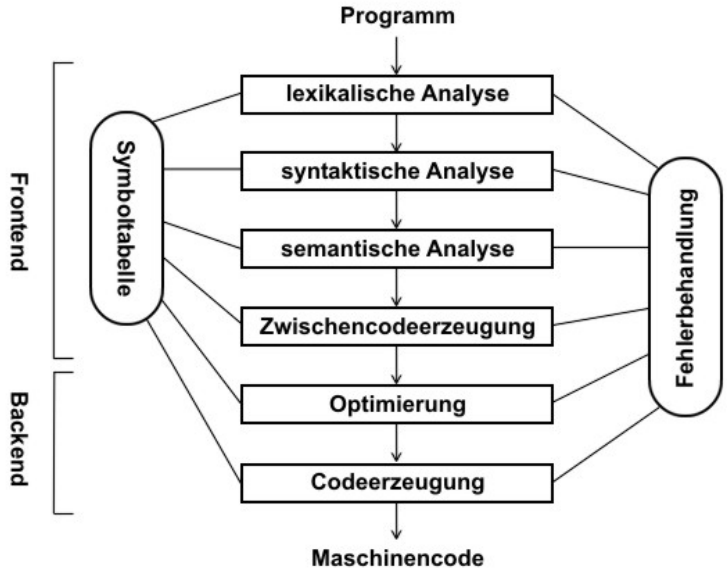
Syntax: formaler Aufbau eines Programms aus strukturellen Einheiten

Semantik: Bedeutung eines Programms (Zustandstransformation)

Pragmatik: benutzerfreundliche Formulierung, Maschinenabhängigkeit

Typischer Aufbau eines Compilers:

logische Phasen laufen üblicherweise verzahnt ab



1. Phase: Lexikalische Analyse

Zusammenfassung von Teilzeichenketten zu semantisch relevanten Symbolen. Diese sind pragmatisch in natürlicher Sprache formuliert ($\rightarrow$ Microsyntax), was aber semantisch irrelevant ist.

- Grundsymbole sind

  Bezeichner: Folge von Buchstaben und Ziffern, z.B. für Konstanten, Variablen, Typen, Methoden, ...

  Zahlwörter: Folge von Ziffern, Sonderzeichen und Buchstaben (z.B. für Hexadezimalzahlen)

  Schlüsselwörter: Bezeichner mit vorgegebener Bedeutung

  einfache Symbole: einzelne Sonderzeichen, z.B. `+`, `-`, `*`, `<`, ...

  zusammengesetzte Symbole: zwei oder mehr Sonderzeichen, z.B. `:=`, `<=`, `&&`, ...

- **Leerzeichen:** z.B. ␣, `\t`, ␣␣, ... (unterschiedliche Bedeutung)

- **Spezielle Symbole:** Kommentare (`/* ... */`, `//`, ...), Pragma (wie z.B. Compileroptionen, Präprozessor)

Symbolklassen werden durch reguläre Ausdrücke beschrieben, z.B. für Bezeichner (*identifier*):

$$\begin{aligned} \textbf{id} &= \textbf{char}(\textbf{char} + \textbf{digit})^* \\ \textbf{char} &= A + B + \ldots + Z + a + \ldots \\ \textbf{digit} &= 0 + 1 + \ldots + 9 \end{aligned}$$

Ausgabe des endlichen Automaten zur lexikalischen Analyse ist für jedes erkannte Symbol eine Darstellung der Form

(Token, Attribut)

mit

Token: Bezeichnung der Symbolklasse (intern: Zahl)

Attribut: je nach Symbolklasse

- leer
- Wert (einer Konstanten)
- Zeiger (in Symboltabelle)

Beispiele für solche Darstellungen sind

- (**if**, _), (**then**, _), (**for**, _)
- (**binop**, +), (**relop**, <=)
- (**id**, $\bullet$)$\mapsto$ Eintrag des Bezeichners in Symboltabelle
- (**num**, $\bullet$)$\mapsto$ Eintrag des Zahlwortes in Symboltabelle

Programme zur lexikalischen Analyse („Scanner“) können aus den regulären Ausdrücken für die Symbolklassen automatisch erzeugt werden.

Tools: `lex`, `flex`

Syntaktische Analyse

Aufteilung der Tokenfolge in syntaktische Einheiten (z.B. Deklaration, Anweisung, Ausdruck, ...), die geschachtelt auftreten ($\rightarrow$ Baumstruktur).

Syntaktische Strukturen werden durch kontextfreie Grammatiken beschrieben, z.B. für Ausdrücke

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \texttt{(}E\texttt{)} \mid \textbf{id} \mid \textbf{num} \end{aligned}$$

Aus Effizienzgründen werden nur Grammatiken verwendet, zu denen sich deterministische Kellerautomaten konstruieren lassen (Charakterisierung bekannt: so genannte LR(k)-Grammatiken).

Programme zur syntaktischen Analyse („Parser“) können aus den kontextfreien Grammatiken für die Syntax automatisch erzeugt werden.

Tools: `yacc`, `bison`

Semantische Analyse

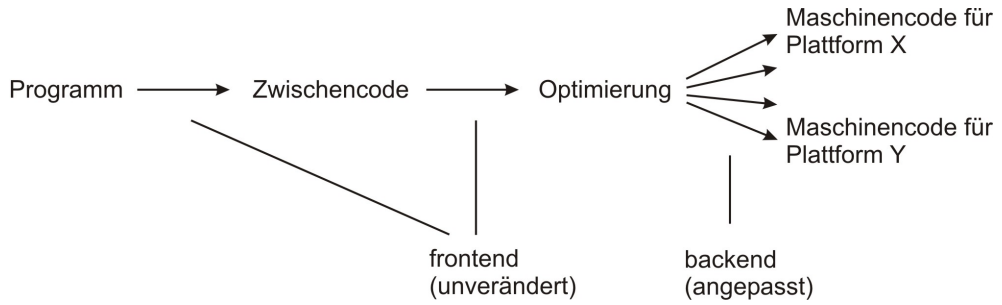
Kontextfreie Grammatiken beschreiben die syntaktische Struktur, können aber keine kontextabhängigen Eigenschaften (z.B. vorherige Deklaration von Bezeichnern) erfassen.

Bestimmung der statischen Semantik (Gültigkeitsbereiche, Typprüfung) durch attribuierte Grammatiken (kontextfreie Grammatiken mit semantischen Regeln für Zusatzinformation im Ableitungsbaum; werden von den Parsergeneratoren unterstützt). Ausgabe: dekorierter Syntaxbaum.

Zwischencodeerzeugung

Statt des Maschinencodes für die Zielplattform wird zunächst Code für eine abstrakte Maschine erzeugt. Vorteile:

- Transparenz des Compilers
- Portabilität des Compilers
- leichtere Codeoptimierung



Backend

Phasen 5 (Optimierung) und 6 (Codegenerierung) sind weniger allgemein darstellbar. In Java fällt Phase 4 weg, da am Ende Maschinencode für eine abstrakte Maschine generiert wird (*Bytecode*) die von einem Interpreter simuliert wird (*Java Virtual Machine*).

Kapitel 4

Rekursiv aufzählbare Sprachen

4.1 Grammatiken und die Chomsky-Hierarchie

Durch Zulassung komplexer Ableitungsregeln können mit Grammatiken größere Klassen als die kontextfreien Sprachen beschrieben werden.

4.1 Definition
Eine Grammatik $G = (V, \Sigma, P, S)$ besteht aus

- *einer endlichen Menge von Variablen V*
- *einem Alphabet Σ*
- *einem Startsymbol $S \in V$*
- *einer endlichen Menge von Ableitungsregeln $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$*

4.2 Bemerkung
Kontextfreie Grammatiken sind ein Spezialfall, der durch Einschränkungen an die Satzform der linken Regelseite definiert ist. In beliebigen Grammatiken können Regeln z.B. Terminale auf der linken Seite haben oder eine Satzform durch eine kürzere ersetzen.

4.3 Definition (Chomsky-Hierarchie)
Eine Grammatik $G = (V, \Sigma, P, S)$ heißt

- *vom Typ 3 (rechtslinear, regulär), falls $P \subseteq V \times (\Sigma \cdot V \cup \{\varepsilon\})$, d.h. alle Regeln sind von der Form $A \to \varepsilon$ oder $A \to aB$.*
- *vom Typ 2 (kontextfrei), falls $P \subseteq V \times (\Sigma \cup V)^*$, d.h. alle Regeln sind von der Form $A \to \gamma$.*
- *vom Typ 1 (kontextsensitiv), falls $P \subseteq (V^+ \times (\Sigma \cup V \setminus \{S\})^+) \cup (S \to \varepsilon)$ und für alle Regeln $(\alpha \to \beta) \neq (S \to \varepsilon)$ gilt $|\alpha| \leq |\beta|$.*

Grammatiken ohne Einschränkungen heißen vom Typ 0 oder rekursiv aufzählbar.

4.4 Bemerkung

1. *Die Grammatiktypen bilden eine Hierarchie, da jede Typ-3 auch eine eine Typ-2, jede Typ-2 auch eine Typ-1, und jede Typ-1 auch eine Typ-0 Grammatik ist.*
2. *Die von einer Typ-i Grammatik erzeugten Sprachen heißen entsprechend Typ-i Sprachen.*
3. *Bei kontextsensitiven (Typ-1) Sprachen kann eine Regel die Ableitung $ABC \to A\gamma C$ ermöglichen, ohne dass $DBC \to D\gamma C$ möglich wird.*
4. *Grammatiken vom Typ 3 erzeugen genau die regulären Sprachen.* Übung

4.5 Beispiel
Die folgende Typ-0 Grammatik erzeugt die (nicht kontextfreie) Sprache $L = \{a^i b^j a^j b^i \in \{a, b\}^* : i > j > 0\}$:

$$\begin{array}{llll} S \to XAbAMBaBX & bA \to Ab & XA \to aX & X \to \varepsilon \\ M \to AMB \mid bAMBa \mid \varepsilon & Ba \to aB & BX \to Xb & \end{array}$$

Gibt es auch eine kontextsensitive Grammatik, die L erzeugt? Übung

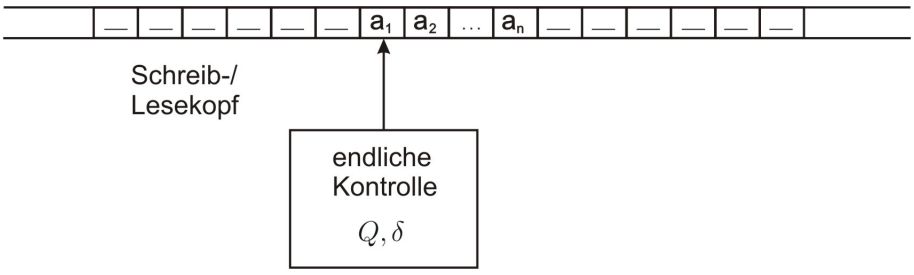
4.2 Turingmaschinen

Auch zu von beliebigen Grammatiken erzeugten Sprachen gibt es ein passendes Maschinenmodell, von dem wir sehen werden, dass es universell einsetzbar ist.

4.6 Definition
Eine (deterministische) Turingmaschine (D)TM $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ *besteht aus*

- *einer endlichen Zustandsmenge* Q
- *einem Eingabealphabet* Σ
- *einem Bandalphabet* Γ *mit* $\Sigma \subset \Gamma$
- *einem Leerzeichen (Blanksymbol)* $\_ \in \Gamma \setminus \Sigma$
- *einer Übergangsfunktion* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$
- *einem Startzustand* $s \in Q$
- *einer Menge von Endzuständen* $F \subseteq Q$, *wobei für alle* $q \in F$, $a \in \Gamma$ *gilt:* $\delta(q, a) = (q, a, N)$.

Interpretation: Anstelle eines Kellers verfügt eine Turingmaschine über ein zweiseitig unendliches Band, das Symbole des Bandalphabets enthält.



Startkonfiguration: Im Startzustand s zeigt der Schreib-/Lesekopf der TM auf den Anfang des Eingabewortes, das auf dem ansonsten nur Leerzeichen enthaltenden Band steht.

Ein Übergang $\delta(q, a) = (q', a', B)$ bedeutet, dass im Zustand q bei aktuellem Bandinhalt $a \in \Gamma$ die Maschine in den Zustand q' übergeht, dabei a durch $a' \in \Gamma$ ersetzt und den Schreib-/Lesekopf entsprechend $B \in \{L, R, N\}$ bewegt:

L: eine Position nach links,

R: eine Position nach rechts,

N: an der aktuellen Stelle stehen bleiben.

In Zeichen: $q \overset{a|a', B}{\longrightarrow} q'$.

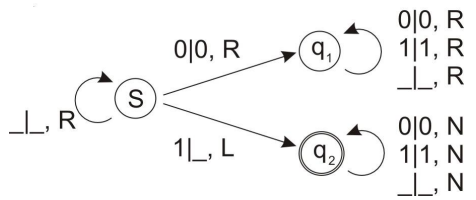
4.7 Bemerkung
Eine TM stoppt im Zustand q, falls beim Lesen des Symbols a gilt: $\delta(q, a) = (q, a, N)$. Insbesondere stoppt sie also bei Erreichen eines Endzustands, und wir vereinbaren, dass sie bei unvollständiger Angabe von δ für nicht definierte Argumente q, a stoppt.

4.8 Definition
Eine Turingmaschine $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ akzeptiert ein Wort $w \in \Sigma^*$, falls sie bei Eingabe von w in einem Zustand aus F stoppt. Die Menge

$$L(\mathcal{A}) = \{w \in \Sigma^* : \mathcal{A} \text{ stoppt bei Eingabe von } w \text{ in einem } q \in F\}$$

heißt die von $\mathcal{A}$ akzeptierte Sprache.

4.9 Beispiel

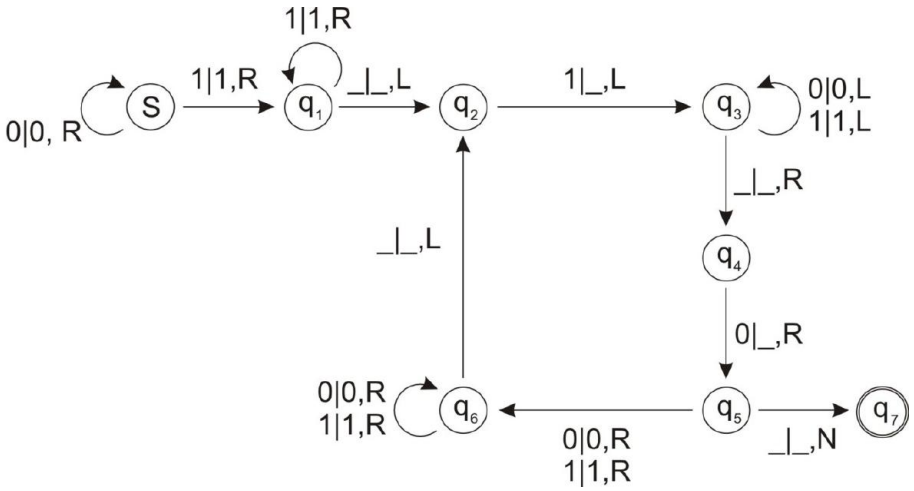


$$L(\mathcal{A}) = \{w \in \{0,1\}^* : w = 1v,\ v \in \{0,1\}^*\}$$

- *Für eine Eingabe aus $L(\mathcal{A})$ löscht $\mathcal{A}$ die führende 1 und bleibt mit dem Schreib-/Lesekopf links neben dem Beginn der Eingabe stehen.*
- *Für eine Eingabe aus $\{0,1\}^* \setminus L(\mathcal{A})$ stoppt $\mathcal{A}$ überhaupt nicht.*

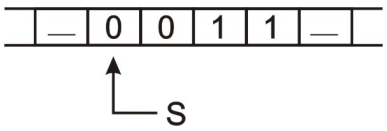
Während einer Berechnung ist eine TM $\mathcal{A}$ in der Konfiguration $w(q)av$ mit $w, v \in \Gamma^*$, $a \in \Gamma$, $q \in Q$, falls sie im Zustand q das Symbol a liest und links und rechts von a die Inschriften w und v stehen.

4.10 Beispiel



$L(\mathcal{A}) = \{0^i 1^i : i \in \mathbb{N}\}$ *(kontextfrei, aber nicht regulär).*

Berechnung für die Eingabe $w = 0011$*:*



$\_(s)0011 \to 0(s)011 \to 00(s)11 \to 001(q_1)1 \to 0011(q_1)\_ \to 001(q_2)1 \to 00(q_3)1\_ \to 0(q_3)01 \to \_(q_3)001 \to \_(q_3)\_001 \to \_(q_4)001 \to \_\_(q_5)01 \to 0(q_6)1 \to 01(q_6)\_ \to 0(q_2)1 \to \_(q_3)0\_ \to \_(q_3)\_0 \to \_(q_4)0 \to \_(q_5)\_ \to \_(q_7)\_$

Die Eingabe wird akzeptiert.

4.11 Satz

Die von einer deterministischen Turingmaschine akzeptierte Sprache ist rekursiv aufzählbar.

■ **Beweis:** Zu DTM $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ konstruieren wir eine Grammatik, die $L(\mathcal{A})$ erzeugt. Zur Vorbereitung kann eine gegebene DTM so modifiziert werden, dass sie als erstes ein neues Zeichen $\# \notin \Gamma$ hinter die Eingabe schreibt, danach (und nur dann) in einem neuen Zustand s' ist, sich niemals nach rechts über # hinausbewegt und nur einen akzeptierenden Zustand q^+ hat, bei dessen Erreichen das ganze Band nur noch Leerzeichen enthält.

Wir gehen daher davon aus, dass die Anfangskonfiguration $(s')a_1 \dots a_n\#$ lautet. Die Produktionen der Grammatik $G = (\{S\} \cup Q \cup (\Gamma \setminus \Sigma) \cup \{\#\}, \Sigma, P, S)$ werden nun so gewählt, dass G aus der akzeptierenden Konfiguration q^+ rückwärts die Eingabe erzeugt:

1. (Schlusskonfiguration und benötigter Platz auf dem Band)
   $S \to q^+, \quad q^+ \to \_q^+ \mid q^+\_$

2. (Rückwärtsrechnung)
   $a'q' \to qa$ für alle $q \in Q$, $a \in \Gamma$ mit $\delta(q, a) = (q', a', R)$
   $q'ba' \to bqa$ für alle $q \in Q$, $a, b \in \Gamma$ mit $\delta(q, a) = (q', a', L)$
   $q'a' \to qa$ für alle $q \in Q$, $a \in \Gamma$ mit $\delta(q, a) = (q', a', N)$

3. (Terminationsregeln)
   $\_s' \to s'$
   $s'a \to as'$ für alle $a \in \Sigma$
   $s'\# \to \varepsilon$

Da die Ableitungen in G immer einer akzeptierenden Berechnung in umgekehrter Reihenfolge entsprechen, und umgekehrt zu jeder akzeptierenden Berechnung eine solche Ableitung möglich ist, gilt $L(G) = L(\mathcal{A})$. □

Für den Beweis der Umkehrung führen wir zunächst wieder eine nichtdeterministische Version des Maschinenmodells ein.

4.12 Definition
Eine nichtdeterministische Turingmaschine (NTM) $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ *besteht aus den gleichen Komponenten wie eine deterministische, jedoch mit einer Übergangsrelation* $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, N, R\})$.

Alternative Sichtweise: $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,N,R\}}$

4.13 Bemerkung
Eine NTM kann in einer gegebenen Situation (Zustand, gelesenes Zeichen) zwischen verschiedenen Aktionen (Zeichen schreiben, Kopfbewegung, Folgezustand) wählen. Dies gilt nicht für Endzustände.

4.14 Definition
Eine nichtdeterministische TM $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ *akzeptiert ein Wort* $w \in \Sigma^*$*, falls es mindestens eine Berechnung gibt, bei der sie in einem Endzustand stoppt. Die von* $\mathcal{A}$ *akzeptierte Sprache* $L(\mathcal{A}) \subseteq \Sigma^*$ *ist die Menge der akzeptierten Wörter.*

4.15 Satz
Zu einer NTM $\mathcal{A}$ *gibt es eine DTM* $\mathcal{B}$ *mit* $L(\mathcal{B}) = L(\mathcal{A})$.

■ **Beweis:** Wir konstruieren zunächst eine DTM $\mathcal{B}_3$ mit drei Bändern, wobei

- das erste Band die Eingabe enthält,
- das zweite Band eine mögliche Berechnung von $\mathcal{A}$ codiert und
- das dritte Band nacheinander die Konfigurationen der jeweiligen Berechnung von $\mathcal{A}$ enthält

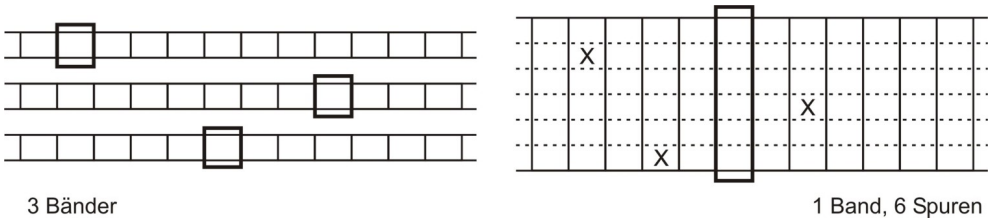
In jedem Schritt hat $\mathcal{A}$ höchstens

$$r = \max_{q,\in Q,\, a \in \Gamma} |\delta(q, a)|$$

viele Möglichkeiten. Jede Berechnung von $\mathcal{A}$ läßt sich also als Wort über dem Alphabet $\{1, \ldots, r\}$ codieren. $\mathcal{B}_3$ braucht daher nur auf dem zweiten Band die Wörter aus $\{1, \ldots, r\}^*$ (sortiert nach wachsender Länge) aufzuzählen und

bei der Berechnung auf dem dritten Band die darin codierten Auswahlen zu treffen. Nach dem Ende jeder Berechnung wird das dritte Band gelöscht und die Eingabe vom ersten Band darauf kopiert.

Eine DTM $\mathcal{B}$ kann nun die drei Bänder von $\mathcal{B}_3$ mit Hilfe eines Bandalphabets $\Gamma_3 = (\Gamma \times \{\mathsf{x}, \_\})^3 \cup \Sigma \cup \{\_\}$ simulieren, das neben den Inschriften der drei Bänder auch die Positionen der 3 Schreib-/Leseköpfe codiert:



Bei Eingabe von $w = a_1 \cdots a_n$ tauscht $\mathcal{B}$ die Bandinschrift zunächst gegen $(a_1, \mathsf{x}, \_, \mathsf{x}, a_1, \mathsf{x})(a_2, \_, \_, \_, a_2, \_) \cdots (a_n, \_, \_, \_, a_n, \_)$ aus. Um einen Schritt von $\mathcal{B}_3$ zu simulieren, geht $\mathcal{B}$

i) vom linkesten Feld zum rechtesten Feld, das mindestens eine x-Markierung hat, und merkt sich (durch entsprechende Zustände) die Zeichen an den Positionen der 3 x-Markierungen, dann

ii) zurück nach links, wobei an den mit x markierten Felder die Γ-Einträge und x-Markierungen entsprechend den Übergangen in $\mathcal{B}_3$ geändert werden

und schließlich in einen Zustand, der dem neuen Zustand von $\mathcal{B}_3$ entspricht. □

4.16 Korollar

Eine (nicht)deterministische TM mit k Bändern kann durch eine (nicht)deterministische TM mit einem Band simuliert werden.

4.17 Satz

Zu jeder Grammatik G existiert eine Turingmaschine $\mathcal{A}$ mit $L(G) = L(\mathcal{A})$.

■ **Beweis:** Eine nichtdeterministische TM $\mathcal{A}$ schreibt das Startsymbol der Grammatik hinter die Eingabe auf das Band, wählt dann jeweils eine Ableitungsregel, führt die Ersetzung durch und vergleicht das Resultat mit der Eingabe. Die Inschriften hinter der Eingabe sind damit genau die in der Grammtik ableitbaren Satzformen, sodass die Eingabe genau dann akzeptiert wird, wenn sie aus dem Startsymbol erzeugt werden kann. □

4.18 Folgerung

Eine Sprache ist genau dann rekursiv aufzählbar, wenn sie von einer Turingmaschine akzeptiert wird.

4.19 Bemerkung

Aus den Ergebnissen des nächsten Abschnitts wird folgen, dass es auch Sprachen gibt, die nicht vom Typ 0 sind.

Kapitel 5

Entscheidbarkeit und Berechenbarkeit

Wir wenden uns nun grundsätzlichen Fragen zu, nämlich den Fragen nach der prinzipiellen Lösbarkeit von Problemen. Dazu stellen wir auch einen Zusammenhang zwischen Sprachen und Berechnungen her.

5.1 Entscheidbarkeit

5.1 Definition

- *Eine Sprache $L \subseteq \Sigma^*$ heißt entscheidbar (auch: rekursiv), wenn es eine Turingmaschine gibt, die auf allen Eingaben $w \in \Sigma^*$ stoppt, und w genau dann akzeptiert, wenn $w \in L$.*

- *Eine Sprache $L \subseteq \Sigma^*$ heißt semi-entscheidbar (auch: rekursiv aufzählbar), wenn es eine Turingmaschine gibt, die eine Eingabe $w \in \Sigma^*$ genau dann akzeptiert, wenn $w \in L$.*

5.2 Bemerkung

Für semi-entscheidbare Sprachen wird nicht verlangt, dass eine akzeptierende TM auf allen Eingaben stoppt. Per Definition sind die semi-entscheidbaren Sprachen genau die Sprachen vom Typ 0.

Frage: Kann für Probleme, bei denen jede Instanz eine eindeutige Antwort hat, diese auch immer berechnet werden? nein

Wir zeigen die Existenz einer Sprache, die nicht entscheidbar ist, für die das Wortproblem also nicht durch eine Turingmaschine gelöst werden kann.

Dazu betrachten wir eine Normalform für Turingmaschinen: Eine DTM $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \_, F)$ ist in Normalform, falls

gegebenenfalls umcodieren!

$$\begin{aligned} Q &= \{q_1, \ldots, q_n\} \\ \Sigma &= \{0, 1\} \\ \Gamma &= \{0, 1, \_\} \\ s &= q_1 \\ F &= \{q_2\} \end{aligned}$$

5.3 Bemerkung
Jede Turingmaschine kann durch eine DTM in Normalform simuliert werden.

Normierte Turingmaschinen können durch Binärwörter codiert werden:

5.4 Definition
Sei $\mathcal{A}$ eine Turingmaschine in Normalform. Eine Transition $\delta(q_i, a_j) = (q_r, a_s, b_t)$ werde durch $0^i10^j10^r10^s10^t$ codiert, wobei $a_1 = 0$, $a_2 = 1$, $a_3 = \_$, $b_1 = L$, $b_2 = N$, $b_3 = R$ bedeute. Sind $c_1, \ldots, c_m$ die Codierungen aller Transitionen von $\mathcal{A}$ (in beliebiger Reihenfolge), dann heißt

$$\langle \mathcal{A} \rangle := 111c_1 11c_2 11 \ldots 11c_m 111$$

die Gödelnummer von $\mathcal{A}$.

5.5 Bemerkung
Jede Turingmaschine kann (in Normalform gebracht und) durch ein Binärwort dargestellt werden. Umgekehrt entspricht jedes Binärwort höchstens einer Turingmaschine (in Normalform).

Wir vereinbaren, dass Binärwörter, die keine Turingmaschine in diesem Sinn beschreiben, eine TM codieren, die auf keiner Eingabe stoppt und somit die leere Sprache $\emptyset$ akzeptiert.

5.6 Definition
Eine Turingmaschine, die bei Eingabe von $\langle \mathcal{A} \rangle w \in \{0, 1\}^*$ die Berechnung von $\mathcal{A}$ mit Eingabe w simuliert, heißt universelle Turingmaschine.

Zu $w \in \{0, 1\}^*$ sei $\mathcal{A}(w)$ die Turingmaschine mit Gödelnummer w, d.h. $\langle \mathcal{A}(w) \rangle = w$ (mit geeigneter Interpretation im Sonderfall, dass w keine Turingmaschine codiert).

5.7 Satz (Diagonalisierung)
Die Diagonalsprache

$$L_D := \{w \in \{0,1\}^* \,:\, w \notin L\left(\mathcal{A}(w)\right)\}$$

ist nicht entscheidbar.

■ **Beweis:** Wäre L_D entscheibar, so gäbe es eine Turingmaschine $\mathcal{A}$, die auf allen Eingaben hält und ein Wort $w \in \{0,1\}^*$ genau dann akzeptiert, wenn $w \in L_D$. Wende $\mathcal{A}$ nun auf die Eingabe $\langle\mathcal{A}\rangle \in \{0,1\}^*$ an:

1. Fall: $(\langle\mathcal{A}\rangle \in L_D)$
Dann akzeptiert $\mathcal{A}$ die Eingabe $\langle\mathcal{A}\rangle$, d.h. $\langle\mathcal{A}\rangle \in L(\mathcal{A})$, im Widerspruch zur Definition von L_D.

2. Fall: $(\langle\mathcal{A}\rangle \notin L_D)$
Dann akzeptiert $\mathcal{A}$ die Eingabe $\langle\mathcal{A}\rangle$ nicht, d.h. $\langle\mathcal{A}\rangle \notin L(\mathcal{A})$, im Widerspruch zur Definition von L_D.

□

Dieser Satz liefert nicht nur ein unentscheidbares Problem (das Wortproblem für L_D), sondern hat weit reichende Konsequenzen:

5.8 Definition (Halteproblem)
Das Halteproblem ist das Wortproblem der Sprache

$$L_H := \{wv \in \{0,1\}^* \,:\, \mathcal{A}(w) \text{ stoppt auf Eingabe } v\} \ .$$

5.9 Satz
Das Halteproblem ist unentscheidbar.

■ **Beweis:** Angenommen, es existiert eine stets haltende TM $\mathcal{A}$ mit $L(\mathcal{A}) = L_H$. Wir konstruieren daraus eine stets haltende TM, die $\overline{L_D} = \{0,1\}^* \setminus L_D$ entscheidet. Dies ist dann ein Widerspruch zur Unentscheidbarkeit von L_D, da mit einer Sprache auch deren Komplement entscheidbar ist. Um zu entscheiden, ob eine Eingabe $w \in \overline{L_D}$, prüft die konstruierte Maschine zunächst, ob w eine Turingmaschine entsprechend Definition 5.4 codiert. Falls ja (andernfalls ist $w \in L_D$, da $L(\mathcal{A}(w)) = \emptyset$ gemäß Bemerkung 5.5) wird $\mathcal{A}$ auf ww angewandt.

1. Fall: $(ww \notin L_H)$
Dann stoppt $\mathcal{A}(w)$ nicht auf w, also ist $w \notin L(\mathcal{A}(w))$ und somit $w \notin \overline{L_D}$.

2. Fall: $(ww \in L_H)$
Dann stoppt $\mathcal{A}(w)$ auf w und es ist $w \in \overline{L_D} \iff w \in L(\mathcal{A}(w))$.

Wir hätten also ein Entscheidungsverfahren für $\overline{L_D}$ und damit auch für L_D. Dies ist ein Widerspruch zu Satz 5.7. □

5.10 Satz
Die universelle Sprache

$$L_U := \{wv \in \{0,1\}^* : v \in L(\mathcal{A}(w))\}$$

ist nicht entscheidbar.

■ **Beweis:** Wäre L_U entscheidbar, so könnte insbesondere für jede Eingabe $ww \in \{0,1\}^*$ entschieden werden, ob $w \in L(\mathcal{A}(w))$ und wir hätten somit ein Entscheidungsverfahren für $\overline{L_D}$ – Widerspruch. □

5.11 Satz
Die universelle Sprache L_U ist semi-entscheidbar.

■ **Beweis:** L_U ist die Sprache der universellen Turingmaschine: Bei Eingabe von $wv \in \{0,1\}^*$ gilt:

- Akzeptiert $\mathcal{A}(w)$ die Eingabe v, dann nach endlich vielen Schritten, sodass wv auch von der universellen TM nach endlich vielen Schritten akzeptiert wird.
- Akzeptiert $\mathcal{A}(w)$ die Eingabe nicht, dann wird wv auch von der universellen TM nicht akzeptiert (unabhängig davon, ob $\mathcal{A}(w)$ stoppt oder nicht).

□

5.12 Lemma
Sind eine Sprache $L \subseteq \Sigma^*$ und ihr Komplement $\overline{L}$ semi-entscheidbar, dann ist L sogar entscheidbar.

■ **Beweis:** Seien $\mathcal{A}$ und $\overline{\mathcal{A}}$ Turingmaschinen mit $L(\mathcal{A}) = L$ und $L(\overline{\mathcal{A}}) = \overline{L}$ sowie die Zustandsmengen Q und $\overline{Q}$ gegeben. Wir konstruieren eine TM $\mathcal{B}$ mit zwei Bändern, die auf allen Eingaben hält und genau L akzeptiert.

Die Maschine $\mathcal{B}$ kopiert zunächst die Eingabe auf das zweite Band. Die Zustandsmenge enthalte alle Paare $Q \times \overline{Q}$ und $\mathcal{B}$ arbeite auf dem ersten Band wie $\mathcal{A}$ und gleichzeitig auf dem zweiten Band wie $\overline{\mathcal{A}}$. Eine der beiden Simulationen stoppt in einem Endzustand, sodass $\mathcal{B}$ eine Eingabe akzeptiert oder verwirft, je nachdem, ob $\mathcal{A}$ oder $\overline{\mathcal{A}}$ akzeptiert hätte. □

5.13 Satz
$\overline{L_U}$ ist nicht rekursiv aufzählbar.

■ **Beweis:** Angenommen, $\overline{L_U}$ sei rekursiv aufzählbar. Da L_U rekursiv aufzählbar ist, folgt dann mit obigem Lemma, dass L_U sogar entscheidbar ist – Widerspruch. □

5.2 Berechenbarkeit

Wenn eine Turingmaschine $\mathcal{A}$ bei Eingabe von $w \in \Sigma^*$ stoppt, kann die Bandinschrift als Ausgabe gedeutet werden. Sie berechnet damit eine (partielle) Funktion $f_{\mathcal{A}} : \Sigma^* \to \Gamma^*$.

5.14 Definition
Eine (partielle) Funktion $f : \Sigma^* \to \Gamma^*$ heißt (Turing-)berechenbar, wenn es eine Turingmaschine gibt, die auf einer Eingabe $w \in \Sigma^*$ genau dann stoppt, wenn $f(w)$ definiert ist, und dabei $f(w) \in \Gamma^*$ ausgibt. In diesem Fall realisiert die Turingmaschine die Funktion f.

5.15 Bemerkung
Eine Sprache $L \subseteq \Sigma^*$ ist entscheidbar genau dann, wenn ihre charakteristische Funktion $f_L : \Sigma^* \to \{0, 1\}$ ($f_L(w) = 1 \iff w \in L$) berechenbar ist.

5.16 Folgerung
Es gibt Funktionen, die nicht (Turing-)berechenbar sind.

5.17 Vermutung (Church-Turing-These)

Jede im intuitiven Sinne berechenbare Funktion ist (Turing-)berechenbar.

Alternativ: Jedes algorithmisch lösbare Problem kann mit Hilfe einer Turingmaschine gelöst werden.

Nach dem für allgemeine Aussagen besonders geeigneten Automatenmodell der Turingmaschine betrachten wir noch eine von Grund auf mathematische Charakterisierung des Berechenbarkeitsbegriffs, der diese These untermauert.

5.18 Definition (Primitiv rekursive Funktionen)

Die Grundfunktionen sind definiert durch

$$\begin{array}{lllll} \underline{\text{null}}^{(k)} : \mathbb{N}_0^k \to \mathbb{N}_0 & \textit{mit} & (n_1, \ldots, n_k) & \mapsto & 0 \\ \underline{\text{succ}} \quad : \mathbb{N}_0 \to \mathbb{N}_0 & \textit{mit} & n & \mapsto & n+1 \\ \underline{\text{proj}}_i^{(k)} : \mathbb{N}_0^k \to \mathbb{N}_0 & \textit{mit} & (n_1, \ldots, n_k) & \mapsto & n_i \end{array}$$

für alle $k \in \mathbb{N}_0$ und $1 \leq i \leq k$. Für alle $k \in \mathbb{N}_0$, $l \in \mathbb{N}$ entsteht $g : \mathbb{N}_0^k \to \mathbb{N}_0$ aus $f : \mathbb{N}_0^l \to \mathbb{N}_0$ und $f_1 : \mathbb{N}_0^k \to \mathbb{N}_0$,...,$f_l : \mathbb{N}_0^k \to \mathbb{N}_0$ durch Komposition, falls für alle $n_1, \ldots, n_k \in \mathbb{N}_0$ gilt

$$g(n_1, \ldots, n_k) = f\left(f_1(n_1, \ldots, n_k), \ldots, f_l(n_1, \ldots, n_k)\right) \ .$$

Für alle $k \in \mathbb{N}_0$ entsteht $g : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ aus $f_1 : \mathbb{N}_0^k \to \mathbb{N}_0$ und $f_2 : \mathbb{N}_0^{k+2} \to \mathbb{N}_0$ durch primitive Rekursion, falls für alle $n_1, \ldots, n_k \in \mathbb{N}_0$ gilt

$$\begin{array}{rcl} g(n_1, \ldots, n_k, 0) & = & f_1(n_1, \ldots, n_k) \\ g(n_1, \ldots, n_k, i+1) & = & f_2(n_1, \ldots, n_k, i, g(n_1, \ldots, n_k, i)) \ . \end{array}$$

Die primitiv rekursiven Funktionen sind genau diejenigen, die durch endlichmalige Anwendung von Komposition und primitiver Rekursion aus den Grundfunktionen gebildet werden können.

5.19 Beispiel

1. *Die natürlichen Zahlen sind (konstante) nullstellige primitiv rekursive Funktionen*

$$\underline{n} = \underbrace{\underline{\text{succ}}(\underline{\text{succ}}(\cdots \underline{\text{succ}}}_{n \text{ mal}}(\underline{\text{null}}^{(0)})\cdots)) \qquad \text{für alle } n \in \mathbb{N}_0$$

2. *Die Additionsfunktion* $\underline{add} : \mathbb{N}_0^2 \to \mathbb{N}_0$ *ist primitiv rekursiv:*

$$\begin{aligned} \underline{add}(n,0) &= \underline{proj}_1^{(1)}(n) \\ \underline{add}(n,m+1) &= \underline{succ}(\underline{proj}_3^{(3)}(n,m,\underline{add}(n,m))) \ . \end{aligned}$$

3. *Die Multiplikationsfunktion* $\underline{mult} : \mathbb{N}_0^2 \to \mathbb{N}_0$ *ist primitiv rekursiv:*

$$\begin{aligned} \underline{mult}(n,0) &= \underline{null}^{(1)}(n) \\ \underline{mult}(n,m+1) &= \underline{add}(\underline{proj}_1^{(3)}(n,m,\underline{mult}(n,m)),\ \underline{proj}_3^{(3)}(n,m,\underline{mult}(n,m))) \ . \end{aligned}$$

4. *Die Ackermannfunktion* $\underline{ack} : \mathbb{N}_0^2 \to \mathbb{N}_0$, *definiert durch*

wird oft mit leichten Abwandlungen definiert

$$\underline{ack}(i,n) = \begin{cases} 2n & \text{für } i = 0,\ n \in \mathbb{N}_0 \\ \underline{ack}(i-1,2) & \text{für } i \in \mathbb{N},\ n = 0 \\ \underline{ack}(i-1,\underline{ack}(i,n-1)) & \text{für } i,n \in \mathbb{N} \ , \end{cases}$$

ist nicht *primitiv rekursiv, denn sie wächst schneller als jede primitiv rekursive Funktion (ohne Beweis). Einen Eindruck vom Wachstum der Ackermann-Funktion bekommt man, wenn man für alle* $i \in \mathbb{N}_0$ *die Funktionen* $A_i(n) = \underline{ack}(i,n)$ *definiert. Diese können wie folgt gedeutet werden:*

$$\begin{aligned} A_0(n) &= \underbrace{2+\ldots+2}_{n\text{-mal}} = 2\cdot n \qquad n\text{-fache Summe von } 2 \\ A_1(n) &= \underbrace{2\cdot\ldots\cdot 2}_{n\text{-mal}} = 2^n \qquad n\text{-faches Produkt von } 2 \\ A_2(n) &= \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n\text{-mal}} \qquad n\text{-fache Potenz von } 2 \\ &\ \ \vdots \end{aligned}$$

Beispielsweise ist $\underline{ack}(2,5) = A_2(5) = 2^{65536} \approx 10^{19728}$.

5.20 Definition (Rekursive Funktionen)
Für alle $k \in \mathbb{N}_0$ entsteht $\mu f : \mathbb{N}_0^k \to \mathbb{N}_0$ aus $f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ durch Minimalisierung, wenn

$$\mu f(n_1, \ldots, n_k) = n \iff \begin{cases} f(n_1, \ldots, n_k, n) = 0 & \text{und} \\ f(n_1, \ldots, n_k, n') > 0 & \forall n' < n \end{cases}$$

Die (μ-)rekursiven Funktionen sind genau diejenigen, die durch endlichmalige Anwendung von Komposition, primitiver Rekursion und Minimalisierung aus den Grundfunktionen gebildet werden können.

5.21 Satz
Eine Funktion ist genau dann (Turing-)berechenbar, wenn sie rekursiv ist.

Kapitel 6

Komplexitätstheorie

Abschließend untersuchen wir lösbare Probleme etwas genauer im Hinblick auf den für die Lösung benötigten Aufwand, d.h. die Effizienz von Lösungsverfahren.

Dazu werden wir zunächst beliebige (formale) Probleme durch formale Sprachen ausdrücken.

6.1 Beispiel (Traveling Salesman Problem, TSP)
Gegeben: *Ein vollständiger Graph $G = (V, E)$, d.h.*

$$\begin{aligned} V &= \{1, \ldots, n\} \qquad (\text{„}\textit{Städte}\text{“}) \\ E &= \{\{i, j\} : i, j \in V, i \neq j\} \qquad (\text{„}\textit{Verbindungen}\text{“}) \end{aligned}$$

und eine positive, ganzzahlige Distanzfunktion $d : E \to \mathbb{N}$ („Entfernungen“).

Gesucht: *(Hier 3 Varianten)*

(1) (Optimierungsproblem, Optimallösungsproblem)
Eine Rundreise (Tour), die jede Stadt genau einmal passiert, und minimale Länge unter allen Rundreisen hat. Äquivalent: eine Permutation $\pi : V \to V$ so, dass

$$d(\pi(n), \pi(1)) + \sum_{i=1}^{n-1} d(\pi(i), \pi(i+1))$$

minimal ist.

(2) *(Optimalwertproblem)*
Die minimale Länge einer Tour.

(3) *(Entscheidungsproblem)*
Antwort auf die Frage „Gibt es eine Tour der Länge höchstens k?“, wobei $k \in \mathbb{N}$ ein zusätzlicher Eingabeparameter ist.

Wir zeigen zunächst, wie ein Entscheidungsproblem in ein Wortproblem übertragen werden kann, und dann, dass mit Hilfe eines Lösungsverfahrens für (3) auch (1) und (2) gelöst werden können.

Ein Problem Π ist gegeben durch

- eine allgemeine Beschreibung der relevanten Parameter
- eine genaue Beschreibung der Eigenschaften einer Lösung

Ein Problembeispiel I (Instanz) von Π entsteht durch Festlegung der Parameter. Ein Codierungsschema $\sigma : \Pi \to \Sigma^*$ ordnet jeder Instanz eines Problems ein Wort über einem Alphabet zu. Zwei Codierungsschemata σ_1, σ_2 heißen äquivalent bzgl. eines Problems Π, falls es Polynome p_1, p_2 mit

$$|\sigma_1(I)| \leq p_1(|\sigma_2(I)|) \quad \text{und} \quad |\sigma_2(I)| \leq p_2(|\sigma_1(I)|)$$

für alle I von Π gibt.

Ist Π ein Entscheidungsproblem, können die Instanzen in eine Klasse von Ja-Beispielen $J_\Pi \subseteq \Pi$ und eine Klasse von Nein-Beispielen $N_\Pi = \Pi \setminus J_\Pi$ aufgeteilt werden.

Ein Codierungsschema $\sigma : \Pi \to \Sigma^*$ liefert dann drei Klassen von Wörtern über Σ, die

- kein Beispiel $I \in \Pi$
- ein Ja-Beispiel $I \in J_\Pi$
- ein Nein-Beispiel $I \in N_\Pi$

codieren.

6.2 Definition
Die zu einem Entscheidungsproblem Π *und Codierungsschema* $\sigma : \Pi \to \Sigma^*$ *gehörige Sprache besteht aus den Codierungen aller Ja-Beispiele, d.h.*

$$L(\Pi, \sigma) := \{w \in \Sigma^* \,:\, w = \sigma(I) \text{ für ein } I \in J_\Pi\}$$

6.3 Beispiel
Zahlen können dezimal, binär, unär, usw. codiert werden. Bei k-ärer Codierung (d.h. durch Wörter über $\{0, \ldots, k-1\}$*) ist die Länge der Codierung*

- *einer natürlichen Zahl* $i \in \mathbb{N}$ *gegeben durch* $|\sigma(i)| = \lfloor 1 + \log_k i \rfloor$,
- *einer rationalen Zahl* $r = \frac{p}{q} \in \mathbb{Q}$ *gegeben durch* $|\sigma(p)| + |\sigma(q)| + 1$ *und* Trennzeichen!
- *eines Vektors* $x = (x_1, \ldots, x_d) \in \mathbb{Q}^d$ *entsprechend* $|\sigma(x)| = \sum_{i=1}^{d} |\sigma(x_i)| + (d-1)$

Ein Graph kann z.B. durch seine Adjazenzmatrix codiert werden.

6.4 Bemerkung
Für $k \geq 2$ *sind alle k-ären Codierungen äquivalent, und nicht äquivalent zur unären Codierung.*

6.5 Definition
Eine Turingmaschine $\mathcal{A}$ *löst ein Entscheidungsproblem* Π *mit Codierungsschema* $\sigma : \Pi \to \Sigma^*$*, falls sie auf jeder Eingabe* $w \in \Sigma^*$ *stoppt und* $L(\mathcal{A}) = L(\Pi, \sigma)$.

6.6 Bemerkung
Ein Entscheidungsproblem Π *ist also lösbar, wenn* $L(\Pi, \sigma)$ *für ein Codierungsschema* σ *entscheidbar ist.*

6.7 Definition
Sei $\mathcal{A}$ *eine deterministische Turingmaschine. Die Zeitkomplexitätsfunktion* $T_\mathcal{A} : \mathbb{N} \to \mathbb{N}$ *ist definiert durch*

$$T_\mathcal{A}(n) = \max \left\{ t \,:\, \begin{array}{l} \mathcal{A} \text{ stoppt bei Eingabe von } w \in \Sigma^n \text{ nach } t \text{ Schritten} \\ \text{bzw. } t = 1, \text{ falls } \mathcal{A} \text{ nicht stoppt} \end{array} \right\} .$$

6.8 Definition
Die Klasse $\mathcal{P}$ ist die Menge aller Sprachen (Probleme), für die eine deterministische Turingmaschine mit polynomialer Zeitkomplexitätsfunktion existiert, d.h. es gibt eine DTM $\mathcal{A}$ und ein Polynom p mit

$$T_{\mathcal{A}}(n) \leq p(n)$$

für alle $n \in \mathbb{N}$.

6.9 Bemerkung
Wir erlauben nur Codierungsschemata, die

- *keine überflüssige Information enthalten*
- *äquivalent zur Binärcodierung sind*

und können daher auf die Angabe des speziellen Codierungsschemas verzichten.

Am Beispiel des TSP zeigen wir noch, wie Optimierungs- und Optimalwertprobleme polynomial auf Entscheidungsprobleme zurückgeführt werden können. Ist das Entscheidungsproblem in polynomialer Zeit lösbar, dann also auch die zugehörigen Optimalwert- und Optimallösungsprobleme.

Algorithmus 3: TSP: Entscheidungsproblem $\curvearrowright$ Optimalwertproblem

$D \leftarrow \max_{i,j} d(i,j)$
$L \leftarrow 0$ und $H \leftarrow n \cdot D$
while $L < H$ **do** — binäre Suche
  if *es ex. Tour der Länge* $\leq \lceil \frac{L+H}{2} \rceil$ **then**
    $H \leftarrow \lceil \frac{L+H}{2} \rceil$
  else
    $L \leftarrow \lceil \frac{L+H}{2} + 1 \rceil$

H ist die Länge einer kürzesten Tour

Algorithmus 4: TSP: Optimalwertproblem $\curvearrowright$ Optimierungsproblem

$D \leftarrow \max_{i,j} d(i,j)$
OPT $\leftarrow$ Länge einer kürzesten Tour
for $i = 1, \ldots, n-1$ **do**
  for $j = i+1, \ldots, n$ **do**
    $d = d(i,j)$
    $d(i,j) = D + 1$
    if *Länge einer kürzesten Tour* $>$ *OPT* **then**
      $d(i,j) \leftarrow d$

eine kürzeste Tour besteht aus allen $\{i,j\}$ mit $d(i,j) < D+1$.

$\mathcal{P}$ ist bezüglich deterministischer TMn definiert. Bei den endlichen Automaten führt Nichtdeterminismus (bei gleich bleibender Erkennungsmächtigkeit) eventuell zu einem Effizienzgewinn.

6.10 Definition
Sei $\mathcal{A}$ eine nichtdeterministische TM und $T_{\mathcal{A}} : \Sigma^* \cup \mathbb{N} \to \mathbb{N}$ definiert durch

$$T_{\mathcal{A}}(w) = \begin{cases} \min \left\{ t \ : \ \begin{array}{l} t \text{ ist die Länge einer} \\ \text{akzeptierenden Berechnung} \end{array} \right\} & \text{falls } w \in L(\mathcal{A}) \\ 1 & \text{falls } w \notin L(\mathcal{A}) \end{cases}$$

und

$$T_{\mathcal{A}}(n) = \max \{ T_{\mathcal{A}}(w) \ : \ w \in \Sigma^n \} \ .$$

6.11 Bemerkung
Da wir uns nur für akzeptierende Berechnungen interessieren, ist $T_{\mathcal{A}}(n) = 1$, falls es kein Wort $w \in \Sigma^n \cap L(\mathcal{A})$ gibt.

6.12 Definition
Die Klasse $\mathcal{NP}$ ist die Menge aller Sprachen (Probleme), für die eine nichtdeterministische Turingmaschine mit polynomialer Zeitkomplexitätsfunktion existiert, d.h. es gibt eine NTM $\mathcal{A}$ und ein Polynom p mit

$$T_{\mathcal{A}}(n) \leq p(n)$$

für alle $n \in \mathbb{N}$.

Es ist häufig praktischer, den Nichtdeterminismus einer NTM zu bündeln. Wir betrachten daher nichtdeterministische TMn, die aus DTMn durch Vorschalten eines Orakels entstehen: Das Orakel fügt zunächst ein Wort aus Γ^* zur Eingabe hinzu, um danach deterministisch weiter zu rechnen (vgl. Beweis der Äquivalenz von NTM und DTM). Die Klasse $\mathcal{NP}$ ändert sich durch diese Modifikation nicht.

Verwendung des Orakels:

1. Das Orakel schreibt eine Lösung (Berechnungsweg, Beweis) aufs Band.
2. Der deterministische Teil der Maschine überprüft die Lösung bzw. wertet sie aus.

6.13 Beispiel (TSP, Entscheidungsvariante)
Zu gegebenem Graphen $G = (V, E)$ schreibt das Orakel eine Permutation $\pi : V \to V$ aufs Band. Eine DTM kann in polynomialer Zeit prüfen, ob die Permutation π einer Tour der Länge höchstens k entspricht. Also TSP $\in \mathcal{NP}$. Es ist allerdings nicht bekannt, ob TSP auch in $\mathcal{P}$ liegt.

6.1 $\mathcal{NP}$-Vollständigkeit

Offensichtlich gilt $\mathcal{P} \subseteq \mathcal{NP}$.

6.14 Frage ($\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Problem)
Gibt es Probleme in $\mathcal{NP}$, die nicht in $\mathcal{P}$ sind?

s. Prolog
Vermutung:
$\mathcal{P} \neq \mathcal{NP}$

Die Frage bleibt offen, wir zeigen aber, dass einige Probleme in $\mathcal{NP}$ mindestens so schwer sind wie alle anderen in $\mathcal{NP}$, sodass aus der Zugehörigkeit eines solchen Problems zu $\mathcal{P}$ sofort $\mathcal{P} = \mathcal{NP}$ folgen würde.

6.15 Definition
Eine polynomiale Transformation einer Sprache $L_1 \subseteq \Sigma_1^*$ in eine Sprache $L_2 \subseteq \Sigma_2^*$ ist eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$ mit folgenden Eigenschaften:

- *es existiert eine polynomiale deterministische TM, die f berechnet*
- *für alle $w \in \Sigma_1^*$ gilt: $w \in L_1 \iff f(w) \in L_2$*

Wir schreiben dann $L_1 \preccurlyeq L_2$ („L_1 ist polynomial transformierbar in L_2“).

„nicht schwieriger als“

6.16 Definition
Eine Sprache L heißt $\mathcal{NP}$-vollständig, falls

1. *$L \in \mathcal{NP}$ und*
2. *für* alle *$L' \in \mathcal{NP}$ gilt $L' \preccurlyeq L$.*

6.17 Folgerung
Falls $L_1, L_2 \in \mathcal{NP}$ und $L_1 \preccurlyeq L_2$ für L_1 $\mathcal{NP}$-vollständig, dann ist auch L_2 $\mathcal{NP}$-vollständig.

Der Nachweis, dass ein Entscheidungsproblem Π $\mathcal{NP}$-vollständig ist, kann nach folgendem Schema geführt werden:

1. Zeige, dass $\Pi \in \mathcal{NP}$.
2. Zeige für *ein* (bekanntermaßen) $\mathcal{NP}$-vollständiges Problem Π', dass $\Pi' \preccurlyeq \Pi$.

Dabei wird ein Problem Π jeweils mit einer zugehörigen Sprache $L(\Pi, \sigma)$ identifiziert. Die Korrektheit des Vorgehens ergibt sich unmittelbar aus der Transitivität der polynomialen Reduzierbarkeit ($L_1 \preccurlyeq L_2$ und $L_2 \preccurlyeq L_3 \implies L_1 \preccurlyeq L_3$).

Eine Verankerung liefert das folgende Problem:

6.18 Problem (SAT: Satisfiability; Erfüllbarkeitsproblem)
Gegeben: *Eine Menge $X = \{x_1, \ldots, x_n\}$ von booleschen Variablen, eine Menge $C = \{C_1, \ldots, C_m\}$ von Disjunktionen aus Literalen $x_i, \overline{x_i}$ über X, d.h. Klauseln der Form $C_i = y_1 \vee \ldots \vee y_k$ mit $y_i \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$.*

Frage: *Gibt es eine (Wahrheits-)Belegung $t : X \to \{0, 1\}$ so, dass*

$$\varphi = C_1 \wedge \ldots \wedge C_m$$

erfüllt ist, d.h.

$$t(\varphi) = t(C_1) = \ldots = t(C_m) = 1 \ ?$$

6.19 Satz (Cook)
SAT ist $\mathcal{NP}$-vollständig.

■ **Beweisskizze:**

1. SAT $\in \mathcal{NP}$, da für eine Instanz mit m Klauseln eine Wahrheitsbelegung der n Variablen geraten und in Zeit proportional zur Länge der Formel überprüft werden kann.

2. Zu einer NTM $\mathcal{A}$ für ein gegebenes Problem in $\mathcal{NP}$ wird eine Konstruktionsvorschrift angegeben, die eine Eingabe für $\mathcal{A}$ in polynomialer Zeit so in eine Formel für SAT umwandelt, dass die Formel genau dann erfüllbar ist, wenn es eine akzeptierende Berechnung gibt.

   Je eine Variable drückt aus,

   - dass $\mathcal{A}$ zum Zeitpunkt t an der Stelle k steht
   - dass $\mathcal{A}$ zum Zeitpunkt t im Zustand q ist
   - dass der Bandinhalt an der Stelle k zum Zeitpunkt t das Symbol a ist

   Die Klauseln drücken aus,

   - dass $\mathcal{A}$ zum Zeitpunkt t in genau einem Zustand ist
   - dass der Schreib-/Lesekopf zum Zeitpunk t an genau einer Stelle steht
   - dass das Band zum Zeitpunkt t an der Stelle k genau ein Zeichen enthält
   - welche Anfangskonfiguration $\mathcal{A}$ hat
   - welche Konfiguration zum Zeitpunkt $t+1$ auf eine gegebene Konfiguration zum Zeitpunkt t folgen kann
   - dass $\mathcal{A}$ in einem akzeptierenden Zustand endet.

   Da die Rechenzeit von $\mathcal{A}$ polynomial begrenzt ist, ist auch die Transformation polynomial.

□

6.20 Satz
Sei L eine $\mathcal{NP}$-vollständige Sprache, dann gilt

1. $L \in \mathcal{P} \Rightarrow \mathcal{P} = \mathcal{NP}$
2. $L \notin \mathcal{P} \Rightarrow \mathcal{P} \neq \mathcal{NP}$ *und für alle $\mathcal{NP}$-vollständigen Sprachen L' gilt* $L' \notin \mathcal{P}$

■ **Beweis:**

1. Ist $L \in \mathcal{P}$, so existiert eine polynomiale DTM, die L akzeptiert. Da L $\mathcal{NP}$-vollständig ist, existiert zu jeder anderen Sprache L' in $\mathcal{NP}$ eine polynomiale DTM, die L' in L transformiert. Die Hintereinanderausführung liefert eine polynomiale DTM, die L' akzeptiert.
2. Sei $L \notin \mathcal{P}$ aber es existiere eine $\mathcal{NP}$-vollständige Sprache $L' \in \mathcal{P}$. Dann folgt mit 1., dass $\mathcal{P} = \mathcal{NP}$ (im Widerspruch zu $L \in \mathcal{NP} \setminus \mathcal{P}$).

□

6.21 Satz
Zu jeder Sprache $L \in \mathcal{NP}$ gibt es ein Polynom p und eine deterministische TM mit exponentieller Zeitkomplexität $2^{p(n)}$, die L akzeptiert.

■ **Beweisskizze:** Die NTM zu L habe Zeitkomplexität beschränkt durch $q(n)$ für ein Polynom q und in jedem Rechenschritt höchstens r Alternativen. Eine DTM braucht dann nur alle Berechnungen der Länge $r^{q(n)} = 2^{q(n)\cdot \log r}$ zu simulieren. □

siehe Beweis $L(NTM) = L(DTM)$

6.22 Problem (k-SAT)
Gegeben: *Eine Menge $X = \{x_1, \ldots, x_n\}$ boolescher Variablen und eine Menge $C = \{C_1, \ldots, C_m\}$ von Klauseln mit $|C_i| = k$ für alle $i = 1, \ldots, m$.*

Frage: *Gibt es eine erfüllende Belegung für $\varphi = C_1 \wedge \cdots \wedge C_m$?*

6.23 Satz
3-SAT ist $\mathcal{NP}$-vollständig.

■ **Beweis:**

1. Da SAT $\in \mathcal{NP}$ ist auch 3-SAT $\in \mathcal{NP}$.

2. Wir zeigen, SAT $\preccurlyeq$ 3-SAT.

   Gegeben sei eine Menge $C = \{C_1, \ldots, C_m\}$ von Klauseln über Variablen $X = \{x_1, \ldots, x_n\}$.

   Klauseln C_i mit $|C_i| = 1, 2$ werden durch Wiederholung des ersten Literals zu 3-elementigen Klauseln verlängert. Betrachte daher $C_i = y_1 \vee \cdots \vee y_k$ mit $k > 3$.

   Wir führen $k - 3$ neue Variablen z_j^i ein und ersetzen C_i durch

$$\begin{aligned} C_1^i &= y_1 \vee y_2 \vee z_1^i \\ C_2^i &= \overline{z_1^i} \vee y_3 \vee z_2^i \\ C_3^i &= \overline{z_2^i} \vee y_4 \vee z_3^i \\ &\vdots \\ C_{k-2}^i &= \overline{z_{k-3}^i} \vee y_{k-1} \vee y_k \end{aligned}$$

   vgl. Chomsky-NF

   $C_1^i \wedge \cdots \wedge C_{k-2}^i$ ist genau dann erfüllbar, wenn eines der Literale $y_1, \ldots, y_k$ mit 1 belegt wird. Damit ist die Eingabe für SAT genau dann erfüllbar, wenn die transformierte Klauselmenge für 3-SAT erfüllbar ist.

   Die Transformation ist polynomial.

□

6.24 Bemerkung
2-SAT $\in \mathcal{P}$.

6.25 Problem (DHC: gerichteter Hamiltonkreis)
Gegeben: *Ein gerichteter Graph* $G = (V, E)$

Frage: *Existiert ein gerichteter Zykel in G, der jeden Knoten genau einmal enthält?*

6.26 Satz
DHC ist $\mathcal{NP}$*-vollständig.*

■ **Beweis:**

1. DHC $\in \mathcal{NP}$, da der Kreis geraten werden kann.

2. Wir zeigen, 3-SAT $\preccurlyeq$ DHC.

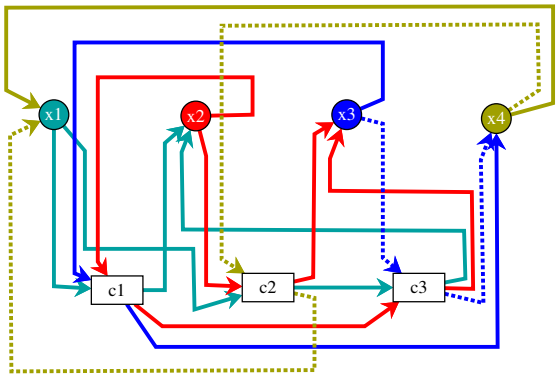
   Dazu konstruieren wir aus einer Instanz (X, C) für 3-SAT einen gerichteten Graphen, der genau dann ein Hamiltonkreis enthält, wenn es eine erfüllende Belegung gibt.

   Der Graph enthält einen Knoten pro Variable und zunächst einen weiteren Knoten pro Klausel.

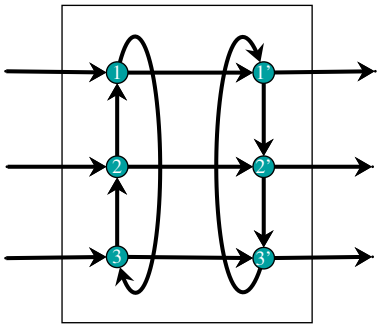
   Für jedes Literal führen wir einen gerichteten Weg ein, der beim zugehörigen Variablenknoten beginnt, dann alle Klauselknoten besucht, in denen das Literal auftritt und beim Knoten der nachfolgenden Variable endet.

   Variablenknoten haben damit je 2 einlaufende und 2 auslaufende Kanten, Klauselknoten deren 3.

   Beispiel: $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$



   Ein Hamiltonkreis benutzt an jedem Variablenknoten genau eine auslaufende Kante und korrespondiert daher mit einer Belegung. Da es aber egal ist, ob eine erfüllende Belegung ein, zwei oder drei Literale einer Klausel erfüllt, ersetzen wir die Klauselknoten noch durch



Zu jeder erfüllenden Belegung existiert nun ein gerichteter Hamiltonkreis und umgekehrt.

□

6.27 Bemerkung
Das Problem DEC (gerichteter Eulerkreis: ein gerichteter Zykel, in dem jede Kante genau einmal vorkommt) ist in $\mathcal{P}$.

6.28 Problem (HC (Hamiltonkreis))
Gegeben: *Ein ungerichteter Graph $G = (V, E)$.*

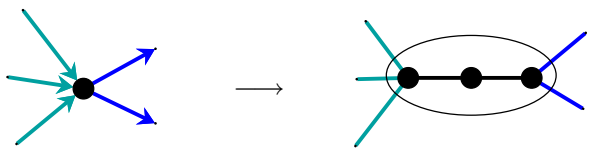
Frage: *Gibt es einen Zykel in G, der jeden Knoten genau einmal enthält?*

6.29 Satz
HC ist $\mathcal{NP}$-vollständig.

■ **Beweis:**

1. HC $\in \mathcal{NP}$, da wir den Zykel raten können.
2. Wir zeigen, dass DHC $\preccurlyeq$ HC.

   Dazu wird im gerichteten Graphen der Instanz für DHC an jedem Knoten die folgende Ersetzung durchgeführt:



Ein gerichteter Hamiltonkreis liefert sofort einen zugehörigen ungerichteten. Für einen ungerichteten Hamiltonkreis legen wir eine Durchlaufrichtung fest, die der Richtung irgendeiner Kante im gerichteten Graph entspricht.

Dies liefert einen gerichteten Hamiltonkreis im gerichteten Graphen, denn würde im ungerichteten Graphen ein Knoten in der „falschen“ Richtung verlassen, könnte der Hamiltonkreis den mittleren Knoten nicht mehr enthalten.

□

6.30 Bemerkung
Es gilt wieder $EC \in \mathcal{P}$ (ungerichteter Eulerkreis)

6.31 Satz
TSP ist $\mathcal{NP}$-vollständig.

■ **Beweis:**

1. $\text{TSP} \in \mathcal{NP}$ wissen wir bereits.

2. Wir zeigen, dass $\text{HC} \preccurlyeq \text{TSP}$.

   Als Instanz für HC sei der ungerichtete Graph $G = (V, E)$ gegeben.

   Wir transformieren ihn in einen vollständigen Graphen $G' = (V, E')$ mit Distanzfunktion:

   $$d(\{i,j\}) = \begin{cases} 1 & \text{falls } \{i,j\} \in E \\ 2 & \text{falls } \{i,j\} \notin E \end{cases}$$

   als Eingabe für TSP. Dann existiert ein Hamiltonkreis in G genau dann, wenn es in G' eine Tour der Länge höchstens $|V|$ gibt.

□

Literaturverzeichnis

[1] Michael R. Garey and David S. Johnson: *Computers and Intractability – A Guide to the Theory of $\mathcal{NP}$-Completeness*. Freeman, 1979.

[2] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Pearson Studium, 2. Auflage 2002.

[3] Uwe Schöning, *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, 4. Auflage 2001.

[4] Ingo Wegener, *Theoretische Informatik – eine algorithmenorientierte Einführung*. B.G. Teubner, 2. Auflage 1999.