Einführung in die Informatik IV

Skript zur Vorlesung

erstellt von Steffen Manthey und Stephan Micklitz
zur Vorlesung von Prof. Dr. Steger im Sommersemester 2000

überarbeitet von Stephan Micklitz
für die Vorlesung von Prof. Dr. Brauer im Sommersemester 2001

Version: 3.03

30. Juli 2001

Inhaltsverzeichnis

Kapitel 1

Formale Sprachen und Automaten

Sei Σ ein Alphabet (geordneter Zeichenvorrat). Eine formale Sprache ist eine (beliebige) Teilmenge von Σ^*. Unser Ziel ist es nun eine möglichst einfache bzw. kurze Beschreibung für eine formale Sprache zu finden.

Beispiel 1.1

$$
\begin{array}{rcl}
<\text{Satz}> & \to & <\text{Subjekt}> <\text{Prädikat}> <\text{Objekt}> \\
<\text{Subjekt}> & \to & <\text{Artikel}> <\text{Attribut}> <\text{Substantiv}> \\
<\text{Artikel}> & \to & \varepsilon \\
<\text{Artikel}> & \to & \textsf{der} \\
<\text{Artikel}> & \to & \textsf{die} \\
<\text{Artikel}> & \to & \textsf{das} \\
<\text{Attribut}> & \to & \varepsilon \\
<\text{Attribut}> & \to & <\text{Adjektiv}> \\
<\text{Attribut}> & \to & <\text{Adjektiv}> <\text{Attribut}> \\
<\text{Adjektiv}> & \to & \textsf{klein} \\
<\text{Adjektiv}> & \to & \textsf{groß} \\
\text{u.s.w.} & &
\end{array}
$$

Beispiel 1.2

$$
\begin{aligned}
L_1 &= \{\ \textsf{aa},\ \textsf{aaaa},\ \textsf{aaaaaa},\ \textsf{aaaaaaaa},\ \textsf{...}\} = \{\ (\textsf{aa})^n \mid n \in \mathbb{N}\} \\
L_2 &= \{\ \textsf{ab},\ \textsf{abab},\ \textsf{ababab},\ \textsf{abababab},\ \textsf{...}\} = \{\ (\textsf{ab})^n \mid n \in \mathbb{N}\} \\
L_3 &= \{\ \textsf{ab},\ \textsf{aabb},\ \textsf{aaabbb},\ \textsf{aaaabbbb},\ \textsf{...}\} = \{\ \textsf{a}^n\textsf{b}^n \mid n \in \mathbb{N}\} \\
L_4 &= \{\ \textsf{a},\ \textsf{b},\ \textsf{aa},\ \textsf{ab},\ \textsf{bb},\ \textsf{aaa},\ \textsf{aab},\ \textsf{...}\} = \{\ \textsf{a}^n\textsf{b}^m \mid n, m \in \mathbb{N}\}
\end{aligned}
$$

1.1 Chomsky Hierarchie

1.1.1 Chomsky Grammatik

Definition 1.1 *Eine Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$ das folgende Bedingungen erfüllt:*

- *V ist eine endliche Menge, die Menge der Variablen.*

- *Σ ist eine endliche Menge, das Terminalalphabet, wobei $V \cap \Sigma = \emptyset$.*

- *P ist die Menge der Produktionen oder Regeln. P ist eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. (Schreibweise: $(u, v) \in P$ schreibt man meist als $u \to v$.)*
- *$S \in V$ ist die Startvariable.*

Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G unmittelbar in v über), falls u und v die folgende Form haben:

- $u = xyz$
- $v = xy'z$ mit $x, z \in (V \cup \Sigma)^*$ und
- $y \to y'$ eine Regel in P ist.

Falls klar ist, welche Grammatik gemeint ist, so schreiben wir oft auch einfach kurz $u \Rightarrow v$ anstelle von $u \Rightarrow_G v$.

Beispiel 1.3
Gegeben sei die Sprache L_1 mit den Produktionen $S \to \mathsf{aa}$ und $S \to \mathsf{aa}S$. Dann ist a^{10} ein Wort dieser Sprache ($\mathsf{a}^{10} \in L_1$), denn:

$$S \Rightarrow \mathsf{aa}S \Rightarrow \mathsf{aaaa}S \Rightarrow \mathsf{aaaaaa}S \Rightarrow \mathsf{aaaaaaaa}S \Rightarrow \mathsf{aaaaaaaaaa} = \mathsf{a}^{10}$$

Definition 1.2 *Die von G definierte (erzeugte, dargestellte) Sprache ist $L(G) := \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$ wobei $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$ ist.*

Eine Folge von Worten $(w_0, w_1, ..., w_n)$ mit $w_0 = S$, $w_n \in \Sigma^*$ und $w_i \Rightarrow w_i + 1$ für $i = 0, ..., n$ heißt Ableitung von w_n

Beispiel 1.4

$$\begin{array}{ll} L_2: & S \to \mathsf{ab},\ S \to \mathsf{ab}S \\ L_4: & S \to A,\ S \to B,\ S \to AB, \\ & A \to \mathsf{a}A,\ A \to \mathsf{a}, \\ & B \to \mathsf{b}B,\ B \to \mathsf{b}, \end{array}$$

1.1.2 Chomsky Hierarchie

Definition 1.3 *Eine Chomsky-Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt: Die endliche Menge V ist die Menge der Variablen, Σ ist eine endliche Menge, das Terminalalphabet, mit $V \cap \Sigma = \emptyset$, und $S \in V$ ist die Startvariable. Die Menge P ist eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ und wird die Regel- oder Produktionenmenge genannt.*

1. *Jede Chomsky-Grammatik ist zunächst automatisch vom Typ 0.*
2. *Eine Chomsky-Grammatik ist monoton, falls für alle Regeln $\alpha \to \beta$ in P mit $\alpha \neq S$ gilt, daß $|\alpha| \leq |\beta|$. Falls $S \to \epsilon$ in P gilt, dann kommt S auf keiner rechten Seite einer Produktion vor.*
3. *Eine Chomsky-Grammatik ist vom Typ 1 oder kontextsensitiv, falls alle Regeln $\alpha \to \beta$ in P mit $\alpha \neq S$ gilt, daß $\alpha = \alpha' A \alpha''$ mit $\alpha', \alpha'' \in (V \cup \Sigma)^*$, $A \in V$ und $\beta = \alpha' \beta' \alpha''$ mit $\beta' \in (V \cup \Sigma)^+$. Falls $S \to \epsilon$ in P gilt, dann kommt S auf keiner rechten Seite einer Produktion vor.*
4. *Eine Chomsky-Grammatik ist vom Typ 2 oder kontextfrei, falls für alle Regeln $\alpha \to \beta$ in P gilt, daß $\alpha \in V$ gilt.*

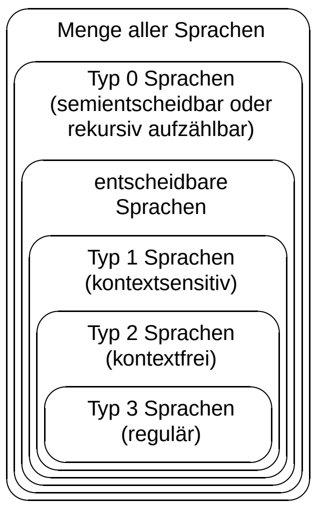
5. *Eine Chomsky-Grammatik ist streng kontextfrei, falls für alle Regeln $\alpha \to \beta$ in P mit $\alpha \neq S$ gilt, daß $\alpha \in V$ und $\beta \in (V \cup \Sigma)^+$. Falls $S \to \epsilon$ in P gilt, dann kommt S auf keiner rechten Seite einer Produktion vor.*

6. *Eine Chomsky-Grammatik ist vom Typ 3, rechtslinear oder regulär, falls für alle Regeln $\alpha \to \beta$ in P gilt, daß $\alpha \in V$ und $\beta \in \Sigma^* \cup \Sigma^* V$.*

7. *Eine Chomsky-Grammatik ist streng regulär, falls für alle Regeln $\alpha \to \beta$ in P mit $\alpha \neq S$ gilt, daß $\alpha \in V$ und $\beta \in \Sigma^+ \cup \Sigma^* V$. Falls $S \to \epsilon$ in P gilt, dann kommt S auf keiner rechten Seite einer Produktion vor.*

Definition 1.4 *Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ i, $i \in \{0, 1, 2, 3\}$, falls es eine Grammatik vom Typ i gibt mit $L(G) = L$.*

ε-Sonderregelung: Wegen $|u| \leq |v|$ kann das leere Wort bei Typ 1,2,3 Grammatiken nicht erzeugt werden. Wir erlauben daher die folgende Sonderregelung: Ist $\varepsilon \in L(G)$ erwünscht, so ist die Regel $S \to \varepsilon$ zugelassen, falls die Startvariable S auf keiner rechten Seite einer Produktion vorkommt.



Lemma 1.1 *Sei G eine Grammatik mit $\forall\,(u \to v) \in P$: $u \in V$. Dann ist $L(G)$ kontextfrei.*

Beweis: Zu zeigen ist, dass eine Grammatik G' existiert mit $L(G') = L(G)$, wobei G' vom Chomsky-Typ 2 ist. Wir erzeugen nun $G' = (V \dot\cup T, P', \Sigma, S)$ aus $G = (V, P, \Sigma, S)$ wie folgt:

1. Ziel ist zunächst, alle Produktionen der Art u $\to \varepsilon$ mit $u \in V$ zu eliminieren, solche Produktionen wollen wir als ε-Produktionen bezeichnen.

   Für jede Variable A bestimmen wir zunächst, ob $A \Rightarrow^* \varepsilon$. Ist dies der Fall, so nennen wir A *nullierbar*. Wir bestimmen alle nullierbaren Variablen wie folgt: Zu Beginn gilt, A ist nullierbar, wenn $A \to \varepsilon$ eine Produktion in P ist. Wenn nun $B \to \alpha$ eine Produktion ist und alle Symbole von α als nullierbar eingestuft worden sind, dann ist auch B nullierbar. Diesen Vorgang wiederholen wir, bis keine weiteren nullierbaren Symbole mehr zu finden sind.

   Wir konstruieren nun die Menge P' der Produktionen von G' wie folgt: Wenn $A \to X_1 X_2 \ldots X_n$ in P ist, dann sind alle Produktionen $A \to \alpha_1 \alpha_2 \ldots \alpha_n$ zu P' hinzuzufügen, wobei folgendes gilt (mit $i \in \{1, \ldots, n\}$):

   (a) wenn X_i nicht nullierbar ist, dann gilt $\alpha_i = X_i$
   (b) wenn X_i nullierbar ist, dann ist α_i entweder X_i oder ε
   (c) nicht alle α_i sind gleich ε.

2. Wir sorgen nun dafür, dass S auf keiner rechten Seite vorkommt. Wir fügen zu P' die Produktion $S \to T$ hinzu, und ersetzen alle S auf der rechten Seite durch T. Dann wird für jede Produktion $S \to u, u \neq \varepsilon, u \in \Sigma \cup V$ eine Produktion $T \to u$ zu P' hinzugefügt.

□

Lemma 1.2 *Lemma 1.1 gilt analog auch für reguläre Sprachen:*
Sei G eine Grammatik mit $\forall\ (u \to v) \in P$: $u \in V \wedge v \in \Sigma \cup \Sigma V$. Dann ist $L(G)$ regulär.

Beispiel 1.5
Im folgenden geben wir ein paar Beispiele für Grammatiken:

Typ 3: $L = \{a^n \mid n \in \mathbb{N}\}$,
Grammatik: $S \to \mathsf{a}$, $S \to \mathsf{a}S$

Typ 2: $L = \{a^n b^n \mid n \in \mathbb{N}\}$,
Grammatik: $S \to \mathsf{ab}$, $S \to \mathsf{a}S\mathsf{b}$

Typ 1: $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$,
Grammatik: $S \to \mathsf{a}SXY$, $S \to \mathsf{a}XY$, $YX \to XY$,
$\mathsf{a}X \to \mathsf{ab}$, $\mathsf{b}X \to \mathsf{bb}$, $\mathsf{b}Y \to \mathsf{bc}$, $\mathsf{c}Y \to \mathsf{cc}$

Backus-Naur-Form: Die Backus-Naur-Form (BNF) ist ein Formalismus zur kompakten Darstellung von Typ 2 Grammatiken.

- Statt $A \to \beta_1,\ A \to \beta_2,\ \ldots\ , A \to \beta_n$ schreibt man $A \to \beta_1|\beta_2|\ldots|\beta_n$.
- Statt $A \to \alpha\gamma,\ A \to \alpha\beta\gamma$ schreibt man $A\ \to\ \alpha[\beta]\gamma$.

  (D.h., das Wort β kann, muß aber nicht, zwischen α und γ eingefügt werden.)
- Statt $A \to \alpha\gamma,\ A \to \alpha B\gamma,\ B \to \beta,\ B \to \beta B$ schreibt man $A\ \to\ \alpha\{\beta\}\gamma$.

  (D.h., das Wort β kann beliebig oft (auch Null mal) zwischen α und γ eingefügt werden.)

1.1.3 Wortproblem

Beispiel 1.6
Arithmetische Ausdrücke:

<exp> = <term>
<exp> = <exp> + <term>
<term> = (<exp>)
<term> = <term> * <term>
<term> = a | b | ... | z

Die Aufgabe eines Compilers ist es, zu prüfen ob ein gegebener String einen gültigen arithmetischen Ausdruck darstellt und, falls ja, ihn in seine Bestandteile zu zerlegen. Abstrakt kann man diese Probleme wie folgt formulieren:

1. *Wortproblem:* Für eine gegebene Grammatik $G = (V, \Sigma, P, S)$ will man feststellen, ob für das Wort $w \in \Sigma^*$ gilt, dass $w \in L(G)$.

2. *Ableitungsproblem:* Für eine gegebene Grammatik $G = (V, \Sigma, P, S)$ und ein gegebenes Wort $w \in L(G)$ will man eine Ableitung von w konstruieren. D.h., Worte $w_0, w_1, ..., w_n \in (\Sigma \cup V)^*$ mit $w_0 = S$, $w_n = w$ und $w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_n$.

Satz 1.1 *Für kontextsensitive Sprachen ist das Wortproblem entscheidbar. Genauer: Es gibt einen Algorithmus, der bei Eingabe einer kontextsensitiven Grammatik G und eines Wortes w in endlicher Zeit entscheidet, ob $w \in L(G)$.*

Beweisidee: Angenommen $w \in L(G)$. Dann gibt es $w_0, w_1, ..., w_n \in (\Sigma \cup V)^*$ mit $w_0 = S$ und $w_n = w$ und $w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_n$.

Wichtig: Da G kontextsensitiv ist, gilt $|w_0| \leq |w_1| \leq ... \leq |w_n|$. D.h., es genügt alle Worte in $L(G)$ der Länge $\leq n$ zu erzeugen.

Beweis: Definiere

$$\begin{aligned} T_m^n \quad := \quad & \{w \in (\Sigma \cup V)^* \mid |w| \leq n \\ & \text{ und } w \text{ lässt sich aus } S \text{ in höchstens } m \text{ Schritten herleiten. } \} \end{aligned}$$

Diese Mengen kann man für alle n induktiv wie folgt berechnen:

$$\begin{aligned} T_0^n \quad &:= \quad \{S\} \\ T_{m+1}^n \quad &:= \quad T_m^n \cup \left\{(w \in (\Sigma \cup V)^* \mid |w| \leq n \text{ und } w' \Rightarrow w \text{ für ein } w' \in T_m^n\right\} \end{aligned}$$

Für alle m gilt:

$$|T_m^n| \leq \sum_{i=1}^{n} |\Sigma \cup V|^i$$

Es muss daher immer ein m_0 geben mit $T_{m_0}^n = T_{m_0+1}^n = ...$ □

Algorithmus:

$n := |w|$;
$T_0^n := \{S\}$;
$m := 0$;
repeat
  $T_{m+1}^n :=$<wie oben>;
  $m := m + 1$;
until $T_m^n = T_{m-1}^n$ or $w \in T_m^n$;
if $w \in T_m^n$ **then return** ("Ja") **else return** ("Nein").

Beispiel 1.7
Gegeben sei eine Grammatik mit den Produktionen $S \rightarrow$ ab und $S \rightarrow$ aSb sowie das Wort $w =$ abab.

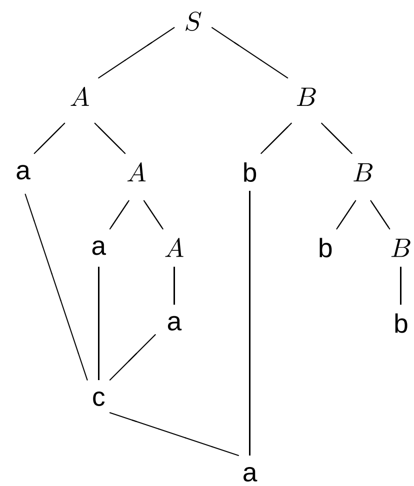
$T_0^4 = \{ S \}$
$T_1^4 = \{ S, \text{ ab}, \text{ a}S\text{b} \}$
$T_2^4 = \{ S, \text{ ab}, \text{ a}S\text{b}, \text{ aabb} \}$ Die Ableitung aa$S$bb besitzt bereits Länge 5.
$T_3^4 = \{ S, \text{ ab}, \text{ a}S\text{b}, \text{ aabb} \}$

Wie unschwer zu erkennen ist lautet die Antwort darauf, ob sich w mit der gegebenen Grammatik erzeugen lässt "Nein".

Wichtig: Das vorgestellte Verfahren ist nicht effizient! Für kontextfreie Sprachen geht es wesentlich effizienter (mehr dazu später).

1.1.4 Ableitungsbäume und -graphen

Beispiel 1.8
Gegeben sei die folgende Grammatik: $S \rightarrow AB, A \rightarrow \mathsf{a}A, A \rightarrow \mathsf{a}, B \rightarrow \mathsf{b}B, B \rightarrow \mathsf{b}, \mathsf{aaa} \rightarrow \mathsf{c}, \mathsf{cb} \rightarrow \mathsf{a}$.



Die Terminale ohne Kante nach unten entsprechen, von links nach rechts gelesen, dem durch den Ableitungsgraphen dargestellten Wort. In obigem Beispiel gilt also $abb \in L(G)$. Der Ableitungsgraph entspricht der Ableitung:

$$S \Rightarrow AB \Rightarrow \mathsf{a}AB \Rightarrow \mathsf{a}A\mathsf{b}B \Rightarrow \mathsf{aa}A\mathsf{b}B \Rightarrow \mathsf{aa}A\mathsf{bb}B \Rightarrow \mathsf{aaabb}B \Rightarrow \mathsf{aaabbb} \Rightarrow \mathsf{cbbb} \Rightarrow \mathsf{abb}$$

Bei kontextfreien Sprachen sind die Ableitungsgraphen immer Bäume. Diese werden dann auch als Ableitungsbäume bezeichnet.

Beispiel 1.9
Gegeben sei die folgende Grammatik: $S \rightarrow \mathsf{a}B, S \rightarrow A\mathsf{c}, A \rightarrow \mathsf{ab}, B \rightarrow \mathsf{bc}$. Für das Wort abc gibt es zwei verschiedene Ableitungsbäume:

Definition 1.5 *Eine Ableitung $S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$ eines Wortes w_n heißt Linksableitung, wenn für jede Anwendung einer Produktion $y \rightarrow y'$ auf Worte $w_i = xyz, w_{i+1} = xy'z$ gilt: Auf jedes echte Präfix von xy lässt sich keine Regel anwenden.*

Eine *kontextfreie* Grammatik wird als *eindeutig* bezeichnet wenn es für jedes Wort $w \in L(G)$ genau eine eindeutige bestimmte Linksableitung gibt. Nicht eindeutige Grammatiken nennt man auch *mehrdeutig*. Gleichermaßen heisst eine Sprache L eindeutig, wenn es eine eindeutige Grammatik G mit $L = L(G)$ gibt.

1.2 Reguläre Sprachen

1.2.1 Deterministische endliche Automaten

Definition 1.6 *Ein deterministischer endlicher Automat (deterministic finite (state) automaton, kurz DFA) wird durch ein 5-Tupel $M = (Z, \Sigma, \delta, z_0, E)$ beschrieben, das folgende Bedingungen erfüllt:*

- *Z ist eine endliche Menge von Zuständen.*
- *Σ ist eine endliche Menge, das Eingabealphabet, wobei $Z \cap \Sigma = \emptyset$.*
- *$z_0 \in Z$ ist der Startzustand.*
- *$E \subseteq Z$ ist die Menge der Endzustände.*
- *$\delta : Z \times \Sigma \to Z$ heißt Transistionsfunktion (bzw. Übergangsfunktion).*

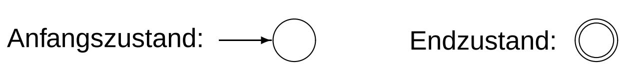
Für die von M akzeptierte Sprache gilt $L(M) := \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$. Sei ferner $\hat{\delta} : Z \times \Sigma^* \to Z$ induktiv definiert durch:

- $\hat{\delta}(z, \varepsilon) = z$
- $\hat{\delta}(z, \mathsf{a}x) = \hat{\delta}(\delta(z, \mathsf{a}), x)$ für $a \in \Sigma,\ x \in \Sigma^*$

Endliche Automaten können durch (gerichtete und beschriftete) Zustandsgraphen veranschaulicht werden:

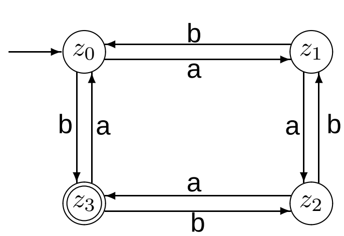
- Knoten $\widehat{=}$ Zustände, symbolisiert durch ◯
- gerichtete Kanten $\widehat{=}$ Übergänge, symbolisiert durch $\longrightarrow$
- genauer: Kante (u, v), beschriftet mit $a \in \Sigma$, entspricht $\delta(u, \mathrm{a}) = v$

Anfangszustände werden durch "eingehende" Pfeile hervorgehoben, Endzustände werden durch doppelte Kreise gekennzeichnet.



Beispiel 1.10
Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein endlicher deterministischer Automat, wobei $Z = \{z_0, z_1, z_2, z_3\}$, $\Sigma = \{a, b\}$, $E = \{z_3\}$ und folgenden Zustandsübergängen:

$$\begin{array}{llll} \delta(z_0, a) = z_1 & \delta(z_1, a) = z_2 & \delta(z_2, a) = z_3 & \delta(z_3, a) = z_0 \\ \delta(z_0, b) = z_3 & \delta(z_1, b) = z_0 & \delta(z_2, b) = z_1 & \delta(z_3, b) = z_2 \end{array}$$



Satz 1.2 *Zu jedem endlichen deterministischen Automat M existiert eine reguläre Grammatik G mit $L(G) = L(M)$.*

Beweisidee, Teil 1: Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein endlicher deterministischer Automat (DFA). Setze $G = (V, \Sigma, P, S)$ mit $V := Z$, $S := z_0$, $P := \{z \to \mathsf{a}\delta(z, \mathsf{a}) \mid z \in Z, \mathsf{a} \in \Sigma\}$. Die Beweisidee ist hier noch nicht ganz vollständig beschrieben, betrachten wir hierzu zunächst ein Beispiel:

Beispiel 1.11
Sei M der Automat aus dem Beispiel von oben und $w =$ abaaa ein Wort der von M erzeugten Sprache $L(M)$. Wir versuchen nun dieses Wort nach der obigen Beweisidee, durch eine Reihe von Produktionen herzuleiten.

$$S = z_0 \Rightarrow \mathsf{a}z_1 \Rightarrow \mathsf{ab}z_0 \Rightarrow \mathsf{aba}z_1 \Rightarrow \mathsf{abaa}z_2 \Rightarrow \mathsf{abaaa}z_3$$

Zwar ist z_3 ein Endzustand, aber es ist offensichtlich, dass weitere Produktionen notwendig sind, um das z_3 zu eliminieren. Damit kommen wir zum Teil 2 der Beweisidee.
Beweisidee, Teil 2: Diese Produktionen erhält man wie folgt:

$$\{z \to \mathsf{a} \mid z \in Z, \mathsf{a} \in \Sigma \wedge \delta(z, \mathsf{a}) \in E\}$$

Beweis: mit den eben aufgestellten Regeln gilt für $a_1, a_2, ..., a_n \in \Sigma$:

$$\begin{array}{ll} & a_1 a_2 ... a_n \in L(M) \\ \Leftrightarrow & \exists \quad z_0, z_1, ... z_n \in Z : \ z_0 \text{ Startzustand des Automaten} \\ & \forall \quad i = 1, ..., (n-1) : \quad \delta(z_i, a_i) = z_{i+1}, \quad z_n \in E \\ \Leftrightarrow & \exists \quad z_0, z_1, ... z_n \in V : \ z_0 \text{ Startvariable der Grammatik} \\ & z_0 \Rightarrow a_1 z_1 \Rightarrow a_1 a_2 z_2 \Rightarrow ... \Rightarrow a_1 ... a_{n-1} z_{n-1} \Rightarrow a_1 ... a_{n-1} a_n \\ \Leftrightarrow & a_1 a_2 ... a_n \in L(G) \end{array}$$

□

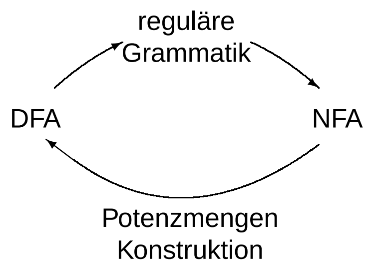
1.2.2 Nichtdeterministische endliche Automaten

Definition 1.7 *Ein nichtdeterministischer endlicher Automat (nondeterministic finite (state) automaton, kurz NFA) wird durch ein 5-Tupel $M = (Z, \Sigma, \delta, S, E)$ beschrieben, das folgende Bedingungen erfüllt:*

- Z *ist eine endliche Menge, die Zustände.*
- Σ *ist eine endliche Menge, das Eingabealphabet, wobei $Z \cap \Sigma = \emptyset$.*
- $S \subseteq Z$ *ist die Menge der Startzustände.*
- $E \subseteq Z$ *ist die Menge der Endzustände.*
- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Z)$ *heißt Transitionsfunktion (bzw. Übergangsfunktion).*

Die von M akzeptierte Sprache ist $L(M) := \{w \in \Sigma^* \mid \hat{\delta}(S, w) \cap E \neq \emptyset\}$. Sei ferner $\hat{\delta} : \mathcal{P}(Z) \times \Sigma^* \to \mathcal{P}(Z)$ induktiv definiert durch:

- $\forall z \in Z : \ \hat{\delta}(z, \varepsilon) = \{z\} \cup \delta(z, \varepsilon)$
- $\hat{\delta}(Z', \mathsf{a}x) = \bigcup\limits_{z \in Z'} \hat{\delta}(\delta(z, \mathsf{a}), x)$ für $Z' \subseteq Z, a \in \Sigma, x \in \Sigma^*$
  speziell gilt: $\hat{\delta}(z, a) = \hat{\delta}(\delta(z, a), \varepsilon)$ für $z \in Z, a \in \Sigma$



Satz 1.3 *Jede von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache ist auch von einem deterministischen endlichen Automaten akzeptierbar.*

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

Unsere Idee ist es nun, einen DFA zu konstruieren, bei dem jede Teilmenge von Z ein Zustand ist. Für diesen DFA gilt:

- $M^\# = (Z^\#, \Sigma, \delta^\#, z_0^\#, E^\#)$
- $Z^\# = \mathcal{P}(Z)$ (Potenzmenge von Z)
- $\delta^\#(Z', a) = \bigcup\limits_{z' \in Z'} \hat{\delta}(z', a) \qquad Z' \subseteq Z, a \in \Sigma$
- $z_0^\# = S$
- $E^\# = \{Z' \subseteq Z \mid Z' \cap E \neq \emptyset\}$

Beachte:

$$
\begin{array}{ll}
 & a_1, ..., a_n \in \Sigma \qquad a_1 ... a_n \in L(M) \\
\Leftrightarrow & \exists \quad Z_1, ..., Z_n \subseteq Z \\
 & \delta^\#(S, a_1) = Z_1, \quad \delta^\#(Z_1, a_2) = Z_2, \quad ... \quad , \delta^\#(Z_{n-1}, a_n) = Z_n \qquad Z_n \cap E \neq \emptyset \\
\Leftrightarrow & a_1 .... a_n \in L(M^\#)
\end{array}
$$

Beachte:: Viele Zustände sind i.a. überflüssig, im "schlimmsten" Fall werden aber alle $2^{|Z|}$ Zustände im deterministischen Automaten tatsächlich benötigt.

□

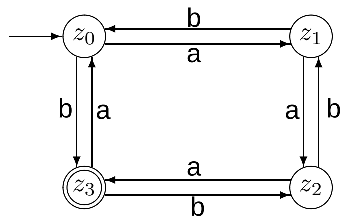
Satz 1.4 *Ist $M = (Z, \Sigma, \delta, z_0, E)$ ein deterministischer endlicher Automat, so ist die durch*

$$P \quad := \quad \{z \to \mathsf{a}z' \mid \delta(z, \mathsf{a}) = z'\} \cup \{z \to \mathsf{a} \mid \delta(z, \mathsf{a}) \in E\}$$

gegebene Grammatik $G = (Z, \Sigma, z_0, P)$ regulär.

Beispiel 1.12
Gegeben sei eine Grammatik mit den folgenden Produktionen:

- $z_0 \to \mathsf{a}z_1$, $z_0 \to \mathsf{b}z_3$, $z_0 \to \mathsf{b}$, $z_1 \to \mathsf{a}z_2$, $z_1 \to \mathsf{b}z_0$,
- $z_2 \to \mathsf{a}z_3$, $z_2 \to \mathsf{b}z_1$, $z_2 \to \mathsf{a}$, $z_3 \to \mathsf{a}z_0$, $z_3 \to \mathsf{b}z_2$



Ein Beispiel für ein Wort der erzeugten Sprache ist $z_0 \Rightarrow \mathsf{a}z_1 \Rightarrow \mathsf{ab}z_0 \Rightarrow \mathsf{aba}z_1 \Rightarrow \mathsf{abaa}z_2 \Rightarrow \mathsf{abaaa} \in L(G)$.

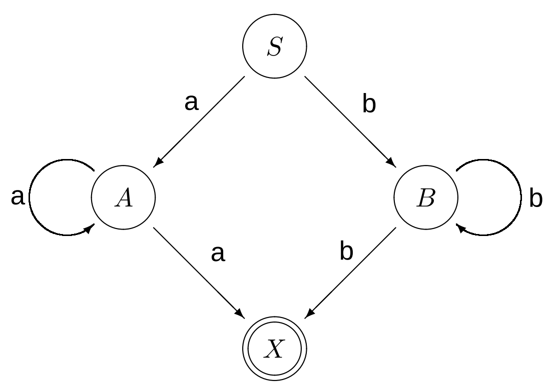
Satz 1.5 *Ist $G = (V, \Sigma, S, P)$ eine reguläre Grammatik, so ist $M = (V \cup \{X\}, \Sigma, \delta, S, E)$ ein nichtdeterministischer endlicher Automat, wobei*

$$E \;:=\; \begin{cases} \{S, X\}, & \text{falls } S \to \varepsilon \in P \\ \{X\}, & \text{sonst} \end{cases}$$

$$\begin{aligned} B \in \delta(A, a) \quad &\Leftrightarrow \quad A \to aB \quad \textit{und} \\ X \in \delta(A, a) \quad &\Leftrightarrow \quad A \to a \end{aligned}$$

Beispiel 1.13
Gegeben sei eine Grammatik mit den folgenden Produktionen:

- $S \to \mathrm{a}A$, $S \to \mathrm{b}B$, $A \to \mathrm{a}$, $A \to \mathrm{a}A$, $B \to \mathrm{b}$, $B \to \mathrm{b}B$



1.2.3 Reguläre Ausdrücke

Reguläre Ausdrücke sollen eine kompakte Notation für spezielle Sprachen darstellen, endliche Ausdrücke sollen hierbei auch unendliche Mengen beschreiben. Welche Sprachen mit regulären Ausdrücken tatsächlich beschrieben werden können, werden wir im Laufe dieses Kapitels feststellen.

Definition 1.8 *Reguläre Ausdrücke sind induktiv definiert durch:*

- *$\emptyset$ ist ein regulärer Ausdruck.*

- *ε ist ein regulärer Ausdruck.*

- *Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.*

- *Wenn α und β reguläre Ausdrücke sind, dann sind auch $(\alpha\beta)$, $(\alpha|\beta)$ (hierfür kann auch $(\alpha + \beta)$ geschrieben werden) und $(\alpha)^*$ reguläre Ausdrücke.*

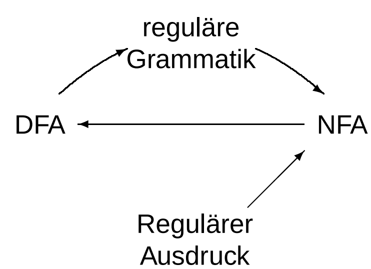
- *Nichts sonst ist ein regulärer Ausdruck.*

Zu einem regulären Ausdruck γ ist eine zugehörige Sprache $L(\gamma)$ induktiv definiert durch:

- Falls $\gamma = \emptyset$, so gilt $L(\gamma) = \emptyset$.
- Falls $\gamma = \varepsilon$, so gilt $L(\gamma) = \{\varepsilon\}$.
- Falls $\gamma = \mathsf{a}$, so gilt $L(\gamma) = \{\mathsf{a}\}$.
- Falls $\gamma = (\alpha\beta)$, so gilt $L(\gamma) = L(\alpha)L(\beta) = \{uv \mid u \in L(\alpha), v \in L(\beta)\}$
- Falls $\gamma = (\alpha|\beta)$, so gilt $L(\gamma) = L(\alpha) \cup L(\beta) = \{u \mid u \in L(\alpha) \vee u \in L(\beta)\}$
- Falls $\gamma = (\alpha)^*$, so gilt $L(\gamma) = L(\alpha)^* = \{u_1 \dots u_n \mid n \in N_0, u_1, \dots, u_n \in L(\alpha)\}$

Beispiel 1.14

- Alle Worte, die gleich 0 sind oder mit 00 enden. (0 | (0|1)*00)
- Alle Worte, die 0110 enthalten. ((0|1)*0110(0|1)*)
- Alle 0-1 Werte, die die Binärdarstellung einer durch 3 teilbaren Zahl darstellen. Beispiel: 3 $\hat{=}$ 11, 6 $\hat{=}$ 110, 9 $\hat{=}$ 1001, ... Regulärer Ausdruck: Übungsaufgabe



Satz 1.6 $L \subseteq \Sigma^*$ *durch einen regulären Ausdruck beschreibbar* $\Leftrightarrow$ L *regulär.*

Beweis: "$\Rightarrow$"-Richtung
Sei also $L = L(\gamma)$. Wir zeigen nun, dass ein NFA M existiert mit $L = L(M)$.

Induktionsanfang:

Gilt $\gamma = \emptyset$, $\gamma = \varepsilon$ oder $\gamma = \mathsf{a}$, so folgt sofort: Ein NFA M mit $L = L(M)$ existiert.

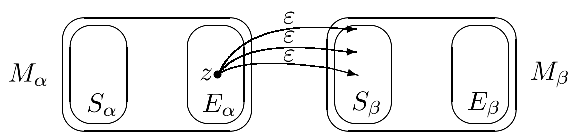
Induktionsschritt:

$\gamma = (\alpha\beta)$: Nach Induktionsannahme existiert ein NFA M_α und ein NFA M_β mit $L(M_\alpha) = L(\alpha)$ und $L(M_\beta) = L(\beta)$.

Unsere Idee ist es nun die beiden Automaten "in Serie" zu schalten. Für alle Endzustände $z \in E_\alpha$ (von M_α) muss dabei ein ε-Übergang nach S_β (von M_β) eingefügt werden. Es entsteht ein NFS M_γ, der wie folgt definiert ist:

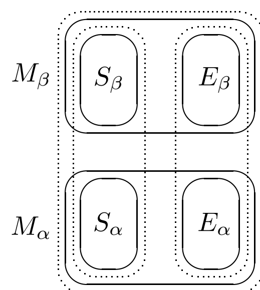
- $M_\gamma = (Z_\alpha \cup Z_\beta, \Sigma, \delta_\gamma, S_\alpha, E_\beta)$

- $\delta_\gamma = \delta_\alpha \cup \delta_\beta \cup \{\, (z, \varepsilon) \to z' \mid z \in E_\alpha, z' \in S_\beta \,\}$

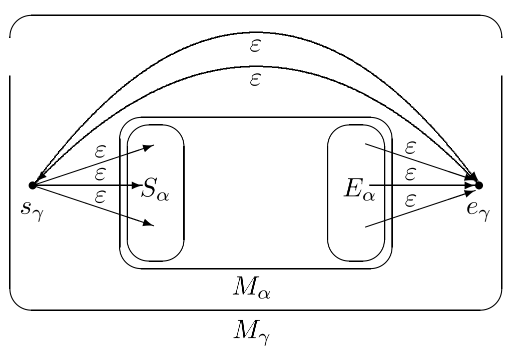


$\gamma = (\alpha|\beta)$: Seien M_α und M_β definiert wie eben. Unsere Idee ist es nun einen neuen NFA M_γ als Vereinigung von M_α und M_β zu konstruieren. Dieser NFA ist wie folgt definiert:

- $M_\gamma = (Z_\gamma, \Sigma, \delta_\gamma, S_\gamma, E_\gamma)$
- $Z_\gamma = Z_\beta \cup Z_\alpha$
- $E_\gamma = E_\beta \cup E_\alpha$
- $S_\gamma = S_\beta \cup S_\alpha$



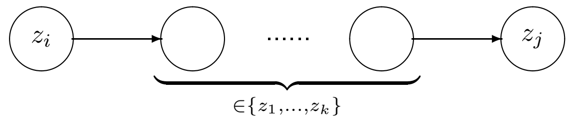
$\gamma = (\alpha)^*$: Auch für diesen Fall existiert ein NFA M_γ. Diesen NFA $M_\gamma = (Z_\alpha \dot\cup\ s_\gamma \dot\cup\ e_\gamma, \Sigma, \delta_\alpha, s_\gamma, e_\gamma)$ mit dem neuen Startzustand s_γ und dem neuen Endzustand e_γ konstruieren wir wie folgt:



Es ist zu beachten, dass $\varepsilon \in L((\alpha)^*)$. Da allerdings nicht unbedingt $\varepsilon \in L(\alpha)$ gilt, muss s_γ über eine ε Kante mit e_γ verbunden werden.

Beweis: "$\Leftarrow$"-Richtung:
Sei $M = (Z, \Sigma, \delta, z_1, E)$ ein deterministischer endlicher Automat. Wir zeigen nun, dass es einen regulären Ausdruck γ gibt mit $L(M) = L(\gamma)$. Ohne Einschränkung sei $Z = \{z_1, ..., z_n\}$. Wir setzen

$R_{ij}^k \;:=\; \{x \in \Sigma^* \mid$ die Eingabe x überführt den im Zustand z_i gestarteten Automaten in den Zustand z_j, wobei alle zwischendurch durchlaufenen Zustände einen Index kleiner gleich k haben$\}$



Behauptung: Für alle $j \in \{1,...,n\}$ und alle $k \in \{1,...,n\}$ gilt: Es gibt einen regulären Ausdruck α_{ij}^k mit $L(\alpha_{ij}^k) = R_{ij}^k$.

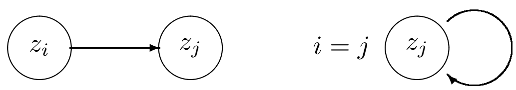
Der Beweis erfolgt nun durch Induktion über k:

Induktionsanfang:

$k = 0$: Hier gilt

$$R_{ij}^0 \quad := \quad \begin{cases} \{a|\delta(z_i,a) = z_j\} & i \neq j \\ \{a|\delta(z_i,a) = z_j\} \cup \{\varepsilon\} & i = j \end{cases}$$

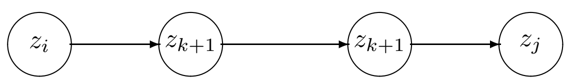
D.h., R_{ij}^0 ist endlich und läßt sich daher durch einen regulären Ausdruck α_{ij}^0 beschreiben.



Induktionsschritt:

$k \Rightarrow k+1$: Hier gilt

$$\begin{aligned} R_{ij}^{k+1} &= R_{ij}^k \cup R_{i(k+1)}^k (R_{(k+1)(k+1)}^k)^* R_{(k+1)j}^k \\ \alpha_{ij}^{k+1} &= \alpha_{ij}^k \mid \alpha_{i(k+1)}^k (\alpha_{k+1,k+1}^k)^* \alpha_{(k+1)j}^k \end{aligned}$$

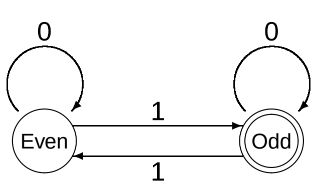


Somit gilt: $L(M) = (\alpha_{s_1,e_1}^n \mid \alpha_{s_1,e_2}^n \mid \ldots \mid \alpha_{s_2,e_1}^n \mid \ldots \mid \alpha_{s_q,e_r}^n)$, wobei $s_1, \ldots, s_q$ die Indizes der Startzustände und $e_1, \ldots, e_r$ die Indizes der Endzustände seien.

Die Algorithmische Berechnung der Ausdrücke entspricht im Prinzip der Dynamischen Programmierung.
Bemerkung: In der Vorlesung von Prof. Brauer wurde eine alternative Methode zur Überführung von endlichen nichtdeterministischen (und deterministischen) Automaten in reguläre Ausdrücke angegeben. Hierzu sei auf die entsprechenden Übungsblätter verwiesen.

$\square$

Beispiel 1.15
Sei L die Menge aller Worte über $\{0,1\}$ die eine ungerade Anzahl von Einsen enthalten. Dazu ist folgender Deterministischer Endlicher Automat gegeben, der die Sprache lesen kann:



Im folgenden stehe z_1 für Even und z_2 für Odd. Wir bauen nun schrittweise die von dem Automaten erzeugte reguläre Sprache auf. Der Automat verfügt zunächst einmal über die folgenden Zustandsübergänge:

$$\begin{array}{lllllll} R_{11}^0 & = & \{\varepsilon, \mathsf{0}\} & \quad & \alpha_{11}^0 & = & (\varepsilon \mid \mathsf{0}) \\ R_{12}^0 & = & \{\mathsf{1}\} & & \alpha_{12}^0 & = & \mathsf{1} \\ R_{21}^0 & = & \{\mathsf{1}\} & & \alpha_{21}^0 & = & \mathsf{1} \\ R_{22}^0 & = & \{\varepsilon, \mathsf{0}\} & & \alpha_{22}^0 & = & (\varepsilon \mid \mathsf{0}) \end{array}$$

Mit dem im obigen Beweis verwendeten Algorithmus konstruieren wir daraus die regulären Ausdrücke, die die von dem Automaten akzeptierte reguläre Sprache beschreiben:

$$\begin{array}{lll} R_{11}^1 & = & R_{11}^0 \cup R_{11}^0 (R_{11}^0)^* R_{11}^0 \\ \alpha_{11}^1 & = & \alpha_{11}^0 \mid \alpha_{11}^0 (\alpha_{11}^0)^* \alpha_{11}^0 \\ & = & (\mathsf{0})^* \end{array}$$

$$\begin{array}{lll} R_{12}^1 & = & R_{12}^0 \cup R_{11}^0 (R_{11}^0)^* R_{12}^0 \\ \alpha_{12}^1 & = & \alpha_{12}^0 \mid \alpha_{11}^0 (\alpha_{11}^0)^* \alpha_{12}^0 \\ & = & (\mathsf{0})^* \mathsf{1} \end{array}$$

$$\begin{array}{lll} \alpha_{21}^1 & = & \mathsf{1}(\mathsf{0})^* \\ \alpha_{22}^1 & = & (\mathsf{1}(\mathsf{0})^*\mathsf{1} \mid \varepsilon \mid \mathsf{0}) \end{array}$$

Nun interessieren uns natürlich die regulären Ausdrücke γ. Für diese gilt $L(\gamma) = L(M)$. Wir zeigen nun das gilt $\gamma = \alpha_{12}^2$

$$\begin{array}{lll} \alpha_{12}^2 & = & \alpha_{12}^1 \mid \alpha_{12}^1 (\alpha_{22}^1)^* \alpha_{22}^1 \\ & = & \alpha_{12}^1 \mid \alpha_{12}^1 (\alpha_{22}^1)^* \\ & = & (\mathsf{0})^*\mathsf{1} \mid (\mathsf{0})^*\mathsf{1}(\mathsf{1}(\mathsf{0})^*\mathsf{1} \mid \varepsilon \mid \mathsf{0})^* \end{array}$$

1.2.4 Pumping Lemma

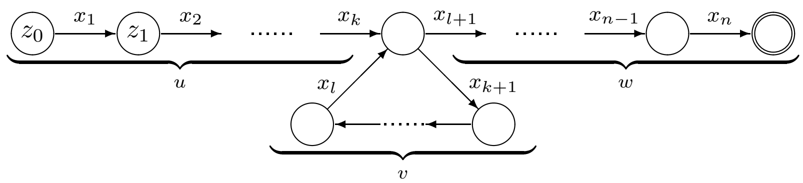
Beispiel 1.16
Betrachten wir die Sprache $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$ hinsichtlich der Fragestellung ob diese Sprache regulär ist. Für diese Sprache lässt sich natürlich sofort die Grammatik: $S \rightarrow \mathsf{ab}, S \rightarrow \mathsf{a}S\mathsf{b}$ angeben. Diese Grammatik ist natürlich nicht regulär. Aber geht es auch anders ... ?

An dieser Stelle kommt das Pumping Lemma ins Spiel. Es ist ein mächtiges Hilfsmittel, um von einer Sprache zu zeigen, dass sie nicht regulär ist.

Satz 1.7 *(Pumping Lemma, uvw-Theorem) Sei L eine reguläre Sprache. Dann gibt es ein $n \in \mathbb{N}$, so dass es für jedes Wort $x \in L$ mit $|x| \geq n$ eine Zerlegung $x = uvw$ gibt, so dass*

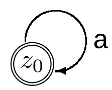
1. $|v| \geq 1$,
2. $|uv| \leq n$,
3. *für alle $i \in \mathbb{N}_0$ gilt:* $uv^iw \in L$.

Beweis: Da L eine reguläre Sprache ist, existiert ein deterministischer endlicher Automat M mit $L(M) = L$. n (gemeint ist das n aus dem Pumping Lemma) ist die Anzahl der Zustände von M. Beim Abarbeiten eines Wortes x, $|x| \geq n$, durchläuft der Automat mindestens $(n + 1)$ Zustände. Damit gibt es mindestens einen Zustand der mindestens zwei Mal durchlaufen wird.



Es ist klar, dass alle Worte uv^iw vom Automaten erkannt werden, also $uv^iw \in L$.

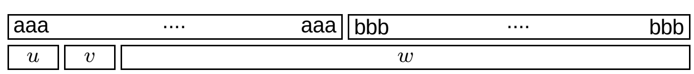
Betrachten wir noch folgenden Spezialfall: Seien $u = \varepsilon$, $w = \varepsilon$ und $v = \mathsf{a}$. In diesem Fall besteht der Automat aus nur einem Knoten und die Schleife nur aus einer Kante.



Es gilt natürlich auch, dass $|uv| \leq n$ (der Automat verfügt ja nur über n Zustände, und demzufolge muss die erste Schleife nach spätestens n Zeichen abgeschlossen sein) und $|v| \geq 1$ (auch wenn die Schleife keinen Knoten enthält, so enthält sie zumindest eine Kante). □

Satz 1.8 *Die Sprache $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$ ist nicht regulär.*

Beweis: Angenommen L wäre eine reguläre Sprache. Sei n die Zahl aus dem Pumping Lemma und betrachten wir das Wort: $x = \mathsf{a}^{3n}\mathsf{b}^{3n} \in L, |x| \geq n$. Hier gibt es folgende Zerlegung $x = uvw$ mit $|uv| \leq n$:



Damit gilt auch, dass u und v nur a's enthalten können. Das heißt, es existiert ein $j \in \mathbb{N}$, so dass $v = a^j$.

Wählen wir zum Beispiel das Wort uv^2w, es enthält $3n + j\mathsf{a}$'s und $3n\mathsf{b}$'s. Damit ist uv^2w kein Wort aus L, da es mehr a's als b's gibt. Dies steht im Wiederspruch zum Pumping Lemma. □

Anmerkung zum Beweis: In obigem Beweis haben wir das Wort $x = \mathsf{a}^{3n}\mathsf{b}^{3n}$ gewählt. Von vornherein ist selbstverständlich nicht klar, dass der Beweis gerade mit diesem Wort gelingen wird – die Wahl von $3n$ im Exponenten war also "nur" ein (sehr guter) "Versuch". Im Nachhinein sehen wir nun sogar, dass wir obigen Beweis auch mit dem Wort $x = \mathsf{a}^n\mathsf{b}^n$ hätten führen können.

1.2.5 Abschlusseigenschaften

Definition 1.9 *Man sagt, eine Menge $\mathcal{L}$ von Sprachen ist abgeschlossen unter der Operation $\circ$ falls gilt: $L_1, L_2 \in \mathcal{L} \Rightarrow L_1 \circ L_2 \in \mathcal{L}$.*

Satz 1.9 *Die Menge der Regulären Sprachen ist abgeschlossen unter :*

- *Vereinigung*
- *Schnitt*
- *Komplement*
- *Produkt*
- *“Stern”*

Beweis: Siehe Beweis zu Satz 1.6 auf Seite 11 und Übungsblatt 2. □

1.2.6 Entscheidbarkeit

Beispiel 1.17
Wie wir bereits wissen ist das Wortproblem für reguläre Sprachen L entscheidbar. Wenn L durch einen deterministischen endlichen Automaten gegeben ist, ist dies sogar in linearer Laufzeit möglich. Allerdings gilt, dass bei der Überführung eines nichtdeterministischen endlichen Automaten mit n Zuständen ein deterministischer endlicher Automat mit $\mathcal{O}(2^n)$ Zuständen entstehen kann, die Zahl der Zustände kann hier also exponentiell wachsen.

Wortproblem: Ist ein Wort x in $L(G)$?
Das Wortproblem ist für alle Sprachen mit einem Chomsky-Typ größer 0 entscheidbar. Allerdings wächst bei Sprachen vom Chomsky-Typ 0 und 1 die Laufzeit exponentiell zur Wortlänge n (Laufzeit $:= O(|\Sigma|^n)$), bei Sprachen vom Chomsky-Typ 2 und 3 gibt es allerdings wesentlich effizientere Algorithmen, mehr dazu später.

Leerheitsproblem: Ist $L(G) = \emptyset$?
Das Leerheitsproblem ist für Sprachen vom Chomsky-Typ 2 und 3 entscheidbar. Für andere Sprachtypen lassen sich Grammatiken konstruieren für die nicht mehr entscheidbar ist, ob die Sprache leer ist.

Endlichkeitsproblem: Ist $|L(G)| \leq \infty$?
Das Endlichkeitsproblem ist für alle regulären Sprachen lösbar.

Korollar 1.1 *Sei n eine zur Sprache L gehörende Zahl aus dem Pumping-Lemma. Dann gilt: $|L| = \infty \Leftrightarrow \exists$ Wort $x \in L$ mit $n \leq |x| < 2n$*

Beweis: Wir zeigen zunächst $\Leftarrow$:

Aus dem Pumping-Lemma folgt, dass gilt: $x = uvw$ für $|x| \geq n$ und $uv^iw \in L$ für alle $i \in \mathbb{N}_0$. Damit erzeuge man ∞ viele Wörter.

Nun wird $\Rightarrow$ gezeigt:

Dass es ein Wort x mit $|x| \geq n$ gibt, ist klar (es gibt ja ∞ viele Wörter). über das Pumping-Lemma lässt sich ein solches Wort auf Länge $< 2n$ reduzieren. $\square$

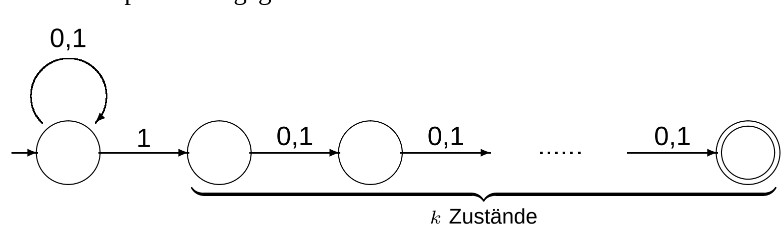
$\Rightarrow$ Damit kann dieses Problem auf das Wortproblem zurückgeführt werden.

Schnittproblem: Ist $L(G_1) \cap L(G_2) = \emptyset$?
Das Schnittproblem ist *nur* für Sprachen vom Chomsky-Typ 3 entscheidbar

Äquivalenzproblem: Ist $L(G_1) = L(G_2)$?
Das Äquivalenzproblem läßt sich auch wie folgt Formulieren: $L_1 = L_2 \Leftrightarrow (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$

Wichtig für eine effiziente Lösung der Probleme ist, wie die Sprache gegeben ist. Hierzu ein Beispiel:

Beispiel 1.18
$L = \{w \in \{0;1\}^* : k\text{-letztes Bit von } w \text{ ist gleich } 1\}$
Ein NFA für diese Sprache ist gegeben durch:



Insgesamt hat der NFA $k+1$ Zustände. Man kann nun diesen NFA in einen deterministischen Automaten umwandeln und stellt fest, dass der entsprechende DFA $\mathcal{O}(2^k)$ Zustände hat.

Da die Komplexizität eines Algorithmus von der Größe der Eingabe abhängt, ist dieser Unterschied in der Eingabegröße natürlich wesentlich: Hat man eine kleine Eingabe wie beim NFA, so hat man "wenig Zeit" für einen effizienten Algorithmus. Ist dagegen bereits die Eingabe groß (wie beim DFA), so bleibt im Verhältnis "mehr Zeit" für einen effizienten Algorithmus.

1.3 Kontextfreie Sprachen

Geklammerte Sprachkonstrukte sind nicht regulär, denn schon die Sprache $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$ ist nicht regulär. Der Beweis erfolgt durch Anwendung des Pumping-Lemmas. Im folgendem sind ein paar Beispiele für kontextfreie Sprachen angegeben:

Beispiel 1.19

- Arithmetische Ausdrücke:

$$\begin{array}{lcl} E & \rightarrow & T|E+T \\ T & \rightarrow & F|T*F \\ F & \rightarrow & a|(E) \end{array}$$

- Klammerpaare in Programmiersprachen:

{ . . . }
begin . . . **end**
repeat . . . **until**
then . . . **end**

Definition 1.10 *Eine Grammatik ist vom Chomsky-Typ 2 oder kontextfrei, wenn für alle Regeln $w_1 \rightarrow w_2$ gilt, dass w_1 eine einzelne Variable ist.*

Beispiel 1.20
Die Sprache $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$ ist demzufolge kontextfrei, da sie sich durch die Grammatik $S \rightarrow \mathsf{ab}|\mathsf{a}S\mathsf{b}$ erzeugt wird.

Die Menge der regulären Sprachen ist also eine echte Untermenge der Menge der kontextfreien Sprachen.

1.3.1 Normalformen

An eine Normalform einer Grammatik stellen wir die folgenden Anforderungen:

- Wir wollen eine möglichst einfache Form für die erlaubten Regeln.
- Jede kontextfreie Sprache soll mit einer Grammatik dieser Form beschrieben werden können.

Der erste Schritt hin zur Normalform besteht darin, an die Grammatik aus der wir die Normalform erzeugen wollen, gewisse Anforderungen zu stellen.

- Die Grammatik ist ε-frei.
- Die Grammatik enthält keine Regeln der Form $A \rightarrow B$, wobei A und B Variablen sind.

Definition 1.11

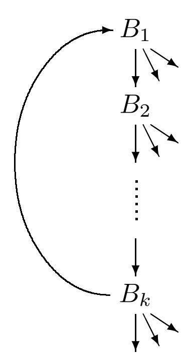
- *Eine Grammtik heißt ε-frei, falls es keine Regel der Form $A \to \varepsilon$ gibt.*
- *Zu jeder kontextfreien Grammatik G mit $\varepsilon \notin L(G)$ gibt es eine* äquivalente *Grammatik G', die ε-frei ist.* Äquivalent *bedeutet: $L(G) = L(G')$*

Um eine Grammatik G in eine äquivalente, ε-freie Grammatik umzuwandeln, verwende man folgendes Vorgehen:

1. Wir teilen die Menge der Variablen von G in zwei disjunkte Teilmengen V_1 und V_2, so dass
$$A \in V_1 \text{ genau dann wenn } A \Rightarrow^* \varepsilon$$
Die Variablen aus V_1 nennen wir *nullierbar*. Wie wir schon im Beweis zum Lemma 1.1 auf Seite 3 gesehen haben, können wir die Menge der Variablen in V_1 wie folgt iterativ bestimmen: Zu Beginn gilt, dass eine Variable A nullierbar ist, wenn $A \to \varepsilon$ eine Produktion aus G ist. Wenn nun $B \to \alpha$ eine Produktion aus G ist und alle Symbole von α als nullierbar eingestuft worden sind, dann ist auch B nullierbar. Diesen Vorgang wiederholen wir, bis keine weiteren nullierbaren Symbole mehr zu finden sind.
2. Wir löschen alle Produktionen der Form $A \to \varepsilon$.
3. Wir konstruieren nun die Menge P' der Produktionen von G' wie folgt: Wenn $A \to X_1X_2 \ldots X_n$ in P ist, dann sind alle Produktionen $A \to \alpha_1\alpha_2 \ldots \alpha_n$ zu P' hinzuzufügen, wobei folgendes gilt (mit $i \in \{1, \ldots, n\}$):
   (a) wenn X_i nicht nullierbar ist, dann gilt $\alpha_i = X_i$
   (b) wenn X_i nullierbar ist, dann ist α_i entweder X_i oder ε
   (c) nicht alle α_i sind gleich ε.

Zur Elimination der Regel der Form $A \to B$, wobei $A, B \in V$ ist, gehe man folgendermaßen vor:

1. Betrachte Zyklen der Form $B_1 \to B_2, B_2 \to B_3, \ldots, B_{k-1} \to B_k, B_k \to B_1$ und ersetze die Variablen B_i durch eine neue Variable B.
2. Wir sorgen nun dafür, dass S auf keiner rechten Seite vorkommt. Wir fügen zu P' die Produktion $S \to T$ hinzu, und ersetzen alle S auf der rechten Seite durch T. Dann wird für jede Produktion $S \to u, u \neq \varepsilon, u \in \Sigma \cup V$ eine Produktion $T \to u$ zu P' hinzugefügt.



3. Ordne die Variablen $\{A_1, \ldots, A_l\}$, so dass $A_i \to A_j$ nur für $i < j$.

4. Eliminiere von hinten nach vorne (d.h. für $k = l-1, \ldots, 1$) durch Ersetzen von $A_k \to A_{k'}$ $k' > k$ durch $A_k \to x_1 | \ldots | x_m$ wobei $x_1, \ldots, x_m$ die Satzformen sind, die aus $A_{k'}$ entstehen können (d.h. $A_{k'} \to x_1 | \ldots | x_m$).

Definition 1.12 *Eine kontextfreie Grammatik G mit $\varepsilon \notin L(G)$ ist in Chomsky Normalform, falls jede Regel eine der beiden Formen $A \to BC$ oder $A \to \mathsf{a}$ hat, wobei A, B und C Variablen sind und a ein Terminalsymbol ist.*

Bemerkung zu Definition 1.12: Ist eine kontextfreie Grammatik G in Chomsky Normalform (CNF), so hat sie die (wichtige) Eigenschaft, dass ihre Ableitungsbäume stets Binärbäume sind. Daraus folgt, dass ein Wort der Länge $n \in L(G)$ kann in genau $2n-1$ Schritten abgeleitet werden kann.

Definition 1.13 *Eine kontextfreie Grammatik G mit $\varepsilon \notin L(G)$ ist in Greibach Normalform, falls jede Regel die Form $A \to \mathsf{a}B_1B_2 \ \ldots \ B_l$ hat, wobei $A, B_1, B_2, \ \ldots \ , B_l$ Variablen sind und a ein Terminalsymbol ist.*

Existenz von Normalformen

Satz 1.10 *Zu jeder kontextfreien Grammatik G mit $\varepsilon \notin L(G)$ gibt es eine Grammatik G' in Chomsky Normalform mit $L(G) = L(G')$.*

Beweis: Existenz der CNF
Alle Regeln der Grammatik $G = (V, \Sigma, P, S)$ haben die Form $A \to a$ oder $A \to x$ mit $x \in (V \cup \Sigma)^*$, $|x| \geq 2$.

- Für jedes Terminalzeichen $a \in \Sigma$ führen wir eine neue Variable B_a ein mit der Regel: $B_a \to a$.

- In den Produktionen wird jedes Terminalsymbol a durch die entsprechende Variable B_a ersetzt.

- Regel der Form $A \to B_1B_2 \ldots B_l$. Diese Regel wird ersetzt durch die Regel $A \to B_1C_1$, wobei für C_1 eine neue Regel der Form $C_1 \to B_2 \ldots B_l$ eingeführt wird. Entsprechend werden neue Regeln der Form $C_3 \to B_3 \ldots B_l \ldots$ eingeführt, bis nur noch Regeln der Form $X \to YZ$ vorhanden sind.

$$
\begin{array}{l}
A \ \to \ B_1\underbrace{\boxed{B_2, \ \ldots \ , B_l}}_{\textstyle C_1 \ \to \ B_2\underbrace{\boxed{B_3, \ \ldots \ , B_l}}_{\textstyle C_2 \ \to \ B_3\underbrace{\boxed{B_4, \ \ldots \ , B_l}}_{\textstyle C_3 \ \to \ \ldots}}}
\end{array}
$$

□

1.3.2 Wortproblem, CYK-Algorithmus

Das Wortproblem für kontextfreie Grammatiken

Gegeben sei eine kontextfreie Grammatik G und ein Wort x mit $x \in \Sigma^*$. Um festzustellen ob $x \in L(G)$ ist, geht man wie folgt vor:

1. Schritt: Überführe G in eine äquivalente Grammatik G' in CNF (Chomsky-Normalform). Algorithmus dazu siehe Seite 19.

2. Schritt: Löse das Wortproblem für die Grammatik G' in CNF. Verwende hierzu den *CYK-Algorithmus* (benannt nach COCKE, YOUNGER, KASAMI).

Der CYK-Algorithmus basiert auf der Idee der dynamischen Programmierung.

Sei $x = x_1 x_2 \dots x_i \dots x_j \dots x_n$ mit $x_i \in \Sigma$. Berechne für alle i, j die Menge $T_{i,j}$ der Variablen, aus denen man das Teilwort $x_i \dots x_j$ ableiten kann. Dann gilt (mit S als Startsymbol):

$$x \in L(G) \Leftrightarrow S \in T_{1,n}$$

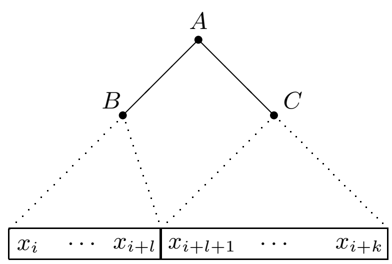
Die $T_{i,j}$ berechnet man induktiv über die Länge k des Teilwortes $x_i \dots x_j$:

Induktionsanfang: $k = 1$:

$$T_{i,i} = \{A \in V \mid A \to x_i\}$$

Induktionsschritt: $k \to k + 1$:

$$T_{i,k+i-1} = \{A \in V \mid A \to BC, \exists 0 < l < k\ B \in T_{i,i+l}, C \in T_{i+l+1,i+k}\}$$



Es existiert also eine Ableitung $A \to BC$, so das B den Anfang des Wortes erzeugt und C genau den Rest.

Beispiel 1.21

Gegeben sei die Grammatik G mit folgenden Produktionen:

$$S \to AB,\ A \to \mathsf{ab} | \mathsf{a}A\mathsf{b},\ B \to \mathsf{c} | \mathsf{c}B$$

Man sieht leicht, dass G kontextfrei ist und für die von G erzeugte Sprache $L(G)$ gilt:

$$L(G) = \{\mathsf{a}^n \mathsf{b}^n \mathsf{c}^m \mid n, m \in \mathbb{N}\}$$

Die Umformung in CNF ergibt:

$$S \to AB,\ A \to CD|CF,\ B \to \mathsf{c}|EB,\ C \to \mathsf{a},\ D \to \mathsf{b},\ E \to \mathsf{c},\ F \to AD$$

Möchte man nun überprüfen ob $x =$ aaabbbcc $\in L(G)$, so geht man folgendermaßen vor:

1. Zur Lösung des Problems verwenden wir eine Tabelle mit Spaltenindex i und Zeilenindex k. An der Position k, i befindet sich am Ende des Verfahrens T_{ik}, also die Menge der Variablen aus denen sich das Teilwort $x_i \dots x_{i+k}$ erzeugen lässt. Existiert keine Variable aus der man das Teilwort ableiten kann, so ist diese Menge leer.

2. Weiterhin enthält die Tabelle im Tabellenkopf in der i. Spalte das Zeichen x_i des Wortes x.

3. In die erste Zeile der Tabelle werden nun in jedem Feld $1, i$ die Variablen X eingetragen, für die eine Produktion $X \rightarrow x_i$ existiert.

   In unserem Beispiel existiert für die Terminalzeichen a und b jeweils nur eine Produktion, nämlich $C \rightarrow$ a bzw. $D \rightarrow$ b. Für das Terminalzeichen c hingegen existieren zwei Produktionen, $E \rightarrow$ c und $B \rightarrow$ c. Dementsprechend ergeben sich die Einträge in der ersten Zeile.

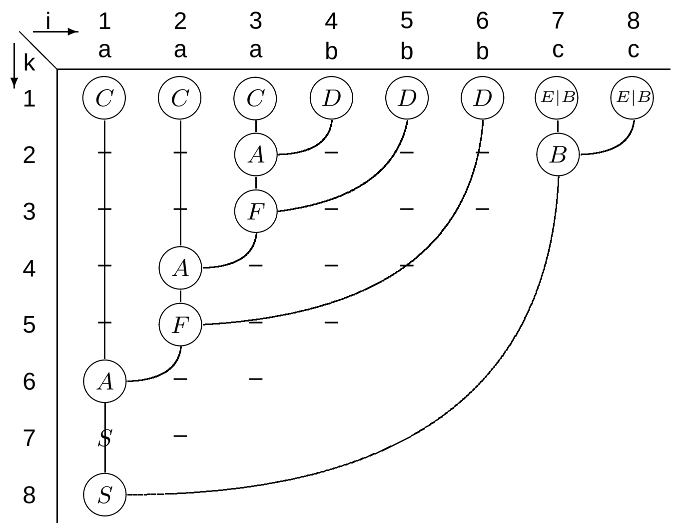
4. Für die weiteren Felder k, i der Tabelle sucht man nun nach Ableitungen $X \rightarrow YZ$, die das betrachtete Teilwort $x_i \dots x_{i+k}$ erzeugen. Und zwar so, dass Y den Anfang des Teilwortes erzeugt und Z genau den Rest. Es muss also ein l mit $1 \leq l < k$ existieren, so dass Y im Feld l, i und Z im Feld $k - l, i + l$ steht.

   In unserem Beispiel existieren für die ersten beiden Felder $2, 1$ und $2, 2$ keine passenden Produktionen (die Grammatik enthält keine Produktion der Art $X \rightarrow CC$). Erst für das Feld $2, 3$ existiert eine passende Produktion, nämlich $A \rightarrow CD$. Dementsprechend wird A im Feld $2, 3$ gespeichert, ...

k \ i	1	2	3	4	5	6	7	8
	a	a	a	b	b	b	c	c
1	C	C	C	D	D	D	$E\|B$	$E\|B$
2	–	–	$\overset{A\rightarrow CD}{A}$	–	–	–	$\overset{B\rightarrow EB}{B}$	
3	–	–	$\overset{F\rightarrow AD}{F}$	–	–	–		
4	–	$\overset{A\rightarrow CF}{A}$	–	–	–			
5	–	$\overset{F\rightarrow AD}{F}$	–	–				
6	$\overset{A\rightarrow CF}{A}$	–	–					
7	$\overset{S\rightarrow AB}{S}$	–						
8	$\overset{S\rightarrow AB}{S}$							

5. Der Algorithmus ist beendet wenn man das Feld $n, 1$ $n = |x|$ ereicht hat. Enthält dieses Feld das Startzeichen S, so ist das Wort x in der Sprache enthalten, sonst nicht.

6. Ist das Wort in der Sprache enthalten, lässt sich aus der Tabelle eine Ableitung (oder je nach Fall auch mehrere) konstruieren.



1.3.3 Abschlusseigenschaften

Satz 1.11 *Die kontextfreien Sprachen sind abgeschlossen unter*

- *Vereinigung*
- *Produkt*
- *Sternbildung*

aber sie sind nicht *abgeschlossen unter*

- *Schnitt*
- *Komplement*

Beweis:

"$\cup$": (Vereinigung) Seien $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$ kontextfrei. Ohne Einschränkung gelte ebenso $V_1 \cap V_2 = \emptyset$. Sei nun

$$\begin{aligned} G &= (V_1 \cup V_2 \cup \{S\}, \Sigma, P, S) \qquad \text{mit} \\ P &= \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2 \end{aligned}$$

Klar ist, dass $L(G) = L(G_1) \cup L(G_2)$ kontextfrei ist.

"$\cdot$": (Produkt) Beweis wie oben, aber: $P = \{S \to S_1S_2\} \cup P_1 \cup P_2$

"$*$": (Sternbildung) Der Beweis erfolgt wie oben, allerdings gilt hier: $P = \{S \to \varepsilon,\ S \to S_1,\ S_1 \to S_1S_1\} \cup P_1 \setminus \{S_1 \to \epsilon\}$. Dann ist $L(G) = (L(G))^*$

"$\cap$": (Schnitt) Der Beweis erfolgt über ein Gegenbeispiel. Die Sprachen L_1 und L_2 mit:

$$\begin{aligned} L_1 &= \{a^nb^nc^m \mid n, m \in \mathbb{N}\} \qquad \text{und} \\ L_2 &= \{a^mb^nc^n \mid n, m \in \mathbb{N}\} \end{aligned}$$

sind beide kontextfrei (siehe hierzu Beispiel 1.21 auf Seite 20). Sei nun $L := L_1 \cap L_2$, dann gilt:

$$L \quad = \quad \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

Die Sprache L ist aber nicht kontextfrei (siehe Beispiel 1.22 auf Seite 24).

"\": (Komplement) Gegenbeispiel: de Morgan

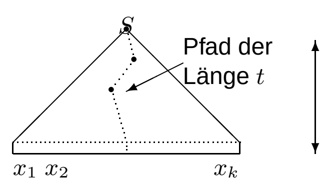
□

1.3.4 Pumping Lemma für kontextfreie Grammatiken

Satz 1.12 *Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl $n \in \mathbb{N}$, so dass sich alle Wörter $z \in L$ mit $|z| \geq n$ in $z = uvwxy$ zerlegen lassen, wobei folgende Eigenschaften erfüllt sind:*

1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. *für alle $i \geq 0$ ist $uv^iwx^iy \in L$.*

Beweis: Setze $n := 2^{\text{\#Variablen}}$. Ebenso sei o.E. G mit $L(G)$ kontextfrei in CNF gegeben. Sei z ein beliebiges Wort aus $L(G)$ mit $k = |z| \geq n$. Wir betrachten nun den im Folgenden dargestellten Ableitungsbaum, wobei in der letzten Ebene nur Produktionen der Form $A \to \mathsf{a}$ zu finden sind (da G ja in CNF gegeben ist). Man beachte hierbei, dass der Ableitungsbaum **ausgenommen der letzten Ebene** ein Binärbaum ist, s. hierzu Bemerkung zu Definition 1.12 auf Seite 19.



Tiefe t
Klar $t \geq \log_2 n + 1$, da $\log_2 n \leq \log_2 k \leq t-1$ (beachte die letzte Ebene!)

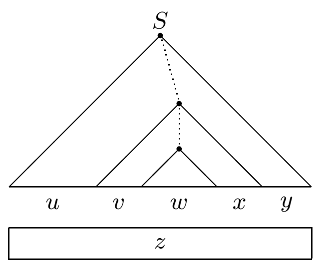
Für einen Pfad der Länge t gilt also: $t \geq \log_2 n + 1 = \text{\#Variablen} + 1$, d.h. der Pfad enthält mindestens eine Variable mindestens zweimal. Man beachte hierbei, dass die Länge eines Pfades die Anzahl der Kanten ist. Unser Pfad der Länge $t = \text{\#Variablen} + 1$ hat also #Variablen $+ 2$ Knoten, von denen #Variablen $+ 1$ mit Variablen markiert sind.

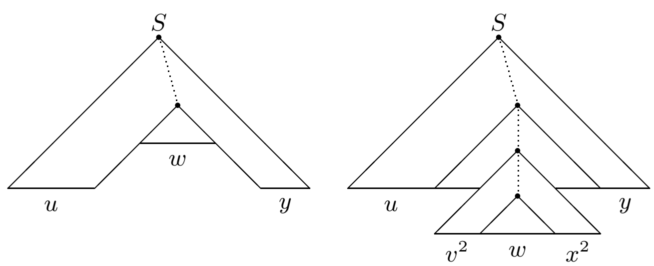
Wir können obige Aussage noch verfeinern: Tatsächlich muss sogar eine Variable in der Nähe des Endes des Pfades zweimal auftreten. Betrachten wir einen längsten Pfad P im Ableitungsbaum. In P muss es nun zwei Knoten v_1 und v_2 geben, die den folgenden Bedingungen genügen:

1. v_1 und v_2 haben dieselbe Markierung, z.B. A
2. Knoten v_1 liegt näher an der Wurzel als Knoten v_2
3. der Teil des Pfades von v_1 zu dem Blatt hat eine Länge von höchstens #Variablen $+ 1$

Um zu sehen, dass v_1 und v_2 immer gefunden werden können, geht man den Pfad P vom Blatt her zurück, wobei man die Spur der besuchten Markierungen verfolgt. Von den ersten #Variablen $+\ 2$ besuchten Knoten hat nur das Blatt ein Terminalzeichen als Markierung. Die verbleibenden #Variablen $+\ 1$ Knoten können daher nicht alle verschiedene Variablen als Markierung haben.

Die Knoten v_1 und v_2 definieren uns nun die Ausgangspunkte, um die im Folgenden graphisch dargestellte $uvwxy$-Zerlegung für das gewählte Wort z der Länge $k = |z| \geq n$ zu finden (die letzte Ebene wird in den folgenden Graphiken nicht gesondert ausgewiesen).



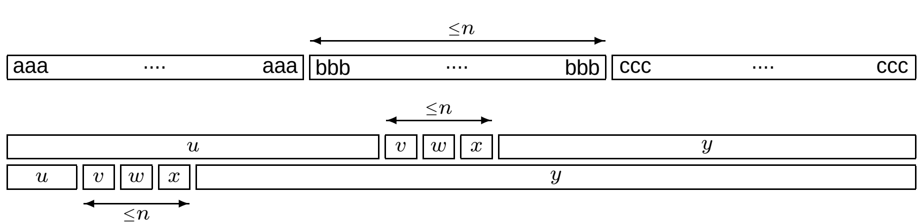


Dieser Ableitungsbaum zeigt $uwy \in L$

Dieser Ableitungsbaum zeigt $uv^2wx^2y \in L$

Die Bedingungen 1 und 2 sind hierbei ebenso erfüllt. Warum? → ÜA. □

Beispiel 1.22
Mit dem Pumping Lemma kann man nun zeigen, dass die Sprache $L = \{\text{a}^m\text{b}^m\text{c}^m \mid m \geq 1\}$ *nicht* kontextfrei ist.
Beweis: Angenommen doch. Dann kann man das Pumping-Lemma anwenden: Sei n die Zahl aus dem Lemma. Betrachte nun das Wort $z = \text{a}^{3n}\text{b}^{3n}\text{c}^{3n}$.



Wegen $|vwx| \leq n$ kann vwx nicht gleichzeitig a's *und* b's *und* c's enthalten. Daher enthält vx unterschiedlich viele a's und c's und das Wort uv^2wx^2y ist nicht in der Sprache L enthalten ($uv^2wx^2y \notin L$). Dies steht im Widerspruch zur Aussage des Pumping Lemmas und damit kann L nicht kontextfrei sein. □
Anmerkung zum Beweis: Man beachte auch die bereits zum Beweis des Satzes 1.8 auf Seite 15 angegebenen Anmerkungen.

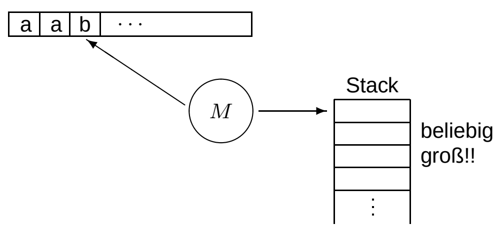
Beispiel 1.23
Weiterhin kann man mit dem Pumping Lemma zeigen:

- $L = \{1^p \mid p \text{ ist Primzahl }\}$ ist *nicht* kontextfrei.
- $L = \{1^n \mid n \text{ ist Quadratzahl }\}$ ist *nicht* kontextfrei.
- Jede kontextfreie Sprache über einem einelementigen Alphabet ist bereits regulär.

1.3.5 Kellerautomaten

Für reguläre Sprachen gilt, dass es zu jeder regulären Sprache einen endlichen Automaten gibt, der diese Sprache erkennt bzw. dass es zu jedem endlichen Automaten eine entsprechende reguläre Sprache gibt.

Für kontextfreie Sprache ergibt sich nun: $L = \{\mathrm{a}^m\mathrm{b}^m \mid m \in \mathbb{N}\}$ kann *nicht* durch einen endlichen Automaten erkannt werden, da dieser keinen Speicher besitzt, um sich die Anzahl der schon gelesenen a's zu merken. Deshalb erlauben wir jetzt einen Speicher in Form eines Kellers / Stacks.



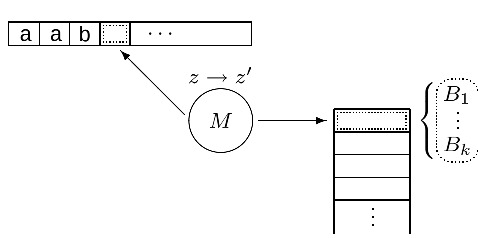
Definition 1.14 *Ein nichtdeterministischer Kellerautomat (englisch: pushdown automata, kurz PDA) wird durch ein 6-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ beschrieben, das folgende Bedingungen erfüllt:*

– Z *ist eine endliche Menge von Zuständen.*
– Σ *ist eine endliche Menge, das Eingabealphabet.*
– Γ *ist eine endliche Menge, das Kelleralphabet.*
– $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \langle$*endliche Teilmengen von* $Z \times \Gamma^*\rangle$, *die Übergangsfunktion.*
– $z_0 \in Z$ *ist der Startzustand.*
– $\# \in \Gamma$ *ist das unterste Kellerzeichen, auch Kellerbodenzeichen genannt.*

Intuitiv bedeutet $\delta(z, a, A) \ni (z', B_1 \ldots B_k)$:
Wenn sich M im Zustand z befindet und das oberste Zeichen des Kellers ein A ist, kann man durch Lesen des Eingabezeichens a in den Zustand z' übergehen, wobei das Zeichen A des Kellers durch $B_1 \ldots B_k$ ersetzt wird.
Die folgenden Spezialfälle sind erlaubt:

$a = \varepsilon$ d.h., Kellerinhalt wird verändert *ohne*, dass ein Eingabezeichen gelesen wird.

$k = 0$ d.h., das Zeichen A des Kellers wird gelöscht ("gePOPt").



Die von M *akzeptierte* Sprache $L(M)$ (also die Sprache, die vom Kellerautomaten M erkannt wird) ist die Menge der Worte, für die es eine Folge von Übergängen gibt, so daß der Keller nach Abarbeiten des Wortes *leer* ist.

Beispiel 1.24
Gegeben sei die Sprache L mit $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$. Der Kellerautomat für L besitzt zwei Zustände:

1. $z_0 \hat{=}$ Interpretation: “Liest a-Teil”
2. $z_1 \hat{=}$ Interpretation: “Liest b-Teil”

Der Automat ergibt sich also zu: $M = (\{z_0, z_1\}, \{\mathsf{a}, \mathsf{b}\}, \{\#, A\}, \delta, z_0, \#)$
Für die Zustandsübergangsfunktion gilt:

$$\begin{array}{rcl} \delta(z_0, \mathsf{a}, \#) & \ni & (z_0, A\#), (z_1, A\#) \\ \delta(z_0, \mathsf{a}, A) & \ni & (z_0, AA), (z_1, AA) \\ \delta(z_1, \mathsf{b}, A) & \ni & (z_1, \varepsilon) \\ \delta(z_1, \varepsilon, \#) & \ni & (z_1, \varepsilon) \end{array}$$

Gibt es nun auch die Möglichkeit, für kontextfreie Sprachen deterministische Kellerautomaten zu konstruieren? Ist dies dann auch für jede kontextfreie Sprache möglich? Hierzu folgendes Beispiel:

Beispiel 1.25

$$L_1 = \{x_1x_2 \ldots x_n\$x_n \ldots x_2x_1 \mid n \in \mathbb{N}, x_i \in \Sigma \setminus \{\$\}\}$$

Diese Sprache kann (wegen des Trennzeichens \$) durch einen deterministischen Kellerautomaten erkannt werden.

$$L_2 = \{x_1x_2 \ldots x_nx_n \ldots x_2x_1 \mid n \in \mathbb{N}, x_i \in \Sigma\}$$

Die Sprache L_2 kann durch einen nichtdeterministischen Kellerautomaten erkannt werden, aber *nicht* durch einen deterministischen Kellerautomaten (die Trennstelle zwischen den beiden x_n muss vom Automaten “erraten” werden).

Definition 1.15 *Für einen deterministischen Kellerautomaten muss (zusätzlich zu den Angaben in der Definition eines nichtdeterministischen Kellerautomaten) gelten:*

- *Er hat eine Menge $E \subseteq Z$ von Endzuständen.*
- *Es muss gelten:*

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

- *Ein Wort x wird genau dann vom Automaten akzeptiert, wenn sich die Maschine in einem Endzustand befindet. Der Stack muss hierbei* nicht *leer sein!*

Es ist klar, dass das Wortproblem von einem deterministischen Kellerautomaten in *linearer* Zeit gelöst werden kann.
Aber leider gibt es *nicht* zu jeder kontextfreien Sprache einen deterministischen Kellerautomaten (siehe hierzu Beispiel 1.25).

Satz 1.13 *Ist $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik, so ist $M = (\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$ ein nichtdeterministischer Kellerautomat, wenn wir δ wie folgt definieren:*

- *Für jede Regel $A \to \alpha$ setzen wir $\delta(z, \epsilon, A) \ni (z, \alpha)$.*

- *Zusätzlich fügen wir für alle $a \in \Sigma$ noch $\delta(z,a,a) \ni (z,\varepsilon)$ ein. .*

Beispiel 1.26
Gegeben sei die kontextfreie Grammatik G mit $S \rightarrow$ ab | aSb. Dann erfolgt die Konstruktion des Kellerautomaten wie folgt:

- Der Kellerautomat verfügt nur über einen Zustand, der mit z bezeichnet wird.
- Das Kelleralphabet Γ ist definiert mit $\Gamma = \{S, \mathsf{a}, \mathsf{b}\}$.
- Das unterste Kellerzeichen ist S.
- Nun erfolgt die Definition der Übergangsfunktionen δ:
  1. Die Regel $S \rightarrow \mathsf{ab}$ führt zu:
     $$\delta(z,\epsilon,S) \ni (z,\mathsf{ab}).$$
  2. Die Regel $S \rightarrow \mathsf{a}S\mathsf{b}$ führt zu:
     $$\delta(z,\epsilon,S) \ni (z,\mathsf{a}S\mathsf{b}).$$
  3. Zusätzlich noch:
     $$\delta(z,\mathsf{a},\mathsf{a}) \ni (z,\varepsilon), \qquad \delta(z,\mathsf{b},\mathsf{b}) \ni (z,\varepsilon).$$

Arbeitsweise bei Erkennung des Wortes aabb:

	ungelesene Zeichen der Eingabe	Kellerinhalt
Startzustand:	aabb	S
Regel (2):	aabb	aSb
Regel (3):	abb	Sb
Regel (1):	abb	abb
Regel (3):	bb	bb
Regel (3):	b	b
Regel (3):	ε	ε

Satz 1.14 *Ist $M = (Z,\Sigma,\Gamma,\delta,z_0,\#)$ ein nichtdeterministischer Kellerautomat, so ist $G = (V,\Sigma,P,S)$ eine kontextfreie Grammatik, wenn wir V und P wie folgt definieren:*
$$V := \{S\} \cup Z \times \Gamma \times Z$$
und

$S \rightarrow [z_0,\#,z]$	*für alle $z \in Z$*
$[z,A,z'] \rightarrow a$	*für alle Übergänge $\delta(z,a,A) \ni (z',\varepsilon)$*
$[z,A,y\ ] \rightarrow a[z',B,y]$	*für alle Übergänge $\delta(z,a,A) \ni (z',B)$* und *alle $y \in Z$*
$[z,A,y'] \rightarrow a[z',B,y][y,C,y']$	*für alle Übergänge $\delta(z,a,A) \ni (z',BC)$* und *alle $y,y' \in Z$*

[*Achtung:* Hier nehmen wir ohne Einschränkung (wieso?!) an, daß es keine Übergänge $\delta(z,a,A) \ni (z',B_1 \dots B_k)$ mit $k \geq 3$ gibt.]

Beispiel 1.27
Sei nun ein Kellerautomat gegeben durch:

$$\begin{array}{l}
\delta(z_0,\mathsf{a},\#) \ni (z_0,A\#),\ (z_1,A\#) \\
\delta(z_0,\mathsf{a},A) \ni (z_0,AA),\ (z_1,AA) \\
\delta(z_1,\mathsf{b},A) \ni (z_1,\varepsilon), \\
\delta(z_1,\varepsilon,\#) \ni (z_1,\varepsilon)
\end{array}$$

Wie unschwer zu erkennen, erzeugt dieser Automat, der uns schon aus Beispiel 1.24 von Seite 26 bekannt ist, die gleiche Sprache wie die Grammatik aus Beispiel 1.26 auf Seite 27. Nun konstruieren wir nach den Regeln aus Satz 1.14 aus dem Kellerautomaten eine Grammatik (bitte mit der aus dem Beispiel 1.26 vergleichen):

- Das neue Startsymbol: S
- Die Variablen: S und 8 Variablen der Form $[z_i, \#/A, z_j]$
- Die Produktionen für Startzustand:

$$S \rightarrow [z_0, \#, z_0] \mid [z_0, \#, z_1]$$

Die Produktionen für Übergang $\delta(z_1, \mathsf{b}, A) \ni (z_1, \varepsilon)$:

$$[z_1, A, z_1] \rightarrow \mathsf{b}$$

Produktionen für Übergang $\delta(z_1, \varepsilon, \#) \ni (z_1, \varepsilon)$:

$$[z_1, \#, z_1] \rightarrow \varepsilon$$

Produktionen für Übergang $\delta(z_0, \mathsf{a}, \#) \ni (z_0, A\#)$:

$$\begin{aligned}
[z_0, \#, z_0] &\rightarrow \mathsf{a}[z_0, A, z_0][z_0, \#, z_0] \\
[z_0, \#, z_1] &\rightarrow \mathsf{a}[z_0, A, z_0][z_0, \#, z_1] \\
[z_0, \#, z_0] &\rightarrow \mathsf{a}[z_0, A, z_1][z_1, \#, z_0] \\
[z_0, \#, z_1] &\rightarrow \mathsf{a}[z_0, A, z_1][z_1, \#, z_1]
\end{aligned}$$

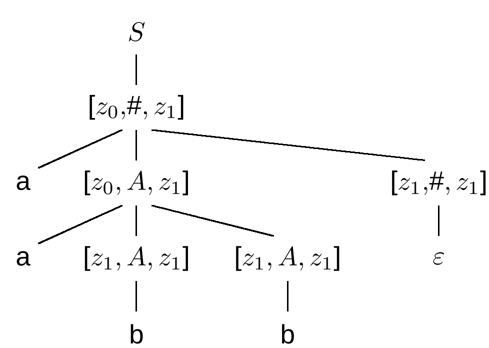
Produktionen für Übergang $\delta(z_0, \mathsf{a}, \#) \ni (z_1, A\#)$:

$$\begin{aligned}
[z_0, \#, z_0] &\rightarrow \mathsf{a}[z_1, A, z_0][z_0, \#, z_0] \\
[z_0, \#, z_1] &\rightarrow \mathsf{a}[z_1, A, z_0][z_0, \#, z_1] \\
[z_0, \#, z_0] &\rightarrow \mathsf{a}[z_1, A, z_1][z_1, \#, z_0] \\
[z_0, \#, z_1] &\rightarrow \mathsf{a}[z_1, A, z_1][z_1, \#, z_1]
\end{aligned}$$

Analog für die beiden restlichen beiden Übergänge:

$$\begin{aligned}
[z_0, A, z_0] &\rightarrow \mathsf{a}[z_0, A, z_0][z_0, A, z_0] \\
[z_0, A, z_1] &\rightarrow \mathsf{a}[z_0, A, z_0][z_0, A, z_1] \\
[z_0, A, z_0] &\rightarrow \mathsf{a}[z_0, A, z_1][z_1, A, z_0] \\
[z_0, A, z_1] &\rightarrow \mathsf{a}[z_0, A, z_1][z_1, A, z_1] \\
[z_0, A, z_0] &\rightarrow \mathsf{a}[z_1, A, z_0][z_0, A, z_0] \\
[z_0, A, z_1] &\rightarrow \mathsf{a}[z_1, A, z_0][z_0, A, z_1] \\
[z_0, A, z_0] &\rightarrow \mathsf{a}[z_1, A, z_1][z_1, A, z_0] \\
[z_0, A, z_1] &\rightarrow \mathsf{a}[z_1, A, z_1][z_1, A, z_1]
\end{aligned}$$

Der Ableitungsbaum für aabb sieht damit wie folgt aus:



Auf Seite 26 haben wir bereits in Definition 1.15 einen deterministischen Kellerautomaten definiert. Ein solcher Automat hat zu jedem Zeitpunkt höchstens eine Alternative. Daher der Name *deterministisch*.

Um nun die von einem deterministischen Kellerautomaten M akzeptierte Sprache $L(M)$ zu definieren, mussten wir $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ noch um eine Menge $E \subseteq Z$ von *Endzuständen* erweitern. $L(M)$ besteht dann aus genau den Worten, für die es eine Folge von Übergängen gibt, so dar sich M nach Abarbeiten des Wortes in einem Endzustand befindet. Darauf aufbauend definieren wir nun:

Definition 1.16 *Eine kontextfreie Grammatik G heisst deterministisch kontextfrei, falls es einen deterministischen Kellerautomaten M gibt mit $L(G) = L(M)$.*

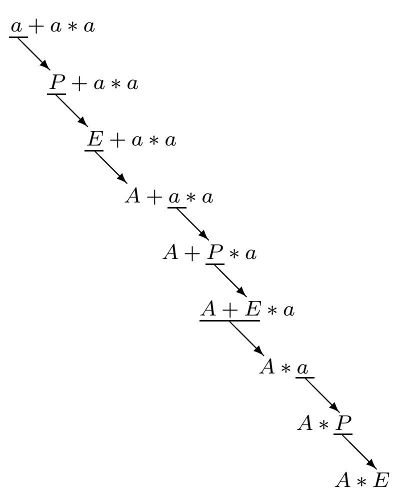
1.3.6 $LR(k)$-Grammatiken

Die *Idee* ist hierbei: Wir wollen eine Grammatik, für die wir zu einem gegebenem Wort "schnell" eine Ableitung erzeugen können.

Beispiel 1.28
Eine Grammatik für Arithmetische Ausdrücke sei durch folgende Regeln gegeben:

$$
\begin{aligned}
&S \to A \\
&A \to E \\
&A \to A + E \\
&A \to A - E \\
&E \to P \\
&E \to E * P \\
&E \to E/P \\
&P \to (A) \\
&P \to a
\end{aligned}
$$

Hierbei tritt das *Problem* auf, dass man beim Reduzieren von links in Sackgassen laufen kann, wie in folgendem Ableitungsbaum deutlich wird:



Zur Behebung führen wir "Lookaheads" (der Länge k) ein. Diese sind wie folgt zu interpretieren: Die Regel darf nur dann angewendet werden, wenn die nächsten k Zeichen mit den erlaubten Lookaheads übereinstimmen.

Beispiel 1.29
Arithmetische Ausdrücke mit Lookaheads

Regeln	Lookaheads (der Länge 1)
$S \to A$	ϵ
$A \to E$	$+, -,), \epsilon$
$A \to A + E$	$+, -,), \epsilon$
$A \to A - E$	$+, -,), \epsilon$
$E \to P$	beliebig
$E \to E * P$	beliebig
$E \to E/P$	beliebig
$P \to (A)$	beliebig
$P \to a$	beliebig

Damit ergibt sich nun folgender Ableitungsbaum:

$$\underline{a} + a * a$$
$$\underline{P} + a * a$$
$$\underline{E} + a * a$$
$$A + \underline{a} * a$$
$$A + \underline{P} * a$$
$$\underline{A + E} * a$$
$$A + \underline{E * P}$$
$$\underline{A + E}$$
$$\underline{A}$$
$$S$$

Dies führt auf folgende Definition einer $LR(k)$-Grammatik

Definition 1.17 *Eine kontextfreie Grammatik ist eine $LR(k)$-Grammatik, wenn man durch Lookaheads der Länge k erreichen kann, dass bei einer Reduktion von links nach rechts in jedem Schritt genau eine Regel anwendbar ist.*

Korollar 1.2 *Jede kontextfreie Sprache für dies es eine $LR(k)$-Grammatik gibt, ist deterministisch kontextfrei.*

1.4 Kontextsensitive und Typ 0 Sprachen

1.4.1 Turingmaschine

Definition 1.18 *Eine nichtdeterministische Turingmaschine (kurz TM) wird durch ein 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ beschrieben, das folgende Bedingungen erfüllt:*

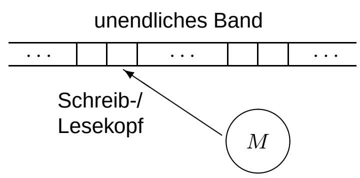
- Z *ist eine endliche Menge von* Zuständen.
- Σ *ist eine endliche Menge, das* Eingabealphabet.
- Γ *ist eine endliche Menge, das* Bandalphabet.
- $\delta : Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$*, die* Übergangsfunktion.
- $z_0 \in Z$ *ist der* Startzustand.

– $\Box \in \Gamma \setminus \Sigma$, *das* Leerzeichen.
– $E \subseteq Z$, *die Menge der* Endzustände.

Definition 1.19 *Eine nichtdeterministische Turingmaschine ist genau dann deterministisch, falls gilt*

$$|\delta(z,a)| = 1 \textit{ für alle } z \in Z, a \in \Gamma.$$

Erläuterung: Intuitiv bedeutet dabei $\delta(z,a) = (z', b, x)$ bzw.$\delta(z,a) \ni (z', b, x)$:
Wenn sich M im Zustand z befindet und unter dem Schreib-/Lesekopf das Zeichen a steht, so geht M im nächsten Schritt in den Zustand z' über, schreibt an die Stelle des a's das Zeichen b und bewegt danach den Schreib-/Lesekopf um eine Position nach rechts (falls $x = R$), links (falls $x = L$) oder läßt ihn unverändert (falls $x = N$).



Beispiel 1.30
Ziel ist die Angabe einer Turingmaschine, die einen gegeben String aus $\{0,1\}^*$ als Binärzahl interpretiert und zu dieser Zahl Eins addiert. Folgende Vorgehensweise bietet sich an:

1. Gehe ganz nach rechts (bis ans Ende der Zahl). Dieses Ende kann durch das erste Auftreten eines Leerzeichens gefunden werden.

2. Gehe wieder nach links bis zur ersten Null und mache aus dieser Null eine Eins. Mache dabei alle Einsen auf dem Weg zu einer Null.

Daraus ergibt sich folgende Beschreibung der Zustandübergänge:

1. $\delta(z_0, 0) = (z_0, 0, R)$
   $\delta(z_0, 1) = (z_0, 1, R)$
   $\delta(z_0, \Box) = (z_1, \Box, L)$

2. $\delta(z_1, 1) = (z_1, 0, L)$
   $\delta(z_1, 0) = (z_e, 1, N)$
   $\delta(z_0, \Box) = (z_e, 1, N)$

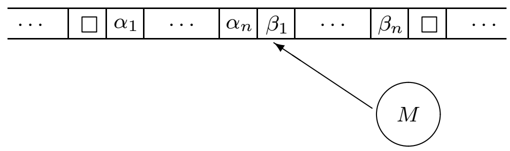
Benötigt wird also die Menge der Zustände $Z = \{z_0, z_1, z_e\}$, wobei sich die Menge der Endzustände zu $E = \{z_e\}$ ergibt.

Definition 1.20 *Eine Konfiguration einer Turingmaschine ist ein Tupel:*

$$(\alpha, \beta, z) \in \Gamma^* \times \Gamma^* \times Z$$

Interpretation: Das Wort $w = \alpha\beta$ entspricht der Belegung des Bandes, wobei dieses rechts und links von w mit dem Leerzeichen $\Box$ gefüllt sei. Der Schreib-/Lesekopf befindet sich auf dem ersten (d.h. linkesten) Zeichen von β.

Definition 1.21 *Die Startkonfiguration der Turingmaschine bei Eingabe $x \in \Sigma^*$ entspricht der Konfiguration (ϵ, x, z_0), d.h., auf dem Band befindet sich genau die Eingabe $x \in \Sigma^*$, der Schreib-/Lesekopf befindet sich auf dem ersten (d.h. linkesten) Zeichen der Eingabe und die Maschine startet im Zustand z_0.*

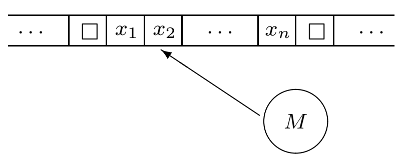


Je nach aktueller Bandkonfiguration und Richtung $x \in \{N, R, L\}$ ergeben sich folgende Konfigurationsübergänge auf dem Band für die Ausführung des Zustandsübergangs $\delta(z, \beta_1) = (z', c, x)$:

$$(\alpha_1 \ldots \alpha_n, \beta_1 \ldots \beta_m, z) \vdash \begin{cases} (\alpha_1 \ldots \alpha_n, c\,\beta_2 \ldots \beta_m, z') & \text{falls } x = N, \\ & n \geq 0, m \geq 1 \\ (\varepsilon, \Box\, c\,\beta_2 \ldots \beta_m, z') & \text{falls } x = L, \\ & n = 0, m \geq 1 \\ (\alpha_1 \ldots \alpha_{n-1}, \alpha_n\, c\,\beta_2 \ldots \beta_m, z') & \text{falls } x = L, \\ & n \geq 1, m \geq 1 \\ (\alpha_1 \ldots \alpha_n\, c, \Box, z') & \text{falls } x = R, \\ & n \geq 0, m = 1 \\ (\alpha_1 \ldots \alpha_n\, c, \beta_2 \ldots \beta_m, z') & \text{falls } x = R, \\ & n \geq 0, m \geq 2 \end{cases}$$

Die von einer Turingmaschine M akzeptierte Sprache kann folgendermaßen beschrieben werden:

$$L(M) := \{x \in \Sigma^+ \mid (\varepsilon, x, z_0) \vdash^* (\alpha, \beta, z) \text{ mit } \alpha \in \Gamma^*, \beta \in \Gamma^*, z \in E\}$$



Definition 1.22 *Eine Turingmaschine heißt linear beschränkt (kurz: LBA), falls für alle $z \in Z$ gilt:*

$$(z', a, y) \in \delta(z, \Box) \qquad \Longrightarrow \qquad a = \Box.$$

gilt, wobei $z' \in Z,\ a \in \Gamma,\ y \in \{N, R, L\}$. Das heisst also, dass ein Leerzeichen hier nie durch ein anderes Zeichen überschrieben wird. Mit anderen Worten: Die Turingmaschine darf ausschliesslich die Positionen beschreiben, an denen zu Beginn die Eingabe x stand.

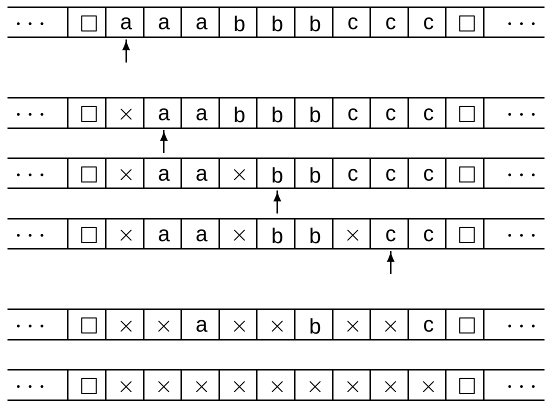
Satz 1.15 *Die von linear beschränkten, nichtdeterministischen Turingmaschinen akzeptierten Sprachen sind genau die kontextsensitiven Sprachen.*

Beweis: Der Beweis hierzu ist bitte in der Literatur nachzuschlagen. □

Satz 1.16 *Die Sprache $L = \{a^m b^m c^m \mid m \in \mathbb{N}\}$ ist kontextsenitiv.*

Beweis: Es gibt zwei Möglichkeiten, diesen Satz zu beweisen:

1. Angabe einer kontextsensitiven Grammatik für L.
2. Angabe einer nichtdeterministischen, linear beschränkten Turingmaschine, die L erkennt.



□

Analog wie in Satz 1.15 kann man zeigen:

Satz 1.17 *Die von Turingmaschinen akzeptierten Sprachen sind genau die Typ 0 Sprachen.*

Hinweis: Hier ist es egal, ob man die Turingmaschine deterministisch oder nichtdeterministisch wählt.
Genauer gilt sogar: Zu jeder nichtdeterministischen Turingmaschine M gibt es eine deterministische Turingmaschine M' mit $L(M) = L(M')$. Es ist zu beachten, dass man bisher noch *nicht* weiss, ob es auch zu jeder nichtdeterministischen, linear beschränkten Turingmaschine eine entsprechende deterministische, linear beschränkte Turingmaschine gibt.

1.5 Zusammenfassung

Zum Abschluss dieses Kapitels werden hier die wesentlichen Punkte aus den vorangegangenen Abschnitten tabellarisch zusammengefasst.

1.5.1 Chomsky-Hierarchie

Typ 3	reguläre Grammatik DFA NFA regulärer Ausdruck
Deterministisch–kontextfrei	$LR(k)$-Grammatik deterministischer Kellerautomat
Typ 2	kontextfreie Grammatik (nichtdeterministischer) Kellerautomat
Typ 1	kontextsensitive Grammatik (nichtdeterministische) linear beschränkte Turingmaschine
Typ 0	endliche Grammatik Turingmaschine

1.5.2 Abschlusseigenschaften

Chomsky-Typ	Schnitt	Vereinigung	Komplement	Produkt	Stern
Typ 3	ja	ja	ja	ja	ja
Det. kf.	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 0	ja	ja	nein	ja	ja

1.5.3 Wortproblem

Chomsky-Typ	Laufzeit
Typ 3, gegeben als DFA	lineare Laufzeit
Det. kf, gegeben als DPDA	lineare Laufzeit
Typ 2, gegeben durch Grammatik in CNF	CYK-Algorithmus, Laufzeit $O(n^3)$
Typ 1	$\langle$nächstes Kapitel$\rangle$
Typ 0	$\langle$nächstes Kapitel$\rangle$

1.5.4 Entscheidbarkeit

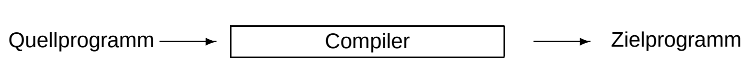
Folgende Tabelle zeigt die Entscheidbarkeit verschiedener Standardprobleme für die einzelnen Tpyen der Chomsky-Hierarchie:

Problem	Wortproblem	Leerheitsproblem	Äquivalenzpr.	Schnittproblem
Typ 3	ja	ja	ja	ja
Det. kf.	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

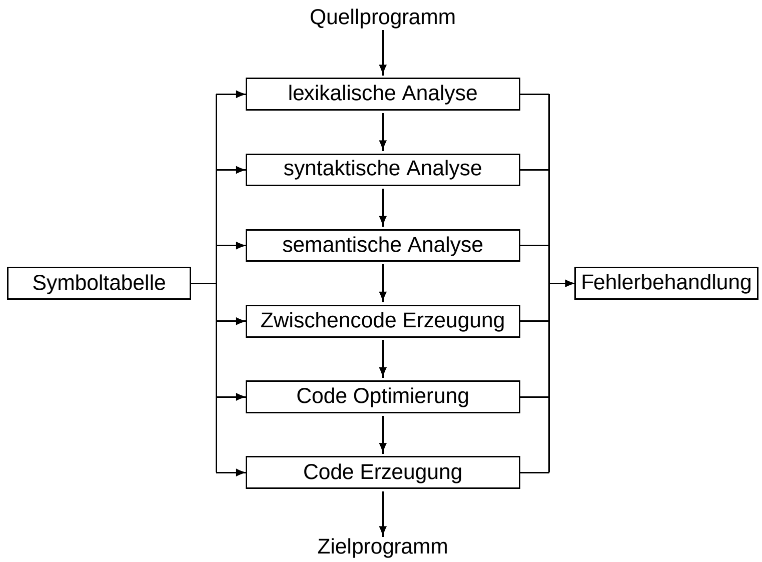
1.6 Compiler

Was ein Compiler macht, sollte eigentlich jedem bekannt sein. Im Folgenden ist noch einmal grob der Ablauf eines Compilerlaufes dargestellt. Die Skizze ist weitgehend selbsterkärend und mit dem im vorigen Semester zu Assemblern erworbenen Wissen sollte ihr Verständnis keine Probleme bereiten.

Im Folgenden wollen wir uns auf zwei Aspekte aus dem Ablaufdiagramm beschränken, nämlich auf die lexikalische Analyse und die syntaktische Analyse. Weitergehende Betrachtungen seien den dafür vorgesehenen Vorlesungen im Hauptstudium vorbehalten.



Phasen eines Compilerlaufes:



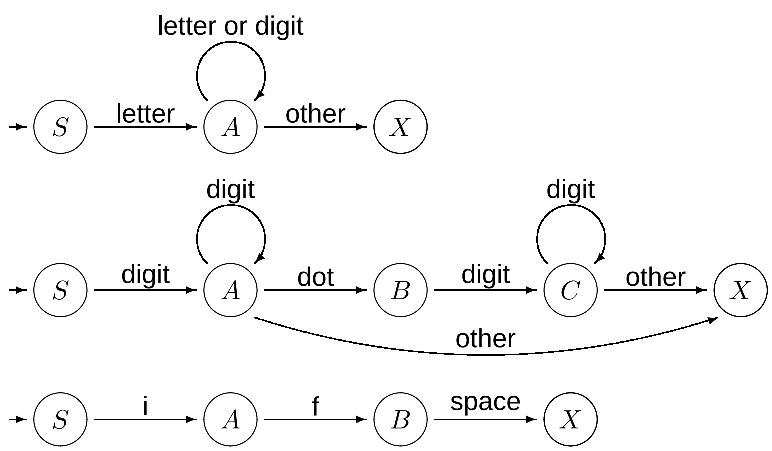
1.6.1 Lexikalische Analyse: Scanner

Aufgabe: Extrahiere aus dem Eingabestring die nächste "Einheit", z.B. Namen einer Variablen, eine Zahl, reserviertes Wort (wie z.B. if, while, etc.), "+"-Zeichen, usw.

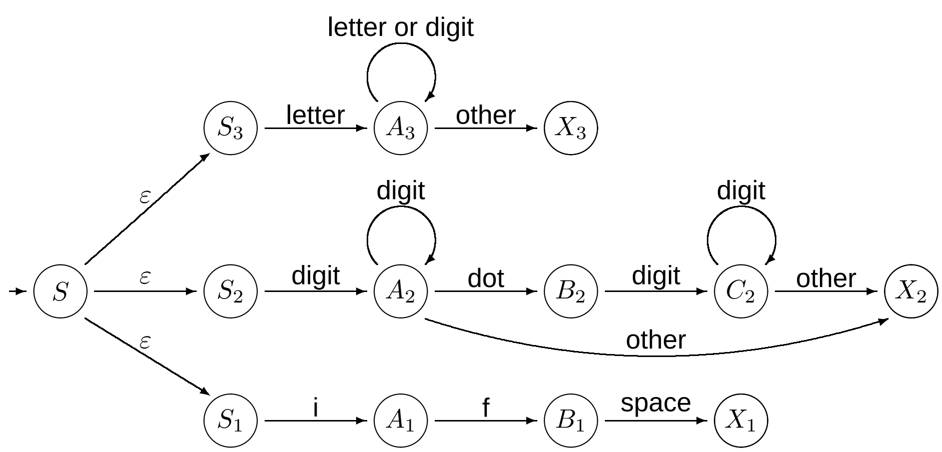
Beispiel 1.31
identifier := letter(letter|digit)*
number := digit$^+$ | digit$^+$.(digit)$^+$
if := if

Anwendung der Erkenntnisse der Vorlesung: Zu jedem regulären Ausdruck gibt es einen endlichen Automaten, der genau die Wörter aus dieser Sprache erkennt.

Beispiel 1.32



Für die lexikalische Analyse verbindet man diese endlichen Automaten zu einem einzigen *nichtdeterministischen* Automaten.



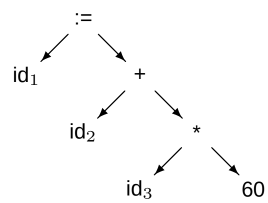
Anwendung der Erkenntnisse der Vorlesung: Mit Hilfe der *Potenzmengenkonstruktion* kann man daraus wieder einen deterministischen endlichen Automaten bauen ... und aus diesem kann man dann relativ einfach ein C oder Java Programm erzeugen.

Tools: Es gibt fertige Programme, die aus regulären Ausdrücken den zugehörigen Scanner erzeugen. Das klassische Tool hierfür ist `lex` (A Lexical Analyzer Generator); die entsprechende GNU-Version ist `flex`. Mehr Informationen hierzu unter `http://www.combo.org/lex_yacc_page/`.

1.6.2 Syntaktische Analyse: Parser

Aufgabe: Extrahiere aus der vom Scanner bereitgestellten Eingabe die logische Struktur.

Beispiel 1.33
Aus dem Ausdruck id_1 := id_2 + id_3 * 60 soll die folgende logische Struktur extrahiert werden:



Ansatz: Die zulässige Syntax eines Programms wird durch eine (möglichst deterministische) kontextfreie Grammatik beschrieben.

Anwendung der Erkenntnisse der Vorlesung:

- Mit Hilfe des CYK-Algorithmus kann man in $O(n^3)$ Zeit aus einem Wort die zugehörige Ableitung rekonstruieren.
- Ist die Grammatik deterministisch kontextfrei geht dies sogar in linearer Zeit.

Tools: Es gibt fertige Programme, die aus einer (geeignet spezifizierten) Grammatik einen zugehörigen (effizienten) Parser erzeugen. Das klassische Tool hierfür ist `yacc` (Yet Another Compiler-Compiler); die entsprechende GNU-Version ist `bison`. Mehr Informationen hierzu unter `http://www.combo.org/lex_yacc_page/`.

Kapitel 2

Berechenbarkeit und Entscheidbarkeit

2.1 Intuitiv berechenbar

Idee: $f : \mathbb{N}_0^k \to \mathbb{N}_0$ ist *berechenbar*, wenn es einen *Algorithmus* gibt, der f berechnet. Genauer: der bei Eingabe $(n_1 \dots n_k) \in \mathbb{N}_0^k$ nach endlich vielen Schritten mit dem Ergebnis $f(n_1 \dots n_k)$ stoppt.
Was bedeutet "Algorithmus" an dieser Stelle? C-Programm, JAVA-Programm, etc? Gibt es einen Unterschied, wenn man sich auf eine bestimmte Programmiersprache beschränkt?
Analog: für *partielle* Funktionen $f : A \to \mathbb{N}_0$, wobei $A \subseteq \mathbb{N}_0^k$, bedeutet berechenbar folgendes:

- Algorithmus soll mit richtigem Ergebnis stoppen, wenn $(n_1 \dots n_k) \in A$
- und nicht stoppen, wenn $(n_1 \dots n_k) \notin A$

Beispiel 2.1
Wir definieren folgende Funktionen:

$$f_1(n) = \begin{cases} 1 & \text{falls } n \text{ interpretiert als Ziffernfolge Anfangsstück} \\ & \text{von } \pi \text{ ist} \\ 0 & \text{sonst} \end{cases}$$
$$f_2(n) = \begin{cases} 1 & \text{falls } n \text{ interpretiert als Ziffernfolge in } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$
$$f_3(n) = \begin{cases} 1 & \text{falls mindestens } n \text{ aufeinanderfolgende Ziffern in } \pi \\ & \text{gleich 7 sind} \\ 0 & \text{sonst} \end{cases}$$

Einige Beispiele für $\pi = 3,141592...$: $f_1(314) = 1$, $f_1(415) = 0$, $f_2(415) = 1$
Zu obigen Funktionen ergeben sich folgende Aussagen zur Berechenbarkeit:

f_1: Wie man leicht einsieht, ist f_1 berechenbar, denn um festzustellen, ob eine Ziffernfolge ein Anfangsstück von π ist, muss π nur auf entsprechend viele Dezimalstellen berechnet werden.

f_2: Für f_2 wissen wir nicht, ob es berechenbar ist. Um festzustellen, dass die Ziffernfolge in π vorkommt, müsste man π schrittweise genauer approximieren. Der Algorithmus würde stoppen, wenn die Ziffernfolge gefunden wurde. Aber was ist wenn die Zahl nicht in π vorkommt? Vielleicht gibt es aber einen (noch zu findenden) mathematischen Satz, der Aussagen über die in π vorkommenden Ziffernfolgen ermöglicht.

f_3: Hingegen ist f_3 berechenbar, denn $f_3 \equiv f_4$, mit

$$f_4(n) = \begin{cases} 1 & n \leq n_0 \\ 0 & \text{sonst} \end{cases}$$

wobei n_0 die maximale Anzahl von aufeinanderfolgenden 7ern in π ist (bzw. ∞). Hierbei ist es nicht von Bedeutung, wie die Zahl n_0 berechnet werden kann – wichtig ist nur, dass eine solche Zahl $n_0 \leq \infty$ existieren muss.

Zurück zur Frage: Was heisst "berechenbar"? Wir verstehen unter berechenbar:

Definition 2.1 *Turing-Berechenbarkeit:*
Es gibt eine Turingmaschine, die für alle Eingaben bin$(n_1)\#$*bin*$(n_2)\# \ldots \#$*bin*(n_k) *nach endlich vielen Schritten mit bin*$(f(n_1, \ldots n_k))$ *auf dem Band stoppt.*

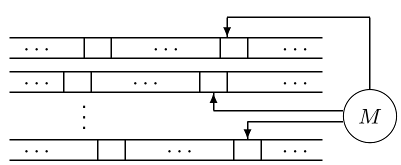
Hierbei steht bin(n) für die Binärdarstellung der Zahl n, das Zeichen $\#$ wird als Trennzeichen für die Eingabe verwendet.

Churche These: Die Turing-berechenbaren Funktionen entsprechen genau den intuitiv berechenbaren Funktionen.

2.2 Turing Berechenbarkeit

Bemerkung: Es gibt viele verschiedene Definitionen von "Turing-Maschine". Vom Standpunkt der Berechenbarkeit sind diese alle äquivalent, d.h. man kann sie gegenseitig simulieren.

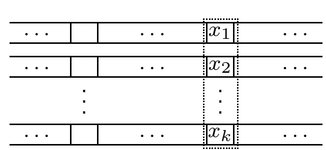
Beispiel 2.2
k-Band Turingmaschine



Wichtig: k Schreib-/Leseköpfe, die unabhängig voneinander bewegt werden können.

Satz 2.1 *Jede k-Band Turingmaschine kann man durch eine 1-Band Turingmaschine simulieren.*

Beweis: Beweisidee: Definiere als (geeignetes) Bandalphabet für 1-Band Turingmaschine $(\Gamma \times \{-, *\})^k$, wobei Γ das Bandalphabet der k-Band Turingmaschine ist:



Ein Zeichen $(x_1, x_2, \ldots, x_k)$ des neuen Bandes erhält man damit zunächst dadurch, dass man die “übereinanderstehenden” Zeichen der k Bänder zusammenfasst. Jedoch hat man damit noch keine Informationen über die Position der Lese-/Schreibköpfe der k-Band Turingmaschine gespeichert. Hierzu werden die beiden Symbole "*" und "-" verwendet. Bei der Konstruktion eines Zeichens der 1-Band Turingmaschine wird jeweils eins dieser Symbole nach jedem einzelnen Zeichen der k-Band Turingmaschine eingefügt. Das Symbol "*" besagt, dass sich der Lesekopf an dem Zeichen befindet. Das Symbol "-" besagt, dass sich der Lesekopf nicht an dem Zeichen befindet. Bei dem Zeichen $(x_1, -, x_2, *, x_3, -, \ldots, x_k, -)$ der 1-Band Turingmaschine befand sich in der zugrundeliegenden k-Band Turingmaschine der Lese-/Schreibkopf für das 2. Band am Zeichen x_2. Damit kann man einen Schritt auf der k-Band Turingmaschine durch eine Folge von Schritten auf der 1-Band Turingmaschine simulieren. □

Im folgendem wollen wir noch kurz auf den Zusammenhang mit einem “normalen” Algorithmus eingehen.

Beispiel 2.3
Berechnung von $m + n$:

```
x := m;
while n ≠ 0 do
{
    x := x + 1;
    n := n − 1;
}
return (x);
```

Idee: Wir verwenden 2 Bänder.

1.Schritt: Schreibe m auf das 1. Band, n auf das 2. Band.

2.Schritt: Schleife: addiere 1 auf dem 1. Band, subtrahiere 1 auf dem 2. Band. Solange bis 2. Band = 0.

2.3 LOOP-, WHILE- und GOTO-Berechenbarkeit

Ziel: Möglichst einfacher Ansatz den Berechenbarkeitsbegriff (Turing-Berechenbarkeit) mit Hilfe einer Programmiersprache zu definieren.

2.3.1 LOOP-berechenbar

LOOP-Programme sind wie folgt definiert:

Variablen: $x_1, x_2, x_3, \ldots \in \mathbb{N}_0$

Konstanten: 0, 1, 2, ...

Trennsymbole: ; :=

Operationszeichen: + -

Schlüsselworte: LOOP DO END

Eingabe-Konvention: Eingabe in $x_1, \ldots, x_n$

Ausgabe-Konvention: Ausgabe in x_0

Aufbau von LOOP-Programmen:

- $x_i := c,\ x_i := x_j + c,\ x_i := x_j - c$ sind LOOP-Programme. Die Interpretation dieser Ausdrücke erfolgt, wie üblich, mit der Einschränkung, dass $x_j - c$ als Null gewertet wird, falls $c > x_j$.

- Sind P_1, P_2 LOOP-Programme so ist auch $P_1; P_2$ ein LOOP-Programm.

  Interpretation: Führe erst P_1 und dann P_2 aus.

- Ist P LOOP-Programm, so ist auch LOOP x_i DO P END ein LOOP-Programm.

  Interpretation: Führe P genau $\langle$Wert von $x_i\rangle$-mal aus. **Achtung:** Änderungen von x_i im Innern von P haben *keinen* Einfluss auf die Anzahl Wiederholungen (wie z.B. bei einer FOR-Schleife in der Programmiersprache PASCAL).

Definition 2.2 *Eine Funktion f heißt LOOP-berechenbar genau dann, wenn es ein LOOP-Programm gibt, das f berechnet.*

LOOP-Programme können IF .. THEN .. ELSE .. END Konstrukte simulieren. Der Ausdruck IF $x = 0$ THEN A END kann durch folgendes Programm nachgebildet werden:

$y := 1$;
LOOP x DO $y := 0$ END;
LOOP y DO A END;

LOOP-berechenbare Funktion sind immer *total*, denn: LOOP-Programme *stoppen immer*. Damit stellt sich natürlich die Frage, ob alle totalen Funktionen LOOP-berechenbar sind. Die Antwort hierauf lautet allerdings: Nein! Dazu folgende Beispiele für nicht LOOP-berechenbare Funktionen:

Beispiel 2.4
Busy Beaver-Funktion (für LOOP-Programme), s. dazu Skript zur Vorlesung Informatik I, WS 1998/99 (Prof. Brauer) [1].

Beispiel 2.5
Ackermann-Funktion: $a : \mathbb{N}_0^2 \to \mathbb{N}$.

$$a(x,y) := \begin{cases} y+1 & \text{falls } x = 0 \\ a(x-1,1) & \text{falls } x \geq 1, y = 0 \\ a(x-1, a(x, y-1)) & \text{falls } x, y \geq 1 \end{cases}$$

Einige Eigenschaften der Ackermann-Funktion, die man per Induktion zeigen kann:

1. $a(1,y) = y+2 \quad \forall y, \;\; a(2,y) = 2y+3 \quad \forall y$
2. $y < a(x,y) \quad \forall x, y$
3. $a(x,y) < a(x,y+1) \quad \forall x, y$
4. $a(x,y+1) \leq a(x+1,y) \quad \forall x, y$
5. $a(x,y) < a(x+1,y) \quad \forall x, y$

Klar ist: Die Ackermannfunktion ist intuitiv berechenbar, hier einige Werte:

$$\begin{aligned} a(1,1) &= 3, \quad a(2,1) = 5, \quad a(3,1) = 13, \\ a(4,1) &= a(3,a(3,1)) = a(3,13) = \text{...maple steigt aus...} \\ a(3,k) &= a(2,a(3,k-1)) \geq 2 * a(3,k-1) \\ &\geq 2^2 * a(3,k-2) \geq ... \geq 2^k * a(3,0) \geq 2^k \\ a(4,1) &\geq 2^{13} \\ a(4,2) &= a(3,a(4,1)) \geq 2^{a(4,1)} \geq 2^{2^{13}} \\ a(4,3) &= a(3,a(4,2)) \geq 2^{a(4,2)} \geq 2^{2^{2^{13}}} \\ &\vdots \end{aligned}$$

Satz 2.2 *Die Ackermann-Funktion ist nicht LOOP-berechenbar.*

Lemma 2.1 *Sei P ein LOOP-Programm mit Variablen $x_1, \ldots, x_k$ und*

$$f_p(n) = \textit{max}\,\{\, \Sigma_{i=1}^{k} n'_i \mid \Sigma_{i=1}^{k} n_i \leq n \text{, wobei die } n'_i \text{ die Werte von } x_i \text{ nach Beendigung von } P \text{ sind und die } n_i \text{ die Startwerte von } x_i \text{ sind}\}$$

Dann gibt es ein $t \in \mathbb{N}$ mit $\forall n: \ f_p(n) < a(t, n)$.

Beweis: Beweis des Satzes 2.2:
Angenommen doch, sei P das zugehörige LOOP-Programm, das $g(n) := a(n, n)$ berechnet. Nach Definition von f_p gilt $g(n) \leq f_p(n)\ \forall n$. Wähle t mit $f_p(n) < a(t, n)\ \forall n$. Setze $n = t$:

$$f_p(t) < a(t, t) =_{Def.} g(t) \leq f_p(t)$$

$\Rightarrow$ Widerspruch! □

Beweis: Beweis des Lemmas 2.1 durch Induktion über den strukturellen Aufbau:

Induktionsanfang:
$P = x_i = x_j \pm c$ (o.E. $c = \{0, 1\}$). Es ist klar, dass

$$f_p(n) \leq 2n + 1 < a(2, n) = 2n + 3\ \forall n$$

$\Rightarrow t = 2$ tut es!

Induktionsschritt:
Hier gibt es 2 Fälle zu behandeln. Entweder ist $P = P_1; P_2$ oder $P =$ LOOP x_i DO P_i END .
1. Fall: $P = P_1; P_2$: Nach Induktionsannahme $\exists k_1, k_2$, so dass $\forall n \ \ \forall i \in \{1, 2\}$

$$f_p(n) < a(k_i, n)$$

$$\begin{aligned} f_p(n) &\leq f_{p_2}(f_{p_1}(n)) \\ &\leq f_{p_2}(a(k_1, n)) \\ &< a(k_2, a(k_1, n)) \quad \text{Setze } k_3 := max\{k_2, k_1 - 1\}. \\ &\leq a(k_3, a(k_3 + 1, n)) \quad \text{Monotonie} \\ &= a(k_3 + 1, n + 1) \\ &\leq a(k_3 + 2, n) \quad \text{Eigenschaft 4 der Ackermannfkt.} \end{aligned}$$

Hieraus erhält man: $t = k_3 + 2$ tut es!
2. Fall: $P =$ LOOP x_i DO Q END . Dieser Beweis kann ähnlich geführt werden, s. hierzu z.B. Schöning.

□

2.3.2 WHILE-berechenbar

Motivation: Die Ackermann Funktion ist intuitiv berechenbar, sie ist Turing-berechenbar, aber sie ist nicht LOOP-berechenbar. Dies zeigt: LOOP-Programme sind nicht mächtig genug.
WHILE-Programme sind wie folgt definiert:

Variablen: x_1, x_2, x_3, ...

Konstanten: 0, 1, 2, ...

Trennsymbole: ; := $\neq$

Operationszeichen: + -

Schlüsselworte: WHILE DO END

Aufbau von WHILE-Programmen:

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j - c$ sind WHILE-Programme.

  Die Interpretation dieser Ausdrücke erfolgt, wie üblich, mit der Einschränkung, dass $x_j - c$ als Null gewertet wird, falls $c > x_j$.

- Sind P_1, P_2 WHILE-Programme do ist auch $P_1; P_2$ ein WHILE-Programm.

  Interpretation: Führe erst P_1 und dann P_2 aus.

- Ist P ein WHILE-Programm so ist auch WHILE $x_i \neq 0$ DO P END ein WHILE-Programm.

  Interpretation: Führe P solange aus bis x_i den Wert Null hat. **Achtung:** Änderungen von x_i im Innern von P haben *einen* Einfluss auf die Anzahl Wiederholungen.

Lemma 2.2 *LOOP-Programme können durch WHILE-Programme simuliert werden.*

Beweis: Der Ausdruck LOOP x DO P END ist äquivalent zum Ausdruck $y := x$; WHILE $y \neq 0$ DO $y := y - 1$; P END wobei y eine Variable ist, die in P nicht verwendet wird. □

Satz 2.3 *Ist eine Funktion WHILE-berechenbar, so ist sie auch Turing-berechenbar.*

Beweis: Beweisidee: Spendiere für jede WHILE-Schleife ein zusätzliches Band, auf dem die zugehörige Schleifenvariable x_i gespeichert wird. □

Unser Ziel ist natürlich ein Satz der Form: Die Menge der Turing-berechenbaren Funktionen entspricht der Menge WHILE-berechenbaren Funktionen.

2.3.3 GOTO-berechenbar

Für GOTO-Programme gilt:

Variablen: $x_1, x_2, x_3, ...$

Konstanten: 0, 1, 2, ...

Trennsymbole: ; :=

Operationszeichen: + -

Schlüsselworte: IF THEN GOTO HALT

Aufbau von GOTO-Programmen, wobei hierbei die A_i Anweisungen und die M_i Marken (für Sprünge) sind :

$$M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$$

Als Anweisungen sind zugelassen:

- Wertzuweisungen: $x_i := x_j \pm c$,
- unbedingter Sprung: GOTO M_i
- bedingter Sprung: IF $x_i = c$ THEN GOTO M_i (wobei c eine Konstante ist)
- Stopanweisung: HALT

Satz 2.4
Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Beweis: Beweisidee: Ersetze jede WHILE-Schleife WHILE $x_i \neq 0$ DO P END durch folgenden Ausdruck:

M_1: IF $x_i = 0$ THEN GOTO M_2
P;
GOTO M_1
M_2: ...

□

Satz 2.5
Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Beweisidee: Gegeben sei das GOTO-Programm $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ Wir simulieren dies durch eine WHILE-Programm mit genau einer WHILE-Schleife:

$c := 1$;
WHILE $c \neq 0$ DO
IF $c = 1$ THEN A'_1 END;
IF $c = 2$ THEN A'_2 END;
⋮
IF $c = k$ THEN A'_k END;
END

wobei

$$A'_i := \begin{cases} x_j := x_l \pm b; c := c + 1 & \text{falls } A_i = x_j := x_l \pm b \\ c := \ell & \text{falls } A_i = \text{GOTO } M_\ell \\ \text{IF } x_j = b \text{ THEN } c := \ell & \\ \text{ELSE } c := c + 1 \text{ END} & \text{falls } A_i = \text{IF } x_j = b \text{ THEN GOTO } M_\ell \\ c := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

□

Satz 2.6 *Turing-berechenbar ⇒ GOTO-berechenbar.*

Beweis: Beweisidee:

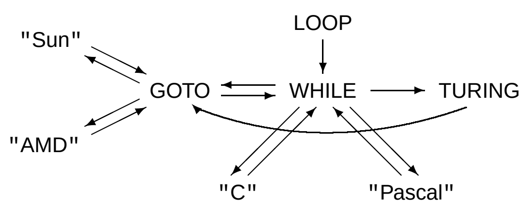
1. Konfiguration einer Turingmaschine (α, β, z) wird kodiert in den Variablen (x_α, x_β, x_z).
2. Umwandlung in ein GOTO-Programm:

$$\text{IF } (x_z = z) \wedge (\text{erstes Zeichen von } x_\beta = b) \text{ THEN } \{x_z := z; \ldots\}$$

□

2.3.4 Zusammenfassung

Im folgenden sei nocheinmal ein kurzer Überblick über die Beziehungen zwischen den einzelnen Berechenbarkeitsarten gegeben. Die Pfeile geben an, dass sich alle durch ihr Herkunftsobjekt berechnenbaren Funktionen auch durch ihr Zielobjekt berechnen lassen.



2.4 Primitiv rekursive und μ-rekursive Funktionen

Idee: Bisher haben wir ausgehend von Programmen versucht, den Begriff der *Berechenbarkeit* zu definieren. Nun wählen wir einen mathematischen Ansatz und versuchen, eine Klasse von *berechenbaren* Funktionen induktiv zu definieren. Am Ende dieses Abschnitts werden wir sehen, dass uns beide Ansätze auf ein- und denselben Berechenbarkeitsbegriff führen werden.

2.4.1 Primitiv rekursive Funktionen

Definition 2.3 *Die Klasse der primitiv rekursiven Funktionen (auf* $\mathbb{N}_0$*) ist induktiv wie folgt definiert:*

- Alle (beliebigstelligen) konstanten Funktionen sind primitiv rekursiv.

$$f(n_1, \ldots, n_m) = c \quad m \geq 1 \quad \forall n_i \in \mathbb{N}_0 \quad \forall i \in \mathbb{N} \qquad (\text{für alle } c \in \mathbb{N}_0)$$

- Die Nachfolgerfunktion $s(n)$ ist primitiv rekursiv.

$$s(n) = n + 1 \quad \forall n \in \mathbb{N}_0$$

- Alle Projektionen sind primitiv rekursiv.

$$f(n_1, \ldots, n_k) = n_j \quad \forall n_1, \ldots, n_k \in \mathbb{N}_0 \qquad (\text{für alle } j \in \{1, \ldots, k\})$$

  Dies soll bedeuten das wir uns ein beliebiges Argument der Funktion f herausgreifen können, um es in einer primitiv rekursiven Funktion zu verwenden.

- Die Komposition von primitiv rekursiven Funktion ist primitiv rekursiv. Seien f, g primitiv rekursiv dann ist auch: $h(n) = f(g(n))$ für alle $n \in \mathbb{N}_0$ eine primitiv rekursive Funktion. Gleichermaßen gilt für beliebigstellige Funktionen: Seien $f, g_1, \ldots g_k$ primitiv rekursive Funktionen, dann ist auch

$$f(g_1(n_1, \ldots, n_m), \ldots, g_k(n_1, \ldots, n_m))$$

  primitiv rekursiv.

- Jede Funktion, die durch *primitive Rekursion* aus primitiv rekursiven Funktionen entsteht, ist primitiv rekursiv. Seien f, g primitiv rekursiv, so ist die folgende Funktion primitiv rekursiv:

$$h(n, x_2, \ldots, x_k) = \begin{cases} f(x_2, \ldots, x_k) & \text{falls } n = 0 \\ g\left(h(n-1, x_2, \ldots, x_k), n-1, x_2, \ldots, x_k\right) & \text{falls } n > 0 \end{cases}$$

Beispiel 2.6

Addition: $\text{add}: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$$(x, y) \mapsto x + y$$

Diese Funktion ist primitiv rekursiv, denn:

$$\begin{aligned} \text{add}(0, x) &= x \\ \text{add}(n+1, x) &= s(\text{add}(n, x)) \end{aligned}$$

Rekursion: Addiere zunächst n und x und addiere dann 1 zum Ergebnis.

Multiplikation: $\text{mult}: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$$(x, y) \mapsto xy$$

Diese Funktion ist ebenso primitiv rekursiv, denn:

$$\begin{aligned} \text{mult}(0, x) &= 0 \\ \text{mult}(n+1, x) &= \text{add}(\text{mult}(n, x), x) \end{aligned}$$

Rekursion: Multipliziere zunächst n und x und addiere x zum Ergebnis.

Bemerkung: Es ist einsichtig, dass primitiv rekursive Funktionen intuitiv berechenbar sind. Damit stellt sich die Frage: Wie mächtig ist diese Klasse? Wir werden später sehen, dass die primitiv rekursiven Funktionen genau die LOOP-berechenbaren Funktionen sind.

Beispiel 2.7

(modifizierte) Vorgängerfunktion:

$$\begin{aligned} \bar{s}: \quad \mathbb{N}_0 &\to \mathbb{N}_0 \\ x &\mapsto \begin{cases} 0 & x = 0 \\ x - 1 & \text{sonst} \end{cases} \end{aligned}$$

primitiv berechenbar, denn

$$\begin{aligned} \bar{s}(0) &= 0 \\ \bar{s}(n+1) &= n \end{aligned}$$

(modifizierte) Subtraktion:

$$\begin{aligned} \text{sub}: \mathbb{N}_0 \times \mathbb{N}_0 &\to \mathbb{N}_0 \\ \text{(x,y)} &\mapsto \begin{cases} 0 & y \geq x \\ x - y & \text{sonst} \end{cases} \end{aligned}$$

primitiv berechenbar, denn

$$\begin{aligned} \text{sub}(x, 0) &= x \\ \text{sub}(x, y+1) &= \bar{s}(\text{sub}(x, y)) \end{aligned}$$

Definition 2.4 *Sei $P(x)$ ein Prädikat, d.h. ein logischer Ausdruck, der in Abhängigkeit von $x \in \mathbb{N}$ den Wert true oder false liefert. Dann können wir diesem Prädikat in natürlicher Weise eine sogenannte 0-1 Funktion $\hat{P} : \mathbb{N}_0 \to \{0, 1\}$ zuordnen, indem wir definieren, dass $\hat{P}(x) = 1$ genau dann, wenn $P(x) =$ true ist. Wir nennen $P(x)$ primitiv rekursiv genau dann, wenn $\hat{P}(x)$ primitiv rekursiv ist.*

Definition 2.5 *Beschränkter max-Operator: Zu einem Prädikat $P(x)$ definieren wir*

$$\begin{array}{rcl} q : \mathbb{N}_0 & \to & \mathbb{N}_0 \\ n & \mapsto & \begin{cases} 0 & \text{falls } \neg P(x) \text{ für alle } x \leq n \\ \max\{x \leq n \mid P(x)\} & \text{sonst} \end{cases} \end{array}$$

Dann gilt: Ist P primitiv rekursiv, so auch q, denn:

$$\begin{array}{rcl} q(0) & = & 0 \\ q(n+1) & = & \begin{cases} n+1 & \text{falls } P(n+1) \\ q(n) & \text{sonst} \end{cases} \\ & = & q(n) + \hat{P}(n+1) * (n+1-q(n)) \end{array}$$

Definition 2.6 *Beschränkter Existenzquantor: Zu einem Prädikat $P(x)$ definieren wir ein neues Prädikat $Q(n)$ mit: $Q(n)$ ist genau dann* true*, wenn ein $x \leq n$ existiert mit $P(x)$ ist* true.

Dann gilt: Ist P primitiv rekursiv, so auch Q, denn:

$$\begin{array}{rcl} \hat{Q}(0) & = & 0 \\ \hat{Q}(n+1) & = & \hat{P}(n+1) + \hat{Q}(n) - \hat{P}(n+1) * \hat{Q}(n) \end{array}$$

Beispiel 2.8
Der Binomialkoeffizient bin: $\mathbb{N}_0 \to \mathbb{N}_0$ mit $x \mapsto \binom{x}{2}$ ist primitiv berechenbar, denn es gilt: $\binom{x+1}{2} = \binom{x}{2} + x$.

Es gibt eine primitiv berechenbare Funktion, die $\mathbb{N}_0 \times \mathbb{N}_0$ *bijektiv* auf $\mathbb{N}_0$ abbildet.

	0	1	2	3	4	$\cdots$
0	0	1	3	6	10	
1	2	4	7	11		
2	5	8	12			
3	9	13				
4	14					
$\vdots$						

$$f(x, y) = \binom{x+y+1}{2} + x.$$

Der Beweis, dass f die oben dargestellte, bijektive Funktion ist, sei dem Leser als Übungsaufgabe überlassen. Analog gibt es eine primitiv berechenbare k-stellige Funktion, die $\mathbb{N}_0 \times \ldots \times \mathbb{N}_0$ *bijektiv* auf $\mathbb{N}_0$ abbildet. Wir bezeichnen diese Funktion mit $\langle n_1 \ldots, n_k \rangle$.

Sei f definiert wie oben. Dann sind auch die Umkehrfunktionen g, h von f, die durch

$$g(f(x,y)) = x, \qquad h(f(x,y)) = y, \qquad f(g(n), h(n)) = n$$

definiert sind primitiv rekursiv, denn:

$$\begin{array}{rcl} g(n) & = & \max\{x \leq n \mid \exists y \leq n : f(x,y) = n\} \\ h(n) & = & \max\{y \leq n \mid \exists x \leq n : f(x,y) = n\} \end{array}$$

(Es ist hierbei zu beachten, dass zunächst gezeigt werden muss, dass die Gleichheitsfunktion primitiv rekursiv ist. Dieser Beweis sei dem Leser als Übungsaufgabe überlassen.)
In gleicher Weise kann die Umkehrfunktionen von $\langle n_1 \ldots, n_k \rangle$ hergeleitet werden.

Satz 2.7 *f primitiv rekursiv $\Leftrightarrow$ f LOOP-berechenbar*

Beweis:
Wir zeigen zunächst "$\Leftarrow$": Sei also P ein LOOP-Programm, das f berechnet. P verwendet die Variablen $x_1, \ldots, x_k$, $k \geq n$, $f : \mathbb{N}_0^n \to \mathbb{N}_0$.
Zu zeigen: f ist primitv rekursiv.
Der Beweis erfolgt durch strukturelle Induktion über den Aufbau von P.

Induktionsverankerung: $P:\ x_i := x_j \pm c$
Wir zeigen: $\exists$ primiv rekursive Funktion

$$g_p(\underbrace{\langle a_1, \ldots, a_k \rangle}_{\substack{\text{Belegung der Variablen}\\ \text{bei Start von } P}}) \quad = \quad \underbrace{\langle b_1, \ldots, b_k \rangle}_{\substack{\text{Belegung der Variablen}\\ \text{bei Ende von } P}}$$

Für $P:\ x_i := x_j \pm c$ erhält man:

$$g_p(\langle a_1, \ldots, a_k \rangle) = \langle a_1, \ldots, a_{i-1}, a_j \pm c, a_{i+1}, \ldots, a_n \rangle$$

Induktionsschritt: Hier unterscheiden wir 2 Fälle:
1.Fall: Sei $P:\ Q; R$. Dann ist: $g_p(x) = g_R(g_Q(x))$
2.Fall: Sei nun $P:$ LOOP x_i DO Q END
IDEE: Definiere Funktion $h(n, x)$, die die Belegung der Variablen berechnet, wenn man mit Belegung x startet und dann Q genau n mal ausführt. Dann ist:

$$\begin{aligned} h(0, x) &= x \\ h(n+1, x) &= g_Q(h(n, x)) \end{aligned}$$

und damit $g_p(x) = h(d_i(x), x)$, wobei d_i die i-te Unkehrfunktion von $\langle x_1, \ldots, x_k \rangle$, also $d_i(\langle x_1, \ldots, x_k \rangle) = x_i$.

Die Richtung "$\Rightarrow$" wird durch strukturelle Induktion über den Aufbau von f gezeigt. Dies sei dem Leser als Übungsaufgabe überlassen. $\square$

Korollar 2.1 *Die Ackermann-Funktion ist* nicht *primitiv rekursiv.*

Hier rufe man sich die Definition der Ackermann-Funktion aus Beispiel 2.5 von Seite 42 in Erinnerung. Intuitiv ist vorstellbar: Diese Funktion wächst so schnell, dass sie nicht mehr mittels primitiver Rekursion dargestellt werden kann.

2.4.2 μ-rekursive Funktionen

Idee: Sei $f : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann liefert der sogenannte *μ-Operator* die kleinste natürliche Zahl für die f den Wert Null annimmt.
Motivation: Der μ-Operator wird uns erlauben, "vorab" die notwendige Anzahl Durchläufe einer while-Schleife zu bestimmen bis die Laufvariable Null wird.

Definition 2.7 *Sei f eine (nicht notwendigerweise totale) $k+1$-stellige Funktion. Die durch Anwendung des μ-Operators entstehende Funktion f_μ ist definiert durch:*

$$\begin{aligned} f_\mu : \mathbb{N}_0^k &\to \mathbb{N}_0 \\ (x_1, \ldots, x_k) &\mapsto \begin{cases} \min\{n \in N \mid f(n, x_1, \ldots, x_k) = 0\} & \textit{falls } f(m, x_1, \ldots, x_k) \\ & \textit{definiert für alle } m \leq n \\ \perp \ \textit{(undefiniert)} & \textit{sonst} \end{cases} \end{aligned}$$

Definition 2.8 *Die Klasse der μ-rekursiven Funktionen ist die kleinste Klasse von (nicht notwendigerweise totalen) Funktionen, die die Basisfunktionen (konstante Funktionen, Nachfolgerfunktion, Projektionen) enthält und alle Funktionen, die man hieraus durch (evtl. wiederholte) Anwendung von Komposition, primitiver Rekursion und/oder des μ-Operators gewinnen kann.*

Satz 2.8 f *μ-rekursiv* $\quad\Leftrightarrow\quad$ f *WHILE-berechenbar*

2.5 Entscheidbarkeit, Halteproblem

2.5.1 Charakteristische Funktionen

Ziel: Wir wollen im folgenden Abschnitt zeigen, dass es keinen Algorithmus geben kann, der als Eingabe ein Programm P und Daten x erhält und (für jedes solches Paar (P, x)!) entscheidet, ob P hält, wenn es mit Eingabe x gestartet wird.
Dazu definieren wir zunächst die charakteristische Funktion bzw. die semi-charakteristische Funktion von $A \subseteq \Sigma^*$ wie folgt:

Definition 2.9 *Sei $A \subseteq \Sigma^*$. Die charakteristische Funktion von A ist definiert als*

$$\begin{array}{rcl} \chi_A : \Sigma^* & \rightarrow & \{0, 1\} \\ w & \mapsto & \begin{cases} 1 & \textit{falls } w \in A \\ 0 & \textit{sonst} \end{cases} \end{array}$$

Definition 2.10 *Sei $A \subseteq \Sigma^*$. Die semi-charakteristische Funktion von A ist definiert als*

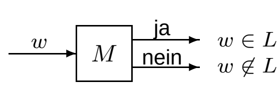
$$\begin{array}{rcl} \chi'_A : \Sigma^* & \rightarrow & \{0, 1\} \\ w & \mapsto & \begin{cases} 1 & \textit{falls } w \in A \\ \textit{undefiniert} & \textit{sonst} \end{cases} \end{array}$$

2.5.2 Entscheidbare Sprachen

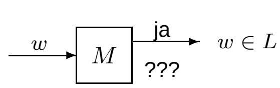
Mit Hilfe der Definitionen aus Abschnitt 2.5.1 zur charakteristischen Funktion können wir nun für Sprachen definieren:

Definition 2.11 *Eine Sprache $A \subseteq \Sigma^*$ heißt entscheidbar, falls die charakteristische Funktion χ_A berechenbar ist.*

Zur Erinnerung: "Berechenbar" heisst für uns immer: Turing-berechenbar (und damit auch WHILE-berechenbar, μ-rekursiv, ...).



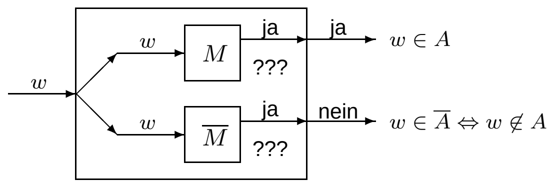
Definition 2.12 *Eine Sprache $A \subseteq \Sigma^*$ heißt semi-entscheidbar, falls die semi-charakteristische Funktion χ'_A berechenbar ist.*



Es ist klar, dass eine entscheidbare Funktion auch semi-entscheidbar ist. Umgekehrt ist es allerdings nicht klar, ob es möglich ist, für eine semi-entscheibare Sprache einen Algorithmus zu bauen, der diese Sprache entscheidet, der also für den Fall $w \notin L$ "Nein" ausgibt und nicht in eine Endlosschleife gerät. Wir werden noch sehen, dass dies *nicht* möglich ist.

Satz 2.9 *Eine Sprache $A \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl A als auch $\overline{A} := \Sigma^* \setminus A$ semi-entscheidbar sind.*

Beweis:
Die Richtung "⇒”" ist klar.
Es bleibt also "⇐" zu zeigen.
Wenn A und $\overline{A}$ beide semi-entscheidbar sind, dann gibt es WHILE-Programme, die für ein Wort w genau dann stoppen, wenn $w \in A$ bzw. wenn $w \notin A$. Führt man daher diese beiden Programme "parallel" aus (genauer: jeweils abwechselnd einen Schritt des einen Programms und dann einen Schritt des anderen Programms) erhält man ein Programm, das A *entscheidet*.



□

2.5.3 Rekursiv aufzählbare Sprachen

Definition 2.13 *Eine Sprache $A \subseteq \Sigma^*$ heisst rekursiv aufzählbar, falls es eine berechenbare Funktion $f : \mathbb{N}_0 \to \Sigma^*$ gibt, so dass*

$$A = \{f(0), f(1), f(2), \ldots, \}$$

Mit anderen Worten: Eine Sprache heisst dann rekursiv aufzählbar, falls es eine berechenbare Funktion f (bzw. einen Algorithmus) gibt, die (der) die Worte der Sprache A "aufzählt".
Hiezu folgende Beispiele:

Beispiel 2.9

1. Σ^* mit $\Sigma = \{0, 1\}$ ist rekursiv aufzählbar. Betrachte dazu die folgende Funktion $f(n)$:

$$\text{alle einstelligen} \left\{ \begin{array}{lcl} f(0) & = & 0 \\ f(1) & = & 1 \end{array} \right.$$

$$\text{alle zweistelligen} \left\{ \begin{array}{lcl} f(2) & = & 00 \\ f(3) & = & 10 \\ f(4) & = & 01 \\ f(5) & = & 11 \end{array} \right.$$

$$\text{alle dreistelligen} \left\{ \begin{array}{lcl} f(6) & = & 000 \\ \vdots & \vdots & \vdots \end{array} \right.$$

   Eine weitere Möglichkeit, eine Funktion $f_1(n)$ anzugeben, die alle Wörter aufzählt, wäre: $f_1(n) =$ Binärkodierung von $n + 2$ *ohne* führende 1. $f_1(0) = 1 \mid 0, f_1(1) = 1 \mid 1, f_1(2) = 1 \mid 00 \ldots$

2. $L_{TM} = \{w \in \{0, 1\}^* \mid w \text{ ist Codierung einer Turing-Maschine}\}$ ist rekursiv aufzählbar.
   Berechnung von $f(n)$: Probiere rekursiv (vgl. dieses Beispiel 2.9, Teil 1) alle Wörter über $\{0, 1\}$ aus und teste jeweils, ob dies die Codierung einer Turing-Maschine ist – solange bis die n-te Turing-Maschine gefunden wurde.

Satz 2.10 *Eine Sprache $A \subseteq \Sigma^*$ ist genau dann semi-entscheidbar, wenn sie rekursiv aufzählbar ist.*

Beweis:
Wir zeigen zunächst "$\Leftarrow$":
Betrachte den folgenden Algorithmus, der eine berechenbare Funktion f verwendet, die A aufzählt:

Lese die Eingabe $w \in \Sigma^*$; x :=0;
do forever
  if $f(x) = w$ **then** { **return** ("Ja"); **halt**; }
  $x := x + 1$;
end;

Nun wird "$\Rightarrow$" gezeigt:
Betrachte den folgenden Algorithmus, der die Aufzählungsfunktion f berechnet und dabei ein WHILE-Programm P simuliert, das die semi-charakteristische Funktion χ'_A berechnet:

Lese die Eingabe $n \in \mathbb{N}$.
$count := 0$; $k := 0$;
repeat
  $k := k + 1$;
  if P stoppt bei Eingabe $word(x(k))$ in genau $y(k)$ Schritten **then**
    $count := count + 1$;
until $count = n$
return $(word(x(k)))$

Hierbei sind $x(k)$ und $y(k)$ die Umkehrfunktionen der Funktion

$$f(x, y) := \binom{x + y + 1}{2} + x$$

und $word(n)$ eine Funktion, die einer natürlichen Zahl n das n-te Wort von Σ^* zuordnet.
□

Anmerkung zu obigem Beweis:
Da die Sprache A semi-entscheidbar ist, müssen wir die Anzahl der Schritte für das Programm P bereits im voraus festlegen – P könnte sonst nie stoppen. Um dennoch alle Schrittzahlen ($y(k)$) für alle Worte n ($n = x(k)$) "durchzuprobieren", verwenden wir die Umkehrfunktionen $x(k)$ und $y(k)$ der Funktion $f(x, y) = \binom{x+y+1}{2} + x$, die wir bereits im Beispiel 2.8 auf Seite 48 als bijektive Abbildung von $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ kennengelernt haben.

Anzahl der Schritte P

		0	1	2	3	4	$\cdots$
w_1	0	0	1	3	6	10	
w_2	1	2	4	7	11		
w_3	2	5	8	12			
w_4	3	9	13				
w_5	4	14					
	$\vdots$						

2.5.4 Halteproblem

Definition 2.14 *Unter dem speziellen Halteproblem H_S versteht man die folgende Sprache:*

$$H_S = \{w \in \{0,1\}^* \mid M_w \textit{ angesetzt auf } w \textit{ hält}\}$$

Notation: M_w bezeichnet die w-te Turing-Maschine, wobei w als Binärzahl interpretiert wird.
Betrachten wir folgende Tabelle, in der alle Turing-Maschinen M_i nach rechts und alle Wörter w_i über $\{0,1\}^*$ nach unten aufgelistet werden. Wir tragen in die Tabelle JA ein, falls die Turingmaschine M_i der jeweiligen Spalte i das Wort w_j der Zeile j akzeptiert. Ansonsten tragen wir NEIN ein.

alle Turing-Maschinen M_i

	M_0	M_1	M_2	M_3	M_4	$\cdots$
w_0	ja	nein	ja	ja	$\cdots$	
w_1	nein	nein	ja	ja	$\cdots$	
w_2	$\vdots$	$\vdots$	$\ddots$			
$\vdots$						

Wir definieren uns nun folgende Sprache:

$$\begin{aligned} L_d &= \{w_i \mid M_i \text{ akzeptiert } w_i \underline{\text{nicht}}\} \\ &= \{w \mid \text{ in der Diagonale der Zeile, die dem Wort } w \text{ entspricht, steht NEIN}\} \end{aligned}$$

Lemma 2.3 *L_d ist nicht rekursiv aufzählbar.*

Beweis: Angenommen doch. Dann gibt es eine Turingmaschine M mit $L(M) = L_d$. Da die obere Zeile alle Turingmaschinen erhält, kommt M in dieser Aufzählung aller Turingmaschinen vor, d.h. es gibt ein i_0 mit $M_{i_0} = M$.
Betrachte nun ein Wort w_{i_0}. Wann gilt: $w_{i_0} \in L_d$?

$$w_{i_0} \in L_d \underset{\text{Def. von } L_d}{\iff} M_{i_0} \text{ akzeptiert } w_{i_0} \text{ nicht}$$

Andererseits:

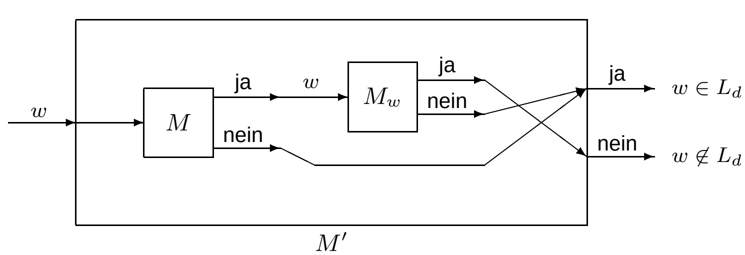
$$w_{i_0} \in L_d \underset{\text{Def. von } M}{\iff} M = M_{i_0} \text{ akzeptiert } w_{i_0}$$

$\Longrightarrow$ Widerspruch! $\square$

Korollar 2.2 *L_d ist nicht entscheidbar.*

Satz 2.11 *H_S ist nicht entscheidbar.*

Beweis: Angenommen doch. Sei M eine Turingmaschine, die H_S entscheidet. Wir konstruieren daraus eine Turingmaschine M', die L_d entscheidet.



$\Longrightarrow$ Widerspruch! $\square$

Definition 2.15 *Unter dem (allgemeinen) Halteproblem H versteht man die Sprache*

$$H = \{w\#x \in \{0,1\}^* \mid M_w \text{ angesetzt auf } x \text{ hält}\}$$

($\#$ wird hierbei als Trennzeichen verwendet.)

Satz 2.12 *Das allgemeine Halteproblem ist nicht entscheidbar.*

Damit sind wir beim Ziel dieses Abschnitts angekommen, denn obiger Satz bedeutet mit anderen Worten: Es kann keinen Algorithmus geben, der als Eingabe ein Programm P und Daten x erhält und (für jedes solche Paar (P,x)!) entscheidet, ob P hält, wenn es mit Eingabe x gestartet wird.

Kapitel 3

Algorithmen und Datenstrukturen

3.1 Analyse von Algorithmen

Idee: Bestimme die Ressourcen, die ein Algorithmus benötigt – als Funktion der Eingabelänge n – im allgemeinen nur bis auf eine (multiplikative) Konstante und "kleinere" Summanden genau (O-Notation).
Folgende Ressourcen sind hierbei zu betrachten:

- Laufzeit
- Speicherplatz
- Anzahl Prozessoren
- ...

In dieser Vorlesung wird von den angegebenen Ressourcen (fast ausschliesslich) die Laufzeit der Algorithmen analysiert.

Beispiel 3.1
Faktultätsfunktion $fak(n)$ berechnet $n!$

```
x := 1;
for i = 2 to n do
    x = x * i;
return (x);
```

Für diesen Algorithmus soll nun die Laufzeit bestimmt werden.
Vorschlag: Laufzeit ist $O(n)$.
Es ist klar, dass die Anzahl arithmetischer Operationen $O(n)$ ist.
ABER: Die Anzahl Bits, die ausgegeben werden, ist $\lceil \log_{10} n! \rceil = \Omega(n \log n)$ – Abschätzung mit Hilfe der Stirlingschen Approximationsformel $n! = \Omega\left(\left(\frac{n}{e}\right)^n\right)$.

Aus obigem Beispiel wird klar, dass unterschiedliche Betrachtungsweisen zu verschiedenen Ergebnissen für die Zeitkomplexität eines Algorithmus führen können. Es ist deshalb sinnvoll, im weiteren formal festzulegen, wie die Zeitkomplexität eines Algorithmus bestimmt werden soll.

3.1.1 Referenzmaschine

Wir wählen als Referenzmaschine:

WHILE-Maschine (also eine Maschine, die WHILE-Programme verarbeiten kann) erweitert durch

- IF ... THEN ... ELSE
- Multiplikation und Division
- Verarbeitung von rationalen Zahlen (d.h. Brüchen $\frac{n}{m}$ mit $n, m \in \mathbb{Z}$) möglich
- indirekte Adressierung
- ... $\langle$ gegebenenfalls weitere arithmetische Operationen wie $\sqrt{n}, \sin n, \ldots$ $\rangle$

3.1.2 Zeitkomplexität

uniformes Kostenmaß: $\#$ Operationen
entspricht Wortlänge der zu verarbeitenden Zahlen unendlich (d.h. beliebig grosse Zahlen können verarbeitet werden)

logarithmisches Kostenmaß: $\#$ Bit-Operationen
entspricht Wortlänge ist 1 (d.h. die Verarbeitung einzelner Bits wird betrachtet)

Welches Kostenmaß soll nun verwendet werden? Darüber soll folgende Faustregel Auskunft geben.
Faustregel: Man verwendet immer das uniforme Kostenmaß, falls sichergestellt ist, dass die größte, vom Algorithmus berechnete Zahl polynomiell in der Eingabegröße ist.

3.1.3 Worst Case Analyse

Sei $\mathcal{A}$ ein Algorithmus. Dann sei

$$T_{\mathcal{A}}(x) := \text{Laufzeit von } \mathcal{A} \text{ bei Eingabe } x$$

Im allgemeinen ist obige Funktion viel zu aufwendig zu berechnen – und (meist) auch nicht aussagekräftig. Interessanter jedoch ist eine Aussage der Form:

$$T_{\mathcal{A}}(n) = \underset{|x| = n}{max} \; T_{\mathcal{A}}(x) \text{ (= maximale Laufzeit bei Eingabelänge } n\text{)}$$

3.1.4 Average Case Analyse

$$T_{\mathcal{A}}^{ave}(n) = \frac{\sum_{x \mid |x|=n} T_{\mathcal{A}}(x)}{|\{x \mid |x| = n\}|}$$

Bemerkung: Wir werden Laufzeiten $T_{\mathcal{A}}(n)$ nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, etc. spielen hierbei nur eine untergeordnete Rolle.

3.2 Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n unsortierten Elementen enthält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im folgenden werden wir stets davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen.

Wir betrachten die Sortierverfahren unter dem Aspekt der Laufzeitanalyse. Dazu folgende **Bemerkung** vorab:
Bei den zu betrachtenden Verfahren (Selection-Sort, Insertion-Sort, Merge-Sort, Quick-Sort, Heap-Sort) gilt meist:

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche})$$

3.2.1 Selection-Sort

Der Algorithmus Selection-Sort arbeitet nach folgendem Prinzip:
Gegeben sei eine Folge $a_1, \ldots, a_n$ von unsortierten Elementen. Im ersten Schritt wird das größte Element der Folge bestimmt und an die n-te Position gesetzt (aufsteigende Sortierung vorausgesetzt). Im zweiten Schritt wiederholt man das Verfahren mit der um eins verkürzten Folge (ohne das letzte Element) usw. Man erhält dann nach n Schritten eine sortierte Folge.
Wir beweisen folgenden Satz:

Satz 3.1 *Selectionsort benötigt zum Sortieren von n Elementen genau $\binom{n}{2}$ Vergleiche.*

Beweis: Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Arrays) ergibt sich beim Selection-Sort zu:

$$\sum_{i=0}^{n-2} (n-1-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}$$

□

Damit ist die Laufzeit von Selection-Sort $O(n^2)$.

3.2.2 Insertion-Sort

Das Sortierverfahren Insertion-Sort arbeitet nach folgenden Prinzip:
Sei $a_1, \ldots, a_n$ eine unsortierte Folge von Elementen. Unter der Annahmen, dass die Folge $a_1, \ldots, a_{i-1}$ bereits sortiert ist, wird das i-te Element a_i an der richtigen Stelle im Anfangsstück eingefügt, wodurch man eine sortierte Teilfolge der Länge i erhält. i läuft hierbei von 1 bis n.
Die Stelle, an der das i-te Element einfügt werden muss, kann hierbei z.B. durch binäre Suche ermittelt werden.

Satz 3.2 *Insertion-Sort benötigt zum Sortieren von n Elementen maximal $\binom{n}{2}$ Vergleiche.*

Satz 3.3 *Der auf der binären Suche basierende Insertionsort (beim Einsortieren des nächsten Elementes in den bereits sortierten Teil) benötigt zum Sortieren von n Elementen maximal $n\lceil\log(n+1)\rceil$ Vergleiche.*

Beweis: Hierzu sehen wir uns zunächst den Fall $n = 3$ als Beispiel an. Man überlegt sich leicht, dass für $n = 3$ insgesamt 2 Vergleiche benötigt werden.
Im allgemeinen Fall ergibt sich für die Anzahl B_n der Vergleiche bei n Elementen folgende Rekursionsgleichung:

$$\begin{aligned} B_{2k+1} &= 1 + B_k \\ B_{2k} &= 1 + B_k \end{aligned}$$

Löst man diese Rekursionsgleichung (z.B. mit Hilfe von MAPLE), so erhält man $B_n = \lceil\log_2(n+1)\rceil$
Damit benötigt Insertion-Sort $\sum_{i=0}^{n-1}\lceil\log(i+1)\rceil \approx n\log n$ viele Vergleiche.

$\square$

Achtung: Die Laufzeit von Insertion-Sort ist dennoch $O(n^2)$. Der Grund hierfür ist folgender: Verwendet man z.B. ein Array als Datenstruktur für Speicherung der zu sortierenden Elemente, so ist bei jeder INSERT-Operation zusätzlicher Aufwand für die Array-Reorganisation nötig. Verwendet man andererseits z.B. eine doppelt verkettete Liste als Datenstrukur für die Speicherung der Elemente, so werden zwar die INSERT-Operationen vereinfacht, andererseits ist auf einer solchen Liste die binäre Suche mit einem höheren Aufwand verbunden.

3.2.3 Merge-Sort

Beim Merge-Sort wird eine unsortierte Folge von Zahlen zunächst rekursiv so lange in Teilfolgen halber Länge zerlegt bis die Länge einer Teilfolge 1 ist. Jeweils 2 dieser Teilfolgen werden dann miteinander verschmolzen, so dass eine sortierte Teilfolge doppelter Länge entsteht. Dies wird fortgesetzt, bis die gesamte Folge sortiert ist.

Satz 3.4 *Die rekursive Version von Mergesort sortiert ein Feld der Länge n mit maximal $n \cdot \lceil \log(n) \rceil$ Vergleichen.*

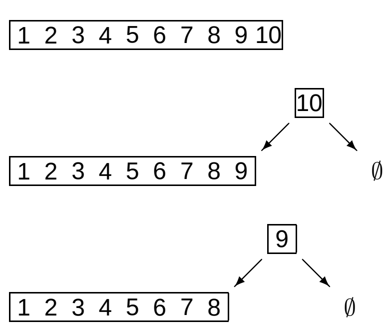
Beweis: s. z.B. Diskrete Strukturen, Band 1 von Prof. Dr. Steger, [7] $\square$

3.2.4 Quick-Sort

Beim Quick-Sort Verfahren wird in jedem Schritt ein Element x der zu sortierenden Folge als Pivot-Element ausgewählt. Wie diese Auswahl getroffen wird, sei hierbei zunächst noch nicht festgelegt (s. hierzu die auf Seite 59 aufgezählten Auswahlmöglichkeiten). Dann wird die zu sortierende Folge so umgeordnet, dass eine Teilfolge links von x entsteht, in der alle Werte der Elemente nicht größer als x sind und ebenso eine Teilfolge rechts von x, in der alle Werte nicht kleiner als das Pivot-Element x sind. Diese Teilfolgen werden dann selbst wieder nach dem gleichen Verfahren rekursiv zerlegt und umsortiert (Quick-Sort zählt also zu den sogenannten *Divide-and-Conquer Verfahren*). Dies geschieht jeweils solange, bis die Teilfolgen die Länge 1 besitzen und damit bereits sortiert sind, so dass man am Ende eine vollständig sortierte Folge erhält.

Satz 3.5 *Quicksort benötigt zum Sortieren eines Feldes der Länge n maximal $\binom{n}{2}$ viele Vergleiche.*

Beispiel 3.2
Besonders ungünstig ist es zum Beispiel wenn man Quick-Sort auf eine schon sortierte Liste ansetzt und dann das erste bzw. letzte Element als Pivot-Element wählt. In diesem Fall läuft das Divide-and-Conquer Verfahren komplett ins Leere, da eine der entstehenden Teilfolgen leer ist und die andere alle restlichen Elemente enthält.



In diesem Fall benötigt Quicksort $\sum_{i=1}^{n} n - i = \binom{n}{2}$ Vergleiche.

Satz 3.6 *Quicksort benötigt zum Sortieren eines Feldes der Länge* n durchschnittlich *nur* $2\ln(2)\, n \log(n) + O(n)$ *viele Vergleiche.*

Beweis: Zum Beweis der beiden obigen Sätze sei auf die detaillierte Analyse von Quicksort im die Vorlesung Diskrete Strukturen II verwiesen. □

Entscheidend für die Laufzeit von Quick-Sort ist hierbei eine "gute" Wahl des Pivotelements. Was aber ist eine gute Wahl?

1. Nehme stets das letzte Element der Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays

2. Median-of-3 Verfahren: wählen den Median (das mittlere Element) des ersten, mittleren, letzten Elements des Array
   Nachteil: s. hierzu Übungsblatt 8

3. zufälliges Pivotelement
   Vorteil: besseres Verhalten bei sortierten Arrays
   Nachteil: zusätzlicher Aufwand für die Randomisierung

3.2.5 Heap-Sort

Zunächst wollen wir definieren, was wir unter einem Heap verstehen:

Definition 3.1 *Definition eines Heaps:*

- *Alle inneren Knoten bis auf maximal einen haben genau zwei Kinder.*
- *Alle Knoten mit weniger als zwei Kindern (insbesondere also die Blätter) befinden sich auf den beiden Leveln größter Tiefe.*
- *Die Blätter im größten Level des Baumes sind von links nach rechts aufgefüllt.*
- *Für jeden Knoten gilt: alle Nachfolger haben einen höchstens gleich großen Schlüssel.*

Der Algorithmus HEAPSORT untergliedert sich ein 2 Phasen. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap nach obiger Definition aufgebaut. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird jeweils das größte Element entnommen (das ja an der Wurzel steht), diese Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften werden anschließend wieder hergestellt. Im folgenden wollen wir die dafür benötigten Teil-Algorithmen entwerfen und diese dann zu einem kompletten HEAPSORT-Algorithmus zusammenführen.
Betrachten wir nun zunächst den Algorithmus REHEAP(h) zum Einfügen der Wurzel in einen ansonsten korrekt sortierten Heap:

Sei v die Wurzel des Heaps h;
while (Heap-Eigenschaft in v nicht erfüllt)
    Sei v^* das Kind von v mit dem größeren Schlüssel;
    Vertausche die Schlüssel in v und v^* und setzte $v = v^*$;

Als nächstes benötigen wir noch einen Algorithmus CREATEHEAP(h) zum Erstellen eines Heaps für n beliebige Elemente, wobei der Baum h der Eingabe alle Eigenschaften eines Heaps bis auf die Sortiereigenschaft erfüllt:

for ℓ := Tiefe des Baumes **down to** 1 **do**
  for each Knoten v auf Level ℓ **do**
    REHEAP(Baum H mit Wurzel v);

Wenn wir am Ende die sortierte Folge dem Heap entnehmen wollen, brauchen wir einen Algorithmus DELETEMAX(h) zum Löschen der Wurzel:

Sei r die Wurzel des Heaps h und sei k der in r gespeicherte Schlüssel;
Sei ℓ das rechteste Blatt im untersten Level;
Kopiere den Schlüssel in ℓ in die Wurzel r;
Lösche das Blatt ℓ und dekrementiere $heap\_size$;
REHEAP(h);

Damit ergibt sich unser Algorithmus HEAPSORT (sortiert absteigend) zu:

CREATEHEAP(h);
while h nicht leer **do**
  Gebe Schlüssel der Wurzel aus;
  DELETEMAX(h);

Satz 3.7 *Mittels Heapsort kann ein Feld der Länge n mit höchstens $2n\log(n)+o(n)$ vielen Vergleichen sortiert werden.*

Beweis: Wir führen zunächst eine Laufzeitanalyse der einzelnen vorgestellten Teilalgorithmen durch, um danach die die Gesamtlaufzeit des Algorithmus angeben zu können. Dazu betrachten wir einen Heap mit l Leveln. Dieser Heap verfügt höchstens $2^l - 1$ Knoten.

REHEAP:

$$\text{\# Vergleiche} \quad \leq 2 \quad \cdot (l-1)$$

CREATEHEAP:

$$\begin{aligned}
\text{\# Vergleiche} \quad &\leq \underbrace{2^{l-1}\cdot 0}_{\text{Level } l} + \underbrace{2^{l-2}\cdot 2\cdot 1}_{\text{Level } l-1} + \ldots + \underbrace{2\cdot 2\cdot (l-2)}_{\text{Level 2}} + \underbrace{1\cdot 2\cdot (l-1)}_{\text{Level 1}} \\
&= \sum_{i=0}^{l-1} 2^i \cdot 2 \cdot (l-1-i) \\
&= 2(l-1)\cdot \underbrace{\sum_{i=0}^{l-1} 2^i}_{=2^l-1} - 2\cdot \underbrace{\sum_{i=0}^{l-1} i2^i}_{=(l-2)2^l+2} \\
&= \ldots \\
&= 2\cdot 2^l - 2l - 2
\end{aligned}$$

DELETEMAX: DELETEMAX verhält sich von der Laufzeit genaso wie REHEAP mit einem Element weniger.

HEAPSORT: Die Anzahl der Vergleiche ist kleiner als die Anzahl der Verlgeiche bei CREATE-HEAP addiert mit der Summe der Vergleiche bei allen DELETEMAX.

Wir benötigen also noch die Gesamtsumme bei DELETEMAX. Dazu hilft uns folgende Überlegung: Spätestens nach dem Löschen von 2^{l-1} vielen Elementen nimmt die Anzahl der Levels des Heap um 1 ab, nach weiteren 2^{l-2} Elementen wieder um 1, usw. Damit gilt

$$\begin{aligned}
\#\text{ Vergleiche} &\leq \underbrace{2^{l-1}\cdot 2(l-1)}_{\substack{\text{die ersten } 2^{l-1} \text{ Elemente}\\ \rightsquigarrow \text{ReHeap für Heap mit}\\ l \text{ Leveln}}} + \underbrace{2^{l-2}\cdot 2(l-2)}_{\substack{\text{die nächsten } 2^{l-2} \text{ Ele-}\\ \text{mente } \rightsquigarrow \text{ ReHeap für}\\ \text{Heap mit } l-1 \text{ Leveln}}} + \ldots + 2\cdot(2\cdot 1) \\
&= 2\cdot \underbrace{\sum_{i=1}^{l-1} i2^i}_{=(l-2)2^l+2}
\end{aligned}$$

Damit ergibt sich die Anzahl der Vergleiche bei Heap-Sort zu:

$$\begin{aligned}
\#\text{ Vergleiche} &\leq 2\cdot 2^l - 2\cdot l - 2 + 2\cdot((l-2)\cdot 2^l + 2) \\
&= 2l\cdot 2^l - 2\cdot 2^l - 2l + 2
\end{aligned}$$

Nachdem wir den Heap-Sort Algorithmus auf einem n-elementigen Array ausführen, ergibt sich die Höhe l zu:

$$l = \lceil \log_2(n+1) \rceil$$

Damit erhalten wir die Anzahl der Vergleiche bei n Elementen nun zu:

$$2n\log_2 n + O(n)$$

□

Bemerkungen:

- Mit einer von Carlssons beschriebenen Variante von Heapsort kann man ein Feld der Länge n mit maximal $n\log(n) + O(n\log\log(n))$ vielen Vergleichen sortieren.
- HeapSort ist ein sogenanntes in-situ Verfahren, d.h es benötigt nur konstant viele zusätzliche Speicherplätze.

3.2.6 Vergleichsbasierte Sortierverfahren

Wir wollen in diesem Abschnitt eine generelle Aussage über die Laufzeit von vergleichsbasierten Sortieralgorithmen treffen:

Satz 3.8 Jedes *vergleichsbasierte Sortierverfahren hat Laufzeit* $\Omega(n\log n)$, *und es benötigt* $n\log_2 n + O(n)$ *viele Vergleiche.*

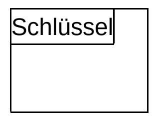
Beweis: Der Beweis zu diesem Satz kann einer Tutoraufgabe des Übungsblatts 8 zur Vorlesung "Einführung in die Informatik IV" von Prof. Dr. Steger, Sommersemester 2000, entnommen werden. □

3.2.7 Bucket-Sort

Bei Bucket-Sort handelt es sich um ein *nicht* vergleichsbasiertes Sortierverfahren. Hier werden z.B. Zahlen $a_1 \ldots a_n$ aus dem Bereich $\{1\ldots n\}$ dadurch sortiert, dass sie in durchnummerierte "Eimer" $1\ldots n$ geworfen werden. Die Inhalte dieser Eimer werden dann nacheinander ausgegeben, man erhält eine sortierte Folge.
Die Laufzeit von Bucket-Sort beträgt also $O(n)$, allerdings muss gewährleistet sein, dass die Zahlen aus dem Bereich $\{1\ldots n\}$ kommen. Da dies bei Buchstaben sehr einfach zu gewähleisten ist (man denke entsprechende Codierungen, z.B. Unicode) funktioniert dieses Sortierverfahren für Wörter sehr gut (vergleiche jeweils die ersten Buchstaben, dann die zweiten, …). Für weitere Analysen zu diesem Verfahren sei auf die Vorlesung Effiziente Algorithmen des Hauptstudiums [5] verweisen.

3.3 Suchverfahren

Problemstellung: Gegeben sind (große) Mengen von Datensätzen, die durch einen (eindeutigen) Schlüssel gekennzeichnet sind.



"In real life" ist der Schlüssel hierbei ein Textstring, eine Zahl, etc. In der Vorlesung wollen wir aber ohne Einschänkung davon ausgehen, dass der Schlüssel der gesamte Datensatz ist (sonst müssten wir zu jedem Schlüssel stets noch einen Zeiger auf den Rest des Datensatzes mitangeben).
Gesucht ist nun eine Datenstruktur, die die folgenden Operationen effizient ermöglicht:

- is_member(k): Teste, ob der Datensatz mit Schlüssel k enthalten ist.
- insert(d): Füge den Datensatz d mit Schlüssel $k = k(d)$ ein, falls es noch keinen Datensatz mit Schlüssel k gibt.
- delete(k): Lösche den Datensatz mit Schlüssel k, falls vorhanden.

Bemerkung: So eine Datenstruktur nennt man *Wörterbuch*.
Grundsätzlich gibt es zwei verschiedene Ansätze, um Wörterbücher zu realisieren:

1. Suchbäume
2. Hashing

3.3.1 Suchbäume

Annahme: Die Schlüssel sind vergleichbar, d.h. die Menge aller Schlüssel (das sogenannte *Universum*) ist total geordnet.

3.3.2 Binäre Suchbäume

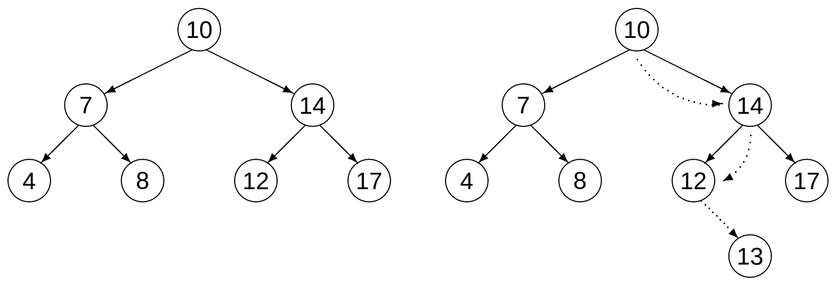
Betrachten wir zunächst *binäre Suchbäume*. In binären Suchbäumen gilt für alle Knoten x:

- $key(x) >$ größter Schlüssel im linken Unterbaum von x
- $key(x) <$ kleinster Schlüssel im rechten Unterbaum von x

Nunmehr stellt sich natürlich die Frage, welchen Aufwand die einzelnen Methoden zur Manipulation der Knoten in einem binären Suchbaum der Höhe h haben.

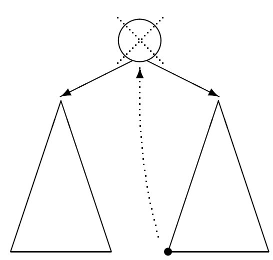
- is-member(k) verfügt über einen zeitlichen Aufwand von $O(h)$. Man muss schlimmstenfalls einmal von der Wurzel des Baumes bis zu einem Blatt laufen (nämlich wenn der gesuchte Schlüssel ein Blatt des Baumes ist oder nicht im Baum enthalten ist.)
- insert(k) verfügt ebenfalls über lineare Laufzeit, also $O(h)$. Um ein Element einzufügen, muss man den Baum nämlich einmal von der Wurzel bis zu dem Knoten durchlaufen, an dem das Blatt eingefügt wird.

Beispiel 3.3
Um in den links dargestellten binären Suchbaum den Schlüssel 13 einzufügen geht man wie folgt vor: Man vergleicht zunächst den Wert des neuen Schlüssels mit dem Wert der Wurzel. In diesem Fall ist der Wert des Schlüssels größer, als der der Wurzel, der Schlüssel muss also im rechten Teilbaum plaziert werden. Der erste Knoten des Teilbaums ist nun größer als der neue Schlüssel, wir wenden uns also nach links. Wir treffen nunmehr auf ein Blatt, und da der Wert dieses Blattes kleiner als der neue Schlüssel ist, wird der Schüssel als rechter Teilbaum dieses Knotens eingefügt.



- delete(k); obwohl hier der Baum unter Umständen reorganisiert werden muss, wird auch für delete(k) nur lineare Zeit verbraucht.

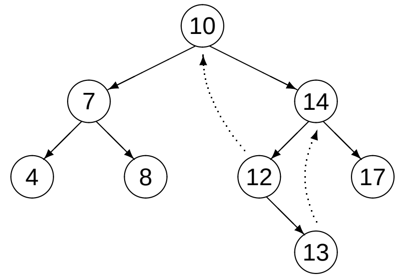
  Löscht man die Wurzel eines Baumes (oder Teilbaumes), so wählt man den linkesten Knoten (*nicht* notwendigerweise ein Blatt) des rechten Teilbaumes der Wurzel als neue Wurzel. Es dürfte klar sein, dass dieses Blatt über den kleinsten Wert aller Blätter und damit auch den kleinsten Wert aller Knoten im rechten Teilbaum verfügt. Da alle Werte im rechten Teilbaum echt größer sind als die im linken Teilbaum, ist dieser Knoten eine gültige Wurzel.



  Da der Aufwand den Knoten zu finden linear ist und die Umordnung konstante Zeit benötigt ergibt sich ein Aufwand von $O(h)$.

  Wenn kein linkestes Blatt existiert muss man den Algorithmus leicht modifizieren, siehe dazu folgendes Beispiel:

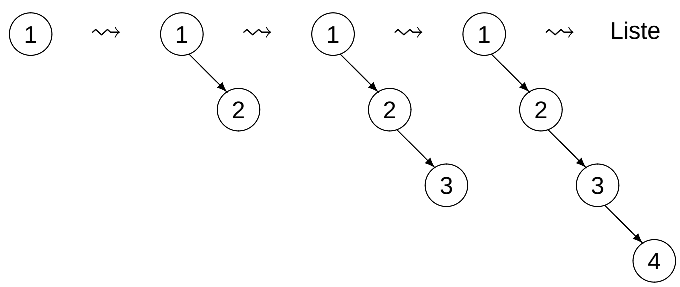
Beispiel 3.4
In unserem nachstehenden Beispielbaum wird die Wurzel (Wert 10) gelöscht. Der Knoten mit dem kleinsten Wert im rechten Teilbaum ist nunmehr 12, er wird zur neuen Wurzel. Der Knoten 13 wird an den ehemaligen Vater von 12 angehängt.



In diesem Fall wird der linkeste Knoten als neue Wurzel verwendet und der rechte Teilbaum dieses Knotens an Stelle des Knotens an dessen Vorgänger angehangen.

Ein Problem der binären Suchbäume ist es aber, dass sie nicht höhenbalanciert sind, so dass sich unter Umständen ein sehr ungünstiges Laufzeitverhalten einstellen kann.

Beispiel 3.5
Wir fügen nacheinander die Schlüssel 1,2,3,4,5, ... in einen zu Beginn leeren Baum ein. Wie in der Skizze zu erkennen erhält man eine Liste, so das die Höhe h des Baumes nicht $O(\log n)$ ist, wie erwünscht, sondern $O(n)$ (wobei n die Zahl der Schlüssel ist)



Wie kann man nun aber sicherstellen, das man einen balancierten Baum erhält? Wir wollen im folgenden auf zwei Möglichkeiten eingehen. Die wirklich effizienten Lösungen werden aber leider erst im Hauptstudium [5] behandelt.

3.3.3 AVL-Bäume

Für alle Knoten x eines *AVL-Baumes* gilt, dass sie Höhe des linken und rechten Unterbaumes von x sich um höchstens 1 unterscheiden.
Wir modifizieren nun die Datenstruktur so, dass an jedem Knoten x die Höhe des Teilbaumes mit Wurzel x gespeichert ist.

Lemma 3.1 *Ein AVL-Baum mit Höhe h enthält mindestens $(\frac{1+\sqrt{5}}{2})^h$ und höchstens $2^{h+1} - 1$ viele Knoten.*

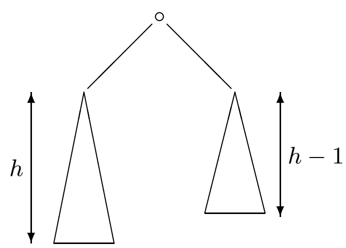
Beweis: Es dürfte klar sein, das jeder binäre Baum der Höhe h höchstens $2^{h+1} - 1$ viele Knoten enthält.

Die andere Aussage des Lemmas beweisen wir durch Induktion über h.

$h = 0$: Hier verfügt der Baum nur über einen Knoten, da $(\frac{1+\sqrt{5}}{2})^0 = 1$ ist die Aussage aus dem obigen Lemma für diesen Punkt erfüllt.

$h = 1$: Ein Baum der Höhe 1 verfügt entweder über 2 oder aber 3 Knoten, für die Zahl der Knoten gilt also ≥ 2. Nun ist aber $(\frac{1+\sqrt{5}}{2})^1 = 1.618 \leq 2$, womit die Aussage auch für diesen Fall gilt.

$h \Rightarrow h + 1$: Damit der gesamte Baum die Höhe $h+1$ hat, muss mindestens einer der beiden Teilbäume die Höhe h haben. Der andere Teilbaum hat dann mindestens die Höhe $h - 1$.



Damit ergibt sich für die Anzahl der Knoten:

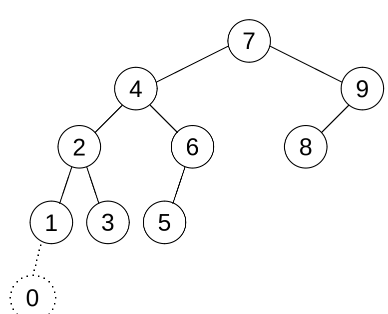
$$\begin{aligned}
\text{\# Knoten} &\geq 1 + \left(\frac{1+\sqrt{5}}{2}\right)^h + \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} \\
&\geq \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} \cdot \left(\frac{1+\sqrt{5}}{2} + 1\right) \\
&= \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} \cdot \left(\frac{1+2\sqrt{5}+5}{4}\right) \\
&= \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^2 \\
&= \left(\frac{1+\sqrt{5}}{2}\right)^{h+1}
\end{aligned}$$

□

Korollar 3.1 *Ein AVL-Baum mit n Knoten hat die Höhe $O(\log n)$.*

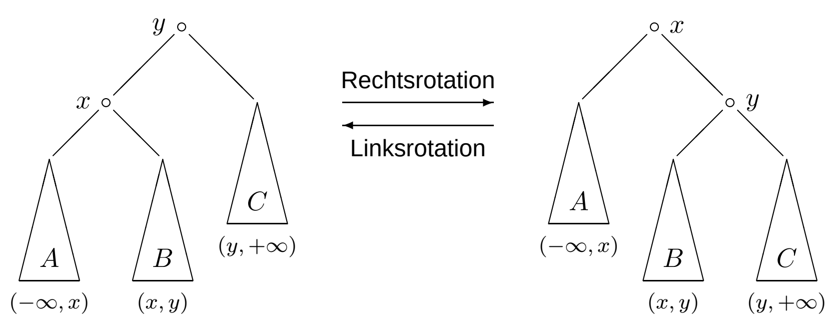
Korollar 3.2 *In einem AVL-Baum mit n Knoten hat* is_member(k) *die Laufzeit $O(\log n)$.*

Wie verhalten sich aber insert und delete? Nehmen wir als Beispiel folgenden Baum und nehmen wir an, wir wollen den Schlüssel 0 einfügen. Fügen wir den Schlüssel korrekt als Blatt des Knoten 1 ein, so ist die Höhenbedingung verletzt. Wir müssen den Baum umsortieren.

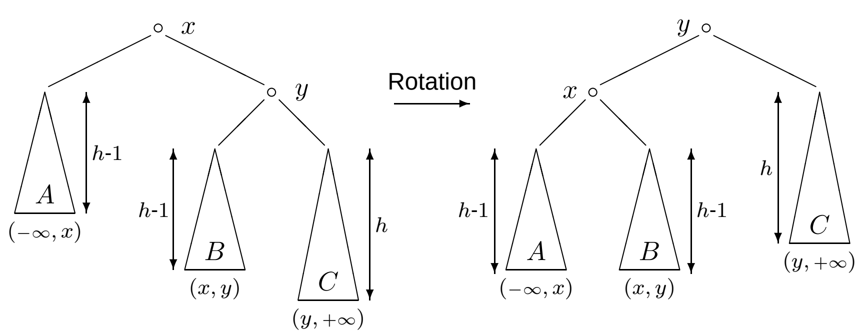


Beschäftigen wir uns also zunächst mit `insert`:

1. Dieser Schritt erfolgt wie bisher.
2. In diesem Schritt wird die Höhenbedingung wieder hergestellt. Es dürfte klar sein, dass die Höhenbedingung nur auf dem Pfad von der Wurzel zu dem eingefügten Knoten verletzt sein kann. Wir verfolgen daher den Pfad von unten nach oben und stellen die Höhenbedingung wieder her. Dazu nutzen wir sogenannte Rotationen. Die folgende Skizze zeigt eine Rotation um x-y:

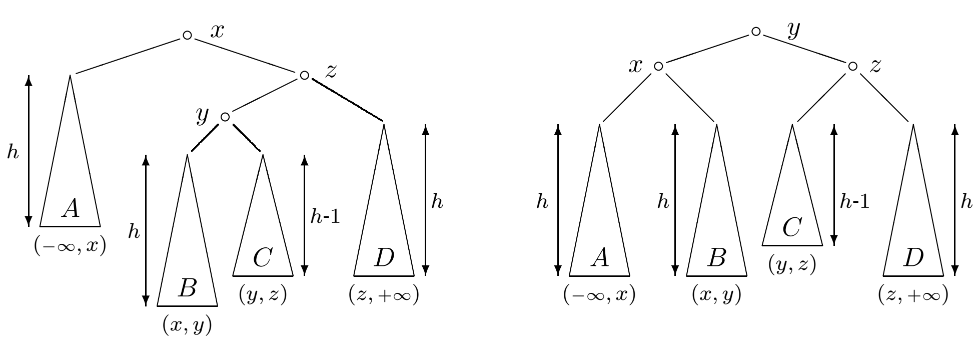


Wie man unschwer erkennen kann wird durch diese Operation die Höhe der Teilbäume verändert. Sehen wir uns daher ein Beispiel an, wie man die Rotation zum Wiederherstellen der Höhenbedingung verwenden kann:



Während im Knoten y die Höhenbedingung nach dem Einfügen noch gilt, ist sie in x verletzt. Nach der Rotation gilt die Höhenbedingung jedoch wieder. Die Höhe der Teilbäume von y ist gleich der Höhe der Teilbäume von x vor dem Einfügen. Damit ist die Höhenbedingung jetzt überall im Baum erfüllt.

Mitunter muss man eine Doppelrotation vornehmen, um den Baum zu rebalancieren.

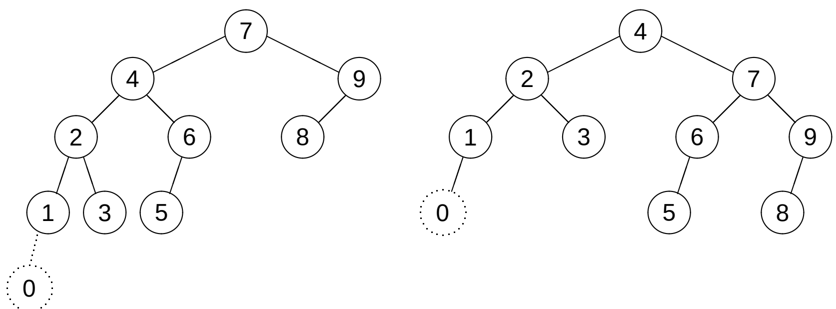


> **Anmerkung:** In obigem Beispiel kann natürlich auch der Teilbaum B die Höhe $h-1$ und der Teilbaum C die Höhe h haben. Mindestens einer der Teilbäume B oder C muss allerdings Höhe h aufweisen, sonst wäre ja die Höhenbedingung nicht verletzt, und eine Rotation zur Rebalancierung wäre dann nicht notwendig.

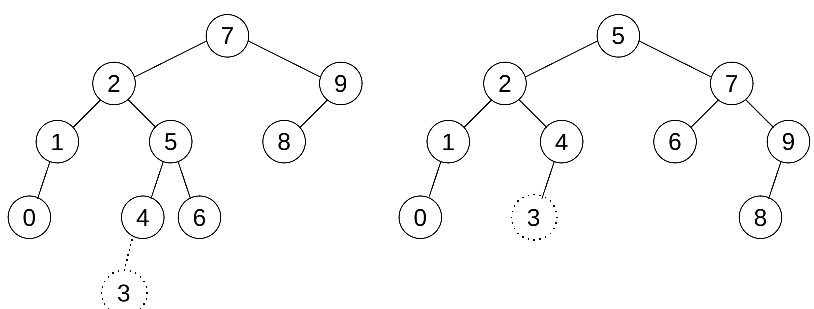
Damit dürfte einsichtig seien, dass sich insert in einem AVL-Baum mit n Knoten in $O(\log n)$ durchführen lässt.

Gleiches gilt im übrigen auch für delete. Auch hier kann es natürlich sein, dass man nach dem Löschen eines Knotens den Baum rebalancieren muss. Die Vorgehensweise ist dann die gleiche wie bei insert.

Beispiel 3.6
Zum Abschluss noch ein Beispiel für das Rebalancieren von AVL-Bäumen. In dem folgenden Baum wird der Schlüssel 0 hinzugefügt und der Baum anschließend durch eine Einfachrotation rebalanciert.



Bei unserem zweiten Beispiel ist nach dem Einfügen des Schlüssels 3 eine Doppelrotation notwendig um den Baum zu rebalancieren.



3.3.4 (a, b)-Bäume

Definition 3.2 *Ein (a, b)-Baum ist ein Suchbaum, so dass gilt:*

- *alle Blätter haben gleiche Tiefe*
- *Schlüssel sind nur in den Blättern gespeichert (externer Suchbaum)*
- *für alle Knoten v außer Wurzel und Blättern: $a \leq$ # Kinder$(v) \leq b$*
- *für Wurzel $2 \leq$ # Kinder $\leq b$*
- $b \geq 2a - 1$

- *für alle inneren Knoten v gilt: hat v l Kinder, so sind in v $l-1$ Werte $k_1, \dots k_{l-1}$ gespeichert und es gilt:*

$$k_{i-1} < key(w) \leq k_i \text{ für alle Knoten } w \text{ im } i\text{-ten Unterraum von } v$$

(wobei $k_0 = -\infty$, $k_l = +\infty$)

Bemerkung: (a, b)-Bäume mit $b = 2a - 1$ nennt man auch *B-Bäume*. Diese B-Bäume wurden erstmals in einer Arbeit von Prof. R. Bayer und W. McCreight im Jahr 1972 beschrieben.

Lemma 3.2 *Sei T ein (a, b)-Baum mit n Blättern. Dann gilt:*

$$\log_b(n) \leq \textit{Höhe(T)} \leq 1 + \log_a\left(\frac{n}{2}\right)$$

Beweis: Dieser Beweis sei dem Leser als Übungsaufgabe überlassen. Alternativ sei auf die Arbeit von Bayer / McCreight verwiesen. □

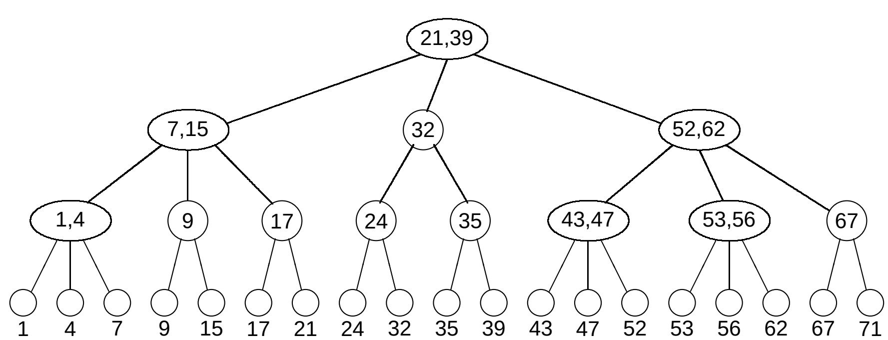
- **is_member(k):**
  - v := Wurzel von T
  - **while** v nicht Blatt **do**
    - i := min$\{1 \leq j \leq \#\text{ Kinder}(v) \mid k \leq k_j\}$
    - v := i-tes Kind von v
  - **if** $k = key(v)$ **then return** ("true")
    - **else return** ("false")

  Die Laufzeit ist O(Höhe(T)).

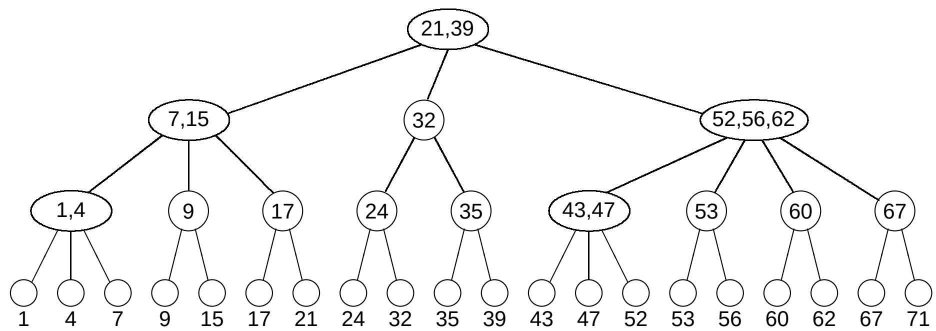
- **insert(v):**
  - Bestimme Blatt v wie in is_member(k)
  - w := parent(v)
  - Füge k als zusätzliches Blatt von w ein.
  - **while** # Kinder(w) $> b$ **do**
    - **if** $w \neq$ Wurzel **then** y := parent(w)
      - **else** y := neue Wurzel mit w als einzigem Kind
    - Zerteile w in zwei Knoten w_1 und w_2 mit den $\lfloor \frac{b+1}{2} \rfloor$ kleinsten bzw. $\lceil \frac{b+1}{2} \rceil$ größten Kindern von w
  - $w := y$

  Die Laufzeit ist O(Höhe(T)).

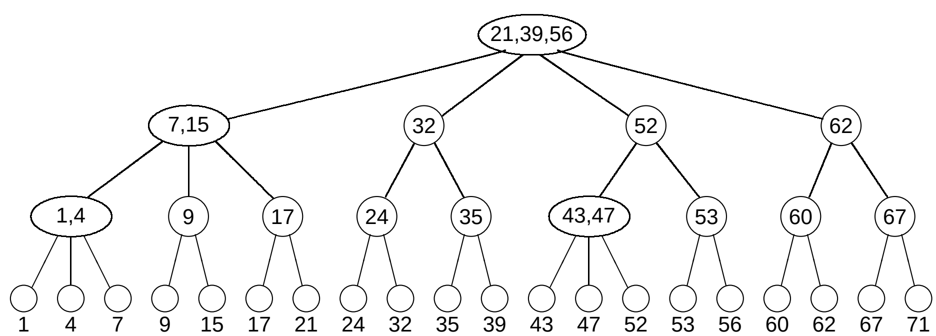
Beispiel 3.7
Beschäftigen wir uns nun kurz mit einem Beispiel für die Operation **insert(v)**. In den folgenden $(2, 3)$-Baum soll ein Schlüssel mit dem Wert 60 eingefügt werden.



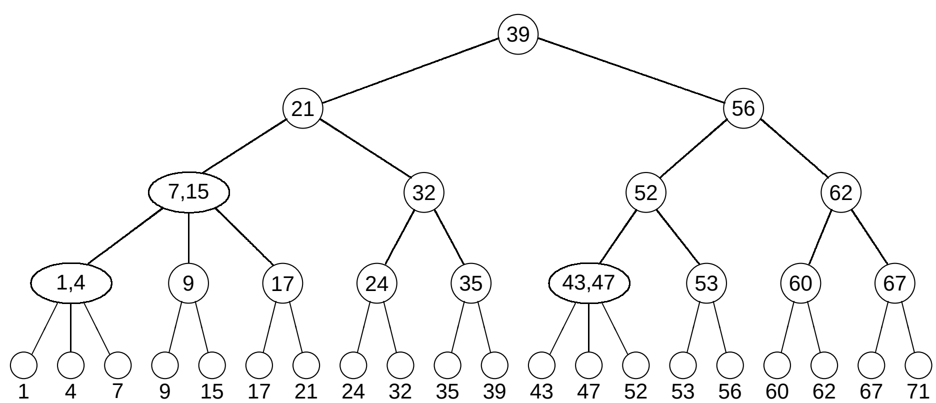
Nach dem Ausführen der Einfügeoperation ergibt sich zunächst der unten stehende Baum. Eigentlich hätte man den Schlüssel 60 an den Knoten 53,56 anfügen müssen, jedoch hätte dieser Knoten dann über mehr als drei Kinder verfügt. Daher wurde der Knoten geteilt.



Durch die Aufteilungsoperation verfügt nunmehr der Knoten 52,56,62 über vier Kinder, so dass auch dieser Knoten geteilt werden muss.



Da nunmehr die Wurzel über mehr als drei Kinder verfügt, wird sie geteilt und eine neue Wurzel wird eingefügt.



- **delete(k):**
  Bestimme Blatt v wie in is_member(k)
  if $key(v) \neq k$ **then stop**
  w := parent(v); Entferne v aus w.
  while (# Kinder(w) $< a$) $\wedge$ ($w \neq$ Wurzel) **do**
    Sei y ein linker oder rechter Nachbar von w.

if # Kinder$(y) = a$ **then** Verschmelze w und y.
else Adoptiere das rechteste bzw. linkeste Kind von y.
$w :=$ parent(w)
if $(w =$ Wurzel$)$ und (Wurzel hat nur ein Kind) **then** Lösche Wurzel.

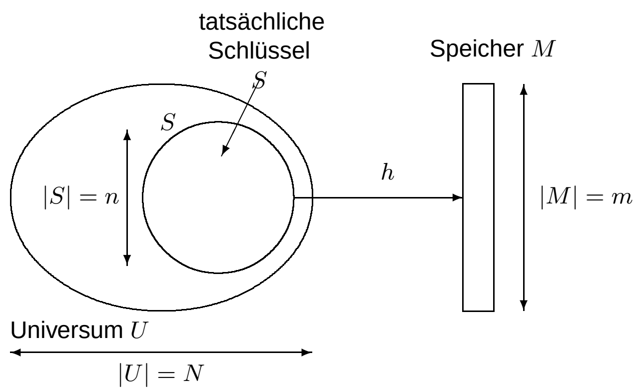
Die Laufzeit ist wiederum $O(\text{Höhe}(T))$.

Korollar 3.3 *In einem (a, b)-Baum mit n Blättern (bzw. gespeicherten Schlüsseln) kann man* is_member, insert, delete *in $O(\log n)$ durchführen.*

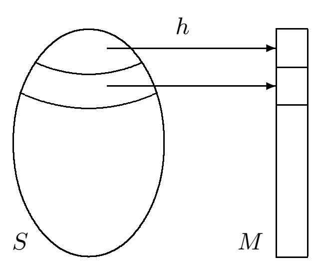
Bemerkung: Die Wahl von a und b hängt wesentlich von der Größe des (a, b)-Baums ab. Allgemein gilt:

- Liegt der (a, b)-Baum im RAM, wählt man b klein. Dadurch hat man geringe Kosten für das Finden des "richtigen" Teilbaums.
- Liegt der (a, b)-Baum auf Sekundärspeichern, wählt man a groß. Der Baum hat dann geringe Höhe, dadurch ist die Anzahl der Zugriffe auf den Sekundärspeicher gering.

3.3.5 Hash-Verfahren



Hierbei soll die Hashfunktion h die Eigenschaft haben, dass man **is_member** effizient ausführen kann – und zwar (möglichst) für *alle* Mengen S.
Ansatz: h ordnet *jedem* Element des Universums einen festen Platz im Speicher zu.



Da $|U| \gg m$ gibt es natürlich Mengen S, so dass deren Elemente auf den gleichen Speicherplatz abgebildet werden, man spricht von *Kollisionen*. Die "Kunst" des Hashens ist dabei: Vermeide solche Kollisionen!
Im Folgenden ist also näher auf diese beiden Punkte einzugehen:

1. Wahl der Hashfunktion h

2. Auflösen von Kollisionen

Zu 1.) Wahl der Hashfunktion
Idee: Betrachte beliebige Funktion h mit $\forall i \; |h^{-1}(i)| = \frac{N}{m}$, wobei $m = |M|$ und $N = |U|$.

$$\Pr_{x,y}\left[h(x) = h(y)\right] = \frac{1}{m}$$

Dies heißt: wählt man den Datensatz zufällig, ist der Erwartungswert für die Anzahl Kollisionen klein.
Aber: In der Praxis ist der Datensatz meist leider *nicht* zufällig. Trotzdem würden wir gerne eine möglichst ideale Hashfunktion wählen. Leider ist uns die Verteilung der Elemente des Universums im Allgemeinen unbekannt, so daß wird die Idealität einer Hashfunktion nicht nachweisen können. Das zufällige Auswählen einer Hashfunktion *zur Laufzeit* aus einem Pool von möglichen Hashfunktionen bietet uns hierfür einen Ausweg. Wir sind damit zwar nicht vor einer schlechten Wahl der Hashfunktion gefeit, allerdings können wir im Folgenden zeigen, daß wir im Erwartungswert eine gut streuende Hashfunktion erhalten, sofern der Pool von möglichen Hashfunktionen geeignet gewählt ist.
Universelles Hashing
Eine Menge $\mathcal{H}$ von Hashfunktionen heißt *universell*, falls gilt:

$$\underbrace{\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|}}_{\Pr_h[h(x)=h(y)]} \leq \frac{1}{m} \quad \forall x, y \in U$$

Beispiel 3.8
Im folgenden sei ein Beispiel für eine universelle Hashfamilie gegeben. O.E. sei m eine Primzahl. Dann ist $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ ist ein Körper (s. hierzu auch Diskrete Strukturen, Band 1, [7])

$$\begin{array}{|c|c|c|c|c|}\hline x_0 & \underbrace{x_1}_{\lfloor \log m \rfloor \text{ Bits}} & x_2 & \quad\cdots\quad & x_r \\ \hline\end{array}$$

Schlüssel $x \in U$: $x = x_0, x_1, x_2, \ldots, x_r$.
Wir können also x_i interpretieren als Zahl $\in \mathbb{Z}_m$. Für $a = (a_0, \ldots, a_r)$ mit $a_i \in \mathbb{Z}_m$ setzen wir:

$$\begin{array}{rcl} h_a : \; U & \rightarrow & \mathbb{Z}_m \\ x & \mapsto & \sum_{i=0}^{r} a_i x_i \bmod m \end{array}$$

Satz 3.9 $\mathcal{H} = \{h_a \mid a = (a_0, \ldots a_r), a_i \in \mathbb{Z}_m\}$ *ist universell.*

Beweis: Sei $x \neq y$, und damit o.E. $x_0 \neq y_0$, aber $h_a(x) = h_a(y)$. Für alle $a_1, \ldots, a_r \in \mathbb{Z}_m$ gilt:
Es gibt genau ein $a_0 \in \mathbb{Z}_m$ mit $h_a(x) = h_a(y)$, denn (mit $\mathbb{Z}_m$ ist Körper):

$$\begin{array}{rrcl} & h_a(x) & = & h_a(y) \\ \iff & \sum_{i=0}^{r} a_i x_i & = & \sum_{i=0}^{r} a_i y_i \mod m \\ \iff & a_0 \underbrace{(x_0 - y_0)}_{\in \mathbb{Z}_m} & = & \sum_{i=1}^{r} a_i (y_i - x_i) \mod m \end{array}$$

Also:

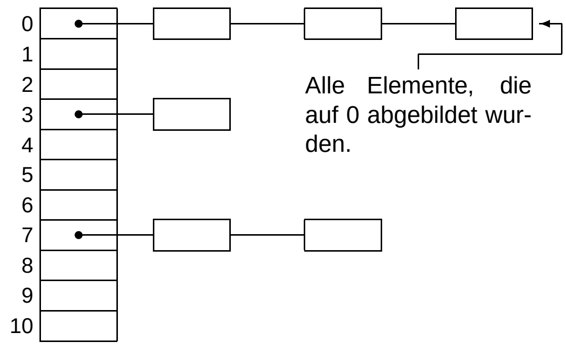
$$\Pr_a\left[h_a(x) = h_a(y)\right] = \frac{\text{\# a's mit } h_a(x) = h_a(y)}{\text{\# aller a's}} = \frac{m^r \cdot 1}{m^{r+1}} = \frac{1}{m}$$

Die $a_1, \ldots, a_r$ können frei gewählt werden (dafür gibt es jeweils m Möglichkeiten), a_0 ist für $h_a(x) = h_a(x)$ fest, andernfalls gibt es auch hier m Möglichkeiten, a_0 zu wählen. $\square$

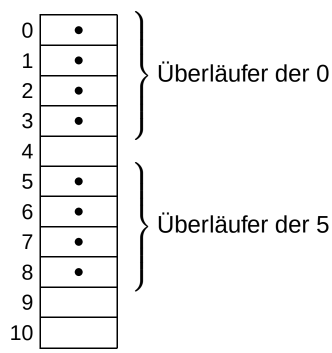
Zu 2.) Auflösen von Kollisionen

Mögliche Strategien für die Auflösung von Kollisionen sind:

Verketten:



Lineares Sortieren: Idee: Wird ein Element x auf den i-ten Speicherplatz in M abgebildet, so wird es dort gespeichert, falls der Platz noch nicht belegt ist. Sonst wird es im $i+1$-ten Speicherplatz gespeichert, falls dieser noch frei ist, andernfalls ...



Faustregel: Diese Möglichkeit ist nur brauchbar, falls $|M| \gg |S|$.

Doppeltes Hashen: $h(k,i) = h_1(k) + i \cdot h_2(k) \bmod m$, wobei h_1 und h_2 *zwei* Hashfunktionen sind.

insert(k):

    $i := 0$
    do forever
        if $M[h(k,i)] =$ nil **then** $M[h(k,i)] := k$; stop;
            else $i = i + 1$

Faustregel: brauchbar – aber nur wenn keine delete's vorkommen.

3.3.6 Vorrangwarteschlangen

Definition 3.3 *Eine Vorrangwarteschlange (engl. Priority Queue) ist eine Datenstruktur, die die folgenden Operationen unterstützt:*

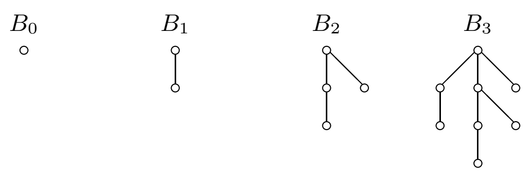
- Insert
- DeleteMin $\rightsquigarrow$ *Finden und Löschen des Elements mit dem kleinsten Schlüssel*
- DecreaseKey $\rightsquigarrow$ Verkleinern *eines Schlüssels*

- Union $\rightsquigarrow$ *Vereinigung zweier Datenstrukturen*

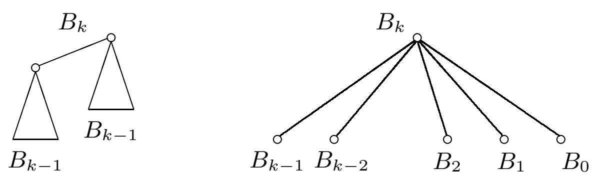
In dieser Vorlesung soll eine Realisierung einer Vorrangwarteschlange als *Binomial Heap* besprochen werden. Dieser wird heute allerdings nicht mehr (häufig) verwendet, da es eine bessere Datenstruktur gibt, die sogenannten *Fibonacci Heaps*. Für weitere Informationen hierzu sei auf die Vorlesung Effiziente Algorithmen und Datenstrukturen des Hauptstudiums [5] verwiesen.

	BinHeap	FibHeap
Insert	$O(\log n)$	$O(1)$
DeleteMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
	worst case	amortisiert

Definition 3.4 *Ein Baum mit einem Knoten ist der Binomialbaum B_0. Entsprechend ist ein Baum mit zwei Knoten der Binomialbaum B_1. Man erhält den Binomialbaum B_k, indem man die Wurzeln der zwei Binomialbäume B_{k-1} mit einer Kante verbindet.*



Es gilt ebenso, dass man B_k erhält, wenn man die Wurzeln der Binomialbäume B_{k-1} $\ldots B_0$ mit einem neuen Wurzelknoten verbindet.



Lemma 3.3 *Für einen Binomialbaum B_k gilt:*

- *er enthält 2^k Knoten, Höhe k*
- *es gibt genau $\binom{k}{i}$ Knoten mit Tiefe i*
- *die Wurzel hat Grad k, alle anderen Knoten haben Grad $\leq k-1$.*

Beweis: Der Beweis sei dem Leser als Übungsaufgabe überlassen. $\square$

Definition 3.5 *Ein Binomial Heap ist eine* Menge $\mathcal{H}$ *von Binomialbäumen, so dass*

1. *jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung:*

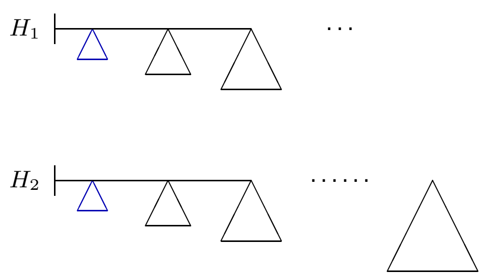
$$key(v) \leq key(w)\ \forall v, w \text{ mit } v \text{ ist Vater von } w$$

2. $\forall k \in \mathbb{N}$*: $\mathcal{H}$ enthält höchstens einen B_k*

Interpretation:
Aus der Eigenschaft (2) für Binomial Heaps folgt: Sind in $\mathcal{H}$ n Schlüssel gespeichert, so besteht $\mathcal{H}$ aus höchstens $\log_2 n$ vielen Bäumen.
Aus der Eigenschaft (1) folgt: In jedem Baum ist der kleinste Schlüssel in der Wurzel gespeichert.

Korollar 3.4 *FindMin benötigt $O(\log n)$ Zeit.*

Beschäftigen wir uns nun eingehender mit den Algorithmen für Union, Insert, Decease Key und Delete Min. Wie wir sehen werden lassen sich die letzten drei Operationen im wesentlichen auf Union zurückführen.

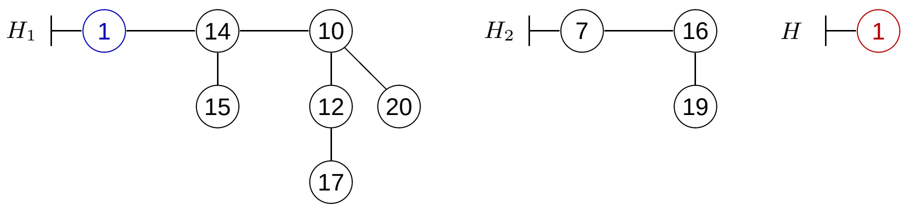


Union: Unser Ziel ist die Vereinigung zweier Binomial Heaps H_1, H_2 zu H. Sei mintree(X) eine Funktion, die den kleinsten Baum (d.h. den Baum mit den wenigsten Knoten) in X zurückliefert, ebenso liefere maxtree(X) den größten Baum (d.h. den Baum mit den meisten Knoten) in X. Zu Beginn des Algorithmus ist H leer, er wird dann schrittweise so vergrößert, dass zu jedem Zeitpunkt gilt, dass maxtree(H) kleiner oder gleich mintree(H_1) und mintree(H_2) ist.

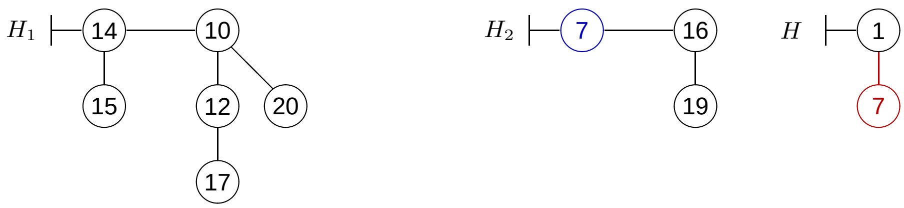
H := leer;
while $H_1 \neq$ leer $\vee$ $H_2 \neq$ leer **do**
  if maxtree(H) $<$ mintree(H_1) $\leq$ mintree(H_2) **then**
    α) Verschiebe kleinsten Baum in H_1 nach H.
  else if maxtree(H) $<$ mintree(H_2) $\leq$ mintree(H_1) **then**
    β) Verschiebe den kleinsten Baum in H_2 nach H.
  else if maxtree(H) $=$ mintree(H_1) $=$ mintree(H_2) **then**
    γ) Verschmelze kleinsten Baum aus H_1 und H_2, hänge neuen Baum in H ein.
  else if maxtree(H) $=$ mintree(H_1) $<$ mintree(H_2) **then**
    δ) Verschmelze kleinsten Baum in H_1 und größten Baum in H. Es entsteht ein neuer Baum in H.
  else if maxtree(H) $=$ mintree(H_2) $<$ mintree(H_1) **then**
    ϵ) Verschmelze kleinsten Baum in H_2 und größten Baum in H. Es entsteht ein neuer Baum in H.

Man definiert hierbei mintree(leerer Heap) $= \infty$.

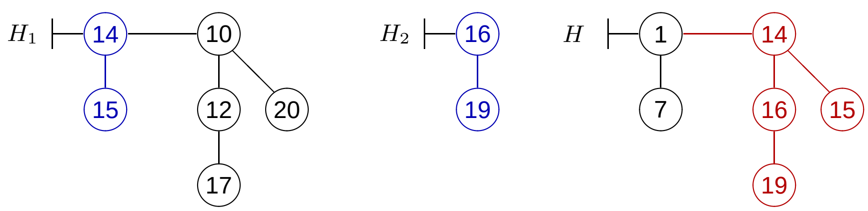
Beispiel 3.9



Im 1. Schritt (d.h. im 1. Durchgang der while-Schleife) wird der erste Knoten von H_1 nach H verschoben. Dies entspricht dem Fall α aus unserem Algorithmus.

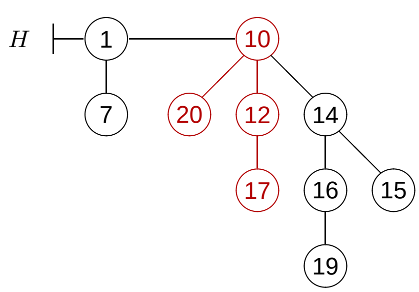


Im 2. Schritt ist es nicht mehr möglich, einfach den kleinsten Teilbaum nach H zu verschieben, da H dann zwei gleich große Teilbäume enthalten würde. Vielmehr verschmelzen wir den kleinsten Teilbaum aus H_2 mit dem größten Teilbaum aus H. Damit tritt der Fall ϵ ein.



Nun im 3. Schritt ist die Situation ähnlich. Wir können keinen der beiden kleinsten Teilbäume aus H_1 und H_2 in H einfügen, da H dann ebenfalls zwei gleich große Teilbäume enthalten würde. Wir verschmelzen also die beiden Teilbäume aus H_1 und H_2 miteinander, indem wir den Teilbaum mit der größeren Wurzel an den mit der kleineren Wurzel anfügen. Der neu entstandene Baum wird nun in H eingefügt. Dies entspricht dem Fall γ ein.

Da wir im 4. Schritt sowohl in H_1, als kleinsten, als auch in H, als größten, Teilbäume der Größe 4 haben, müssen wir die Teilbäume verschmelzen. Dies ist der Fall δ unseres Algorithmus.



Wenden wir uns nun der *Invariante* des Algorithmus Union zu. Vor und nach jedem Durchlauf der while-Schleife gilt:

$$\text{maxtree}(H) \leq min\{\text{mintree}(H_1), \text{mintree}(H_2)\}$$

Beweis: Zu Beginn trifft die Bedingung zu. Ebenso weißt man leicht nach, dass die Bedingung auch für die Fälle α, β, γ, δ und ε erhalten bleibt. Wichtig: Wir verwenden die Tatsache, dass H_1, H_2 jeden Baum B_k höchstens einmal enthält. □

Wir haben also gesehen, dass der angegebene Algorithmus für Union funktioniert. Die Laufzeit des Algorithmus ist:

$$O(\text{\# Bäume in } H_1 + \text{\# Bäume in } H_2) =$$
$$O\left(\log(\text{\# Schlüssel in } H_1) + \log(\text{\# Schlüssel in } H_2)\right)$$

Insert(H, k): Wir gehen folgendermaßen vor:

1. Erzeuge einen neuen Heap H' mit Schlüssel k als einzigem Element.

2. $H :=$ Union(H, H')

Klar ist, dass sich die Laufzeit zu $O\left(\log(\text{\# Elemente in } H)\right)$ ergibt.

DeleteMin(H): Wir wissen, dass das Minimum stets die Wurzel eines Baumes ist.

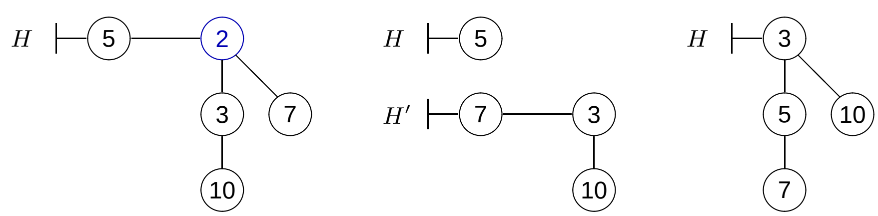
1. Bestimme den Baum in H mit der kleinsten Wurzel.

2. Gebe Wurzel des Baumes aus und erzeuge einen neuen Heap H' mit allen Unterbäumen der Wurzel.

3. $H :=$ Union(H, H')

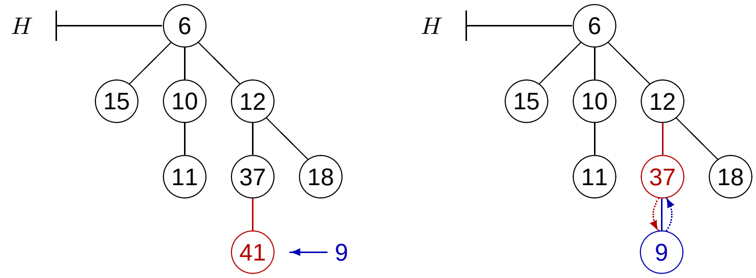
Klar ist wiederum: Laufzeit $O\left(\log(\text{\# Elemente in } H)\right)$.

Beispiel 3.10
Im folgenden Beispiel entfernen wir den Knoten 2 aus dem Binomial Heap (links). Dadurch zerfällt der Heap in zwei Teile (mitte), die dann mittels Union wieder vereinigt werden (rechts).

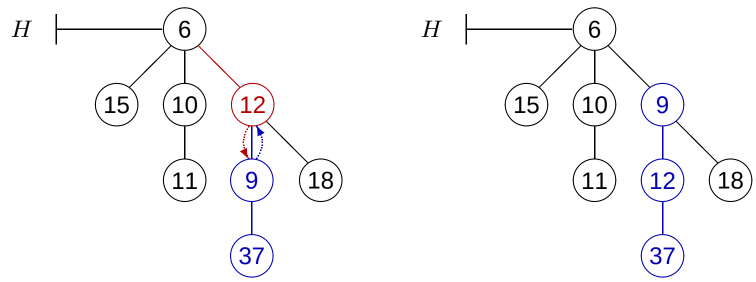


DecreaseKey(H, x, k): Zur Erinnerung sei noch einmal erwähnt, dass diese Operation den Schlüssel von x auf k setzen sollte, falls $k < key[x]$. Doch zunächst folgendes Beispiel:

Beispiel 3.11
In dem folgenden Heap soll der Schlüssel 41 durch 9 ersetzt werden (links). Tut man dies, so ist die Heapbedingung nicht mehr erfüllt (rechts), da der Vater von 9, der Knoten 37, dann größer als 9 ist. Um die Heapbedingung wieder herzustellen, vertauscht man beide Knoten miteinander.



Nach dem ersten Vertauschen ist die Heapbedingung jedoch immer noch verletzt (links), so dass man noch einmal die Knoten vertauschen muss. Auf diese Weise wandert der Knoten solange im Baum nach oben bis die Heapbedingung erfüllt ist (rechts).



Für DecreaseKey ergibt sich somit der folgende Algorithmus:

if $key[x] < k$ **then** error;
  else $key[x] = k$; $y := x$; $z :=$ Vater(x);
    while $z \neq$ nil $\wedge\; key[y] < key[z]$ **do**
      Vertausche Inhalt von y und z;
      $y := z$; $z :=$ Vater(x);

Für Laufzeit seine Laufzeit gilt schließlich:

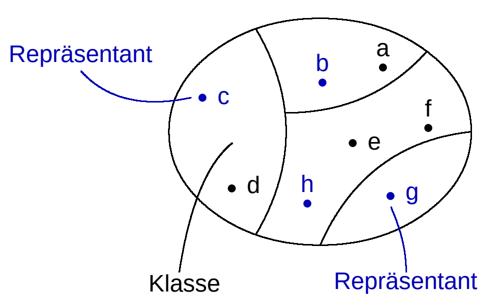
$$O(\text{max. Höhe eines Baumes in } H) = O\left(\log(\text{\# Schlüssel in } H)\right)$$

3.4 Mengendarstellungen – Union-Find Strukturen

Problemstellung: Gegeben sei eine (endliche) Menge S, die in Klassen X_i partitioniert ist:

$$S = X_1 \,\dot\cup X_2 \,\dot\cup\, \ldots \,\dot\cup\, X_l$$

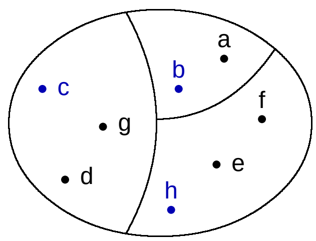
Für jede Klasse X_i gibt es hierbei einen Repräsentanten $r_i \in X_i$.



Gesucht wird eine Datenstruktur, welche die folgenden Operationen unterstützt:

- Init(S) $\rightsquigarrow$ Jedes Element bildet eine eigene Klasse mit sich selbst als Repräsentanten.
- Union(r, s) $\rightsquigarrow$ Vereinige die beiden Klassen mit den Repräsentanten r und s, wähle r als neuen Repräsentanten.
- Find(x) $\rightsquigarrow$ Bestimme zu $x \in S$ den Repräsentanten der Klasse, die x enthält.

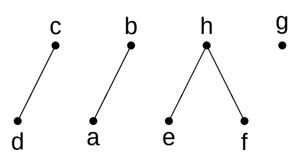
In unserem Beispiel würde Find(d) den Wert c liefern. Führt man die Operation Union(c,g) auf unserem Beispiel auf so erhält man folgendes Resultat:



Eine solche Datenstruktur nennt man auch *Union-Find-Struktur*.

Eine triviale Implementierung benötigt $O(|S|)$ für die Operation Union. Ziel ist eine Laufzeit für Union von $O(\log(|S|)$. Wir werden sehen, dass es sogar noch besser geht.

Um logarithmische Laufzeit zu erreichen, realisieren wir die Datenstruktur als Vereinigung von zur Wurzel hin gerichteten Bäumen. Dabei bilden die Repräsentanten die Wurzeln und die restlichen Elemente der Klasse die restlichen Knoten.



Die Operationen Find und Union sind dann folgendermaßen zu implementieren:

Find(x): Laufe von x zur Wurzel und gebe diese aus. Da wir davon ausgehen, dass x ein Zeiger auf den Knoten bzw. der Knoten selbst ist, beträgt die Laufzeit O(max. Höhe eines Baumes).

Union: Unser Ziel ist, Union so zu implementieren, dass die maximale Höhe eine Baumes $\leq log_2|S|$ ist. Die Operation Union hat dabei Laufzeit $O(1)$.

Bevor wir die Betrachtung von Union-Find Strukturen fortsetzen, betrachten wir folgendes Anwendungsbeispiel:

Beispiel 3.12
Kruskals Algorithmus für Minimum Spanning Tree (MST)
In der Vorlesung Diskrete Strukturen I (s. auch [7]) haben wir uns ausführlich mit Kruskals Algorithmus für Minimale Spannbäume auseinander gesetzt. Zur Erinnerung sei er hier noch einmal wiedergegeben:

`Kruskals Algorithmus  Version 2`
Eingabe: zshgd. Graph $G = (V, E)$, wobei $E = \{e_1, \ldots, e_m\}$, Gewichte $w[e]$;

Ausgabe: Kantenmenge T eines min. spann. Baums in G.
Sortiere die Kantenmenge so, dass gilt:
$w[e_1] \leq w[e_2] \leq \ldots \leq w[e_m]$;
T := $\emptyset$;
for i := 1 **to** m **do**
if $T \cup \{e_i\}$ kreisfrei **then** T := $T \cup \{e_i\}$;
od

Allerdings haben wir damals die Frage ausgeklammert, wie man effizient bestimmt ob das Einfügen einer Kante einen Kreis erzeugt. Dies wollen wir nun nachholen, da damit die Effizienz des Algorithmus steht und fällt.

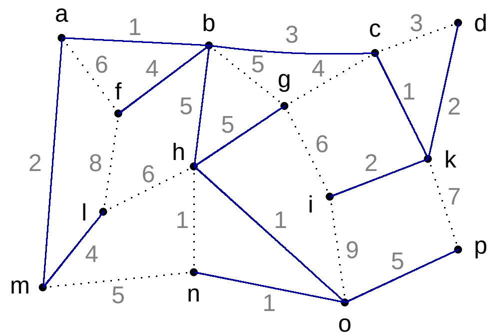
Geht man nach einem primitiven Ansatz vor, so erreicht man quadratische Laufzeit. Man fügt die aktuell betrachtete Kante temporär in den bestehenden kreisfreien Graphen ein und testet ob der Graph dann immer noch kreisfrei ist.

Union-Find-Strukturen ermöglichen es nun, eine Laufzeit von $O(|E| \log |V|)$ zu erreichen. Beim Aufbauen des Spannbaums nutzen wir diese Datenstruktur derart, dass alle Knoten die in einer Zusammenhangskomponente liegen jeweils einer Klasse angehören. Zu Beginn liegt jeder Knoten in einer eigenen Klasse. Wollen wir testen, ob wir eine Kante in den Spannbaum einfügen können, so testen wir einfach, ob die sie begrenzenden Knoten in verschiedenen Klassen und damit in verschiedenen Zusammenhangskomponenten liegen. Wenn ja, erzeugt das Einfügen der Kante keinen Kreis, und wir führen ein Union auf den beiden Klassen der Endknoten der Kante aus. Liegen beide Knoten hingegen in der selben Klasse, so wird die Kante nicht in den aufzubauenden Spannbaum aufgenommen.

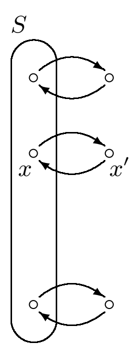
Damit ergibt sich der folgende Algorithmus:

<u>`Kruskals Algorithmus - Version 3`</u>
Eingabe: zusammenhängender Graph $G = (V, E)$; Gewichte $w[e]$;
Ausgabe: Kantenmenge T eines min. spann. Baums in G.
Sortiere die Kantenmenge so, dass gilt:
$w[e_1] \leq w[e_2] \leq \ldots \leq w[e_m]$;
$T := \emptyset$;
Init(V);
for i := 1 **to** m **do**
Bestimme $u, v \in V$ mit $e_i = \{u, v\}$;
if Find(u) $\neq$ Find(v) **then**
$T := T \cup \{e_i\}$;
Union(Find(u), Find(v));
fi
od

Im Folgenden ist ein Beispielgraph mit gewichteten Kanten abgebildet, in dem der minimale Spannbaum nach dem Algorithmus von Kruskal bestimmt wurde.

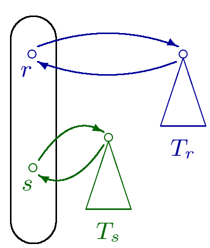


Bevor wir uns eingehender mit der Implementierung der Operationen Init(s), Find(x) und Union(r, s) beschäftigen, wollen wir uns noch einmal über ihre grundlegende Arbeitsweise klar werden:



- Init(s) legt von jedem Element $x \in S$ eine *Kopie* an.
- Union(r, s) hängt die *Kopien* geeignet zusammen.
- Find(x) verfolgt zunächst den Link zur Kopie von x und von dort aus weiter zur Wurzel.

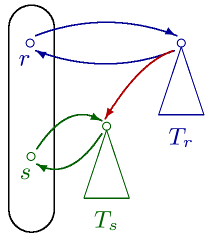
Wie kann nun ein Union(r, s) implementiert werden? Vor der Union-Operation stellt sich folgende Situation dar:



Annahme: Jede Wurzel kennt die Höhe ihres Baumes. Zu Beginn gilt: $\forall x \in S$: Höhe$(x) = 0$. Wir unterscheiden nun 2 Fälle:

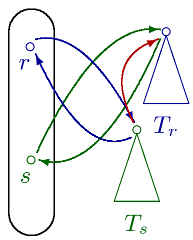
1. Fall: Höhe$[r] \geq$ Höhe$[s]$
Füge Baum mit Wurzel s als neuen Unterbaum von r ein.

$$\text{Höhe}[r] := max\{\text{Höhe}[r], \text{Höhe}[s] + 1\}$$



2. Fall: Höhe$[r] <$ Höhe$[s]$

Hierbei muss darauf geachtet werden, dass nach Definition von Union gesteuert werden soll, welche der beiden Wurzeln die neue Wurzel sein soll.



1. Schritt: Vertausche Wurzeln von T_r und T_s.
2. Schritt: Weiter wie im 1. Fall.

Klar ist nun: Union benötigt $O(1)$ Zeit.

Lemma 3.4 *Für jeden Baum T_r gilt:*

$$\textit{Höhe}(T_r) \leq \log_2 (\textit{\# Knoten in } T_r)$$

Beweis: Zu Beginn stimmt die Behauptung des Lemmas. Für Union zweier Bäume T_r und T_s gilt: Sei h_r die Höhe von T_r und n_r die Anzahl Knoten in T_r. Analoges gelte für h_s und n_s.

1. Fall: $h_r = h_s$

- Höhe h des neuen Baumes ist $h = h_r + 1$.
- Anzahl Knoten des neuen Baumes ist $n = n_r + n_s$.

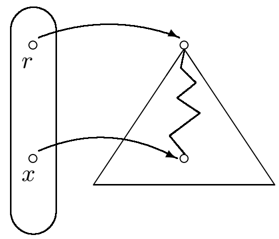
Zu zeigen ist $h \leq \log_2 n$:

$$\begin{array}{rcll} \log_2 n & = & \log_2(n_r + n_s) & \\ & \geq & \log_2(2n_r) & \text{o.E. ist } n_r < n_s \\ & = & 1 + \log_2(n_r) & \\ & \geq & 1 + h_r & \text{Induktion} \end{array}$$

2. Fall: $h_r > h_s$
Dann $h = h_r \leq \log_2(n_r) \leq \log_2(n)$.

□

Aus dem Lemma folgt: Find(x) hat worst case Laufzeit $O(\log |S|)$. Kann diese Laufzeit von Find(x) verbessert werden?



Den angegebenen Pfad von r nach x durchläuft man nun zweimal (einmal von x nach r und dann wieder zurück). Beim Zurücklaufen erhalten alle Knoten auf dem Weg von r nach x mitgeteilt, dass r ihre Wurzel ist. Zugriffe auf diese Knoten können dann im nächsten Schritt in konstanter Zeit erfolgen!

Dies ändert zwar nichts an der worst case Laufzeit, allerdings kann man nun zeigen, dass sich die Laufzeit im average case dadurch verbessert. Die vorgestellt Methode wird hierbei *Path Compression* genannt.

Analyse von **Path Compression**:
Bei Find(x) werden alle Knoten auf dem Pfad von x zur Wurzel direkt unter die Wurzel gehängt. Für die Analyse betrachtet man Folgen von m (beliebigen) Union / Find Aufrufen nach einem Init auf eine n-elementige Menge.

Eine triviale Abschätzung ist $O(m \log n)$. Mit etwas mehr Mühe (s. Hauptstudium) erhält man: $O(m \log^* n)$. Hierbei bedeutet $\log^* n$ die Anzahl der $\log$ Operationen, die hintereinander angewendet werden müssen, bis das Ergebnis ≤ 1 ist. Mit viel mehr Mühe erhält man: $O\left(m \cdot \alpha(m,n)\right)$. Hierbei ist $\alpha(m,n)$ die Inverse der Ackermannfunktion, und definiert als:

$$\alpha(m,n) := min\{i : A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \log_2 n\}$$

Man bemerke hierbei, dass die Ackermann-Funktion (s. Beispiel 2.5 auf Seite 42) *sehr* schnell wächst (und nicht primitiv rekursiv ist). Entsprechend langsam wächst die Inverse dieser Funktion

Für Spezialfälle (Details s. Hauptstudiumsvorlesung Effiziente Algorithmen und Datenstrukturen [5]) geht es noch besser. Insbesondere kann man damit Kruskals Algorithmus in

$$\underbrace{O(m \log n)}_{\text{Sortieren}} + \underbrace{O(m)}_{\text{Schleife}}$$

implementieren. Für praktische Anwendungen ist bereits der Faktor $\log n$ sehr kritisch.

3.5 Graphenalgorithmen

3.5.1 Kürzeste Pfade

Problemstellung:
Gegeben: Graph $G = (V, E)$, $|V| = n$, $|E| = m$, wobei eine Längenfunktion $l : E \to \mathbb{Z}$; $s, t \in V$
Gesucht: ein bezüglich l kürzester Pfad von s nach t

Aus der Vorlesung Diskrete Strukturen I (s. auch [7]) wissen wir: Kürzeste Pfade in ungewichteten Graphen kann man mit Breitensuche in $O(m + n)$ bestimmen.

Algorithmus von Dijkstra

Dieser Algorithmus funktioniert nur für Graphen mit $l \geq 0$. Es ist nicht immer möglich, die Kantengewichte auf solche Werte zurückzuführen (auch das Addieren einer großen Zahl bei negativen Gewichten hilft dann nicht in jedem Fall weiter).

Verwende folgende Datenstrukturen:

- $W :=$ Werte der Knoten x, zu denen ein kürzester $s - x$ Pfad bekannt ist.

- $\rho[x]$: Array, um die Länge des kürzesten schon bekannten $s - x$ Pfades zu speichern.

Der Algorithmus von Dijkstra stellt sich nun folgendermaßen dar (wobei $\Gamma(s)$ die Nachbarschaft eines Knotens s bezeichne):

DIJKSTRA(G, ℓ, s, t)
$\star$ **Initialisierung** $\star$

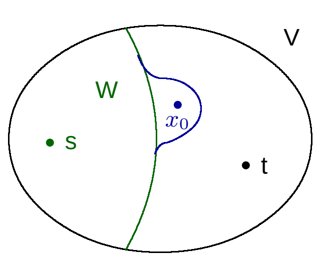
$$W := \{s\};$$

$$\rho[v] := \begin{cases} 0 & \text{falls } v = s \\ \ell(s,v) & \text{falls } v \in \Gamma(s) \\ \infty & \text{sonst} \end{cases}$$

$$\text{pred}[v] := \begin{cases} s & \text{falls } v \in \Gamma(s) \\ \text{nil} & \text{sonst} \end{cases}$$

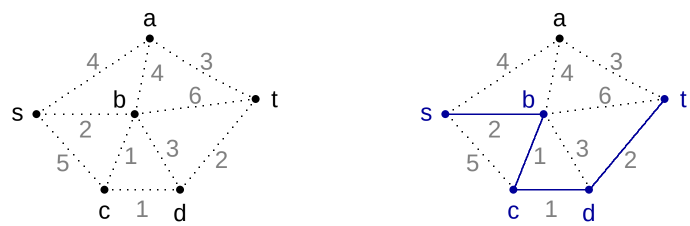
$\star$ **Hauptschleife** $\star$
while $t \notin W$ **do**
  Wähle $x_0 \in V \setminus W$ so dass $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$.
  $W := W \cup \{x_0\}$;
  for all $v \in \Gamma(x_0) \cap (V \setminus W)$ so dass $\rho[v] > \rho[x_0] + \ell(x_0, v)$ **do**
    $\rho[v] := \rho[x_0] + \ell(x_0, v)$; pred$[v] := x_0$;
$\star$ **Ausgabe** $\star$
return t, pred$[t]$, pred[pred$[t]$]$, \ldots, s$.

Sei W die Menge der Knoten aus V, für die die kürzesten Pfade bekannt sind. In der Schleife des Algorithmus von Dijktra werden folgende Operationen ausgeführt:



- Suche einen Knoten $x_0 \in V \setminus W$ mit minimalem Abstand zu s.
- Füge x_0 zu W hinzu.
- Aktualisiere $\rho[v]$, die Liste mit der Länge des kürzesten bekannten s-v-Pfades für jeden Knoten v, pred$[v]$, die Liste mit den Vorgängern $\Gamma(v)$ von v in dem s-v-Pfad.

Beispiel 3.13
Im Folgenden wollen wir uns die Arbeitsweise des Algorithmus an einem Beispiel verdeutlichen. Für den linken der beiden Graphen soll der kürzeste Pfad von s nach t bestimmt werden. Das Ergebnis ist rechts zu sehen.



Initialisierung: $W = \{s\}$

	s	a	b	c	d	t
$\rho[v]$	0	4	2	5	∞	∞
pred$[v]$	nil	s	s	s	nil	nil

1. Iteration: $x_0 = b$, $W = \{s, b\}$

	s	a	b	c	d	t
$\rho[v]$	0	4	2	3	5	8
pred$[v]$	nil	s	s	b	b	b

2. Iteration: $x_0 = c$, $W = \{s, b, c\}$

	s	a	b	c	d	t
$\rho[v]$	0	4	2	3	4	8
pred$[v]$	nil	s	s	b	c	b

3. Iteration: $x_0 = a$, $W = \{s, b, c, a\}$

	s	a	b	c	d	t
$\rho[v]$	0	4	2	3	4	7
pred$[v]$	nil	s	s	b	c	a

4. Iteration: $x_0 = d$, $W = \{s, b, c, a, d\}$

	s	a	b	c	d	t
$\rho[v]$	0	4	2	3	4	6
pred$[v]$	nil	s	s	b	c	d

5. Iteration: $x_0 = t$. Der kürzeste Pfad kann nun rückwärts (jeweils über die Vorgänger) abgelesen werden: t, d, c, b, s mit Länge 6.

Beweis: (Korrektheitsbeweis)
Wir zeigen: Vor und nach jeder Iteration gilt:

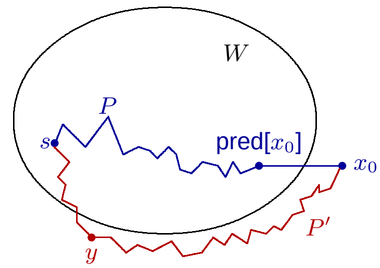
1. $\forall w \in W$:
   $\rho[w] =$ Länge eines kürzesten $s - w$ Pfades.

2. $\forall x \in V \setminus W$:
   $\rho[x] =$ Länge eines kürzesten $s - x$ Pfades, der als innere Knoten nur Knoten aus W enthält.

Zu Beginn: Hier trifft die oben angegebene Behauptung auf jeden Fall zu.

Schritt: Betrachte Schritt $W \leftarrow W \cup \{x_0\}$.

(1): $\rho[x_0] =$ Länge eines kürzesten $s - x_0$ Pfades mit Knoten aus W.



Angenommen es gibt doch einen anderen, kürzeren $s - x_0$ Pfad P'. Dann muss dieser Pfad mindestens einen Knoten enthalten, der außerhalb von W liegt. Sei y nun der erste Knoten auf P' außerhalb von W.

Dann gilt nach Wahl von x_0: $\rho[y] \geq \rho[x_0]$.

$\Longrightarrow l(P') = \rho[y] +$ Länge des Teilstücks $y - x_0 \geq \rho[x_0] = l(P)$ (hier wird wesentlich die Eigenschaft verwendet, dass für alle Kanten $l \geq 0$ ist).

Dies ist aber ein Widerspruch dazu, dass der Pfad P' kürzer ist als P.

(2): Wir müssen nun auch Pfade berücksichtigen, die den Knoten x_0 enthalten. $\rightsquigarrow$ **for all**–Schleife.

$\square$

Bemerkung: Der Beweis zeigt: Der Algorithmus von Dijkstra funktioniert nur, falls $l \geq 0$ ist.

Laufzeit: Hängt *wesentlich* von der Wahl der Datenstrukturen ab:
Verwendet man ein Array, so gilt:

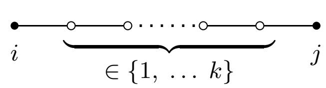
$$\underbrace{n}_{\text{Schleife}} \cdot \big(\underbrace{O(n)}_{\text{min}} + \underbrace{O(n)}_{\text{for all}}\big) = O\left(n^2\right)$$

Unter Verwendung von sogenanten *Priority Queues* (Vorrangwarteschlangen):
Implementiere $\rho[\ ]$ als Priority Queue, wobei der Schlüssel eines $v \in V$ genau der Wert $\rho[v]$ ist. Man erhält:

	Anzahl Aufrufe	Kosten pro Aufruf	
		Bin.Heaps	Fib. Heaps
Insert	n	$O(\log n)$	$O(1)$
DeleteMin	n	$O(\log n)$	$O(\log n)$
DecreaseKey	m pro Kante ≤ 1 Aufruf	$O(\log n)$	$O(1)$
Insgesamt		$O\left((n+m) \cdot \log n\right)$	$O(n \log n + m)$

Der Algorithmus von Floyd-Warshall

Ziel: Bestimme kürzeste Pfade zwischen allen Paaren von Knoten.
Ansatz: Verwende dynamische Programmierung.



$$V = \{1, \ldots, n\}$$

$$F^k[i,j] := \text{Länge eines kürzesten } i - j \text{ Pfades mit Zwischenknoten} \in \{1, \ldots, k\}$$

Initialisierung: $F^0[i,j] = \begin{cases} l(\{i,k\}) & \{i,j\} \in E \\ 0 & i = j \\ \infty & \text{sonst} \end{cases}$

Rekursion: Hier ergibt sich folgender Algorithmus:
for $k = 1$ **to** n **do**
$\forall i,j: \ F^k[i,j] = min\{F^{k-1}[i,j], F^{k-1}[i,k] + F^{k-1}[k,j]\}$
Ausgabe: $F^n[i,j]$

Beweis: Der Korrektheisbeweis sei dem Leser als Übungsaufgabe überlassen. □

Laufzeit: Einsichtig ist: $O\left(n^3\right)$.

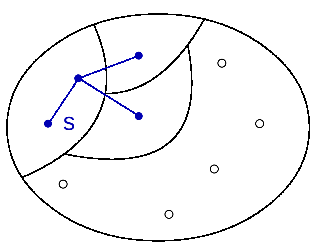
Bemerkung:

- Dieser Algorithmus funktioniert auch bei negativen Kantengewichten. Details hierzu können z.B. den Tutoraufgaben auf Übungsblatt 9 zur Volesung "Einführung in die Informatik IV" von Prof. Dr. Steger im Sommersemester 2000 entnommen werden.
- Man beachte die Ähnlichkeit zum CYK-Algorithmus. Beide Algorithmen beruhen auf Dynamischer Programmierung.

3.5.2 Minimale Spannbäume

Hier kennen wir bereits den Algorithmus von Kruskal, dessen Laufzeit $O(m \log n)$ ist ($m = |E|$, $n = |V|$). Eine Alternative ist der Algorithmus von Prim.

Bei dem Algorithmus von Prim wird der MST folgendermaßen bestimmt: Um den Algorithmus zu starten, wählen wir einen Knoten aus dem Graphen G aus. Dieser Knoten bildet nun das erste Element der Menge W. W ist dabei die Menge der Knoten aus G, für die der MST schon bestimmt wurde.



Nunmehr vergrößern wir W derart, dass wir immer die billigste Kante, die aus W herausführt, auswählen und ihren Endknoten in W einfügen.
Wurden alle Knoten von G in W eingefügt, so bilden die Knoten aus W zusammen mit den gewählten Kanten den MST.

Der Algorithmus von Prim funktioniert damit fast genauso wie der Algorithmus von Dijkstra. Und dementsprechend ähnlich ist er diesem:

PRIM(G,ℓ)
⋆ **Initialisierung** ⋆
Wähle einen Knoten $s \in V$ beliebig;
$W := \{s\}$;

$$\begin{array}{lcl} W & := & \{s\}; \\ T & := & (V,\emptyset);\ \text{// } T \text{ ist der zu berechnende, minimale Spannbaum} \\ \rho[v] & := & \begin{cases} 0 & \text{falls } v = s \\ \ell(s,v) & \text{falls } v \in \Gamma(s) \\ \infty & \text{sonst} \end{cases} \\ \text{pred}[v] & := & \begin{cases} s & \text{falls } v \in \Gamma(s) \\ \text{nil} & \text{sonst} \end{cases} \end{array}$$

⋆ **Hauptschleife** ⋆
while $W \neq V$ **do**
Wähle $x_0 \in V \setminus W$, so dass $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;

$W := W \cup \{x_0\}$; Füge Kante $\{x_0, \text{pred}[x_0]\}$ zu T hinzu;
for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \ell(x_0, v)$ **do**
$\rho[v] := \ell(x_0, v)$; pred$[v] := x_0$;
$\star$ **Ausgabe** $\star$
return T

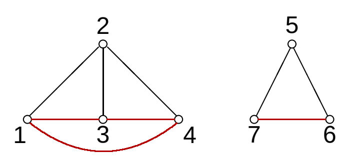
Die Laufzeit des Algorithmus von Prim ist: $O(n \log n + m)$.

3.5.3 Transitive Hülle

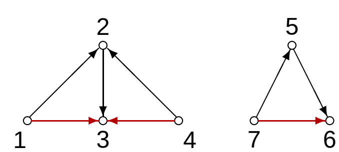
Problemstellung:
Gegeben: Gerichteter Graph $D = (V, A)$
Gesucht: Transitive Hülle von D, d.h. Graph $D_T = (V_T, A_T)$, wobei

$$\{x, y\} \in A_T \iff \exists \text{gerichtete } x - y \text{ Pfad in } D$$

Bemerkung: Für *ungerichtete* Graphen ist dies einfach: wir bestimmen mit BFS (oder DFS) die Zusammenhangskomponenten und fügen eine Kante zwischen je zwei Knoten in der gleichen Zusammenhangskomponente ein.



Die Laufzeit beträgt: $\underbrace{O(n+m)}_{\text{BFS}} + O(|E_T|)$
Für gerichtete Graphen ist dieses Problem viel schwieriger, vgl. Vorlesung Effiziente Algorithmen und Datenstrukturen, [5].



Kapitel 4

Komplexitätstheorie

Motivation: Bilder aus dem Buch von Garey & Johnson, *Computers and Intractability* (1979), s. auch [2].

4.1 Definitionen

Definition 4.1 *Sei $\Sigma = \{0,1\}$ und M eine deterministische Turingmaschine. Wir setzen:*

- $\mathsf{TIME}_M(x) :=$ *# Schritte, die M bei Eingabe x durchführt*
- $\mathsf{DTIME}(f(x)) :=$ *Menge aller Sprachen, für die es eine deterministische Mehrband–Turingmaschine M gibt mit $\mathsf{TIME}_M(x) \le f(|x|)\ \forall x \in \Sigma^*$.*

$$\mathcal{P} = \bigcup_{p \text{ Polynom}} \mathsf{DTIME}(p(n))$$

Man sagt auch: $\mathcal{P}$ enthält die polynomiell lösbaren Probleme bzw. $\mathcal{P}$ entspricht den effizient lösbaren *Problemen.*

Beispiel 4.1
2-COLORING
Gegeben: Graph $G = (V, E)$
Frage: Gilt $\chi(G) \le 2$?
Es gilt: 2-COLORING $\in \mathcal{P}$. Denn: $\chi(G) \le 2 \iff G$ bipartit.

Definition 4.2 *Für nichtdeterministische Turingmaschinen definieren wir:*

- $\mathsf{TIME}_M(x) = \begin{cases} \text{Minimale \# Schritte, die } M \text{ für eine} & \\ \text{ akzeptierende Berechnung benötigt} & x \in L(M) \\ 0 & x \notin L(M) \end{cases}$
- $\mathsf{NTIME}(f(x)) :=$ *Menge aller Sprachen, für die es eine nichtdeterministische (Mehrband-) Turingmaschine M gibt mit $\mathsf{TIME}_M(x) \le f(|x|)\ \forall x \in \Sigma^*$.*

$$\mathcal{NP} = \bigcup_{p \text{ Polynom}} \mathsf{NTIME}(p(n))$$

Beachte: Bei $\mathcal{NP}$-Problemen wird nur gefordert, dass es eine akzeptierende Berechnung *gibt*, die nur polynomiell lange dauert. Es wird aber nichts darüber gesagt, *wie* man diese finden kann.
Eine alternative Formulierung der Definition von $\mathcal{NP}$ wäre:

Definition 4.3

$L \in \mathcal{NP} \iff$ *jedem $x \in \Sigma^*$ kann man eine Menge von* Lösungen $sol(x) \subseteq \Sigma^*$ *zuordnen, so dass gilt:*
1. $x \in L \Leftrightarrow sol(x) \neq \emptyset$
2. $\exists$ deterministische *Turingmaschine, die bei Eingabe (x, y) in polynomiell (in $|x|$) vielen Schritten entscheidet, ob $y \in sol(x)$.*

Beispiel 4.2
k-COLORING
Gegeben: Graph $G = (V, E)$
Frage: Gilt $\chi(G) \leq k$?
Das Problem ist in $\mathcal{NP}$. Setze bespielsweise:

$$sol(G) = \{c : V \to \{1, \ldots, k\} \mid c(u) \neq c(v) \; \forall \{u, v\} \in E\}$$

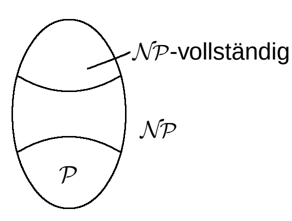
Beispiel 4.3
COLORING
Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$
Frage: Gilt $\chi(G) \leq k$?
Auch dieses Problem gehört zu $\mathcal{NP}$.

Beispiel 4.4
SAT
Gegeben: Boolsche Formel F in konjunktiver Normalfom (KNF, CNF)
Frage: Gibt es eine erfüllende Belegung für F?
Eine Beispiel-Formel wäre (x_1, x_2, x_3 sind Literale):

$$F = \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{\text{Klausel}} \wedge (x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_3})$$

Eine erfüllende Belegung wäre: $x_1 =$ WAHR, $x_2 =$ WAHR, $x_3 =$ FALSCH ist erfüllende Belegung.

Eine der wichtigsten offenen Fragen der Informatik ist: Gilt $\mathcal{P} = \mathcal{NP}$?
Klar ist: $\mathcal{P} \subseteq \mathcal{NP}$.
Vermutung: Nein!



4.2 NP-Vollständigkeit

Definition 4.4 *Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heisst A auf B polynomiell reduzierbar (Schreibweise $A \leq_p B$), falls es eine totale, polynomiell berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt mit*

$$x \in A \iff f(x) \in B \;\; \forall x \in \Sigma^*$$

Bemerkung:

1. Relation $\leq_p$ ist transitiv, d.h.
$$A \leq_p B \wedge B \leq_p C \Longrightarrow A \leq_p C$$
2. (a) $A \leq_p B \wedge B \in \mathcal{P} \Longrightarrow A \in \mathcal{P}$
   (b) $A \leq_p B \wedge B \in \mathcal{NP} \Longrightarrow A \in \mathcal{NP}$

Definition 4.5

- *Eine Sprache A heisst $\mathcal{NP}$-schwer, falls für alle Sprachen $L \in \mathcal{NP}$ gilt: $L \leq_p A$.*
- *Eine Sprache A heisst $\mathcal{NP}$-vollständig, falls $A \in \mathcal{NP}$ und A $\mathcal{NP}$-schwer ist.*

Intuitiv:

- Die $\mathcal{NP}$-vollständigen Probleme sind "die schwersten" Probleme in $\mathcal{NP}$.
- Ein $\mathcal{NP}$-schweres Problem ist und "mindestens so schwierig" wie *jedes* Problem in $\mathcal{NP}$. Beachte, dass $\mathcal{NP}$-schwere Probleme nicht unbedingt $\in \mathcal{NP}$ sein müssen.

Bemerkung: $\mathcal{NP}$-schwere (engl. $\mathcal{NP}$-hard) Probleme werden im Deutschen zuweilen auch als $\mathcal{NP}$-hart bezeichnet.
Der Begriff $\mathcal{NP}$-Vollständigkeit wurde 1971 von Cook eingeführt. Er bewies folgenden Satz, der unabhängig davon auch 1973 von Levin gezeigt wurde:

Satz 4.1 SAT *ist $\mathcal{NP}$-vollständig.*

Beweis: Zu zeigen ist:

1. SAT $\in \mathcal{NP}$. Dies wurde bereits oben gezeigt.
2. $\forall L \in \mathcal{NP}$ gilt: $L \leq_p$ SAT

   Sei $L \in \mathcal{NP}$ beliebig. Wir zeigen $L \leq_p$ SAT, d.h. wir müssen zeigen: es gibt eine polynomiell berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ mit
$$x \in L \Longleftrightarrow f(x) \in \text{SAT}$$

   Anders ausgedrückt: wir müssen uns eine Konstruktion ausdenken, die aus $x \in \Sigma^*$ eine Boolesche Formel F_x erzeugt, so dass gilt:
$$x \in L \Longleftrightarrow F_x \text{ erfüllbar}$$

   Idee: Nach Annahme ist $L \in \mathcal{NP}$. Also gibt es eine nichtdeterministische Turingmaschine M für L und ein Polynom p mit
$$\text{TIME}_M(x) \leq p(|x|) \ \ \forall x \in \Sigma^*$$

   Ziel: Konstruiere aus M für jedes $n \in \mathbb{N}$ eine Formel F mit Variablen $x_1, \ldots, x_n, z_1, \ldots, z_{q(n)}$, wobei $q()$ ein Polynom ist, so dass:

M akzeptiert $x = x_1 x_2 \ldots x_n$	$\Longleftrightarrow$	$\exists$ Belegung für $z_1, \ldots, z_{q(n)}$, so dass $x_1, \ldots, x_n, z_1, \ldots, z_{q(n)}$ erfüllende Belegung für F ist.

   Wie macht man das? Wir müssen folgende Variablen einführen:

- $P^i_{s,t} \hat{=}$ s-te Zelle des Bandes enthält zum Zeitpunkt t das Symbol i.
- $Q^j_t \hat{=}$ Turingmaschine befindet sich zum Zeitpunkt t im j-ten Zustand.
- $S_{s,t} \hat{=}$ Schreib- / Lesekopf der Turingmaschine ist zum Zeitpunkt t an Position s.

wobei

- $i \in \Gamma = \{0, 1, \Box\}$,
- $1 \leq t \leq t^* = p(n)$ mit $n = |x|$,
- $1 \leq s \leq t^*$,
- $0 \leq j \leq k$ und
- $Z = \{z_0, \underbrace{z_1, \dots, z_e}_{=E}, z_{e+1}, \dots z_k\}$.

Klar: # Variablen ist polynomiell in $|x|$. Wir müssen sicherstellen, dass die Variablen auch die intendierte Eigenschaft haben. Dazu benötigen wir noch Klauseln:

$$\text{Formel } \mathcal{F} = A \wedge B \wedge C \wedge D \wedge E \wedge F$$

Teilformel A**:** stellt sicher, dass zu jedem Zeitpunkt der Schreib– / Lesekopf an genau einer Stelle ist.

$$A = A_1 \wedge \dots \wedge A_{t^*}$$

$$A_\gamma = (S_{1,\gamma} \vee \dots \vee S_{t^*,\gamma}) \wedge \bigwedge_{\alpha \neq \beta} (S_{\alpha,\gamma} \Longrightarrow \neg S_{\beta,\gamma})$$

Teilformel B**:** stellt sicher, dass zu jedem Zeitpunkt jede Zelle genau ein Symbol enthält. Die Details sind analog zu A.

Teilformel C**:** zu jedem Zeitpunkt ist Turingmaschine in genau einem Zustand. Die Details sind wiederum analog zu A.

Teilformel D**:** stellt sicher, dass die Turingmaschine zum Zeitpunkt $t = 1$ im Zustand z_0 ist mit Schreib– / Lesekopf an Position 1 und Eingabe $x_1, \dots, x_n$ an Position $1, \dots, n$ des Bandes.

$$D = P^{x_1}_{1,1} \wedge \dots \wedge P^{x_n}_{n,1} \wedge P^{\Box}_{n+1,1} \wedge \dots \wedge P^{\Box}_{t^*,1} \wedge Q^0_1 \wedge S_{1,1}$$

Teilformel E**:** spätestens zum Zeitpunkt t^* ist die Turingmaschine in einem akzeptierenden Zustand (aus $\{z_1, \dots, z_e\}$, s. oben).

$$E = \left(Q^1_1 \vee Q^2_1 \vee \dots \vee Q^e_1\right) \vee \left(Q^1_2 \vee \dots \vee Q^e_2\right) \vee \dots \vee \left(Q^1_{t^*} \vee \dots \vee Q^e_{t^*}\right)$$

Teilformel F**:** modelliert Übergänge der Turingmaschine (also δ).
$\forall(\underbrace{z_j}_{\in Z}, \underbrace{\sigma}_{\in \Gamma})$ betrachte Menge der Tupel

$$(z_{i_k}, \sigma_{i_k}, \underbrace{m_{i_k}}_{\{-1,0,1\}}) \ni \delta(z_k, \sigma)$$

wobei $1 \leq k \leq |\delta(z_j, \sigma)| = r$.

Teilformel F enthält für jedes solche Tupel (z_j, σ):

$$\bigvee_{1\leq s\leq t^*} \Big[\Big(Q_t^j \wedge S_{s,t} \wedge P_{s,t}^{\sigma}\Big) \implies \big(\big(Q_{t+1}^{i_1} \wedge S_{s+m_{i_1},t+1} \wedge P_{s,t+1}^{\sigma_{i_1}}\big) \vee \ldots \ldots \vee \big(Q_{t+1}^{i_r} \wedge S_{s+m_{i_r},t+1} \wedge P_{s,t+1}^{\sigma_{i_r}}\big)\big)\Big]$$

□

Als nächstes sollen einige Beispiele für $\mathcal{NP}$-vollständige Probleme angeführt werden:

Beispiel 4.5
3 SAT
Gegeben: Boolesche Formel in KNF, wobei jede Klausel höchstens drei Literale enthält.
Frage: Gibt es eine erfüllende Wahrheitsbelegung?

Satz 4.2 3 SAT *ist $\mathcal{NP}$-vollständig.*

Bemerkung: Es gilt: 2 SAT $\in \mathcal{P}$. Die Angabe eine polynomiellen Algorithmus für 2 SAT sei dem Leser hierbei als Übungsaufgabe überlassen.

Beweis:

1. Klar ist: 3 SAT $\in \mathcal{NP}$.

2. Zu zeigen bleibt: $\forall L \in \mathcal{NP}$: $L \leq_p$ 3 SAT

   Wir wissen: $\leq_p$ ist transitiv und SAT ist $\mathcal{NP}$ vollständig.

   D.h. es genügt zu zeigen: SAT $\leq_p$ 3 SAT.

   Sei F eine beliebige SAT Formel: Wir transformieren jede Klausel von F in eine 3 SAT Formel wie folgt:

   Eine Klausel
   $$(x_1 \vee \ldots \vee x_k)$$
   wird umgeformt zu
   $$(x_1 \vee z_1) \wedge (\overline{z_1} \vee x_2 \vee z_2) \wedge (\overline{z_2} \vee x_3 \vee z_3) \wedge \ldots \wedge (\overline{z_{k-2}} \vee x_{k-1} \vee z_{k-1}) \wedge (\overline{z_{k-1}} \vee x_k)$$

   Dann gilt:

   Alle x_i haben den Wert "falsch" $\implies$ Es gibt keine Belegung für z_i, so dass alle neuen Klauseln erfüllt sind.

   Ist aber mindestens ein x_i wahr, dann gibt es eine erfüllende Belegung für die z_j's.

   Klar ist zusätzlich:

   Es werden nur polynomiell mehr Variablen benötigt, ebenso wird auch die Formel nur polynomiell größer.

□

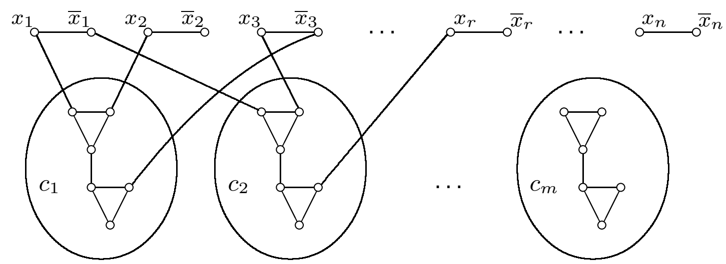
Beispiel 4.6
3 COL
Gegeben: Graph $G = (V, E)$
Frage: Gilt $\chi(G) \leq 3$?

Satz 4.3 3 COL *ist $\mathcal{NP}$-vollständig.*

Beweis:

1. Offensichtlich ist 3 COL $\in \mathcal{NP}$.

2. Wir zeigen: 3 SAT $\leq_p$ 3 COL. Zu einer gegebenen Boole'schen Formel F mit Variablen $x_1, \ldots, x_n$ konstruieren wir einen Graphen G wie folgt: Wir erzeugen zunächst $2n$ Knoten mit den Variablen $(x_1, \overline{x}_1, x_1, \overline{x}_1, \ldots x_n, \overline{x}_n)$, wobei jeweils x_i und $\overline{x}_i$ durch eine Kante verbunden sind. Weiterhin enthält der Graph die Klauseln C_1 bis C_m, die wie in der nachfolgenden Skizze konstruiert werden. Die Klauseln sind mit den in ihnen enthaltenen Variablen durch Kanten verbunden.

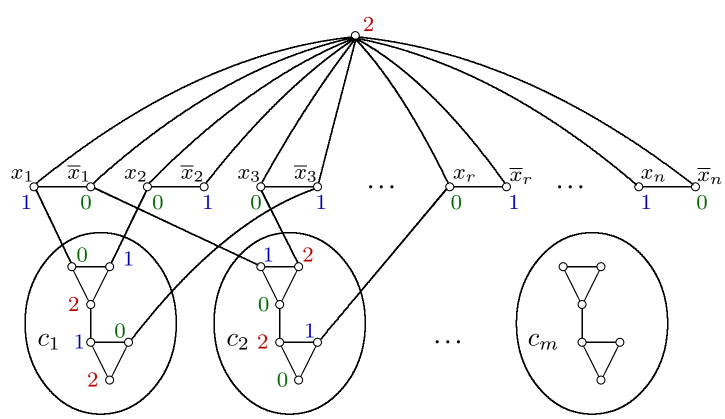


Die oben dargestellten Klauseln c_1 und c_2 ergeben sich also zu:

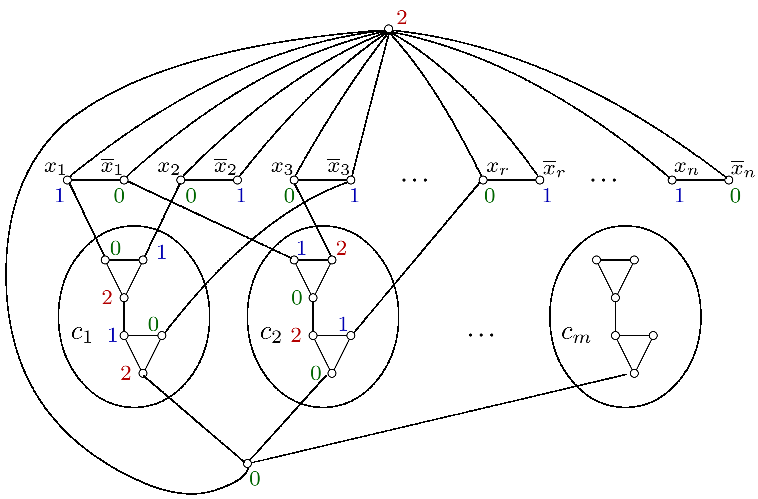
$$c_1 = x_1 \vee x_2 \vee \overline{x}_3$$

$$c_2 = \overline{x}_1 \vee x_3 \vee x_r$$

Um sicherzustellen das alle Variablen mit nur zwei Farben (0,1) gefärbt werden können verbinden wir sie jeweils mit einem zusätzlichen Knoten, der die Färbung 2 erhält. Haben alle Variablen mit denen eine Klausel durch Kanten verbunden sind die Färbung 0, ist also die Klausel nicht erfüllt, so hat auch der "Ausgang" die Farbe 0. Hat hingegen mindestens eine der Variablen die Farbe 1, ist also die Klausel erfüllt, so kann man die Klausel so färben, das der "Ausgang" die Farbe 1 oder 2 hat.



Um sicherzustellen das der Graph nur Klauseln enthält, deren "Ausgang" den Wert 1 oder 2 hat, erweitern wir den Graphen um einen weiteren Knoten – mit Färbung 0. Diesen Knoten verbinden wir mit jedem Ausgang, so dass sich folgendes Bild ergibt:



Nun gilt:

$$F \text{ erfüllbar} \Longleftrightarrow \text{Graph } G \text{ ist mit 3 Farben färbbar}$$

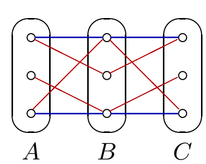
"$\Rightarrow$": Wie dargestellt färbe man den obersten Knoten mit Farbe 2 und den untersten Knoten mit Farbe 0. Da eine erfüllende Belegung für F existiert, kann der Rest des Graphen so gefärbt werden, dass insgesamt nur 3 Farben benötigt werden.

"$\Leftarrow$": Sei nun eine 3-Färbung von G gegeben. O.E. hat der oberste Knoten die Farbe 2 und der unterste Knoten die Farbe 0 (andernfalls können die Farben entsprechend permutiert werden). Dann muss die Färbung der Literale einer erfüllenden Belegung entsprechen.

□

Beispiel 4.7
3-partites Matching
Gegeben: Gegeben drei paarweise disjunkte Mengen X, Y und Z und eine Menge $S \subseteq X \times Y \times Z$.
Frage: Gibt es ein *perfektes Matching*, also eine Teilmenge $M \subseteq S$, so dass jedes Element $v \in X \cup Y \cup Z$ in *genau einem Element* von M enthalten ist?

Satz 4.4 MaxMatching *in Graphen kann man in polynomieller Zeit konstruieren, s. auch Vorlesung Effiziente Algorithmen und Datenstrukturen, [5].*



Satz 4.5 3-partites Matching *ist $\mathcal{NP}$-vollständig.*

Beweis:

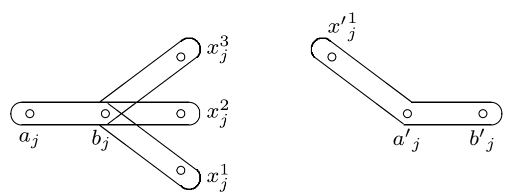
1. Offensichtlich ist 3-partites Matching $\in \mathcal{NP}$.

2. Wir zeigen wieder 3 SAT $\leq_p$ 3-partites Matching.

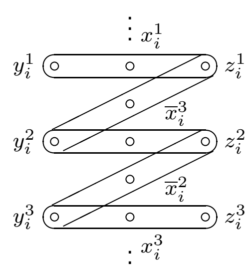
$$
\begin{aligned}
X &:= \{x_i^j, \bar{x}_i^j \mid 1 \leq i \leq n, 1 \leq j \leq m\} \\
Y &:= \{y_i^j \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{a^j \mid 1 \leq j \leq m\} \\
&\quad \cup \{c_k \mid 1 \leq k \leq (n-1)m\} \\
Z &:= \{z_i^j \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{b^j \mid 1 \leq j \leq m\} \\
&\quad \cup \{d_k \mid 1 \leq k \leq (n-1)m\} \\
S_1 &:= \{(x_i^j, y_i^j, z_i^j), (\bar{x}_i^j, y_i^{j+1}, z_i^j) \mid 1 \leq i \leq n, 1 \leq j \leq m-1\} \\
&\quad \cup \{(x_i^m, y_i^m, z_i^m), (\bar{x}_i^m, y_i^1, z_i^m) \mid 1 \leq i \leq n\} \text{ (wobei } y_i^{m+1} = y_i^1) \\
S_2 &:= \{(\lambda^j, a^j, b^j) \mid 1 \leq j \leq m,\ \lambda \text{ Literal von } C_j\} \\
S_3 &:= \{(x, c_k, d_k) \mid x \in X,\ 1 \leq k \leq (n-1)m\} \\
S &:= S_1 \cup S_2 \cup S_3
\end{aligned}
$$

Sei also F eine Boolesche Formel mit Variablen $x_1, \ldots, x_n$ und Klauseln $c_1, \ldots, c_m$.
Ziel: Konstruiere X, Y, Z und S, so dass gilt

$$F \text{ erfüllbar} \iff \exists \text{ perfektes Matching}$$



Für Variablen x_i:



□

Beispiel 4.8
Subset Sum
Gegeben: Natürliche Zahlen $a_1, \ldots, a_n$ und K.
Frage: Gibt es ein $I \subseteq \{1, \ldots, n\}$, so dass $\sum_{i \in I} a_i = K$?

Satz 4.6 Subset Sum *ist $\mathcal{NP}$-vollständig.*

Korollar 4.1 Knapsack *ist $\mathcal{NP}$-vollständig.*

Aber: $\exists\ O(n^2 \cdot p_{max})$ Algorithmus für Knapsack, s. hierzu auch [7]. Dies ist *kein* Widerspruch, da die Eingabe des Algorithmus $\log$ der Profite ist, damit ist p_{max} exponentiell von der Eingabegröße abhängig.

Beispiel 4.9
Clique
Gegeben: Graph $G = (V, E)$, Zahl $K \in \mathbb{N}$.
Frage: Enthält G einen vollständigen Subgraphen der Größe K?
k-Clique
Gegeben: Graph $G = (V, E)$.
Frage: Enthält G einen vollständigen Subgraphen der Größe k?

Satz 4.7 Clique *ist* $\mathcal{NP}$*-vollständig.*

Satz 4.8 k-Clique $\in \mathcal{P}\ \forall k \in \mathbb{N}$.

Beweis: **Idee:** Wir probieren alle möglichen Subgraphen der Größe k durch und testen jeweils, ob sie vollständig sind. Hierbei gibt es $\binom{n}{k}$ mögliche Subgraphen (wobei n die Anzahl der Knoten von G ist). Ebenso gilt: $\binom{n}{k} \in O(n^k)$. Da das k nun fest ist (und nicht zur Eingabegröße gehört), haben wir einen polynomiellen Algorithmus für k-Clique gefunden. $\square$

Bemerkung: Für grosse k, z.B. $k = 10^{10}$, ist der Algorithmus zwar immer noch polynomiell in der Eingabe (also in n), praktisch aber wäre ein Algorithmus mit einer Laufzeit von $O\left(n^{(10^{10})}\right)$ wohl kaum zu verwenden.

Literaturverzeichnis

[1] *Prof. Dr. Dr. h.c. Wilfried Brauer,*
Vorlesung: **Einführung in die Informatik 1 (WS 98/99)**,
WWW: http://wwwbrauer.in.tum.de/lehre/infoI/WS9899/infoI.shtml,
Skript: http://www.in.tum.de/~manthey/script.htm,
Referenzen: 2.4

[2] *M. Garey, D. Johnson,*
Computers and Intractability – A Guide to the Theory of NP-completeness,
Freeman, 1979
Referenzen: 4

[3] *V. Heun*
Grundlegende Algorithmen
Vieweg Verlag, Braunschweig – Wiesbaden, 1. Auflage, 2000

[4] *J.E. Hopcroft, J.D. Ullman,*
Introduction to automata theory, languages and computation
Addison-Wesley, 1979

[5] *Prof. Dr. Ernst W. Mayr,*
Vorlesung: **Effiziente Algorithmen und Datenstrukturen (WS 98/99)**,
WWW: http://wwwmayr.in.tum.de/lehre/1998WS/ea/,
Skript: http://wwwmayr.in.tum.de/skripten/ead_ws9899.ps.gz,
Referenzen: 3.2.7, 3.3.2, 3.3.6, 3.4, 3.5.3, 4.4

[6] *Prof. Dr. Uwe Schöning,*
Theoretische Informatik - kurzgefasst,
Spektrum Hochschultaschenbuch, Heidelberg – Berlin, 3. Auflage, 1997

[7] *Prof. Dr. Angelika Steger,*
Diskrete Strukturen, Band 1,
Springer Verlag, Berlin – Heidelberg – New York, 1. Auflage, 2001
Referenzen: 3.2.3, 3.8, 3.12, 3.5.1, 4.8

Index