Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

WS 2004/05 (Stand: 28.03.2006)

Inhalt der Vorlesung

Teil I: Automaten und formale Sprachen

Zentrale Fragestellungen

- Wie definiert man (formal) die Syntax einer Sprache (z.B. einer Programmiersprache)?
- Wie überprüft man, ob ein Wort (d.h. eine Zeichenreihe) zur Sprache gehört?

Teil II: Berechenbarkeit und Komplexität

Zentrale Fragestellungen

- Welche Probleme (Aufgabenstellungen) sind prinzipiell mit einem Computer lösbar bzw. nicht lösbar?
- Welche Probleme sind mit vernünftigem Aufwand (an Speicherplatz und Laufzeit) lösbar?

Grundlagen für beide Teile: Zeichen, Wörter und Sprachen

Definition 0.1 *Ein Alphabet oder Zeichenvorrat ist eine nichtleere endliche Menge. Die Elemente eines Alphabets nennen wir Zeichen oder Symbole.*

Konvention: Alphabete werden mit $\Sigma, \Sigma_1, \Sigma', \ldots$ bezeichnet.

Beispiele

- $\Sigma_1 = \{1\}$
- $\Sigma_2 = \{0, 1\}$
- $\Sigma_3 = \{0, \ldots, 9\}$
- $\Sigma_4 = \{a, \ldots, z, A, \ldots, Z\}$
- $\Sigma_5 =$ Menge aller ASCII-Zeichen
- $\Sigma_6 = \{\mathit{begin}, \mathit{end}, \mathit{if}, \mathit{then}, \mathit{else}, ;, :=, +, *, \ldots\}$

Definition 0.2 *Ein Wort über dem Alphabet Σ ist eine endliche Folge $w = (a_1, \ldots, a_n)$, wobei $n \geq 0$ und $a_i \in \Sigma$ für $i = 1, \ldots, n$. n heißt Länge des Wortes w, a_i heißt i-tes Zeichen von w. Das Wort $()$ heißt leeres Wort.*

Kurzschreibweise

- $a_1 \ldots a_n$ statt $(a_1, \ldots, a_n)$
  Vorsicht: Zeichen a und Wort a nicht mehr unterscheidbar!
- ε statt $()$

Weitere Schreibweisen

- $|w|$ = Länge von w
- Σ^* = Menge aller Wörter über Σ
- $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ = Menge aller nichtleeren Wörter über Σ

Beispiele

- $\{1\}^* = \{\varepsilon, 1, 11, 111, \ldots\}$
- $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$
- $\varepsilon, 0, 1000, 0001 \in \{0, \ldots, 9\}^*$
- Der Inhalt einer Datei ist *ein* Wort über dem ASCII-Alphabet (denn Leerzeichen, Zeilenwechsel etc. sind Zeichen dieses Alphabets).
- Ein Programm (in irgendeiner Programmiersprache) ist ebenfalls ein Wort über dem ASCII-Alphabet.
- Der Inhalt eines Buches ist ein Wort (über dem Alphabet der Sprache, in der es geschrieben ist).

Alternative Definition für Σ^* und Σ^+

- $\Sigma^n = \{(a_1, \ldots, a_n) \mid a_i \in \Sigma \text{ für } i = 1, \ldots, n\}$
  $= \{a_1 \ldots a_n \mid a_i \in \Sigma \text{ für } i = 1, \ldots, n\}$
  $=$ Menge aller *Wörter der Länge n* über Σ

- Speziell: $\Sigma^0 = \{()\}$
  $= \{\varepsilon\}$

- $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$
  $=$ Menge *aller Wörter* über Σ

  $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$
  $= \Sigma^* \setminus \{\varepsilon\}$

Definition 0.3 *Seien* $v = a_1 \dots a_m$ *und* $w = b_1 \dots b_n$ *Wörter über* Σ. *Dann ist die* Konkatenation $v \circ w$ *von* v *und* w *definiert durch*

$$v \circ w \;=\; a_1 \dots a_m b_1 \dots b_n$$

Kurzschreibweise: vw statt $v \circ w$

Eigenschaften der Konkatenation

- *Assoziativität*: $(uv)w = u(vw)$ für alle $u, v, w \in \Sigma^*$
- *Neutrales Element*: $w\varepsilon = \varepsilon w = w$ für alle $w \in \Sigma^*$
- Also: $(\Sigma^*, \circ)$ ist ein *Monoid* mit neutralem Element ε (vgl. natürliche Zahlen mit Multiplikation)

Definition 0.4 *Sei $w \in \Sigma^*$ und $n \in \mathbb{N}$. Dann ist die n-te Potenz w^n von w definiert durch $w^n = \underbrace{w \ldots w}_{n}$ (speziell: $a^n = \underbrace{a \ldots a}_{n}$ falls $a \in \Sigma$).*

Besser: Definition von w^n durch Induktion über n

- $w^0 = \varepsilon$
- $w^n = ww^{n-1}$ falls $n > 0$

Oder:

- $w^0 = \varepsilon$
- $w^n = w^{n-1}w$ falls $n > 0$

Beispiele

- $(bla)^2 = blabla$
- $(bla\ )^3 = bla\ bla\ bla$

Definition 0.5 *Für* $w = a_1 \dots a_n \in \Sigma^*$ *sei* $w^R \in \Sigma^*$ *das Wort, das durch "Spiegelung" aus* w *entsteht , d.h.*

$$w^R = a_n \dots a_1$$

(R *steht für engl. reverse)*

Beispiel

$(01001)^R = 10010$

Alternative

Definition von w^R durch Induktion über $|w|$
(s. Übung!)

Definition 0.6 *Sei $x \in \Sigma^*$.*

- *u heißt Anfangswort oder Präfix von x, falls ein $v \in \Sigma^*$ existiert mit $x = uv$.*
- *v heißt Endwort oder Suffix von x, falls ein $u \in \Sigma^*$ existiert mit $x = uv$.*
- *v heißt Teilwort von x, falls $u, w \in \Sigma^*$ existieren mit $x = uvw$.*

Beispiel

- Anfangswörter von 010: $\varepsilon, 0, 01, 010$
- Endwörter von 010: $\varepsilon, 0, 10, 010$
- Teilwörter von 010: $\varepsilon, 0, 1, 01, 10, 010$

Die Beziehungen ‘Anfangswort’, ‘Endwort’ und ‘Teilwort’ sind zweistellige Relationen auf Σ^*. Jede dieser Relationen ist

- *reflexiv*: Jedes $w \in \Sigma^*$ steht in Relation zu sich selbst.
- *transitiv*: Wenn u in Relation zu v und v in Relation zu w steht, dann steht auch u in Relation zu w.
- *antisymmetrisch*: Wenn u in Relation zu v steht und v in Relation zu u, dann ist $v = u$.

Also: Jede der Relationen definiert eine *partielle Ordnung* auf Σ^*.

Definition 0.7 *Eine Sprache (oder formale Sprache) über dem Alphabet Σ ist eine (beliebige) Teilmenge von Σ^*.*

Konvention:

Sprachen bezeichnen wir meist mit $L, L_1, L', \ldots$ (engl. *language*).

Alternative Formulierung

Eine Sprache über Σ ist ein Element der *Potenzmenge* $\wp(\Sigma^*)$.

Beispiele

- $L_1 = \emptyset$
- $L_2 = \Sigma^*$
- $L_3 = \{\varepsilon\}$
- $L_4 = \{1^n \mid n \text{ ist eine Primzahl}\}$
- $L_5 =$ Menge aller Binärdarstellungen natürlicher Zahlen
- $L_6 =$ Menge aller Dezimaldarstellungen natürlicher Zahlen
- $L_7 =$ Menge aller Java-Programme
- $L_8 =$ Menge aller Sätze der deutschen Sprache
- $L_9 = \{0^n 1^n \mid n \geq 0\}$
- $L_{10} = \{vw \mid v \in \{0\}^*, w \in \{1\}^*\}$

Definition 0.8 *Seien* $L, L_1, L_2 \subseteq \Sigma^*$. *Dann sei*

- $L_1 \cup L_2$ *die* Vereinigung *von* L_1 *und* L_2.
- $L_1 \cap L_2$ *der* Durchschnitt *von* L_1 *und* L_2.
- $L_1 \setminus L_2 = \{w \in L_1 \mid w \notin L_2\}$ *die* Mengendifferenz *von* L_1 *und* L_2.
- $\bar{L} = \Sigma^* \setminus L$ *das* Komplement *von* L.
- $L_1 \circ L_2 = \{vw \mid v \in L_1, w \in L_2\}$ *die* Konkatenation *von* L_1 *und* L_2.

Kurzschreibweise: $L_1 L_2$ statt $L_1 \circ L_2$

Eigenschaften der Konkatenation

- *Assoziativität*: $(L_1 L_2)L_3 = L_1(L_2 L_3)$ für alle $L_1, L_2, L_3 \subseteq \Sigma^*$
- *Neutrales Element*: $L\{\varepsilon\} = \{\varepsilon\}L = L$ für alle $L \subseteq \Sigma^*$
- Also: $(\wp(\Sigma^*), \circ)$ ist ein *Monoid* mit neutralem Element $\{\varepsilon\}$.

Definition 0.9 *Seien* $L \subseteq \Sigma^*$ *und* $n \in \mathbb{N}$. *Dann ist die* n-te Potenz L^n *von* L *definiert durch*

$$\begin{aligned} L^n &= \underbrace{L \circ \ldots \circ L}_{n} \\ &= \underbrace{L \ldots L}_{n} \\ &= \{w_1 \ldots w_n \mid w_i \in L \text{ für } i = 1, \ldots, n\} \end{aligned}$$

Besser: Definition von L^n durch Induktion über n

- $L^0 = \{\varepsilon\}$
- $L^n = LL^{n-1}$ falls $n > 0$

Oder:

- $L^0 = \{\varepsilon\}$
- $L^n = L^{n-1}L$ falls $n > 0$

Definition 0.10 *Für $L \subseteq \Sigma^*$ sei*

- $L^R = \{w^R \mid w \in L\}$ *die "Spiegelung" von* L
- $L^* = \bigcup_{n \geq 0} L^n$

  $= \{w_1 \dots w_n \mid n \geq 0, w_i \in L \text{ für } i = 1, \dots, n\}$

  der Kleene-Abschluss von L.
- *Speziell:*

  $\{w\}^* = \{w^n \mid n \geq 0\}$ *falls* $w \in \Sigma^*$

  $\{a\}^* = \{a^n \mid n \geq 0\}$ *falls* $a \in \Sigma$
- $L^+ = \bigcup_{n \geq 1} L^n$

  $= \{w_1 \dots w_n \mid n \geq 1, w_i \in L \text{ für } i = 1, \dots, n\}$

  $= LL^*$

Folgerung: $L^* = L^+ \cup L^0 = L^+ \cup \{\varepsilon\}$

Rechenregeln für Potenzen

. . . von Wörtern

- $w^0 = \varepsilon$ (neutrales Element)
- $w^1 = w$
- $w^m w^n = w^{m+n}$
- $(w^m)^n = w^{mn}$

. . . von Sprachen

- $L^0 = \{\varepsilon\}$ (neutrales Element)
- $L^1 = L$
- $L^m L^n = L^{m+n}$
- $(L^m)^n = L^{mn}$

Diese Regeln gelten in *jedem* Monoid (wenn man das Potenzieren nach dem üblichen Schema definiert). Denn sie folgen allein aus der Assoziativität und der Eigenschaft des neutralen Elements.

Warnung: Es gilt *nicht* $(w_1 w_2)^n = w_1^n w_2^n$ oder $(L_1 L_2)^n = L_1^n L_2^n$. Denn dazu bräuchte man die Kommutativität.

Rechenregeln für * und $^+$

- $L \subseteq L^*$
- $(L^*)^* = L^*$
- $L \subseteq L^+$
- $(L^+)^+ = L^+$

Mengenoperatoren mit diesen beiden Eigenschaften nennt man *Abschluss-* oder *Hüllenoperatoren*.

Beweis für *

1. $L \subseteq L^*$ ist klar wegen $L = L^1$ und $L^* = \bigcup_{n \geq 0} L^n$.
2. $L^* = (L^*)^*$:
   '$\subseteq$' ist klar wegen 1.
   '$\supseteq$': Sei $w \in (L^*)^*$, d.h. $w = w_1 \dots w_n$ mit $n \geq 0$ und $w_i \in L^*$ für $i = 1, \dots, n$. Dann existiert für jedes i ein $m_i \geq 0$ mit $w_i \in L^{m_i}$, also folgt $w = w_1 \dots w_n \in L^{m_1} \dots L^{m_n} = L^{m_1 + \dots + m_n} \subseteq L^*$. $\square$

Die Operatoren $\cup, \circ, ^*, \ldots$ lassen sich auch zur *Beschreibung* von Sprachen verwenden. Konvention: Die Postfixoperatoren ^{n}, $^+$ und * binden am stärksten, $\circ$ bindet stärker als $\cup$.

Beispiele

- Die Menge aller Dezimaldarstellungen natürlicher Zahlen:
  $L_{nat} = \{0, \ldots, 9\}^+ = \{0, \ldots, 9\}\{0, \ldots, 9\}^*$
- Die Menge aller Dezimaldarstellungen ganzer Zahlen:
  $L_{int} = \{-, \varepsilon\} L_{nat}$
- Die Menge aller dezimalen Festpunktzahlen:
  $L_{fix} = L_{int}\{.\} \cup \{.\}L_{nat} \cup L_{int}\{.\}L_{nat}$
- Die Menge aller dezimalen Fließpunktzahlen:
  $L_{float} = L_{int}\{e, E\}L_{int} \cup L_{fix}(\{\varepsilon\} \cup \{e, E\}L_{int})$

Die Menge aller Wörter über dem Alphabet $\{0,1\}$, ...

- ... die mit 00 beginnen:
  $L_1 = \{00\}\{0,1\}^*$
- ... die 00 als Teilwort enthalten:
  $L_2 = \{0,1\}^*\{00\}\{0,1\}^*$
- ... die mindestens zwei Nullen enthalten:
  $L_3 = \{0,1\}^*\{0\}\{0,1\}^*\{0\}\{0,1\}^* = \{1\}^*\{0\}\{1\}^*\{0\}\{0,1\}^*$
- ... die genau zwei Nullen enthalten:
  $L_4 = \{1\}^*\{0\}\{1\}^*\{0\}\{1\}^*$
- ... die höchstens zwei Nullen enthalten:
  $L_5 = \{1\}^*\{0,\varepsilon\}\{1\}^*\{0,\varepsilon\}\{1\}^*$
- ... deren erstes und letztes Zeichen übereinstimmen:
  $L_6 = \{0,1\} \cup \{0\}\{0,1\}^*\{0\} \cup \{1\}\{0,1\}^*\{1\}$

Definition 1.1 *Die Menge aller regulären Sprachen über Σ ist induktiv definiert durch:*

- *Jede endliche Menge $L \subseteq \Sigma^*$ ist regulär.*
- *Wenn $L_1, L_2 \subseteq \Sigma^*$ regulär sind, dann sind auch $L_1 \cup L_2$ und $L_1 \circ L_2$ regulär.*
- *Wenn L regulär ist, dann ist auch L^* regulär.*

Mit anderen Worten:

Eine Sprache ist genau dann regulär, wenn sie sich durch wiederholte Anwendung der Operatoren $\cup, \circ$ und * aus endlichen Sprachen aufbauen lässt.

Damit haben wir eine *endliche Beschreibung* für jede reguläre Sprache.

Beispiele: $L_{nat}, L_{int}, L_{fix}, L_{float}, L_1, \ldots, L_6$ sind reguläre Sprachen.

Reguläre Sprachen und endliche Automaten

Die 'Mengenausdrücke' aus Definition 1.1 sind gut lesbare Beschreibungen für reguläre Sprachen (zumindest für die Beispiele aus der Praxis wie Zahldarstellungen, Variablennamen, ...)

Aber:

- Nachteil in der Praxis:
  Ein Mengenausdruck für eine Sprache liefert noch keinen Algorithmus, um die Zugehörigkeit zur Sprache zu testen.
- Nachteil in der Theorie:
  Mit Definition 1.1 kann man nur schwer zeigen, dass eine Sprache *nicht* regulär ist.

Deshalb:

- Alternative Charakterisierungen der regulären Sprachen
- Insbesondere *algorithmische* Charakterisierungen

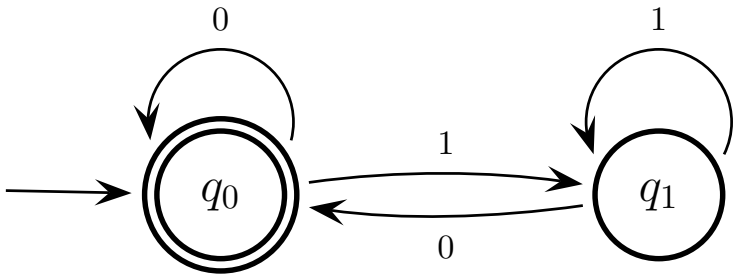
Ein *endlicher Automat* ist eine 'Maschine', die (nur) *endlich viele Zustände* annehmen kann. Einer der Zustände heißt *Startzustand*, einige Zustände heißen *Endzustände* (besser: *akzeptierende Zustände*).

Arbeitsweise

- Der Automat erhält ein Wort w als Eingabe.
- Zu Beginn befindet er sich im Startzustand.
- Er liest w zeichenweise von links nach rechts, wobei jedes Zeichen einen Zustandsübergang bewirkt.
- Ist er nach Abarbeitung von w in einem Endzustand, so wird w akzeptiert, andernfalls wird w abgelehnt.

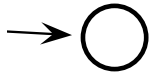
Ein endlicher Automat kann also dazu benutzt werden, die Zugehörigkeit zu einer Sprache zu testen.

Graphische Darstellung eines endlichen Automaten



Konvention:

 sind die Zustände

 ist der Startzustand

 sind die Endzustände

- **Akzeptierte Wörter:** 0, 010, 01100
- **Abgelehnte Wörter:** 1, 101, 11011

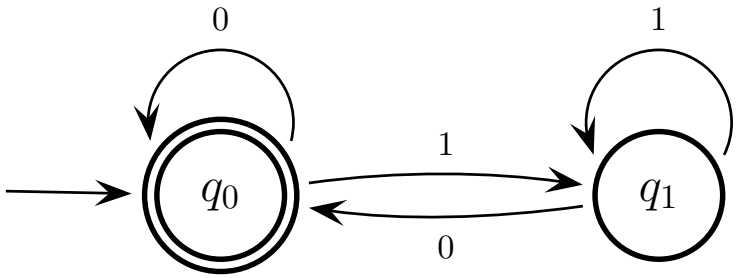
Noch zu klären:

- Darf der Startzustand zugleich Endzustand sein?
- Dürfen alle Zustände Endzustände sein?
- Dürfen die Endzustände ganz fehlen?
- Darf von einem Zustand mehr als ein Pfeil mit ein und demselben Zeichen ausgehen?
- Muss von jedem Zustand ein Pfeil mit jedem Zeichen ausgehen?

Definition 1.2 *Ein deterministischer endlicher Automat (kurz: DEA) ist ein 5-Tupel $A = (\Sigma, Q, s, F, \delta)$ mit:*

- Σ *ist ein Alphabet*
- Q *ist eine endliche Menge, deren Elemente wir Zustände nennen*
- $s \in Q$ *ist der sogenannte Startzustand*
- $F \subseteq Q$ *ist die Menge der sogenannten Endzustände oder akzeptierende Zustände*
- $\delta : Q \times \Sigma \to Q$ *ist die sogenannte Übergangsfunktion (totale Funktion!)*

Beispiel (wie oben):



$A = (\Sigma, Q, s, F, \delta)$ mit

- $\Sigma = \{0, 1\}$
- $Q = \{q_0, q_1\}$
- $s = q_0$
- $F = \{q_0\}$

- $\delta : Q \times \Sigma \to Q$
  $(q_0, 0) \mapsto q_0$
  $(q_0, 1) \mapsto q_1$
  $(q_1, 0) \mapsto q_0$
  $(q_1, 1) \mapsto q_1$

Oder: δ als Tabelle

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

Definition 1.3 *Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA.*

- *Eine Konfiguration von A ist ein Element $(q, w) \in Q \times \Sigma^*$.*
- *Auf der Menge der Konfigurationen von A definieren wir die Übergangsschrittrelation $\vdash_A$ und die Schreibweisen $\vdash_A^n$ für $n \geq 0$, $\vdash_A^+$ und $\vdash_A^*$ durch*

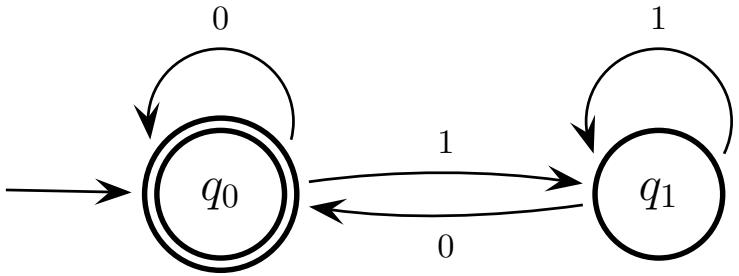
$$
\begin{array}{lcl}
(q,w) \vdash_A (q',w') & \Leftrightarrow & \text{es existiert ein } a \in \Sigma \text{ mit} \\
 & & w = aw' \text{ und } \delta(q,a) = q' \\
(q,w) \vdash_A^n (q',w') & \Leftrightarrow & \text{es existieren } (q_0,w_0),\ldots,(q_n,w_n) \in Q \times \Sigma^* \\
 & & \text{mit } (q,w) = (q_0,w_0), (q',w') = (q_n,w_n) \\
 & & \text{und } (q_0,w_0) \vdash_A \ldots \vdash_A (q_n,w_n). \\
(q,w) \vdash_A^+ (q',w') & \Leftrightarrow & \text{es existiert ein } n > 0 \text{ mit } (q,w) \vdash_A^n (q',w') \\
(q,w) \vdash_A^* (q',w') & \Leftrightarrow & \text{es existiert ein } n \geq 0 \text{ mit } (q,w) \vdash_A^n (q',w')
\end{array}
$$

Intuition

- Eine Konfiguration (q, w) beschreibt die Situation des Automaten zu einem Zeitpunkt: Er befindet sich im Zustand q und hat noch das Wort w zu lesen.
- $\vdash_A$ beschreibt den *Übergangsschritt* von einem Zeitpunkt zum nächsten: Das erste Zeichen des Wortes wird gelesen und der Automat wechselt den Zustand gemäß der Übergangsfunktion δ.
- $\vdash_A^n$ steht für eine Folge von n Übergangsschritten, $\vdash_A^+$ für eine nichtleere und $\vdash_A^*$ für eine möglicherweise leere Folge

$\vdash_A^+$ ist der *transitive Abschluss* der Relation $\vdash_A$, d.h. die kleinste transitive Relation, die $\vdash_A$ enthält.
$\vdash_A^*$ ist der *reflexive transitive Abschluss* von $\vdash_A$, d.h. die kleinste reflexive transitive Relation, die $\vdash_A$ enthält.

Beispiel (wie oben):



Eine mögliche Folge von Übergangsschritten:

$$
\begin{aligned}
(q_0, 1001) &\vdash_A (q_1, 001) && \text{wegen } \delta(q_0, 1) = q_1 \\
&\vdash_A (q_0, 01) && \text{wegen } \delta(q_1, 0) = q_0 \\
&\vdash_A (q_0, 1) && \text{wegen } \delta(q_0, 0) = q_0 \\
&\vdash_A (q_1, \varepsilon) && \text{wegen } \delta(q_0, 1) = q_1
\end{aligned}
$$

Also: $(q_0, 1001) \vdash_A^* (q_1, \varepsilon)$

Oder: $(q_0, 1001) \vdash_A^* (q_0, 1)$

Warum schon wieder ^{n},$^+$,*?
Wo ist das Monoid?

Definition 1.4 *Sei M eine Menge (z.B. $M = Q \times \Sigma^*$). Die* Komposition *$R_1 \circ R_2$ zweier Relationen $R_1, R_2 \subseteq M \times M$ ist definiert durch*

$$R_1 \circ R_2 = \{(x,y) \in M \times M \mid \text{es existiert ein } z \in M \text{ mit } (x,z) \in R_1 \text{ und } (z,y) \in R_2\}$$

- Dann ist $(\wp(M \times M), \circ)$ ein Monoid.
- Neutrales Element ist die *Gleichheit* $E_M = \{(x,x) \mid x \in M\}$.
- Also können wir Potenzen R^n von Relationen $R \subseteq M \times M$ wie üblich definieren (insbesondere $\vdash_A^n$).
- Und weil auch Relationen Mengen sind, definieren wir analog zu Sprachen: $R^+ = \bigcup_{n>0} R^n$ und $R^* = \bigcup_{n\geq 0} R^n$ (insbesondere $\vdash_A^+$ und $\vdash_A^*$)

Definition 1.5 *Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA.*

- *Ein Wort $w \in \Sigma^*$ wird von A akzeptiert genau dann wenn es ein $q \in F$ gibt mit $(s, w) \vdash_A^* (q, \varepsilon)$.*
- *Die von A akzeptierte (oder: erkannte) Sprache $L(A)$ ist definiert durch*

$$\begin{aligned} L(A) &= \{w \in \Sigma^* \mid w \text{ wird von } A \text{ akzeptiert}\} \\ &= \{w \in \Sigma^* \mid \text{es existiert ein } q \in F \text{ mit } (s, w) \vdash_A^* (q, \varepsilon)\} \end{aligned}$$

Beispiel

Für den oben definierten Automaten A gilt:

$$\begin{aligned} L(A) &= \{w \in \{0,1\}^* \mid (q_0, w) \vdash^* (q_0, \varepsilon)\} \\ &= \{w \in \{0,1\}^* \mid w \text{ endet auf } 0\} \cup \{\varepsilon\} \end{aligned}$$

Nächstes Ziel:

"Inhaltliches" Verständnis von endlichen Automaten

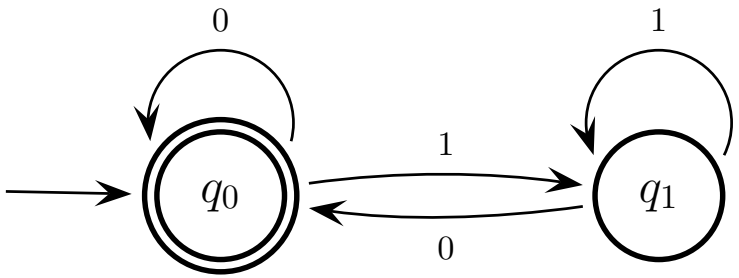
Fragestellungen

- Wie bestimmt man die Sprache $L(A)$ zu einem vorgegebenen Automaten A?
- Wie beweist man, dass ein Automat tatsächlich die gewünschte Sprache akzeptiert?
- Wie findet man einen Automaten, der eine vorgegebene Sprache akzeptiert?
- Wie findet man heraus, ob es überhaupt einen solchen Automaten für eine vorgegebene Sprache gibt?

Dazu: Grundlegende Überlegungen

- Jeder endliche Automat hat nur ein endliches “Gedächtnis”, nämlich seine endlich vielen Zustände.
- Die einzige Information über das bereits gelesene Anfangsstück eines Wortes steckt im aktuellen Zustand.
- Diese begrenzte Information muss ausreichen, um den Rest des Wortes ‘richtig’ abzuarbeiten.

Beispiel (wie oben):



Welche Information über das bereits gelesene Wort w steckt in den Zuständen q_0 bzw. q_1?

- q_0 bedeutet: Entweder $w = \varepsilon$ oder w endet auf 0.
- q_1 bedeutet: w endet auf 1.

Setzt man diese Bedeutung der Zustände voraus, so ist klar, wie der Automat auf das nächste Zeichen reagieren muss:

- Mit 0 muss er stets in q_0 übergehen, da $w0$ auf 0 endet.
- Mit 1 muss er stets in q_1 übergehen, da $w1$ auf 1 endet.

Entwurf eines Automaten für eine vorgegebene Sprache

- Man macht sich klar, welche Information über das bereits gelesene Anfangswort nötig ist, um mit dem Restwort 'richtig' weiterzuarbeiten?
- Reicht eine endliche Anzahl n von unterschiedlichen Informationen aus, so entwirft man einen Automaten mit n Zuständen.
- Die Zustandsübergänge legt man so fest, dass nach dem Lesen eines Zeichens stets wieder die richtige Information vorliegt.
- Reicht eine endliche Anzahl unterschiedlicher Informationen *nicht* aus, so gibt es keinen endlichen Automaten für die Sprache.

Beispiel: Automat A_{int} für die Sprache

$$L_{int} = \{\varepsilon, -\}\{0, \ldots, 9\}^+ = \{\varepsilon, -\}\{0, \ldots, 9\}\{0, \ldots, 9\}^*$$

Sei $\Sigma = \{-, 0, \ldots, 9\}$. Man muss folgende Fälle für das bereits gelesene Wort w unterscheiden:

- $w = \varepsilon$: Noch nichts gelesen, die gesamte Zahl wird noch erwartet.
- $w = -$: Vorzeichen gelesen, es wird noch eine *nichtleere* Ziffernfolge erwartet.
- $w \in L_{int} = \{\varepsilon, -\}\{0, \ldots, 9\}^+$: Es war schon mindestens eine Ziffer da, jetzt darf noch eine *beliebige* Ziffernfolge kommen.
- w enthält '$-$' an der falschen Stelle: Schon alles verdorben!

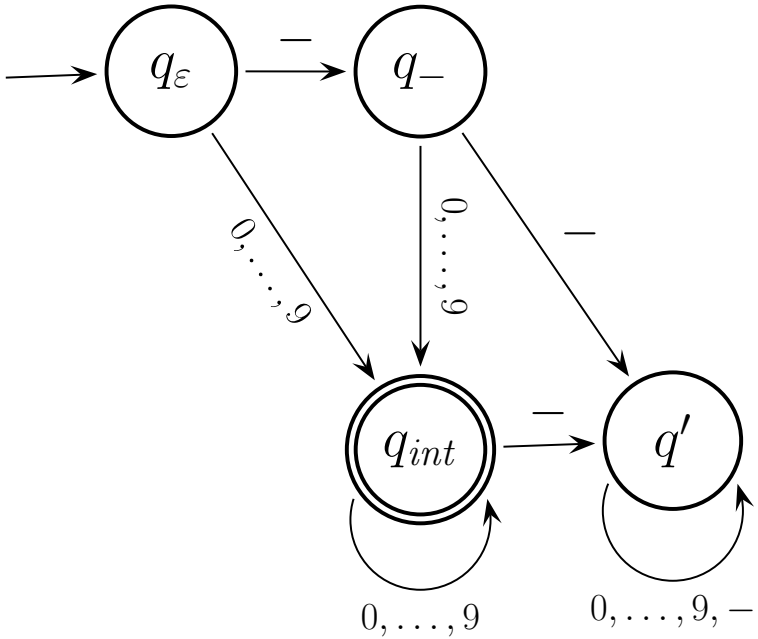
Also 4 Zustände für diese 4 Situationen: $q_\varepsilon, q_-, q_{int}, q'$

Deshalb definiert man $A_{int} = (\Sigma, Q, s, F, \delta)$, wobei

- $Q = \{q_\varepsilon, q_-, q_{int}, q'\}$
- $s = q_\varepsilon$
- $F = \{q_{int}\}$
- $\delta : Q \times \Sigma \to Q$

$$
\begin{aligned}
(q_\varepsilon, -) &\mapsto q_- \\
(q_\varepsilon, a) &\mapsto q_{int} \quad \text{für alle } a \in \{0, \ldots, 9\} \\
(q_-, -) &\mapsto q' \\
(q_-, a) &\mapsto q_{int} \quad \text{für alle } a \in \{0, \ldots, 9\} \\
(q_{int}, -) &\mapsto q' \\
(q_{int}, a) &\mapsto q_{int} \quad \text{für alle } a \in \{0, \ldots, 9\} \\
(q', a) &\mapsto q' \quad \text{für alle } a \in \Sigma \text{ (Sackgasse!)}
\end{aligned}
$$

Graphische Darstellung



Ein Pfeil mit mehreren Zeichen $a_1, \ldots, a_n$ ist Abkürzung für n Pfeile mit den einzelnen Zeichen.

Wie beweist man, dass $L(A_{int}) = L_{int}$? Man zeigt, dass jedes $w \in \Sigma^*$ zum gewünschten Zustand führt, d.h.:

1. Wenn $w = \varepsilon$, dann $(q_\varepsilon, w) \vdash^* (q_\varepsilon, \varepsilon)$.
2. Wenn $w = -$, dann $(q_\varepsilon, w) \vdash^* (q_-, \varepsilon)$.
3. Wenn $w \in L_{int}$, dann $(q_\varepsilon, w) \vdash^* (q_{int}, \varepsilon)$.
4. Wenn $w \notin L_{int} \cup \{\varepsilon, -\}$, dann $(q_\varepsilon, w) \vdash^* (q', \varepsilon)$.

Dies geschieht durch Induktion über $|w|$:
$|w| = 0$, d.h. $w = \varepsilon$: Es gilt $(q_\varepsilon, w) \vdash^0 (q_\varepsilon, \varepsilon)$.
$|w| > 0$, d.h. $w = va$: Fallunterscheidung für v. Wenn $v \in L_{int}$, dann gilt nach Induktionsannahme $(q_\varepsilon, v) \vdash^* (q_{int}, \varepsilon)$. Ist $a \in \{0, \ldots, 9\}$, dann folgt $(q_\varepsilon, w) = (q_\varepsilon, va) \vdash^* (q_{int}, a) \vdash (q_{int}, \varepsilon)$ und das ist der gewünschte Zustand, weil $w = va \in L_{int}$. Ist $a = -$, dann folgt $(q_\varepsilon, w) = (q_\varepsilon, va) \vdash^* (q_{int}, a) \vdash (q', \varepsilon)$ und das ist wieder der richtige Zustand, weil $w \notin L_{int} \cup \{\varepsilon, -\}$. Ähnlich für die übrigen Fälle! □

Beispiel: Ein Automat A für die Sprache $L = \{a^n b^n \mid n \geq 0\}$?

Intuition:

- Um $w \in L$ zu testen, müsste ein endlicher Automat (insbesondere) die Anzahl der a's und b's in w vergleichen.
- Also muss er stets die Anzahl der bisher gelesenen a's kennen.
- Da diese Anzahl beliebig groß werden kann, reichen die endlich vielen Zustände zum Speichern dieser Information nicht aus.
- Also gibt es keinen Automaten A mit $L = L(A)$.

Schlagwort: "Ein endlicher Automat kann nicht zählen."
(Soll heißen: Er kann nicht beliebig hoch zählen).

Exakter Beweis:

Angenommen, es existiert ein DEA $A = (\Sigma, Q, s, F, \delta)$ mit $L(A) = L$.

Wir betrachten alle Wörter der Form a^n mit $n \geq 0$.

Für jedes $n \geq 0$ existiert ein Zustand $q \in Q$ mit $(s, a^n) \vdash_A^* (q, \varepsilon)$ (der Zustand, der durch Lesen von a^n erreicht wird). Da es unendlich viele Wörter a^n gibt, aber nur endlich viele $q \in Q$, muss es (mindestens) zwei Wörter a^i, a^j $(i \neq j)$ geben, mit denen man den *gleichen* Zustand q erreicht, d.h. $(s, a^i) \vdash_A^* (q, \varepsilon)$ und $(s, a^j) \vdash_A^* (q, \varepsilon)$.

Sei $q' \in Q$ mit $(q, b^i) \vdash_A^* (q', \varepsilon)$. Dann gilt $(s, a^i b^i) \vdash_A^* (q, b^i) \vdash_A^* (q', \varepsilon)$ und $(s, a^j b^i) \vdash_A^* (q, b^i) \vdash_A^* (q', \varepsilon)$. Aber $a^i b^i \in L$ und $a^j b^i \notin L$, d.h. sie dürfen nicht beide zum gleichen Zustand q' führen.

Damit ist die Annahme zum Widerspruch geführt, d.h. es existiert kein DEA A mit $L(A) = L$. □

Im Beweis wurde benutzt:

- Das *Schubfachprinzip* (engl. *pigeonhole principle*):
  Wenn man eine Menge von Dingen auf eine gewisse Anzahl von Schubfächern verteilt, und die Anzahl der Dinge ist größer als die Anzahl der Schubfächer, dann landen mindestens zwei Dinge im gleichen Schubfach.
  Hier: Unendlich viele Wörter a^n wurden auf endlich viele Zustände verteilt.
- Folgende Eigenschaft von $\vdash_A^*$:
  Wenn $(q, u) \vdash_A^* (q', \varepsilon)$, dann gilt auch $(q, uv) \vdash_A^* (q', v)$.

Lemma 1.6 *Sei* $A = (\Sigma, Q, s, F, \delta)$ *ein DEA, seien* $u, v \in \Sigma^*$ *und* $q, q' \in Q$. *Dann gilt:*

1. $(q, u) \vdash_A^* (q', \varepsilon) \Leftrightarrow (q, uv) \vdash_A^* (q', v)$
2. $(q, uv) \vdash_A^* (q', \varepsilon) \Leftrightarrow$ *es existiert ein* $q'' \in Q$ *mit* $(q, uv) \vdash_A^* (q'', v) \vdash_A^* (q', \varepsilon)$

Begründung:

1. *Die Übergangsschritte, die* A *bei Abarbeitung eines Wortes* u *macht, werden von einem hinter* u *stehenden Wort* v *nicht beeinflusst. Denn* A *arbeitet ja von links nach rechts, und kann deshalb das Wort* v *während der Abarbeitung von* u *gar nicht sehen.*
2. *Jede Folge von Übergangsschritten für ein Wort* uv *lässt sich aufteilen in die Schritte für* u *und die Schritte für* v. *Das liegt daran, dass der Automat zeichenweise von links nach rechts liest.*

Frage:
Für welche Sprachen L existiert ein DEA A mit $L(A) = L$?

Wir werden beweisen:
Ein DEA A mit $L = L(A)$ existiert genau dann, wenn L eine reguläre Sprache ist. (vgl. Übungsblatt 3, Aufgabe 2)

Die Funktionsschreibweise δ^*

Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA.

Dann existiert für jede Konfiguration $(q, w) \in Q \times \Sigma^*$ *genau ein* Zustand $q' \in Q$ mit $(q, w) \vdash_A^* (q', \varepsilon)$ (weil für jedes einzelne Zeichen von w stets ein eindeutiger Übergangsschritt existiert).

Diesen eindeutigen Zustand q' bezeichnen wir mit $\delta^*(q, w)$.

Die so definierte Funktion $\delta^* : Q \times \Sigma^* \to Q$ lässt sich auch ohne den Umweg über $\vdash_A^*$ definieren, nämlich durch Induktion über die Länge von $|w|$:

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$

oder:

- $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$

Vor- und Nachteile beider Schreibweisen:

In der Funktionsschreibweise haben wir eine eigenständige Bezeichnung für den Ergebniszustand q' (in der die Existenz und Eindeutigkeit implizit enthalten ist). In der Relationsschreibweise müssen wir immer erst die Existenz (und evtl. Eindeutigkeit) von q' erwähnen (und damit einen neuen Namen für einen Zustand einführen).

In der Relationsschreibweise kann man Übergangsschritte besser aneinanderhängen und man hat die Schreibweisen $\vdash^n, \vdash^+, \vdash^*$ mit schönen Gesetzmäßigkeiten (Monoid!). Mit der Funktionsschreibweise geht das nicht so einfach (denn $\delta : Q \times \Sigma \rightarrow Q$ kann man nicht mit sich selbst verknüpfen).

Konventionen

Als typische Bezeichnungen verwenden wir (bisher):

- a, b, c für Zeichen
- u, v, w, x, y, z für Wörter
- Σ für ein Alphabet
- L für eine Sprache
- A, B für Automaten
- p, q, r, s für Zustände, speziell s für den Startzustand
- P, Q, F für Zustandsmengen, speziell Q für die Menge aller Zustände, F für die Menge der Endzustände

Bisher: Determistische endliche Automaten

- Von jedem Zustand $q \in Q$ geht *genau ein* Pfeil mit einem Zeichen $a \in \Sigma$ aus.
- Formal: $\delta : Q \times \Sigma \to Q$ ist eine totale Funktion.

Jetzt: Nichtdeterminismus erlaubt

- Von jedem Zustand $q \in Q$ dürfen *beliebig viele* Pfeile mit einem Zeichen $a \in \Sigma$ ausgehen.
- Formal: An die Stelle der Funktion $\delta : Q \times \Sigma \to Q$ tritt eine *Relation* $\Delta \subseteq Q \times \Sigma \times Q$.

Definition 1.7 *Ein nichtdeterministischer endlicher Automat (kurz: NDEA) ist ein 5-Tupel* $A = (\Sigma, Q, s, F, \Delta)$ *mit:*

- Σ, Q, s, F *wie beim DEA*
- $\Delta \subseteq Q \times \Sigma \times Q$ *ist die sogenannte Übergangsrelation*

Noch zu klären:

- Wie arbeitet ein NDEA?
- Welche Sprache erkennt er?

Definition 1.8

- *Die Relation* $\vdash_A$ *auf der Menge* $Q \times \Sigma^*$ *aller* Konfigurationen *von* A *ist definiert durch*

$$(q, w) \vdash_A (q', w') \quad \Leftrightarrow \quad \text{es existiert ein } a \in \Sigma \text{ mit } w = aw' \text{ und } (q, a, q') \in \Delta$$

- *Die Relationen* $\vdash_A^n$ $(n \geq 0)$, $\vdash_A^+$, $\vdash_A^*$ *und die von* A akzeptierte *oder* erkannte *Sprache* $L(A)$ *sind wie beim DEA definiert.*

Man beachte:

Selbst $\vdash_A$ kann man für DEAs und NDEAs einheitlich definieren, denn: Auch eine Funktion $\delta : Q \times \Sigma \to Q$ lässt sich als Relation $\delta \subseteq Q \times \Sigma \times Q$ auffassen, dann bedeutet $\delta(q, a) = q'$ nichts anderes als $(q, a, q') \in \delta$.

Was ist also jetzt neu?

Zur Erinnerung:

- $w \in \Sigma^*$ wird von A *akzeptiert*, genau dann wenn es ein $q \in F$ gibt mit $(s, w) \vdash_A^* (q, \varepsilon)$.
- $L(A) = \{w \in \Sigma^* \mid w \text{ wird von } A \text{ akzeptiert}\}$

Unterschied zum DEA:

- Ein Zustand q mit $(s, w) \vdash_A^* (q, \varepsilon)$ muss nicht existieren (denn der passende Übergang für ein Zeichen aus w kann fehlen.)
- Es kann mehr als ein solcher Zustand q existieren (denn für ein Zeichen aus w kann es mehr als einen Übergang geben).

Wann wird ein Wort w also akzeptiert?

Wenn *mindestens einer* der Zustände, die man mit w erreicht, ein Endzustand ist.

Intuition:

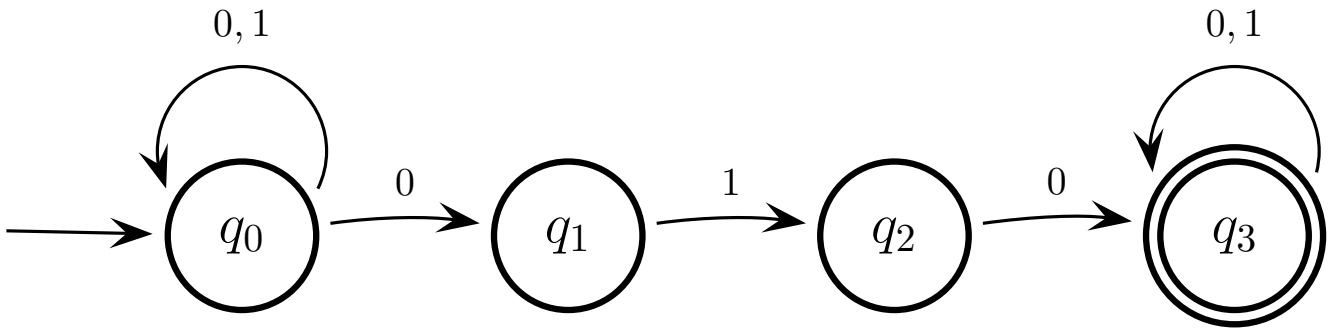
- In einem NDEA können verschiedene Wege für ein Wort w existieren.
- Wir dürfen davon ausgehen, dass der NDEA immer den *richtigen* Weg "errät".
- D.h. wenn *ein* Weg zu einem Endzustand führt, dann brauchen uns die anderen Wege nicht zu interessieren.

Beispiel:

$\Sigma = \{0, 1\}$

$L = \{w \in \{0,1\}^* \mid 010 \text{ ist Teilwort von } w\}$.

Ein NDEA A mit $L(A) = L$ ist:



Formal:

$A = (\Sigma, Q, s, F, \Delta)$ mit $\Sigma = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $s = q_0$ und

$\Delta = \{\ (q_0, 0, q_0), (q_0, 1, q_0), (q_0, 0, q_1), (q_1, 1, q_2), (q_2, 0, q_3),$
$(q_3, 0, q_3), (q_3, 1, q_3)\}$

Wieso gilt $L = L(A)$?

Intuition:

Weil der NDEA A die Stelle, an der das Teilwort 010 beginnt, *erraten* und dann mit 010 in den Endzustand q_3 übergehen kann.

Exakter Beweis:

'$\subseteq$': Sei $w \in L$.

Dann existieren $u, v \in \{0,1\}^*$ mit $w = u010v$.

Also gilt $(s, w) = (q_0, u010v) \vdash_A^* (q_0, 010v) \vdash_A^* (q_3, v) \vdash_A^* (q_3, \varepsilon)$.

Damit ist $w \in L(A)$ bewiesen, weil $q_3 \in F$.

'$\supseteq$': Sei $w \in L(A)$, d.h. $(q_0, w) \vdash_A^* (q_3, \varepsilon)$, weil F nur q_3 enthält.

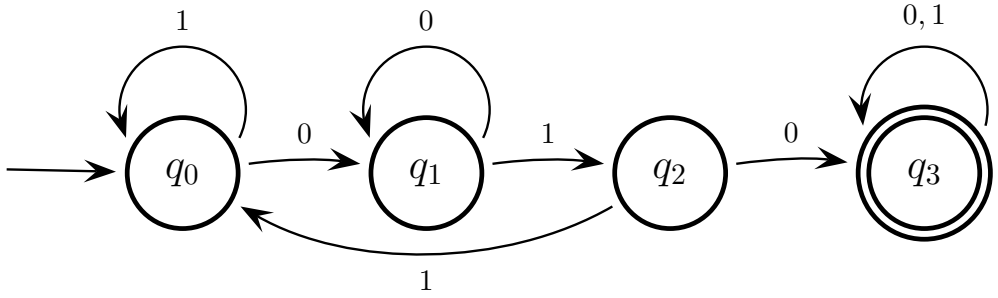
Dann muss w das Wort 010 als Teilwort enthalten, weil man nur mit diesem Wort von q_0 nach q_3 gelangen kann.

Also ist $w \in L$. □

Wieso ist der Beweis so einfach?

Weil dieser NDEA auf einer sehr einfachen Idee beruht.

Zum Vergleich: Ein DEA A' für die gleiche Sprache



Beweisidee:

Sei w das bisher gelesene Wort.

q_0 bedeutet: $w = \varepsilon$, $w = 1$ oder w endet auf 11, aber enthält nicht 010.

q_1 bedeutet: w endet auf 0, aber enthält nicht 010.

q_2 bedeutet: w endet auf 01, aber enthält nicht 010.

q_3 bedeutet: w enthält 010, d.h. $w \in L$.

Fazit:

- Oft findet man leichter einen NDEA als einen DEA für eine vorgegebene Sprache (weil man mehr Freiheiten hat)
- und auch der Korrektheitsbeweis ist oft einfacher (wenn der NDEA auf einer einfacheren Idee beruht).

Wie kann man einen NDEA implementieren?

- Als nichtdeterministisches Programm mit Zufallsgenerator?

  Nein, denn man kann nicht erwarten, dass das Programm den richtigen Weg errät.

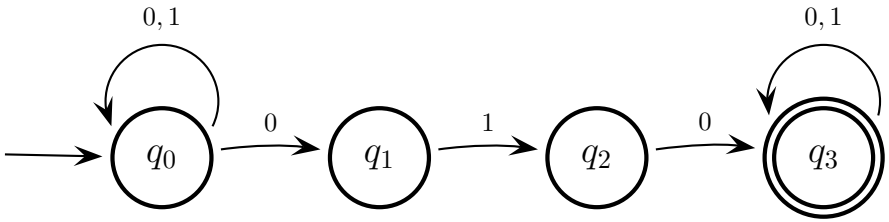
- Durch einen backtracking-Algorithmus, der alle möglichen Wege nacheinander durchspielt?

  Ja, aber man muss sich überlegen, dass es nur endlich viele mögliche Wege gibt, und wie man sie systematisch ausprobiert.

- Durch einen "parallelen" Algorithmus, der alle möglichen Wege gleichzeitig durchspielt?

  Das ist die einfachste Möglichkeit: Man führt einfach Buch über die *Menge* der Zustände, die vom Startzustand aus mit dem bereits gelesenen Wort erreichbar sind. Diese Menge kann auch *leer* sein!

Beispiel: $w = 0011010$ **auf dem NDEA** A



Menge der möglichen Zustände nach Lesen von

- $\varepsilon : \{q_0\}$
- $0 : \{q_0, q_1\}$
- $00 : \{q_0, q_1\}$
- $001 : \{q_0, q_2\}$
- $0011 : \{q_0\}$
- $00110 : \{q_0, q_1\}$
- $001101 : \{q_0, q_2\}$
- $0011010 : \{q_0, q_1, q_3\}$, also $w \in L(A)$, da q_3 mit w erreichbar

Fazit:

- Wir haben einen *deterministischen* Algorithmus für einen *N*DEA A,
  - bei dem man das Eingabewort w zeichenweise von links nach rechts liest,
  - bei dem man sich aber Zustands*mengen* anstelle von einzelnen Zuständen merken muss.
- Dieser Algorithmus lässt sich sogar *auf einem DEA* A' durchführen.
- Man muss A' nur mit einem großen Gedächtnis ausstatten, damit er sich *Teilmengen* der Zustandsmenge Q von A merken kann.
- Als Zustände von A' wählen wir deshalb alle Teilmengen von Q,

  und legen dann die Übergänge zwischen diesen Zustandsmengen passend fest (wie im Beispiel).
- So erhalten wir einen zu A *äquivalenten* DEA, d.h. einen DEA A' mit $L(A) = L(A')$.

Definition 1.9 *Sei* $A = (\Sigma, Q, s, F, \Delta)$ *ein NDEA. Der* Potenzautomat *zu* A *ist definiert als der DEA* $A' = (\Sigma, Q', s', F', \delta)$ *mit*

- $Q' = \wp(Q)$, *die Potenzmenge von* Q
- $s' = \{s\}$
- $F' = \{P \in \wp(Q) \mid P \cap F \neq \emptyset\}$
  $= \{P \subseteq Q \mid P$ *enthält mindestens einen Zustand aus* $F\}$
- $\delta : \wp(Q) \times \Sigma \to \wp(Q)$

  $\delta(P, a) = \{q \in Q \mid$ *es existiert ein* $p \in P$ *mit* $(p, a, q) \in \Delta\}$

  (d.h. $\delta(P, a)$ *enthält genau die Zustände, die von einem Zustand* $p \in P$ *durch einen mit* a *markierten Pfeil erreichbar sind)*

Satz 1.10 *Sei A ein NDEA und A' der Potenzautomat zu A. Dann gilt $L(A) = L(A')$.*

Beweis

Sei $A = (\Sigma, Q, s, F, \Delta)$ und sei $A' = (\Sigma, Q', s', F', \delta)$ der Potenzautomat zu A.

- Wir zeigen zunächst:

$$\delta^*(\{s\}, w) = \{q \in Q \mid (s, w) \vdash_A^* (q, \varepsilon)\}$$

In Worten: $\delta^*(\{s\}, w)$ ist die Menge aller Zustände, die der NDEA A vom Zustand s aus durch Abarbeitung von w erreichen kann. Das bedeutet, dass der Potenzautomat A' die gewünschte Buchführung für den NDEA A richtig durchführt.

Also zu zeigen: $\delta^*(\{s\}, w) = \{q \in Q \mid (s, w) \vdash_A^* (q, \varepsilon)\}$

Beweis durch Induktion über $|w|$:

$|w| = 0$, d.h. $w = \varepsilon$:

$\delta^*(\{s\}, \varepsilon) = \{s\}$ per Definition von δ^*

$\{q \in Q \mid (s, \varepsilon) \vdash_A^* (q, \varepsilon)\} = \{s\}$ per Definition von $\vdash_A^*$

$|w| > 0$, d.h. $w = va$:

$$
\begin{aligned}
\delta^*(\{s\}, va) &= \delta(\delta^*(\{s\}, v), a) \\
&\quad \text{per Definition der Schreibweise } \delta^* \\
&= \delta(\{q \in Q \mid (s, v) \vdash_A^* (q, \varepsilon)\}, a) \\
&\quad \text{nach Induktionsannahme für } v \\
&= \{q' \in Q \mid \text{ es existiert ein } q \in Q \text{ mit } (s, v) \vdash_A^* (q, \varepsilon) \\
&\qquad\qquad \text{und } (q, a) \vdash_A (q', \varepsilon)\} \\
&\quad \text{per Definition von } \delta \text{ im Potenzautomaten } A' \\
&= \{q' \in Q \mid (s, va) \vdash_A^* (q', \varepsilon)\}
\end{aligned}
$$

Die Behauptung ist bewiesen:

$\delta^*(\{s\}, w) = \{q \in Q \mid (s, w) \vdash_A^* (q, \varepsilon)\}$

und es folgt:

$$
\begin{aligned}
w \in L(A) \quad &\Leftrightarrow \quad \text{es existiert ein } q \in F \text{ mit } (s, w) \vdash_A^* (q, \varepsilon) \\
&\qquad\quad \text{per Definition der akzeptierten Sprache } L(A) \\
&\Leftrightarrow \quad \text{es existiert ein } q \in F \text{ mit } q \in \delta^*(\{s\}, w) \\
&\qquad\quad \text{nach der bewiesenen Behauptung} \\
&\Leftrightarrow \quad \delta^*(\{s\}, w) \cap F \neq \emptyset \\
&\Leftrightarrow \quad \delta^*(\{s\}, w) \in F' \\
&\qquad\quad \text{per Definition von } F' \text{ im Potenzautomaten } A' \\
&\Leftrightarrow \quad w \in L(A') \\
&\qquad\quad \text{weil } \{\text{s}\} \text{ der Startzustand von } A' \text{ ist}
\end{aligned}
$$

Also ist $L(A) = L(A')$, d.h. A und A' sind äquivalent. $\square$

Man beachte:

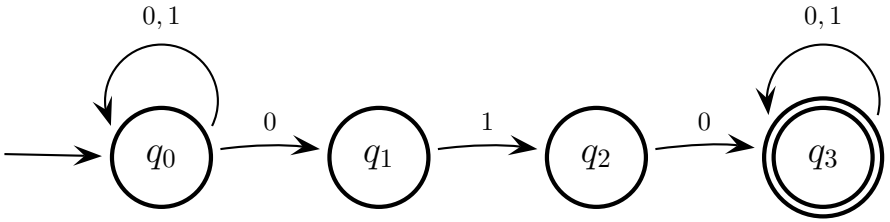
- Im Beweis wurde benutzt, dass Lemma 1.6 auch für NDEAs gilt.
- Die Definition des Potenzautomaten ist *konstruktiv*, d.h. man hat einen *Algorithmus*, um den Potenzautomaten A' aus dem NDEA A zu erhalten.
- Der Potenzautomat ist natürlich sehr groß, denn $|\wp(Q)| = 2^{|Q|}$. Oft sind aber viele Zustände überflüssig, so dass man sie nachträglich entfernen oder sogar von Anfang an weglassen kann.

Definition 1.11 *Sei* $A = (\Sigma, Q, s, F, \delta)$ *ein DEA. Ein Zustand* $q \in Q$ *heißt* erreichbar, *wenn es ein Wort* w *gibt mit* $(s, w) \vdash_A^* (q, \varepsilon)$, *ansonsten heißt er* unerreichbar.

- Unerreichbare Zustände kann man offensichtlich aus einem DEA entfernen, ohne dass sich die akzeptierte Sprache verändert (und ohne dass man die Definition eines DEA verletzt, denn s ist stets erreichbar, und δ lässt sich auf die Menge der erreichbaren Zustände einschränken.)
- Einige "Entwurfsmethoden" für Automaten, wie z.B. die Konstruktion des Potenzautomaten, kann man so abändern, dass der entworfene Automat von vornherein nur erreichbare Zustände enthält.

Beispiel

Sei A wieder der NDEA



Der Potenzautomat von A hat 16 Zustände. Wir bestimmen die erreichbaren unter ihnen.

Der Startzustand $\{q_0\}$ ist erreichbar.

$\delta(\{q_0\}, 0) = \{q_0, q_1\}$, also ist $\{q_0, q_1\}$ erreichbar.

$\delta(\{q_0\}, 1) = \{q_0\}$

$\delta(\{q_0, q_1\}, 0) = \{q_0, q_1\}$

$\delta(\{q_0, q_1\}, 1) = \{q_0, q_2\}$, also ist $\{q_0, q_2\}$ erreichbar.

$\delta(\{q_0, q_2\}, 0) = \{q_0, q_1, q_3\}$, also ist $\{q_0, q_1, q_3\}$ erreichbar.

$\delta(\{q_0, q_2\}, 1) = \{q_0\}$

$\delta(\{q_0, q_1, q_3\}, 0) = \{q_0, q_1, q_3\}$

$\delta(\{q_0, q_1, q_3\}, 1) = \{q_0, q_2, q_3\}$, also ist $\{q_0, q_2, q_3\}$ erreichbar.

$\delta(\{q_0, q_2, q_3\}, 0) = \{q_0, q_1, q_3\}$

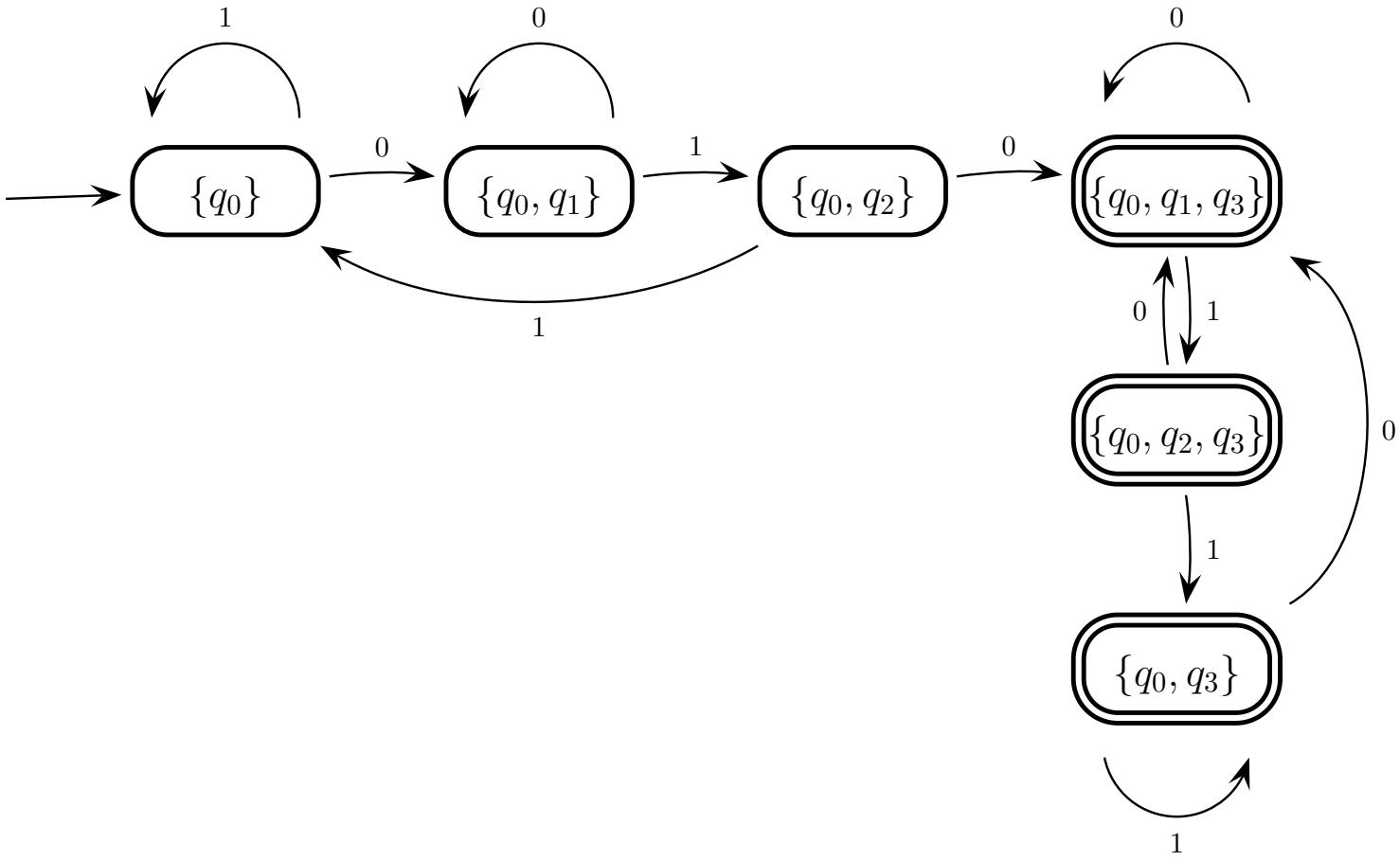
$\delta(\{q_0, q_2, q_3\}, 1) = \{q_0, q_3\}$, also ist $\{q_0, q_3\}$ erreichbar.

$\delta(\{q_0, q_3\}, 0) = \{q_0, q_1, q_3\}$

$\delta(\{q_0, q_3\}, 1) = \{q_0, q_3\}$

Wenn wir uns also auf die erreichbaren Zustände beschränken, so erhalten wir einen DEA, der nur 6 statt 16 Zustände hat. Drei davon sind Endzustände, nämlich diejenigen, die den Endzustand q_3 von A enthalten.

Graphisch: Der ereichbare Teil des Potenzautomaten



Wie bestimmt man die Menge der erreichbaren Zustände?

- Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA. Für jedes $n \geq 0$ sei

  $R_n = \{q \in Q \mid$ es ex. ein $w \in \Sigma^*$ mit $|w| \leq n$ und $(s, w) \vdash_A^* (q, \varepsilon)\}$

- Die Mengen R_n lassen sich durch Induktion über n definieren:
  - $R_0 = \{s\}$
  - $R_{n+1} = R_n \cup \{\delta(q, a) \mid q \in R_n, a \in \Sigma\}$
- Es gilt $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots$ und die Menge R *aller* erreichbaren Zustände erhält man durch

  $R = \bigcup_{n \geq 0} R_n$

- Da alle R_n in der endlichen Menge Q enthalten sind, muss ein n existieren mit $R_n = R_{n+1}$.
- Daraus folgt $R_n = R_m$ für alle $m \geq n$, also $R = R_n$.

Algorithmus

- Berechne die Mengen $R_0, R_1, R_2, \ldots$ mit der induktiven Definition
- Sobald $R_n = R_{n+1}$ gilt, ist R_n die Menge der erreichbaren Zustände.

Man beachte:

- Nach den vorangegangen Überlegungen terminiert der Algorithmus und liefert das korrekte Ergebnis.
- Der gleiche Algorithmus funktioniert auch für NDEAs und auch für einen beliebigen Zustand p anstelle des Startzustands s.
- Er funktioniert sogar, wenn man noch nicht alle Übergänge kennt, denn im Induktionsschritt braucht man nur die Übergänge, die von den bereits gefundenen Zuständen ausgehen.
- Eigentlich handelt es sich um einen Graphalgorithmus (vgl. Vorlesung Algorithmen): Da die Markierung der Pfeile im Algorithmus keine Rolle spielt, kann der Automat als gerichteter Graph gesehen werden.

Gibt es noch weitere überflüssige Zustände?

- Ein Zustand heißt *tot*, wenn man von ihm aus keinen Endzustand mehr erreichen kann.
- Beispiele:

  der Zustand q' im Automaten A_{int},

  der Zustand $\emptyset$ in einem Potenzautomaten.
- Auch tote Zustände kann man aus einem Automaten entfernen, ohne dass sich die erkannte Sprache verändert.
- **Aber:** Dabei kann aus der totalen Funktion δ eine partielle Funktion werden, d.h. der entstehende Automat muss kein DEA mehr sein.

Anmerkung zur Literatur:

- Manche Autoren lassen partielle Funktionen in DEAs zu.

  Das entspricht besser dem intuitiven Begriff "deterministisch"

  und man spart sich tote Zustände.

- Aber partielle Funktionen bereiten technische Probleme bei der Schreibweise: Man kann sie nicht auf jedes Element anwenden.

  Deshalb lassen wir sie nicht in DEAs, sondern nur (als spezielle Relationen) in NDEAs zu.

  Das ist kein großer Nachteil, denn man kommt stets mit *einem* toten Zustand aus.

Sprachklassen

Bei vorgegebenem Alphabet Σ sei

- $\mathcal{L}_{reg} = \{L \subseteq \Sigma^* \mid L \text{ regulär.}\}$
- $\mathcal{L}_{DEA} = \{L \subseteq \Sigma^* \mid \text{es existiert ein DEA } A \text{ mit } L = L(A)\}$
- $\mathcal{L}_{NDEA} = \{L \subseteq \Sigma^* \mid \text{es existiert ein NDEA } A \text{ mit } L = L(A)\}$

Solche Mengen von Sprachen bezeichnet man als *Sprachklassen*,

Konvention: $\mathcal{L}$ für Sprachklassen.

Wir wollen zeigen:

$\mathcal{L}_{DEA} = \mathcal{L}_{NDEA} = \mathcal{L}_{reg}$

Die erste Gleichheit haben wir schon.

Satz 1.12 $\mathcal{L}_{DEA} = \mathcal{L}_{NDEA}$

Beweis:

'$\subseteq$':

Klar, da jeder DEA ein NDEA ist.

Genauer:

Wenn man einen DEA A als NDEA auffasst, d.h. wenn man die Übergangsfunktion δ als Relation auffasst, dann stimmen alle Definitionen ($\vdash_A, \vdash_A^*, \ldots$) für den DEA und den NDEA überein, also ist insbesondere die erkannte Sprache in beiden Fällen die gleiche.

'$\supseteq$':

Sei $L = L(A)$ für einen NDEA A. Dann gilt $L = L(A')$ für den Potenzautomaten A' von A, und A' ist ein DEA. $\square$

Um zu zeigen, dass auch $\mathcal{L}_{reg}$ die gleiche Sprachklasse ist, betrachten wir zunächst eine weitere Verallgemeinerung unseres Automatenbegriffs. Wir lassen Pfeile zu, die mit dem leeren Wort ε markiert sind.

Definition 1.13 *Ein ε-NDEA ist ein 5-Tupel $A = (\Sigma, Q, s, F, \Delta)$ mit:*

- Σ, Q, s, F *wie beim DEA*
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$

Intuition:

- Ein ε-NDEA kann *spontane* Zustandsübergänge durchführen, ohne ein Zeichen zu lesen.
- Durch diese weitere Freiheit kann es einfacher sein, einen endlichen Automaten für eine gegebene Sprache zu finden.

  z.B. indem man dem Automaten ermöglicht, ein 'optionales' Zeichen oder Teilwort mit ε zu überspringen (Sprachen L_{int}, L_{float}).

Arbeitsweise eines ε-NDEA:

- Nur $\vdash_A$ muss neu definiert werden

$$(q, w) \vdash_A (q', w') \;\Leftrightarrow\; \text{es existiert ein } a \in \Sigma \cup \{\varepsilon\} \text{ mit } w = aw' \text{ und } (q, a, q') \in \Delta$$

- Alles andere wie beim NDEA.
- Die neue Sprachklasse bezeichnen wir mit $\mathcal{L}_{\varepsilon\text{-}NDEA}$.

Übliche Frage:

Kann ein ε-NDEA prinzipiell mehr leisten als ein NDEA?

D.h. gibt es eine Sprache in $\mathcal{L}_{\varepsilon\text{-}NDEA} \setminus \mathcal{L}_{NDEA}$?

Satz 1.14 *Zu jedem ε-NDEA lässt sich ein NDEA konstruieren, der die gleiche Sprache akzeptiert.*

Beweis:

Sei $A = (\Sigma, Q, s, F, \Delta)$ ein ε-NDEA.

Wie kann man die ε-Übergänge aus A entfernen, ohne die erkannte Sprache zu verändern?

Dazu definiert man zunächst den *ε-Abschluss* $E(p)$ eines Zustands $p \in Q$ durch:

$$E(p) = \{q \in Q \mid (p, \varepsilon) \vdash_A^* (q, \varepsilon)\}$$

$E(p)$ enthält also genau die Zustände, die man von p aus mit einer Folge von ε-Schritten erreichen kann.

Insbesondere ist stets $p \in E(p)$, weil auch eine Folge von 0 Schritten zugelassen ist.

Sei nun $A' = (\Sigma, Q, s, F', \Delta')$ mit

- $\Delta' = \{(p, a, q) \in Q \times \Sigma \times Q \mid \text{es ex. ein } p' \in E(p) \text{ mit } (p', a, q) \in \Delta\}$

  d.h. in A' nimmt man genau dann einen a-Übergang von p nach q auf, wenn es in A eine Folge von ε-Übergängen von p zu einem Zustand p', und von p' einen a-Übergang nach q gibt.

- $F' = \{p \in Q \mid E(p) \cap F \neq \emptyset\}$

  d.h. die Endzustände von A' sind genau die Zustände von A, die mit einer Folge von ε-Schritten in einen Endzustand übergehen können.

- A' ist ein NDEA, denn per Definition ist $\Delta' \subseteq Q \times \Sigma \times Q$, d.h. A' enthält *keine* ε-Übergänge mehr.

- Andererseits gilt $\Delta \cap (Q \times \Sigma \times Q) \subseteq \Delta'$, denn wegen $p \in E(p)$ kann man in der Definition von Δ' auch $p' = p$ wählen.

  Das bedeutet, dass alle Pfeile von A, die *nicht* mit ε markiert sind, in A' erhalten bleiben.

Behauptung: $L(A) = L(A')$

Beweis:

'$\subseteq$':

- Sei $w = a_1 \ldots a_n \in L(A)$, d.h. $(s, w) \vdash_A^* (f, \varepsilon)$ für ein $f \in F$.
- Diese Folge lässt sich so aufteilen:

$$\begin{aligned}
(s, a_1 \ldots a_n) &\vdash_A^* (q_0, a_1 \ldots a_n) \vdash_A (q_1, a_2 \ldots a_n) \\
&\vdots \\
&\vdash_A^* (q_{n-1}, a_n) \vdash_A (q_n, \varepsilon) \\
&\vdash_A^* (f, \varepsilon)
\end{aligned}$$

wobei die $\vdash_A^*$ nur aus ε-Schritten bestehen.

- Also gilt per Definition von Δ':

  $(s, a_1 \ldots a_n) \vdash_{A'} (q_1, a_2 \ldots a_n) \vdash_{A'} \ldots \vdash_{A'} (q_n, \varepsilon)$

  und per Definition von F' ist $q_n \in F'$, also $w \in L(A')$

'$\supseteq$':

- Sei $w \in L(A')$, d.h. $(s, w) \vdash_{A'}^* (f', \varepsilon)$ für ein $f' \in F'$
- Da jeder Übergangsschritt in A' einer Folge von Übergangsschritten in A entspricht, gilt dann auch $(s, w) \vdash_A^* (f', \varepsilon)$

  und per Definition von F' gilt $(f', \varepsilon) \vdash_A^* (f, \varepsilon)$ für ein $f \in F$.
- Also gilt $(s, w) \vdash_A^* (f, \varepsilon)$ und damit $w \in L(A)$.

Noch zu zeigen:

Der NDEA A' lässt sich tatsächlich aus A *konstruieren*.

Dazu muss man die Menge $E(p)$ für jeden Zustand $p \in Q$ berechnen. Das geht analog zur Menge R der erreichbaren Zustände.

Für jedes $n \geq 0$ sei

$$E_n(p) = \{q \in Q \mid \text{es exisitiert ein } m \leq n \text{ mit } (p, \varepsilon) \vdash_A^m (q, \varepsilon)\}$$

(die Menge aller Zustände, die von p aus mit höchstens n ε-Schritten erreichbar sind).

Auch diese Mengen lassen sich durch Induktion über n berechnen:

- $E_0(p) = \{p\}$
- $E_{n+1}(p) = E_n(p) \cup \{q \in Q \mid \text{es ex. ein } p' \in E_n(p) \text{ mit } (p', \varepsilon, q) \in \Delta\}$

und bilden eine aufsteigende Folge $E_0(p) \subseteq E_1(p) \subseteq \ldots$

Mit der gleichen Argumentation wie oben folgt:

Es existiert ein n mit $E_n(p) = E_{n+1}(p)$, und für diese Zahl n ist dann $E(p) = E_n(p)$.

Schließlich erhält man mit Hilfe der Mengen $E(p)$ die Übergangsrelation Δ' und die Endzustandsmenge F', und damit ist die Konstruktion von A' beendet. □

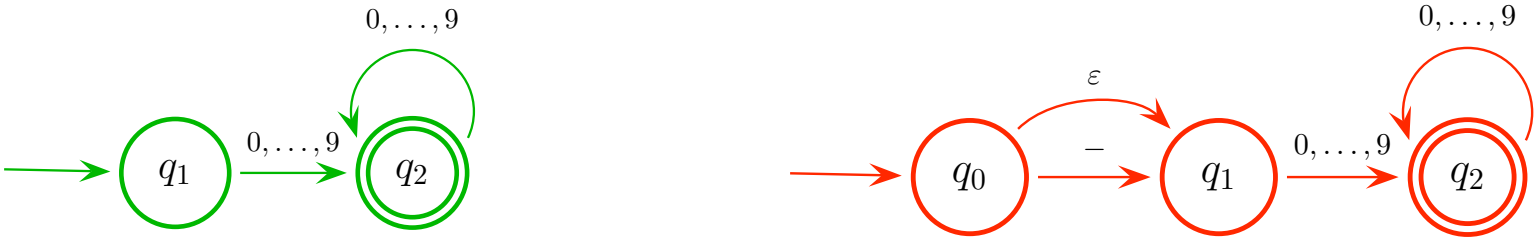
Als unmittelbare Folgerung von Satz 1.14 erhalten wir

Satz 1.15

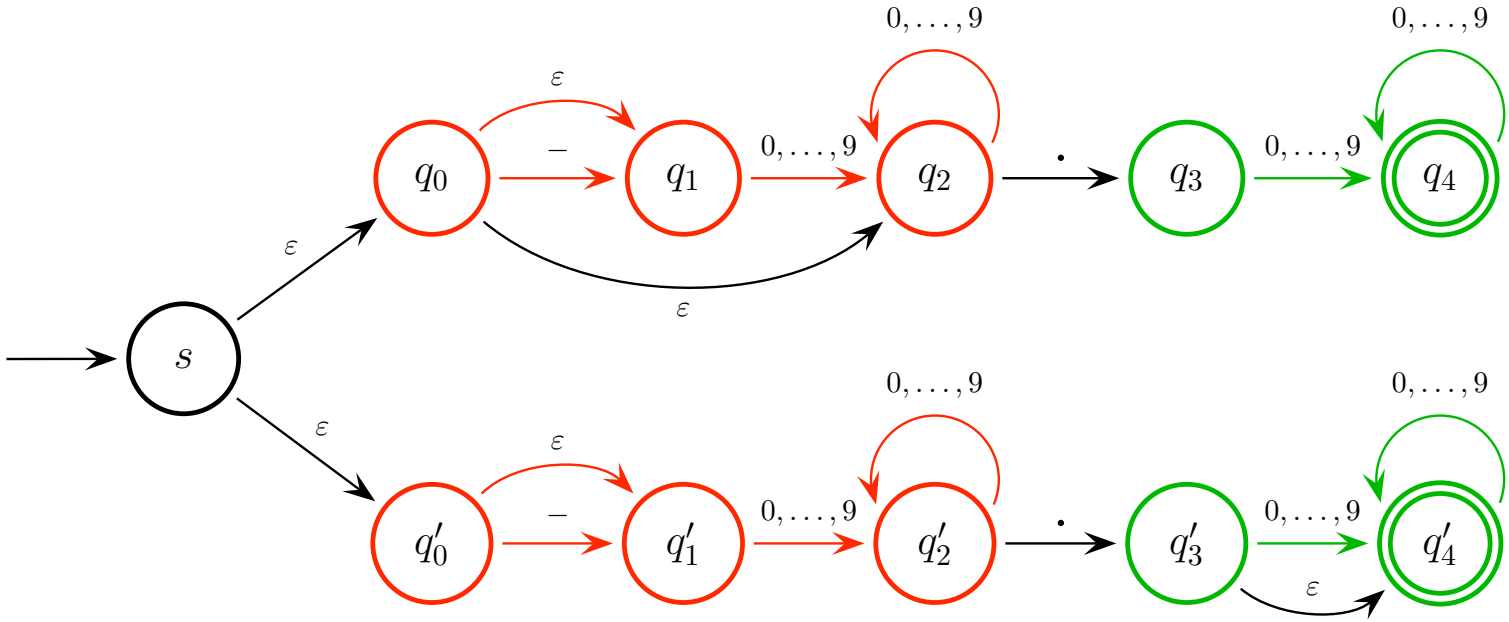
$$\mathcal{L}_{\varepsilon\text{-}NDEA} = \mathcal{L}_{NDEA}$$

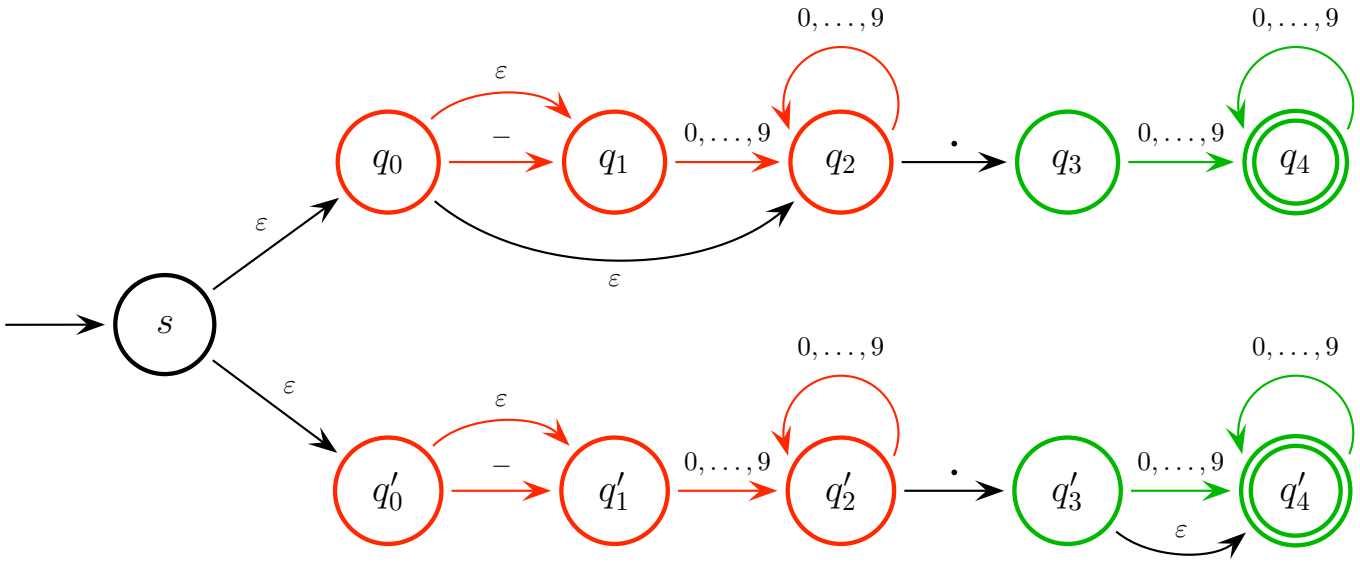
Beispiel:

ε-NDEAs für $L_{nat} = \{0,\ldots,9\}^+$ und $L_{int} = \{-,\varepsilon\}L_{nat}$



und für $L_{fix} = (L_{int} \cup \{\varepsilon\})\{.\}L_{nat} \cup L_{int}\{.\}(L_{nat} \cup \{\varepsilon\})$





Umwandlung in einen NDEA: Die *neuen Übergänge* sind

$(s, -, q_1)$ da $q_0 \in E(s)$

(s, z, q_2) für jede Ziffer z, da $q_1 \in E(s)$

$(s, ., q_3)$ da $q_2 \in E(s)$

(q_0, z, q_2) für jede Ziffer z, da $q_1 \in E(q_0)$

$(q_0, ., q_3)$ da $q_2 \in E(q_0)$

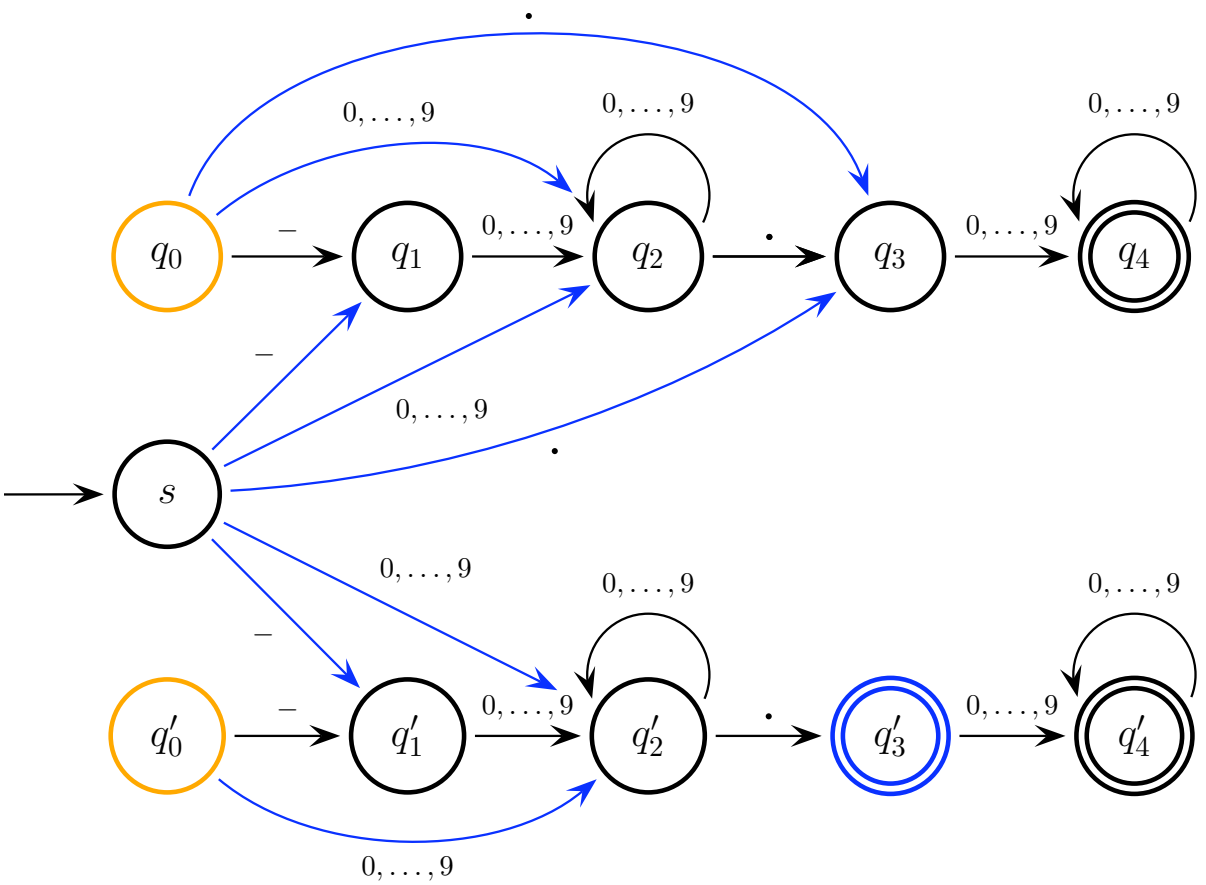
(s, z, q'_2) für jede Ziffer z, da $q'_1 \in E(s)$

$(s, -, q'_1)$ da $q'_0 \in E(s)$

(q'_0, z, q'_2) für jede Ziffer z, da $q'_1 \in E(q'_0)$

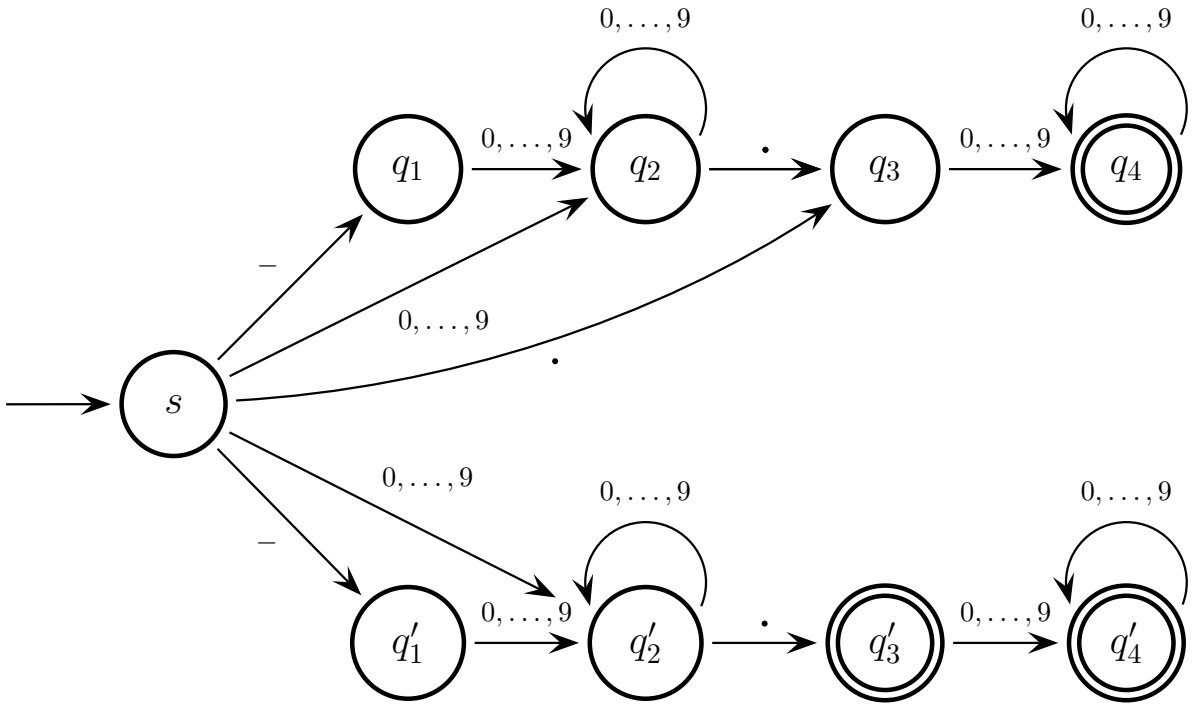
und q'_3 ist *neuer Endzustand*, weil $q'_4 \in E(q'_3)$.

Ergebnis: ein NDEA für L_{int}



Unerreichbare Zustände: q_0 und q'_0

Nach Entfernen der unerreichbaren Zustände:



Am Beispiel war schon zu sehen, wie man einen ε-NDEA konstruieren kann, der eine gegebene reguläre Sprache erkennt.

Dieses Verfahren wird jetzt präzisiert. So erhalten wir den Beweis, dass $\mathcal{L}_{reg} \subseteq \mathcal{L}_{\varepsilon\text{-}NDEA}$.

Da wir schon wissen, dass

$$\mathcal{L}_{DEA} = \mathcal{L}_{NDEA} = \mathcal{L}_{\varepsilon\text{-}NDEA}$$

gilt, bezeichnen wir diese Sprachklasse jetzt mit

$$\mathcal{L}_{EA}$$

und benutzen den Begriff *endlicher Automat* (kurz: *EA*) als Sammelbegriff (d.h. als Synonym für ε-NDEA).

Satz 1.16 *Für die Sprachklasse $\mathcal{L}_{EA}$ (über dem Alphabet Σ) gilt:*

1. *$\mathcal{L}_{EA}$ enthält die Sprachen $\emptyset$, $\{\varepsilon\}$ und $\{a\}$ für alle $a \in \Sigma$.*
2. *$\mathcal{L}_{EA}$ ist abgeschlossen unter den Operationen $\cup$, $\circ$, $^+$ und * d.h. wenn die Sprachen L_1, L_2 von endlichen Automaten erkannt werden, dann werden auch $L_1 \cup L_2$, $L_1 \circ L_2$, L_1^+ und L_1^* von endlichen Automaten erkannt.*
3. *Darüber hinaus gibt es für jede der Operationen in 2. einen Algorithmus, mit dem man den neuen Automaten konstruieren kann, also z.B. einen Algorithmus, der zu zwei endlichen Automaten A_1 und A_2 einen endlichen Automaten A mit $L(A) = L(A_1) \cup L(A_2)$ konstruiert.*

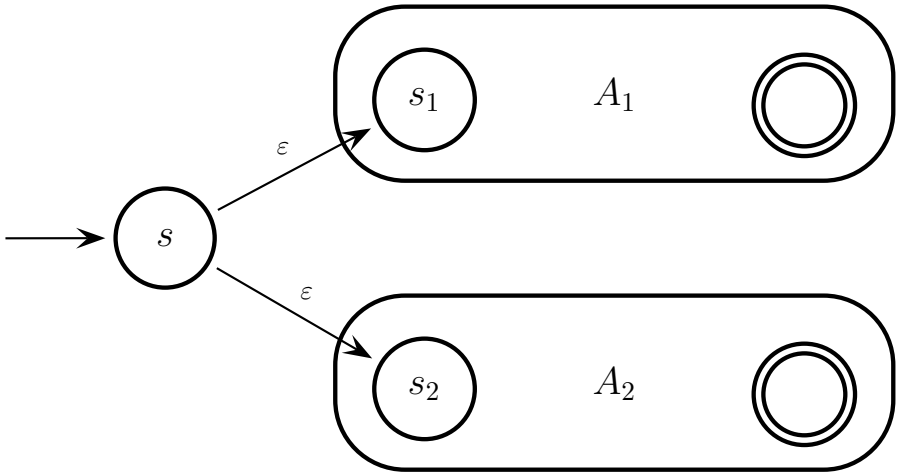
Beweis:

1. NDEAs, die $\emptyset$, $\{\varepsilon\}$ und $\{a\}$ erkennen:

2. Abgeschlossenheit unter $\cup$:

   Seien A_1, A_2 endliche Automaten.

   Ein ε-NDEA A, der $L(A_1) \cup L(A_2)$ erkennt:



Formal:

Seien $A_1 = (\Sigma, Q_1, s_1, F_1, \Delta_1)$ und $A_2 = (\Sigma, Q_2, s_2, F_2, \Delta_2)$ endliche Automaten mit $Q_1 \cap Q_2 = \emptyset$ (kann man durch Umbenennung erreichen!)

Wir definieren $A = (\Sigma, Q, s, F, \Delta)$ mit:

- s ist ein neuer Zustand, d.h. $s \notin Q_1 \cup Q_2$
- $Q = Q_1 \cup Q_2 \cup \{s\}$
- $F = F_1 \cup F_2$
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, \varepsilon, s_1), (s, \varepsilon, s_2)\}$

Zu zeigen: $L(A) = L(A_1) \cup L(A_2)$

$\subseteq$: Sei $w \in L(A)$, d.h. $(s, w) \vdash_A^* (f, \varepsilon)$ für ein $f \in F = F_1 \cup F_2$.

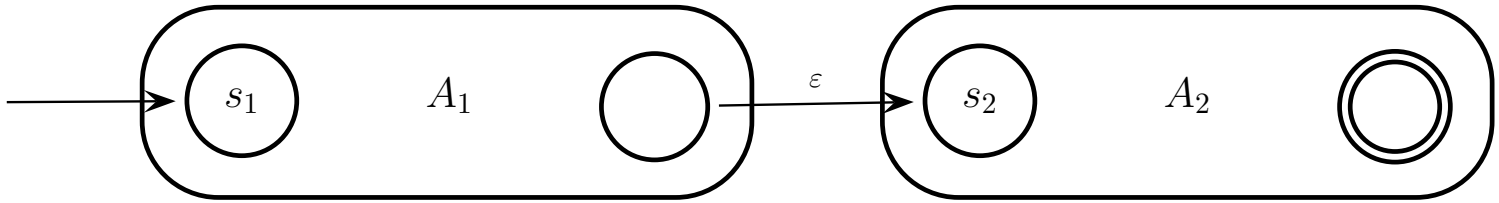
Ist $f \in F_1$, so kann der erste Schritt nur $(s, w) \vdash_A (s_1, w)$ sein, und die restlichen Schritte können nur in A_1 liegen, also gilt $(s_1, w) \vdash_{A_1}^* (f, \varepsilon)$, d.h. $w \in L(A_1)$. Analog für $f \in F_2$.

$\supseteq$: Sei $w \in L(A_1)$, d.h. $(s_1, w) \vdash_{A_1}^* (f, \varepsilon)$ für ein $f \in F_1 \subseteq F$.

Dann gilt $(s, w) \vdash_A (s_1, w) \vdash_{A_1}^* (f, \varepsilon)$ und wegen $\Delta_1 \subseteq \Delta$ folgt daraus $(s, w) \vdash_A (s_1, w) \vdash_A^* (f, \varepsilon)$, also $w \in L(A)$. Analog für $w \in L(A_2)$.

Abgeschlossenheit unter $\circ$:

Ein ε-NDEA A, der $L(A_1) \circ L(A_2)$ erkennt:



Formal:

Seien A_1, A_2 wie oben (mit disjunkten Zustandsmengen).

Wir definieren $A = (\Sigma, Q, s, F, \Delta)$ mit:

- $Q = Q_1 \cup Q_2$
- $s = s_1$
- $F = F_2$
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{(f, \varepsilon, s_2) \mid f \in F_1\}$

Zu zeigen: $L(A) = L(A_1) \circ L(A_2)$

$\subseteq$: Sei $w \in L(A)$, d.h. $(s_1, w) \vdash_A^* (f_2, \varepsilon)$ für ein $f_2 \in F_2 = F$.

Da man nur durch die neuen ε-Übergänge von A_1 nach A_2 gelangen kann, muss es eine Zerlegung $w = uv$ und einen Zustand $f_1 \in F_1$ geben mit $(s, w) = (s_1, uv) \vdash_A^* (f_1, v) \vdash_A (s_2, v) \vdash_A^* (f_2, \varepsilon)$.

Der erste Teil dieser Folge kann nur aus A_1-Schritten bestehen und der letzte Teil nur aus A_2-Schritten.

Also folgt $(s_1, u) \vdash_{A_1}^* (f_1, \varepsilon)$ (mit Lemma 1.6) und $(s_2, v) \vdash_{A_2}^* (f_2, \varepsilon)$, d.h. $u \in L(A_1)$ und $v \in L(A_2)$. Damit ist $w = uv \in L(A_1) \circ L(A_2)$.

$\supseteq$: Sei $w \in L(A_1) \circ L(A_2)$, d.h. $w = uv$ mit $u \in L(A_1)$ und $v \in L(A_2)$.

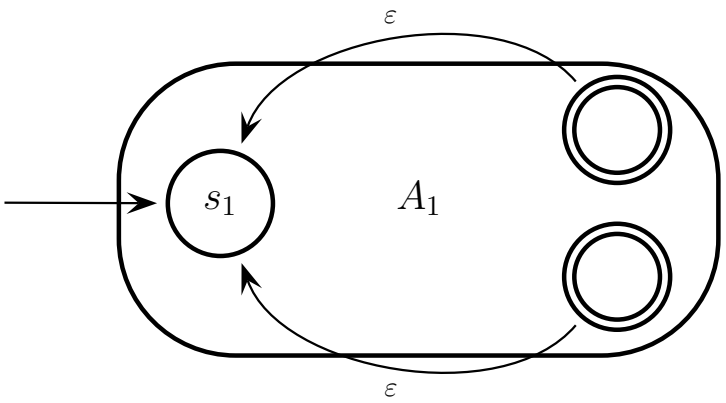
Dann gibt es $f_1 \in F_1$ und $f_2 \in F_2 = F$ mit $(s, u) = (s_1, u) \vdash_{A_1}^* (f_1, \varepsilon)$ und $(s_2, v) \vdash_{A_2}^* (f_2, \varepsilon)$.

Wegen $\Delta_1, \Delta_2 \subseteq \Delta$ folgt mit Lemma 1.6 $(s, w) = (s, uv) \vdash_A^* (f_1, v) \vdash_A (s_2, v) \vdash_A^* (f_2, \varepsilon)$, also $w \in L(A)$.

Abgeschlossenheit unter $^+$ und *.

Es genügt, $^+$ zu betrachten, denn $L^* = L^+ \cup \{\varepsilon\}$ und wir wissen schon, dass $\{\varepsilon\} \in \mathcal{L}_{EA}$ und dass $\mathcal{L}_{EA}$ unter $\cup$ abgeschlossen ist.

Ein ε-NDEA A, der $L(A_1)^+$ erkennt:



Formal:

Sei wieder $A_1 = (\Sigma, Q, s_1, F_1, \Delta_1)$.

Wir definieren $A = (\Sigma, Q, s_1, F_1, \Delta)$ mit:

$\Delta = \Delta_1 \cup \{(f, \varepsilon, s_1) \mid f \in F_1\}$

Zu zeigen: $L(A) = L(A_1)^+$

$\subseteq$: Sei $w \in L(A)$, d.h. $(s_1, w) \vdash_A^* (f, \varepsilon)$ für ein $f \in F_1$.

Wenn wir diese Folge von Übergangsschritten bei den neuen ε-Übergängen aufteilen, so erhalten wir eine Zerlegung $w = w_1 \dots w_n$ $(n \geq 1)$ mit:

$$\begin{array}{lllll} (s_1, w) & = & (s_1, w_1 \dots w_n) & \vdash_A^* & (f_1, w_2 \dots w_n) \\ & \vdash_A & (s_1, w_2 \dots w_n) & \vdash_A^* & (f_2, w_3 \dots w_n) \\ & \vdots & & & \\ & \vdash_A & (s_1, w_n) & \vdash_A^* & (f_n, \varepsilon) \end{array}$$

wobei $f_1, \dots, f_n \in F_1$, $f_n = f$ und alle $\vdash_A^*$ nur aus A_1-Schritten bestehen.

(Der Fall $n = 1$ tritt ein, wenn die Folge *keinen* der neuen ε-Übergänge enthält. Dann ist $w \in L(A_1) \subseteq L(A_1)^+$.)

Mit Lemma 1.6 folgt dann $(s_1, w_i) \vdash_{A_1}^* (f_i, \varepsilon)$, also $w_i \in L(A_1)$ für $i = 1, \dots, n$, und damit $w = w_1 \dots w_n \in L(A_1)^+$.

$\supseteq$: Sei $w \in L(A_1)^+$, d.h. $w = w_1 \dots w_n$ $(n \geq 1)$ mit $w_i \in L(A_1)$ für $i = 1, \dots, n$.

Dann gibt es $f_1, \dots, f_n \in F_1$ mit $(s_1, w_i) \vdash_{A_1}^* (f_i, \varepsilon)$, also folgt mit Lemma 1.6 $(s_1, w) = (s_1, w_1 \dots w_n) \vdash_A^* (f_1, w_2 \dots w_n) \vdash_A (s_1, w_2 \dots w_n) \vdash_A^* \dots \vdash_A^* (f_n, \varepsilon)$, und damit $w \in L(A)$. $\square$

Als Folgerung erhalten wir

Satz 1.17 $\mathcal{L}_{reg} \subseteq \mathcal{L}_{EA}$.

Beweis:

Eine Sprache ist regulär, wenn sie sich duch wiederholte Anwendung von $\cup$, $\circ$ und * aus endlichen Mengen aufbauen lässt.

Da $\mathcal{L}_{EA}$ abgeschlossen ist unter $\cup$, $\circ$ und *, bleibt nur noch zu zeigen, dass $\mathcal{L}_{EA}$ alle endlichen Teilmengen von Σ^* enthält.

Zunächst gilt dies für die einelementigen Mengen:

- $\{\varepsilon\} \in \mathcal{L}_{EA}$ nach Satz 1.16.
- Ist $w = a_1 \dots a_n \neq \varepsilon$, so ist $\{w\} = \{a_1\} \circ \dots \circ \{a_n\} \in \mathcal{L}_{EA}$ nach Satz 1.16.

Daraus folgt es für alle endlichen Mengen:

- $\emptyset \in \mathcal{L}_{EA}$ nach Satz 1.16.
- Ist $L = \{w_1, \dots, w_n\} \neq \emptyset$, so ist $L = \{w_1\} \cup \dots \cup \{w_n\} \in \mathcal{L}_{EA}$ nach Satz 1.16.

Die Sätze 1.16 und 1.17 (und ihre Beweise) sind wichtig für die Praxis (Compilerbau).

Denn sie liefern ein Verfahren, um aus der 'Mengenbeschreibung' einer regulären Sprache einen endlichen Automaten zu konstruieren.

Ein solches Verfahren wird in Scanner-Generatoren (z.B. Lex) verwendet, wobei man dort natürlich mehr auf die Effizienz der Algorithmen achten muss (vgl. Vorlesung Compilerbau) als wir es hier tun.

Als Eingabe verlangt ein Scanner-Generator sogenannte *reguläre Ausdrücke*, das sind formale Versionen unserer Mengenbeschreibungen, die wir später noch einführen werden.

Weitere Abschlusseigenschaften von $\mathcal{L}_{EA}$

Satz 1.18 *$\mathcal{L}_{EA}$ ist abgeschlossen unter Komplement, Durchschnitt, Mengendifferenz, Potenzierung und Spiegelung, d.h. wenn die Sprachen L_1 und L_2 von endlichen Automaten erkannt werden, dann werden auch $L_1 \cap L_2$, $\Sigma^* \setminus L_1$, $L_1 \setminus L_2$, $L_1^n\,(n \geq 0)$ und L_1^R von endlichen Automaten erkannt (und es gibt jeweils einen Algorithmus, mit dem man den neuen Automaten konstruieren kann).*

Beweis:

- Komplement:

  Ist $L_1 \in \mathcal{L}_{EA}$, so existiert ein *D*EA $A_1 = (\Sigma, Q_1, s_1, F_1, \delta_1)$ mit $L_1 = L(A_1)$.

  Sei $A = (\Sigma, Q_1, s_1, Q_1 \setminus F_1, \delta_1)$, d.h. A ist der DEA, der aus A_1 durch 'Komplementierung' der Endzustandsmenge entsteht.

  Dann gilt $L(A) = \Sigma^* \setminus L_1$, denn für jedes $w \in \Sigma^*$ gilt:

  $w \in L(A) \Leftrightarrow \delta_1^*(s, w) \in Q_1 \setminus F_1 \Leftrightarrow \delta_1^*(s, w) \notin F_1 \Leftrightarrow w \notin L(A_1) = L_1$

- Durchschnitt:

  Wenn $L_1, L_2 \in \mathcal{L}_{EA}$, dann gilt $L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2)) \in \mathcal{L}_{EA}$ wegen der Abgeschlossenheit unter Vereinigung und Komplement.

- Mengendifferenz:

  Wenn $L_1, L_2 \in \mathcal{L}_{EA}$, dann gilt $L_1 \setminus L_2 = L_1 \cap (\Sigma^* \setminus L_2) \in \mathcal{L}_{EA}$ wegen der Abgeschlossenheit unter Durchschnitt und Komplement.

- Potenzierung:

  Wenn $L_1 \in \mathcal{L}_{EA}$, dann gilt $L_1^n = L_1 \circ \ldots \circ L_1 \in \mathcal{L}_{EA}$ für alle $n \geq 1$ wegen der Abgeschlossenheit unter $\circ$, und $L_1^0 = \{\varepsilon\} \in \mathcal{L}_{EA}$ gilt sowieso.

- Spiegelung:

  Sei $A_1 = (\Sigma, Q_1, s_1, F_1, \Delta_1)$ ein NDEA, der L_1 erkennt.

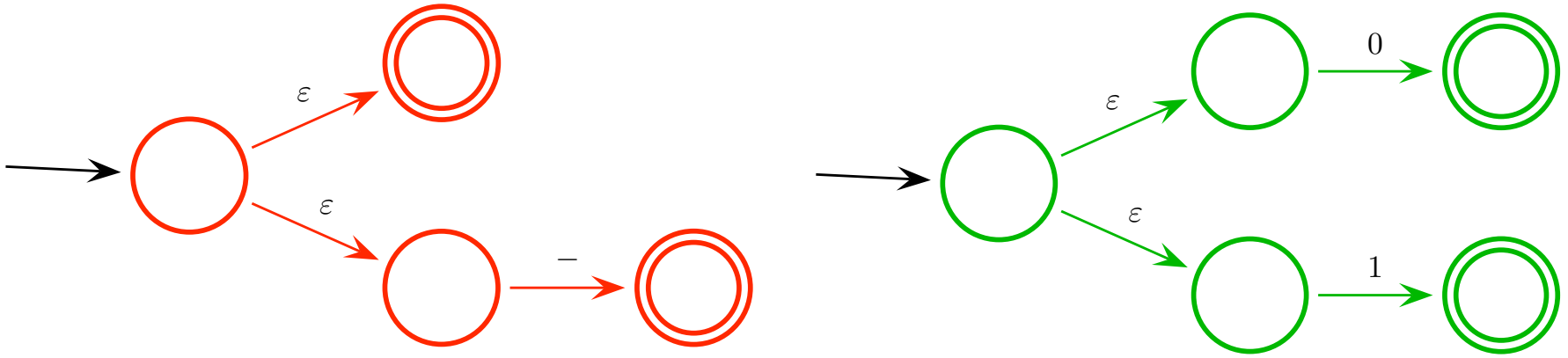
  Wir definieren einen MNDEA (Übung 5, Aufgabe 1) $A = (\Sigma, Q_1, S, F, \Delta)$ mit $S = F_1$, $F = \{s_1\}$ und $\Delta = \{(q, a, p) \mid (p, a, q) \in \Delta_1\}$,

  d.h. A ist der MNDEA, der aus A_1 entsteht, indem man alle Pfeile umkehrt und Start- und Endzustände vertauscht.

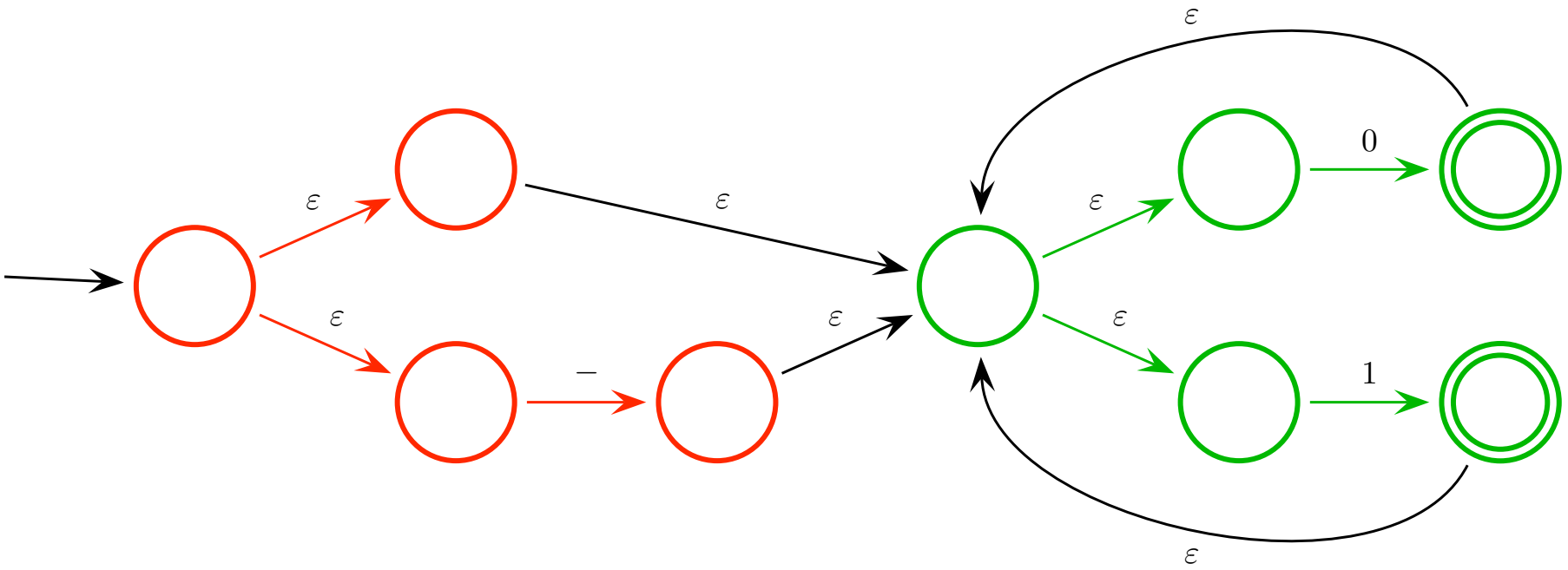
  Dann ist $L(A) = L^R$, weil man in A_1 genau dann mit w von s_1 zu einem Zustand $f \in F_1$ kommt, wenn man in A mit w^R von f nach s_1 kommt. $\square$

Die Beweise der Sätze 1.16, 1.17 und 1.18 liefern uns Algorithmen zur Konstruktion von ε-NDEAs aus Mengenausdrücken.

Beispiel:
Mit den Konstruktionen für $\{\varepsilon\}$, $\{a\}$ und $\cup$ erhält man ε-NDEAs für die Sprachen $\{\varepsilon, -\}$ und $\{0, 1\}$



Daraus ergibt sich mit den Konstruktionen für $^+$ und $\circ$ ein ε-NDEA für die Sprache $L_{bin} = \{\varepsilon, -\} \circ \{0,1\}^+$ aller Binärdarstellungen ganzer Zahlen.



Für diesen könnte man jetzt wieder einen äquivalenten NDEA bzw. DEA konstruieren.

Zum Beweis von $\mathcal{L}_{reg} = \mathcal{L}_{EA}$ fehlt noch

Satz 1.19 $\mathcal{L}_{EA} \subseteq \mathcal{L}_{reg}$

Beweis:

Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA mit $Q = \{q_1, \ldots, q_n\}$ und $s = q_1$.

Es ist zu zeigen, dass $L(A)$ regulär ist.

Dazu betrachten wir—für alle $i, j \in \{1, \ldots, n\}$—die Sprachen

$$L_{ij} = \{w \in \Sigma^* \mid (q_i, w) \vdash_A^* (q_j, \varepsilon)\}$$

$L(A)$ lässt sich als (endliche) Vereinigung einiger dieser Sprachen L_{ij} darstellen, nämlich:

$$\begin{aligned} L(A) &= \{w \in \Sigma^* \mid \text{ es existiert ein } q_j \in F \text{ mit } (q_1, w) \vdash_A^* (q_j, \varepsilon)\} \\ &= \bigcup_{q_j \in F} L_{1j} \end{aligned}$$

Deshalb genügt es zu zeigen, dass die Sprachen L_{ij} regulär sind.

Dazu betrachten wir eine weitere 'Verfeinerung' der Sprachen L_{ij}.

Für jedes $k \in \{1, \ldots, n+1\}$ definieren wir

$$L_{ij}^k = \{w \in \Sigma^* \mid (q_i, w) \vdash_A^* (q_j, \varepsilon), \text{ wobei in } \vdash_A^* \text{ nur Zwischenzustände } q_l \text{ mit } l < k \text{ benutzt werden}\}$$

Dann gilt $L_{ij} = L_{ij}^{n+1}$ (weil $l < n+1$ keine Einschränkung ist),

und die Sprachen L_{ij}^k lassen sich durch Induktion über k definieren:

$$L_{ij}^1 = \{w \in \Sigma^* \mid (q_i, w) \vdash_A^* (q_j, \varepsilon) \quad \textit{ohne} \quad \text{Zwischenzustände}\}$$

$$= \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} \text{ falls } i \neq j \\ \{a \in \Sigma \mid \delta(q_i, a) = q_j\} \cup \{\varepsilon\} \text{ falls } i = j \end{cases}$$

$$L_{ij}^{k+1} = L_{ij}^{k} \cup L_{ik}^{k} \circ (L_{kk}^{k})^* \circ L_{kj}^{k}$$

Diese Gleichung erhält man durch folgende Überlegung:

Wenn $w \in L_{ij}^{k+1}$, dann gibt es zwei Möglichkeiten:

Entweder q_k taucht gar nicht als Zwischenzustand in der Folge $(q_i, w) \vdash_A^* (q_j, \varepsilon)$ auf.

Dann gilt schon $w \in L_{ij}^{k}$.

Oder wir können die Folge $(q_i, w) \vdash_A^* (q_j, \varepsilon)$ überall dort unterteilen, wo q_k auftaucht.

Dann erhalten wir eine Zerlegung $w = w_1 \dots w_m$ $(m \geq 2)$ mit

$$(q_i, w_1 \dots w_m) \vdash_A^* (q_k, w_2 \dots w_m) \vdash_A^* \dots \vdash_A^* (q_k, w_m) \vdash_A^* (q_j, \varepsilon)$$

in der alle $\vdash_A^*$ nur noch Zwischenzustände q_l mit $l < k$ enthalten.

Mit Lemma 1.6 folgt dann

$$\begin{aligned}(q_i, w_1) &\vdash_A^* (q_k, \varepsilon)\\ (q_k, w_l) &\vdash_A^* (q_k, \varepsilon) \text{ für } l = 2, \ldots, m-1\\ (q_k, w_m) &\vdash_A^* (q_j, \varepsilon)\end{aligned}$$

wobei die $\vdash_A^*$ immer noch die gleichen Zwischenzustände enthalten, also nur Zustände q_l mit $l < k$.

Also gilt $w_1 \in L_{ik}^k$, $w_2, \ldots, w_{m-1} \in L_{kk}^k$ und $w_m \in L_{kj}^k$

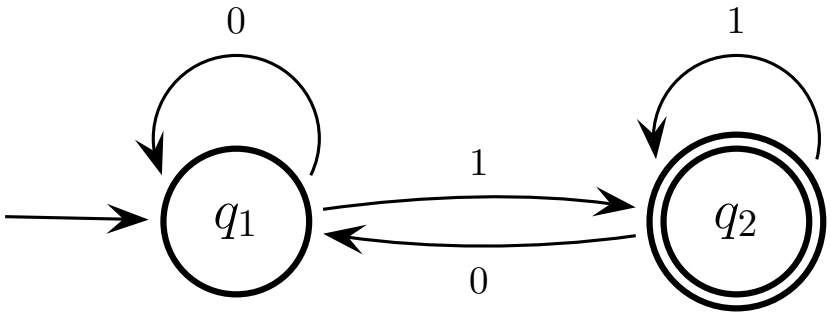
und damit $w \in L_{ik}^k \circ (L_{kk}^k)^* \circ L_{kj}^k$.

Damit ist '$\subseteq$' bewiesen und '$\supseteq$' ergibt sich ähnlich (einfacher!), indem man Folgen von Übergangsschritten *zusammensetzt*.

Da alle L_{ij}^1 endlich sind, und jedes L_{ij}^{k+1} durch Anwendung der Operationen $\cup$, $\circ$ und * aus einigen $L_{i'j'}^k$ entsteht, folgt (durch Induktion über k), dass alle L_{ij}^k regulär sind. $\square$

Auch der Beweis von Satz 1.19 ist konstruktiv. Er zeigt uns, wie wir einen 'Mengenausdruck' für die von einem DEA erkannte Sprache finden können.

Beispiel: Sei A der DEA



Wir bestimmen einen Mengenausdruck für $L(A)$, wobei wir (schon bei den Zwischenrechnungen) Vereinfachungen durchführen, damit der Gesamtausdruck nicht zu groß wird.

$$\begin{aligned} L(A) &= L_{12} \quad \text{da } q_2 \text{ einziger Endzustand ist.} \\ &= L_{12}^3 \quad \text{da } A \text{ zwei Zustände hat.} \end{aligned}$$

$$
\begin{aligned}
L_{12}^3 &= L_{12}^2 \cup L_{12}^2 \circ (L_{22}^2)^* \circ L_{22}^2 && \text{laut Gleichung für } L_{ij}^{k+1} \\
&= L_{12}^2 \cup L_{12}^2 \circ (L_{22}^2)^+ && \text{da } L^* \circ L = L^+ \\
&= L_{12}^2 \circ \{\varepsilon\} \cup L_{12}^2 \circ (L_{22}^2)^+ && \text{da } \{\varepsilon\} \text{ neutrales Element für } \circ \text{ ist} \\
&= L_{12}^2 \circ (\{\varepsilon\} \cup (L_{22}^2)^+) && \text{da } \circ \text{ distributiv über } \cup \text{ ist} \\
&= L_{12}^2 \circ (L_{22}^2)^* && \text{da } \{\varepsilon\} \cup L^+ = L^*
\end{aligned}
$$

$$
\begin{aligned}
L_{12}^2 &= L_{12}^1 \cup L_{11}^1 \circ (L_{11}^1)^* \circ L_{12}^1 && \text{laut Gleichung für } L_{ij}^{k+1} \\
&= \{1\} \cup \{\varepsilon, 0\} \circ \{\varepsilon, 0\}^* \circ \{1\} && \text{laut Gleichung für } L_{ij}^1 \\
&= \{1\} \cup \{\varepsilon, 0\}^+ \circ \{1\} && \text{da } L \circ L^* = L^+ \\
&= \{1\} \cup \{0\}^* \circ \{1\} && \text{da } (L \cup \{\varepsilon\})^+ = L^* \\
&= \{0\}^* \circ \{1\} && \text{da } \{1\} \subseteq \{0\}^* \circ \{1\}
\end{aligned}
$$

$$
\begin{aligned}
L_{22}^2 &= L_{22}^1 \cup L_{21}^1 \circ (L_{11}^1)^* \circ L_{12}^1 && \text{laut Gleichung für } L_{ij}^{k+1} \\
&= \{\varepsilon, 1\} \cup \{0\} \circ \{\varepsilon, 0\}^* \circ \{1\} && \text{laut Gleichung für } L_{ij}^1 \\
&= \{\varepsilon, 1\} \cup \{0\} \circ \{0\}^* \circ \{1\} && \text{da } (L \cup \{\varepsilon\})^* = L^* \\
&= \{\varepsilon, 1\} \cup \{0\}^+ \circ \{1\} && \text{da } L \circ L^* = L^+
\end{aligned}
$$

Also gilt $L(A) = \{0\}^* \circ \{1\} \circ (\{\varepsilon, 1\} \cup \{0\}^+ \circ \{1\})^*$

Natürlich ließe sich dieser Ausdruck noch weiter vereinfachen, letztendlich zu $\{0,1\}^* \circ \{1\}$, weil das die von A erkannte Sprache ist. In seiner jetzigen Form ist er aber aufschlussreicher, weil er noch die Konstruktionsidee aus dem Beweis von Satz 1.19 erkennen lässt:

Die Menge $\{0\}^* \circ \{1\}$ enthält alle Wörter, die von q_1 nach q_2 führen, wobei nur q_1 als Zwischenzustand benutzt wird. Und die Menge $\{\varepsilon, 1\} \cup \{0\}^+ \circ \{1\}$ enthält alle Wörter, die von q_2 nach q_2 führen, wobei auch wieder nur q_1 als Zwischenzustand benutzt wird.

Reguläre Sprachen und endliche Automaten

Mit den Sätzen 1.17 und 1.19 ist bewiesen, dass

$$\mathcal{L}_{reg} = \mathcal{L}_{EA}$$

und deshalb sprechen wir ab jetzt nur noch von *regulären Sprachen*.

Mit anderen Worten:

Wir haben bisher nur *eine* Sprachklasse kennengelernt, nämlich die Klasse der regulären Sprachen, aber wir haben unterschiedliche Formalismen zur Darstellung regulärer Sprachen:

- DEAs
- NDEAs
- ε-NDEAs
- und 'Mengenausdrücke'

Problem: Für 'Mengenausdrücke' haben wir keine formale (sondern nur die übliche mathematische) Schreibweise.

Deshalb: Reguläre Ausdrücke

Definition 1.20 *Sei Σ ein Alphabet. Die Menge aller regulären Ausdrücke über Σ ist induktiv definiert durch:*

1. *$\emptyset$ ist ein regulärer Ausdruck über Σ.*
2. *Jedes $a \in \Sigma$ ist ein regulärer Ausdruck über Σ.*
3. *Wenn α und β reguläre Ausdrücke über Σ sind, dann sind auch $(\alpha\beta)$, $(\alpha \mid \beta)$ und $(\alpha)^\star$ reguläre Ausdrücke über Σ.*

Ein regulärer Ausdruck über Σ ist also ein Wort über dem Alphabet $\Sigma' = \Sigma \cup \{\emptyset, (,), |, \star\}$, das nach den Regeln 1. bis 3. aufgebaut ist.

Damit ist die *Syntax* der regulären Ausdrücke festgelegt.

Es fehlt noch die *Semantik*.

Ein regulärer Ausdruck soll natürlich die Sprache beschreiben, die durch den 'entsprechenden Mengenausdruck' definiert ist.

Definition 1.21 *Die von einem regulären Ausdruck* α beschriebene Sprache $L(\alpha)$ *ist durch Induktion über die Größe von* α *wie folgt definiert:*

1. $L(\varnothing) = \emptyset$
2. $L(a) = \{a\}$
3. $L((\alpha\beta)) = L(\alpha) \circ L(\beta)$

   $L((\alpha \mid \beta)) = L(\alpha) \cup L(\beta)$

   $L((\alpha)^\star) = (L(\alpha))^*$

Man beachte:
Links stehen *Zeichen* ø, | und *.
Rechts stehen mathematische Schreibweisen: $\emptyset$, { }, $\cup$, $\circ$ und *.

Vereinbarung zur Einsparung von Klammern

- Die Konkatenation bindet stärker als '|'.
- '*' bindet am stärksten.
- Konkatenation und '|' sind linksassoziativ, d.h. $\alpha\beta\gamma$ steht für $(\alpha\beta)\gamma$ und $\alpha \mid \beta \mid \gamma$ für $(\alpha \mid \beta) \mid \gamma$.

  (Ebenso gut könnte man Rechtsassoziativität vereinbaren, da die Mengenoperationen $\cup$ und $\circ$ sowieso assoziativ sind.)

Beispiele

- $L_{nat} = \{0, \ldots, 9\}^+ = L((0 \mid \ldots \mid 9)\,(0 \mid \ldots \mid 9)^\star)$
- $L_{int} = \{\varepsilon, -\} \circ L_{nat} = L((\emptyset^\star \mid -)(0 \mid \ldots \mid 9)\,(0 \mid \ldots \mid 9)^\star)$
- $L(a^\star\, b^\star) = \{a\}^* \circ \{b\}^* = \{a^m\, b^n \mid m, n \geq 0\}$
- $L((ab)^\star) = (\{a\} \circ \{b\})^* = \{ab\}^* = \{(ab)^n \mid n \geq 0\}$

Per Definition der Semantik regulärer Ausdrücke ist es klar, dass die von einem regulären Ausdruck beschriebene Sprache stets regulär ist. Umgekehrt lässt sich jede reguläre Sprache durch einen regulären Ausdruck beschreiben, denn:

- Alle endlichen Sprachen erhält man mit Konkatenation und Vereinigung aus $\emptyset$, $\{\varepsilon\}$ und den Mengen $\{a\}$ mit $a \in \Sigma$ (vgl. Beweis zu Satz 1.19).
- Diese Mengen kann man durch reguläre Ausdrücke beschreiben, nämlich $\emptyset = L(\varnothing)$, $\{\varepsilon\} = \emptyset^* = L(\varnothing^\star)$ und $\{a\} = L(a)$.

Also gilt:

Satz 1.22 *Eine Sprache ist genau dann regulär, wenn sie sich durch einen regulären Ausdruck beschreiben lässt.*

Mit den regulären Ausdrücken haben wir eine *formale Syntax* zur Beschreibung regulärer Sprachen (anstelle der "üblichen mathematischen Schreibweise" bei Mengenausdrücken).

Natürlich übertragen sich die bisherigen Beobachtungen über Mengenausdrücke unmittelbar auf reguläre Ausdrücke, insbesondere:

Die Konstruktion von ε-NDEAs aus dem Beweis zu Satz 1.16 (und Satz 1.17) liefert zu jedem regulären Ausdruck einen äquivalenten ε-NDEA.

Die Konstruktion der Sprachen L_{ij} aus dem Beweis von Satz 1.19 lässt sich so abwandeln, dass sie zu jedem DEA einen äquivalenten regulären Ausdruck liefert.

Damit haben wir *Übersetzungsalgorithmen*, um all die unterschiedlichen Darstellungen regulärer Sprachen (DEAs, NDEAs, ε-NDEAs und reguläre Ausdrücke) ineinander überzuführen.

Bezug zur Praxis

Reguläre Ausdrücke werden benutzt

- als Eingabe für Scanner-Generatoren (Compilerbau)
- als Suchmuster in Editoren und Suchmaschinen
- zur Textverarbeitung in Programmiersprachen (z.B. Java)

Meistens hat man eine erweiterte Syntax für reguläre Ausdrücke:

- zusätzliche Zeichen wie ϵ, $^+$ oder ein Zeichen für das Komplement
- Schreibweisen für endliche Zeichenmengen, z.B. [a-$z$$A$-$Z$]
- Einführung von Namen zur Wiederverwendung eines Ausdrucks, z.B $nat = [0\text{-}9]^+$, $int = (\epsilon \mid -)\, nat$

Intern werden reguläre Ausdrücke in EAs umgewandelt,
im Prinzip nach unserer Theorie, aber mit effizienteren Algorithmen
(z.B. direkt vom regulären Ausdruck zum DEA).

Reguläre Sprachen und endliche Automaten

Nachdem wir wissen, dass $\mathcal{L}_{reg} = \mathcal{L}_{EA}$ gilt, können wir jetzt all unsere Formalismen und auch die bewiesenen Abschlusseigenschaften kombinieren, um zu zeigen, dass eine Sprache regulär ist.

Beispiel:
Sei $L = \{w \in \{0,1\}^* \mid w \text{ enthält } 010, \text{ aber nicht } 110\}$.
Dann gilt $L = L((0 \mid 1)^* 010 (0 \mid 1)^*) \setminus L((0 \mid 1)^* 110 (0 \mid 1)^*)$,
also ist L regulär.

Aber auch, um zu zeigen, dass eine Sprache *nicht* regulär ist.

Beispiel:
Sei $L = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$
Dann ist $L \cap L((a^\star b^\star)) = \{a^n b^n \mid n \geq 0\}$,
d.h. wenn L regulär wäre, dann wäre auch $\{a^n b^n \mid n \geq 0\}$ als Durchschnitt zweier regulärer Sprachen wieder regulär.
Aber von dieser Sprache wissen wir schon, dass sie nicht regulär ist.

Entscheidbarkeitsfragen

Neben den Übersetzungsalgorithmen interessieren uns auch *Entscheidungsalgorithmen*. Das sind Algorithmen, die gewisse Fragestellungen stets korrekt mit 'ja' oder 'nein' beantworten, z.B.

- die Frage, ob ein Automat die leere Sprache erkennt
- die Frage, ob zwei Automaten äquivalent sind.

In jedem Fall muss geklärt werden,

- welche Eingabedaten der Algorithmus erhält,
- welche Frage über die Eingabedaten er beantworten soll

  (also wie seine Ausgabe von der Eingabe abhängen soll)

Satz 1.23 *Für jede der folgenden Fragestellungen gibt es einen Entscheidungsalgorithmus.*

1. *Eingabe: Ein DEA A und ein Wort w. Frage: Ist $w \in L(A)$?*

   (das sogenannte Wortproblem für DEAs)
2. *Eingabe: Ein DEA A. Frage: Ist $L(A) = \emptyset$?.*
3. *Eingabe: Ein DEA A. Frage: Ist $L(A) = \Sigma^*$?.*
4. *Eingabe: Zwei DEAs A_1 und A_2. Frage: Ist $L(A_1) \subseteq L(A_2)$?*
5. *Eingabe: Zwei DEAs A_1 und A_2. Frage: Ist $L(A_1) = L(A_2)$?*

Beweis:

1. Der Algorithmus muss nur den Ablauf des DEA A bei Eingabe w simulieren und dann überprüfen, ob der erreichte Zustand ein Endzustand ist.

2. $L(A) = \emptyset$ gilt genau dann, wenn *kein* Endzustand erreichbar ist. Also berechnet man die Menge der erreichbaren Zustände von A und überprüft, ob darunter ein Endzustand ist.

3. Es gilt $L(A) = \Sigma^* \Leftrightarrow \Sigma^* \setminus L(A) = \emptyset$. Also konstruiert man zu A einen DEA $\bar{A}$ mit $L(\bar{A}) = \Sigma^* \setminus L(A)$ (Algorithmus schon bekannt) und überprüft, ob $L(\bar{A}) = \emptyset$.

4. $L(A_1) \subseteq L(A_2)$ gilt genau dann, wenn $L(A_1) \cap (\Sigma^* \setminus L(A_2)) = \emptyset$. Also konstruiert man zu A_1 und A_2 einen DEA A mit $L(A) = L(A_1) \cap (\Sigma^* \setminus L(A_2))$ und überprüft, ob $L(A) = \emptyset$.

5. Um $L(A_1) = L(A_2)$ zu testen, überprüft man, ob $L(A_1) \subseteq L(A_2)$ und $L(A_2) \subseteq L(A_1)$ gilt. □

Natürlich gilt Satz 1.23 auch für NDEAs, ε-NDEAs oder reguläre Ausdrücke anstelle von DEAs.

Denn wir können jeden dieser Formalismen in einen äquivalenten DEA übersetzen und brauchen dann nur noch die entsprechende Frage für den DEA zu beantworten

(weil es ja Fragen über die erkannte Sprache sind).

Grenzen regulärer Sprachen

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

Oder allgemeiner:

Wie findet man heraus, *ob* eine Sprache regulär ist oder nicht?

Wir hatten schon ein Beispiel—nämlich $L = \{a^n b^n \mid n \geq 0\}$—mit dem *Schubfachprinzip* bewiesen.

Jeder DEA A verteilt die Wörter $w \in \Sigma^*$ auf endlich viele 'Schubladen', nämlich auf seine Zustände.

Wenn wir also *unendlich viele* Wörter finden, von denen je zwei *nicht* in der gleichen Schublade stecken dürfen, so kann kein DEA für die Sprache existieren.

Bei der Sprache L waren das die Wörter a^n mit $n \geq 0$.

Die Annahme, dass zwei dieser Wörter, a^i und a^j, in ein- und demselben Zustand landen, führt zum Widerspruch,

denn dann würden auch $a^i b^i$ und $a^j b^i$ in einem gemeinsamen Zustand landen, obwohl $a^i b^i \in L$ und $a^j b^i \notin L$.

Definition 1.24 *Sei $L \subseteq \Sigma^*$. Zwei Wörter $u_1, u_2 \in \Sigma^*$ heißen L-unterscheidbar, wenn ein Wort $v \in \Sigma^*$ existiert mit $u_1 v \in L$ und $u_2 v \notin L$ oder umgekehrt (d.h. wenn man sie nicht in die gleiche Schublade stecken darf).*

Damit erhalten wir das folgende

Beweisprinzip: Um zu zeigen, dass eine Sprache L nicht regulär ist, genügt es, *unendlich viele* Wörter in Σ^* zu finden, die *paarweise L-unterscheidbar* sind.

Beispiele nicht regulärer Sprachen:

1. "Ein EA kann nicht zählen"

- $L = \{a^n b^n \mid n \geq 0\}$: Die Wörter a^n mit $n \geq 0$ sind paarweise L-unterscheidbar, da $a^i b^i \in L$ und $a^j b^i \notin L$ für alle $i \neq j$.
- $L_1 = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$: Die Wörter a^n mit $n \geq 0$ sind paarweise L_1-unterscheidbar, da $a^i b^i \in L_1$ und $a^j b^i \notin L_1$ für alle $i \neq j$.
- $L_2 = \{a^n b^m \mid 0 \leq n < m\}$: Die Wörter a^n mit $n \geq 0$ sind paarweise L_2-unterscheidbar, denn: Wenn $i \neq j$, dann dürfen wir $i < j$ annehmen (weil L-Unterscheidbarkeit eine symmetrische Relation ist) und erhalten $a^i b^{i+1} \in L_2$, aber $a^j b^{i+1} \notin L_2$ weil $j \geq i+1$.
- $L_3 = \{a^n b^m \mid 0 \leq n < 2^m\}$: Die Wörter a^{2^n-1} mit $n \geq 0$ sind paarweise L_3-unterscheidbar, denn: Für alle $i < j$ gilt $a^{2^i-1} b^i \in L_2$ weil $2^i - 1 < 2^i$ und $a^{2^j-1} b^i \notin L_2$ weil $2^j - 1 \geq 2^i + 2^i - 1 \geq 2^i$.

2. "Ein EA kann sich keine beliebig großen Wörter merken"

- $L_4 = \{ww \mid w \in \{a,b\}^*\}$: Die Wörter $a^n b$ mit $n \geq 0$ sind paarweise L_4-unterscheidbar, denn: Für alle $i \neq j$ ist $a^i b a^i b \in L_4$ und $a^j b a^i b \notin L_4$. (Man kann auch die Wörter a^n betrachten, denn a^i und a^j lassen sich durch ba^ib unterscheiden.)
- $L_5 = \{w \in \{a,b\}^* \mid w = w^R\}$ siehe Übungsblatt 5, Aufgabe 3

3. "Eine reguläre Sprache kann keine beliebig großen Lücken enthalten"

- $L_5 = \{a^{n^2} \mid n \geq 0\}$: Die Wörter aus L_5 sind paarweise L_5-unterscheidbar, denn für alle $i < j$ gilt $a^{i^2} a^{2i+1} = a^{i^2+2i+1} = a^{(i+1)^2} \in L_5$ und $a^{j^2} a^{2i+1} = a^{j^2+2i+1} \notin L_5$, weil $j^2 + 2i + 1 < j^2 + 2j + 1 = (j+1)^2$ zwischen den beiden aufeinanderfolgenden Quadratzahlen j^2 und $(j+1)^2$ liegt, und deshalb selbst *keine* Quadratzahl sein kann.

- $L_6 = \{a^{2^n} \mid n \geq 0\}$: Die Wörter aus L_6 sind paarweise L_6-unterscheidbar, denn für alle $i < j$ gilt $a^{2^i}a^{2^i} = a^{2^i+2^i} = a^{2^{i+1}} \in L_6$ und $a^{2^j}a^{2^i} = a^{2^j+2^i} \notin L_6$, weil $2^j + 2^i < 2^j + 2^j = 2^{j+1}$ zwischen den beiden aufeinanderfolgenden Zweierpotenzen 2^j und 2^{j+1} liegt und deshalb selbst *keine* Zweierpotenz sein kann.

- $L_7 = \{a^p \mid p$ ist Primzahl$\}$:

  Wir wollen beweisen, dass zwei *beliebige* Wörter $a^i, a^j \in \{a\}^*$ L_7-unterscheidbar sind, wobei wir wieder $i < j$ annehmen dürfen.

  Dazu brauchen wir eine Zahl k mit $a^j a^k \in L_7$ und $a^i a^k \notin L_7$, d.h. $j + k$ ist Primzahl und $i + k$ nicht.

  Um ein solches k zu finden, genügt es zu wissen, dass die Menge der Primzahlen beliebig große Lücken enthält, d.h. für jedes $n > 0$ existieren n aufeinanderfolgende Zahlen, die keine Primzahlen sind.

Sei p die kleinste Primzahl, die über einer solchen Lücke der Größe j liegt.

Dann sind (mindestens) die Zahlen $p-j, \ldots, p-1$ *keine* Primzahlen.

Also können wir $k = p - j$ wählen, denn $j + k = p$ ist eine Primzahl und $i + k = i + p - j = p - (j - i)$ fällt in die Lücke und ist deshalb *keine* Primzahl.

Die Existenz beliebig großer Lücken in der Menge der Primzahlen lässt sich leicht einsehen:

Eine Lücke der Größe n bilden z.B. die Zahlen $(n+1)! + 2, \ldots, (n+1)! + n + 1$, denn:

$(n+1)!$ ist durch jede der Zahlen $2, \ldots, n+1$ teilbar, also ist $(n+1)! + m$ durch m teilbar für jedes $m \in \{2, \ldots, n+1\}$, und damit *keine* Primzahl.

Eine alternative Formulierung des Beweisprinzips

Definition 1.25 *Sei $L \subseteq \Sigma^*$. Zwei Wörter $u_1, u_2 \in \Sigma^*$ heißen L-äquivalent ($u_1 \sim_L u_2$), wenn sie nicht L-unterscheidbar sind, d.h. wenn für jedes $v \in \Sigma^*$ entweder $u_1v, u_2v \in L$ oder $u_1v, u_2v \notin L$.*

Wir schreiben $\not\sim_L$ für die Verneinung von $\sim_L$, d.h. $u_1 \not\sim_L u_2$ bedeutet, dass u_1 und u_2 L-unterscheidbar sind.

Für jede Sprache $L \subseteq \Sigma^*$ ist $\sim_L$ eine Äquivalenzrelation auf Σ^* (reflexiv, transitiv und symmetrisch). Das sieht man am besten, wenn man die Definition von $\sim_L$ etwas umformuliert: Für jedes $u \in \Sigma^*$ sei

$$Erg_L(u) = \{v \in \Sigma^* \mid uv \in L\}$$

die Menge der *L-Ergänzungen* von u. Dann gilt

$$\begin{aligned} u_1 \sim_L u_2 &\Leftrightarrow Erg_L(u_1) = Erg_L(u_2) \\ &\Leftrightarrow \text{für alle } v \in \Sigma^* gilt : u_1v \in L \Leftrightarrow u_2v \in L \end{aligned}$$

Intuition:
Die L-Ergänzungen von u sind genau die Restwörter, die das Anfangswort u in die Sprache L überführen, also genau die Wörter, die man 'noch erwartet', wenn man bereits u gelesen hat.
Also sind zwei Wörter u_1, u_2 genau dann L-äquivalent, wenn man nach Einlesen von u_1 und u_2 die gleichen Restwörter erwartet, d.h. wenn man u_1 und u_2 in die gleiche 'Schublade' packen darf.

Schreibweise:
Mit $[u]_L$ bezeichnen wir die *L-Äquivalenzklasse* eines Wortes $u \in \Sigma^*$, d.h.

$$[u]_L \;=\; \{u' \in \Sigma^* \mid u' \sim_L u\}$$

Es gilt also

$$u_1 \sim_L u_2 \;\Leftrightarrow\; [u_1]_L = [u_2]_L$$
$$u_1 \not\sim_L u_2 \;\Leftrightarrow\; [u_1]_L \neq [u_2]_L$$

Damit können wir unser **Beweisprinzip** neu formulieren:

Um zu zeigen, dass eine Sprache L *nicht* regulär ist, genügt es, unendlich viele Äquivalenzklassen $[u]_L$ zu finden

(denn unendlich viele paarweise L-unterscheidbare Wörter sind nichts anderes als die Vertreter von unendlich vielen Äquivalenzklassen).

Wir wollen zeigen, dass auch die Umkehrung gilt, d.h. dass eine Sprache L regulär ist, wenn es nur endlich viele L-Äquivalenzklassen gibt.

Beides wird zusammengefasst im **Satz von Myhill und Nerode**, zu dessen Vorbereitung wir zunächst noch einige Eigenschaften der Relation $\sim_L$ beweisen.

Lemma 1.26

1. *Für alle* $u_1, u_2, v \in \Sigma^*$ *gilt: Wenn* $u_1 \sim_L u_2$, *dann* $u_1 v \sim_L u_2 v$.

   (Eine Äquivalenzrelation mit dieser Eigenschaft bezeichnet man als Rechtskongruenzrelation*.)*

2. *Für jedes* $u \in \Sigma^*$ *gilt:* $u \in L \Leftrightarrow [u]_L \subseteq L$. *(Also* $L = \bigcup_{u \in L} [u]_L$*.)*

Beweis:

1. Seien $u_1, u_2, v \in \Sigma^*$ und $u_1 \sim_L u_2$. Dann gilt für alle $v' \in \Sigma^*$:

   $(u_1 v)v' \in L \;\Leftrightarrow\; u_1(vv') \in L \;\Leftrightarrow\; u_2(vv') \in L \;\Leftrightarrow\; (u_2 v)v' \in L$

   Also ist $u_1 v \sim_L u_2 v$.

2. '$\Leftarrow$' ist klar.

   '$\Rightarrow$': Sei $u \in L$ und $u' \sim_L u$.

   Wegen $u\varepsilon \in L$ ist dann auch $u'\varepsilon \in L$, also $u' \in L$. □

Satz 1.27 (Satz von Myhill und Nerode) *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann regulär, wenn die Äquivalenzrelation $\sim_L$ nur endlich viele Äquivalenzklassen besitzt.*

Beweis:

'$\Rightarrow$': Sei $L \subseteq \Sigma^*$ regulär.

Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA mit $L(A) = L$

und seien $u_1, u_2 \in \Sigma^*$ mit $\delta^*(s, u_1) = \delta^*(s, u_2)$.

Dann gilt für alle $v \in \Sigma^*$:

$\delta^*(s, u_1 v) = \delta^*(\delta^*(s, u_1), v) = \delta^*(\delta^*(s, u_2), v) = \delta^*(s, u_2 v)$,

also $u_1 v \in L \Leftrightarrow \delta^*(s, u_1 v) \in F \Leftrightarrow \delta^*(s, u_2 v) \in F \Leftrightarrow u_2 v \in L$.

Das bedeutet $u_1 \sim_L u_2$.

Damit haben wir bewiesen, dass zwei Wörter, die in A zum gleichen Zustand führen, stets L-äquivalent sind.

Also besitzt $\sim_L$ nur endlich viele Äquivalenzklassen,

nämlich höchstens n, wobei n die Anzahl der erreichbaren Zustände von A ist.

'$\Leftarrow$': Sei $L \subseteq \Sigma^*$ eine Sprache, die nur endlich viele L-Äquivalenzklassen besitzt.

Da L-äquivalente Wörter in die gleiche 'Schublade' gesteckt werden können, definieren wir einen DEA, der für jede L-Äquivalenzklasse eine passende Schublade besitzt.

Deshalb nehmen wir die L-Äquivalenzklassen selbst als Schubladen, d.h. als Zustände eines DEA.

Sei $A = (\Sigma, Q, s, F, \delta)$ mit:

$Q = \{[u]_L \mid u \in \Sigma^*\}$ (die endliche Menge der L-Äquivalenzklassen)

$s = [\varepsilon]_L$

$F = \{[u]_L \mid u \in L\}$ $(= \{[u]_L \mid [u]_L \subseteq L\}$ nach Lemma 1.26)

$\delta : Q \times \Sigma \to Q$ $\delta([u]_L, a) = [ua]_L$

Man beachte, dass δ wohldefiniert ist, d.h. dass $[ua]_L$ unabhängig von der Wahl des speziellen Vertreters $u \in [u]_L$ ist.

Wenn nämlich $u' \sim_L u$ ein anderes Element aus $[u]_L$ ist, so folgt $u'a \sim_L ua$ und damit $[u'a]_L = [ua]_L$.

Um zu zeigen, dass A tatsächlich die Sprache L erkennt, beweisen wir, dass jedes Wort $w \in \Sigma^*$ in die passende Schublade gerät, d.h. dass

$$\delta^*(s, w) = [w]_L \quad \text{für alle } w \in \Sigma^* \qquad (*)$$

$w = \varepsilon$:

$$\begin{aligned} \delta^*(s,\varepsilon) &= s && \text{per Definition von } \delta^* \\ &= [\varepsilon]_L && \text{per Definition von } s \end{aligned}$$

$w = va$:

$$\begin{aligned} \delta^*(s,va) &= \delta(\delta^*(s,v),a) && \text{per Definition von } \delta^* \\ &= \delta([v]_L,a) && \text{nach Induktionsannahme für } v \\ &= [va]_L && \text{per Definition von } \delta \end{aligned}$$

Damit erhalten wir:

$$\begin{aligned} w \in L(A) &\Leftrightarrow \delta^*(s,w) \in F && \text{per Definition von } L(A) \\ &\Leftrightarrow [w]_L \in F && \text{wegen (*)} \\ &\Leftrightarrow [w]_L \subseteq L && \text{per Definition von } F \\ &\Leftrightarrow w \in L && \text{nach Lemma 1.26} \end{aligned}$$

□

Im Beweis des Satzes von Myhill und Nerode haben wir gesehen:

1. Wenn $A = (\Sigma, Q, s, F, \delta)$ ein DEA ist, der die Sprache L erkennt,

   und $u_1, u_2 \in \Sigma^*$ mit $\delta^*(s, u_1) = \delta^*(s, u_2)$,

   dann ist $[u_1]_L = [u_2]_L$.

   Also ist die Anzahl der L-Äquivalenzklassen höchstens so groß wie die Anzahl der (erreichbaren) Zustände von A.

2. Wenn $L \subseteq \Sigma^*$ nur endlich viele L-Äquivalenzklassen besitzt,

   dann gibt es einen DEA, der die Sprache L erkennt, und der die L-Äquivalenzklassen als Zustände besitzt.

   Also hat dieser DEA—wegen 1.—die *kleinstmögliche* Anzahl von Zuständen unter allen DEAs, die L erkennen.

Einen solchen DEA nennen wir einen *minimalen DEA* (für L).

Den speziellen minimalen DEA aus dem Beweis von Satz 1.27 nennen wir den *Myhill-Nerode-Automaten* für L.

Wir werden später sehen, dass er—in gewissem Sinne—der einzige minimale DEA für L ist.

Der Beweis von Satz 1.27 zeigt sogar, wie man den Myhill-Nerode-Automaten für L *konstruiert*, unter der Voraussetzung, dass man die L-Äquivalenzklassen 'kennt', d.h. dass man ihre Anzahl und ihre Elemente kennt (oder zumindest einen Teil der Elemente).

Wir wollen das an einem Beispiel illustrieren.

Beispiel: $\Sigma = \{0,1\}$ und $L = \{w \in \Sigma^* \mid 0100 \text{ ist Teilwort von } w\}$.
Bestimmung des Myhill-Nerode-Automaten:

$[\varepsilon]_L$	$Erg_L(\varepsilon) = L$
$[0]_L$	$Erg_L(0) = L \cup \{100\} \circ \Sigma^*$
$[1]_L = [\varepsilon]_L$	denn $Erg_L(1) = L = Erg_L(\varepsilon)$
	also $\delta([\varepsilon]_L, 1) = [1]_L = [\varepsilon]_L$
$[00]_L = [0]_L$	denn $Erg_L(00) = L \cup \{100\} \circ \Sigma^* = Erg_L(0)$
	also $\delta([0]_L, 0) = [00]_L = [0]_L$
$[01]_L$	$Erg_L(01) = L \cup \{00\} \circ \Sigma^*$
$[010]_L$	$Erg_L(010) = L \cup \{0, 100\} \circ \Sigma^*$

Man beachte, dass in $Erg_L(010)$ nicht nur die Wörter enthalten sind, die mit 0 beginnen, sondern auch die, die mit 100 beginnen, weil ja als letztes eine 0 gelesen wurde.

Mit anderen Worten: Man muss immer *alle* Präfixe des Suchmusters 0100 beachten, die man gerade gelesen hat, in diesem Falle nicht nur das Präfix 010, sondern auch das Präfix 0.

Weitere L-Äquivalenzklassen:

$[011]_L = [\varepsilon]_L$	denn $Erg_L(011) = L = Erg_L(\varepsilon)$
	also $\delta([01]_L, 1) = [011]_L = [\varepsilon]_L$
$[0100]_L$	$Erg_L(0100) = \Sigma^*$
$[0101]_L = [01]_L$	denn $Erg_L(0101) = L \cup \{00\} \circ \Sigma^* = Erg_L(01)$
	also $\delta([010]_L, 1) = [0101]_L = [01]_L$
$[0100a]_L = [0100]_L$	denn $Erg_L(0100a) = \Sigma^* = Erg_L(0100)$
	also $\delta([0100]_L, a) = [0100a]_L = [0100]_L$

Damit haben wir alle L-Äquivalenzklassen und die 'nichttrivialen' Übergänge zwischen ihnen bestimmt.

Die 'trivialen' Übergänge sind

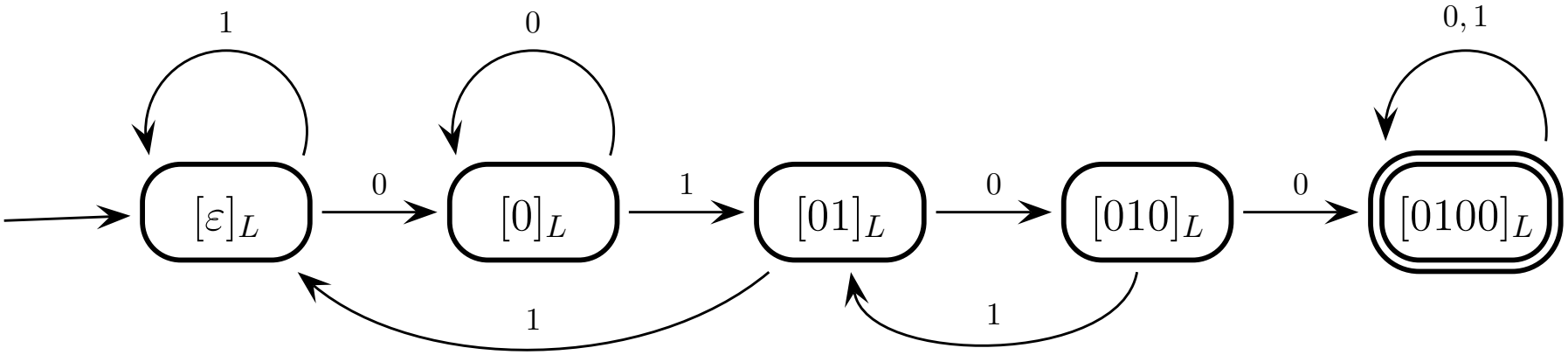
$\delta([\varepsilon]_L, 0) = [0]_L$

$\delta([0]_L, 1) = [01]_L$

$\delta([01]_L, 0) = [010]_L$

$\delta([010]_L, 0) = [0100]_L$

Also sieht der Myhill-Nerode-Automat für L so aus:



Allgemeiner:

Sei $u = a_1 \dots a_n \in \Sigma^*$.

Wir wollen den Myhill-Nerode-Automaten für die Sprache

$$L = L_u = \{w \in \Sigma^* \mid u \text{ ist Teilwort von } w\}$$

konstruieren,
also den, der 'nach dem Wort u sucht'.

Am Beispiel haben wir gesehen, dass die Präfixe des Suchmusters u eine wichtige Rolle spielen.

Sei also $u_i = a_1 \dots a_i$ das Präfix von u mit Länge i (für $i \in \{0, \dots, n\}$) und $v_i = a_{i+1} \dots a_n$ das zugehörige Suffix von u.

Die Äquivalenzklassen $[u_i]_L$ sind paarweise verschieden,
denn für $i < j$ gilt $u_j v_j \in L$ und $u_i v_j \notin L$
($u_i v_j$ ist kürzer als u und kann schon deshalb nicht in L liegen).

Es stellt sich die Frage, ob wir (wie im Beispiel) mit den $[u_i]_L$ bereits *alle* L-Äquivalenzklassen kennen.

Dazu betrachten wir ein beliebiges Wort $w \in \Sigma^*$.

Wenn $w \in L$, so ist $Erg_L(w) = \Sigma^* = Erg_L(u)$, also $[w]_L = [u]_L$.

Wenn $w \notin L$, dann gilt

$$\begin{aligned} Erg_L(w) &= \{v \in \Sigma^* \mid wv \in L\} \\ &= \{v \in \Sigma^* \mid u \text{ ist Teilwort von } wv\} \\ &= \{v \in \Sigma^* \mid u \text{ ist Teilwort von } v \\ &\qquad \text{oder es existiert ein } i \in \{1, \ldots, n-1\} \text{ mit:} \\ &\qquad u_i \text{ ist Suffix von } w \text{ und } v_i \text{ ist Präfix von } v\} \end{aligned}$$

Also ist $Erg_L(w)$ im Falle $w \notin L$ eindeutig bestimmt durch die Menge

$$S(w) = \{u_i \in \{u_0, \ldots, u_{n-1}\} \mid u_i \text{ ist Suffix von } w\}$$

d.h. die Menge aller Präfixe des Suchwortes u, die zugleich Suffixe von w sind.
Man beachte den Spezialfall $S(w) = \{u_0\} = \{\varepsilon\}$: In diesem Fall gilt

$$Erg_L(w) = \{v \in \Sigma^* \mid u \text{ ist Teilwort von } v\} = L = Erg_L(\varepsilon)$$

Sei nun $j \in \{0, \ldots, u_{n-1}\}$ maximal mit $u_j \in S(w)$,
also u_j das *längste* Präfix von u, das Suffix von w ist.
(u_j existiert, denn wegen $u_0 = \varepsilon \in S(w)$ ist $S(w) \neq \emptyset$.)

Dann gilt $S(u_j) = S(w)$, denn:
'$\subseteq$' ist klar, weil die Suffix-Relation transitiv ist.
'$\supseteq$': Wenn u_i ein Suffix von w ist, dann ist es kürzer als das längste Suffix u_j von w, und damit ist es Suffix von u_j.

Also ist (nach den obigen Überlegungen, die ja für jedes Wort w gelten, auch für u_j) $Erg_L(w) = Erg_L(u_j)$ und damit $[w]_L = [u_j]_L$.

Damit ist gezeigt, dass $[u_0]_L, \ldots, [u_n]_L$ die *einzigen* L-Äquivalenzklassen sind, und wir wissen sogar, wie man zu einem Wort $w \in \Sigma^*$ den Index j mit $w \in [u_j]_L$ berechnet.

Also ergibt sich der Myhill-Nerode-Automat für L_u wie folgt:

$A = (\Sigma, Q, s, F, \delta)$ mit

- $Q = \{[u_0]_L, \ldots, [u_n]_L\}$
- $s = [u_0]_L = [\varepsilon]_L$
- $F = \{[u_n]_L\} = \{[u]_L\}$
- $\delta : Q \times \Sigma \to Q$

  $\delta([u_n]_L, a) = \delta([u]_L, a) = [ua]_L = [u]_L = [u_n]_L$ (weil $ua \in L$)

  und für alle $i < n$:

  $\delta([u_i]_L, a) = [u_i a]_L = [u_j]$

  mit $j = \max\{i \in \{0, \ldots, n\} \mid u_j \in S(u_i a)\}$

Wir haben also einen *Algorithmus*, der der zu jedem Wort $u \in \Sigma^*$ einen minimalen 'Suchautomaten' für u liefert, d.h. einen minimalen DEA A_u, der genau die Texte akzeptiert, die das Wort u enthalten.

Auf diesem Algorithmus basieren *Suchprogramme*:

Für jedes Suchwort u wird ein DEA konstruiert, der L_u erkennt.
Dieser DEA wird auf den zu durchsuchenden Text angesetzt.

Lohnt sich der Aufwand?

Ja, weil die Länge m des zu durchsuchenden Textes meist viel größer ist als die Länge n des Suchwortes.
Der Zeitaufwand für die Konstruktion des DEA ist linear in n.
Die Laufzeit des DEA beträgt m.
Also ist die Gesamtlaufzeit nur wenig größer als m.

Zum Vergleich:
Beim 'naiven' Suchalgorithmus testet man für $i = 1, \ldots, m-n+1$, ob das Suchwort u an der Stelle i des Textes w beginnt.

Das erfordert im schlimmsten Fall Laufzeit $n\,(m-n+1)$,
z.B. wenn $u = 0^{n-1}1$ und $w = 0^m$:
An jeder Stelle i von w muss man bis zum letzten Zeichen von u laufen, um zu sehen, dass u nicht passt.

In der Praxis ist man natürlich nicht nur daran interessiert, *ob* das Suchwort im Text vorkommt, sondern auch *wo* es vorkommt. Dazu konstruiert man sich einen DEA A'_u, der die Sprache $L'_u = \{w \in \Sigma^* \mid u$ ist *Suffix* von $w\}$ erkennt, und der immer dann eine Markierung im Suchtext setzt, wenn er gerade im Endzustand ist, d.h. wenn er das Suchwort gerade gefunden hat.
A'_u unterscheidet sich von A_u nur darin, dass er nicht unbedingt im Endzustand bleibt, wenn er das Suchwort u schon gefunden hat, sondern von dort mit jedem Zeichen a in den Zustand $[u_j]$ mit $j = \max\{u_i \in \{u_0, \ldots, u_n\} \mid u_i$ ist Suffix von $ua\}$ übergeht.

Minimalisierung

Bisher:
Konstruktion eines minimalen DEA für L aus den L-Äquivalenzklassen.

Nachteil:
Zur Bestimmung der L-Äquivalenzklassen gibt es keinen Algorithmus, sondern es ist im allgemeinen 'mathematische Argumentation' nötig (die in Spezialfällen wie bei der Sprache L_u einen Algorithmus zur Konstruktion des DEA liefern kann).

Deshalb stellt sich die Frage:
Kann man aus einem beliebigen DEA einen äquivalenten minimalen DEA erhalten?.

Lösungsansatz:
Wenn ein DEA unnötige Unterscheidungen trifft, d.h. wenn L-äquivalente Wörter in verschiedenen Zuständen landen, dann kann man diese Zustände miteinander 'verschmelzen'.

'Verschmelzen' bedeutet aus mathematischer Sicht:
Man fasst die Zustände zu *Äquivalenzklassen* zusammen.

Definition 1.28 *Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA. Zwei Zustände $p, q \in Q$ heißen A-äquivalent (Schreibweise: $p \sim_A q$), wenn für alle $w \in \Sigma^*$ gilt:*

$$\delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$$

Mit anderen Worten:

$$\begin{aligned} p \sim_A q \quad \Leftrightarrow \quad & \text{für alle } w \in \Sigma^* \text{ gilt:} \\ & \text{entweder } \delta^*(p, w), \delta^*(q, w) \in F \\ & \text{oder } \delta^*(p, w), \delta^*(q, w) \notin F \end{aligned}$$

A-Äquivalenz ist eine Äquivalenzrelation auf der Menge Q. Die A-Äquivalenzklasse eines Zustands q bezeichnen wir mit $[q]_A$, also

$$[q]_A = \{p \in Q \mid p \sim_A q\}$$

Lemma 1.29

1. *Wenn* $p \sim_A q$*, dann gilt auch* $\delta^*(p,v) \sim_A \delta^*(q,v)$ *für alle* $v \in \Sigma^*$.
2. *Für jeden Zustand* $q \in Q$ *gilt:* $q \in F \Leftrightarrow [q]_A \subseteq F$
   (Also $F = \bigcup_{q \in F} [q]_A$*.)*

Beweis:

1. Sei $p \sim_A q$ und $v \in \Sigma^*$.

   Dann gilt für alle $v' \in \Sigma^*$:

   $\delta^*(p, vv') \in F \Leftrightarrow \delta^*(q, vv') \in F$,

   also $\delta^*(\delta^*(p,v), v') \in F \Leftrightarrow \delta^*(\delta^*(q,v), v') \in F$,

   und das bedeutet $\delta^*(p,v) \sim_A \delta^*(q,v)$.

2. '$\Leftarrow$' ist klar.

   '$\Rightarrow$': Sei $q \in F$ und $p \sim_A q$.

   Dann ist wegen $\delta(q, \varepsilon) \in F$ auch $p = \delta(p, \varepsilon) \in F$. □

Satz 1.30 *Zu jedem DEA A lässt sich ein äquivalenter minimaler DEA $\bar{A}$ konstruieren. $\bar{A}$ ist isomorph zum Myhill-Nerode-Automaten für die Sprache $L(A)$, d.h. er unterscheidet sich von ihm nur durch eine Umbenennung der Zustände.*

Beweis:

Sei $A = (\Sigma, Q, s, F, \delta)$.

Wir dürfen annehmen, dass A nur erreichbare Zustände besitzt.

Wir definieren $\bar{A} = (\Sigma, \bar{Q}, \bar{s}, \bar{F}, \bar{\delta})$ durch

$\bar{Q} = \{[q]_A \mid q \in Q\}$

$\bar{s} = [s]_A$

$\bar{F} = \{[q]_A \mid q \in F\} \quad (= \{[q]_A \mid [q]_A \subseteq F\})$

$\bar{\delta} : \bar{Q} \times \Sigma \to \bar{Q}$

$\bar{\delta}([q]_A, a) = [\delta(q, a)]_A$

$\bar{\delta}$ ist wohldefiniert, weil aus $p \sim_A q$ mit Lemma 1.29 stets $\delta(p,a) \sim_A \delta(q,a)$ folgt, und durch Induktion über die Länge von w folgt leicht, dass für alle $q \in Q$ und alle $w \in \Sigma^*$ gilt:

$$\bar{\delta}^*([q]_A, w) = [\delta^*(q,w)]_A \qquad (*)$$

Damit folgt:

$$\begin{aligned}
w \in L(\bar{A}) &\Leftrightarrow \bar{\delta}^*([s]_A, w) \in \bar{F} && \text{per Definition von } L(\bar{A}) \\
&\Leftrightarrow [\delta^*(s,w)]_A \in \bar{F} && \text{wegen } (*) \\
&\Leftrightarrow [\delta^*(s,w)]_A \subseteq F && \text{per Definition von } \bar{F} \\
&\Leftrightarrow \delta^*(s,w) \in F && \text{nach Lemma 1.29} \\
&\Leftrightarrow w \in L(A) && \text{per Definition von } L(A)
\end{aligned}$$

Also ist $L(\bar{A}) = L(A)$, d.h. $\bar{A}$ ist äquivalent zu A.

Außerdem gilt für alle $u_1, u_2 \in \Sigma^*$:

$$
\begin{aligned}
u_1 \sim_{L(A)} u_2 \quad &\Leftrightarrow \quad \text{für alle } w \in \Sigma^* \text{gilt} \\
&\qquad u_1 w \in L(A) \Leftrightarrow u_2 w \in L(A) \\
&\Leftrightarrow \quad \text{für alle } w \in \Sigma^* \text{gilt} \\
&\qquad \delta^*(s, u_1 w) \in F \Leftrightarrow \delta^*(s, u_2 w) \in F \\
&\Leftrightarrow \quad \text{für alle } w \in \Sigma^* \text{gilt} \\
&\qquad \delta^*(\delta^*(s, u_1), w) \in F \Leftrightarrow \delta^*(\delta^*(s, u_2), w) \in F \\
&\Leftrightarrow \quad \delta^*(s, u_1) \sim_A \delta^*(s, u_2) \\
&\Leftrightarrow \quad \overline{\delta}^*([s]_A, u_1) = \overline{\delta}^*([s]_A, u_2)
\end{aligned}
$$

wobei die letzte Umformung wieder wegen (*) gilt.

Zwei Wörter u_1, u_2 sind also genau dann $L(A)$-äquivalent, wenn sie in $\bar{A}$ zum gleichen Zustand führen.

Das bedeutet, dass die Anzahl der erreichbaren Zustände von $\bar{A}$ nicht größer ist als die Anzahl der $L(A)$-Äquivalenzklassen, d.h. die Anzahl der Zustände des Myhill-Nerode-Automaten.

Und weil $\bar{A}$—wie der ursprüngliche Automat A—nur erreichbare Zustände hat, folgt daraus, dass er die kleinstmögliche Anzahl von Zuständen hat, d.h. $\bar{A}$ ist minimal.

Die folgende genauere Argumentation zeigt, dass er sogar zum Myhill-Nerode-Automaten isomorph ist.

Sei $\tilde{A} = (\Sigma, \tilde{Q}, \tilde{s}, \tilde{F}, \tilde{\delta})$ der Myhill-Nerode-Automat für $L(A)$.

Wir definieren eine Abbildung

$$\Phi : \tilde{Q} \rightarrow \bar{Q}$$

$$\Phi([w]_{L(A)}) = \bar{\delta}^*([s]_A, w)$$

Nach den obigen Betrachtungen über die $L(A)$-Äquivalenzklassen ist Φ wohldefiniert und injektiv, und weil $\bar{A}$ nur erreichbare Zustände hat, ist Φ auch surjektiv.

Darüber hinaus gilt:

- $\Phi(\tilde{s}) = \Phi([\varepsilon]_{L(A)}) = \bar{\delta}^*([s]_A, \varepsilon) = [s]_A$
- $\Phi([w]_{L(A)}) \in \bar{F} \Leftrightarrow \bar{\delta}^*([s]_A, w) \in \bar{F} \Leftrightarrow w \in L(\bar{A})$
  $\Leftrightarrow w \in L(A) \Leftrightarrow [w]_{L(A)} \in \tilde{F}$
- $\Phi(\tilde{\delta}([w]_{L(A)}, a)) = \Phi([wa]_{L(A)}) = \bar{\delta}^*([s]_A, wa)$
  $= \bar{\delta}(\bar{\delta}^*([s]_A, w), a) = \bar{\delta}(\Phi([w]_{L(A)}), a)$

Also ist Φ eine bijektive Abbildung, unter der sich die Start- und Endzustände und die Zustandsübergänge in den beiden Automaten entsprechen.

Das bedeutet, dass Φ ein Isomorphismus ist, d.h. eine reine Umbenennung der Zustände.

Es bleibt zu zeigen, dass sich der DEA $\bar{A}$ aus dem DEA A *konstruieren* lässt, d.h. dass sich die A-Äquivalenzklassen bestimmen lassen (alles andere ist klar).

Dazu definieren wir Relationen $\sim_n$ auf Q durch Induktion über n:

$$\begin{aligned} p \sim_0 q &\Leftrightarrow (p \in F \Leftrightarrow q \in F) \\ &\Leftrightarrow \text{entweder } p, q \in F \text{ oder } p, q \notin F \\ p \sim_{n+1} q &\Leftrightarrow p \sim_n q \text{ und für alle } a \in \Sigma \text{ gilt } \delta(p,a) \sim_n \delta(q,a) \end{aligned}$$

Durch Induktion über n folgt leicht, dass für jedes n gilt:

$$\begin{aligned} p \sim_n q \Leftrightarrow\ & \text{für alle } w \in \Sigma^* \text{ mit } |w| \leq n \text{ gilt} \\ & \delta^*(p,w) \in F \Leftrightarrow \delta^*(q,w) \in F \end{aligned}$$

Damit ist klar, dass jedes $\sim_n$ eine Äquivalenzrelation ist, und dass $\sim_A$ der Durchschnitt aller $\sim_n$ ist.

Also bleibt nur noch zu zeigen, dass sich $\bigcap_{n\geq 0} \sim_n$ berechnen lässt.

Zunächst beachte man, dass (per Definition) $\sim_{n+1} \subseteq \sim_n$ für alle $n \geq 0$ gilt, d.h. die $\sim_n$ bilden eine absteigende Folge $\sim_0 \supseteq \sim_1 \supseteq \ldots$ von Relationen auf Q, also eine absteigende Folge von Teilmengen von $Q \times Q$.

Da $Q \times Q$ endlich ist, kann eine solche Folge *nicht echt absteigend* sein, also gibt es eine Zahl m mit $\sim_m = \sim_{m+1}$.

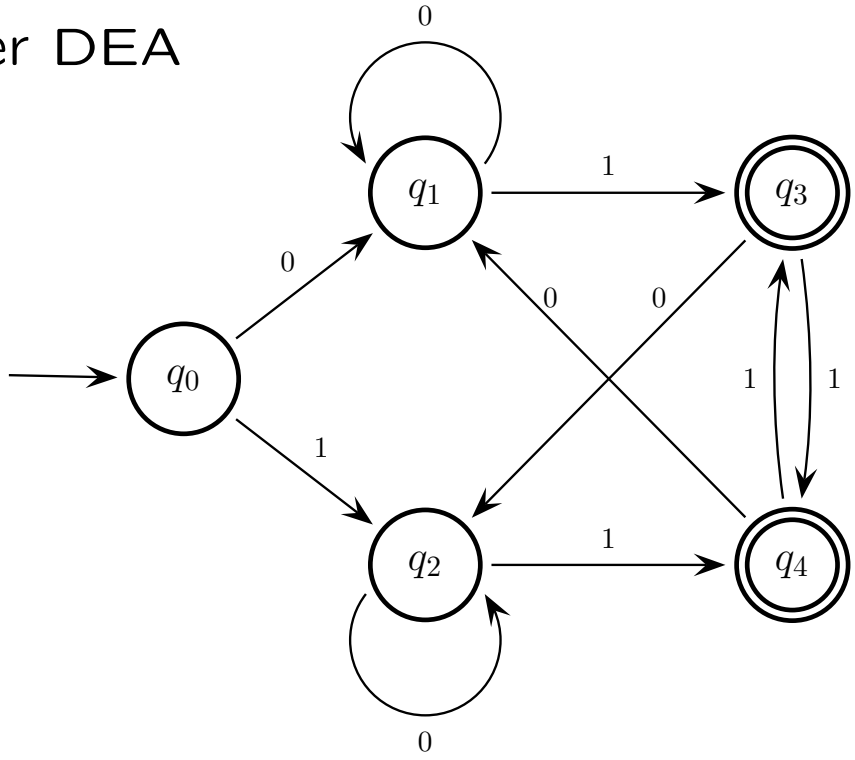
Weil sich $\sim_{m+2}$ wieder auf die gleiche Art aus $\sim_{m+1}$ ergibt wie $\sim_{m+1}$ aus $\sim_m$, folgt dann $\sim_m = \sim_n$ für alle $n \geq m$,

und das bedeutet, dass $\sim_m = \bigcap_{n\geq 0} \sim_n = \sim_A$.

Damit haben wir einen Algorithmus zur Berechnung von $\sim_A$ gefunden: Wir berechnen die Relationen $\sim_0, \sim_1, \ldots$ bis wir das erste m finden mit $\sim_m = \sim_{m+1}$.

Für dieses m gilt dann $\sim_m = \sim_A$. □

Beispiel: Sei A der DEA



$\sim_0$ hat die Äquivalenzklassen $Q \setminus F = \{q_0, q_1, q_2\}$ und $F = \{q_3, q_4\}$.

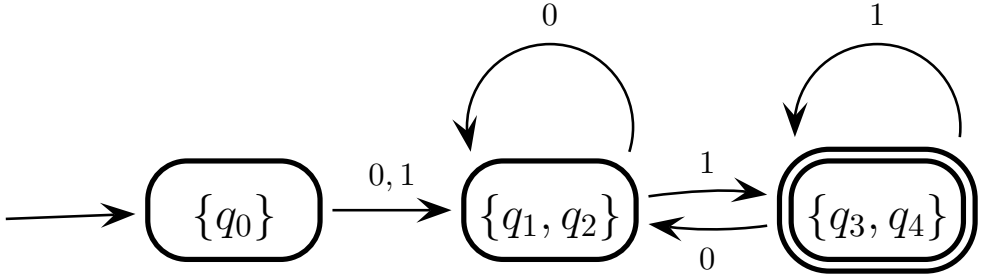
1. Verfeinerungsschritt:

$\delta(q_0, 0) = \delta(q_1, 0) = q_1$ und $\delta(q_2, 0) = q_2$ liegen in $Q \setminus F$,
aber $\delta(q_0, 1) = q_2 \in Q \setminus F$ und $\delta(q_1, 1) = q_3, \delta(q_2, 1) = q_4$ liegen in F.
Also zerfällt $\{q_0, q_1, q_2\}$ in zwei $\sim_1$-Klassen $\{q_0\}$ und $\{q_1, q_2\}$.

$\delta(q_3, 0) = q_2$ und $\delta(q_4, 0) = q_1$ liegen in $Q \setminus F$,
$\delta(q_3, 1) = q_4$ und $\delta(q_4, 1) = q_3$ liegen in F.
Also bleibt die $\sim_0$-Klasse $\{q_3, q_4\}$ als $\sim_1$-Klasse erhalten.
Damit besitzt $\sim_1$ die drei Äquivalenzklassen $\{q_0\}, \{q_1, q_2\}$ und $\{q_3, q_4\}$.

2. Verfeinerungsschritt:

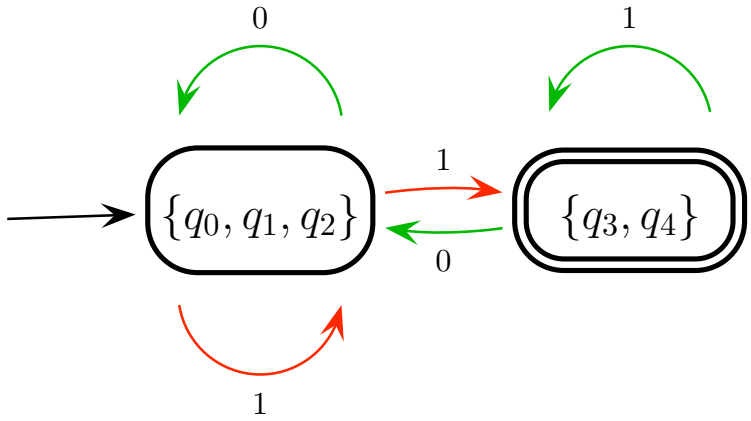
Die $\sim_1$-Klasse $\{q_0\}$ ist nicht mehr weiter aufteilbar.
$\delta(q_1, 0), \delta(q_2, 0) \in \{q_1, q_2\}$ und $\delta(q_1, 1), \delta(q_2, 1) \in \{q_3, q_4\}$.
Also bleibt die $\sim_1$-Klasse $\{q_1, q_2\}$ als $\sim_2$-Klasse erhalten.
$\delta(q_3, 0), \delta(q_4, 0) \in \{q_1, q_2\}$ und $\delta(q_3, 1), \delta(q_4, 1) \in \{q_3, q_4\}$.
Also bleibt auch die $\sim_1$-Klasse $\{q_1, q_2\}$ als $\sim_2$-Klasse erhalten.
Damit ist $\sim_1 = \sim_2$ gezeigt, also ist $\sim_2 = \sim_A$,
und man erhält den folgenden zu A äquivalenten minimalen DEA:



Eine anschauliche Erklärung für die Relationen $\sim_n$:

Man kann jede dieser Äquivalenzrelationen als 'Versuch' auffassen, den minimalen DEA zu konstruieren:
Man nimmt die Äquivalenzklassen von $\sim_n$ als Zustände, und versucht die Zustandsübergänge zwischen ihnen richtig festzulegen.
Scheitert der Versuch, so sieht man daran, wie die Äquivalenzklassen weiter aufgeteilt werden müssen. Gelingt der Versuch, so hat man den minimalen DEA gefunden (weil man mit $\sim_0$, d.h. mit der kleinstmöglichen Anzahl von Zuständen begonnen und immer nur die notwendigen Verfeinerungen vorgenommen hat).

An unserem Beispiel sieht das so aus:
Man versucht zunächst, einen DEA zu konstruieren, der nur die beiden Äquivalenzklassen $\{q_0, q_1, q_2\}$ und $\{q_3, q_4\}$ von $\sim_0$ als Zustände hat:



Das geht gut für die Übergänge

$\delta(q_0, 0), \delta(q_1, 0), \delta(q_2, 0) \in \{q_0, q_1, q_2\}$,

$\delta(q_3, 0), \delta(q_4, 0) \in \{q_0, q_1, q_2\}$,

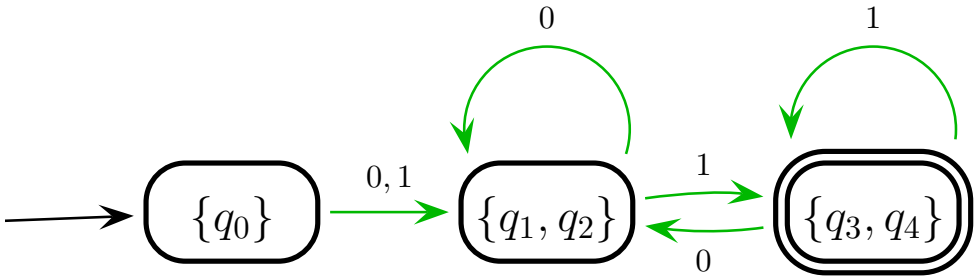
und $\delta(q_3, 1), \delta(q_4, 1) \in \{q_3, q_4\}$,

scheitert aber an den sich widersprechenden Übergängen

$\delta(q_0, 1) \in \{q_0, q_1, q_2\}$ und $\delta(q_1, 1), \delta(q_2, 1) \in \{q_3, q_4\}$.

An den gescheiterten Übergängen sieht man, dass $\{q_0, q_1, q_2\}$ in $\{q_0\}$ und $\{q_1, q_2\}$ aufgeteilt werden muss.

Also versucht man jetzt, einen DEA zu konstruieren, der die drei Äquivalenzklassen $\{q_0\}$, $\{q_1, q_2\}$ und $\{q_3, q_4\}$ von $\sim_1$ als Zustände hat:



Jetzt geht mit den Übergängen alles gut:

$\delta(q_0, 0) \in \{q_1, q_2\}$ $\quad$ $\delta(q_0, 1) \in \{q_1, q_2\}$

$\delta(q_1, 0), \delta(q_2, 0) \in \{q_1, q_2\}$ $\quad$ $\delta(q_1, 1), \delta(q_2, 1) \in \{q_3, q_4\}$

$\delta(q_3, 0), \delta(q_4, 0) \in \{q_1, q_2\}$ $\quad$ $\delta(q_3, 1), \delta(q_4, 1) \in \{q_3, q_4\}$

Also ist der minimale DEA gefunden.

Das Pumping Lemma (dt.: **Schleifensatz**) bietet eine weitere Methode, um zu zeigen, dass eine Sprache *nicht* regulär ist.

Es beschreibt eine *Eigenschaft* regulärer Sprachen. Um zu beweisen, dass eine Sprache *nicht* regulär ist, genügt es zu zeigen, dass sie diese Eigenschaft nicht besitzt.

Mit anderen Worten:
Das Pumping Lemma gibt eine *notwendige* Bedingung dafür an, dass eine Sprache regulär ist. Kann man zeigen, dass diese Bedingung nicht erfüllt ist, so weiß man, dass die Sprache nicht regulär ist.

Zum Vergleich:
Der Satz von Myhill und Nerode gibt eine *hinreichende und notwendige* Bedingung dafür an, dass eine Sprache regulär ist. Deshalb kann er für beide Zwecke benutzt werden: Sowohl zum Nachweis, dass eine Sprache regulär ist, als auch dazu, dass sie nicht regulär ist.

Die Idee:

Sei A ein DEA für die reguläre Sprache L, und sei n die Anzahl der Zustände von A.

Sei $x \in L$ mit $|x| \geq n$.

Dann werden $n+1$ Zustände bei der Abarbeitung von x auf A durchlaufen. Da A nur n Zustände hat, wird also mindestens ein Zustand mehrmals durchlaufen, d.h. der Weg, auf dem x abgearbeitet wird, enthält eine *Schleife*.

Durchläuft man diese Schleife öfter oder weniger oft als im Wort x, so erhält man andere Wörter, die zum gleichen Endzustand führen wie x, die also ebenfalls in L liegen.

Wie sehen diese Wörter aus?

Sei $x = uvw$, wobei $v \neq \varepsilon$ das Teilwort ist, das in der Schleife abgearbeitet wird. Dann kann man v beliebig oft wiederholen, und erhält damit Wörter der Form $x_i = uv^iw$ $(i \geq 0)$, die alle in L liegen.
Das Wort x_i entsteht sozusagen durch *Aufpumpen* des Wortes w an der Stelle v mit "Pumpfaktor" i.

Satz 1.31 (Pumping Lemma) *Sei $L \subseteq \Sigma^*$ regulär. Dann existiert eine Zahl $n \geq 1$, so dass für jedes $x \in L$ mit $|x| \geq n$ gilt: Es gibt eine Zerlegung $x = uvw$ mit $v \neq \varepsilon$ und $uv^iw \in L$ für alle $i \geq 0$.*

Intuition:
Wenn L regulär ist, dann kann man jedes hinreichend lange Wort $x \in L$ an mindestens einer Stelle v mit beliebigem Faktor i aufpumpen, ohne dass man die Sprache L verlässt.

Beweis:

Sei $A = (\Sigma, Q, s, F, \delta)$ ein DEA, der L erkennt.

Wir wählen $n = |Q|$.

Wenn $x \in L$ mit $|x| \geq n$, dann wird bei der Abarbeitung von x mindestens ein Zustand mehrmals besucht,

d.h. es existieren $p \in Q, q \in F$ und $u, v, w \in \Sigma^*$ mit $x = uvw$ und

$$(s, x) = (s, uvw) \vdash_A^* (p, vw) \vdash_A^+ (p, w) \vdash_A^* (q, \varepsilon)$$

Dabei ist $v \neq \varepsilon$, weil in $\vdash_A^+$ mindestens ein Zeichen verarbeitet wird.

Indem man die $\vdash_A^+$-Schleife i-mal wiederholt, folgt

$$(s, uv^i w) \vdash_A^* (p, v^i w) \vdash_A^+ \ldots \vdash_A^+ (p, w) \vdash_A^* (q, \varepsilon)$$

also $uv^i w \in L$ für jedes $i \geq 0$. □

Korollar 1.32 *Zum Nachweis, dass L nicht regulär ist, genügt es zu zeigen: Für jedes $n \geq 1$ existiert ein $x \in L$ mit $|x| \geq n$, so dass für alle Zerlegungen $x = uvw$ mit $v \neq \varepsilon$ gilt: Es gibt ein $i \geq 0$ mit $uv^iw \notin L$.*

Intuition:
Es ist zu zeigen, dass *beliebig lange* Wörter $x \in L$ existieren, die sich an *jeder* Stelle v so aufpumpen lassen, dass das Ergebnis nicht mehr in L liegt.

Der Beweis, dass eine Sprache L nicht regulär ist, läuft also so ab:
Für *jedes* $n \geq 1$ gibt man ein *passendes* Wort x mit $|x| \geq n$ an. Dann wählt man für *jede* Zerlegung $x = uvw$ mit $v \neq \varepsilon$ einen *passenden* Pumpfaktor i mit $uv^iw \notin L$.

Merke:
x und i darf man *passend* wählen, aber man muss *alle* Zerlegungen betrachten. Oft ist dabei eine Fallunterscheidung notwendig, d.h. man muss *alle* Möglichkeiten für die Position des Wortes v durchspielen.

Beispiel:

$L_1 = \{a^n b^m \mid 0 \leq n \leq m\}$ ist nicht regulär, denn:

Sei $n \geq 1$.

Man wähle $x = a^n b^n$, also $|x| = 2n \geq n$.

Sei $x = uvw$ eine Zerlegung von x mit $v \neq \varepsilon$.

1. Fall: v liegt in a^n, d.h. $v = a^k$ für ein $k \geq 1$.

Dann gilt $uv^2w = a^{n+k}b^n \notin L_1$, weil $n + k > n$ wegen $k \geq 1$.

2. Fall: v liegt an der Schnittstelle, d.h. $v = a^i b^j$ mit $i, j \geq 1$.

Dann gilt $uv^2w = a^{n-i}a^i b^j a^i b^j b^{n-j} = a^n b^j a^i b^n \notin L_1$, denn wegen $i, j \geq 1$ stehen as hinter bs.

3. Fall: v liegt in b^m, d.h. $v = b^k$ für ein $k \geq 1$.

Dann gilt $uv^0w = a^n b^{n-k} \notin L_1$, weil $n - k < n$ wegen $k \geq 1$.

Am Beispiel wird klar:
Die Schwierigkeit bei der Anwendung des Pumping Lemmas besteht darin, dass man nicht weiß, wo das Teilwort v liegt. Das kann zu mühsamen Fallunterscheidungen führen oder ganz schiefgehen:

Beispiel:

$L_2 = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$.

Dann gibt es für *jedes* $x \in L_2$ mit $|x| \geq 1$ eine Zerlegung $x = uvw$ mit $v \neq \varepsilon$, so dass $uv^iw \in L$ für *alle* $i \geq 0$.

Nämlich $u = w = \varepsilon$ und $v = x$.

Dann ist $uv^iw = x^i \in L$ für alle $i \geq 0$.

Also kann man gar kein passendes Wort x finden, um mit Satz 1.31 nachzuweisen, dass L_2 nicht regulär ist.

Mit anderen Worten:

L_2 erfüllt die im Pumping Lemma genannte Eigenschaft regulärer Sprachen, ohne selbst regulär zu sein. Also ist diese Eigenschaft eine notwendige, aber keine hinreichende Bedingung dafür, dass die Sprache regulär ist.

Die meisten Beispiele lassen sich bewältigen, wenn man das Pumping Lemma wie folgt verstärkt.

Satz 1.33 (Starkes Pumping Lemma) *Sei $L \subseteq \Sigma^*$ regulär. Dann existiert eine Zahl $n \geq 1$, so dass für jedes $x \in L$ mit $|x| \geq n$ gilt: Es gibt eine Zerlegung $x = uvw$ mit $v \neq \varepsilon$, $|uv| \leq n$ und $uv^iw \in L$ für alle $i \geq 0$.*

Man erhält diese Aussage, indem man den Beweis von Satz 1.31 leicht abändert: Anstelle des Wortes x betrachtet man das Präfix y von x der Länge n. Schon bei der Abarbeitung von y muss mindestens ein Zustand mehrmals auftauchen, also können wir das Wort v so wählen, dass es in y liegt, und das bedeutet $|uv| \leq n$.

Das starke Pumping Lemma eröffnet neue Möglichkeiten, weil wir nicht mehr *alle* Zerlegungen betrachten müssen, sondern nur noch die mit $|uv| \leq n$.

Damit vereinfacht sich z.B. der Beweis für $L_1 = \{a^n b^m \mid 0 \leq n \leq m\}$: Man wählt nach wie vor $x = a^n b^n$. Aber jetzt braucht man nur noch Zerlegungen der Form $x = uvw$ mit $|uv| \leq n$ zu untersuchen. $|uv| \leq n$ bedeutet aber, dass v ganz in a^n liegen muss, also bleibt von den drei Fällen im obigen Beweis nur der erste übrig.

Der Beweis für $L_2 = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$ gelingt jetzt auch:

Sei $n \geq 1$.

Man wähle $x = a^n b^n$, also $x \in L_2$ und $|x| = 2n \geq n$.

Sei $x = uvw$ eine Zerlegung mit $v \neq \varepsilon$ und $|uv| \leq n$.

Dann liegt v in a^n, also $v = a^k$ für ein $k \geq 1$.

Also ist $uv^2w = a^{n+k}b^n \notin L_2$, weil $n + k \neq n$ wegen $k \geq 1$.

Weiteres Beispiel:

Sei $L_3 = \{a^p \mid p \text{ ist Primzahl}\}$.
Wir haben schon mit dem Satz von Myhill und Nerode bewiesen, dass diese Sprache nicht regulär ist, aber mit dem Pumping Lemma geht es wesentlich einfacher:

Sei $n \geq 1$.
Man wählt $x = a^p$ für eine Primzahl $p \geq n$, also ist $x \in L_3$ und $|x| \geq n$.
Sei $x = uvw$ eine Zerlegung mit $v \neq \varepsilon$, d.h. $v = a^k$ mit $k \geq 1$.
Dann gilt $uv^{p+1}w = a^{p+kp} = a^{p(1+k)} \notin L_3$, denn wegen $1 + k \geq 2$ ist p ein echter Teiler von $p(1 + k)$, also $p(1 + k)$ keine Primzahl.

Übrigens ist auch die im starken Pumping Lemma genannte Eigenschaft noch keine hinreichende Bedingung für die Regularität einer Sprache, aber in den meisten Fällen reicht es aus, um zu zeigen, dass eine Sprache nicht regulär ist.

Andere Verstärkungen des Pumping Lemmas sind denkbar, z.B. kann man $|uv| \leq n$ durch $|vw| \leq n$ ersetzen.

Offene Frage:

Kann man das Pumping Lemma so verstärken, dass es eine hinreichende und notwendige Bedingung für die Regularität der Sprache liefert?

Kontextfreie Sprachen und Kellerautomaten

- Reguläre Sprachen haben eine sehr einfache Struktur.

  Deshalb kann man mit ihnen nur die kleinsten (d.h. einfachsten) syntaktischen Einheiten einer Programmiersprache erfassen,

  z.B. Schlüsselwörter, Zahldarstellungen, Bezeichner, Kommentare.

- Wie beschreibt man die komplizierteren syntaktischen Strukturen,

  z.B. arithmetische und boolesche Ausdrücke, Anweisungen, Deklarationen?

  Am besten durch eine *Konstruktionsvorschrift*,

  d.h. aus mathematischer Sicht: durch eine *induktive Definition*.

Beispiel: Vollständig geklammerte arithmetische Ausdrücke

Konstruktionsvorschrift

1. Jede Dezimalzahl ist ein arithmetischer Ausdruck.
2. Jeder Bezeichner ist ein arithmetischer Ausdruck.
3. Wenn e ein arithmetischer Ausdruck ist, dann ist auch $(-e)$ ein arithmetischer Ausdruck.
4. Wenn e_1, e_2 arithmetische Ausdrücke sind, dann ist auch $(e_1 + e_2)$ ein arithmetischer Ausdruck.
5. Wenn e_1, e_2 arithmetische Ausdrücke sind, dann ist auch $(e_1 - e_2)$ ein arithmetischer Ausdruck.
6. Wenn e_1, e_2 arithmetische Ausdrücke sind, dann ist auch $(e_1 * e_2)$ ein arithmetischer Ausdruck.

Mathematische Formulierung

Die Sprache L der vollständig geklammerten arithmetischen Ausdrücke ist *induktiv definiert* durch:

1. Jede Dezimalzahl liegt in L.
2. Jeder Bezeichner liegt in L.
3. Wenn $e \in L$, dann ist auch $(-e) \in L$.
4. Wenn $e_1, e_2 \in L$, dann ist auch $(e_1 + e_2) \in L$.
5. Wenn $e_1, e_2 \in L$, dann ist auch $(e_1 - e_2) \in L$.
6. Wenn $e_1, e_2 \in L$, dann ist auch $(e_1 * e_2) \in L$.

Was bedeutet '*induktiv definiert*'? Es bedeutet, dass L die *kleinste* Menge mit den Eigenschaften 1. bis 6. ist (eine andere Menge mit diesen Eigenschaften ist z.B. Σ^*).

Die mathematische Formulierung genügt uns nicht (so wie uns die Mengenausdrücke nicht genügten). Wir brauchen eine *formale Schreibweise* (wie die regulären Ausdrücke).

Definition 2.1 *Eine kontextfreie Grammatik (kurz: KFG) ist ein 4-Tupel $G = (\Sigma, N, S, P)$, wobei gilt:*

- *Σ und N sind Alphabete mit $\Sigma \cap N = \emptyset$. Die Zeichen aus Σ heißen Terminalzeichen, die aus N heißen Nichtterminalzeichen von G.*
- *$S \in N$. S heißt Startzeichen von G.*
- *$P \subseteq N \times (\Sigma \cup N)^*$. Die Elemente von P heißen Regeln oder Produktionen von G.*

Eine Produktion ist also ein *Paar* (A, u), wobei A ein Nichtterminalzeichen ist und u ein Wort, das sowohl Terminal- als auch Nichtterminalzeichen enthalten darf. Statt (A, u) schreibt man $A \to u$.

Konvention:
Als Nichtterminalzeichen benutzen wir Großbuchstaben.

Beispiel:

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

- $\Sigma = \{0, 1, x, y, -, +, *, (,)\}$
- $N = \{E\}$
- $S = E$
- $P = \{E \to 0,\ E \to 1,\ E \to x,\ E \to y,\ E \to (-E),$
  $E \to (E + E),\ E \to (E - E),\ E \to (E * E)\}$

ist eine KFG für vollständig geklammerte arithmetische Ausdrücke (in denen nur die Zahlen $0, 1$ und nur die Bezeichner x, y vorkommen).

Kurzschreibweise:

Man schreibt

$$A \to u_1 \mid \ldots \mid u_n$$

als Abkürzung für eine *Menge* von Produktionen

$$A \to u_1, \ldots, A \to u_n$$

mit dem gleichen Nichtterminalzeichen auf der linken Seite.

Im Beispiel kann man also *alle* Produktionen zusammenfassen zu

$$E \to 0 \mid 1 \mid x \mid y \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

wobei man "|" als "oder" liest.

Wie ist eine KFG zu verstehen, d.h. welche Sprache beschreibt sie?

- Entweder als *Konstruktionsvorschrift* für eine Sprache:

  Dann dienen die Produktionen dazu, die Wörter der Sprache *abzuleiten* oder zu *erzeugen*.

- Oder als *induktive Definition* einer Sprache:

  Dann liest man die Produktionen als *Regeln*, die die Sprache erfüllen muss.

Beide Auffassungen sind äquivalent zueinander, üblich ist aber die Formulierung als Konstruktionsvorschrift.

Definition 2.2 *Sei $G = (\Sigma, N, S, P)$ eine KFG.*

Auf der Menge $(\Sigma \cup N)^*$ definieren wir eine Relation $\Rightarrow_G$ (oder kürzer: $\Rightarrow$) durch:

$$u \Rightarrow_G v \quad \Leftrightarrow \quad \text{es gibt eine Produktion } A \to y \text{ in } P$$
$$\text{und Wörter } x, z \in (\Sigma \cup N)^*$$
$$\text{so dass } u = xAz \text{ und } v = xyz$$

Man bezeichnet $u \Rightarrow_G v$ als einen Ableitungsschritt (in G).

In Worten: Ein Ableitungsschritt $u \Rightarrow_G v$ besteht darin, ein Vorkommen eines Nichtterminalzeichens A im Wort u durch die rechte Seite y einer Produktion $A \to y$ zu ersetzen.

Mit $\overset{n}{\Rightarrow}_G$ $(n \geq 0)$, $\overset{+}{\Rightarrow}_G$ und $\overset{*}{\Rightarrow}_G$ bezeichnen wir wieder die n-te Potenz, den transitiven Abschluss und den reflexiven, transitiven Abschluss der Relation $\Rightarrow_G$.

Definition 2.3 *Sei G eine KFG.*

1. *v heißt ableitbar aus u (in G), wenn $u \overset{*}{\Rightarrow}_G v$, d.h. wenn eine Folge $u = w_0 \Rightarrow_G \ldots \Rightarrow_G w_n = v$ $(n \geq 0)$ von Ableitungsschritten in G existiert.*

   Eine solche Folge bezeichnet man als Ableitung von v aus u.

2. *Die von G erzeugte Sprache $L(G)$ ist definiert durch*

$$L(G) \;=\; \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

   $L(G)$ besteht also aus allen Wörtern über dem Terminalalphabet Σ, die aus dem Startzeichen S ableitbar sind.

Definition 2.4 *Eine Sprache $L \subseteq \Sigma^*$ heißt* kontextfrei, *wenn es eine kontextfreie Grammatik G gibt mit $L = L(G)$.*

Beispiel:

Sei G die oben definierte KFG für arithmetische Ausdrücke.

Dann gilt z.B.
$E \Rightarrow (E+E) \Rightarrow ((E*E)+E) \Rightarrow ((x*E)+E) \Rightarrow ((x*y)+E) \Rightarrow ((x*y)+1)$
also $((x * y) + 1) \in L(G)$.

Die Sprache $L(G)$ ist (per Definition) kontextfrei.

Aber ist $L(G)$ die gewünschte Sprache L der vollständig geklammerten arithmetischen Ausdrücke?
Um diese Frage zu beantworten, bräuchte man eine Definition für L, die unabhängig von der Grammatik ist. Die ist schwer zu finden!
Also stellen wir uns auf den Standpunkt, dass L durch die Grammatik G *definiert* ist. Dann ist nichts mehr zu beweisen.

Sei G wie oben.

Neben der bereits genannten Ableitung
$E \Rightarrow (E+E) \Rightarrow ((E*E)+E) \Rightarrow ((x*E)+E) \Rightarrow ((x*y)+E) \Rightarrow ((x*y)+1)$
gibt es andere Ableitungen für das gleiche Wort, z.B.
$E \Rightarrow (E+E) \Rightarrow (E+1) \Rightarrow ((E*E)+1) \Rightarrow ((E*y)+1) \Rightarrow ((x*y)+1)$

In beiden Ableitungen werden die 'gleichen' Ableitungsschritte in unterschiedlicher Reihenfolge benutzt.

Die Reihenfolge der Ableitungsschritte spielt keine Rolle, weil die Ableitungsschritte in einer kontextfreien Grammatik sehr einfach aussehen: Es wird stets ein Nichtterminalzeichen A durch ein Wort y ersetzt. Der *Kontext* des Zeichens A, d.h. der Text vor und hinter A, spielt dabei keine Rolle. Er beeinflusst den Ableitungsschritt nicht, und er wird durch den Ableitungsschritt nicht verändert (daher die Bezeichnung '*kontextfrei*'). Also ist es unerheblich, ob man zuerst A oder zuerst ein Nichtterminalzeichen im Kontext von A ersetzt.

Diese Überlegungen kann man präzisieren,

- entweder indem man eine spezielle Reihenfolge für die Ableitungsschritte *vorschreibt*, und dann zeigt, dass jede Ableitung so umgeformt werden kann, dass sie der Vorschrift genügt,
- oder indem man eine Schreibweise benutzt, die von der Reihenfolge der Ableitungsschritte *abstrahiert*.

Definition 2.5 *Ein Ableitungsschritt* $u \Rightarrow v$ *heißt* Linksableitungsschritt, *wenn es Wörter* $x \in \Sigma^*$, $z \in (N \cup \Sigma)^*$ *und eine Produktion* $A \to y$ *in* G *gibt mit* $u = xAz$ *und* $v = xyz$. *Eine* Linksableitung *ist eine Ableitung, die nur aus Linksableitungsschritten besteht.*

Eine Linksableitung ist also eine Ableitung, bei der stets das erste, d.h. das am weitesten links stehende Nichtterminalzeichen ersetzt wird.

Ein Beispiel für eine Linksableitung haben wir schon gesehen:

$E \Rightarrow (E+E) \Rightarrow ((E*E)+E) \Rightarrow ((x*E)+E) \Rightarrow ((x*y)+E) \Rightarrow ((x*y)+1)$

Das folgende Lemma besagt, dass Linksableitungen ausreichen.

Lemma 2.6 *$u \stackrel{*}{\Rightarrow}_G v$ gilt genau dann, wenn es eine Linksableitung von v aus u in G gibt.*

Beweisidee:

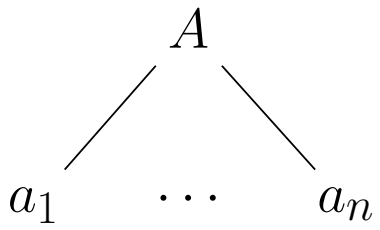
Man kann die Schritte einer beliebigen Ableitung so umordnen, dass eine Linksableitung entsteht. Die Details sind mühsam! □

Man kann Lemma 2.6 so interpretieren:

Wenn wir beliebige Ableitungen betrachten, so gibt es (fast in jeder Grammatik) viele Ableitungen, die sich nur 'unwesentlich' unterscheiden, nämlich nur in der Reihenfolge der Ersetzungen. Betrachten wir nur Linksableitungen, so entfallen diese unwesentlichen Unterschiede, weil die Reihenfolge der Ableitungsschritte fest vorgeschrieben ist.

Ein alternativer Ansatz besteht darin, von der Reihenfolge der Ableitungsschritte zu abstrahieren, indem man *Ableitungsbäume* betrachtet.

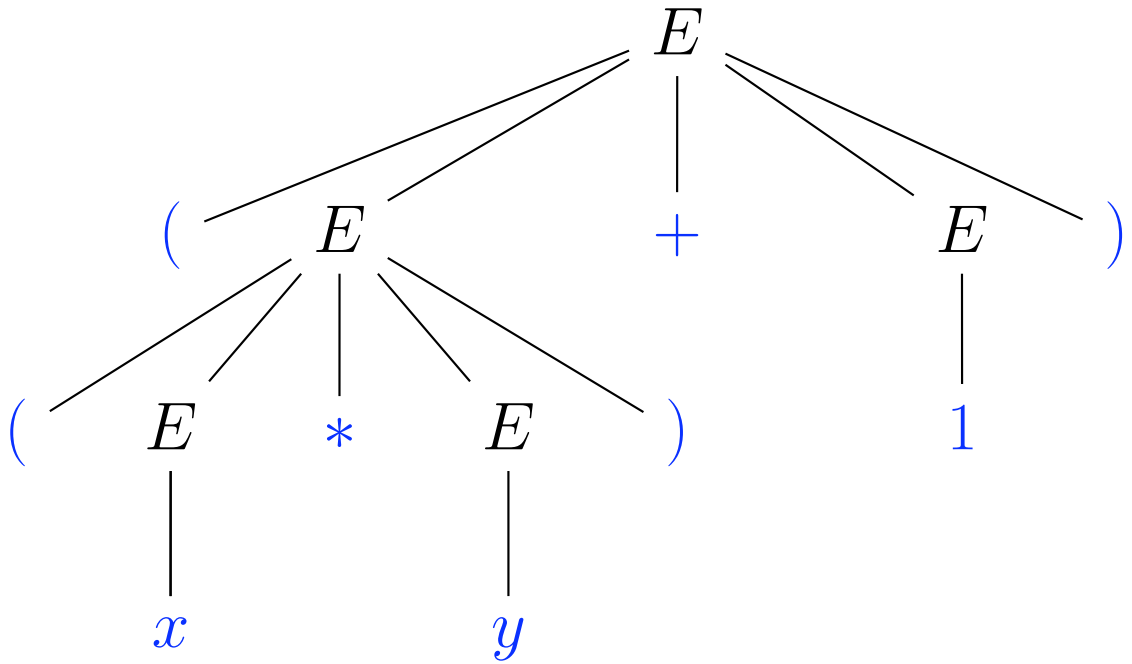
Definition 2.7 *Ein* Ableitungsbaum *(oder* Syntaxbaum*) für die Grammatik* $G = (\Sigma, N, S, P)$ *ist ein Baum, dessen Knoten mit Zeichen aus* $\Sigma \cup N$ *markiert sind, und zwar so, dass jeder innere Knoten die Form*



hat, wobei $A \to a_1 \ldots a_n$ *eine Produktion in* P *ist.*

Beispiel:

Ein Ableitungsbaum für unsere Grammatik G ist



An diesem Baum sieht man, dass $((x * y) + 1)$ aus E ableitbar ist (ohne dass die Reihenfolge der Ableitungsschritte festgelegt wird).

Lemma 2.8

- *Zu jeder Ableitung $A \stackrel{*}{\Rightarrow} u$ existiert ein 'entsprechender' Ableitungsbaum mit Wurzel A und Blattwort u.*
- *Umgekehrt existiert zu jedem Ableitungsbaum mit Wurzel A und Blattwort u genau eine Linksableitung $A \stackrel{*}{\Rightarrow} u$.*

Ein Ableitungsbaum steht also für eine ganze *Menge* von Ableitungen, die sich nur in der Reihenfolge der Ableitungsschritte unterscheiden. Unter all diesen Ableitungen gibt es genau eine Linksableitung, die man erhält, indem man den Baum 'von links nach rechts' durchläuft.

Ableitungsbäume sind einerseits abstrakter als Ableitungen (weil sie von der Reihenfolge der Ableitungsschritte abstrahieren), andererseits sind sie anschaulicher, weil sie die *Struktur* des abgeleiteten Wortes erkennen lassen.

In der Praxis (Compilerbau) ist es wichtig, dass jedes Wort eine eindeutige Struktur besitzt, denn ein Compiler soll ja nicht nur testen, ob eine eingegebene Zeichenreihe ein arithmetischer Ausdruck ist, sondern er soll diesen Ausdruck auch weiterverarbeiten, d.h. letzten Endes in Maschinencode übersetzen.

Definition 2.9

- *Eine kontextfreie Grammatik heißt* eindeutig, *wenn für jedes Wort* $w \in L(G)$ *genau ein Ableitungsbaum mit Wurzel* S *und Blattwort* w *existiert (oder äquivalent dazu: genau eine Linksableitung* $S \stackrel{*}{\Rightarrow} w$*). Andernfalls heißt sie* mehrdeutig.
- *Eine kontextfreie Sprache* L *heißt* inhärent mehrdeutig, *wenn* jede *kontextfreie Grammatik* G *mit* $L = L(G)$ *mehrdeutig ist.*

Beispiel:
Unsere Grammatik G für vollständig geklammerte arithmetische Ausdrücke ist eindeutig.

Intuitive Begründung:
Durch die vollständige Klammerung ist die Struktur jedes Ausdrucks eindeutig festgelegt.

Beweisskizze:
Man beweist zunächst (durch eine einfache Induktion über die Länge der Ableitung von e), dass

- jedes Wort $e \in L(G)$ genauso viele öffnende wie schließende Klammern hat,
- jedes echte nichtleere Präfix eines Wortes $e \in L(G)$ *mehr* öffnende als schließende Klammern hat.

Es folgt sofort, dass kein Wort $e \in L(G)$ echtes Präfix eines anderen Wortes $e' \in L(G)$ sein kann.

Damit kann man nun zeigen, dass jeder Ausdruck $e \in L(G)$ eine eindeutige Struktur hat: Nehmen wir z.B. an, dass ein Ausdruck $e \in L(G)$ sich sowohl in der Form $(e_1 + e_2)$ als auch in der Form $(e'_1 * e'_2)$ darstellen lässt (mit $e_1, e_2, e'_1, e'_2 \in L(G)$).

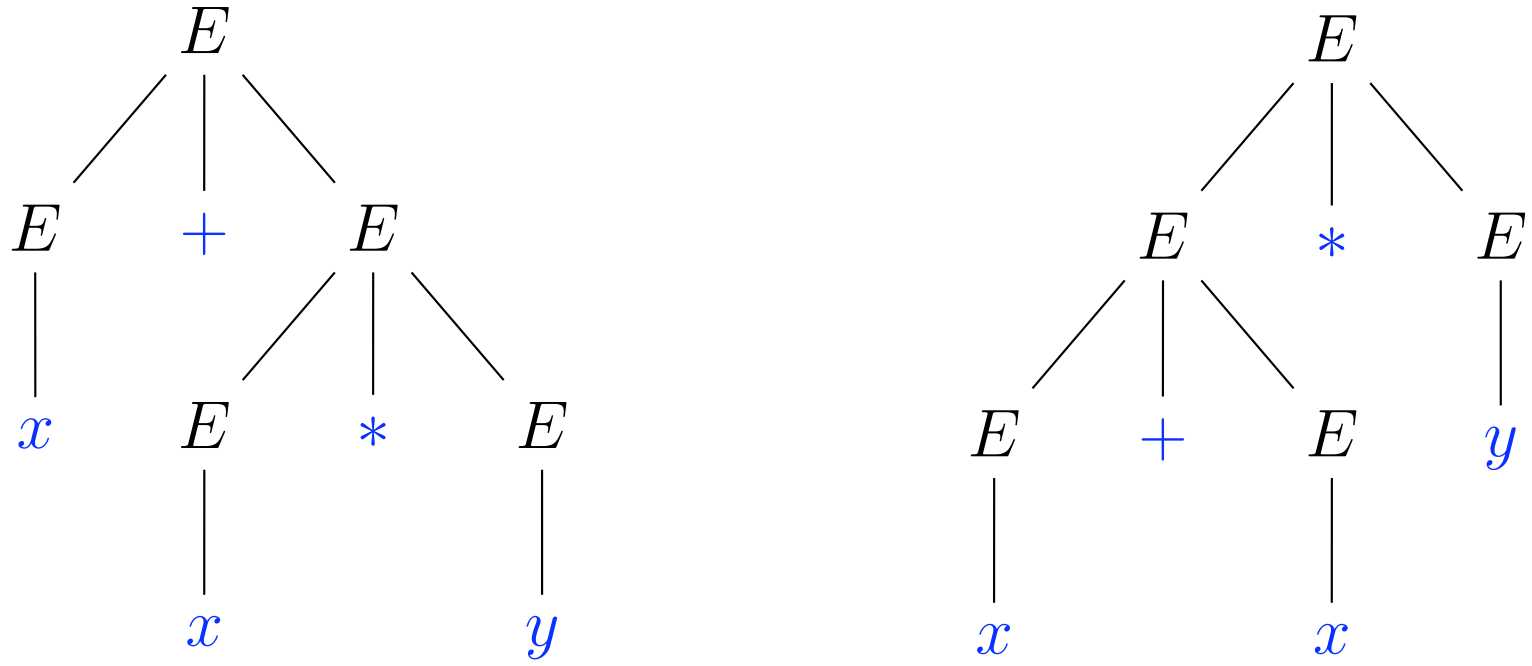
Dann ist einer der Ausdrücke e_1, e'_1 kürzer als der andere, und müsste deshalb echtes nichtleeres Präfix des anderen sein (weil die beiden Zeichenreihen (e_1 und (e'_1 Präfixe von e sind). Das ist aber nicht möglich. □

Beispiel für eine mehrdeutige Grammatik:

Sei $G_1 = (\Sigma, N, E, P)$ mit

- $\Sigma = \{0, 1, x, y, -, +, *, (,)\}$
- $N = \{E\}$
- $P = \{E \to 0,\ E \to 1,\ E \to x,\ E \to y,\ E \to -E,$
  $E \to E + E,\ E \to E - E,\ E \to E * E,\ E \to (E)\}$

G_1 erzeugt die Sprache der unvollständig geklammerten arithmetischen Ausdrücke und ist (in hohem Maße) mehrdeutig,
z.B. hat das Wort $x + x * y$ die beiden folgenden Ableitungsbäume.



Gibt es eine eindeutige KFG für $L(G_1)$?

Sei $G_2 = (\Sigma, N, E, P)$ mit:

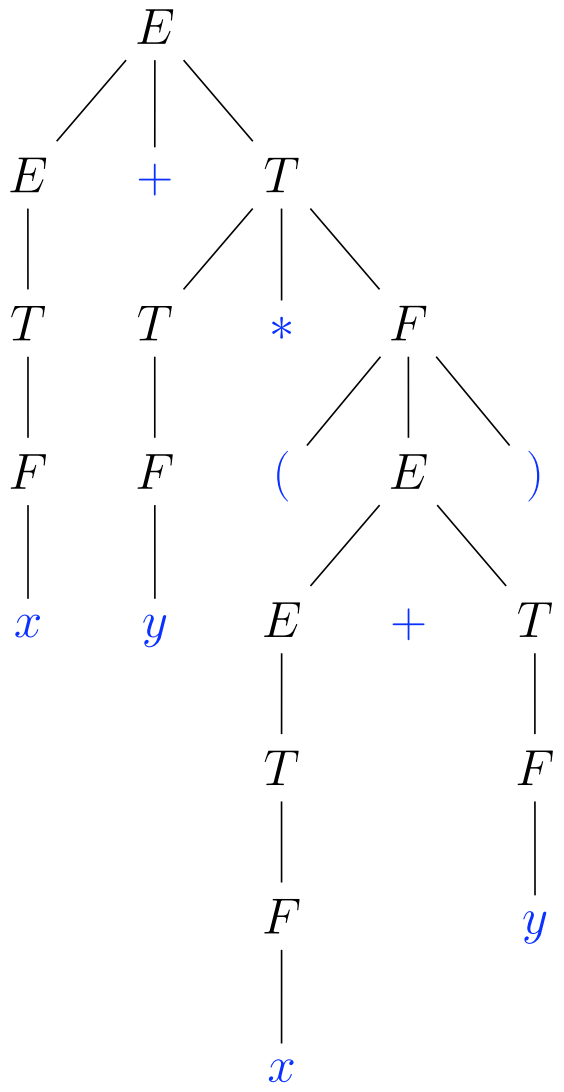
- $\Sigma = \{0, 1, x, y, -, +, *, (,)\}$
- $N = \{E, T, F\}$
- $P = \{$

$$
\begin{array}{lcll}
E & \rightarrow & E + T, & (1)\\
E & \rightarrow & E - T, & (2)\\
E & \rightarrow & T, & (3)\\
T & \rightarrow & T * F, & (4)\\
T & \rightarrow & F, & (5)\\
F & \rightarrow & (E), & (6)\\
F & \rightarrow & -F, & (7)\\
F & \rightarrow & 0 \mid 1 \mid x \mid y\} & (8)
\end{array}
$$

Dann sieht die eindeutige Linksableitung für $E \stackrel{*}{\Rightarrow} x + y * (x + y)$ so aus:

$$
\begin{array}{lcll}
E & \Rightarrow & E + T & (1)\\
 & \stackrel{*}{\Rightarrow} & x + T & (3), (5), (8)\\
 & \Rightarrow & x + T * F & (4)\\
 & \stackrel{*}{\Rightarrow} & x + y * F & (5), (8)\\
 & \Rightarrow & x + y * (E) & (6)\\
 & \Rightarrow & x + y * (E + T) & (1)\\
 & \stackrel{*}{\Rightarrow} & x + y * (x + T) & (3), (5), (8)\\
 & \stackrel{*}{\Rightarrow} & x + y * (x + y) & (5), (8)
\end{array}
$$

und der zugehörige Ableitungsbaum



Jetzt wäre noch zu zeigen, dass G_2 eindeutig ist und dass $L(G_2) = L(G_1)$. Das beweisen wir nicht.

Man beachte, dass im Ableitungsbaum die üblichen Prioritäten der Operatoren zum Ausdruck kommen: $*$ bindet stärker als $+$, deshalb ist der Gesamtausdruck von der Form $E + T$. Etwas anderes lässt die Grammatik G_2 nicht zu, weil links von $*$ niemals ein $+$ stehen kann, das nicht durch Klammern 'geschützt' ist.

Wie beweist man, dass eine KFG die gewünschte Sprache erzeugt?

Beispiel:

Sei $L = \{a^n b^n \mid n \geq 0\}$
und $G = (\{a, b\}, \{S\}, S, P)$ mit $P = \{S \to aSb, S \to \varepsilon\}$

Behauptung: $L(G) = L$.

Beweis:

'$\subseteq$': Sei $w \in \Sigma^*$ mit $S \stackrel{*}{\Rightarrow} w$.

Es ist zu zeigen, dass $w \in L$.

Dazu beweist man eine 'passende Behauptung' durch *Induktion über die Länge der Ableitung*.

Für den Induktionsschritt hat man zwei Möglichkeiten: Entweder man spaltet den *ersten* Ableitungsschritt ab oder den *letzten*.

Will man den *letzten* Ableitungsschritt abspalten, so muss man eine Behauptung für alle $u \in (N \cup \Sigma)^*$ mit $S \stackrel{*}{\Rightarrow} u$ aufstellen, z.B.:

$$\text{Wenn } u \notin \Sigma^*, \text{ dann ist } u = a^n S b^n \text{ für ein } n \geq 0. \qquad (*)$$

$S \stackrel{0}{\Rightarrow} u$:

Dann ist $u = S = a^0 S b^0$.

$S \stackrel{+}{\Rightarrow} u$, d.h. $S \stackrel{*}{\Rightarrow} v \Rightarrow u$ für ein $v \notin \Sigma^*$:

Dann ist $v = a^n S b^n$ nach Induktionsannahme. Wegen $u \notin \Sigma^*$ kann im Ableitungsschritt $v \Rightarrow u$ nur die Produktion $S \to aSb$ angewandt worden sein, also ist $u = a^{n+1} S b^{n+1}$.

Man kann die Argumentation noch verkürzen: Es genügt offensichtlich zu zeigen, dass (*) für S gilt und bei jedem Ableitungsschritt erhalten bleibt. (*) ist also eine *Invariante* für die Ableitungen aus S.

Aus (*) folgt schließlich $L(G) \subseteq L$:

Wenn $w \in L(G)$, dann gilt $S \stackrel{*}{\Rightarrow} u \Rightarrow w$ für ein $u \notin \Sigma^*$, also $u = a^n S b^n$ für ein $n \geq 0$. Wegen $w \in \Sigma^*$ kann $u \Rightarrow w$ nur mit der Produktion $S \to \varepsilon$ erfolgen, also ist $w = a^n b^n \in L$.

Die Abspaltung des *ersten* Ableitungsschrittes ist in unserem Falle wesentlich einfacher.

Man stellt eine Behauptung für alle $w \in \Sigma^*$ mit $S \stackrel{*}{\Rightarrow} w$ auf, nämlich:

$$w \in L \qquad (**)$$

$S \stackrel{1}{\Rightarrow} w$:

Dann ist $w = \varepsilon \in L$.

$S \Rightarrow aSb \stackrel{*}{\Rightarrow} w$:

Dann ist $w = avb$ mit $S \stackrel{*}{\Rightarrow} v$, also gilt $v \in L$, d.h. $v = a^n b^n$ für ein $n \geq 0$ und damit $w = a^{n+1} b^{n+1}$.

Geht es immer so einfach?

Im allgemeinen muss man für *jedes* $A \in N$ eine Behauptung über alle $w \in \Sigma^*$ mit $A \stackrel{*}{\Rightarrow} w$ aufstellen, und diese Behauptungen durch *simultane Induktion* beweisen.

Mit anderen Worten: Wenn $|N| = k$, so muss man eine Behauptung über die k Sprachen $L_A(G) = \{w \in \Sigma^* \mid A \stackrel{*}{\Rightarrow} w\}$ aufstellen und durch simultane Induktion beweisen.

'$\supseteq$': Sei $w \in L = \{a^n b^n \mid n \geq 0\}$.

Durch Induktion über $|w|$ beweist man $w \in L(G)$.

$|w| = 0$, d.h. $w = \varepsilon$:

Dann gilt $S \Rightarrow w$ mit Produktion $S \to \varepsilon$.

$|w| > 0$, d.h. $w = a^n b^n$ mit $n > 0$:

Dann ist $w = aa^{n-1}b^{n-1}b$, also $w = avb$ mit $v \in L$. Nach Induktionsannahme gilt $S \stackrel{*}{\Rightarrow} v$, also $S \Rightarrow aSb \stackrel{*}{\Rightarrow} avb = w$.

Auch diese Richtung wird schwieriger, wenn man mehrere Nichtterminalzeichen hat.

Dann kann man wieder eine geeignete Behauptung über die Sprachen $L_A(G)$ aufstellen und durch Induktion beweisen.

Wir haben im Beispiel die folgenden Eigenschaften von Ableitungen benutzt:

Lemma 2.10

1. *Für alle* $A \in N$ *und* $u, v, w \in (N \cup \Sigma)^*$ *gilt:*

   Wenn $A \stackrel{*}{\Rightarrow} v$, *dann* $uAw \stackrel{*}{\Rightarrow} uvw$.

2. *Für alle* $A \in N$ *und* $u, w, x \in \Sigma^*$ *gilt:*

   Wenn $uAw \stackrel{*}{\Rightarrow} x$, *dann existiert ein* $v \in \Sigma^*$ *mit* $A \stackrel{*}{\Rightarrow} v$ *und* $x = uvw$.

Weiteres Beispiel:

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

Es gibt eine einfache Grammatik, die L erzeugt, nämlich:

$G = (\{a,b\}, \{S\}, S, P)$ mit $P = \{S \to SS, S \to aSb, S \to bSa, S \to \varepsilon\}$

Behauptung: $L(G) = L$.

Beweis:

'$\subseteq$': Für jedes $u \in (N \cup \Sigma)^*$ mit $S \stackrel{*}{\Rightarrow} u$ gilt:

$$\#_a(u) = \#_b(u) \qquad (*)$$

denn (*) gilt für S und bleibt bei jedem Ableitungsschritt erhalten (weil in jeder Produktion gleich viele as und bs hinzukommen).

Also gilt (*) insbesondere für jedes $w \in L(G)$, d.h. $L(G) \subseteq L$.

'$\supseteq$': Sei $w \in L$, $w = a_1 \dots a_n$.

Wenn $w = \varepsilon$, so gilt $S \Rightarrow w$ mit Produktion $S \to \varepsilon$.

Es bleibt der Fall $|w| > 0$ zu betrachten.

Sei $f : \{a, b\}^* \to \mathbb{Z}$ mit $f(v) = \#_a(v) - \#_b(v)$.

Dann gilt $L = \{v \in \{a, b\}^* \mid f(v) = 0\}$.

Wir betrachten den 'Verlauf' der Funktion f für die Präfixe von w, d.h. für die Wörter $w_i = a_1 \dots a_i$ mit $0 \leq i \leq n$.

Es gilt $f(w_0) = f(\varepsilon) = 0$ und $f(w_n) = f(w) = 0$, weil $w \in L$.

1. Fall: $f(w_i) \geq 0$ für $i = 0, \dots, n$

Dann ist $f(w_1) = 1$ und $f(w_{n-1}) = 1$, also $w = avb$ für ein $v \in L$.

Nach Induktionsannahme gilt $S \stackrel{*}{\Rightarrow} v$, also $S \Rightarrow aSb \stackrel{*}{\Rightarrow} avb = w$.

2. Fall: $f(w_i) \leq 0$ für $i = 0, \ldots, n$

Analog zum 1. Fall folgt $w = bva$ für ein $v \in L$,

also gilt $S \Rightarrow bSa \stackrel{*}{\Rightarrow} bva = w$.

3. Fall: Es existieren $i, j \in \{0, \ldots, n\}$ mit $f(w_i) > 0$ und $f(w_j) < 0$.

Dann existiert eine Zahl k zwischen i und j mit $f(w_k) = 0$, also $w_k = a_1 \ldots a_k \in L$.

Dann ist auch $w'_k = a_{k+1} \ldots a_n \in L$ und es gilt $w = w_k w'_k$.

Nach Induktionsannahme gilt $S \stackrel{*}{\Rightarrow} w_k$ und $S \stackrel{*}{\Rightarrow} w'_k$, also $S \Rightarrow SS \stackrel{*}{\Rightarrow} w_k S \stackrel{*}{\Rightarrow} w_k w'_k = w$. $\square$

Die Beispiele zeigen, dass es kontextfreie Sprachen gibt, die *nicht* regulär sind.

Wir beweisen jetzt: Jede reguläre Sprache ist kontextfrei.

Um reguläre Sprachen zu erzeugen, reichen sehr spezielle Grammatiken aus.

Definition 2.11 *Eine Grammatik heißt* rechtslinear, *wenn jede Produktion von der Form* $A \to aB$ *oder* $A \to \varepsilon$ *(mit* $A, B \in N$ *und* $a \in \Sigma$*) ist. Sie heißt* linkslinear, *wenn jede Produktion von der Form* $A \to Ba$ *oder* $A \to \varepsilon$ *ist.*

In der Literatur sind oft noch andere Produktionen zugelassen, etwa $A \to a$, $A \to B$ oder $A \to wB$ mit $w \in \Sigma^*$. Dadurch werden die Grammatiken nicht mächtiger.

Beispiel:

Sei $L = \{a^{2n} \mid n \geq 0\}$
und $G = (\{a\}, \{S_0, S_1\}, S_0, P)$ mit $P = \{S_0 \to aS_1, S_1 \to aS_0, S_0 \to \varepsilon\}$

Behauptung: $L(G) = L$.

Beweis:

'$\subseteq$': Jedes aus S_0 ableitbare Wort, das nicht in $\{a\}^*$ liegt, hat die Form $a^{2n}S_0$ oder $a^{2n+1}S_1$. (Induktion über die Länge der Ableitung).

Da jedes $w \in L(G)$ nur mit Produktion $S_0 \to \varepsilon$ aus einem solchen Wort entstehen kann, muss $w = a^{2n}$ sein für ein $n \geq 0$.

'$\supseteq$': Für jedes $n \geq 0$ gilt

$S_0 \Rightarrow aS_1 \Rightarrow aaS_0 \Rightarrow \ldots \Rightarrow a^{2n}S_0 \Rightarrow a^{2n}$ $\square$

Am Beispiel sieht man, dass die Nichtterminalzeichen einer rechtslinearen Grammatik eine ähnliche Rolle spielen wie die Zustände eines endlichen Automaten.

In jedem Ableitungsschritt (außer dem letzten) wird ein Terminalzeichen erzeugt und das Nichtterminalzeichen am Ende des Wortes wird eventuell verändert.

So kann man sich im Nichtterminalzeichen eine Information über die bereits erzeugten Terminalzeichen merken.

Im Beispiel: S_0 bedeutet, dass bisher eine gerade Anzahl von as erzeugt wurde, S_1 bedeutet, dass eine ungerade Anzahl von as erzeugt wurde.

Satz 2.12

1. *Zu jedem NDEA A kann man eine rechtslineare Grammatik G mit $L(G) = L(A)$ konstruieren.*
2. *Zu jeder rechtslinearen Grammatik G kann man einen NDEA A mit $L(A) = L(G)$ konstruieren.*

Beweis:

1. Sei $A = (\Sigma, Q, s, F, \Delta)$ ein NDEA.

   Wir dürfen annehmen, dass $\Sigma \cap Q = \emptyset$.

   Sei $G = (\Sigma, Q, s, P)$ mit

   $P = \{p \to aq \mid (p, a, q) \in \Delta\} \cup \{p \to \varepsilon \mid p \in F\}$

   Dann folgt durch eine einfache Induktion über $|w|$:

$$p \stackrel{*}{\Rightarrow}_G wq \;\Leftrightarrow\; (p, w) \vdash (q, \varepsilon) \qquad (*)$$

Also gilt

$$
\begin{aligned}
w \in L(G) \quad &\Leftrightarrow \quad s \stackrel{*}{\Rightarrow}_G w \\
&\Leftrightarrow \quad \text{es existiert ein } p \in Q \text{ mit } s \stackrel{*}{\Rightarrow}_G pw \Rightarrow w \\
&\qquad\quad \text{und } (p \to \varepsilon) \in P \\
&\qquad\quad \text{(weil der letzte Ableitungsschritt nur mit einer} \\
&\qquad\quad \varepsilon\text{-Produktion erfolgen kann)} \\
&\Leftrightarrow \quad \text{es existiert ein } p \in Q \text{ mit } (s, w) \vdash_A^* (p, \varepsilon) \\
&\qquad\quad \text{und } p \in F \\
&\qquad\quad \text{(wegen (*) und der Definition von } P) \\
&\Leftrightarrow \quad w \in L(A)
\end{aligned}
$$

und damit ist $L(G) = L(A)$ bewiesen.

2. Sei $G = (\Sigma, N, S, P)$ eine rechtslineare Grammatik.

   Wir definieren $A = (\Sigma, N, S, F, \Delta)$ mit

   $F = \{B \in N \mid (B \to \varepsilon) \in P\}$ und

   $\Delta = \{(B, a, C) \in N \times \Sigma \times N \mid (B \to aC) \in P\}$

   Dann folgt durch eine einfache Induktion über $|w|$:

$$B \stackrel{*}{\Rightarrow}_G wC \;\Leftrightarrow\; (B, w) \vdash_A^* (C, \varepsilon)$$

   und ähnlich wie oben ergibt sich $L(A) = L(G)$ □

Damit haben wir eine weitere Charakterisierung regulärer Sprachen.

Korollar 2.13 *Eine Sprache ist genau dann regulär, wenn sie sich von einer rechtslinearen Grammatik erzeugen lässt.*

Außerdem ist damit natürlich bewiesen, dass jede reguläre Sprache kontextfrei ist, also gilt

Satz 2.14 $\mathcal{L}_{reg} \subseteq \mathcal{L}_{kf}$

wobei $\mathcal{L}_{kf}$ die Klasse der kontextfreien Sprachen bezeichnet.

Unsere Beispiele zeigen, dass diese Inklusion echt ist, wenn das Alphabet Σ mindestens zwei Elemente enthält. (Bei einelementigem Alphabet stimmen die beiden Sprachklassen überein.)

Definition 2.15 *Ein* Kellerautomat *(engl.* pushdown automaton *oder* PDA*) ist ein 6-Tupel* $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ *mit:*

- Σ *ist ein Alphabet (das* Eingabealphabet*)*
- Γ *ist ein Alphabet (das* Kelleralphabet*)*
- Q *ist eine endliche Menge (von* Zuständen*)*
- $s \in Q$ *(der* Startzustand*)*
- $F \subseteq Q$ *(Menge der* Endzustände *oder* akzeptierenden Zustände*)*
- Δ *ist eine* endliche *Teilmenge von* $(Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*)$

Δ *heißt* Übergangsrelation *von* M*, die Elemente von* Δ *heißen* Übergänge *oder* Transitionen *von* M*.*

Man beachte: Ein Kellerautomat kann 'extrem nichtdeterministisch' sein, weil die Übergangsrelation Δ ähnlich wie die eines ε-NDEA 'spontane Zustandsübergänge' erlaubt, bei denen sowohl das Eingabewort als auch der Kellerinhalt ignoriert werden.

Intuition für die Übergänge:

$((p, u, \beta), (q, \gamma)) \in \Delta \subseteq (Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*)$ bedeutet:

Wenn der Kellerautomat M

- sich im Zustand p befindet,
- das aktuelle Eingabewort mit u beginnt
- und der aktuelle Kellerinhalt mit β,

dann darf er in einem Übergangsschritt

- u entfernen,
- in den Zustand q wechseln
- und β durch γ ersetzen.

Dabei wird der Kellerinhalt von oben nach unten gelesen, d.h. β besteht aus den ersten Zeichen, die *oben* im Keller stehen.

Spezialfälle:

- $u = \varepsilon$:

  Das aktuelle Eingabewort wird ignoriert und bleibt unverändert.

- $\beta = \varepsilon$:

  Der aktuelle Kellerinhalt wird ignoriert und mit γ aufgefüllt.

- $\gamma = \varepsilon$:

  β wird durch ε ersetzt, d.h. vom Keller entfernt.

- $\beta = \gamma$:

  Der aktuelle Kellerinhalt bleibt unverändert.

- $\beta = \alpha\gamma$:

  $\alpha\gamma$ wird durch γ ersetzt, d.h. α wird vom Keller entfernt.

- $\gamma = \alpha\beta$:

  β wird durch $\alpha\beta$ ersetzt, d.h. der aktuelle Kellerinhalt wird mit α aufgefüllt.

Definition 2.16 *Sei $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ ein Kellerautomat.*

1. *Eine Konfiguration von M ist ein Element $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$.*

   Intuition: q ist der aktuelle Zustand, w das aktuelle Eingabewort und α der aktuelle Kellerinhalt.

2. *Die Relation $\vdash_M$ (oder kurz: $\vdash$) auf der Menge der Konfigurationen ist wie folgt definiert:*

   Wenn $((p, u, \beta), (q, \gamma)) \in \Delta$, dann gilt für alle $v \in \Sigma^*$ und $\alpha \in \Gamma^*$:

   $$(p, uv, \beta\alpha) \vdash_M (q, v, \gamma\alpha)$$

   und das sind die einzigen Konfigurationen, die in Relation $\vdash_M$ stehen.

Ein Paar $(p, uv, \beta\alpha) \vdash_M (q, v, \gamma\alpha)$ heißt Übergangsschritt von M.

$\vdash_M^n$ $(n \geq 0,)$ $\vdash_M^+$ und $\vdash_M^*$ sind wie üblich definiert.

Beispiel:

Sei $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ mit

- $\Sigma = \Gamma = \{a, b\}$
- $Q = \{s, f\}$
- $F = \{f\}$
- $\Delta = \{$ $((s, a, \varepsilon), (s, a))$, (1) a in den Keller
  $((s, \varepsilon, \varepsilon), (f, \varepsilon))$, (2) spontaner Zustandswechsel
  $((f, b, a), (f, \varepsilon))$ $\}$ (3) b gegen a aufheben

Dann gilt z.B.

$(s, aabb, \varepsilon) \vdash_M (s, abb, a)$ mit (1)
$\vdash_M (s, bb, aa)$ mit (1)
$\vdash_M (f, bb, aa)$ mit (2)
$\vdash_M (f, b, a)$ mit (3)
$\vdash_M (f, \varepsilon, \varepsilon)$ mit (3)

Definition 2.17 *Sei $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ ein Kellerautomat.*

1. *M akzeptiert das Wort $w \in \Sigma^*$, wenn ein $q \in F$ existiert mit $(s, w, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon)$.*
2. *$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$ heißt die von M akzeptierte oder erkannte Sprache.*

In Worten: M startet im Startzustand s mit dem Wort w auf dem Eingabeband und mit leerem Keller. w wird akzeptiert, wenn M einen Endzustand q erreichen kann, so dass sowohl das Eingabeband als auch der Keller leer sind.

Beispiel:
Der oben definierte Kellerautomat M akzeptiert das Wort $aabb$, weil $(s, aabb, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)$ und $f \in F$. Er 'errät' dabei den Zeitpunkt, in dem er vom Zustand s in den Zustand f wechselt.

Behauptung: $L(M) = \{a^n b^n \mid n \geq 0\}$.

Beweis:

'$\supseteq$': Sei $n \geq 0$. Dann gilt (auch für $n = 0$):

$$\begin{aligned} (s, a^n b^n, \varepsilon) &\vdash_M^n (s, b^n, a^n) && \text{mit (1)} \\ &\vdash_M (f, b^n, a^n) && \text{mit (2)} \\ &\vdash_M^n (f, \varepsilon, \varepsilon) && \text{mit (3)} \end{aligned}$$

'$\subseteq$': Sei $w \in L(M)$, d.h. $(s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)$, da f einziger Endzustand ist. In $\vdash_M^*$ muss genau ein Schritt mit Transition (2) vorkommen, davor können nur (1)-Schritte stehen, dahinter nur (3)-Schritte. Also existieren $n \geq 0$, $v \in \Sigma^*$ mit $w = a^n v$ und

$$(s, w, \varepsilon) \vdash_{(1)}^* (s, v, a^n) \vdash_{(2)} (f, v, a^n) \vdash_{(3)}^* (f, \varepsilon, \varepsilon)$$

Das kann aber nur gelten, wenn $v = b^n$, also $w = a^n b^n$. □

Anmerkungen zur Literatur:

Sowohl die Definition des Kellerautomaten als auch die Definition des Akzeptierens variieren stark in der Literatur.

- Unsere Kellerautomaten sind sehr flexibel:

  Sowohl vom Eingabeband als auch vom Keller darf in jedem Schritt ein ganzes Wort (auch ε) gelesen werden. Oft wird stattdessen

  $$\Delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$$

  gefordert. Das setzt natürlich voraus, dass im Keller stets mindestens ein Zeichen steht, d.h. man benötigt ein spezielles Startzeichen, das zu Beginn schon im Keller steht.

- Unsere Definition des Akzeptierens ist sehr streng:

  Der Kellerautomat muss im Endzustand sein *und* der Keller muss leer sein. Oft wird nur eins von beiden gefordert.

Man kann sich davon überzeugen, dass solche unterschiedlichen Definitionen zur gleichen Sprachklasse führen.

Weiteres Beispiel:

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

Sei $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ mit

- $\Sigma = \Gamma = \{a, b\}$
- $Q = F = \{s\}$
- $\Delta = \{$ $((s,a,\varepsilon),(s,a))$, (1) a in den Keller
  $((s,b,\varepsilon),(s,b))$, (2) b in den Keller
  $((s,a,b),(s,\varepsilon))$, (3) a gegen b aufheben
  $((s,b,a),(s,\varepsilon))$ $\}$ (4) b gegen a aufheben

Behauptung: $L = L(M)$.

Beweis:

'$\subseteq$': Man *kann* die Übergangsschritte stets so wählen, dass im Keller der Überschuss an as bzw. bs des bisher gelesenen Wortes steht, d.h. wenn $w \in \Sigma^*$ mit $\#_a(w) = m$ und $\#_b(w) = n$, dann gilt:

$(s, w, \varepsilon) \vdash_M^* (s, \varepsilon, a^{m-n})$ falls $m \geq n$

$(s, w, \varepsilon) \vdash_M^* (s, \varepsilon, b^{n-m})$ falls $m \leq n$

Das zeigt man durch Induktion über $|w|$:

$w = \varepsilon$:

$(s, w, \varepsilon) \vdash_M^0 (s, \varepsilon, a^{0-0}) = (s, \varepsilon, b^{0-0})$

$w = va$:

Wenn $m > n$, dann ist $m - 1 \geq n$, also nach Induktionsannahme $(s, v, \varepsilon) \vdash_M^* (s, \varepsilon, a^{m-1-n})$ und damit $(s, w, \varepsilon) \vdash_M^* (s, a, a^{m-1-n}) \vdash_M (s, \varepsilon, a^{m-n})$. Wenn $m \leq n$, dann ist $m - 1 < n$, also nach Induktionsannahme $(s, v, \varepsilon) \vdash_M^* (s, \varepsilon, b^{n-(m-1)})$ und damit $(s, w, \varepsilon) \vdash_M^* (s, a, b^{n-(m-1)}) \vdash_M (s, \varepsilon, b^{n-m})$. Damit ist auch im Falle $m = n$ alles bewiesen, weil dann $b^{n-m} = \varepsilon = a^{m-n}$.

$w = vb$:

Analog zu $w = va$ (wegen der Symmetrie).

Damit ist '$\subseteq$' bewiesen, denn für $w \in L$ gilt $m = n$, also $(s, w, \varepsilon) \vdash_M^* (s, \varepsilon, a^0) = (s, \varepsilon, \varepsilon)$ und das bedeutet $w \in L(M)$.

'$\supseteq$': Sei $w \in L(M)$, d.h. $(s, w, \varepsilon) \vdash_M^* (s, \varepsilon, \varepsilon)$.

Für jeden Übergangsschritt von M gilt: Entweder es wird nur ein Zeichen verschoben (vom Eingabeband in den Keller), oder es wird gleichzeitig ein a und ein b entfernt (eines vom Eingabeband, das andere vom Keller).

Da in der letzten Konfiguration $(s, \varepsilon, \varepsilon)$ gleich viele as und bs vorhanden sind (Eingabewort und Kellerwort zusammengerechnet), muss dies auch für jede vorhergehende Konfiguration gelten, also insbesondere für die Anfangskonfiguration (s, w, ε).

Das bedeutet aber $\#_a(w) = \#_b(w)$, d.h. $w \in L$. $\square$

Im Beweis wurde die Tatsache benutzt, dass eine Verlängerung des Eingabewortes vom Kellerautomaten ignoriert werden kann. Gleiches gilt für das Kellerwort.

Lemma 2.18 *Wenn*

$$(p, u, \alpha) \vdash_M^* (q, v, \beta)$$

dann gilt auch

$$(p, uw, \alpha\gamma) \vdash_M^* (q, vw, \beta\gamma)$$

für alle $v \in \Sigma^*$ *und* $\gamma \in \Gamma^*$.

Beweisidee:

Mit $(p, uv, \alpha\gamma)$ kann man die gleiche Folge von Übergangsschritten durchführen wie mit (p, u, α), indem man v und γ ignoriert.

(Die Tatsache, dass man mit $(p, uv, \alpha\gamma)$ vielleicht auch neue Übergangsschritte durchführen kann, indem man γ benutzt, interessiert hier nicht.) □

Satz 2.19 *Zu jeder kontextfreien Grammatik G lässt sich ein Kellerautomat M mit $L(M) = L(G)$ konstruieren.*

Beweisidee:

Man konstruiert den Kellerautomaten so, dass er eine Ableitung für das Eingabewort w in der Grammatik G 'erraten' kann.

Diese Ableitung führt er im Keller durch, d.h. er schreibt zu Beginn das Startzeichen in den Keller und führt dann die Ableitungsschritte der Grammatik im Keller aus.

Problem:

In einem Ableitungsschritt der Grammatik wird ein Nichtterminalzeichen A an einer *beliebigen* Stelle des bisher abgeleiteten Wortes durch die rechte Seite γ einer Produktion $A \to \gamma$ ersetzt.

Das kann der Kellerautomat nicht leisten, da er immer nur auf die Zeichen zugreifen kann, die *oben* im Keller stehen.

Lösung(?):

Wir betrachten nur *Linksableitungen*, bei denen ja immer das linkeste Nichtterminalzeichen ersetzt wird.

Aber auch das muss nicht ganz oben im Keller stehen, es können noch Terminalzeichen im Wege sein.

Die müssen aber, wenn die bisherige Ableitung richtig erraten wurde, mit den ersten Zeichen des Eingabewortes w übereinstimmen.

Also kann man sie in diesem Fall entfernen (und der andere Fall interessiert nicht, weil dann die bisherige Ableitung schon falsch erraten wurde).

Damit ist klar, dass der Kellerautomat drei Arten von Transitionen besitzen sollte, nämlich:

1. eine Transition, um das Startzeichen in den Keller zu legen,
2. für jede Produktion der Grammatik eine Transition, die den entsprechenden Ableitungsschritt durchführt,
3. Transitionen, um Terminalzeichen auf dem Eingabeband und im Keller miteinander zu vergleichen und zu entfernen.

Formal:

Sei $G = (\Sigma, N, S, P)$. Wir definieren $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ mit

- $\Gamma = N \cup \Sigma$
- $Q = \{s, f\}$
- $F = \{f\}$
- $$\begin{aligned} \Delta &= \{((s, \varepsilon, \varepsilon), (f, S))\} && (1) \\ &\cup \{((f, \varepsilon, A), (f, \gamma)) \mid (A \to \gamma) \in P\} && (2) \\ &\cup \{((f, a, a), (f, \varepsilon)) \mid a \in \Sigma\} && (3) \end{aligned}$$

Bevor wir $L(M) = L(G)$ beweisen, wollen wir uns die Arbeitsweise dieses Kellerautomaten M an einem kleinen **Beispiel** verdeutlichen:

Sei $G = (\{a, b\}, \{S\}, S, P\}$ mit $P = \{S \to aSb, S \to \varepsilon\}$.

Dann ist $M = (\{a, b\}, \{a, b, S\}, \{s, f\}, s, \Delta)$

mit

$$\begin{aligned}\Delta \ = \{ \ & ((s, \varepsilon, \varepsilon), (f, S)), \\ & (((f, \varepsilon, S), (f, aSb)), \\ & (((f, \varepsilon, S), (f, \varepsilon), \\ & ((f, a, a), (f, \varepsilon)), \\ & ((f, b, b), (f, \varepsilon))\}\end{aligned}$$

also gilt z.B. bei Eingabe $aabb$:

$$\begin{aligned}(s, aabb, \varepsilon) &\vdash_M (f, aabb, S) \\ &\vdash_M (f, aabb, aSb) \\ &\vdash_M (f, abb, Sb) \\ &\vdash_M (f, abb, aSbb) \\ &\vdash_M (f, bb, Sbb) \\ &\vdash_M (f, bb, bb) \\ &\vdash_M (f, b, b) \\ &\vdash_M (f, \varepsilon, \varepsilon)\end{aligned}$$

Am Beispiel sieht man, dass in einer erfolgreichen Berechnung des Kellerautomaten das aktuelle Eingabewort stets aus dem aktuellen Kellerwort ableitbar ist. Diese Beobachtung bringen wir durch folgende Äquivalenz zum Ausdruck: Für alle $\alpha \in \Gamma^*$ und $w \in \Sigma^*$ gilt

$$(f, w, \alpha) \vdash_M^* (f, \varepsilon, \varepsilon) \;\Leftrightarrow\; \alpha \stackrel{*}{\Rightarrow}_G w \qquad (*)$$

Aus (*) folgt $L(M) = L(G)$, denn:

$$\begin{aligned} w \in L(M) &\Leftrightarrow (s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon) \\ &\Leftrightarrow (f, w, S) \vdash_M^* (f, \varepsilon, \varepsilon) \\ &\quad \text{weil als erster Übergangsschritt nur} \\ &\quad (s, w, \varepsilon) \vdash_M (f, w, S) \text{ in Frage kommt} \\ &\Leftrightarrow S \stackrel{*}{\Rightarrow}_G w \\ &\quad \text{wegen (*) mit } \alpha = S \\ &\Leftrightarrow w \in L(G) \end{aligned}$$

Es bleibt also (*) zu beweisen

$$(f, w, \alpha) \vdash_M^* (f, \varepsilon, \varepsilon) \;\Leftrightarrow\; \alpha \stackrel{*}{\Rightarrow}_G w \qquad (*)$$

'$\Rightarrow$': Induktion über die Anzahl n der Übergangsschritte in $\vdash_M^*$

$n = 0$, d.h. $w = \alpha = \varepsilon$:

Dann gilt $\alpha \stackrel{*}{\Rightarrow}_G w$.

$n > 0$, d.h. $(f, w, \alpha) \vdash_M (f, v, \beta) \vdash_M^{n-1} (f, \varepsilon, \varepsilon)$ mit $v \in \Sigma^*$ und $\beta \in \Gamma^*$

Dann gilt nach Induktionsannahme $\beta \stackrel{*}{\Rightarrow}_G v$.

Wenn der erste Übergangsschritt mit (3) erfolgt, so existiert ein $a \in \Sigma$ mit $w = av$ und $\alpha = a\beta$, also gilt $\alpha = a\beta \stackrel{*}{\Rightarrow}_G av = w$.

Wenn der erste Übergangsschritt mit (2) erfolgt, so gilt $w = v$ und β entsteht aus α durch Anwendung einer Produktion $(A \to \gamma) \in P$. Dann gilt $\alpha \Rightarrow_G \beta \stackrel{*}{\Rightarrow}_G v = w$.

$$(f, w, \alpha) \vdash_M^* (f, \varepsilon, \varepsilon) \;\Leftrightarrow\; \alpha \stackrel{*}{\Rightarrow}_G w \qquad (*)$$

'$\Leftarrow$': Induktion über die Länge n einer Linksableitung $\alpha \stackrel{n}{\Rightarrow}_G w$

$n = 0$, d.h. $\alpha = w \in \Sigma^*$:

Dann gilt $(f, w, \alpha) = (f, w, w) \vdash_M^{|w|} (f, \varepsilon, \varepsilon)$ mit (3).

$n > 0$, d.h. es existiert ein $\alpha_1 \in \Gamma^*$ mit $\alpha \Rightarrow_G \alpha_1 \stackrel{n-1}{\Rightarrow}_G w$

Sei $A \to \gamma$ die Produktion im Linksableitungsschritt $\alpha \Rightarrow_G \alpha_1$, d.h. es existieren $u \in \Sigma^*, \beta \in \Gamma^*$ mit $\alpha = uA\beta$ und $\alpha_1 = u\gamma\beta$. Wegen $u \in \Sigma^*$ existiert dann ein $v \in \Sigma^*$ mit $w = uv$ und $\gamma\beta \stackrel{n-1}{\Rightarrow}_G v$, also

$$\begin{aligned} (f, w, \alpha) = (f, uv, uA\beta) &\vdash_M^{|u|} (f, v, A\beta) \text{ mit (3)} \\ &\vdash_M (f, v, \gamma\beta) \text{ mit (2)} \\ &\vdash_M^* (f, \varepsilon, \varepsilon) \text{ nach Induktionsannahme } \square \end{aligned}$$

Die Umkehrung von Satz 2.19 gilt ebenfalls.

Satz 2.20 *Zu jedem Kellerautomaten M lässt sich eine kontextfreie Grammatik G konstruieren mit $L(G) = L(M)$.*

Beweis: s. Literatur □

Also erhalten wir eine zweite Charakterisierung für die Klasse der kontextfreien Sprachen.

Satz 2.21 *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann kontextfrei, wenn es einen Kellerautomaten M gibt mit $L = L(M)$.*

Beweis: Das folgt unmittelbar aus den Sätzen 2.19 und 2.20. □

Abschlusseigenschaften

Satz 2.22 *Die Klasse $\mathcal{L}_{kf}$ der kontextfreien Sprachen ist abgeschlossen unter den Operationen $\cup, \circ, ^*$ und $^+$, d.h. wenn $L_1, L_2 \subseteq \Sigma^*$ kontextfrei sind, dann sind auch die Sprachen*

1. $L_1 \cup L_1$
2. $L_1 \circ L_2$
3. L_1^*
4. L_1^+

kontextfrei. Darüber hinaus gibt es Algorithmen, um Grammatiken für diese Sprachen aus den Grammatiken für L_1 und L_2 zu konstruieren.

Beweis:

Seien $G_i = (\Sigma, N_i, S_i, P_i)$ $(i = 1, 2)$ kontextfreie Grammatiken mit $L(G_i) = L_i$. Wir dürfen annehmen, dass $N_1 \cap N_2 = \emptyset$.

1. siehe Übung 11, Aufgabe 4
2. Sei $G = (\Sigma, N, S, P)$, wobei

   - S ein neues Zeichen ist, d.h. $S \notin N_1 \cup N_2 \cup \Sigma$,
   - $N = N_1 \cup N_2 \cup \{S\}$,
   - $P = \{S \to S_1 S_2\} \cup P_1 \cup P_2$.

   Dann gilt $L(G) = L_1 \circ L_2$.

   '$\supseteq$': Sei $w \in L_1 \circ L_2$, d.h. $w = w_1 w_2$ mit $w_i \in L_i$ für $i = 1, 2$.

   Dann gilt $S_i \stackrel{*}{\Rightarrow}_{G_i} w_i$ und damit auch $S_i \stackrel{*}{\Rightarrow}_{G} w_i$ für $i = 1, 2$, und es folgt $S \Rightarrow_G S_1 S_2 \stackrel{*}{\Rightarrow}_G w_1 S_2 \stackrel{*}{\Rightarrow}_G w_1 w_2 = w$, d.h. $w \in L(G)$.

'$\subseteq$': Sei $w \in L(G)$, d.h. es gibt eine Linksableitung $S \overset{*}{\Rightarrow}_G w$.

Da S ein neues Zeichen ist, kann der erste Ableitungsschritt nur $S \Rightarrow S_1 S_2$ sein, also hat die gesamte Ableitung die Form $S \Rightarrow S_1 S_2 \overset{*}{\Rightarrow}_G w_1 S_2 \overset{*}{\Rightarrow}_G w_1 w_2 = w$, wobei $S_1 \overset{*}{\Rightarrow}_G w_1$ und $S_2 \overset{*}{\Rightarrow}_G w_2$.

Wegen $N_1 \cap N_2 = \emptyset$ können in $S_1 \overset{*}{\Rightarrow}_G w_1$ nur Ableitungsschritte für G_1 vorkommen und in $S_2 \overset{*}{\Rightarrow}_G w_2$ nur solche für G_2.

Also gilt $S_1 \overset{*}{\Rightarrow}_{G_1} w_1$ und $S_2 \overset{*}{\Rightarrow}_{G_2} w_2$, d.h. $w_1 \in L_1, w_2 \in L_2$ und damit $w \in L_1 \circ L_2$.

3. Sei $G = (\Sigma, N, S, P)$, wobei

   $S \notin N_1 \cup \Sigma$,

   $N = N_1 \cup \{S\}$,

   $P = \{S \to S_1 S, S \to \varepsilon\} \cup P_1$.

Dann gilt $L(G) = L_1^*$.

'$\supseteq$': Sei $w \in L_1^*$, d.h. es existieren $n \geq 0$ und $w_1, \ldots, w_n \in L_1$ mit $w = w_1 \ldots w_n$.

Dann gilt $S \Rightarrow S_1 S \Rightarrow \ldots \Rightarrow S_1^n S \Rightarrow S_1^n$, und wegen $S_1 \stackrel{*}{\Rightarrow}_G w_i$ für $i = 1, \ldots n$ folgt $S \stackrel{*}{\Rightarrow}_G w_1 \ldots w_n = w$, also $w \in L(G)$.

'$\subseteq$': Sei $w \in L(G)$. Durch Induktion über die Länge einer Linksableitung $S \stackrel{*}{\Rightarrow}_G w$ beweisen wir $w \in L_1^*$.

Da S ein neues Zeichen ist, kann der erste Ableitungsschritt nur von der Form $S \Rightarrow \varepsilon$ oder $S \Rightarrow S_1 S$ sein.

Im ersten Fall (Induktionsanfang) ist $w = \varepsilon \in L_1^*$, im zweiten Fall ist die Ableitung von der Form $S \Rightarrow S_1 S \stackrel{*}{\Rightarrow}_G w_1 S \stackrel{*}{\Rightarrow}_G w_1 w' = w$, wobei $S_1 \stackrel{*}{\Rightarrow}_G w_1$ und $S \stackrel{*}{\Rightarrow}_G w'$.

Da $S_1 \stackrel{*}{\Rightarrow}_G w_1$ nur Ableitungsschritte für G_1 enthalten kann, gilt dann $w_1 \in L(G_1) = L_1$, und nach Induktionssannahme ist $w' \in L_1^*$.

Also folgt $w = w_1 w' \in L_1 \circ L_1^* \subseteq L_1^*$.

4. Die Abgeschlossenheit unter $^+$ folgt aus der Abgeschlossenheit unter $\circ$ und *, weil $L_1^+ = L_1 \circ L_1^*$. □

Es ist kein Zufall, dass Durchschnitt und Komplement in Satz 2.22 fehlen. Wir werden später sehen, dass der Durchschnitt zweier kontextfreier Sprachen im allgemeinen *nicht* kontextfrei ist.

Daraus folgt sofort, dass auch das Komplement einer kontextfreien Sprache im allgemeinen *nicht* kontextfrei ist, denn aus der Abgeschlossenheit unter Vereinigung und Komplement würde sich die Abgeschlossenheit unter Durchschnitt ergeben.

Für den Durchschnitt gilt eine schwächere Aussage, die sich besser mit Automaten (als mit Grammatiken) beweisen lässt:

Satz 2.23 *Wenn $L_1 \subseteq \Sigma^*$ kontextfrei und $L_2 \subseteq \Sigma^*$ regulär ist, dann ist $L_1 \cap L_2$ kontextfrei.*

Beweis:

Sei $M_1 = (\Sigma, \Gamma, Q_1, s_1, F_1, \Delta_1)$ ein Kellerautomat mit $L(M_1) = L_1$ und sei $A_2 = (\Sigma, Q_2, s_2, F_2, \Delta_2)$ ein NDEA mit $L(A_2) = L_2$.

Wir konstruieren einen Kellerautomaten M mit $L(M) = L_1 \cap L_2$.

Idee: M arbeitet wie M_1 und simuliert "parallel dazu" noch die Übergänge des endlichen Automaten A_2.

Diese "Parallelverarbeitung" bringt man dadurch zum Ausdruck, dass man auf der Zustandsmenge $Q_1 \times Q_2$ arbeitet: In der ersten Komponente merkt man sich den aktuellen Zustand von M_1, in der zweiten Komponente den von A_2.

Sei also $M = (\Sigma, \Gamma, Q_1 \times Q_2, (s_1, s_2), F_1 \times F_2, \Delta)$ mit

$$\Delta = \{(((p_1, p_2), u, \beta), ((q_1, q_2), \gamma)) \mid ((p_1, u, \beta), (q_1, \gamma)) \in \Delta_1 \text{ und } (p_2, u) \vdash_{A_2}^* (q_2, \varepsilon)\}$$

Δ ist gerade so konstruiert, dass in jedem Übergangsschritt von M ein Übergangsschritt von M_1 und eine entsprechende Folge von Übergangsschritten von A_2 simuliert werden, d.h. es gilt stets:

$$((p_1, p_2), u, \alpha) \vdash_M ((q_1, q_2), \varepsilon, \alpha') \Leftrightarrow (p_1, u, \alpha) \vdash_{M_1} (q_1, \varepsilon, \alpha') \text{ und } (p_2, u) \vdash_{A_2}^* (q_2, \varepsilon)$$

Das ergibt sich unmittelbar aus der Definition von Δ und der Definition der möglichen Übergangsschritte eines Kellerautomaten.

Der gleiche Zusammenhang gilt dann auch für *Folgen* von Übergangsschritten:

$$((p_1,p_2),u,\alpha) \vdash_M^* ((q_1,q_2),\varepsilon,\alpha') \Leftrightarrow (p_1,u,\alpha) \vdash_{M_1}^* (q_1,\varepsilon,\alpha') \text{ und } (p_2,u) \vdash_{A_2}^* (q_2,\varepsilon)$$

Das ergibt sich leicht durch Induktion über die Länge der Folge.

Daraus erhält man schließlich das gewünschte Ergebnis:

$$\begin{aligned} w \in L(M) &\Leftrightarrow \text{es existiert ein } (q_1,q_2) \in F \text{ mit} \\ &\quad ((s_1,s_2),w,\varepsilon) \vdash_M^* ((q_1,q_2),\varepsilon,\varepsilon) \\ &\Leftrightarrow \text{es existieren } q_1 \in F_1, q_2 \in F_2 \text{ mit} \\ &\quad (s_1,w,\varepsilon) \vdash_{M_1}^* (q_1,\varepsilon,\varepsilon) \\ &\quad \text{und } (s_2,w) \vdash_{A_2}^* (q_2,\varepsilon) \\ &\Leftrightarrow w \in L(M_1) \cap L(A_2) = L_1 \cap L_2 \end{aligned}$$

□

Entscheidbarkeitsfragen

Wir haben gesehen: Viele Fragestellungen über reguläre Sprachen sind entscheidbar, d.h. es gibt (einfache, manchmal auch effiziente) Algorithmen, die stets die richtige Antwort liefern.

Gilt das auch für kontextfreie Sprachen?

Wichtigste Fragestellung ist das *Wortproblem*:

1. Das *spezielle* Wortproblem für *eine* kontextfreie Grammatik G:

   Eingabe: Ein Wort $w \in \Sigma^*$.

   Frage: Ist $w \in L(G)$?

2. Das *allgemeine* Wortproblem für kontextfreie Grammatiken:

   Eingabe: Eine kontextfreie Grammatik G und ein Wort $w \in \Sigma^*$.

   Frage: Ist $w \in L(G)$?

Für die Praxis ist das spezielle Wortproblem wichtig, denn das ist die Frage, die ein Parser für die Sprache $L(G)$ beantworten muss: Ist die Zeichenreihe, die der Programmierer eingibt, ein syntaktisch korrektes Programm? Diese Frage muss nicht nur entscheidbar sein, sondern sie muss sich *effizient* lösen lassen.

Wir zeigen hier, dass sogar das allgemeine Wortproblem entscheidbar ist (allerdings nicht sehr effizient).

Lösungsansatz:

Um $w \in L(G)$ zu überprüfen, sucht man systematisch nach einer Ableitung für w aus dem Startsymbol S. Man kann z.B. erst alle Wörter $u \in (N \cup \Sigma)^*$ bestimmen, die in einem Schritt aus S ableitbar sind, dann die, die in zwei Schritten ableitbar sind usw.

Wenn w dabei irgendwann auftaucht, ist die Antwort "ja". Aber wann kann man die Antwort "nein" geben, d.h. wann kann man sicher sein, dass w nicht mehr auftaucht?

Wenn wir wüssten, dass bei jedem Ableitungsschritt mindestens ein Zeichen hinzukommt, dann hätten wir ein Abbruchkriterium: Dann bräuchten wir nur Ableitungen zu untersuchen, die höchstens die Länge $|w|$ haben.

Probleme bereiten also Ableitungsschritte, bei denen das Wort *nicht* länger wird. Solche Ableitungsschritte entstehen, wenn man Produktionen $A \to \gamma$ mit $|\gamma| \leq 1$ anwendet.

Definition 2.24

- *Eine Produktion der Form $A \to \varepsilon$ (mit $A \in N$) heißt* ε-Produktion.
- *Eine Produktion der Form $A \to B$ (mit $A, B \in N$) heißt* Einheitsproduktion.

Wir werden zeigen, dass man solche Produktionen (bis auf eine) aus einer kontextfreien Grammatik entfernen kann, ohne dass sich die von der Grammatik erzeugte Sprache verändert.

Satz 2.25 *Zu jeder kontextfreien Grammatik $G = (\Sigma, N, S, P)$ kann man eine äquivalente kontextfreie Grammatik $G' = (\Sigma, N, S, P')$ mit folgenden Eigenschaften konstruieren:*

1. *P' enthält höchstens eine ε-Produktion, nämlich $S \to \varepsilon$,*
2. *diese ε-Produktion wird nur zur Ableitung von ε benötigt.*

Beweis:

Die Konstruktion von G' lässt sich in zwei Phasen unterteilen.

In der *ersten Phase* wird P schrittweise zu einer neuen Produktionenmenge P'' erweitert, wobei jeder einzelne Erweiterungsschritt so aussieht:

Wenn bereits zwei Produktionen der Form $B \to \beta A \gamma$ und $A \to \varepsilon$ vorhanden sind (d.h. wenn sie in P liegen oder durch vorhergehende Erweiterungsschritte hinzugekommen sind), dann wird die Produktion $B \to \beta\gamma$ aufgenommen (die sogar eine ε-Produktion sein kann).

Diese Erweiterungsschritte führt man so lange durch, bis keine neuen Produktionen mehr entstehen.

Das Verfahren *terminiert*, weil die rechte Seite einer neuen Produktion stets kürzer ist als die rechte Seite einer der Produktionen in P. Damit kommen von vornherein nur endlich viele unterschiedliche rechte Seiten und damit auch nur endlich viele unterschiedliche neue Produktionen in Frage.

Das Verfahren ist auch *korrekt*, d.h. durch die neu hinzugenommenen Produktionen wird die erzeugte Sprache nicht größer:

Jede Anwendung einer neuen Produktion $B \to \beta\gamma$ entspricht nämlich einer Anwendung von $B \to \beta A\gamma$, gefolgt von einer Anwendung von $A \to \varepsilon$, d.h. man kann jede Anwendung einer Produktion in P'' letztendlich durch mehrere (aufeinanderfolgende) Anwendungen von Produktionen der ursprünglichen Menge P ersetzen.

In der *zweiten Phase* verkleinert man P'' zu P', indem man einfach alle ε-Produktionen außer $S \to \varepsilon$ aus P'' entfernt.

Es bleibt zu zeigen, dass die erzeugte Sprache durch das Entfernen dieser Produktionen nicht kleiner wird.

Dazu geben wir an, wie man eine Ableitung $S \stackrel{*}{\Rightarrow} w$ eines Wortes $w \in \Sigma^*$ in P'' schrittweise zu einer Ableitung in P' umformen kann.

Jeder einzelne Umformungsschritt sieht so aus:

Wenn im ersten Ableitungsschritt von $S \stackrel{*}{\Rightarrow} w$ eine ε-Produktion angewandt wird, so ist $w = \varepsilon$ und die Ableitung liegt bereits in P'.

Wenn in einem späteren Schritt von $S \stackrel{*}{\Rightarrow} w$ eine ε-Produktion $A \to \varepsilon$ angewandt wird, dann ist das Nichtterminalzeichen A irgendwann vorher durch eine Produktion $B \to \beta A \gamma$ entstanden.

Also kann man sich die Anwendung von $A \to \varepsilon$ ersparen, indem man in diesem früheren Schritt anstelle von $B \to \beta A \gamma$ gleich die Produktion $B \to \beta\gamma$ anwendet (die wir ja in P'' aufgenommen haben).

Den soeben geschilderten Umformungsschritt wiederholt man so oft wie möglich.

Das Verfahren terminiert, weil die Ableitung bei jedem Umformungsschritt um einen Ableitungsschritt kürzer wird (was ja nicht unendlich oft passieren kann).

Also erreicht man irgendwann eine Situation, in der keine Umformung mehr möglich ist.

Das bedeutet aber, dass die Ableitung entweder keine Anwendungen von ε-Produktionen mehr enthält oder nur noch aus dem Ableitungsschritt $S \Rightarrow \varepsilon$ besteht.

In beiden Fällen ist es dann eine Ableitung in P'. □

Satz 2.26 *Zu jeder kontextfreien Grammatik $G = (\Sigma, N, S, P)$ kann man eine äquivalente kontextfreie Grammatik $G' = (\Sigma, N, S, P')$ konstruieren, die keine Einheitsproduktionen enthält (und falls G durch das Verfahren in Satz 2.25 entstanden ist, kann man die Konstruktion so durchführen, dass keine neuen ε-Produktionen entstehen).*

Beweis:

Die Konstruktion von G' verläuft wieder in zwei Phasen.

In der ersten Phase wird P schrittweise zu P'' erweitert, wobei jeder einzelne Erweiterungsschritt so aussieht:

Wenn bereits zwei Produktionen der Form $A \to B$ und $B \to \gamma$ vorhanden sind, dann wird die Produktion $A \to \gamma$ aufgenommen (wobei man hier *keine* ε-Produktionen aufnimmt, falls G nach dem Verfahren in Satz 2.25 entstanden ist).

Diese Erweiterungsschritte führt man so lange durch, bis keine neuen Produktionen mehr entstehen.

Das Verfahren *terminiert*, weil sich die neuen Produktionen von den alten nur durch das Nichtterminalzeichen auf der linken Seite unterscheiden, also können aus jeder Produktion in P höchstens $|N| - 1$ neue entstehen.

Das Verfahren ist *korrekt* weil auch hier jede neue Produktion nur eine 'Abkürzung' für eine Folge von alten Produktionen ist.

In der zweiten Phase verkleinert man P'' zu P', indem man alle Einheitsproduktionen aus P'' entfernt.

Hier muss man sich wieder davon überzeugen, dass die erzeugte Sprache nicht kleiner wird.

Dazu zeigt man, dass sich die Einheitsproduktionen aus jeder Ableitung $S \stackrel{*}{\Rightarrow} w$ eines Wortes $w \in \Sigma^*$ entfernen lassen.

Sei $A \to B$ die *letzte* Einheitsproduktion, die in der Ableitung $S \stackrel{*}{\Rightarrow} w$ angewandt wird.

Weil das entstehende Nichtterminalzeichen B nicht in w vorkommen kann, muss weiter rechts in der Ableitung eine Produktion $B \to \gamma$ auf *dieses* Zeichen B angewandt werden (dabei ist $B \to \gamma$ *keine* ε-Produktion, falls G durch das Verfahren in Satz 2.25 entstanden ist).

Also kann man diese Anwendung von $B \to \gamma$ einsparen, indem man gleich $A \to \gamma$ anstelle von $A \to B$ anwendet.

Man beachte, dass dabei eine Einheitsproduktion aus der Ableitung verschwindet: Per Definition ist $A \to B$ ja die *letzte* Einheitsproduktion in der Ableitung, also ist $B \to \gamma$ und damit auch das neu entstehende $A \to \gamma$ *keine* Einheitsproduktion.

Durch Wiederholung dieses Umformungsschrittes kann man also nach und nach alle Anwendungen von Einheitsproduktionen aus der Ableitung $S \stackrel{*}{\Rightarrow} w$ entfernen, und erhält so eine Ableitung mit der neuen Produktionenmenge P'. $\square$

Satz 2.27 *Für jede der folgenden Fragestellungen gibt es einen Entscheidungsalgorithmus.*

1. *Eingabe: Eine kontextfreie Grammatik G und ein Wort w.*

   Frage: Ist $w \in L(G)$?

2. *Eingabe: Eine kontextfreie Grammatik G.*

   Frage: Ist $L(G) = \emptyset$?.

Beweis:

1. Wegen Satz 2.25 und Satz 2.26 dürfen wir annehmen, dass G keine Einheitsproduktionen enthält und höchstens eine ε-Produktion $S \to \varepsilon$, die nur zur Ableitung von ε benötigt wird.

Im Falle $w = \varepsilon$ brauchen wir nur nachzusehen, ob $S \to \varepsilon$ in der Produktionenmenge liegt.

Im Falle $w \neq \varepsilon$ können wir die Länge einer Ableitung $S \stackrel{*}{\Rightarrow} w$ nach oben abschätzen.

In $S \stackrel{*}{\Rightarrow} w$ werden ja nur Produktionen der Form $A \to \gamma$ mit $|\gamma| > 1$ oder $A \to a$ mit $a \in \Sigma$ angewandt.

Jeder Ableitungsschritt mit einer Produktion der ersten Form macht das bereits abgeleitete Wort länger, also können höchstens $|w| - |S| = |w| - 1$ solche Ableitungsschritte in $S \stackrel{*}{\Rightarrow} w$ vorkommen.

Jeder Ableitungsschritt mit einer Produktion $A \to a$ produziert ein Terminalzeichen, das nicht mehr entfernt werden kann, also können höchstens $|w|$ solche Ableitungsschritte vorkommen.

Damit beträgt die Gesamtlänge der Ableitung $S \stackrel{*}{\Rightarrow} w$ höchstens $2|w| - 1$.

Um zu testen, ob w in $L(G)$ liegt, brauchen wir also nur alle Ableitungen aus S bis zur Länge $2|w| - 1$ zu betrachten und nachzusehen, ob eine von ihnen das Wort w liefert.

2. Um zu testen, ob $L(G) = \emptyset$ ist, müssen wir systematisch nach einer Ableitung eines *beliebigen* Wortes $z \in \Sigma^*$ suchen.

   Auch hier stellt sich die Frage, ob man die Suche irgendwann abbrechen kann, d.h. ob man irgendwann sicher sein kann, dass kein Wort mehr gefunden wird.

   Wir zeigen dazu:

   > Wenn $G = (\Sigma, N, S, P)$ und $L(G) \neq \emptyset$, dann existiert ein Ableitungsbaum für ein Wort $z \in L(G)$ der höchstens die Höhe $|N|$ hat. (*)

   Um zu testen, ob $L(G) = \emptyset$ ist, brauchen wir also nur alle Ableitungsbäume (mit Wurzel S) bis zur Höhe $|N|$ zu erzeugen und nachzusehen, ob einer von ihnen ein Blattwort $z \in \Sigma^*$ hat.

Es bleibt (*) zu zeigen.

> Wenn $G = (\Sigma, N, S, P)$ und $L(G) \neq \emptyset$, dann existiert ein Ableitungsbaum für ein Wort $z \in L(G)$ der höchstens die Höhe $|N|$ hat. (*)

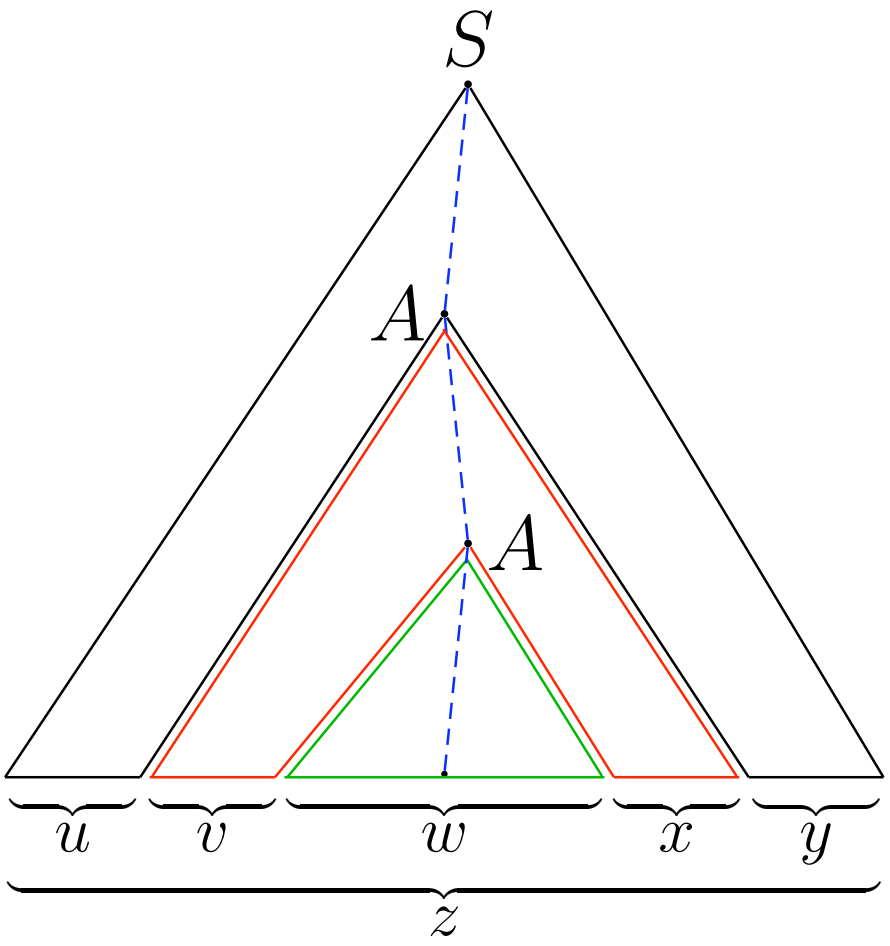
Wenn $L(G) \neq \emptyset$, so existiert zunächst *irgendein* Ableitungsbaum für ein Wort $z \in L(G)$.

Wenn dieser Ableitungsbaum noch nicht die gewünschte Eigenschaft hat, d.h. wenn seine Höhe größer als $|N|$ ist, so existiert ein Pfad von der Wurzel S zu einem Blatt, dessen Länge größer als $|N|$ ist.

Dieser Pfad enthält also mindestens $|N| + 2$ Knoten, von denen nur der letzte (das Blatt) mit einem Terminalzeichen markiert ist.

Also sind mindestens $|N| + 1$ Knoten mit Nichtterminalzeichen markiert, d.h. mindestens ein Nichtterminalzeichen A muss mehrmals auf diesem Pfad vorkommen.

Deshalb sieht der Ableitungsbaum für $S \stackrel{*}{\Rightarrow} z$ so aus:



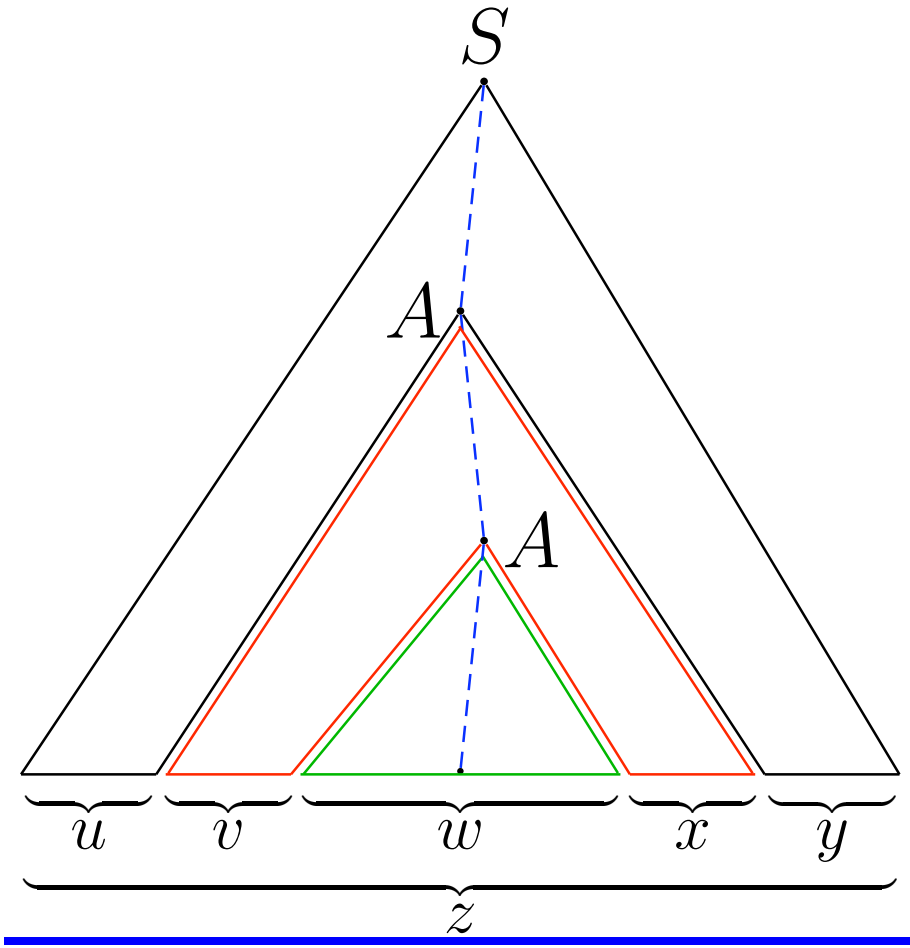
Auf dem *blauen Pfad* kommt A mindestens zweimal vor. u, v, x, y sind die Teilwörter von z, die links bzw. rechts des ersten bzw. zweiten A entstehen. w ist das Wort, das aus dem zweiten A entsteht. Es gilt also

$$S \stackrel{*}{\Rightarrow} uAy$$
$$A \stackrel{+}{\Rightarrow} vAx$$
$$A \stackrel{*}{\Rightarrow} w$$

und damit insgesamt

$$S \stackrel{*}{\Rightarrow} uAy \stackrel{+}{\Rightarrow} uvAxy \stackrel{*}{\Rightarrow} uvwxy = z$$

Jetzt entfernen wir das rote Stück aus dem Ableitungsbaum, und erhalten einen Ableitungsbaum für ein neues Wort z', aus dem mindestens eine Wiederholung eines Nichtterminalzeichens entfernt ist.



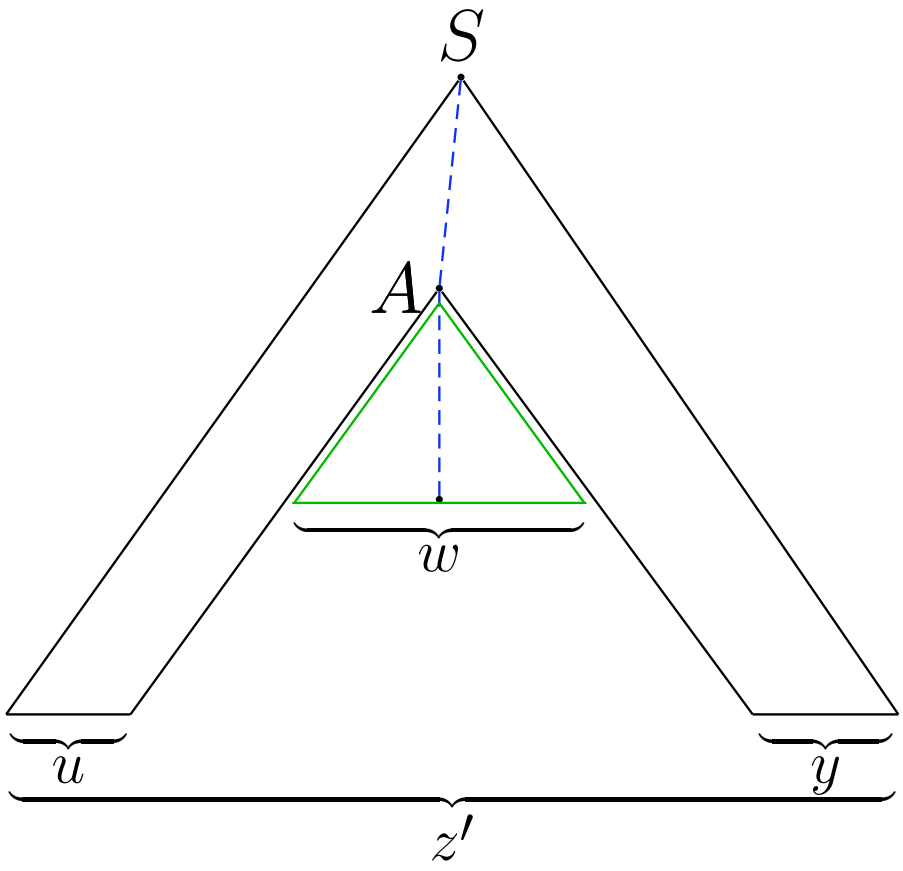
Eine zugehörige Ableitung ist

$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uwy = z'$.

Indem man dieses Verfahren wiederholt, erhält man irgendwann einen Ableitungsbaum, dessen Pfade *keine* Wiederholungen von Nichtterminalzeichen mehr enthalten.

Der hat dann höchstens die Höhe $|N|$. □

Jetzt entfernen wir das rote Stück aus dem Ableitungsbaum, und erhalten einen Ableitungsbaum für ein neues Wort z', aus dem mindestens eine Wiederholung eines Nichtterminalzeichens entfernt ist.



Eine zugehörige Ableitung ist

$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uwy = z'$.

Indem man dieses Verfahren wiederholt, erhält man irgendwann einen Ableitungsbaum, dessen Pfade *keine* Wiederholungen von Nichtterminalzeichen mehr enthalten.

Der hat dann höchstens die Höhe $|N|$. □

Für einige andere Fragestellungen über kontextfreie Grammatiken gibt es *keine* Entscheidungsalgorithmen, z.B. für das *Äquivalenzproblem*:

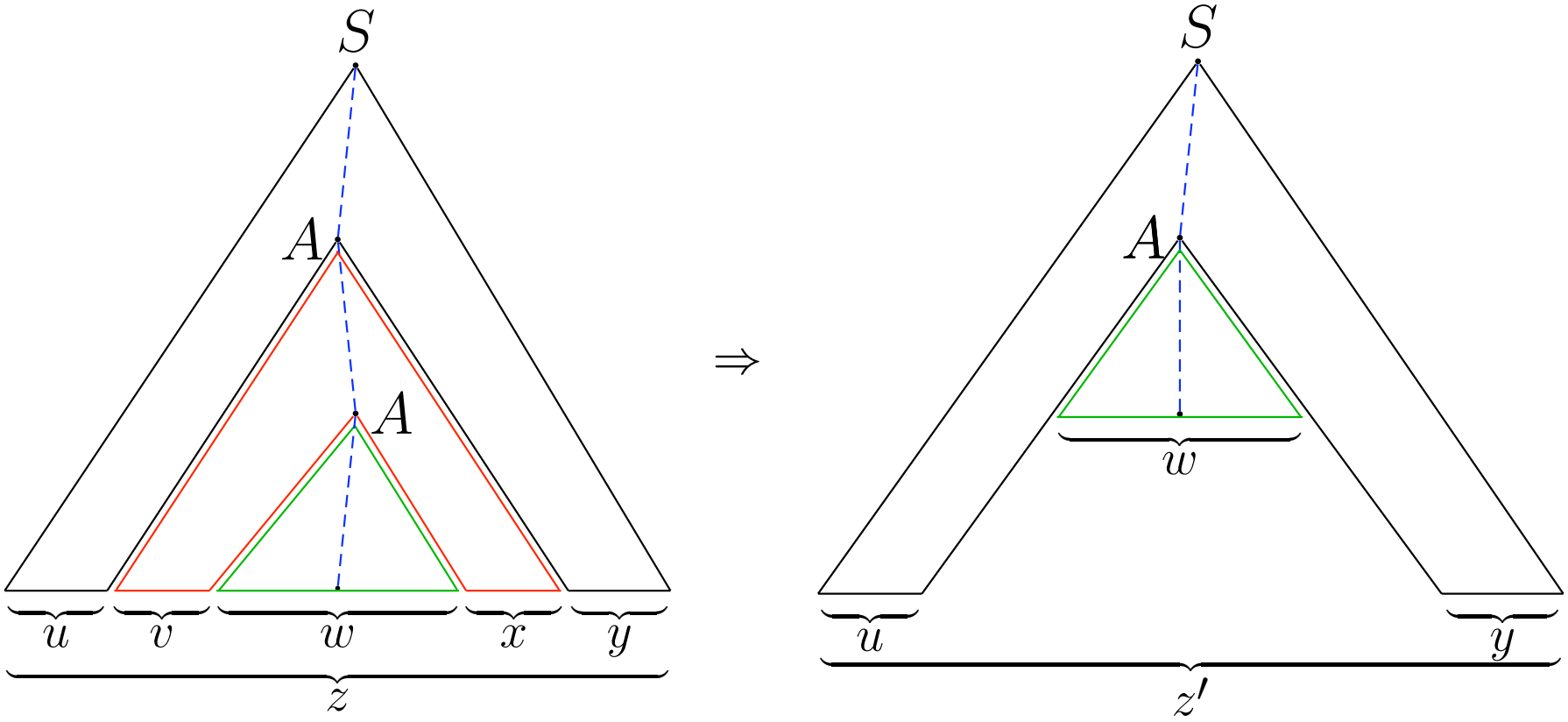
Eingabe: Zwei kontextfreie Grammatiken G_1 und G_2.
Frage: Ist $L(G_1) = L(G_2)$?

Es gibt *keinen* Algorithmus, der diese Frage stets korrekt beantwortet.

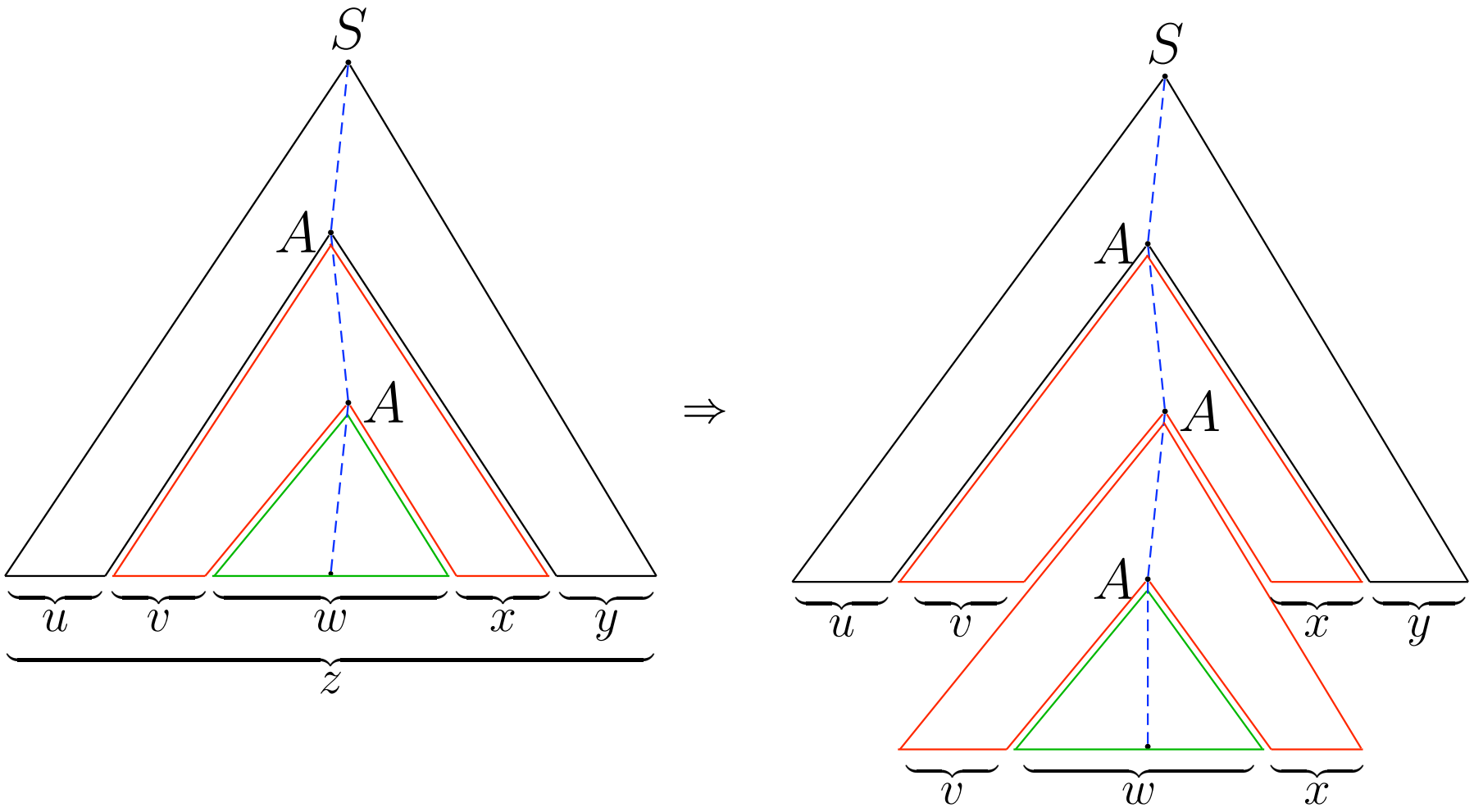
Solche Ergebnisse können wir zur Zeit noch nicht beweisen.
Sie gehören zur *Berechenbarkeitstheorie* (Teil II der Vorlesung).

Grenzen kontextfreier Sprachen

Im Beweis zu Satz 2.27 haben wir aus einem 'hinreichend großen' Ableitungsbaum ein Stück herausgeschnitten.



Stattdessen kann man das Stück auch wiederholen:



So erhält man aus der ursprünglichen Ableitung

$$S \overset{*}{\Rightarrow} uAy \overset{+}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy$$

neue Ableitungen der Form

$$S \overset{*}{\Rightarrow} uAy \overset{+}{\Rightarrow} \ldots \overset{+}{\Rightarrow} uv^iAx^iy \overset{*}{\Rightarrow} uv^iwx^iy$$

indem man i-mal die Ableitung $A \overset{+}{\Rightarrow} vAx$ wiederholt.

Das ist die Beweisidee für

Satz 2.28 (Pumping Lemma für kontextfreie Sprachen) *Sei $L \subseteq \Sigma^*$ kontextfrei. Dann existiert eine Zahl $n \geq 1$, so dass für jedes $z \in L$ mit $|z| \geq n$ gilt: Es gibt eine Zerlegung $z = uvwxy$ mit $vx \neq \varepsilon$ und $uv^iwx^iy \in L$ für alle $i \geq 0$.*

Beweis:

Sei $L = L(G)$, wobei $G = (\Sigma, N, S, P)$ eine Grammatik ist, die mit den Konstruktionen aus Satz 2.25 und Satz 2.26 entstanden ist, und sei $p = \max\{\,|\gamma| \mid (A \to \gamma) \in P\}$.

Dann hat das Blattwort eines Ableitungsbaums der Höhe m höchstens die Länge p^m (weil der Ableitungsbaum von einer Ebene zur nächsten höchstens um den *Faktor* p breiter werden kann).

Wir wählen $n = p^{|N|} + 1 > p^{|N|}$.

Dann hat jeder Ableitungsbaum für ein Wort z mit $|z| \geq n$ mindestens die Höhe $|N| + 1$, also ist die Ableitung für z von der Form

$$S \overset{*}{\Rightarrow} uAy \overset{+}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy$$

und wir erhalten daraus Ableitungen

$$S \overset{*}{\Rightarrow} uAy \overset{+}{\Rightarrow} \ldots \overset{+}{\Rightarrow} uv^iAx^iy \overset{*}{\Rightarrow} uv^iwx^iy$$

für alle Wörter uv^iwx^iy mit $i \geq 0$. Dabei gilt $vx \neq \varepsilon$, weil $A \overset{+}{\Rightarrow} \varepsilon A\varepsilon = A$ ohne Einheits- und ε-Produktionen nicht möglich ist. $\square$

Korollar 2.29 *Zum Nachweis, dass L nicht kontextfrei ist, genügt es zu zeigen: Für jedes $n \geq 1$ existiert ein $z \in L$ mit $|z| \geq n$, so dass für alle Zerlegungen $z = uvwxy$ mit $vx \neq \varepsilon$ gilt: Es gibt ein $i \geq 0$ mit $uv^iwx^iy \notin L$.*

Beispiel:

Die Sprache $L = \{a^nb^nc^n \mid n \geq 0\}$ ist nicht kontextfrei, denn:
Sei $n \geq 1$.
Wir wählen $z = a^nb^nc^n$, also $z \in L$ und $|z| \geq n$.
Sei nun $z = uvwxy$ eine beliebige Zerlegung von z mit $vx \neq \varepsilon$.
Wenn v oder x mehr als eines der Zeichen a, b, c enthält, dann ist $uv^2wx^2y \notin L$, weil in diesem Wort die Zeichen a, b, c nicht mehr in der richtigen Reihenfolge stehen.
Wenn $v = a^k$ und $x = b^l$, dann sind k und l nicht beide 0, also ist $uv^2wx^2y \notin L$, weil dieses Wort mehr as als cs oder mehr bs als cs enthält.

Alle anderen Fälle (z.B. $v = a^k$ und $x = a^l$) verlaufen analog, weil mindestens eines der Zeichen a, b, c in vx fehlt. Beim Pumpen bleibt die Anzahl dieses Zeichens gleich, während sich die Anzahl eines der anderen beiden Zeichen erhöht.

Korollar 2.30 *Der Durchschnitt zweier kontextfreier Sprachen ist im allgemeinen nicht kontextfrei.*

Beweis:

Sei $L_1 = \{a^n b^n \mid n \geq 0\} \circ \{c\}^*$

und $L_2 = \{a\}^* \circ \{b^n c^n \mid n \geq 0\}$.

L_1 und L_2 sind kontextfrei, weil sie beide durch Konkatenation aus kontextfreien Sprachen entstehen,

aber $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ ist *nicht* kontextfrei. □

Auch das Pumping Lemma für kontextfreie Sprachen lässt sich (wie das für reguläre Sprachen) so verstärken, dass es besser einsetzbar ist (vgl. Übung 11, Aufgabe 3).

Satz 2.31 (Starkes Pumping Lemma für kontextfreie Sprachen)
Sei $L \subseteq \Sigma^*$ kontextfrei. Dann existiert eine Zahl $n \geq 1$, so dass für jedes $z \in L$ mit $|z| \geq n$ gilt: Es gibt eine Zerlegung $z = uvwxy$ mit $vx \neq \varepsilon$, $|vwx| \leq n$ und $uv^iwx^iy \in L$ für alle $i \geq 0$.

Beweis:

Um die Abschätzung $|vwx| \leq n$ zu erhalten, muss man im Beweis von Satz 2.28

- die Zahl n etwas größer wählen,

- und das Nichtterminalzeichen A sorgfältiger auswählen.

Sei $n = p^{|N|+1}$ und $z \in \Sigma^*$ mit $|z| \geq n$.

Dann wählen wir im Ableitungsbaum für z einen Pfad *maximaler Länge* (von der Wurzel S zu einem Blatt).

Da n sogar größer gewählt ist als im Beweis von Satz 2.28, muss auf diesem Pfad mindestens ein Nichtterminalzeichen mehrmals vorkommen.

Wir wählen das Nichtterminalzeichen A aus, das sich vom Blatt aus gesehen als *erstes* wiederholt.

Diese erste Wiederholung passiert (wenn man wieder vom Blatt ausgeht) nach spätestens $|N|+1$ Schritten, d.h. der Teilpfad vom Blatt zu diesem Zeichen A hat höchstens die Länge $|N|+1$.

Da der Pfad von S zum Blatt maximal gewählt war, kann es von diesem A aus keinen Pfad zu einem anderen Blatt geben, der länger ist als $|N|+1$.

Also hat das Wort vwx, das ja aus diesem Zeichen A entsteht, höchstens die Länge $p^{|N|+1} = n$. □

Beispiel:

Die Sprache $L = \{ww \mid w \in \{a,b\}^*\}$ ist nicht kontextfrei (im Gegensatz zur Sprache $L' = \{ww^R \mid w \in \{a,b\}^*\}$).

Sei $n \geq 1$. Wir wählen $z = a^n b^n a^n b^n$, also $z \in L$ und $|z| = 4n \geq n$.

Mit dem schwachen Pumping Lemma (Satz 2.28) kann man hier nicht argumentieren. Wenn nämlich $u = \varepsilon$, $v = x = a^n$ und $w = y = b^n$, so ist $z = uvwxy$ und für *alle* $i \geq 0$ gilt $uv^i wx^i y = a^{in} b^n a^{in} b^n \in L$.

Aber mit dem starken Pumping Lemma kommt man zum Ziel:

Wenn $z = uvwxy$ mit $|vwx| \leq n$ und $vx \neq \varepsilon$, so unterscheiden wir die folgenden drei Fälle:

1. vwx beginnt im linken a^n: Dann liegt vwx in der linken Worthälfte $a^n b^n$.
2. vwx beginnt im linken b^n: Dann liegt vwx im Mittelstück $b^n a^n$.
3. vwx liegt in der rechten Worthälfte $a^n b^n$.

In allen drei Fällen betrachten wir $z_0 = uv^0wx^0y$. Dieses Wort entsteht aus $z = uvwxy$, indem man v und x (also einen Teil von vwx) entfernt. Deshalb hat es in den drei oben genannten Fällen die Form $a^kb^la^nb^n$ oder $a^nb^ka^lb^n$ oder $a^nb^na^kb^l$, wobei $k < n$ oder $l < n$ (oder beides) gilt. All diese Wörter sind offensichtlich *nicht* von der Form ww, also gilt in jedem Fall $z_0 \notin L$, d.h. L ist nicht kontextfrei. □

Das Beispiel erklärt, warum (die meisten) Programmiersprachen nicht kontextfrei sind. Üblicherweise müssen die Programme nämlich sogenannte 'Kontextbedingungen' erfüllen, z.B.: "Jede Prozedur muss deklariert sein, bevor sie aufgerufen wird."
Um eine solche Bedingung zu überprüfen, muss man den Prozedurnamen an der Deklarationsstelle mit dem Prozedurnamen an der Aufrufstelle vergleichen (so wie man in L die erste mit der zweiten Worthälfte vergleichen muss). Solche Vergleiche sind—wie wir an der Sprache L gesehen haben—mit einer kontextfreien Grammatik nicht möglich.

Kontextfreie Sprachen und Kellerautomaten

Trotzdem spielen kontextfreie Grammatiken in der Praxis eine wichtige Rolle. Bei der Definition einer Programmiersprache L geht man nämlich so vor:

Zunächst definiert man die sogenannte 'kontextfreie Syntax' von L, d.h. man definiert eine kontextfreie Sprache $L' \supseteq L$. Dann definiert man L als die Menge aller Wörter aus L', die gewisse 'Kontextbedingungen' (wie die oben genannte Bedingung, dass jede Prozedur deklariert sein muss, bevor man sie aufruft) erfüllen.

Ganz analog geht man bei der Implementierung der Programmiersprache vor: Durch die *syntaktische Analyse* (die vom *Parser* durchgeführt wird) wird überprüft, ob die vom Programmierer eingegebene Zeichenreihe der kontextfreien Syntax genügt, d.h. ob sie in L' liegt. Wenn ja, so erzeugt der Parser den Syntaxbaum für die Zeichenreihe. Bei der anschließenden *semantischen Analyse* werden dann am Syntaxbaum die *Kontextbedingungen* überprüft.

Weitere Beispiele:

Sei $\Sigma = \{a\}$. Dann sind die früher betrachteten Sprachen

- $\{a^{n^2} \mid n \geq 0\}$
- $\{a^{2^n} \mid n \geq 0\}$
- $\{a^p \mid p \text{ Primzahl}\}$

nicht kontextfrei.

Beweis:
Mit Hilfe des Pumping Lemmas für reguläre Sprachen haben wir bewiesen, dass diese Sprachen nicht regulär sind. Da es bei einem einelementigen Alphabet keinen Unterschied macht, ob man an *einer* oder an *zwei* Stellen pumpt, lassen sich diese Beweise leicht so abändern, dass man in ihnen das Pumping Lemma für kontextfreie Sprachen benutzt. Also sind diese Sprachen auch nicht kontextfrei. □

In der Tat gilt über jedem *einelementigen* Alphabet:

$$\mathcal{L}_{reg} = \mathcal{L}_{kf}$$

Das lässt sich aber nicht mit dem Pumping Lemma beweisen, denn in beiden Fällen (sowohl bei regulären, als auch bei kontextfreien Sprachen) gibt das Pumping Lemma nur eine notwendige und keine hinreichende Bedingung an.

Wir verzichten auf den Beweis dieser Gleichheit (und verwenden sie auch nicht).

Über jedem Alphabet mit mindestens zwei Zeichen gilt natürlich

$$\mathcal{L}_{reg} \subsetneq \mathcal{L}_{kf}$$

wie wir bereits bewiesen haben.

Deterministisch kontextfreie Sprachen

Die syntaktische Analyse von Programmen wird (im Prinzip) mit Kellerautomaten durchgeführt. Wir haben bereits gesehen, wie man aus einer kontextfreien Grammatik G einen Kellerautomaten M mit $L(M) = L(G)$ erhält, aber dieser Kellerautomat war (in hohem Maße) nichtdeterministisch. Für die Praxis benötigt man natürlich "deterministische" Kellerautomaten. Dieser Begriff muss nun erst einmal definiert werden. Es genügt sicherlich *nicht*, zu fordern, dass die Übergangsrelation Δ des Kellerautomaten M eine (partielle) Funktion ist. Man betrachte z.B.

$$\Delta = \{((p, a, \varepsilon), \ldots), ((p, \varepsilon, b), \ldots)\}$$

Diese Relation Δ ist eine partielle Funktion, aber der zugehörige Kellerautomat hat trotzdem die Auswahl zwischen zwei Transitionen, wenn die aktuelle Eingabe mit a beginnt und der aktuelle Kellerinhalt mit b.

Solche "Konflikte" zwischen zwei Transitionen sollten in einem deterministischen Kellerautomaten ausgeschlossen sein. Das erreicht man durch die folgende

Definition 2.32 *Sei $M = (\Sigma, \Gamma, Q, s, F, \Delta)$ ein Kellerautomat.*

1. *Zwei Wörter α_1, α_2 heißen* konsistent*, wenn α_1 Präfix von α_2 oder α_2 Präfix von α_1 ist.*

2. *Zwei Transitionen $((p_1, u_1, \beta_1), (q_1, \gamma_1))$ und $((p_2, u_2, \beta_2), (q_2, \gamma_2))$ heißen* kompatibel*, wenn gilt:*

   - $p_1 = p_2$,

   - *u_1, u_2 sind konsistent und*

   - *β_1, β_2 sind konsistent*

   Ansonsten heißen sie inkompatibel*.*

3. *M heißt* deterministisch*, wenn die Transitionen in Δ paarweise inkompatibel sind.*

Kontextfreie Sprachen und Kellerautomaten

Man beachte, dass nur kompatible Transitionen $((p_1, u_1, \beta_1), (q_1, \gamma_1))$ und $((p_2, u_2, \beta_2), (q_2, \gamma_2))$ auf die gleiche Konfiguration (p, u, β) anwendbar sein können, denn dazu muss gelten:

- $p_1 = p_2 = p$,
- u_1 und u_2 konsistent, weil beides Präfixe von u sind,
- β_1 und β_2 konsistent, weil beides Präfixe von β sind.

Also sind zwei Transitionen eines deterministischen Kellerautomaten niemals auf die gleiche Konfiguration anwendbar, d.h. für einen deterministischen Kellerautomaten M ist die Relation $\vdash_M$ eine partielle Funktion.

Im Prinzip definieren wir jetzt die deterministisch kontextfreien Sprachen als diejenigen, die von deterministischen Kellerautomaten erkannt werden, wobei wir den Automaten aber noch die Möglichkeit geben, das Ende des Eingabeworts zu erkennen.

Definition 2.33 *Eine Sprache* $L \subseteq \Sigma^*$ *heißt* deterministisch kontextfrei, *wenn es einen deterministischen Kellerautomaten* M *gibt, der die Sprache*

$$L\$ = \{w\$ \mid w \in L\}$$

erkennt, wobei \$ *ein neues Zeichen ist, d.h.* $\$ \notin \Sigma$.

(In der Praxis, d.h. bei einem Parser, hat man auch eine solche Markierung für das Ende der Eingabe, nämlich das Zeichen 'end of file'.)

Beispiel:

Die Sprache $L = \{a^n b^n \mid n \geq 0\}$ ist deterministisch kontextfrei.

Ein deterministischer Kellerautomat, der $L\$$ erkennt, arbeitet wie folgt:

Wenn er im Startzustand schon das Zeichen \$ liest, geht er sofort in den Endzustand.

Wenn er im Startzustand ein a liest, geht er in einen Zustand q_a, in dem weitere as gelesen werden können. Diese as werden in den Keller verschoben.

Sobald er das erste b liest, wechselt er in einen Zustand q_b, in dem nur noch bs (oder \$) gelesen werden dürfen.

Die bs werden gegen die as im Keller aufgehoben.

Sobald \$ erscheint, wechselt der Automat in den Endzustand.

Wenn gleich viele as und bs gelesen wurden, ist der Keller am Ende leer, also wird das Wort akzeptiert.

Andernfalls bleibt der Automat entweder stecken (d.h. das Eingabeband ist nicht leer) oder der Keller ist am Ende nicht leer, also wird das Wort nicht akzeptiert.

Formale Definition dieses Kellerautomaten:

$M = (\{a, b, \$\}, \{a\}, \{s, q_a, q_b, f\}, s, \{f\}, \Delta)$ mit:

$$
\begin{aligned}
\Delta = \{\ & ((s, \$, \varepsilon), (f, \varepsilon)), && (1)\\
& ((s, a, \varepsilon), (q_a, \varepsilon)), && (2)\\
& ((q_a, a, \varepsilon), (q_a, a)), && (3)\\
& ((q_a, b, \varepsilon), (q_b, \varepsilon)), && (4)\\
& ((q_b, b, a), (q_b, \varepsilon)), && (5)\\
& ((q_b, \$, \varepsilon), (f, \varepsilon))\ \} && (6)
\end{aligned}
$$

Wir zeigen, dass tatsächlich $L(M) = L\$ = \{a^n b^n \$ \mid n \geq 0\}$ gilt.

'$\supseteq$': Sei $w = a^n b^n \$$.

Im Falle $n = 0$ gilt $(s, w, \varepsilon) = (s, \$, \varepsilon) \vdash_{(1)} (f, \varepsilon, \varepsilon)$, also $w \in L(M)$.

Im Falle $n > 0$ gilt:

$$\begin{aligned}
(s, w, \varepsilon) \;=\; (s, a^n b^n \$, \varepsilon) &\vdash_{(2)} (q_a, a^{n-1} b^n \$, \varepsilon) \\
&\vdash_{(3)}^{n-1} (q_a, b^n \$, a^{n-1}) \\
&\vdash_{(4)} (q_b, b^{n-1} \$, a^{n-1}) \\
&\vdash_{(5)}^{n-1} (q_b, \$, \varepsilon) \\
&\vdash_{(6)} (f, \varepsilon, \varepsilon)
\end{aligned}$$

'$\subseteq$': Sei $w \in L(M)$, d.h. $(s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)$.

Wir überlegen uns, wie diese Folge von Übergangsschritten aussehen kann.

An den Transitionen sieht man zunächst, dass nur die folgenden Zustandsübergänge möglich sind:

- von s mit (1) nach f oder mit (2) nach q_a,
- von q_a mit (3) nach q_a oder mit (4) nach q_b,
- von q_b mit (5) nach q_b oder mit (6) nach f.

Also ist die gesamte Schrittfolge von der Form

- $(s, w, \varepsilon) \vdash_{(1)} (f, \varepsilon, \varepsilon)$ oder
- $(s, w, \varepsilon) \vdash_{(2)} (q_a, w_1, \beta_1) \vdash_{(3)}^{m} (q_a, w_2, \beta_2)$
  $\vdash_{(4)} (q_b, w_3, \beta_3) \vdash_{(5)}^{n} (q_b, w_4, \beta_4)$
  $\vdash_{(6)} (f, \varepsilon, \varepsilon)$

  mit $m, n \geq 0$

Im ersten Fall ist $w = \$ = a^0 b^0 \$ \in L\$$, und im zweiten Fall erhalten wir wegen der Form der Transitionen:

- $w = aw_1$ und $\beta_1 = \varepsilon$
- $w_1 = a^m w_2$ und $\beta_2 = a^m$
- $w_2 = bw_3$ und $\beta_3 = \beta_2 = a^m$
- $w_3 = b^n w_4$ mit $n \leq m$ und $\beta_4 = a^{m-n}$
- $w_4 = \$$ und $\beta_4 = \varepsilon$, also $m = n$

Insgesamt ergibt sich daraus $w = a^{n+1} b^{n+1} \$ \in L\$$. □

Lemma 2.34 *Jede deterministisch kontextfreie Sprache ist kontextfrei.*

Beweis:

Wenn $L \subseteq \Sigma^*$ deterministisch kontextfrei ist, dann existiert ein Kellerautomat $M = (\Sigma \cup \{\$\}, \Gamma, Q, s, F, \Delta)$, der $L\$$ erkennt.

Das bedeutet aber zunächst nur, dass $L\$$ kontextfrei ist.

Sei nun $M' = (\Sigma, \Gamma, Q \times \{0,1\}, (s,0), F \times \{1\}, \Delta')$ mit

$$\begin{aligned}
\Delta' &= \{(((p,0),u,\beta),((q,0),\gamma)) \mid ((p,u,\beta),(q,\gamma)) \in \Delta\} && (1)\\
&\cup \{(((p,0),u,\beta),((q,1),\gamma)) \mid ((p,u\$,\beta),(q,\gamma)) \in \Delta\} && (2)\\
&\cup \{(((p,1),\varepsilon,\beta),((q,1),\gamma)) \mid ((p,\varepsilon,\beta),(q,\gamma)) \in \Delta\} && (3)
\end{aligned}$$

Dann gilt $L = L(M')$, denn:

'$\subseteq$': Sei $w \in L$, also $w\$ \in L\$ = L(M)$.

Dann existiert ein $f \in F$ mit $(s, w\$, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)$

Wenn wir diese Folge von Übergangsschritten dort aufteilen, wo das Ende der Eingabe gelesen wird, dann hat sie die Form

$$(s, w\$, \varepsilon) \vdash_M^* (p, u\$, \beta) \vdash_M (q, \varepsilon, \gamma) \vdash_M^* (f, \varepsilon, \varepsilon)$$

Daraus folgt per Definition von Δ'

$$((s,0),w,\varepsilon) \vdash^*_{(1)} ((p,0),u,\beta) \vdash_{(2)} ((q,1),\varepsilon,\gamma) \vdash^*_{(3)} ((f,1),\varepsilon,\varepsilon)$$

also $w \in L(M')$.

'$\supseteq$': Sei $w \in L(M')$, d.h. es existiert ein $f \in F$ mit

$$((s,0),w,\varepsilon) \vdash^*_{M'} ((f,1),\varepsilon,\varepsilon)$$

Da man nur mit (2) von 0 zu 1 wechseln kann, und da man mit (3) keine Eingabe mehr lesen kann, muss diese Folge so aussehen:

$$((s,0),w,\varepsilon) \vdash^*_{(1)} ((p,0),u,\beta) \vdash_{(2)} ((q,1),\varepsilon,\gamma) \vdash^*_{(3)} ((f,1),\varepsilon,\varepsilon)$$

Daraus folgt nun umgekehrt mit der Definition von Δ'

$$(s,w\$,\varepsilon) \vdash^*_M (p,u\$,\beta) \vdash_M (q,\varepsilon,\gamma) \vdash^*_M (f,\varepsilon,\varepsilon)$$

also $w\$ \in L(M) = L\$$ und damit $w \in L$. $\square$

Satz 2.35 *Die Klasse $\mathcal{L}_{dkf}$ der deterministisch kontextfreien Sprachen ist abgeschlossen unter Komplement.*

Beweisidee:

Man vertauscht die Endzustände mit den Nicht-Endzuständen (wie beim DEA).

Das geht aber nicht gut, weil ein Kellerautomat (auch ein deterministischer) viele Möglichkeiten hat, ein Wort 'abzulehnen':

Er kann

- mit nichtleerem Keller halten,

- bei der Berechnung 'steckenbleiben', d.h. in eine Konfiguration geraten, in der das Eingabewort noch nicht abgearbeitet, aber keine Transition mehr anwendbar ist,

- in eine Endlosschleife geraten, ohne dass er das Eingabewort abgearbeitet hat.

Zum Beweis von Satz 2.35 muss man diese Möglichkeiten erst einmal ausschließen,

d.h. man muss zeigen, dass für jede deterministisch kontextfreie Sprache ein "normalisierter" Kellerautomat existiert, der seine Eingabe immer vollständig abarbeitet und den Keller immer leer macht.

Deshalb verzichten wir hier auf den Beweis. □

Wir benutzen Satz 2.35, um eine kontextfreie Sprache zu finden, die *nicht* deterministisch kontextfrei ist.

Mit anderen Worten:
Wir zeigen, dass die Klasse $\mathcal{L}_{dkf}$ der deterministisch kontextfreien Sprachen *echt* zwischen den Klassen $\mathcal{L}_{reg}$ und $\mathcal{L}_{kf}$ liegt.
(Jede reguläre Sprache ist deterministisch kontextfrei, weil man einen DEA als deterministischen Kellerautomaten auffassen kann, der den Keller nicht benutzt.)

Korollar 2.36 *Es gibt eine kontextfreie Sprache, die nicht deterministisch kontextfrei ist.*

Beweis:

Sei $L = \{a^l b^m c^n \mid l \neq m \text{ oder } m \neq n\}$.

L ist kontextfrei. Um dies einzusehen, kann man L in der Form

$$L = \{a^l b^m \mid l \neq m\} \circ \{c\}^* \cup \{a\}^* \circ \{b^m c^n \mid m \neq n\}$$

schreiben und für die Sprachen $\{a^l b^m \mid l \neq m\}$ und $\{b^m c^n \mid m \neq n\}$ kontextfreie Grammatiken angeben.

Wäre L deterministisch kontextfrei, dann wäre nach Satz 2.35 auch das Komplement $\bar{L} = \{a, b, c\}^* \setminus L$ (deterministisch) kontextfrei, und damit wäre nach Satz 2.23 auch $\bar{L} \cap \{a\}^*\{b\}^*\{c\}^*$ kontextfrei, weil $\{a\}^*\{b\}^*\{c\}^*$ regulär ist.

Aber es gilt $\bar{L} \cap \{a\}^*\{b\}^*\{c\}^* = \{a^l b^m c^n \mid l = m \text{ und } m = n\} = \{a^n b^n c^n \mid n \geq 0\}$, und diese Sprache ist *nicht* kontextfrei, wie bereits gezeigt wurde. □

Kontextfreie Sprachen und Kellerautomaten

Es sei noch erwähnt, dass die anderen "üblichen" Abschlusseigenschaften (Vereinigung, Konkatenation, Kleene-Abschluss) für $\mathcal{L}_{dkf}$ *nicht* gelten.

Für die Vereinigung erhalten wir leicht ein Gegenbeispiel:

Sei $L_1 = \{a^n b^n \mid n \geq 0\} \circ \{c\}^*$ und $L_2 = \{a\}^* \circ \{b^n c^n \mid n \geq 0\}$.

L_1 und L_2 sind deterministisch kontextfrei (man konstruiere deterministische Kellerautomaten ähnlich wie für $\{a^n b^n \mid n \geq 0\}$).

Also sind nach Satz 2.35 auch ihre Komplemente $\bar{L}_1$ und $\bar{L}_2$ deterministisch kontextfrei.

Wäre $\mathcal{L}_{dkf}$ abgeschlossen unter Vereinigung, dann wäre $\bar{L}_1 \cup \bar{L}_2$ deterministisch kontextfrei,

also nach Satz 2.35 auch das Komplement von $\bar{L}_1 \cup \bar{L}_2$.

Das ist aber die Sprache $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$, von der wir wissen, dass sie noch nicht einmal kontextfrei ist.

Teil II: Berechenbarkeit (und Komplexität)

Hier: Nur **Berechenbarkeitstheorie**

Zentrale Fragestellungen:

Welche Probleme lassen sich mit Hilfe von Algorithmen (Computer-Programmen) lösen bzw. nicht lösen?

Welche Funktionen lassen sich mit ihnen berechnen bzw. nicht berechnen?

In der Berechenbarkeitstheorie geht es um die *prinzipielle* Berechenbarkeit, d.h. Zeit- und Speicherplatzbedarf werden ignoriert.

Betrachtungen über Zeitaufwand und Platzbedarf sind Thema der Komplexitätstheorie.

Interessant sind in der Berechenbarkeitstheorie deshalb vor allem die *negativen* Ergebnisse.

Wir hatten schon **Beispiele:**

1. Das Äquivalenzproblem für DEAs ist algorithmisch lösbar (entscheidbar), d.h. es gibt einen Algorithmus, der

   als Eingabe zwei DEAs A_1 und A_2 erhält,

   als Ausgabe die korrekte Antwort auf die Frage

   $$L(A_1) = L(A_2)?$$

   liefert.

2. Das Äquivalenzproblem für KFGs ist *nicht* algorithmisch lösbar (unentscheidbar), d.h. es gibt *keinen* Algorithmus, der

   als Eingabe zwei KFGs G_1 und G_2 erhält,

   als Ausgabe die korrekte Antwort auf die Frage

   $$L(G_1) = L(G_2)?$$

   liefert.

Zum Beweis der Entscheidbarkeit bzw. Berechenbarkeit eines Problems braucht man 'nur' einen Algorithmus anzugeben.

Aber wie beweist man, dass ein Problem *nicht* entscheidbar bzw. eine Funktion *nicht* berechenbar ist?

Gibt es überhaupt solche Probleme bzw. Funktionen?
(Wir haben es noch nicht bewiesen.)

Dazu muss man zunächst die Bedeutung der Begriffe *Berechenbarkeit*, *Entscheidbarkeit*, ... genauer festlegen

Intuition:

Eine Funktion heißt berechenbar, wenn es einen Algorithmus gibt, der sie berechnet.

Folgende Fragen sind zu klären:

1. Über welche Art von Funktionen sprechen wir hier?

   Partielle oder totale Funktionen?

   Funktionen auf Zahlen, Wörtern, . . . ?

   (d.h. was ist als Eingabe bzw. Ausgabe für die Algorithmen zugelassen?)

2. Was versteht man unter einem Algorithmus?

3. Was versteht man unter der von einem Algorithmus berechneten Funktion?

Antworten:

1. Wir benutzen hier den üblichen Ansatz, d.h. wir lassen *partielle* Funktionen zu. (Schreibweise: $f : A \hookrightarrow B$)

   Warnung: In einem Teil der Literatur beschränkt man sich auf totale Funktionen. Diesen Ansatz benutzen wir nicht, weil er an manchen Stellen der Theorie zu einer umständlichen Ausdrucksweise führt.

   Wir betrachten zunächst Funktionen auf natürlichen Zahlen, d.h. Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$. Später werden wir auch Funktionen auf Wörtern betrachten, d.h. Funktionen $f : (\Sigma^*)^k \hookrightarrow \Sigma^*$ über beliebigem Alphabet Σ.

2. Unter einem Algorithmus verstehen wir (zunächst) ein Programm in einer 'hinreichend mächtigen' Programmiersprache.

3. Die von einem Algorithmus berechnete Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ wird gleich definiert.

Bemerkungen:

Es kommt nicht darauf an, ob man Funktionen auf Zahlen oder auf Wörtern benutzt, denn:

Einerseits lassen sich Zahlen als Wörter darstellen (dezimal, binär, unär).

Andererseits lassen sich Wörter als Zahlen codieren:

Die Wörter über einem Alphabet Σ lassen sich 'durchnummerieren', z.B. indem man zuerst die Wörter der Länge 0, dann die der Länge 1, . . . , jeweils in alphabetischer Reihenfolge aufzählt (vgl. Übung 4, Aufgabe 1 a).

So erhält man eine Folge von Wörtern $w_0, w_1, \ldots$, in der *alle* Wörter aus Σ^* enthalten sind. Die Zahl n kann man dann als *Codierung* des Wortes w_n betrachten.

Wie passen unsere früheren Beispiele in diesen Rahmen (Algorithmen, die als Eingabe DEAs oder KFGs erhalten)?

Ein DEA oder eine KFG besitzt eine *endliche Darstellung*, lässt sich also als Wort über einem geeigneten Alphabet auffassen.

Zum Vergleich:

Algorithmen, die eine (beliebige) Sprache als Eingabe erhalten oder die auf reellen Zahlen arbeiten, passen *nicht* in unseren Rahmen, weil Sprachen oder reelle Zahlen keine endliche Darstellung besitzen.

(Sie können keine endliche Darstellung besitzen, denn die Menge der Sprachen über einem Alphabet Σ oder die Menge der reellen Zahlen sind *überabzählbar*, vgl. Übung 4, Aufgabe 1 b bzw. Übung 1, Aufgabe 1).

Definition 3.1 *Sei $k \geq 0$, $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ eine partielle Funktion und π ein Algorithmus, d.h. ein Programm über einer geeigneten Programmiersprache.*

1. *π berechnet die Funktion f, wenn für alle $(n_1, \ldots, n_k) \in \mathbb{N}^k$ gilt:*

   π hält genau dann für die Eingabe $(n_1, \ldots, n_k)$, wenn $f(n_1, \ldots, n_k)$ definiert ist, und liefert in diesem Fall die Ausgabe $f(n_1, \ldots, n_k)$.

2. *f heißt berechenbar, wenn es ein Programm π gibt, das f berechnet.*

Die 'geeignete(n) Programmiersprache(n)' definieren wir später. Dann wird (jeweils) genau festgelegt, wie ein Programm π seine Eingabe erhält, und wo es seine Ausgabe abliefert. Manche dieser Programmiersprachen arbeiten auf Wörtern statt auf Zahlen. Dann tritt Σ^* (mit beliebigem Alphabet Σ) an die Stelle von $\mathbb{N}$.

Beispiele:

Beispiele für berechenbare Funktionen sind leicht zu finden, etwa

1. Die Funktion

$$exp : \mathbb{N}^2 \to \mathbb{N}$$

$$(m, n) \mapsto m^n$$

   ist berechenbar (durch irgendein Programm zum Potenzieren).

2. Die *leere* Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$, die für alle $(n_1, \ldots, n_k) \in \mathbb{N}^k$ undefiniert ist, ist berechenbar durch ein Programm, das für *keine* Eingabe $(n_1, \ldots, n_k)$ hält.

3. Jede konstante Funktion

$$const_m : \mathbb{N}^k \to \mathbb{N}$$

$$(n_1, \ldots, n_k) \mapsto m$$

   ist berechenbar durch ein Programm, das unabhängig von der Eingabe immer hält und die Zahl m zurückliefert.

Ein konkretes Gegenbeispiel (mit Beweis) können wir noch nicht angeben, aber wir können schon zeigen, dass es welche gibt.

Satz 3.2 *Es gibt Funktionen (partielle oder totale), die nicht berechenbar sind.*

Beweis:

Programme π sind Wörter über einem geeigneten Alphabet (dem Alphabet der Programmiersprache).

Also ist die Menge aller Programme abzählbar, und damit auch (für jedes $k \geq 0$) die Menge aller berechenbaren Funktionen.

Andererseits gibt es (für jedes $k \geq 0$) *überabzählbar* viele totale Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ (Beweis durch Diagonalisierung), also sind sogar die "meisten" Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ (nämlich überabzählbar viele) *nicht* berechenbar. □

Satz 3.2 liefert nur die *Existenz* von Funktionen, die nicht berechenbar sind. Unser nächstes Ziel ist es, *konkrete* (interessante) Funktionen dieser Art anzugeben. Dazu benötigen wir einige Vorüberlegungen.

Da die Programme π unserer 'hinreichend mächtigen' Programmiersprache Wörter über einem geeigneten Alphabet Σ sind, können wir sie ebenfalls 'durchnummerieren', d.h. wir erhalten eine Folge von Programmen $\pi_0, \pi_1, \pi_2, \ldots$, in der *alle* Programme enthalten sind.

Die Zahl i nennt man die *Gödelnummer* des Programms π_i. Sie lässt sich als Codierung des Programms auffassen.

Mit Hilfe dieser Codierung lässt sich ein Programm als Eingabe für ein anderes Programm verwenden, indem man einfach die *Gödelnummer* des einen Programms in das andere eingibt.

Hätten wir Programme betrachtet, die auf Wörtern arbeiten, dann hätten wir es an dieser Stelle einfacher: Dann könnten wir nämlich den *Text* des einen Programms in das andere Programm eingeben. Insofern kann man die Codierung eines Programms durch seine Gödelnummer als einen technischen Trick betrachten, der nötig ist, weil unsere Programme auf Zahlen und nicht auf Wörtern arbeiten.

Auf jeden Fall können wir jetzt Programme betrachten, die mit anderen Programmen 'rechnen' oder die Fragen über andere Programme beantworten. (Beispiele solcher Programme aus der Praxis sind Compiler oder Interpreter.)

Insbesondere können wir jedes Programm 'auf sich selbst' anwenden, d.h. auf seine eigene Codierung.

Diese Möglichkeit werden wir ausnutzen, um eine nicht entscheidbare Menge (und damit eine nicht berechenbare Funktion) zu finden.

Definition 3.3

1. *Eine Menge* $B \subseteq \mathbb{N}^k$ *heißt* entscheidbar, *wenn die* charakteristische Funktion

$$c_B : \mathbb{N}^k \to \mathbb{N}$$

$$c_B(n_1, \ldots, n_k) = \begin{cases} 1 & \text{falls } (n_1, \ldots, n_k) \in B \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

2. *Eine Menge* $B \subseteq \mathbb{N}^k$ *heißt* semi-entscheidbar *oder* akzeptierbar, *wenn die* Akzeptorfunktion

$$a_B : \mathbb{N}^k \hookrightarrow \mathbb{N}$$

$$a_B(n_1, \ldots, n_k) = \begin{cases} 1 & \text{falls } (n_1, \ldots, n_k) \in B \\ \text{undefiniert} & \text{sonst} \end{cases}$$

berechenbar ist.

Mit anderen Worten:

1. B ist entscheidbar, wenn es einen *Entscheidungsalgorithmus* für B gibt, d.h. einen Algorithmus, der für die Elemente aus B die Ausgabe 1 liefert und für alle übrigen Elemente die Ausgabe 0.
2. B ist semi-entscheidbar, wenn es einen *Semi-Entscheidungsalgorithmus* für B gibt, d.h. einen Algorithmus, der für die Elemente aus B die Ausgabe 1 liefert und der für alle übrigen Elemente nicht hält.

Beispiele:

Beispiele entscheidbarer Mengen sind wieder leicht zu finden, etwa:

1. Die Menge $G \subseteq \mathbb{N}$ der geraden Zahlen ist entscheidbar.
2. Die Menge $P \subseteq \mathbb{N}$ der Primzahlen ist entscheidbar. Ein einfacher (aber ineffizienter) Entscheidungsalgorithmus besteht darin, zu überprüfen, ob n durch eine Zahl m mit $2 \leq m < n$ teilbar ist.

Damit haben wir auch Beispiele für semi-entscheidbare Mengen, denn es gilt

Satz 3.4 *Jede entscheidbare Menge B ist semi-entscheidbar.*

Beweis:

Sei B entscheidbar, d.h. es existiert ein Programm π, das die charakteristische Funktion c_B von B berechnet.

Ein (mögliches) Programm π', das die Akzeptorfunktion a_B von B berechnet, arbeitet wie folgt:

Zuerst wird π ausgeführt.

Wenn π mit Ausgabe 0 hält, dann geht π' in eine Endlosschleife.

Wenn π mit Ausgabe 1 hält, dann gibt π' ebenfalls die 1 aus. □

Ein interessanteres Beispiel für eine semi-entscheidbare Menge ist

$$H = \{(m, n) \in \mathbb{N}^2 \mid \pi_m \text{ hält, wenn es mit Eingabe } n \text{ gestartet wird}\}$$

Ein Semi-Entscheidungsalgorithmus für H arbeitet wie folgt:

1. Zunächst wird das Programm π_m mit Gödelnummer m bestimmt.
2. Dann wird die Berechnung von π_m auf Eingabe n simuliert

   (an dieser Stelle nehmen wir an, dass wir in unserer hinreichend mächtigen Programmiersprache einen Interpreter für die Sprache selbst schreiben können).
3. Wenn die Berechnung von π_m auf n hält, so wird ihr Ergebnis ignoriert und es wird stattdessen 1 ausgegeben,

   und wenn sie nicht hält, hält natürlich auch unser Algorithmus nicht (weil Phase 2. schon nicht hält). □

Die Menge H bezeichnet man als *(allgemeines) Halteproblem*.
Wir haben also gezeigt, dass das Halteproblem semi-entscheidbar ist.
Jetzt wollen wir noch zeigen, dass es *nicht* entscheidbar ist.

Satz 3.5 *Eine Menge $B \subseteq \mathbb{N}^k$ ist genau dann entscheidbar, wenn ihr Komplement $\mathbb{N}^k \setminus B$ entscheidbar ist.*

Beweis:

Aus Symmetriegründen genügt es, eine Richtung zu beweisen.

Sei also B entscheidbar, d.h. es existiert ein Programm π, das die charakteristische Funktion c_B von B berechnet.

Ein Programm $\bar{\pi}$, das $c_{\mathbb{N}^k \setminus B}$ berechnet, arbeitet wie folgt:

Zunächst wird π ausgeführt.

Wenn π mit Ausgabe 1 hält, wird 0 ausgegeben.

Wenn π mit Ausgabe 0 hält, wird 1 ausgegeben. □

Um die Unentscheidbarkeit des Halteproblems H zu beweisen, betrachten wir zunächst die 'speziellere' Menge

$$K = \{n \in \mathbb{N} \mid \pi_n \text{ hält, wenn es mit Eingabe } n \text{ gestartet wird}\}$$

K wird manchmal als *spezielles Halteproblem* bezeichnet, treffender wäre die Bezeichnung *Selbsthalteproblem*.

K enthält genau die (Codierungen der) Programme, die halten, wenn sie 'auf sich selbst', d.h. auf ihre eigenen Codierung angesetzt werden.

Das Komplement von K bezeichnen wir mit $\bar{K}$, also

$$\begin{aligned} \bar{K} &= \mathbb{N} \setminus K \\ &= \{n \in \mathbb{N} \mid \pi_n \text{ hält } \textit{nicht}, \text{ wenn es mit Eingabe } n \text{ gestartet wird}\} \end{aligned}$$

Satz 3.6 (Unentscheidbarkeit des Selbsthalteproblems)

1. $\bar{K}$ *ist* nicht *semi-entscheidbar (also erst recht nicht entscheidbar).*
2. K *ist* nicht *entscheidbar.*

Beweis:

1. Angenommen, $\bar{K}$ ist semi-entscheidbar, d.h. es existiert ein Programm π, das die Akzeptorfunktion $a_{\bar{K}}$ berechnet.

   Sei m die Gödelnummer von π, also $\pi = \pi_m$.

   Dann gilt für *alle* $n \in \mathbb{N}$:

   π_m hält für die Eingabe $n \quad \Leftrightarrow \quad n \in \bar{K}$

   $\Leftrightarrow \quad \pi_n$ hält nicht für die Eingabe n

   Also ergibt sich für $n = m$:

   π_m hält für die Eingabe $m \quad \Leftrightarrow \quad \pi_m$ hält nicht für die Eingabe m

Damit ist die Annahme zum Widerspruch geführt.

Also ist $\bar{K}$ *nicht* entscheidbar.

2. Aus 1. folgt sofort, dass auch K nicht entscheidbar sein kann, weil nach Satz 3.5 das Komplement einer entscheidbaren Menge ebenfalls entscheidbar ist. □

Im Beweis wurde das Prinzip der *Diagonalisierung* benutzt: Man stelle sich eine 'unendliche Tabelle' vor, in der *alle* Programme $\pi_0, \pi_1, \pi_2, \ldots$ und alle möglichen Eingabewerte $0, 1, 2, \ldots$ aufgelistet sind.

	0	1	2	...
π_0	+	−	+	...
π_1	−	+	+	...
π_2	−	−	−	...
⋮				

Ein '+' an Stelle (i, j) soll bedeuten, dass π_i für die Eingabe j hält, ein '$-$' soll bedeuten, dass es nicht hält. Jede Folge von '+' und '$-$' beschreibt also das 'Terminierungsverhalten' eines Programms, insbesondere beschreibt Zeile i das Terminierungsverhalten von π_i.

Jetzt betrachten wir die Diagonale: An ihrer i-ten Stelle steht genau dann ein '+', wenn π_i für die Eingabe i hält, d.h. wenn $i \in K$.

Ein Programm π, das $a_{\bar{K}}$ berechnet, dürfte aber genau an diesen Stellen *nicht* halten, d.h. sein Verhalten würde gerade durch das *Komplement* der Diagonale beschrieben (die Folge, die sich aus der Diagonale ergibt, indem man '+' in '$-$' und '$-$' in '+' umwandelt).

Dieses Komplement stimmt mit keiner der Zeilen überein, weil es sich von Zeile i mindestens an der i-ten Stelle unterscheidet (dem Schnittpunkt der Diagonale mit Zeile i). Also kann es das Programm π nicht geben, weil es unter den π_i nicht vorkommt.

Satz 3.7 (Unentscheidbarkeit des Halteproblems) *Die Menge*

$$H = \{(m,n) \in \mathbb{N}^2 \mid \pi_m \textit{ hält für die Eingabe } n\}$$

ist nicht entscheidbar.

Beweis:

- Idee: Das Selbsthalteproblem ist ein *Spezialfall* des Halteproblems. Also würde aus der Entscheidbarkeit von H die Entscheidbarkeit von K folgen, und die steht im Widerspruch zu Satz 3.6.
- Genauer: Per Definition gilt

$$\begin{aligned} K &= \{n \in \mathbb{N} \mid \pi_n \text{ hält für die Eingabe } n\} \\ &= \{n \in \mathbb{N} \mid (n,n) \in H\} \end{aligned}$$

  Gäbe es einen Entscheidungsalgorithmus π für H, dann könnte man für jede Zahl $n \in \mathbb{N}$ entscheiden, ob sie in K liegt, indem man den Algorithmus π auf das Paar (n,n) anwendet.

Damit hätte man einen Entscheidungsalgorithmus für K im Widerspruch zu Satz 3.6. □

Was bedeutet die Unentscheidbarkeit des Halteproblems?

Wenn wir die technischen Details (d.h. die Codierung von Programmen durch ihre Gödelnummern) ignorieren, dann können wir die Menge H in der Form

$$H = \{(\pi, n) \mid \text{ das Programm } \pi \text{ hält für die Eingabe } n\}$$

schreiben.

Die Unentscheidbarkeit des Halteproblems bedeutet dann, dass es *keinen* Algorithmus gibt, der für *jedes* Programm π und für *jede* Zahl n die Frage

$$\text{Hält } \pi, \text{ wenn es mit Eingabe } n \text{ gestartet wird?}$$

korrekt beantwortet.

Die Schwierigkeit besteht natürlich darin, in den richtigen Fällen die Antwort 'Nein' zu geben.

Die Antwort 'Ja' kann man stets finden, indem man einfach π mit Eingabe n ausführt (simuliert), und 'Ja' antwortet, sobald es hält. (Das ist die *Semi-Entscheidbarkeit* des Halteproblems H, die wir bereits bewiesen haben.)

Aber die Antwort 'Nein' kann man auf diese Weise nicht erhalten, weil man niemals weiß, ob π irgendwann doch noch hält.

Die Unentscheidbarkeit des Halteproblems bedeutet, dass nicht nur diese naive Methode fehlschlägt, sondern dass es auch keinen anderen Algorithmus gibt, um das Problem für alle Programme und alle Eingabewerte zu entscheiden.

Voraussetzung dabei ist natürlich immer, dass die Programmiersprache mächtig genug ist (was aber für alle gängigen Programmiersprachen der Fall ist).

Zusammenfassung der bisher gefundenen Beispiele und Gegenbeispiele:

1. Das Halteproblem H ist semi-entscheidbar, aber nicht entscheidbar.
2. Das gleiche gilt für das Selbsthalteproblem K.
3. Das Komplement $\bar{K}$ von K ist noch nicht einmal semi-entscheidbar.
4. c_H und c_K sind totale Funktionen, die nicht berechenbar sind.
5. $a_{\bar{K}}$ ist eine (echt) partielle Funktion, die nicht berechenbar ist.

Satz 3.8 *$B \subseteq \mathbb{N}^k$ ist genau dann entscheidbar, wenn B und $\mathbb{N}^k \setminus B$ semi-entscheidbar sind.*

Beweis:

- Wenn B entscheidbar ist, dann ist nach Satz 3.5 auch $\mathbb{N}^k \setminus B$ entscheidbar, also sind beide Mengen auch semi-entscheidbar.
- Wenn B und $\mathbb{N}^k \setminus B$ semi-entscheidbar sind, dann gibt es Semi-Entscheidungsalgorithmen π und $\bar{\pi}$ für die beiden Mengen.

  Ein Entscheidungsalgorithmus für B arbeitet dann wie folgt:

  Man startet π und $\bar{\pi}$ gleichzeitig mit der Eingabe $(n_1, \ldots, n_k) \in \mathbb{N}^k$.

  Wenn π zuerst hält, so gibt man 1 aus (weil $(n_1, \ldots, n_k) \in B$),

  wenn $\bar{\pi}$ zuerst hält, so gibt man 0 aus (weil $(n_1, \ldots, n_k) \in \mathbb{N}^k \setminus B$).

  Der Algorithmus hält für jede Eingabe, weil entweder π oder $\bar{\pi}$ hält, und es ist klar, dass er dann die richtige Antwort liefert. $\square$

Definition 3.9 *Eine Menge $B \subseteq \mathbb{N}$ heißt* rekursiv aufzählbar, *wenn entweder $B = \emptyset$ ist oder wenn es eine totale berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ gibt mit $f(\mathbb{N}) = B$.*

Intuition:

$B = f(\mathbb{N})$ bedeutet, dass sich die Elemente von B mit Hilfe der Funktion f 'durchnummerieren' oder 'aufzählen' lassen:

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

Die Berechenbarkeit von f bedeutet dann, dass sich diese Aufzählung mit einem *Algorithmus* durchführen lässt. Einen solchen Algorithmus bezeichnet man als (einen) *Aufzählungsalgorithmus* für B.

Ein Aufzählungsalgorithmus liefert also für jede Zahl $i \in \mathbb{N}$ das i-te Element einer Folge $f(0), f(1), \ldots$, in der genau die Elemente von B enthalten sind.

Da f total ist, kann man den Aufzählungsalgorithmus nacheinander mit Eingabe $0, 1, \ldots$ ausführen und auf diese Weise tatsächlich die Elemente $f(0), f(1), \ldots$ der Menge B 'aufzählen'.

Man beachte, dass die Folge $f(0), f(1), \ldots$ auch Wiederholungen enthalten darf (denn wir haben ja *nicht* gefordert, dass f injektiv ist). Das bedeutet insbesondere, dass die Menge B *endlich* sein kann.

Beispiele:

- Die Menge G der geraden Zahlen ist rekursiv aufzählbar, denn es gilt z.B. $G = f(\mathbb{N})$ für die totale berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = 2n$ für alle $n \in \mathbb{N}$.
  Anstelle von f kann man aber auch viele andere berechenbare Funktionen zur 'Aufzählung' der Menge G finden, etwa die Funktion $g : \mathbb{N} \to \mathbb{N}$ mit $g(n) = 2n$ falls n ungerade und $g(n) = n$ falls n gerade.

- Ähnlich erhält man die rekursive Aufzählbarkeit für die Menge der Quadratzahlen, die Menge der Zweierpotenzen, ...
- Die Menge der Primzahlen ist ebenfalls rekursiv aufzählbar. Dies kann man z.B. zeigen, indem man einen Algorithmus angibt, der Primzahlen in aufsteigender Reihenfolge liefert. Ein effizienter Algorithmus dieser Art ist unter dem Namen 'Sieb des Eratosthenes' bekannt (s. Literatur).

Weitere Beispiele können wir uns ersparen, denn es gilt:

Satz 3.10 *Eine Menge $B \subseteq \mathbb{N}$ ist genau dann rekursiv aufzählbar, wenn sie semi-entscheidbar ist.*

Beweis:

- Sei B rekursiv aufzählbar.

  Wenn $B = \emptyset$ ist, dann ist B sogar entscheidbar.

Wenn $B \neq \emptyset$ ist, dann gibt es nach Voraussetzung eine totale berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(\mathbb{N}) = B$.

Sei π ein Algorithmus, der f berechnet.

Ein Semi-Entscheidungsalgorithmus für B arbeitet wie folgt:

Bei Eingabe $n \in \mathbb{N}$ berechnet man mit dem Algorithmus π nacheinander die Elemente $f(0), f(1), \ldots$ bis man ein $i \in \mathbb{N}$ findet mit $f(i) = n$. In diesem Falle gibt man 1 aus.

Ist $n \in B$, so existiert eine solche Zahl i, und man findet diese Zahl, weil die Berechnungen der Elemente $f(0), \ldots, f(i)$ alle terminieren (da f eine totale Funktion ist). Also hält unser Algorithmus in diesem Falle korrekt mit der Ausgabe 1.

Ist dagegen $n \notin B$, so gibt es kein $i \in \mathbb{N}$ mit $f(i) = n$, also hält unser Algorithmus in diesem Falle nicht, was ebenfalls korrekt ist.

- Sei B semi-entscheidbar.

  Wenn $B = \emptyset$ ist, dann ist B per Definition rekursiv aufzählbar.

  Wenn $B \neq \emptyset$ ist, dann geben wir einen Algorithmus an, der eine unendliche Folge $b_0, b_1, \ldots$ von Elementen in B erzeugt, und überzeugen uns anschließend davon, dass dies *alle* Elemente von B sind.

  Sei dazu π ein Semi-Entscheidungsalgorithmus.

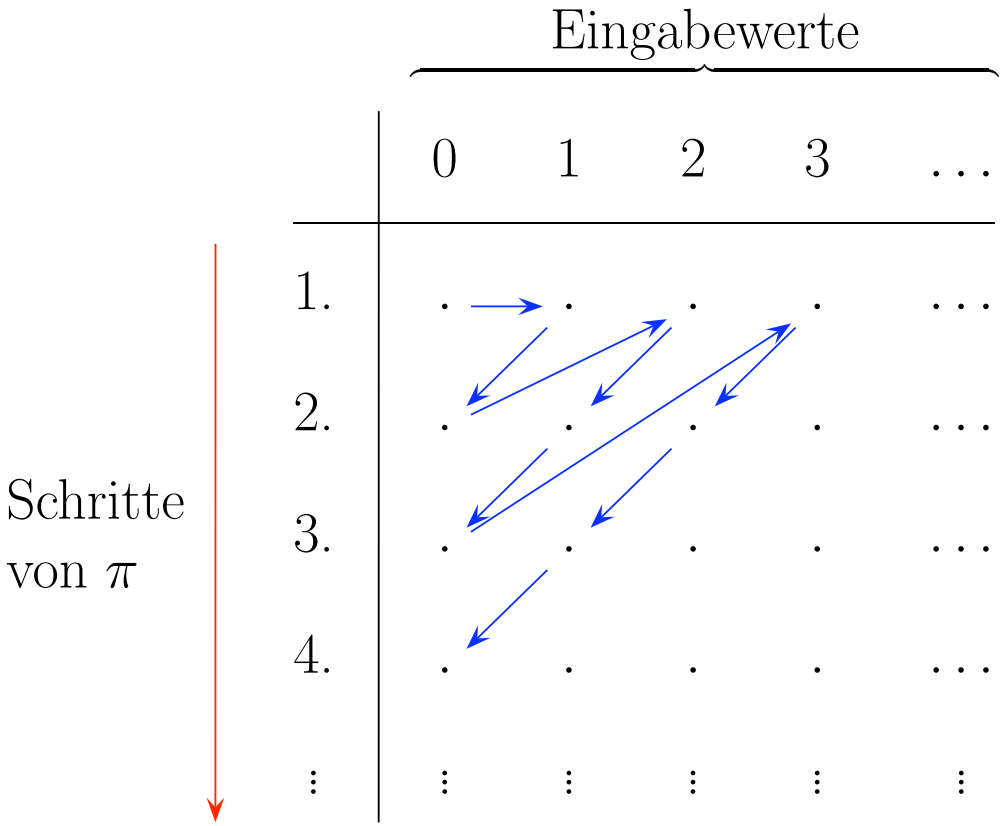
  Wir lassen π 'auf allen Eingabewerten $n \in \mathbb{N}$ parallel laufen', und zwar nach dem sogenannten *dovetailing*-Verfahren:

  Zunächst führen wir den ersten Schritt von π für Eingabe 0 aus.

  Dann führen wir den ersten Schritt von π für Eingabe 1 aus und den zweiten Schritt für Eingabe 0.

  Dann den ersten Schritt für Eingabe 2, den zweiten für Eingabe 1 und den dritten für Eingabe 0, usw.

Diese 'Parallelausführung' von π auf allen möglichen Eingabewerten $n \in \mathbb{N}$ kann man durch nebenstehende Grafik veranschaulichen (die auch den Begriff *dovetailing* erklärt). Dabei bleibt noch zu klären, was wir tun, wenn π für eine Eingabe n im m-ten Schritt hält, d.h. wenn der $(m+1)$-te Schritt gar nicht mehr existiert. Für diesen Fall vereinbaren wir, dass im $(m+1)$-ten und in allen weiteren Schritten der Eingabewert n ausgegeben wird.



Es ist leicht zu sehen, dass dieser dovetailing-Algorithmus eine unendliche Folge von Elementen ausgibt, in der genau die Elemente von B enthalten sind:

Ist nämlich $n \in B$, so hält der Semi-Entscheidungsalgorithmus π für die Eingabe n nach einer gewissen Schrittzahl m, also gibt der dovetailing-Algorithmus die Zahl n immer dann aus, wenn er den $(m+1)$-ten, $(m+2)$-ten, ... Schritt von π bei Eingabe n ausführen müsste. Deshalb wird jedes $n \in B$ sogar unendlich oft ausgegeben (und wegen $B \neq \emptyset$ existiert mindestens ein solches n).

Andererseits gibt der dovetailing-Algorithmus eine Zahl $n \in \mathbb{N}$ nur dann aus, wenn der Semi-Entscheidungsalgorithmus π zuvor für die Eingabe n gehalten hat. Das bedeutet, dass tatsächlich *nur* Elemente von B ausgegeben werden.

Also wird eine unendliche Folge von Elementen ausgegeben, die *genau* die Elemente von B enthält. $\square$

Unsere bisherige Definition von rekursiver Aufzählbarkeit hat (gegenüber Entscheidbarkeit und Semi-Entscheidbarkeit) einen technischen Mangel: Sie bezieht sich nur auf Teilmengen von $\mathbb{N}$ (und nicht auf Teilmengen von $\mathbb{N}^k$ für beliebiges $k \geq 0$).

Um dies zu beheben, verallgemeinern wir zunächst den Begriff der Berechenbarkeit.

Für alle $i, m \in \mathbb{N}$ mit $i \leq m$ sei pr_i^m die i-te Projektion auf der Menge $\mathbb{N}^m$, d.h.

$$pr_i^m : \mathbb{N}^m \to \mathbb{N}$$
$$pr_i^m (n_1, \ldots, n_m) = n_i$$

Definition 3.11 *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}^m$ heißt* berechenbar, *wenn die Funktionen $pr_i^m \circ f : \mathbb{N}^k \to \mathbb{N}$ für $i = 1, \ldots, m$ berechenbar sind.*

Intuition: $pr_i^m \circ f$ ist die Funktion, die die i-te Komponente des Resultats von f liefert. f ist also berechenbar, wenn 'jede Komponente von f' berechenbar ist.

Beispiele:

- Die *Diagonalfunktion*

$$d : \mathbb{N} \to \mathbb{N}^2$$
$$n \mapsto (n, n)$$

  ist berechenbar, weil $pr_1^2 \circ d = pr_2^2 \circ d$ die Identität auf $\mathbb{N}$ ist.

- Wenn $f : \mathbb{N} \hookrightarrow \mathbb{N}$ berechenbar ist, dann ist auch

$$f' : \mathbb{N} \hookrightarrow \mathbb{N}^2$$
$$n \mapsto (n, f(n))$$

  berechenbar, weil $pr_1^2 \circ f'$ die Identität ist und $pr_2^2 \circ f' = f$.

Damit können wir jetzt auch die Definition rekursiv aufzählbarer Mengen verallgemeinern.

Definition 3.12 *Eine Menge $B \subseteq \mathbb{N}^k$ heißt* rekursiv aufzählbar, *wenn entweder $B = \emptyset$ ist oder wenn es eine totale berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}^k$ gibt mit $f(\mathbb{N}) = B$.*

Beispiele:

- Die *Diagonale* $D = \{(n, n) \mid n \in \mathbb{N}\} \subseteq \mathbb{N}^2$ ist rekursiv aufzählbar, weil $D = d(\mathbb{N})$ für die Diagonalfunktion $d : \mathbb{N} \to \mathbb{N}^2$.
- Der *Graph* einer berechenbaren Funktion $f : \mathbb{N} \hookrightarrow \mathbb{N}$, d.h. die Menge

  $$\mathit{Graph}(f) = \{(n, f(n)) \mid n \in \mathit{Def}(f)\}$$

  ist stets rekursiv aufzählbar. Es gilt nämlich $\mathit{Graph}(f) = f'(\mathbb{N})$, wobei $f' : \mathbb{N} \hookrightarrow \mathbb{N}^2$ wie im letzten Beispiel definiert ist.

Alles, was wir bisher über rekursiv aufzählbare Mengen $B \subseteq \mathbb{N}$ gesagt haben, gilt auch für rekursiv aufzählbare Mengen $B \subseteq \mathbb{N}^k$. Insbesondere bleibt Satz 3.10 erhalten. Die Beweisidee bleibt die gleiche, allerdings muss man jetzt (im zweiten Teil des Beweises) den Semi-Entscheidungsalgorithmus π auf allen Elementen $(n_1, \ldots, n_k) \in \mathbb{N}^k$ 'parallel' ausführen. Dazu muss man die Menge $\mathbb{N}^k$ in irgendeiner Reihenfolge 'durchnummerieren', damit man vom ersten, zweiten, dritten, ... Element von $\mathbb{N}^k$ sprechen kann.

Im folgenden wollen wir einige nützliche Zusammenhänge zwischen berechenbaren Funktionen und (semi-)entscheidbaren Mengen beweisen.

Dazu werden als erstes einige mathematische Grundbegriffe geklärt.

Definition 3.13 *Seien X, Y, Z Mengen, $f : X \hookrightarrow Y, g : Y \hookrightarrow Z$ partielle Funktionen und $A \subseteq X, B \subseteq Y$.*

- *$Def(f)$ bezeichnet den Definitionsbereich der Funktion f.*
- *Die Komposition der Funktionen g und f ist die Funktion*

$$g \circ f : X \hookrightarrow Z$$

$$(g \circ f)(x) = \begin{cases} g(f(x)) & \text{falls } x \in Def(f) \text{ und } f(x) \in Def(g) \\ \text{undefiniert} & \text{sonst} \end{cases}$$

- *Das Bild der Menge A unter der Funktion f ist die Menge*

$$f(A) = \{f(x) \mid x \in Def(f) \cap A\}$$

- *Das Urbild der Menge B unter der Funktion f ist die Menge*

$$f^{-1}(B) = \{x \in Def(f) \mid f(x) \in B\}$$

Satz 3.14 *Wenn* $f : \mathbb{N}^k \hookrightarrow \mathbb{N}^l$ *und* $g : \mathbb{N}^l \hookrightarrow \mathbb{N}^m$ *berechenbar sind, dann ist auch die Komposition* $g \circ f : \mathbb{N}^k \hookrightarrow \mathbb{N}^m$ *berechenbar.*

Beweis:

Wir überlegen uns zuerst, dass wir uns auf den Fall $m = 1$ *beschränken können.*

Berechenbarkeit der Funktion g *bedeutet, dass die Funktionen* $pr_i^m \circ g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ *für* $i = 1, \ldots, m$ *berechenbar sind,*

ebenso bedeutet Berechenbarkeit von $g \circ f$*, dass die Funktionen* $pr_i^m \circ (g \circ f)$ *für* $i = 1, \ldots, m$ *berechenbar sind.*

Wegen $pr_i^m \circ (g \circ f) = (pr_i^m \circ g) \circ f$ *genügt es also zu zeigen, dass (für jedes* i*) aus der Berechenbarkeit von* f *und* $pr_i^m \circ g$ *die Berechenbarkeit der Komposition* $(pr_i^m \circ g) \circ f$ *folgt,*

d.h. es genügt, den Satz für Funktionen $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ *zu beweisen.*

Sei also π_g *ein Algorithmus, der* $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ *berechnet, und für jedes* $i \in \{1, \ldots, l\}$ *sei* π_f^i *ein Algorithmus, der* $pr_i^l \circ f$ *berechnet.*

Ein (möglicher) Algorithmus zur Berechnung der Komposition $g \circ f$ sieht dann so aus:

Auf der Eingabe $(n_1, \ldots, n_k) \in \mathbb{N}^k$ führt man nacheinander die Algorithmen $\pi_f^1, \ldots, \pi_f^l$ aus. Wenn alle halten, so erhält man l Ergebnisse $p_1, \ldots, p_l$, und führt dann den Algorithmus π mit Eingabe $(p_1, \ldots, p_l)$ aus.

Die Korrektheit des Algorithmus sieht man so ein:

Wenn $(n_1, \ldots, n_k) \in \mathit{Def}(f)$ und $f(n_1, \ldots, n_k) \in \mathit{Def}(g)$, dann halten alle π_f^i für $(n_1, \ldots, n_k)$ und es gilt $(p_1, \ldots, p_l) = f(n_1, \ldots, n_k)$. Wenn dann noch $f(n_1, \ldots, n_k) \in \mathit{Def}(g)$ gilt, so hält auch π auf $(p_1, \ldots, p_l)$ und liefert das korrekte Ergebnis $g(f(n_1, \ldots, n_k))$.

Wenn $(n_1, \ldots, n_k) \notin \mathit{Def}(f)$, so hält keiner der Algorithmen π_i und wenn $(n_1, \ldots, n_k) \in \mathit{Def}(f)$, aber $f(n_1, \ldots, n_k) \notin \mathit{Def}(g)$, dann hält π' nicht. Also verhält sich der Algorithmus auch in diesen Fällen korrekt. □

Der folgende Satz gibt einige alternative Charakterisierungen für semi-entscheidbarer Mengen an:

Satz 3.15 *Sei $B \subseteq \mathbb{N}^k$. Dann sind die folgenden Aussagen äquivalent.*

1. *B ist semi-entscheidbar.*
2. *B ist Definitionsbereich einer berechenbaren Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}^m$.*
3. *$B = f^{-1}(\{1\})$ für eine berechenbare Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$.*
4. *$B = f^{-1}(A)$ für eine (semi-)entscheidbare Menge $A \subseteq \mathbb{N}^m$ und eine berechenbare Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}^m$.*
5. *B ist rekursiv aufzählbar.*
6. *$B = f(\mathbb{N})$ für eine (partielle) berechenbare Funktion $f : \mathbb{N} \hookrightarrow \mathbb{N}^k$.*
7. *$B = f(A)$ für eine (semi-)entscheidbare Menge $A \subseteq \mathbb{N}^m$ und eine (partielle) berechenbare Funktion $f : \mathbb{N}^m \hookrightarrow \mathbb{N}^k$.*

Beweis:

'1. $\Rightarrow$ 2.': Wenn B semi-entscheidbar ist, dann ist die Akzeptorfunktion $a_B : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechenbar. Wegen $B = \mathit{Def}(a_B)$ kann man also $m = 1$ und $f = a_B$ wählen.

'2. $\Rightarrow$ 3.': Sei $g = \mathit{const}_1 \circ f$, wobei $\mathit{const}_1 : \mathbb{N}^m \to \mathbb{N}$ die konstante Funktion mit Resultat 1 ist. Da f und const_1 berechenbar sind, ist nach Satz 3.14 auch g berechenbar. Außerdem gilt $B = \mathit{Def}(f) = \mathit{Def}(g) = g^{-1}(\{1\})$.

'3. $\Rightarrow$ 4.': Man wähle $m = 1$ und $A = \{1\}$.

'4. $\Rightarrow$ 1.': Wenn $B = f^{-1}(A)$ ist, dann gilt $a_B = a_A \circ f$ (wie man leicht nachrechnet). Da A semi-entscheidbar ist, ist a_A berechenbar, also ist nach Satz 3.14 auch a_B berechenbar und damit B semi-entscheidbar.

'1. $\Leftrightarrow$ 5.': gilt nach Satz 3.10.

'5. $\Rightarrow$ 6.': Per Definition existiert sogar eine totale Funktion $f : \mathbb{N} \to \mathbb{N}^k$ mit $f(\mathbb{N}) = B$.

'6. $\Rightarrow$ 5.': Sei π ein Algorithmus, der f berechnet. Aus π erhält man einen Aufzählungsalgorithmus für B nach dem gleichen Prinzip wie im Beweis von Satz 3.10: Man führt π auf allen $n \in \mathbb{N}$ 'parallel' aus (*dovetailing*), und immer wenn es für ein n hält gibt man von diesem Zeitpunkt an das Ergebnis (also $f(n)$) aus.

'6. $\Rightarrow$ 7.': Man wähle $m = 1$ und $A = \mathbb{N}$.

'7. $\Rightarrow$ 6.': Nach Satz 3.10 ist A rekursiv aufzählbar, d.h. es existiert eine berechenbare Funktion $g : \mathbb{N} \to \mathbb{N}^m$ mit $A = g(\mathbb{N})$. Nach Satz 3.14 ist dann auch $f \circ g : \mathbb{N} \hookrightarrow \mathbb{N}^k$ berechenbar. Also ist $B = f(A) = f(g(\mathbb{N})) = (f \circ g)(\mathbb{N})$ rekursiv aufzählbar. $\square$

Auch entscheidbare Mengen lassen sich auf unterschiedliche Weise charakterisieren.

Satz 3.16 *Sei $B \subseteq \mathbb{N}^k$. Dann sind die folgenden Aussagen äquivalent.*

1. *B ist entscheidbar.*
2. *$B = f^{-1}(\{1\})$ für eine totale berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}$.*
3. *$B = f^{-1}(A)$ für eine entscheidbare Menge $A \subseteq \mathbb{N}^m$ und eine totale berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}^m$.*

Beweis:

'1. $\Rightarrow$ 2.': Man wähle $f = c_B$.

'2. $\Rightarrow$ 3.': Man wähle $m = 1$ und $A = \{1\}$.

'3. $\Rightarrow$ 1.': Wenn $B = f^{-1}(A)$, dann ist $c_B = c_A \circ f$. Da A entscheidbar ist, ist c_A berechenbar, also nach Satz 3.14 auch c_B, und damit ist B entscheidbar. □

Oft können wir die Entscheidbarkeit oder Unentscheidbarkeit einer Menge nachweisen, indem wir ein neues Problem auf ein bereits bekanntes 'zurückführen'. Diese Idee lässt sich wie folgt präzisieren.

Definition 3.17 *Sei $A \subseteq \mathbb{N}^k$ und $B \subseteq \mathbb{N}^m$. A heißt* reduzierbar *auf B, wenn es eine totale berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}^m$ gibt mit $A = f^{-1}(B)$.*

Intuition:

Wenn A auf B reduzierbar ist, dann lässt sich das Problem A folgendermaßen auf das Problem B 'zurückführen' oder 'reduzieren':

Um zu überprüfen, ob ein Element $(n_1, \ldots, n_k)$ in A liegt, berechnet man einfach den Wert $f(n_1, \ldots, n_k)$ und überprüft, ob er in B liegt.

Diese Intuition kommt auch in folgendem Satz zum Ausdruck.

Satz 3.18 *Sei A auf B reduzierbar. Dann gilt:*

1. *Wenn B entscheidbar ist, dann ist auch A entscheidbar.*
2. *Wenn B semi-entscheidbar ist, dann ist auch A semi-entscheidbar.*

Beweis:

1. ist Teil von Satz 3.16.
2. ist Teil von Satz 3.15. □

Man kann Satz 3.18 auf zweierlei Art verwenden, nämlich:

a. Man beweist die *Entscheidbarkeit* eines Problems A, indem man es auf ein bereits bekanntes entscheidbares Problem B reduziert.

b. Man beweist die *Unentscheidbarkeit* eines Problems B, indem man ein bereits bekanntes unentscheidbares Problem A auf B reduziert.

Das Gleiche gilt natürlich für Semi-Entscheidbarkeit anstelle von Entscheidbarkeit.

Methode a. haben wir im ersten Teil der Vorlesung häufig benutzt. Ein Beispiel (unter vielen) ist die Zurückführung des Wortproblems für NDEAs auf das Wortproblem für DEAs: Sei

$$\begin{aligned} W_{DEA} &= \{(A,w) \mid A \text{ ist ein DEA und } w \in L(A)\} \\ W_{NDEA} &= \{(A,w) \mid A \text{ ist ein NDEA und } w \in L(A)\} \end{aligned}$$

Wenn wir Automaten und Wörter als natürliche Zahlen codieren, so sind W_{DEA} und W_{NDEA} Teilmengen von $\mathbb{N}^2$ und die Potenzautomatenkonstruktion ist eine totale berechenbare Funktion $pot : \mathbb{N} \to \mathbb{N}$, die zu jedem NDEA A einen äquivalenten DEA $pot(A)$ liefert. Also ist auch die Funktion

$$\begin{aligned} f &: \mathbb{N}^2 \to \mathbb{N}^2 \\ (A,w) &\mapsto (pot(A), w) \end{aligned}$$

berechenbar und es gilt $W_{NDEA} = f^{-1}(W_{DEA})$ (wie man leicht nachrechnet). Damit folgt die Entscheidbarkeit von W_{NDEA} aus der Entscheidbarkeit von W_{DEA}.

Methode b. haben wir auch schon benutzt, nämlich beim Beweis der Unentscheidbarkeit des Halteproblems H.

Wir hatten zuvor bewiesen, dass das Selbsthalteproblem K unentscheidbar ist. Dann hatten wir ausgenutzt, dass

$$K = \{n \in \mathbb{N} \mid (n,n) \in H\}$$

gilt. Mit anderen Worten: $K = d^{-1}(H)$, wobei $d : \mathbb{N} \to \mathbb{N}^2$ definiert ist durch $d(n) = (n,n)$. Das beweist, dass K auf H reduzierbar ist (weil d eine totale berechenbare Funktion ist), also ergibt sich die Unentscheidbarkeit von H aus der Unentscheidbarkeit von K.

Mit der gleichen Methode beweisen wir nun noch einige weitere Unentscheidbarkeitsresultate (die eng mit dem Halteproblem H zusammenhängen).

Satz 3.19 *Die folgenden Mengen sind unentscheidbar.*

1. $H_0 = \{n \in \mathbb{N} \mid \pi_n$ *hält für die Eingabe* $0\}$
2. $H_\exists = \{n \in \mathbb{N} \mid \pi_n$ *hält für mindestens einen Eingabewert* $m \in \mathbb{N}\}$
3. $H_\forall = \{n \in \mathbb{N} \mid \pi_n$ *hält für alle Eingabewerte* $m \in \mathbb{N}\}$
4. $Equiv = \{(m,n) \in \mathbb{N}^2 \mid \pi_m$ *und* π_n *berechnen die gleiche Funktion*$\}$

Bemerkungen:

H_0 ist das *spezielle Halteproblem* für die Eingabe 0 (wobei man anstelle von 0 natürlich auch jede andere Zahl wählen könnte).

$H_\exists$ enthält genau die (Gödelnummern der) Programme, die eine *nicht-leere* Funktion berechnen.

$H_\forall$ ist das *Totalitätsproblem*, es enthält genau die (Gödelnummern der) Programme, die eine *totale* Funktion berechnen.

Equiv ist das *Äquivalenzproblem* für Programme (denn zwei Programme die die gleiche Funktion berechnen, nennt man *äquivalent*).

Beweis:

1. Wir zeigen, dass sich das Halteproblem H auf H_0 reduzieren lässt (dies ist etwas überraschend, weil H_0 ja ein Spezialfall von H ist).

   Für jedes Programm π und jede Zahl $n \in \mathbb{N}$ sei $\pi^{(n)}$ ein Programm, das seine eigene Eingabe ignoriert und dann das Programm π auf Eingabe n ausführt.

   Ein solches Programm $\pi^{(n)}$ lässt sich stets aus π und n konstruieren (wobei wir für die Details der Konstruktion unsere Programmiersprache festlegen müssten), d.h. die totale Funktion

   $$f : \mathbb{N}^2 \to \mathbb{N}$$
   $$(m, n) \mapsto \text{Gödelnummer von } \pi_m^{(n)}$$

   ist berechenbar.

Nun gilt für alle $(m,n) \in \mathbb{N}^2$:

$$
\begin{aligned}
(m,n) \in H &\Leftrightarrow \pi_m \text{ hält für Eingabe } n \\
&\Leftrightarrow \pi_m^{(n)} \text{ hält für Eingabe 0} \\
&\quad \text{(da } \pi_m^{(n)} \text{ seine Eingabe ignoriert und sich wie } \pi_m \text{ bei Eingabe } n \text{ verhält)} \\
&\Leftrightarrow f(m,n) \in H_0 \\
&\quad \text{(da } f(m,n) \text{ die Gödelnummer von } \pi_m^{(n)} \text{ ist)}
\end{aligned}
$$

Das bedeutet $H = f^{-1}(H_0)$, d.h. H ist auf H_0 reduzierbar, also folgt die Unentscheidbarkeit von H_0 aus der Unentscheidbarkeit von H.

2. und 3. Mit den Bezeichnungen aus 1. gilt auch:

$$
\begin{aligned}
(m,n) \in H &\Leftrightarrow \pi_m^{(n)} \text{ hält für } \textit{mindestens einen} \text{ Eingabewert} \\
&\Leftrightarrow \pi_m^{(n)} \text{ hält für } \textit{alle} \text{ Eingabewerte}
\end{aligned}
$$

Also ist auch $H = f^{-1}(H_\exists) = f^{-1}(H_\forall)$, und damit folgt die Unentscheidbarkeit dieser beiden Mengen ebenfalls aus der Unentscheidbarkeit von H.

4. Wir zeigen, dass sich $H_\forall$ auf *Equiv* reduzieren lässt.

   Sei $\pi^{(0)}$ ein Programm, das stets hält und 0 ausgibt (also ein Programm, das die Funktion $const_0$ berechnet).

   Für jedes Programm π sei $\pi;\pi^{(0)}$ die Hintereinanderausführung von π und $\pi^{(0)}$.

   $\pi;\pi^{(0)}$ hält für die gleichen Eingabewerte wie π und gibt stets 0 aus (falls es hält).

   Also gilt für jedes Programm π:

   π ist genau dann total, wenn $\pi;\pi^{(0)}$ die Funktion $const_0$ berechnet, d.h. wenn $\pi;\pi^{(0)}$ äquivalent zu $\pi^{(0)}$ ist.

Damit ist im Prinzip schon gezeigt, wie man Totalität auf Äquivalenz reduziert.

Es folgt noch die formale Argumentation:

Da sich $\pi;\pi^{(0)}$ stets aus π konstruieren lässt, ist die Funktion

$$f : \mathbb{N} \to \mathbb{N}^2$$

$$n \mapsto (\text{Gödelnummer von } \pi_n;\pi^{(0)},\ \text{Gödelnummer von } \pi^{(0)})$$

berechenbar.

Nun gilt für alle $n \in \mathbb{N}$:

$$\begin{aligned} n \in H_\forall &\Leftrightarrow \pi_n \text{ berechnet eine totale Funktion} \\ &\Leftrightarrow \pi_n;\pi^{(0)} \text{ ist äquivalent zu } \pi^{(0)} \\ &\Leftrightarrow f(n) \in \mathit{Equiv} \end{aligned}$$

Also ist $H_\forall = f^{-1}(\mathit{Equiv})$, d.h. $H_\forall$ ist auf *Equiv* reduzierbar. □

Wie 'schwierig' sind die Probleme aus Satz 3.19?
Sind sie wenigstens semi-entscheidbar?

Satz 3.20

1. H_0 *ist semi-entscheidbar.*
2. $H_\exists$ *ist semi-entscheidbar.*
3. $H_\forall$ *ist* nicht *semi-entscheidbar.*
4. *Equiv ist* nicht *semi-entscheidbar.*

Beweis:

1. Ein möglicher Semi-Entscheidungsalgorithmus für H_0 sieht so aus: Bei Eingabe n führt man π_n für die Eingabe 0 aus. Falls es hält, gibt man 1 aus.

   Aternative Argumentation: H_0 ist auf H reduzierbar, denn es gilt $H_0 = f^{-1}(H)$, wobei $f : \mathbb{N} \to \mathbb{N}$ definiert ist durch $f(n) = (n, 0)$.

Also folgt die Unentscheidbarkeit von H_0 aus der Unentscheidbarkeit von H.

2. Ein möglicher Semi-Entscheidungsalgorithmus für $H_\exists$ sieht so aus: Bei Eingabe n führt man π_n für alle Eingaben $m \in \mathbb{N}$ 'parallel' aus (*dovetailing*). Sobald eine Eingabe m gefunden ist, für die es hält, gibt man 1 aus.

3. Wir zeigen durch einen Diagonalisierungsbeweis, dass $H_\forall$ nicht rekursiv aufzählbar, und damit auch nicht semi-entscheidbar ist.

   Angenommen, $H_\forall$ ist rekursiv aufzählbar, d.h. es existiert eine totale berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(\mathbb{N}) = H_\forall$.

   Dann betrachten wir den Algorithmus π, der für jede Eingabe $n \in \mathbb{N}$ wie folgt arbeitet:

   - Zuerst wird $f(n)$ berechnet.

   - Dann wird das Programm $\pi_{f(n)}$ für die Eingabe n ausgeführt.

- Dann wird das Ergebnis um 1 erhöht und ausgegeben.

Der Algorithmus π hält für *jede* Eingabe $n \in \mathbb{N}$, denn:

- Die Berechnung von $f(n)$ hält, weil f eine totale Funktion ist.
- Das Programm $\pi_{f(n)}$ hält für alle Eingabewerte, weil $f(n) \in H_\forall$, also hält es insbesondere für die Eingabe n.

Damit ist gezeigt, dass π eine totale Funktion berechnet, d.h. π müsste mit einem der Programme $\pi_{f(n)}$ übereinstimmen.

Das ist aber nicht möglich, da sich—per Definition des Algorithmus π—die Resultate von π und $\pi_{f(n)}$ bei Eingabe n um 1 unterscheiden.

4. Im Beweis von Satz 3.19 haben wir schon gesehen, dass sich $H_\forall$ auf *Equiv* reduzieren lässt.

   Also würde aus der Semi-Entscheidbarkeit von *Equiv* die Semi-Entscheidbarkeit von $H_\forall$ folgen im Widerspruch zu 3. □

Die bisherigen Überlegungen haben gezeigt, dass viele *semantische Eigenschaften* von Programmen unentscheidbar sind. Dabei verstehen wir unter einer semantischen Eigenschaft eines Programms π eine Eigenschaft, die sich nur auf die von π berechnete *Funktion* bezieht. Es stellt sich die Frage, ob es auch entscheidbare semantische Eigenschaften gibt.

Satz 3.21 (Satz von Rice) *Sei S eine echte, nichtleere Teilmenge der Menge aller partiellen berechenbaren Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$. Dann ist die Menge*

$$\mathit{Prog}_S = \{n \in \mathbb{N} \mid \text{die von } \pi_n \text{ berechnete Funktion liegt in } S\}$$

unentscheidbar.

Der Satz von Rice besagt, dass jede 'nichttriviale semantische Eigenschaft' von Programmen unentscheidbar ist. Prog_S enthält nämlich genau die (Gödelnummern der) Programme, die die semantische Eigenschaft S haben, wobei nur die 'trivialen' Eigenschaften $S = \emptyset$ und $S = \{f : \mathbb{N}^k \hookrightarrow \mathbb{N} \mid f \text{ berechenbar}\}$ ausgeschlossen sind.

Anwendungsbeispiele für den Satz von Rice sind unsere bisherigen Unentscheidbarkeits-Resultate für die Mengen

- H_0 (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f$ berechenbar und $0 \in \mathit{Def}(f)\}$)
- $H_\exists$ (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f$ berechenbar und $\mathit{Def}(f) \neq \emptyset\}$)
- $H_\forall$ (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f$ berechenbar und $\mathit{Def}(f) = \mathbb{N}\}$)

Die Unentscheidbarkeit von *Equiv* ergibt sich nicht direkt aus dem Satz von Rice, weil sich *Equiv* ja auf *Paare* von Funktionen bezieht. Stattdessen kann man zeigen, dass ein *spezielles Äquivalenzproblem* unentscheidbar ist, z.B. das Problem

$$\mathit{Equiv}_0 = \{n \in \mathbb{N} \mid \pi_n \text{ berechnet die Funktion } \mathit{const}_0\}$$

(wähle $S = \{\mathit{const}_0\}$). Daraus ergibt sich dann wieder die Unentscheidbarkeit von *Equiv*, weil sich Equiv_0 auf *Equiv* reduzieren lässt.

Gegenbeispiele:

- Die Menge

  $$\{n \in \mathbb{N} \mid \pi_n \text{ hält für Eingabe 0 nach höchstens 10 Schritten}\}$$

  ist entscheidbar.

  Die in der Menge genannte Eigenschaft bezieht sich ja nicht (nur) auf die von π berechnete Funktion, sondern auf das Programm π selbst. Deshalb ist der Satz von Rice hier *nicht* anwendbar.

  Ein Entscheidungsalgorithmus für diese Menge besteht einfach darin, dass man die ersten 10 Schritte der Berechnung durchführt, und dann die entsprechende Antwort ausgibt.

- Mit der gleichen Argumentation erhält man die Entscheidbarkeit der Menge

  $$H' = \{(n, m, k) \in \mathbb{N}^3 \mid \pi_n \text{ hält für Eingabe } m \text{ nach höchstens } k \text{ Schritten}\}$$

Berechenbarkeitstheorie

Unseren bisherigen Betrachtungen zur Berechenbarkeitstheorie sind etwas vage, weil wir den Begriff *Algorithmus* nicht präzise definiert haben.

Bisher verstehen wir unter einem Algorithmus ein 'Programm in einer hinreichend mächtigen Programmiersprache'.

Dabei haben wir (in den Beweisen) gewisse Annahmen über diese Programmiersprache gemacht, z.B.

- Zu zwei Programmen π_1, π_2 gibt es ein Programm π, das die beiden Programme (auf dem gleichen Eingabewert) 'parallel' ausführt.
- Zu jedem Programm π gibt es ein Programm π', das π auf allen möglichen Eingabewerten 'parallel' ausführt (mittels dovetailing).
- Jedes Programm lässt sich als natürliche Zahl codieren.
- *In* der Programmiersprache kann man einen Interpreter *für* die Programmiersprache schreiben.

Es bleibt zu zeigen, dass eine solche Programmiersprache existiert.

Dazu gibt es viele unterschiedliche Ansätze, von denen wir die folgenden drei betrachten:

1. Die Sprache der *while*-Programme.
2. Die Sprache der μ-rekursiven Funktionen.
3. Die Sprache der Turing-Programme (= 'Turing-Maschinen').

1. ist eine kleine imperative Programmiersprache mit Zuweisungen, Komposition, bedingten Anweisungen und while-Schleifen,

   (also eine kleine Teilmenge von Sprachen wie Pascal, C, Java).
2. ähnelt einer funktionalen Programmiersprache.
3. ist ein Maschinenmodell, bei dem sehr kleine Berechnungsschritte auf einem (mit Zeichen beschriebenen) unendlichen Band ausgeführt werden.

Es wird sich herausstellen, dass diese drei Programmiersprachen die gleiche Mächtigkeit haben, d.h. dass sie zum gleichen Begriff von *Berechenbarkeit* führen. Solche Programmiersprachen nennt man *Turing-mächtig*.

Bisher wurde keine Programmiersprache gefunden, mit der sich mehr Funktionen berechnen lassen als mit den oben genannten. Das gibt Anlass zu folgender Vermutung.

Churchsche These: Jede intuitiv berechenbare Funktion lässt sich mit einer Turing-Maschine (oder einem *while*-Programm oder einer μ-rekursiven Funktion) berechnen.

Man kann die Churchsche These nicht beweisen, da sie sich auf einen intuitiven Begriff bezieht. Es könnte höchstens sein, dass sie eines Tages widerlegt wird, indem man eine mächtigere Programmiersprache findet. Aber auch das würde die Berechenbarkeitstheorie nicht zum Einsturz bringen (weil man in den Beweisen ja immer nur annimmt, dass die Programmiersprache mächtig genug ist).

Syntax von *while*-Programmen

Vorgegeben sei eine abzählbar unendliche Menge $Loc = \{X_0, X_1, \ldots\}$, deren Elemente wir *Speicherplätze* nennen.
Ein *while-Programm* (zur Berechnung einer k-stelligen Funktion) hat die Form

$$\text{read } X_{i_1}, \ldots, X_{i_k};\ \mathit{stl}\ \text{write } X_i$$

wobei die X_{i_j} verschieden sind und *stl* eine Anweisungsliste ist.
Anweisungen *st* und Anweisungslisten *stl* sind definiert durch:

$$\begin{array}{rcl} \mathit{st} & ::= & X_i := 0 \mid X_i := X_j \mid X_i := \mathit{succ}(X_j) \mid X_i := \mathit{pred}(X_j) \\ & \mid & \text{if } X_i \text{ then } \mathit{stl}_1 \text{ else } \mathit{stl}_2 \text{ fi} \\ & \mid & \text{while } X_i \text{ do } \mathit{stl}_0 \text{ od} \\ \mathit{stl} & ::= & \varepsilon \\ & \mid & \mathit{st};\ \mathit{stl} \end{array}$$

Semantik von *while*-Programmen

while-Programme arbeiten auf einem 'Speicher'. Jede Anweisung (oder Anweisungsliste) verändert den Inhalt einiger Speicherplätze, d.h. sie verändert den 'Speicherzustand'.

Definition 3.22

- *Ein* Speicherzustand *(oder kurz:* Zustand*) ist eine totale Funktion* $\sigma : Loc \to \mathbb{N}$.
- *Der Speicherzustand* $\sigma_0 : Loc \to \mathbb{N}$ *sei definiert durch* $\sigma_0(X_i) = 0$ *für alle* $X_i \in Loc$.
- *Mit Store bezeichnen wir die Menge aller Speicherzustände.*

Intuition:
$\sigma(X_i)$ ist der Inhalt des Speicherplatzes X_i im Zustand σ. Den speziellen Speicherzustand σ_0 benötigen wir, um den Anfangszustand eines Programms zu beschreiben.

Um Veränderungen des Speicherzustands auszudrücken, benötigen wir die folgende Schreibweise.

Definition 3.23 *Für $\sigma \in Store, X_i \in Loc$ und $n \in \mathbb{N}$ sei*

$$\sigma[n/X_i] : Loc \to \mathbb{N}$$

$$X_i \mapsto n$$

$$X_j \mapsto \sigma(X_j) \text{ für alle } j \neq i$$

Intuition:
$\sigma[n/X_i]$ ist der Speicherzustand, der sich aus σ ergibt, indem man den Inhalt von X_i durch n überschreibt (also der Zustand, der durch eine Zuweisung $X_i := n$ aus σ entstehen würde).

Den Ablauf eines *while*-Programms beschreiben wir durch Übergangsschritte zwischen Konfigurationen. In einer Konfiguration merken wir uns

- den aktuellen Speicherzustand,
- die Anweisungsliste, die noch abzuarbeiten ist.

Definition 3.24 *Eine* Konfiguration *ist ein Paar* (stl, σ)*, wobei* stl *eine Anweisungsliste ist und* $\sigma \in Store$*.*

Auf der Menge aller Konfigurationen definieren wir eine Relation $\vdash$ (die *Übergangsschrittrelation*). Wir schreiben wieder $\vdash^n$, $\vdash^+$ und $\vdash^*$ für die n-te Potenz, den transitiven Abschluss und den reflexiven transitiven Abschluss von $\vdash$.

$\vdash$ ist definiert als die kleinste Relation mit folgenden Eigenschaften:

- $(X_i := 0; \mathit{stl}, \sigma) \vdash (\mathit{stl}, \sigma[0/X_i])$
- $(X_i := X_j; \mathit{stl}, \sigma) \vdash (\mathit{stl}, \sigma[\sigma(X_j)/X_i])$
- $(X_i := \mathit{succ}(X_j); \mathit{stl}, \sigma) \vdash (\mathit{stl}, \sigma[\sigma(X_j) + 1/X_i])$
- $(X_i := \mathit{pred}(X_j); \mathit{stl}, \sigma) \vdash (\mathit{stl}, \sigma[\sigma(X_j) \dot{-} 1/X_i])$
  wobei $m \dot{-} n = m - n$ falls $m \geq n$ und $m \dot{-} n = 0$ sonst
- (if X_i then stl_1 else stl_2 fi; stl, σ) $\vdash$ ($\mathit{stl}_1\, \mathit{stl}$, σ) falls $\sigma(X_i) \neq 0$
- (if X_i then stl_1 else stl_2 fi; stl, σ) $\vdash$ ($\mathit{stl}_2\, \mathit{stl}$, σ) falls $\sigma(X_i) = 0$
- (while X_i do stl_0 od; stl, σ) $\vdash$ (stl_0 while X_i do stl_0 od; stl, σ)
  falls $\sigma(X_i) \neq 0$
- (while X_i do stl_0 od; stl, σ) $\vdash$ (stl, σ) falls $\sigma(X_i) = 0$

Die von einem *while*-Programm

$$P \;=\; \text{read } X_{i_1}, \ldots, X_{i_k};\ \mathit{stl}\ \text{write } X_i$$

berechnete Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist definiert durch:

$$f(n_1, \ldots, n_k) = \begin{cases} \sigma(X_i) & \text{falls } (\mathit{stl}, \sigma_0[n_1/X_{i_1}] \ldots [n_k/X_{i_k}]) \vdash^* (\varepsilon, \sigma) \\ \text{undefiniert} & \text{falls kein solches } \sigma \text{ existiert} \end{cases}$$

In Worten:

- Durch read $X_{i_1}, \ldots, X_{i_k}$ werden die X_{i_j} mit den Eingabewerten n_j besetzt und alle anderen Speicherplätze mit 0. Das liefert den Anfangszustand $\sigma_0[n_1/X_{i_1}] \ldots [n_k/X_{i_k}]$.
- In diesem Anfangszustand wird die Anweisungsliste *stl* ausgeführt.
- Wenn deren Ausführung terminiert, d.h. wenn sich irgendwann eine Konfiguration der Form (ε, σ) ergibt, in der *stl* vollständig abgearbeitet ist, dann wird durch write X_i der Wert $\sigma(X_i)$ als Ergebnis der Funktion f abgeliefert. Andernfalls ist f undefiniert.

Beispiel:

Sei P das Programm

read X_0, X_1;

while X_0 do $X_0 := \mathit{pred}(X_0)$; $X_1 := \mathit{succ}(X_1)$ od;

write X_1

Welches Ergebnis liefert dieses Programm für die Eingabe $(2, 5)$?

Dazu betrachten wir die Abarbeitung der Anweisungsliste im Anfangszustand $\sigma_0[2/X_0][5/X_1]$.

(while X_0 do $X_0 := \mathit{pred}(X_0)$; $X_1 := \mathit{succ}(X_1)$; od;,

$\sigma_0[2/X_0][5/X_1]$)

$\vdash$ ($X_0 := \mathit{pred}(X_0)$; $X_1 := \mathit{succ}(X_1)$; while X_0 do ... od;,

$\sigma_0[2/X_0][5/X_1]$)

$$\vdash \quad (X_1 := \mathit{succ}(X_1);\ \textsf{while}\ X_0\ \textsf{do}\ \ldots\ \textsf{od};,$$
$$\sigma_0[1/X_0][5/X_1])$$
$$\vdash \quad (\textsf{while}\ X_0\ \textsf{do}\ X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \textsf{od};,$$
$$\sigma_0[1/X_0][6/X_1])$$
$$\vdash \quad (X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \textsf{while}\ X_0\ \textsf{do}\ \ldots\ \textsf{od};,$$
$$\sigma_0[1/X_0][6/X_1])$$
$$\vdash \quad (X_1 := \mathit{succ}(X_1);\ \textsf{while}\ X_0\ \textsf{do}\ \ldots\ \textsf{od};,$$
$$\sigma_0[0/X_0][6/X_1])$$
$$\vdash \quad (\textsf{while}\ X_0\ \textsf{do}\ X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \textsf{od};,$$
$$\sigma_0[0/X_0][7/X_1])$$
$$\vdash \quad (\varepsilon,\ \sigma_0[0/X_0][7/X_1])$$

Für den Endzustand $\sigma = \sigma_0[0/X_0][7/X_1]$ gilt $\sigma(X_1) = 7$.
Also gilt für die vom Programm P berechnete Funktion f:

$$f(2,5) = 7$$

Definition 3.25

- *Die vom while-Programm P berechnete Funktion bezeichnen wir mit $[\![P]\!]$.*
- *Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ heißt while-berechenbar, wenn es ein while-Programm P (für k-stellige Funktionen) gibt mit $[\![P]\!] = f$.*

Vermutung: Unser Beispiel-Programm berechnet die Funktion

$$\mathit{plus} : \mathbb{N}^2 \to \mathbb{N}$$

$$(m,n) \mapsto m + n$$

Also ist *plus while*-berechenbar.

Mit Hilfe unserer formalen Semantik können wir eine solche Aussage über ein *while*-Programm beweisen. Dazu formulieren wir eine geeignete Behauptung über die im Programm vorkommende *while*-Schleife: Für alle $\sigma \in \mathit{Store}$ gilt

$$\begin{aligned}&(\text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od};,\ \sigma)\\ &\quad \vdash^*\ (\varepsilon,\ \sigma[0/X_0][\sigma(X_0)+\sigma(X_1)/X_1])\end{aligned} \qquad (*)$$

Diese Behauptung beweisen wir durch Induktion über $\sigma(X_0)$.

$\sigma(X_0) = 0$:

In diesem Falle gilt

$$(\text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od};,\ \sigma) \ \vdash\ (\varepsilon,\ \sigma)$$

und wegen $\sigma(X_0) = 0$ ist

$$\sigma \ = \ \sigma[0/X_0][\sigma(X_0)+\sigma(X_1)/X_1]$$

d.h. σ ist der in (*) verlangte Endzustand.

$\sigma(X_0) > 0$:

In diesem Falle gilt

$$(\text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od};,\ \sigma)$$

$$\vdash \quad (X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{while } X_0 \text{ do } \ldots \text{ od};,\ \sigma)$$

$$\vdash^2 \quad (\text{while } X_0 \text{ do } \ldots \text{ od};,\ \underbrace{\sigma[\sigma(X_0) - 1/X_0][\sigma(X_1) + 1/X_1]}_{\sigma'})$$

Nun ist $\sigma'(X_0) = \sigma(X_0) - 1 < \sigma(X_0)$, also gilt die Behauptung (*) nach Induktionsannahme für den Zustand σ', d.h.

$$(\text{while } X_0 \text{ do } \ldots \text{ od};,\ \sigma') \vdash^* (\varepsilon,\ \sigma'[0/X_0][\sigma'(X_0) + \sigma'(X_1)/X_1])$$

Durch Zusammensetzen der beiden Berechnungsfolgen erhalten wir

$$(\text{while } X_0 \text{ do } \ldots \text{ od};,\ \sigma) \vdash^* (\varepsilon,\ \underbrace{\sigma'[0/X_0][\sigma'(X_0) + \sigma'(X_1)/X_1]}_{\sigma''})$$

also bleibt zu zeigen, dass σ'' der in (*) verlangte Endzustand ist.

Dazu betrachten wir die Werte $\sigma''(X_i)$ für alle $X_i \in \mathit{Loc}$:

- $\sigma''(X_0) = 0$
- $\sigma''(X_1) = \sigma'(X_0) + \sigma'(X_1) = \sigma(X_0) - 1 + \sigma(X_1) + 1 = \sigma(X_0) + \sigma(X_1)$
- $\sigma''(X_i) = \sigma'(X_i) = \sigma(X_i)$ für alle $i \geq 2$

Also ist $\sigma'' = \sigma[0/X_0][\sigma(X_0) + \sigma(X_1)/X_1]$, d.h. (*) ist bewiesen.

Insbesondere gilt dann für alle $(m, n) \in \mathbb{N}^2$:

$(\text{while } X_0 \text{ do } \ldots \text{ od};, \ \sigma_0[m/X_0][n/X_1]) \vdash^* (\varepsilon, \ \sigma_0[0/X_0][m+n/X_1])$

also berechnet das Programm

$$P = \text{read } X_0, X_1; \ \text{while } X_0 \text{ do } \ldots \text{ od}; \ \text{write } X_1$$

tatsächlich die Funktion $\mathit{plus} : \mathbb{N}^2 \to \mathbb{N}$ mit $\mathit{plus}(m, n) = m + n$. □

Um uns von der Mächtigkeit der *while*-Programmiersprache zu überzeugen, überlegen wir uns, wie wir einen Interpreter *für* *while*-Programme *in* der *while*-Programmiersprache schreiben können.

Ein *Interpreter* für *while*-Programme der Stelligkeit k ist ein *while*-Programm I^k, das für jede Eingabe $(m, n_1, \ldots, n_k) \in \mathbb{N}^{k+1}$ die Ausführung des *while*-Programms mit Gödelnummer m auf der Eingabe $(n_1, \ldots, n_k)$ simuliert.

Eine solche Simulation können wir durchführen, indem wir uns an unserer formalen Semantik für *while*-Programme orientieren.

Deshalb codieren wir nicht nur *while*-Programme als natürliche Zahlen, sondern auch Anweisungen, Anweisungslisten und Speicherzustände.

Als ersten Schritt in diese Richtung überlegen wir uns, wie man eine endliche Liste $n_1, \ldots, n_k$ von natürlichen Zahlen als eine einzige Zahl $\langle n_1, \ldots, n_k \rangle \in \mathbb{N}$ codiert.

Wir nutzen dazu die Tatsache aus, dass jede natürliche Zahl eine eindeutige Primfaktorzerlegung besitzt, also eine eindeutige Darstellung als Produkt von Primzahlpotenzen.

Sei $p_1, p_2, \ldots$ die Folge *aller* Primzahlen in aufsteigender Reihenfolge, also $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, ...

Dann codieren wir jede Liste $n_1, \ldots, n_k$ natürlicher Zahlen als die Zahl

$$\langle n_1, \ldots, n_k \rangle = \prod_{i=1}^{k} p_i^{n_i+1}$$

z.B. gilt $\langle 3, 0, 2, 0 \rangle = 2^4 * 3^1 * 5^3 * 7^1 = 42000$.

Man beachte, dass man aus der *Zahl* $\langle n_1, \ldots, n_k \rangle$ stets die *Liste* der Zahlen $n_1, \ldots, n_k$ rekonstruieren kann. Wegen der Eindeutigkeit der Primfaktorzerlegung gilt nämlich

- $k = \max\{i \in \mathbb{N} \mid p_i \text{ ist Teiler von } \langle n_1, \ldots, n_k \rangle\}$
- $n_i = \max\{n \in \mathbb{N} \mid p_i^{n+1} \text{ ist Teiler von } \langle n_1, \ldots, n_k \rangle\}$

Die Codierung von Zahlenlisten können wir verwenden, um Speicherzustände, Anweisungen, Anweisungslisten und schließlich ganze Programme zu codieren.

Für jeden Speicherzustand σ und jede Zahl $m \in \mathbb{N}$ definieren wir die *m-stellige Codierung* von σ als die Zahl

$$\langle \sigma \rangle_m = \langle \sigma(X_0), \ldots, \sigma(X_m) \rangle$$

Diese Art der Codierung genügt, weil jedes *while*-Programm nur auf endlich vielen Speicherplätzen arbeitet. Wenn nur $X_0, \ldots, X_m$ in P vorkommen, dann genügt es, während der Ausführung des Programms P die Werte $\sigma(X_0), \ldots, \sigma(X_m)$ bzw. deren Codierung $\langle \sigma \rangle_m$ zu kennen.

Beispiele:

- $\langle \sigma_0 \rangle_3 = \langle 0, 0, 0, 0 \rangle = 2^1 * 3^1 * 5^1 * 7^1 = 210$
- $\langle \sigma_0[1/X_0][2/X_2] \rangle_3 = \langle 1, 0, 2, 0 \rangle = 2^2 * 3^1 * 5^3 * 7^1 = 10500$

Für alle Anweisungen *st* und Anweisungslisten *stl* definieren wir die Codierungen $\lceil st \rceil \in \mathbb{N}$ und $\lceil stl \rceil \in \mathbb{N}$ durch folgende Induktion über die Länge von *st* bzw. *stl*:

- $\lceil X_i := 0 \rceil = \langle 0, i \rangle$
- $\lceil X_i := X_j \rceil = \langle 1, i, j \rangle$
- $\lceil X_i := succ(X_j) \rceil = \langle 2, i, j \rangle$
- $\lceil X_i := pred(X_j) \rceil = \langle 3, i, j \rangle$
- $\lceil$if X_i then stl_1 else stl_2 fi$\rceil = \langle 4, i, \lceil stl_1 \rceil, \lceil stl_2 \rceil \rangle$
- $\lceil$while X_i do stl od$\rceil = \langle 5, i, \lceil stl \rceil \rangle$
- $\lceil st_1; \ldots st_n; \rceil = \langle \lceil st_1 \rceil, \ldots, \lceil st_n \rceil \rangle$ für alle $n \in \mathbb{N}$

  (d.h. im Falle $n = 0$: $\lceil \varepsilon \rceil = \langle\,\rangle = \prod_{i=1}^{0} p_i^{n_i} = 1$)

Schließlich definieren wir die Codierung $\lceil P \rceil$ eines Programms P durch

$$\lceil \text{read } X_{i_1}, \ldots, X_{i_k}; \mathit{stl} \text{ write } X_i \rceil = \langle \lceil \mathit{stl} \rceil, m, i, i_1, \ldots, i_k \rangle$$

wobei $m = \max\{j \in \mathbb{N} \mid X_j \text{ kommt in } P \text{ vor}\}$

Beispiel:

$$\begin{aligned}
&\lceil \text{read } X_0, X_1; \\
&\text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od}; \\
&\text{write } X_1 \rceil \\
&= \langle \lceil \text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od}; \rceil, 1, 1, 0, 1 \rangle \\
&= \langle \langle \lceil \text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od} \rceil \rangle, 1, 1, 0, 1 \rangle \\
&= \langle \langle \langle 5, 0, \lceil X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1); \rceil \rangle \rangle, 1, 1, 0, 1 \rangle \\
&= \langle \langle \langle 5, 0, \langle \lceil X_0 := \mathit{pred}(X_0) \rceil, \lceil X_1 := \mathit{succ}(X_1) \rceil \rangle \rangle \rangle, 1, 1, 0, 1 \rangle \\
&= \langle \langle \langle 5, 0, \langle \langle 3, 0, 0 \rangle, \langle 2, 1, 1 \rangle \rangle \rangle \rangle \rangle, 1, 1, 0, 1 \rangle \\
&= \langle \langle \langle 5, 0, \langle 240, 1800 \rangle \rangle \rangle, 1, 1, 0, 1 \rangle \ = \ \ldots
\end{aligned}$$

Der Interpreter

Kernstück des Interpreters ist ein Programmstück $\mathit{Step}(X_i, X_j, X_k)$, das die Übergangsschritt-Relation simuliert, d.h.: Wenn

- X_i eine Zahl $m \in \mathbb{N}$ enthält
- X_j die Codierung $\lceil \mathit{stl} \rceil$ einer *nichtleeren* Anweisungsliste *stl*
- und X_k die m-stellige Codierung $\langle \sigma \rangle_m$ eines Zustands σ

dann soll nach Ausführung von $\mathit{Step}(X_i, X_j, X_k)$ gelten:

- X_i enthält immer noch m
- X_j enthält $\lceil \mathit{stl}' \rceil$
- X_k enthält $\langle \sigma' \rangle_m$

wobei (stl', σ') die Nachfolgekonfiguration von (stl, σ) ist, d.h. die eindeutige Konfiguration mit $(\mathit{stl}, \sigma) \vdash (\mathit{stl}', \sigma')$.

Wir wollen skizzieren, wie man $\mathit{Step}(X_i, X_j, X_k)$ nach und nach aus kleineren Programmstücken zusammensetzt. Es werden benötigt

- Programmstücke für die Grundrechenarten Addition, Subtraktion, Multiplikation, Potenzieren,. . .
- Programmstücke für Abfragen auf Gleichheit, $<$-Beziehung, Teilbarkeit, Primzahltest,. . .
- Programmstücke zur Codierung und Decodierung von Listen, z.B.

  - eines, das die i-te Primzahl liefert,

  - eines, das den Exponenten der Primzahl p in der Primfaktorzerlegung einer Zahl n liefert,

- Programmstücke, die auf den Codierungen von Anweisungslisten und Zuständen arbeiten, z.B.

  - eines, das aus $m, \langle\sigma\rangle_m, n$ und i die Zahl $\langle\sigma[n/X_i]\rangle_m$ berechnet,

  - eines, das aus $\lceil \mathit{stl}_1 \rceil$ und $\lceil \mathit{stl}_2 \rceil$ die Zahl $\lceil \mathit{stl}_1 \mathit{stl}_2 \rceil$ berechnet.

Problem:

while-Programme bieten keinerlei Unterstützung für den modularen Programmaufbau, insbesondere

1. keine Prozeduren (mit vernünftiger Parameterübergabe),
2. keine lokalen Variablen.

2. führt dazu, dass stets die ganze Information, die man im Programm mitführt, sichtbar ist (kein 'information hiding').

Deshalb muss man sich als Programmierer selbst überlegen, wie man Programme organisiert, d.h. man muss sich eine gewisse 'Programmierdisziplin' ausdenken und diese streng befolgen.

Beispiel:

Wir hatten bereits ein Programm(stück) zur Addition, nämlich

$$\text{while } X_0 \text{ do } X_0 := \mathit{pred}(X_0);\ X_1 := \mathit{succ}(X_1);\ \text{od};$$

Dieses Programmstück ist nicht gut wiederverwendbar, da es die Inhalte der Eingabe-Speicherplätze X_0 und X_1 verändert.

Eine bessere Lösung ist das Programmstück

$$Y_0 := X_i;\ X_k := X_j;$$
$$\text{while } Y_0 \text{ do } Y_0 := \mathit{pred}(Y_0);\ X_k := \mathit{succ}(X_k);\ \text{od};$$

das wir mit $\mathit{Add}(X_i, X_j, X_k)$ bezeichnen. Dabei seien X_i, X_j, X_k, Y_0 *verschiedene* (aber ansonsten beliebige) Speicherplätze.

$\mathit{Add}(X_i, X_j, X_k)$ benutzt die Speicherplätze X_i und X_j zur Eingabe (und lässt sie unverändert) und den Speicherplatz X_k zur Ausgabe. Y_0 dient als 'Hilfs-Speicherplatz'.

Da Y_0 im Gesamtprogramm sichtbar ist, muss man darauf achten, dass der Inhalt von Y_0 nicht für andere Programmstücke von Bedeutung ist (d.h. man kann Y_0 durchaus in anderen Teilen des Programms wiederverwenden, aber man muss eben damit rechnen, dass sein Inhalt durch $\mathit{Add}(X_i, X_j, X_k)$ verändert wird).

Deshalb merken wir uns die folgende 'Spezifikation':

- $\mathit{Add}(X_i, X_j, X_k)$ wirkt wie eine Zuweisung $X_k := X_i + X_j$ (die es in der *while*-Programmiersprache so nicht gibt).
- Neben X_i, X_j, X_k benutzt es nur den Speicherplatz Y_0.

Unter den Hilfs-Speicherplätzen Y_i kann man sich Speicherplätze in einem bestimmten 'Speicherbereich' vorstellen, den man sonst nicht benötigt, z.B $Y_i = X_{i+100}$.

Natürlich lässt sich die Semantik des Programmstücks $\mathit{Add}(X_i, X_j, X_k)$ auch formal beschreiben:

Für alle $\sigma \in \mathit{Store}$ gilt

$$(\mathit{Add}(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma[\sigma(X_i) + \sigma(X_j)/X_k][0/Y_0]) \qquad \text{(ADD)}$$

Um (ADD) zu beweisen, formuliert man zunächst eine geeignete Aussage über die *while*-Schleife, nämlich:

Für alle $\sigma \in \mathit{Store}$ gilt

$$\begin{aligned}&(\text{while } Y_0 \text{ do } Y_0 := \mathit{pred}(Y_0);\ X_k := \mathit{succ}(X_k);\ \text{od};,\ \sigma)\\ &\quad \vdash^*\ (\varepsilon,\ \sigma[0/Y_0][\sigma(Y_0) + \sigma(X_k)/X_k])\end{aligned} \qquad (*)$$

(*) ergibt sich—analog zum alten Additionsprogramm—durch Induktion über $\sigma(Y_0)$. Aus (*) erhält man dann (ADD), indem man noch die Semantik der Zuweisungen $Y_0 := X_i$; $X_k := X_j$; berücksichtigt.

Alternative:

Da der Endinhalt des Hilfs-Speicherplatzes Y_0 nicht interessiert, kann man (ADD) durch eine schwächere Aussage ersetzen, etwa:

Für jedes $\sigma \in \mathit{Store}$ existiert ein $\sigma' \in \mathit{Store}$ mit

$$(\mathit{Add}(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) + \sigma(X_j)/X_k])$$
und σ' unterscheidet sich von σ nur in Y_0. (ADD')

Entsprechend schwächt man die Induktionsbehauptung (*) ab. Der Induktionsbeweis geht dann immer noch durch, und die schwächere Aussage (ADD') genügt, um Aussagen über Programme zu beweisen, die $\mathit{Add}(X_i, X_j, X_k)$ verwenden.

Weitere nützliche Programmstücke:

Für die *Subtraktion* benutzen wir das Programmstück

$$Y_0 := X_j;\ X_k := X_i;$$

$$\text{while } Y_0 \text{ do } Y_0 := \mathit{pred}(Y_0);\ X_k := \mathit{pred}(X_k);\ \text{od};$$

das wir mit $\mathit{Sub}(X_i, X_j, X_k)$ bezeichnen. Es wirkt wie eine Zuweisung $X_k := X_i \dot{-} X_j$ und verändert ansonsten nur den Inhalt von Y_0.

Formale Semantik:
Für alle $\sigma \in \mathit{Store}$ gilt

$$(\mathit{Sub}(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma[\sigma(X_i) \dot{-} \sigma(X_j)/X_k][0/Y_0]) \qquad \text{(SUB)}$$

Schwächere Spezifikation:
Für jedes $\sigma \in \mathit{Store}$ existiert ein $\sigma' \in \mathit{Store}$ mit

$$(\mathit{Sub}(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) \dot{-} \sigma(X_j)/X_k]) \qquad \text{(SUB')}$$

und σ' unterscheidet sich von σ nur in Y_0.

Ein Programmstück $\mathit{Mul}(X_i, X_j, X_k)$ zur Multiplikation erhält man durch Wiederverwendung von $\mathit{Add}(X_i, X_j, X_k)$. Es sei definiert durch

$Y_1 := X_j$; $X_k := 0$;

while Y_1 do $Y_1 := \mathit{pred}(Y_1)$; $Y_2 := X_k$; $\mathit{Add}(Y_2, X_i, X_k)$ od;

Hier werden *neue* Hilfs-Speicherplätze Y_1, Y_2 verwendet. Y_0 zu verwenden wäre falsch, da sein Inhalt durch $\mathit{Add}(\ldots)$ verändert wird.

Spezifikation:

Für jedes $\sigma \in \mathit{Store}$ existiert ein $\sigma' \in \mathit{Store}$ mit:

$$(\mathit{Mul}(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) * \sigma(X_j)/X_k]) \quad \text{(MUL)}$$

und σ' unterscheidet sich von σ nur in Y_0, Y_1, Y_2.

Um (MUL) zu beweisen, formuliert man wieder eine geeignete Induktionsbehauptung für die *while*-Schleife und verwendet dann die Spezifikation (ADD') für das Programmstück $\mathit{Add}(Y_2, X_i, X_k)$.

Mit der Subtraktion lassen sich *Vergleichsoperatoren* implementieren:

Less(X_i, X_j, X_k) sei das Programmstück

$$\mathit{Sub}(X_j, X_i, X_k)$$

Es testet, ob $\sigma(X_i) < \sigma(X_j)$ ist, und zwar in folgendem Sinne: Wenn ja, so liefert es eine Zahl $n > 0$ (= '*true*') im Speicherplatz X_k ab, andernfalls liefert es die Zahl 0 (= '*false*') ab.

Leq(X_i, X_j, X_k) sei das Programmstück

$$Y_1 := \mathit{succ} X_j;\ \mathit{Less}(X_i, Y_1, X_k)$$

Es testet, ob $\sigma(X_i) \leq \sigma(X_j)$ ist.

Schließlich sei *Eq*(X_i, X_j, X_k) das Programmstück

$$\mathit{Leq}(X_i, X_j, X_k);\ \text{if } X_k \text{ then } \mathit{Leq}(X_j, X_i, X_k) \text{ else fi};$$

Es testet, ob $\sigma(X_i) = \sigma(X_j)$ ist.

Mit Hilfe der Vergleichsoperatoren können wir jetzt *Teilbarkeit* testen.

Um zu überprüfen, ob m ein Teiler von n ist, berechnet man die Vielfachen $0 * m, 1 * m, 2 * m, ...$, die nicht größer als n sind, und überprüft, ob eine dieser Zahlen mit n übereinstimmt.

Auf dieser Idee basiert das Programmstück $\mathit{Divides}(X_i, X_j, X_k)$:

$Y_1 := 0$; $Y_2 := \mathit{succ}(Y_1)$;

while Y_2

do $\mathit{Eq}(Y_1, X_j, X_k)$

if X_k then $Y_2 := 0$; else $\mathit{Leq}(Y_1, X_j, Y_2)$ fi

if Y_2 then $Y_3 := Y_1$; $\mathit{Add}(Y_3, X_i, Y_1)$ else fi

od

Es testet, ob $\sigma(X_i)$ ein Teiler von $\sigma(X_j)$ ist (unter der Voraussetzung $\sigma(X_i) \neq 0$).

Erläuterung der Arbeitsweise von *Divides*(X_i, X_j, X_k):

Sei $\sigma(X_i) = m \neq 0$ und $\sigma(X_j) = n$. Der Speicherplatz Y_1 enthält stets das aktuelle Vielfache von m, das mit n verglichen wird. Er wird mit 0 vorbesetzt und in jedem Schleifendurchlauf um m erhöht.

Im Speicherplatz Y_2 merkt man sich, ob die Suche nach dem passenden Vielfachen von m noch fortgesetzt werden muss. Deshalb wird er mit 1 (= '*true*) vorbesetzt. Sein Inhalt wird zu 0 verändert

- entweder durch $Y_2 := 0$, wenn mit *Eq*(Y_1, X_j, X_k) festgestellt wurde, dass das aktuelle Vielfache von m mit n übereinstimmt,
- oder durch *Leq*(Y_1, X_j, Y_2), wenn das aktuelle Vielfache von m bereits größer als n ist.

Das Programmstück terminiert, weil einer dieser beiden Fälle irgendwann eintritt. In beiden Fällen wird X_k durch den letzten Test *Eq*(Y_1, X_j, X_k) mit der korrekten Antwort besetzt.

Nach diesen Beispielen dürfte es klar sein, wie man größere *while*-Programme nach und nach aus kleinen Bausteinen zusammensetzen kann. Deshalb geben wir für alle weiteren Bausteine unseres Interpreters nur noch die Idee an:

Mit Hilfe von *Divides*(X_i, X_j, X_k) programmiert man einen *Primzahltest*: Eine Zahl $n \geq 2$ ist genau dann eine Primzahl, wenn man unter den Zahlen $2, \ldots, n-1$ keinen Teiler von n findet.

Mit Hilfe des Primzahltests schreibt man ein Programm, das zu jeder Zahl $i > 0$ die *i-te Primzahl* p_i findet: Man führt den Primzahltest für $2, 3, 4, \ldots$ aus, bis er zum i-ten Mal erfolgreich ist.

Damit erhält man leicht ein Programm, das zu zwei Zahlen i und n die *Primzahl-Potenz* p_i^n berechnet.

Indem man dieses Programm mit dem Teilbarkeitstest kombiniert, kann man zu zwei Zahlen $i, n \neq 0$ die größte Zahl k berechnen, für die p_i^k Teiler von n ist. Diese Zahl k ist der *Exponent* von p_i in der Primfaktorzerlegung von n.

Damit können wir die *Länge* und die einzelnen *Komponenten* einer codierten Zahlenliste bestimmen: Wenn $n = \langle n_1, \ldots, n_m \rangle$, so ist $m = i - 1$, wobei p_i die erste Primzahl ist, die n nicht teilt, und $n_i = k - 1$, wobei k der Exponent von p_i in der Primfaktorzerlegung von n ist.

Insbesondere erhalten wir den *Inhalt eines Speicherplatzes* aus einem codierten Zustand: Wenn $n = \langle \sigma \rangle_m = \langle \sigma(X_0), \ldots, \sigma(X_m) \rangle$, so ist $\sigma(X_i)$ die $(i+1)$-te Komponente der durch n codierten Liste.

Zur Simulation des Übergangsschrittes müssen wir uns jetzt noch überlegen

1. wie man einen Zustand an einer Stelle verändert, d.h. wie man aus der Codierung von σ die Codierung von $\sigma[n/X_k]$ erhält,
2. wie man aus einer Anweisungsliste die erste Anweisung abspaltet, d.h. wie man $\lceil st \rceil$ und $\lceil stl \rceil$ aus $\lceil st; stl \rceil$ erhält,
3. wie man Anweisungslisten aneinanderhängt, d.h. wie man $\lceil stl_1 stl_2 \rceil$ aus $\lceil stl_1 \rceil$ und $\lceil stl_2 \rceil$ erhält.

Die *Veränderung eines Zustands* führt man so durch:

Gegeben seien $m, n, k \in \mathbb{N}$ und die Codierung $\langle\sigma\rangle_m$ eines Zustands σ. Zu berechnen ist die Codierung $\langle\sigma'\rangle_m$ des veränderten Zustands $\sigma' = \sigma[n/X_k]$. Per Definition der Codierung gilt

$$\langle\sigma'\rangle_m = \langle\sigma'(X_0), \ldots \sigma'(X_m)\rangle = \prod_{i=0}^{m} p_{i+1}^{\sigma'(X_i)+1}$$

Man berechnet dieses Produkt, indem man für $i = 0, \ldots, m$

- p_{i+1} bestimmt,
- $\sigma(X_i)$ aus $\langle\sigma\rangle_m$ errechnet,
- im Falle $i = k$ die Potenz p_i^{n+1} und im Falle $i \neq k$ die Potenz $p_i^{\sigma(X_i)+1}$ berechnet,
- und das Zwischenergebnis mit dieser Potenz multipliziert.

Dies kann man in einer *while*-Anweisung durchführen.

Aneinanderhängen von Anweisungslisten:

Anweisungslisten sind ja als Zahlenlisten codiert, also muss man aus

$$\langle n_1, \ldots n_k \rangle = \prod_{i=1}^{k} p_i^{n_i+1} \quad \text{und} \quad \langle m_1, \ldots, m_l \rangle = \prod_{i=1}^{l} p_i^{m_i+1}$$

die Zahl

$$\langle n_1, \ldots n_k, m_1, \ldots, m_l \rangle = \prod_{i=1}^{k} p_i^{n_i+1} * \prod_{i=1}^{l} p_{k+i}^{m_i+1}$$

errechnen. Das lässt sich ähnlich durchführen wie die Veränderung eines Zustands.

Aufspalten einer Anweisungsliste:

Hier muss man aus $\langle n_1, \ldots n_k \rangle$ die erste Zahl n_1 und die Codierung

$$\langle n_2, \ldots n_k \rangle = \prod_{i=2}^{k} p_{i-1}^{n_i+1}$$

errechnen. Auch das lässt sich wieder ähnlich durchführen.

Damit haben wir jetzt alle Bausteine, die wir für das Programmstück *Step*(X_i, X_j, X_k) (zur Simulation von Übergangsschritten) benötigen. *Step*(X_i, X_j, X_k) arbeitet wie folgt:

- Aus X_i erhält es eine Zahl m, aus X_j die Codierung $\lceil st; stl \rceil$ einer nichtleeren Anweisungsliste und aus X_k die Codierung $\langle \sigma \rangle_m$ des aktuellen Zustands.
- $\lceil st; stl \rceil$ wird aufgespalten in $\lceil st \rceil$ und $\lceil stl \rceil$. $\lceil st \rceil$ ist wieder Codierung einer Zahlenliste. Aus der ersten Zahl i dieser Liste kann man ablesen, um welche Art von Anweisung es sich handelt.
- Im Falle $0 \leq i \leq 3$ ist *st* eine Zuweisung. Dann liest man aus den restlichen Komponenten ab, wie die linke und rechte Seite der Zuweisung aussehen und errechnet aus $\langle \sigma \rangle_m$ die Codierung $\langle \sigma' \rangle_m$ des neuen Zustands σ'. Schließlich speichert man die Codierung $\lceil stl \rceil$ der noch auszuführenden Anweisungsliste in X_j ab und die Codierung $\langle \sigma' \rangle_m$ des neuen Zustands in X_k.

- Im Falle $i = 4$ ist *st* von der Form if X_n then *stl*$_1$ else *stl*$_2$ fi. Dann liest man aus den restlichen Komponenten den Index n und die Codierungen $\lceil$*stl*$_1\rceil$ und $\lceil$*stl*$_2\rceil$ des then- und else-Teils ab. Aus $\langle\sigma\rangle_m$ und n errechnet man $\sigma(X_n)$. Ist $\sigma(X_n) = 0$, so errechnet man die Zahl $\lceil$*stl*$_1$ *stl*$\rceil$ aus $\lceil$*stl*$_1\rceil$ und $\lceil$*stl*$\rceil$ und speichert sie in X_j ab. Andernfalls speichert man $\lceil$*stl*$_2$ *stl*$\rceil$ in X_j ab. Den Inhalt von X_k $(= \langle\sigma\rangle_m)$ lässt unverändert.
- Im Falle $i = 5$ ist *st* von der Form while X_n do *stl*$_1$ od. Auch hier berechnet man wieder $\sigma(X_n)$ und speichert je nach Ergebnis entweder $\lceil$*stl*$_1$ *stl*$\rceil$ oder $\lceil$*stl*$\rceil$ in X_j ab. Den Inhalt von X_k lässt man wieder unverändert.

Mit dem Programmstück *Step*(X_i, X_j, X_k) erhält man dann leicht den eigentlichen Interpreter I^k (für k-stellige Funktionen), der wie folgt arbeitet:

Als Eingabe erhält er die Codierung (= Gödelnummer)

$$\lceil P \rceil = \langle \lceil \mathit{stl} \rceil, m, i, i_1, \ldots, i_k \rangle$$

eines Programms P und die Eingabewerte $n_1, \ldots, n_k$ für P. Aus $m, i_1, \ldots, i_k$ und $n_1, \ldots, n_k$ errechnet er die Codierung $\langle \sigma \rangle$ des Anfangszustands $\sigma = \sigma_0[n_1/X_{i_0}] \ldots [n_k/X_{i_k}]$. Dann simuliert er den Ablauf der Anweisungsliste *stl* für diesen Anfangszustand σ mit Hilfe von *Step*(X_i, X_j, X_k). Wenn *stl* abgearbeitet ist (d.h. wenn nur noch die leere Anweisungsliste übrig ist) berechnet er den Inhalt von X_i aus der Codierung des Endzustands und gibt ihn aus.

Berechenbarkeitstheorie

Wozu brauchen wir den Interpreter?
In der Berechenbarkeitstheorie haben wir an einigen Stellen die Existenz eines Interpreters vorausgesetzt, z.B.

- beim Beweis der Semi-Entscheidbarkeit des Halteproblems,
- beim Beweis, dass die Menge $H_\forall$ *nicht* semi-entscheidbar ist.

Aber auch an anderen Stellen haben wir Annahmen über die Mächtigkeit unserer Programmiersprache gemacht, die sich am besten mit einem Interpreter nachweisen lassen, z.B.

- die 'Parallelausführung' zweier Programme auf dem gleichen Eingabewert,
- die 'Parallelausführung' eines Programms auf unendlich vielen Eingabewerten mit der *dovetailing*-Technik.

Fazit:

Wir haben gezeigt, dass selbst eine so einfache Programmiersprache wie die *while*-Programme 'hinreichend mächtig' im Sinne der Berechenbarkeitstheorie ist.

Damit ist die Berechenbarkeitstheorie erst recht auf alle realistischen (imperativen) Programmiersprachen anwendbar, denn die enthalten ja *while*-Programme als Teilmenge. Insbesondere wissen wir, dass in all diesen Sprachen das Halteproblem unentscheidbar ist (und nach dem Satz von Rice sogar jedes nichttriviale semantische Problem).

Bevor wir alternative Ansätze zur Definition von Berechenbarkeit untersuchen, betrachten wir eine *schwächere* Programmiersprache, nämlich die sogenannten *loop-Programme*. Die *loop*-Programmiersprache erhält man aus der *while*-Programmiersprache, indem man die Produktion

$$st ::= \text{while } X_i \text{ do } stl \text{ od}$$

ersetzt durch

$$st ::= \text{loop } X_i \text{ do } stl \text{ od}$$

Der Übergangsschritt für diese *loop*-Anweisungen ist definiert durch

$$(\text{loop } X_i \text{ do } stl \text{ od}, \sigma) \vdash (\underbrace{stl \ldots stl}_{m}, \sigma) \quad \text{falls } \sigma(X_i) = m$$

Man beachte, dass sich die Zahl m aus dem Anfangsinhalt von X_i ergibt, d.h. auch wenn X_i in *stl* vorkommt, wird *stl* nur m-mal ausgeführt, also kann insbesondere keine Endlosschleife entstehen.

Die von einem *loop*-Programm P berechnete Funktion $[\![P]\!]$ ist genau wie bei *while*-Programmen definiert. Eine Funktion f heißt *loop-berechenbar*, wenn es ein *loop*-Programm P gibt mit $[\![P]\!] = f$.

Satz 3.26 *Jede loop-berechenbare Funktion ist while-berechenbar.*

Beweis: Die Anweisung loop X_i do *stl* od lässt sich in der *while*-Programmiersprache simulieren durch

$$Y_0 := X_i;\ \text{while } Y_0 \text{ do } Y_0 := \mathit{pred}(Y_0);\ \mathit{stl} \text{ od}$$

wobei Y_0 ein Speicherplatz ist, der sonst nirgends vorkommt. □

Fast alle Funktionen, die wir mit *while*-Programmen berechnet haben, sind auch *loop*-berechenbar, denn immer wenn sich die Schrittzahl einer *while*-Schleife von vornherein nach oben abschätzen lässt, kann man auch eine *loop*-Schleife verwenden.

Die Umkehrung von Satz 3.26 gilt allerdings nicht, d.h. die *loop*-Programmiersprache ist *nicht* Turing-mächtig. Es gilt nämlich

Satz 3.27 *Jede loop-berechenbare Funktion ist total.*

Beweis: Es genügt zu zeigen, dass eine Anweisungsliste *stl* immer terminiert, d.h. dass für *jeden* Zustand σ ein Zustand σ' existiert mit $(stl, \sigma) \vdash^* (\varepsilon, \sigma')$. Das folgt leicht durch Induktion über die Länge von *stl*. Insbesondere nutzt man beim Induktionsschritt für loop X_i do *stl* od aus, dass $\underbrace{stl \ldots stl}_{m}$ für alle Zustände terminiert, wenn *stl* für alle Zustände terminiert. □

Im übrigen gibt es sogar totale *while*-berechenbare Funktionen, die nicht *loop*-berechenbar sind. Ein konkretes Beispiel ist die sogenannte *Ackermann*-Funktion (vgl. Literatur).

Wir betrachten nun den zweiten Ansatz zur Definition von Berechenbarkeit, die sogenannten μ-rekursiven Funktionen. Bei diesem Ansatz kommt man im Prinzip *ohne* Programmiersprache aus. Andererseits wird eine präzise mathematische Schreibweise eingeführt, die man durchaus als Programmiersprache auffassen kann.

Man nennt eine Funktion *primitiv rekursiv*, wenn sie sich aus gewissen *Grundfunktionen* durch (wiederholte) Anwendung der Operatoren

1. *Substitution* und
2. *primitive Rekursion*

aufbauen lässt.

Man nennt sie *μ-rekursiv*, wenn neben 1. und 2. zusätzlich noch der *μ-Operator* zugelassen ist.

Diese neuen Begriffe werden wir jetzt nach und nach präzise definieren.

Zu den *Grundfunktionen* (oder: *Basisfunktionen*) gehören

1. für jedes $k, c \in \mathbb{N}$ die *konstante Funktion*

$$const_c^k : \mathbb{N}^k \to \mathbb{N}$$
$$(n_1, \ldots, n_k) \mapsto c$$

2. für jedes $i, k \in \mathbb{N}$ mit $1 \leq i \leq k$ die *Projektion*

$$proj_i^k : \mathbb{N}^k \to \mathbb{N}$$
$$(n_1, \ldots, n_k) \mapsto n_i$$

3. die *Nachfolgerfunktion* $succ : \mathbb{N} \to \mathbb{N}$

$$n \mapsto n + 1$$

Man beachte, dass bei den konstanten Funktionen die Stelligkeit 0 zugelassen ist und bei den Projektionen die Stelligkeit 1: $const_c^0$ ist nichts anderes als die Zahl c, und $proj_1^1$ ist die Identität auf $\mathbb{N}$.

Substitution ist eine Verallgemeinerung der Komposition von Funktionen.

Definition 3.28 *Sei $k \in \mathbb{N}$. Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ entsteht durch Substitution (oder Einsetzung) aus den Funktionen $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ und $h_1, \ldots, h_l : \mathbb{N}^k \hookrightarrow \mathbb{N}$, wenn für alle $n_1, \ldots, n_k \in \mathbb{N}$ gilt:*

$$f(n_1, \ldots, n_k) \;=\; g\,(h_1(n_1, \ldots, n_k), \ldots, h_l(n_1, \ldots, n_k)) \qquad (*)$$

Die Funktion f bezeichnet man dann mit $Sub\,(g; h_1, \ldots, h_l)$.

Gleichung (*) ist übrigens so zu verstehen, dass $f(n_1, \ldots, n_k)$ undefiniert ist, sobald einer der Werte $h_i(n_1, \ldots, n_k)$ undefiniert ist, und zwar selbst dann, wenn die Funktion g nicht alle Argumente benötigt, z.B. wenn g eine konstante Funktion oder eine Projektion ist.

Im Fall $l = 1$ vereinfacht sich (*) zu

$$f(n_1, \ldots, n_k) \;=\; g\,(h\,(n_1, \ldots, n_k))$$

es gilt also $\mathit{Sub}\,(g;\, h) = g \circ h$.

Auch im Fall $l > 1$ kann man $\mathit{Sub}\,(g;\, h_1, \ldots, h_l)$ als Komposition von g mit einer geeigneten Funktion h auffassen, nämlich

$$h : \mathbb{N}^k \hookrightarrow \mathbb{N}^l$$

$$(n_1, \ldots, n_k) \mapsto (h_1(n_1, \ldots, n_k), \ldots, h_l(n_1, \ldots, n_k))$$

Diese Funktion h bezeichnet man manchmal mit $\langle h_1, \ldots, h_l \rangle$. Dann gilt also $\mathit{Sub}\,(g;\, h_1, \ldots, h_l) = g \circ \langle h_1, \ldots, h_l \rangle$.

Warnung:
Die gleiche Schreibweise haben wir für die Codierung von Zahlenlisten benutzt. Um eine Verwechslung auszuschließen, sollte man also stets wissen, ob man Zahlen oder Funktionen vor sich hat.

Beispiele:

1. Sei $mul : \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto m * n$ die Multiplikation

   und $square : \mathbb{N} \to \mathbb{N},\ n \mapsto n^2$ die Quadratfunktion.

   Dann gilt für alle $n \in \mathbb{N}$

   $$square\,(n) = mul\,(n,n) = mul\,(proj_1^1(n), proj_1^1(n))$$

   also ist $square = Sub\,(mul;\ proj_1^1, proj_1^1)$.

2. Sei $add : \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto m + n$ die Addition

   und $sos : \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto m^2 + n^2$ ('*s*um *o*f *s*quares')

   Dann gilt für alle $(m,n) \in \mathbb{N}^2$

   $$\begin{aligned} sos(m,n) &= add\,(square\,(m), square\,(n)) \\ &= add\,(square\,(proj_1^2(m,n)), square\,(proj_2^2(m,n))) \end{aligned}$$

   also ist $sos = Sub\,(add;\ square \circ proj_1^2, square \circ proj_2^2)$

Primitive Rekursion ist ein 'Schema' zur induktiven Definition von Funktionen.

Definition 3.29 *Sei* $k \in \mathbb{N}$. *Eine Funktion* $f : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ *entsteht durch* primitive Rekursion *aus den Funktionen* $g : \mathbb{N}^k \hookrightarrow \mathbb{N}$ *und* $h : \mathbb{N}^{k+2} \hookrightarrow \mathbb{N}$, *wenn für alle* $n_1, \ldots, n_k, m \in \mathbb{N}$ *gilt:*

$$\begin{aligned} f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k) \\ f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \end{aligned}$$

Die Funktion f *bezeichnet man dann mit* $\mathit{Prim}(g, h)$.

Man beachte, dass die Induktion nur über den letzten Parameter m von f laufen darf. Man bezeichnet m als *Rekursionsparameter* und $n_1, \ldots, n_k$ als *Nebenparameter*. Die Funktion g liefert den Wert für den Induktionsanfang (in Abhängigkeit von den Nebenparametern), und die Funktion h beschreibt den Induktionsschritt.

Im Spezialfall $k = 0$ ist g eine 0-stellige Funktion, d.h. $g \in \mathbb{N}$. Dann vereinfacht sich das Schema der primitiven Rekursion zu:

$$\begin{aligned} f(0) &= g \\ f(m+1) &= h\,(m, f(m)) \end{aligned}$$

Beispiele:

1. Sei $\mathit{pred} : \mathbb{N} \to \mathbb{N},\ m \mapsto m \dot{-} 1$ die Vorgängerfunktion.

   Dann ist

   $$\mathit{pred}\,(0) \;=\; 0 \;=\; \mathit{const}_0^0$$

   und für alle $m \in \mathbb{N}$ gilt

   $$\mathit{pred}\,(m+1) \;=\; m \;=\; \mathit{proj}_1^2\,(m,\, \mathit{pred}\,(m))$$

   Also ist $\mathit{pred} = \mathit{Prim}\,(\mathit{const}_0^0;\, \mathit{proj}_1^2)$.

2. Für die Addition $add : \mathbb{N}^2 \to \mathbb{N}$ gilt

$$add\,(n, 0) \;=\; n \;=\; proj_1^1(n)$$

und

$$\begin{aligned} add\,(n, m+1) &= n + (m+1) \\ &= (n+m)+1 \\ &= succ\,(add\,(n,m)) \\ &= succ\,(proj_3^3(n,\, m,\, add\,(n,m))) \\ &= (succ \circ proj_3^3)\,(n,\, m,\, add\,(n,m)) \end{aligned}$$

Also ist

$$\begin{aligned} add &= Prim\,(proj_1^1;\, succ \circ proj_3^3) \\ &= Prim\,(proj_1^1;\, Sub\,(succ;\, proj_3^3)) \end{aligned}$$

Definition 3.30 *Die Menge $\mathcal{PR}$ der primitiv rekursiven Funktionen ist die kleinste Menge von Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ (für alle $k \in \mathbb{N}$), die*

- *alle Grundfunktionen enthält und*
- *abgeschlossen ist unter Substitution und primitiver Rekursion.*

Mit anderen Worten:

Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist genau dann primitiv rekursiv, wenn sie durch (wiederholte) Anwendung der Operatoren *Sub* und *Prim* aus den Grundfunktionen $const_c^k$, $proj_i^k$ und *succ* entsteht.

Beispiele:

1. *pred* ist primitiv rekursiv wegen $pred = Prim\,(const_0^0;\, proj_1^2)$
2. *add* ist primitiv rekursiv wegen $add = Prim\,(proj_1^1;\, succ \circ proj_3^3)$

3. Für die Subtraktion $subtr : \mathbb{N}^2 \to \mathbb{N},\ (n,m) \mapsto n \dot{-} m$ gilt

$$n \dot{-} 0 \quad = \quad n \quad = \quad proj_1^1(n)$$

und

$$n \dot{-} (m+1) \quad = \quad (n \dot{-} m) \dot{-} 1 \quad = \quad pred\,(proj_3^3(n, m, n \dot{-} m))$$

also ist $subtr = Prim\,(proj_1^1;\ pred \circ proj_3^3) \in \mathcal{PR}$ wegen $pred \in \mathcal{PR}$.

4. Für die Multiplikation gilt

$$n * 0 \quad = \quad 0 \quad = \quad const_0^1(n)$$

und

$$\begin{aligned} n * (m+1) &= (n * m) + n \\ &= add\,(proj_3^3(n, m, n * m), proj_1^3(n, m, n * m)) \end{aligned}$$

also ist $mul = Prim\,(const_0^1;\ Sub(add;\ proj_3^3,\ proj_1^3) \in \mathcal{PR}$ wegen $add \in \mathcal{PR}$.

5. Wegen $add, mul \in \mathcal{PR}$ folgt $square, sos \in \mathcal{PR}$.

Satz 3.31 *Jede primitiv rekursive Funktion ist total.*

Beweis: Es genügt zu zeigen:

1. Die Grundfunktionen sind total.
2. Wenn $g, h_1, \ldots, h_l$ total sind, dann ist auch $\mathit{Sub}\,(g;\, h_1, \ldots, h_l)$ total.
3. Wenn g und h total sind, dann ist auch $\mathit{Prim}\,(g, h)$ total.

1. und 2. sind klar.

3. Sei $f = \mathit{Prim}\,(g, h)$ mit totalen Funktionen g und h.

   Dann folgt durch Induktion über m, dass $f(n_1, \ldots, n_k, m)$ für alle $n_1, \ldots, n_k$ definiert ist. □

Abschlusseigenschaften von $\mathcal{PR}$:

Definition 3.32 *Seien $f, g, h_1, h_2 : \mathbb{N}^k \hookrightarrow \mathbb{N}$. f entsteht durch Fallunterscheidung aus g, h_1 und h_2, wenn für alle $\bar{n} \in \mathbb{N}^k$ gilt*

$$f(\bar{n}) = \begin{cases} h_1(\bar{n}) & \text{falls } g(\bar{n}) > 0 \\ h_2(\bar{n}) & \text{falls } g(\bar{n}) = 0 \end{cases}$$

Die Funktion f bezeichnen wir dann mit $If(g; h_1, h_2)$.

Beispiel:

Sei $dist : \mathbb{N}^2 \to \mathbb{N},\ (m, n) \mapsto |m - n|$. Dann gilt

$$dist(n_1, n_2) = \begin{cases} n_1 \dot{-} n_2 & \text{falls } n_1 \dot{-} n_2 > 0 \\ n_2 \dot{-} n_1 & \text{falls } n_1 \dot{-} n_2 = 0 \end{cases}$$

Also ist $dist = If(subtr; subtr, Sub(subtr; proj^2_2, proj^2_1))$.

Satz 3.33 *$\mathcal{PR}$ ist abgeschlossen unter Fallunterscheidung, d.h. wenn $g, h_1, h_2 \in \mathcal{PR}$, dann ist auch $If(g, h_1, h_2) \in \mathcal{PR}$.*

Beweis: Seien $g, h_1, h_2 : \mathbb{N}^k \hookrightarrow \mathbb{N}$ primitiv rekursiv. Man definiert zunächst $f : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ durch

$$\begin{aligned} f(\bar{n}, 0) &= h_2(\bar{n}) \\ f(\bar{n}, m+1) &= h_1(\bar{n}) \\ &= Sub\,(h_1; proj_1^{k+2}, \ldots, proj_k^{k+2})\,(\bar{n}, m, f(\bar{n}, m)) \end{aligned}$$

Dann ist $f = Prim\,(h_2, Sub\,(h_1; \ldots)) \in \mathcal{PR}$, und es gilt

$$\begin{aligned} If\,(g, h_1, h_2)\,(\bar{n}) &= f(\bar{n}, g\,(\bar{n})) \\ &= Sub\,(f; proj_1^k, \ldots, proj_k^k, g)\,(\bar{n}) \end{aligned}$$

Also ist auch $If\,(g, h_1, h_2) = Sub\,(f; proj_1^k, \ldots, proj_k^k, g) \in \mathcal{PR}$. □

Beispiel: Die Funktion *dist* aus dem letzten Beispiel ist primitiv rekursiv, da sie durch Fallunterscheidung und Substitution aus *subtr* und einigen Grundfunktionen entsteht.

Definition 3.34 *Seien* $g : \mathbb{N}^k \hookrightarrow \mathbb{N}^k$ *und* $f : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}^k$. f *entsteht durch* Iteration *aus* g, *wenn für alle* $\bar{n} \in \mathbb{N}^k$ *und* $m \in \mathbb{N}$ *gilt*

$$f(\bar{n}, m) = g^m(\bar{n})$$

Die Funktion f *bezeichnen wir dann mit* $Iter(g)$.

Beispiele:

1. $add = Iter\,(succ)$, da $add(n, m) = succ^m(n)$ für alle $m, n \in \mathbb{N}$.
2. $subtr = Iter\,(pred)$.
3. Sei $square : \mathbb{N} \to \mathbb{N},\ n \mapsto n^2$.

   Dann ist $Iter\,(square) : \mathbb{N}^2 \to \mathbb{N}$ definiert durch

$$Iter\,(square)\,(n, m) = square^m(n) = n^{(2^m)}$$

4. Die Fibonaccifunktion $fib : \mathbb{N} \to \mathbb{N}$ ist definiert durch

$$\begin{aligned} fib\,(0) &= 0 \\ fib\,(1) &= 1 \\ fib\,(n+2) &= fib\,(n) + fib\,(n+1) \end{aligned}$$

Sei $next : \mathbb{N}^2 \to \mathbb{N}^2,\ (n_1, n_2) \mapsto (n_2, n_1 + n_2)$. Dann gilt für alle $n \in \mathbb{N}$

$$(fib\,(n+1), fib\,(n+2)) = next\,(fib(n), fib(n+1))$$

also

$$\begin{aligned} (fib\,(n), fib\,(n+1)) &= next^n(0,1) \\ &= Iter\,(next)\,(0,1,n) \\ &= Iter\,(next)\,(const_0^1(n), const_1^1(n), proj_1^1(n)) \end{aligned}$$

und damit

$$fib = proj_1^2 \circ Sub\,(Iter\,(next); const_0^1, const_1^1, proj_1^1)$$

Satz 3.35 *$\mathcal{PR}$ ist abgeschlossen unter Iteration einstelliger Funktionen, d.h. wenn $g : \mathbb{N} \hookrightarrow \mathbb{N}$ primitiv rekursiv ist, dann ist auch $Iter(g) : \mathbb{N}^2 \hookrightarrow \mathbb{N}$ primitiv rekursiv.*

Beweis: Für alle $n \in \mathbb{N}$ gilt

$$\begin{aligned} Iter\,(g)\,(n,0) &= g^0(n) \\ &= n \\ &= proj_1^1(n) \end{aligned}$$

und für alle $n, m \in \mathbb{N}$

$$\begin{aligned} Iter\,(g)\,(n,m+1) &= g^{m+1}(n) \\ &= g\,(g^m(n)) \\ &= g\,(Iter\,(g)\,(n,m)) \\ &= (g \circ proj_3^3)(n, m, Iter\,(g)\,(n,m)) \end{aligned}$$

Also ist $Iter\,(g) = Prim\,(proj_1^1, g \circ proj_3^3) \in \mathcal{PR}$. □

Beispiel: Satz 3.35 liefert die primitive Rekursivität der Funktion $f : \mathbb{N}^2 \to \mathbb{N},\ (n, m) \mapsto n^{(2^m)}$, da $f = \mathit{Iter}\,(\mathit{square})$.

Lässt sich Satz 3.35 auf Funktionen $g : \mathbb{N}^k \hookrightarrow \mathbb{N}^k$ verallgemeinern?

Dazu müssen wir als erstes unsere Definition erweitern:

Wir nennen eine Funktion $g : \mathbb{N}^k \hookrightarrow \mathbb{N}^k$ *primitiv rekursiv*, wenn die Funktionen $proj_i^k \circ g : \mathbb{N}^k \hookrightarrow \mathbb{N}$ für $i = 1, \ldots, k$ (die die einzelnen Komponenten des Ergebnisses von g liefern) primitiv rekursiv sind.

Satz 3.36 *$\mathcal{PR}$ ist abgeschlossen unter beliebiger Iteration, d.h. wenn $g : \mathbb{N}^k \hookrightarrow \mathbb{N}^k$ primitiv rekursiv ist, dann ist auch $\mathit{Iter}(g) : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}^k$ primitiv rekursiv.*

Beweisskizze:

Der Beweis wäre der gleiche wie im Fall $k = 1$, wenn im Schema der primitiven Rekursion Funktionen zugelassen wären, die Tupel zurückliefern.
Da dies nicht der Fall ist, muss man den allgemeinen Fall auf den Fall $k = 1$ zurückführen, indem man die Listencodierung für Tupel benutzt.
Insbesondere muss man dann nachweisen, dass die Funktionen zur Codierung und Decodierung von Listen primitiv rekursiv sind. □

Beispiel: Satz 3.35 liefert die primitive Rekursivität der Fibonaccifunktion, denn: Zunächst ist die Hilfsfunktion *next* primitiv rekursiv, weil sie durch Substitution aus *add* und den Grundfunktionen entsteht. Daraus folgt die primitive Rekursivität der Funktion $fib = proj_1^2 \circ Sub\,(Iter\,(next); const_0^1, const_1^1, proj_1^1)$.

Wir haben gesehen, dass alle primitiv rekursiven Funktionen *total* sind. Es fehlt also noch ein Operator, der uns erlaubt, *partielle* Funktionen zu definieren.

Definition 3.37 *Sei $k \geq 1$. Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ entsteht durch Anwendung des (unbeschränkten) μ-Operators aus $g : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$, wenn für alle $n_1, \ldots, n_k \in \mathbb{N}$ gilt*

$$\begin{aligned} f(n_1, \ldots, n_k) &= \mu z.\,(g(n_1, \ldots, n_k, z) = 0) \\ &= \min\{z \in \mathbb{N} \mid (n_1, \ldots, n_k, i) \in \mathit{Def}(g) \text{ für alle } i < z \\ &\qquad \text{und } g(n_1, \ldots, n_k, z) = 0\} \end{aligned}$$

wobei das Minimum der leeren Menge undefiniert ist.

Die Funktion f bezeichnet man dann mit $\mu(g)$.

$\mu(g)\,(n_1, ..., n_k) = \mu z.\,(g\,(n_1, \ldots, n_k, z) = 0)$ ist also die *kleinste* Zahl z mit $g\,(n_1, \ldots, n_k, z) = 0$, wobei die Einschränkung zu beachten ist, dass $g\,(n_1, \ldots, n_k, i)$ für alle $i < z$ definiert sein muss. Wenn eine solche Zahl z nicht existiert, so ist $\mu(g)\,(n_1, ..., n_k)$ *undefiniert*.

Beispiele:

1. Für jedes $k \geq 1$ ist $\mu(const_1^{k+1}) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ die *leere* Funktion, denn es gilt

$$\begin{aligned} \mu(const_1^{k+1})\,(n_1, \ldots, n_k) &= \mu z.\,(const_1^{k+1}(n_1, \ldots, n_k, z) = 0) \\ &= \mu z.\,(1 = 0) \end{aligned}$$

   ist undefiniert.

   Allgemeiner gilt: Wenn $g : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ eine Funktion ist, die nie den Wert 0 annimmt, so ist $\mu(g) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ die leere Funktion.

2. Sei *p_div* : $\mathbb{N}^2 \hookrightarrow \mathbb{N}$, $(n_1, n_2) \mapsto n_1 \div n_2$ die partielle ganzzahlige Division, definiert durch

$$n_1 \div n_2 = \begin{cases} \text{undefiniert} & \text{falls } n_2 = 0 \\ \max\{z \in \mathbb{N} \mid z * n_2 \leq n_1\} & \text{sonst} \end{cases}$$

Wegen $\max\{z \in \mathbb{N} \mid z * n_2 \leq n_1\}$

$$\begin{aligned} &= \min\{z \in \mathbb{N} \mid (z+1) * n_2 > n_1\} \\ &= \min\{z \in \mathbb{N} \mid (z+1) * n_2 \geq n_1 + 1\} \\ &= \min\{z \in \mathbb{N} \mid n_1 + 1 \dot{-} (z+1) * n_2 = 0\} \end{aligned}$$

gilt $n_1 \div n_2 = \mu z.\, n_1 + 1 \dot{-} (z+1) * n_2 = 0$ (auch im Falle $n_2 = 0$).

Also ist *p_div* $= \mu(g)$, wobei

$$g : \mathbb{N}^3 \hookrightarrow \mathbb{N},\ (n_1, n_2, z) \mapsto n_1 + 1 \dot{-} (z+1) * n_2$$

Definition 3.38 *Die Menge $\mu - \mathcal{REC}$ der μ-rekursiven Funktionen ist die kleinste Menge von Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ (für alle $k \in \mathbb{N}$), die*

- *alle Grundfunktionen enthält und*
- *abgeschlossen ist unter Substitution, primitiver Rekursion und μ-Rekursion.*

Mit anderen Worten:

Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist genau dann μ-rekursiv, wenn sie durch (wiederholte) Anwendung der Operatoren *Sub*, *Prim* und μ aus den Grundfunktionen $const_c^k$, $proj_i^k$ und *succ* entsteht.

Beispiele:

1. Die leere Funktion $\mu(const^{k+1}) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist μ-rekursiv.
2. *p_div* ist μ-rekursiv, weil $p\_div = \mu(g)$ für die primitiv rekursive Funktion $g : \mathbb{N}^3 \hookrightarrow \mathbb{N},\ (n_1, n_2, z) \mapsto n_1 + 1 \dot{-} (z + 1) * n_2$.

Der unbeschränkte μ-Operator führt zu partiellen Funktionen, weil die 'Suche' nach dem Element $z \in \mathbb{N}$ mit $g(n_1, \ldots, n_k, z) = 0$ unbeschränkt ist. Mit einer 'beschränkten Suche' bleibt man im Bereich der totalen Funktionen.

Definition 3.39 *Sei $k \in \mathbb{N}$. Eine Funktion $f : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ entsteht durch Anwendung des beschränkten μ-Operators aus $g : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$, wenn für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt*

$$\begin{aligned} f(n_1, \ldots, n_k, m) &= \mu z \leq m.\,(g\,(n_1, \ldots, n_k, z) = 0) \\ &= \min\{z \leq m+1 \mid \\ &\qquad (n_1, \ldots, n_k, i) \in \mathit{Def}(g) \text{ für alle } i < z \\ &\qquad \text{und } (g\,(n_1, \ldots, n_k, z) = 0 \text{ oder } z = m+1)\} \end{aligned}$$

Die Funktion f bezeichnet man dann mit $\mu^{\leq}(g)$.

In Worten:

$\mu^{\leq}(g)(\bar{n}, m) = \mu z \leq m.(g(\bar{n}, z) = 0)$ ist die *kleinste* Zahl $z \leq m$ mit $g(\bar{n}, z) = 0$, wieder mit der Einschränkung, dass $g(\bar{n}, i)$ für alle $i < z$ definiert sein muss.

Existiert eine solche Zahl z nicht, so unterscheidet man zwei Fälle:

1. Wenn $g(\bar{n}, i)$ für $i = 0, \ldots, m$ definiert ist (d.h. wenn die 'Suche' nach der Zahl z terminiert und die Antwort 'nein' liefert), so setzt man $\mu^{\leq}(g)(\bar{n}, m) = m + 1$.
2. Andernfalls (wenn man bei der 'Suche' schon in eine Endlosschleife gerät) ist $\mu^{\leq}(g)(\bar{n}, m)$ undefiniert.

Die Festlegung auf das Resultat $m+1$ im 1. Fall ist etwas willkürlich, aber doch sinnvoll: Man kann es so verstehen, dass man bei der Suche nach der Zahl $z \leq m$ über die obere Schranke m hinaus gelaufen ist. Jedenfalls erkennt man am Resultat $m+1$, dass man eine Zahl z mit der gewünschten Eigenschaft nicht gefunden hat.

Für totale Funktionen g lässt sich die Definition von $\mu^{\leq}(g)$ vereinfachen zu:

$$\mu^{\leq}(g)(n_1,\ldots,n_k,m) \;=\; \min\left(\{m+1\} \cup \{z \leq m \mid g(n_1,\ldots,n_k,z) = 0\}\right)$$

Die Menge, deren Minimum man hier bildet, ist niemals leer, weil sie mindestens die Zahl $m+1$ enthält. Also ist das Minimum stets wohldefiniert, d.h. $\mu^{\leq}(g)$ ist total, wenn g total ist.

Damit ist es klar, dass der beschränkte μ-Operator schwächer als der unbeschränkte ist, denn der unbeschränkte erlaubt uns ja, die Menge der totalen Funktionen zu verlassen.

Es stellt sich nun die Frage, ob man mit dem beschränkten μ-Operator überhaupt über die Menge $\mathcal{PR}$ hinauskommt, oder ob er—wie die Operatoren *If* und *Iter*—nur die Definition primitiv rekursiver Funktionen erleichtert. Die Antwort liefert der folgende Satz.

Satz 3.40 *$\mathcal{PR}$ ist abgeschlossen unter dem beschränkten μ-Operator.*

Beweis:

Sei $g : \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv. Dann kann man $\mu^{\leq}(g)(\bar{n}, m)$ durch folgende Induktion über m definieren:

$$\mu^{\leq}(g)(\bar{n}, 0) = \begin{cases} 0 & \text{falls } g(\bar{n}, 0) = 0 \\ 1 & \text{sonst} \end{cases}$$

$$\mu^{\leq}(g)(\bar{n}, m+1) = \begin{cases} \mu^{\leq}(g)(\bar{n}, m) & \text{falls } \mu^{\leq}(g)(\bar{n}, m) \leq m \\ m+1 & \text{falls } \mu^{\leq}(g)(\bar{n}, m) > m \\ & \quad \text{und } g(\bar{n}, m+1) = 0 \\ m+2 & \text{sonst} \end{cases}$$

An der induktiven Definition sieht man im Prinzip schon, dass sich $\mu^{\leq}(g)$ durch primitive Rekursion und Fallunterscheidung auf g und andere primitiv rekursive Funktionen zurückführen lässt.

Für einen exakten Beweis führen wir weitere Umformungen durch.

Für den Induktionsanfang gilt

$$\mu^{\leq}(g)\,(\bar{n}, 0)$$

$$= \begin{cases} \mathit{const}_0^k(\bar{n}) & \text{falls } g\,(\mathit{proj}_1^k(\bar{n}), \ldots, \mathit{proj}_k^k(\bar{n}), \mathit{const}_0^k(\bar{n})) = 0 \\ \mathit{const}_1^k(\bar{n}) & \text{sonst} \end{cases}$$

$$= g'(\bar{n})$$

wobei $g' = \mathit{If}\,(\mathit{Sub}\,(g;\, \mathit{proj}_1^k, \ldots, \mathit{proj}_k^k, \mathit{const}_0^k);\, \mathit{const}_1^k, \mathit{const}_0^k)$ primitiv rekursiv ist, weil g primitiv rekursiv ist.

Für den Induktionsschritt definieren wir zunächst die Hilfsfunktion

$$f : \mathbb{N}^{k+1} \to \mathbb{N}$$

$$(\bar{n}, m) \mapsto \begin{cases} m & \text{falls } g(\bar{n}, m) = 0 \\ m+1 & \text{sonst} \end{cases}$$

Es gilt $f = \mathit{If}\ (g;\ \mathit{succ} \circ \mathit{proj}^{k+1}_{k+1}, \mathit{proj}^{k+1}_{k+1})$, also ist f primitiv rekursiv, weil g primitiv rekursiv ist.

Mit Hilfe von f können wir den Induktionsschritt umformen zu

$$\mu^{\leq}(g)(\bar{n}, m+1) = \begin{cases} \mu^{\leq}(g)(\bar{n}, m) & \text{falls } \mu^{\leq}(g)(\bar{n}, m) \dot{-} m = 0 \\ f(\bar{n}, m+1) & \text{sonst} \end{cases}$$

$$= h(\bar{n}, m, \mu^{\leq}(g)(\bar{n}, m))$$

wobei

$$h = \mathit{If}(\mathit{Sub}(\mathit{subtr}; \mathit{proj}^{k+2}_{k+2}, \mathit{proj}^{k+2}_{k+1}); \mathit{Sub}(f; \mathit{proj}^{k+2}_1, \ldots, \mathit{proj}^{k+2}_k, \mathit{succ} \circ \mathit{proj}^{k+2}_{k+1}), \mathit{proj}^{k+2}_{k+2}).$$

h ist primitiv rekursiv, weil *subtr* und f primitiv rekursiv sind. Also ist schließlich auch $\mu^{\leq}(g)$ primititiv rekursiv, weil $\mu^{\leq}(g) = \mathit{Prim}(g', h)$. □

Wir beweisen jetzt die folgenden Äquivalenzen zwischen den unterschiedlichen Berechenbarkeits-Begriffen:

$$\mu\text{-rekursiv} \;=\; \textit{while}\text{-berechenbar}$$

$$\text{primitiv rekursiv} \;=\; \textit{loop}\text{-berechenbar}$$

Die Beweise dieser Äquivalenzen sind *konstruktiv*, d.h.:

- Wir können unsere Schreibweisen für primitiv rekursive bzw. μ-rekursive Funktionen als *Programmiersprachen* auffassen.
- Dann liefern die Beweise *Übersetzungs-Algorithmen* ('Compiler') zwischen μ-rekursiven Programmen und *while*-Programmen bzw. zwischen primitiv rekursiven Programmen und *loop*-Programmen.

Satz 3.41 *Zu jedem primitiv rekursiven Programm lässt sich ein äquivalentes loop-Programm konstruieren, also ist jede primitiv rekursive Funktion loop-berechenbar.*

Beweis:

Es genügt zu zeigen

1. wie sich die Grundfunktionen als *loop*-Programme implementieren lassen,
2. wie sich aus *loop*-Programmen für $g, h_1, \ldots, h_l$ ein *loop*-Programm für $Sub\,(g; h_1, \ldots, h_l)$ konstruieren lässt,
3. wie sich aus *loop*-Programmen für g und h ein *loop*-Programm für $Prim\,(g, h)$ konstruieren lässt.

1. s. Übungsblatt 14, Aufgabe 2.

2. Seien $P, P_1, \ldots, P_l$ *loop*-Programme (oder *while*-Programme), die die Funktionen $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ und $h_1, \ldots, h_l : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen.

   Wir dürfen annehmen, dass die Programme P und P_i $(i = 1, \ldots, l)$ 'geeignete' Form haben, etwa:

   $$P_i = \text{read } X_1, \ldots, X_k;\ \mathit{stl}_i \text{ write } X_{k+i}$$

   (d.h. alle P_i arbeiten mit den gleichen Eingabe-Speicherplätzen und jedes hat seinen eigenen Ausgabe-Speicherplatz)

   $$P = \text{read } X_{k+1}, \ldots, X_{k+l};\ \mathit{stl} \text{ write } X_0$$

   (d.h. P benutzt die Ausgabe-Speicherplätze der P_i zur Eingabe)

   und dass sie alle 'sauber' programmiert sind, d.h. dass die Inhalte von $X_1, \ldots, X_k$ nicht verändert werden und dass alle anderen Speicherplätze explizit vorbesetzt werden.

Dann lässt sich die Funktion $\mathit{Sub}\,(g;\, h_1, \ldots, h_l) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen durch

$$\text{read } X_1, \ldots, X_k;\ \mathit{stl}_1\ \ldots\ \mathit{stl}_l\ \mathit{stl}\ \text{write } X_0$$

3. Seien P_0, P_1 Programme, die die Funktionen $g : \mathbb{N}^k \hookrightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+2} \hookrightarrow \mathbb{N}$ berechnen. Wir dürfen annehmen, dass

$$\begin{aligned} P_0 &= \text{read } X_1, \ldots, X_k;\ \mathit{stl}_0\ \text{write } X_{k+2} \\ P_1 &= \text{read } X_1, \ldots, X_{k+2};\ \mathit{stl}_1\ \text{write } X_{k+2} \end{aligned}$$

und dass beide 'sauber' programmiert sind. Dann lässt sich die Funktion $\mathit{Prim}\,(g, h) : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ berechnen durch

$$\begin{array}{l} \text{read } X_1, \ldots, X_k, X_0; \\ X_{k+1} := 0;\ \mathit{stl}_0 \\ \text{loop } X_0 \text{ do } \mathit{stl}_1\ X_{k+1} := \mathit{succ}\,(X_{k+1}) \text{ od} \\ \text{write } X_{k+2} \end{array}$$

Erläuterung:

Das angegebene Programm berechnet nacheinander die Werte

$$\mathit{Prim}\,(g,h)\,(\bar{n},0),\ldots,\mathit{Prim}\,(g,h)\,(\bar{n},m)$$

wobei X_{k+2} als Speicherplatz für das Zwischenergebnis benutzt wird. X_{k+1} dient als Zähler, der die Werte $0,\ldots,m$ durchläuft (wobei der letzte Wert m nicht mehr benötigt wird).

Durch stl_0 wird X_{k+2} zunächst mit $g(\bar{n}) = \mathit{Prim}\,(g,h)\,(\bar{n},0)$ vorbesetzt. Dann wird der Inhalt von X_{k+2} in jedem Schleifendurchlauf durch stl_1 verändert: Wenn zu Beginn eines Schleifendurchlaufs der Wert $\mathit{Prim}\,(g,h)\,(\bar{n},i)$ in X_{k+2} steht, dann bewirkt stl_1, dass am Ende des Schleifendurchlaufs der Wert $h\,(\bar{n},m,\mathit{Prim}\,(g,h)\,(\bar{n},i)) = \mathit{Prim}\,(g,h)\,(\bar{n},i+1)$ dort steht.

Da in X_0 das $(k+1)$-te Argument m eingelesen wird, finden m Schleifendurchläufe statt, also enthält X_{k+2} am Ende des Programms das gewünschte Ergebnis $\mathit{Prim}\,(g,h)\,(\bar{n},m)$. □

Satz 3.42 *Zu jedem μ-rekursiven Programm lässt sich ein äquivalentes while-Programm konstruieren, also ist jede μ-rekursive Funktion while-berechenbar.*

Beweis: Es bleibt nur noch zu zeigen, wie sich Anwendungen des μ-Operators in die *while*-Programmiersprache übersetzen lassen. Sei also

$$P = \text{read } X_1, \ldots, X_k, X_{k+1};\ \mathit{stl}\ \text{write } X_0$$

ein 'sauberes' *while*-Programm, das die Funktion $g : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ berechnet. Dann lässt sich $\mu(g) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen durch

read $X_1, \ldots, X_k$;

$X_{k+1} := 0;\ X_0 := 1;$

while X_0 do *stl* $X_{k+1} := \mathit{succ}(X_{k+1})$ od;

$X_{k+1} := \mathit{pred}(X_{k+1});$

write X_{k+1}

Erläuterung:

Das angegebene Programm berechnet nacheinander die Werte

$$g(\bar{n}, 0), g(\bar{n}, 1) \ldots .$$

bis es die erste Zahl m mit $g(\bar{n}, m) = 0$ findet.

X_{k+1} dient als Zähler für die Anzahl der Schleifendurchläufe und wird deshalb mit 0 vorbesetzt. Im i-ten Schleifendurchlauf wird durch Ausführung von *stl* der Wert $g\,(\bar{n}, i)$ berechnet und in X_0 gespeichert. Anschließend wird X_k auf $i + 1$ gesetzt.

Also terminiert die Schleife, sobald die erste Zahl m mit $g(\bar{n}, m) = 0$ gefunden ist, und dann enthält X_{k+1} den Wert $m + 1$. Deshalb muss X_{k+1} noch um 1 heruntergesetzt werden.

Man beachte noch, dass das Programm *nicht* terminiert, wenn *kein* Wert m mit $g(\bar{n}, m) = 0$ und $(\bar{n}, i) \in \mathit{Def}\,(f)$ für alle $i < m$ existiert. Auch das entspricht der Definition des μ-Operators. □

Satz 3.43 *Zu jedem loop-Programm lässt sich ein äquivalentes primitiv rekursives Programm konstruieren, also ist jede loop-berechenbare Funktion primitiv rekursiv.*

Beweis: Sei $m \in \mathbb{N}$. Für jede Anweisungsliste *stl*, die höchstens die Speicherplätze $X_0, \ldots, X_m$ enthält, definieren wir die *Komponentenfunktionen*

$$[\![stl]\!]_i : \mathbb{N}^{m+1} \hookrightarrow \mathbb{N}$$

$$(n_0, \ldots n_m) \mapsto \begin{cases} \sigma(X_i) & \text{falls } (stl, \sigma_0[n_0/X_0] \ldots [n_m/X_m]) \vdash^* (\varepsilon, \sigma) \\ \text{undefiniert} & \text{sonst} \end{cases}$$

für $i = 0, \ldots m$, und die *Gesamtfunktion*

$$[\![stl]\!] : \mathbb{N}^{m+1} \hookrightarrow \mathbb{N}^{m+1}$$

$$(n_0, \ldots, n_m) \mapsto ([\![stl]\!]_0(n_0, \ldots, n_m), \ldots, [\![stl]\!]_m(n_0, \ldots, n_m))$$

Die i-te Komponentenfunktion gibt also an, wie sich der Endinhalt des Speicherplatzes X_i aus den Anfangsinhalten der Speicherplätze $X_0, \ldots, X_m$ ergibt. Die Gesamtfunktion fasst all diese Informationen zusammen.

Wenn wir zeigen können, dass die Funktionen $[\![\mathit{stl}]\!]_i$ stets primitiv rekursiv sind (was gleichbedeutend ist zur primitiven Rekursivität von $[\![\mathit{stl}]\!]$), so ist auch die von einem Programm

$$P = \text{read } X_{i_1}, \ldots, X_{i_k};\ \mathit{stl}\ \text{write } X_i$$

berechnete Funktion $[\![P]\!] : \mathbb{N}^k \hookrightarrow \mathbb{N}$ primitiv rekursiv, denn es gilt

$$[\![P]\!] = \mathit{Sub}\,([\![\mathit{stl}]\!]_i;\ h_0, \ldots, h_m) \quad \text{mit} \quad h_l = \begin{cases} \mathit{proj}_j^k & \text{falls } l = i_j \\ \mathit{const}_0^k & \text{falls } l \notin \{i_1, \ldots, i_k\} \end{cases}$$

Die primitive Rekursivität der Funktionen $[\![\mathit{stl}]\!]_i$ (bzw. $[\![\mathit{stl}]\!]$) beweist man durch Induktion über die Größe von *stl*.

- Den Induktionsanfang bilden die Zuweisungen, z.B. gilt

$$[\![X_j := \mathit{pred}(X_k);]\!]_i = \left\{ \begin{array}{ll} \mathit{pred} \circ \mathit{proj}_k^{m+1} & \text{falls } i = j \\ \mathit{proj}_i^{m+1} & \text{sonst} \end{array} \right\} \in \mathcal{PR}$$

  weil $\mathit{pred} \in \mathcal{PR}$. Analog für die übrigen Zuweisungen.

- Im Induktionsschritt sind die übrigen Anweisungslisten zu betrachten. Es gilt

$$\begin{array}{rcl} [\![\mathit{stl}\ \mathit{stl}']\!] & = & [\![\mathit{stl}']\!] \circ [\![\mathit{stl}]\!] \\ [\![\text{if } X_j \text{ then } \mathit{stl} \text{ else } \mathit{stl}' \text{ fi}]\!] & = & \mathit{If}\,(\mathit{proj}_j^{m+1}; [\![\mathit{stl}]\!], [\![\mathit{stl}']\!]) \\ [\![\text{loop } X_j \text{ do } \mathit{stl} \text{ od}]\!] & = & \mathit{Iter}\,(\mathit{proj}_j^{m+1}; [\![\mathit{stl}]\!]) \end{array}$$

  Diese Funktionen sind primitiv rekursiv, weil nach Induktionsannahme die Funktionen $[\![\mathit{stl}]\!]$ und $[\![\mathit{stl}']\!]$ primitiv rekursiv sind. □

Satz 3.44 *Zu jedem while-Programm lässt sich ein äquivalentes μ-rekursives Programm konstruieren, also ist jede while-berechenbare Funktion μ-rekursiv.*

Beweis: Es ist nur noch zu zeigen, dass sich jede *while*-Anweisung

$$\text{while } X_i \text{ do } stl \text{ od}$$

in die Schreibweise für μ-rekursive Funktionen übersetzen lässt.

Die Idee besteht darin, mit Hilfe des μ-Operators die kleinste Anzahl k von Schleifendurchläufen zu bestimmen, bei der ein Zustand σ mit $\sigma(X_i) = 0$ entsteht, und dann die Schleife k-mal auszuführen.

Genauer: Für jedes Tupel $(n_0, \ldots, n_m) \in \mathbb{N}^{m+1}$ betrachten wir die Zustände σ_j mit

$$(\underbrace{stl \ldots stl}_{j}, \sigma_0[n_0/X_0] \ldots [n_m/X_m]) \vdash^* (\varepsilon, \sigma_j)$$

und suchen die Zahl $k = \min\{j \in \mathbb{N} \mid \sigma_j(X_i) = 0\}$.

Diese Zahl k erhalten wir als Ergebnis einer μ-rekursiven Funktion:

$$\begin{aligned} k &= \min\{j \in \mathbb{N} \mid \mathit{proj}_i^{m+1}(\llbracket \mathit{stl} \rrbracket^j(n_0,\ldots,n_m)) = 0\} \\ &= \min\{j \in \mathbb{N} \mid \mathit{proj}_i^{m+1}(\mathit{Iter}\,\llbracket \mathit{stl} \rrbracket\,(n_0,\ldots,n_m,j)) = 0\} \\ &= \underbrace{\mu\,(\mathit{proj}_i^{m+1} \circ \mathit{Iter}\,\llbracket \mathit{stl} \rrbracket)}_{f}\,(n_0,\ldots,n_m) \end{aligned}$$

Also ergeben sich auch die Inhalte der Speicherplätze im Zustand σ_k durch eine μ-rekursive Funktion:

$$\begin{aligned} &(\sigma_k(X_0),\ldots,\sigma_k(X_m)) \\ &\quad= \llbracket \mathit{stl} \rrbracket^k(n_0,\ldots,n_m) \\ &\quad= \mathit{Iter}\,\llbracket \mathit{stl} \rrbracket\,(n_0,\ldots n_m,k) \\ &\quad= \mathit{Iter}\,\llbracket \mathit{stl} \rrbracket\,(n_0,\ldots n_m,f(n_0,\ldots,n_m)) \\ &\quad= \mathit{Sub}\,(\mathit{Iter}\,\llbracket \mathit{stl} \rrbracket;\, \mathit{proj}_1^{m+1},\ldots,\mathit{proj}_{m+1}^{m+1},f)\,(n_0,\ldots n_m) \end{aligned}$$

Andererseits ist σ_k natürlich der Endzustand der *while*-Anweisung:

$$(\text{while } X_i \text{ do } \mathit{stl} \text{ od}, \sigma_0[n_0/X_0] \ldots [n_m/X_m]) \vdash^* (\varepsilon, \sigma_k)$$

d.h. die obige μ-rekursive Funktion ist die gesuchte "Übersetzung":

$$[\![\text{while } X_i \text{ do } \mathit{stl} \text{ od}]\!] \;=\; \mathit{Sub}\,(\mathit{Iter}\,[\![\mathit{stl}]\!];\, proj_1^{m+1}, \ldots, proj_{m+1}^{m+1}, f)$$

Man beachte noch, dass diese Überlegungen auch dann korrekt sind, wenn die *while*-Anweisung *nicht* terminiert. Dann existiert nämlich der oben genannte Zustand σ_k nicht, also liefert auch die Anwendung der Funktion $f = \mu\,(\mathit{proj}_i^{m+1} \circ \mathit{Iter}\,[\![stl]\!])$ auf die Anfangswerte $(n_0, \ldots, n_m)$ kein Ergebnis. □

Damit sind alle gewünschten Äquivalenzen zwischen unseren Programmiersprachen bewiesen. Man kann sich nun noch fragen, wie mächtig die Sprache der primitiv rekursiven Funktionen (bzw. die dazu äquivalente Sprache der *loop*-Programme) ist. Eine gewisse Antwort gibt der folgende Satz.

Satz 3.45 *In der Programmiersprache der primitiv rekursiven Programme lässt sich kein Interpreter für die Sprache selbst schreiben.*

Beweis: (durch Diagonalisierung)
Wie bei jeder Programmiersprache können wir die Programme "durchnummerieren", d.h. wir haben eine unendliche Folge $\pi_1, \pi_2, \ldots$, die *alle* primitiv rekursiven Programme enthält. Die vom Programm π_i berechnete (totale) Funktion bezeichnen wir mit f_i.

Von einem Interpreter π erwarten wir, dass er bei Eingabe (i, n) das Programm π_i für den Eingabewert n simuliert, also das Ergebnis $f_i(n)$ liefert. Angenommen, π berechnet selbst eine primitiv rekursive Funktion f. Dann ist auch $g : \mathbb{N} \to \mathbb{N}$, $i \mapsto f(i, i) + 1$ primitiv rekursiv, also müsste g mit einer der Funktionen f_i übereinstimmen. Es gilt aber

$$g(i) = f(i, i) + 1 = f_i(i) + 1 \neq f_i(i)$$

für jedes $i \in \mathbb{N}$. Das ist ein Widerspruch. □

Als unmittelbare Folgerung von Satz 3.45 erhalten wir

Korollar 3.46 *Es gibt eine totale berechenbare Funktion, die nicht primitiv rekursiv ist.*

Beweis:
Ein Beispiel ist die Funktion f im Beweis von Satz 3.45.

Sie ist total, da sie für jedes Paar (i, n) das Ergebnis $f_i(n)$ liefert. Sie ist berechenbar, weil sich ein Interpreter für primitiv rekursive Programme z.B. in der *while*-Programmiersprache schreiben lässt. □

Ein anderes Beispiel ist die bereits erwähnte Ackermann-Funktion (vgl. Literatur).