Grundlagen der theoretischen Informatik

Dr. Jörg Vogel

SS 2003

Inhaltsverzeichnis

1 Berechenbarkeit **8**
1.1 Was ist ein Algorithmus? . . . 8

2 Turing-Maschinen **12**
2.1 Wörter und Sprachen . . . 12
2.1.1 Spezielle Alphabete . . . 13
2.2 Der Begriff der Turing-Maschine . . . 14
2.2.1 Beispiele für Turing-Maschinen . . . 16
2.3 Exkurs über Zahlenfunktionen . . . 18
2.3.1 b-näre Zahlendarstellung . . . 18
2.3.2 b-adische Darstellung . . . 19
2.4 Turing-Berechenbarkeit . . . 19
2.4.1 Normierte Startsituation . . . 20
2.4.2 partielle Funktionen . . . 20
2.4.3 Wortfunktionen . . . 21
2.4.4 Entscheidbarkeit, Semientscheidbarkeit und Aufzählbarkeit . . . 23
2.4.5 Turing-Maschinen als Akzeptoren und Erkenner . . . 24
2.5 Techniken zur Programmierung von Turing-Maschinen . . . 25
2.5.1 Endliche Sprache mit schnellem Zugriff . . . 25
2.5.2 Turing-Bänder mit mehreren Spuren – Mehrspurmaschinen . . . 25
2.5.3 Mehrbandmaschninen . . . 27
2.5.4 Unterprogramme . . . 29
2.5.5 Komposition von Turing-Maschinen . . . 30
2.6 Rechenzeit und Speichersplatz . . . 33

3 Partiell-rekursive Funktionen **35**
3.1 Primitiv-rekursive Funktionen . . . 35
3.2 Die Ackermann-Funktion . . . 38
3.3 μ-rekursive Funktionen . . . 40
3.4 Allgemein-rekursive Funktionen . . . 43
3.4.1 Eigenschaften primitiv-rekursiver Funktionen . . . 47
3.5 Partiell-rekursive und Turing-berechenbare Funktionen . . . 48
3.5.1 Eigenschaften partiell-rekursiver Funktionen . . . 53

4 Formale Sprachen und formale Grammatiken **54**
4.1 Grammatiken und Sprachen vom Typ-0 . . . 55

4.1.1 Beziehungen zwischen Typ-0-Grammatiken und Turing-Maschinen 60
4.2 Grammatiken und Sprachen vom Typ-1 . . . 63
4.3 Grammatiken und Sprachen vom Typ-2 . . . 65
4.4 Grammatiken und Sprachen vom Typ-3 . . . 69
4.5 Zusammenfassung . . . 70

5 Die Hierarchie der Sprachklassen 71
5.1 Das Wortproblem für Sprachen vom Typ-i ($i = 0, 1, 2, 3$) . . . 71
5.2 Das Pumping-Lemma für kontextfreie Sprachen . . . 73
5.3 Das Pumping-Lemma für reguläre Sprachen . . . 75
5.4 Kontextfreie Sprachen über einelementigen Alphabeten . . . 78

6 Die Hierarchie der Automaten 79
6.1 Das Phänomen „Nichtdeterminismus“ . . . 79
6.2 Turing-Maschinen vom Typ-0 . . . 81
6.3 Turing-Maschinen vom Typ-1 . . . 83
6.4 Turing-Maschinen vom Typ-2 . . . 83
6.4.1 Beschreibung von Linksableitungen durch Kellerautomaten . . . 84
6.4.2 Akzeptanzverhalten eines Kellerautomaten . . . 85
6.5 Turing-Maschinen vom Typ-3 . . . 87

7 Ausblick in die Theorie der Entscheidbarkeit 88
7.1 Entscheidbarkeit und Aufzählbarkeit . . . 88
7.2 Charakterisierung der aufzählbaren Mengen . . . 88
7.3 Codierung von Turing-Maschinen . . . 90

Auflistung der Theoreme

Sätze

Satz 1.2 Hauptsatz der Algorithmentheorie . . . 11
Satz 1.3 These von Church . . . 11
Satz 3.5 stückweise definierte Funktionen . . . 47
Satz 3.6 obere Schranken . . . 47
Satz 3.7 Simultane Rekursion . . . 47
Satz 3.8 Wertverlaufsfunktionen . . . 48
Satz 3.11 Kleene'scher Normalformsatz . . . 53
Satz 3.12 Parametrisierungssatz . . . 53
Satz 5.1 Hierarchiesatz für Klassen der Chomski-Hierarchie . . . 71
Satz 5.4 Das Pumping-Lemma für CF . . . 73
Satz 5.5 Das Pumping-Lemma für REG . . . 75
Satz 7.2 Anhaltesatz . . . 92

Definitionen und Festlegunden

Definition 2.1 Formale Beschreibung der Turing-Maschine . . . 15
Definition 3.1 Grundfunktionen . . . 36
Definition 3.2 Einsetzungsprinzip . . . 36
Definition 3.3 Schema der primitiven Rekursion . . . 36
Definition 3.4 primitiv-rekursive Funktionen . . . 37
Definition 4.6 Typ-1-Grammatik . . . 63
Definition 4.7 Typ-1-Normalform-Grammatik . . . 63
Definition 4.8 kontextsensitive Grammatik . . . 64

Definition 6.1 nichtdeterministische Turing-Maschinen 81
Definition 6.2 Formale Definition des Kellerautomaten 85

Vorwort

Dieses Dokument wurde als Skript für die auf der Titelseite genannte Vorlesung erstellt und wird jetzt im Rahmen des Projekts „Vorlesungsskripte der Fakultät für Mathematik und Informatik“ weiter betreut. Das Dokument wurde nach bestem Wissen und Gewissen angefertigt. Denoch garantiert weder der auf der Titelseite genannte Dozent, die Personen, die an dem Dokument mitgewirkt haben, noch die Mitglieder des Projekts für dessen Fehlerfreiheit. Für etwaige Fehler und dessen Folgen wird von keiner der genannten Personen eine Haftung übernommen. Es steht jeder Person frei, dieses Dokument zu lesen, zu verändern oder auf anderen Medien verfügbar zu machen, solange ein Verweis auf die Internetadresse des Projekts `http://uni-skripte.lug-jena.de/` *enthalten ist.*

Diese Ausgabe trägt die Versionsnummer 2562 und ist vom 4. Dezember 2009. Eine (mögliche) aktuellere Ausgabe ist auf der Webseite des Projekts verfügbar.

Jeder ist dazu aufgerufen, Verbesserungen, Erweiterungen und Fehlerkorrekturen für das Skript einzureichen bzw. zu melden oder diese selbst einzupflegen – einfach eine E-Mail an die Mailingliste `<uni-skripte@lug-jena.de>` *senden. Weitere Informationen sind unter der oben genannten Internetadresse verfügbar.*

Hiermit möchten wir allen Personen, die an diesem Skript mitgewirkt haben, vielmals danken:

- *Jörg Sommer* `<joerg@alea.gnuu.de>` *(2005)*

1 Berechenbarkeit

1.1 Was ist ein Algorithmus?

1935 formulierte Alan Turing folgende Frage: „Gibt es eine *effektive Prozedur*, die das Hilbertsche Problem löst?“

Hilbert'sche Problem: Gegeben sei eine Menge von prädikaten-logischen Formeln und eine weitere Formel F. Ist F eine Folgerung aus der Menge der gegebenen Formeln?

Umformulierung: Ist eine gegebene Formel erfüllbar oder nicht?

Das Problem von Turing war, dass es bis 1935 keine Präzisierung dessen gab, was ein solches Verfahren („effektive Prozedur“) ist, obwohl es schon seit jahrtausenden Rechenverfahren gab.

Beispiele für solche Verfahren:

- Grundrechenarten für mehrstellige Zahlen
- Lösungsverfahren für Gleichungen
  - lineare Gleichungen
  - Polynome
  - Differentialgleichungen
  - Diophantische Gleichungen
- Konstruktionsverfahren
  - regelmäßiges 17-Eck
- Rezepturen...
  - ...für Speißen
  - ...für Arzeneimittel
  - ...für Parfüme

allen Rezepturen ist gemeinsam:

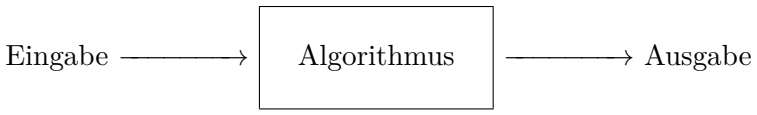
- wohlbestimmte Menge an Ausgangsstoffen

- Folge exakter Handlungsanweisungen
- $\rightarrow$ Produkt

den Rechenverfahren ist gemeinsam:

- wohldefinierte Menge von Ausgangsgrößen
- Folge von Rechenschritten
- $\rightarrow$ Resultat

Dies liefert einen ersten Ansatz für die Formulierung:



Ein konkretes Beispiel für einen Algorithmus ist der Euklidische Algorithmus. Dazu später mehr.

Definition 1.1
Für zwei natürliche Zahlen x und y bezeichnet $\boldsymbol{x}$ **mod** $\boldsymbol{y}$ diejenige natürliche Zahl, die als **Rest** bei der **ganzzahligen Division** mit dem Dividenden x und dem Divisor y bleibt.

Frage: Ist diese Festlegung sinnvoll? Was ist das Resultat?

Satz 1.1
Für je zwei natürliche Zahlen x und y gibt es zwei eindeutig bestimmte Zahlen q und r mit folgenden Eigenschaften:

- $x = qy + r$
- $0 \leq r < y$

Bemerkung 1.1
Satz 1.1 rechtfertigt die folgende Bezeichnung:

1. $q = x \div y$
2. $r = x \bmod y$

Euklidischer Algorithmus – informal

Eingabe: zwei natürliche Zahlen a und b

Ausgabe: r_{k-1}

1. Schritt: Bestimmt $r = a \bmod b$
Falls $r = 0$, dann gib b aus, sonst gehe zum nächsten Schritt.

2. Schritt: Bestimme $r_2 = b \bmod r$
Falls $r_2 = 0$, dann gebe r aus, sonst gehe zum nächsten Schritt.

3. Schritt: Bestimmt $r_3 = r \bmod r_2$
Falls $r_3 = 0$, dann gebe r_2 aus, sonst gehe zum nächsten Schritt.

4. Schritt: Wiederhole das Verfahren solange, bis ein Wert $r_k = 0$ erreicht wird und gib den Wert r_{k-1} aus.

Euklidischer Algorithmus – formal

1. $a_0 = a; a_1 = b; i = 0$
2. `while` $a_{i+1} \neq 0$ `do`
3. $\quad a_{i+2} = a_i$ `mod` a_{i+1}
4. $\quad i = i + 1$
5. `return` a_i

Zwei Fragen ergeben sich aus dieser Definition

1. Ist die Ausgabe a_i bzw. r_{k-1} tatsächlich der größte gemeinsame Teiler der Zahlen a und b? – Ist das gegebene Verfahren *korrekt*?
2. Liefert das Verfahren das Resultat in einer praktikablen Zeit? – Ist das gegebene Verfahren *effizient*?

In unserem Fall lassen sich beide Fragen mit *ja* beantworten.

Eigenschaften von Algorithmen:

- endliche Beschreibung
- endlicher Datenbereich
- beschränkte Menge von Anfangs-Grundoperationen
- schrittweise Ausführung (bzw. endliche Parallelität)
- endliche Dauer jeder Grundoperation
- Determiniertheit – zu jedem Zeitpunkt steht fest, welches der nächste Schritt ist
- Terminierung – das Verfahren endet

mathematische Formalisierung des Algorithmenbegriffs

verschiedene Zugänge:

Termersetzungssysteme

- Turing-Maschine (A. Turing, 1936) – Kapitel 2
- Post'sche Systeme (M. Post, 1943)

- Markovalgorithmen (M. Markov, 1951)

algebraischer Zugang

- partiell-rekursive Funktionen (Kleene, Herbrand, Gödel, 1936) – Kapitel 3
- λ-definierbare Funktionen (Church, 1936)

theoretische Modelle

- Registermaschinen (Sheperson, 1964)

Satz 1.2 (Hauptsatz der Algorithmentheorie)
Alle verschiedene Formalisierungen des Algorithmenbegriffs sind äuqivalent.

Satz 1.3 (These von Church)
Alle Algorithmenbegriffe beschreiben die Klasse der im intuitiven Sinne berechenbaren Funktionen!

2 Turing-Maschinen

2.1 Wörter und Sprachen

Eine endliche Menge Σ, z. B. $\Sigma = \{a_1, a_2, \ldots, a_n\}$, bezeichnet man als **Alphabet**. Ein Element des Alphabets $x \in \Sigma$ nennt man **Buchstabe**. Jede endliche Folge $x_1 \ldots x_m$ von Buchstaben aus Σ heißt **Wort** über dem Alphabet Σ. Die Anzahl m der Buchstaben eines Wortes w heißt **Länge des Wortes** und wird mit $|w|$ bezeichnet.

Eine besondere Rolle spielt das „**leere Wort**“ der Länge null – also kein Buchstabe. Die Bezeichnung dafür ist λ oder ε.

Σ^* bezeichnet die **Menge aller endlichen Wörter** (inkl. λ). Angenommen die Buchstaben $a_1, \ldots, a_n$ von Σ sind geordnet $a_1 < a_2 < \ldots < a_n$. Dann ergibt sich hieraus eine **kanonische Ordnung** für Σ^*.

$$\begin{aligned} w_1 < w_2 \;\; gdw_{def} \;\; & |w_1| < |w_2| \text{ oder} \\ & (w_1 = wx_1w_1', w_2 = wx_2w_2' \wedge x_1 < x_2) \end{aligned}$$

oder in Worten ausgedrück: wenn w_1 und w_2 die gleiche Länge haben, dann ist der erste Buchstabe von links, in dem sich w_1 und w_2 unterscheiden, in w_1 kleine als der in w_2. Die Reihenfolge von Σ^* heißt auch **quasilexikographische Reihenfolge**

Es sei Σ ein Alphabet. Dann bezeichnet

$$\begin{aligned} \Sigma^0 &:= \{\lambda\} \\ \Sigma^1 &:= \Sigma \\ \Sigma^{n+1} &:= \{wx \colon w \in \Sigma^n, x \in \Sigma\} \\ \Sigma^* &= \bigcup_{n=0}^{\infty} \Sigma^n \\ \Sigma^+ &:= \Sigma^* \setminus \{\lambda\} \end{aligned}$$

Für zwei Wörter $u, v \in \Sigma^*$ mit $u = x_1 \ldots x_m$ und $v = y_1 \ldots y_n$ bezeichnet die **Konkatenation** (Hintereinanderschreibung) $u{\cdot}v$ (oder uv) das Wort $x_1 \ldots x_m y_1 \ldots y_n$.

Für ein Wort $w \in \Sigma^*$ mit $w = x_1 \ldots x_m$ bezeichnet man das Wort $w^R = x_m x_{m-1} \ldots x_1$ als **Spiegelwort**.

Man bezeichnet $L \subseteq \Sigma^*$ als **formale Sprache**. Für zwei Sprachen $L_1, L_2 \subseteq \Sigma^*$ ist die **Konkaternation** $L_1 \cdot L_2 := \{u \cdot v \colon u \in L_1, v \in L_2\}$. Für eine Sprache $L \subseteq \Sigma^*$ ist die **Spiegelsprache** L^R definiert durch $\{w^R \colon w \in L\}$.

Für ein Wort $w \in \Sigma^*$ sind die **Potenzen** von w gleichermaßen definiert:

$$\begin{aligned} w^0 &:= \lambda \\ w^1 &:= w \\ w^{n+1} &:= w^n \cdot w \end{aligned}$$

Die n-te Pozenz ist also das Wort, das durch n-fache Hintereinanderschreibung entsteht.

Für eine Sprache $L \subseteq \Sigma^*$ sind die **Potenzen** von L folgendermaßen definiert:

$$\begin{aligned} L^0 &:= \{\lambda\} \\ L^1 &:= L \\ L^{n+1} &:= L^n \cdot L \end{aligned}$$

Die n-te Potenz einer Sprache besteht also aus all denjenigen Wörtern, die das n-fache Produkt irgendwelcher Wörter aus L sind.

Für eine Sprache $L \subseteq \Sigma^*$ bezeichnet

$$L^* := \bigcup_{n=0}^{\infty} L^n$$

die **Kleene-Hülle** von L.

2.1.1 Spezielle Alphabete

$$\begin{aligned} \Sigma_{latein} &:= \{a, b, c, \ldots, x, y, z\} \\ \Sigma_{dezi} &:= \{0, \ldots, 9\} \\ \Sigma_{bool} &:= \{0, 1\} \\ \Sigma_{Tastatur} &:= \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, -, +, :, \ldots\} \cup \{\sqcup\} \end{aligned}$$

Achtung: Es gilt $|\sqcup| = 1$!

Bemerkung 2.1
Jeder Roman ist ein Wort über dem Alphabet $\Sigma_{Tastatur}$.

Wir definieren für Wörter $w \in \Sigma_{bool}^*$ der Form $x_1 \dots x_n$ die Funktion

$$No_2(w)\colon \Sigma_{bool}^* \to \mathbb{N}, w \mapsto \sum_{i=1}^{n} x_i 2^{n-i}$$

Das kürzeste Wort w mit der Eigenschaft $No_2(w) = n$ heißt **Binärdarstellung** von n. Schreibweise: $bin(n) := w$.

Für Wörter $w \in \Sigma_{dezi}^*$ definieren wir analog:

$$No_{10}(w) := \sum_{i=1}^{n} x_i 10^{n-i}$$

und die **Dezimaldarstellung** $dezi(n)$ als das kürzeste Wort w mit $No_{10}(w) := n$.

Eine Abbildung $f\colon \Sigma^* \to \Delta^*$ heißt **Homomorphismus** von Σ^* in Δ^* gdw_{def} für alle Wörter $u, v \in \Sigma^*$ gilt: $f(u \cdot v) = f(u) \cdot f(v)$.

2.2 Der Begriff der Turing-Maschine

Zunächst eine informale Beschreibung der Bestandteile einer Turing-Maschine:

- Ein nach rechts und links unendliches Arbeitsband (**Turing-Band**), das in Felder (Zellen) unterteilt ist.
- In jeder Zelle kann ein Buchstabe eines gegebenen Alphabets Σ stehen oder die Zelle ist leer.

  Bezeichnung für eine leere Zelle: $\square$

  Beispiel für $\Sigma = \Sigma_{latin}$ todo: Beispiel des Turing-Bandes
- Auf dem Band agiert ein Lese-Schreib-Kopf.
- Dieser Kopf steht auf genau einer Zelle des Bandes und kann in einem Takt den Inhalt dieser Zelle lesen und überschreiben.
- Ein endliches Gedächtnis steuert diesen Arbeitskopf und kann einen von endlich vielen inneren Zuständen annehmen.
- In Abhängigkeit vom gelesenen Symbol und dem inneren Zustand wird eine Aktion ausgeführt.
- Eine solche Aktion hat drei Bestandteile:
  - neues Symbol schreiben
  - neuen inneren Zustand annehmen
  - Bewegung des Kopfes um eine Zelle

Definition 2.1 (Formale Beschreibung der Turing-Maschine)
Eine **Turing-Maschine** M ist gegeben durch ein Sextupel $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ mit den Elementen

Q ... eine endliche Menge (Zustandsmenge)
Σ ... ein endliches Alphabet (Eingabealphabet)
Γ ... ein endliches Alphabet (Arbeitsalphabet), wobei $\Sigma \subseteq \Gamma$
δ ... Überführungsfunktion $Q \times \Gamma \to Q \times \Gamma \times \{L, 0, R\}$ (links, stehen, rechts)
$q_0 \in Q$ ... Startzustand
$F \subseteq Q$ ... Finalzustände

Vereinbarung: **Blanksymbol** $\square \in \Gamma \setminus \Sigma$ (damit die Turing-Maschine die Enden der Eingabe erkennen kann) und $Q \cap \Gamma = \emptyset$ (dies ist für die Definition einer Konfiguration (Definition 2.2) erforderlich).

Beispiel 2.1
Beispiel einer aktuellen Situation einer Turing-Maschine.

todo: bild einfügen

Es sei $\delta(q, b) = (q', a, R)$. Damit ist die neue Situation:

todo: bild einfügen

Bemerkung 2.2
Dieser Übergang heißt **Takt** der Maschine.

Definition 2.2
Es ein $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine Turing-Maschine. Eine **Konfiguration** von M ist ein Wort $\alpha \cdot q \cdot \beta \in \Gamma^* \times Q \times \Gamma^*$. Dabei bezeichnet $q \in Q$ den aktuellen Zustand der Maschine und $\alpha \cdot \beta \in \Gamma^*$ die aktuelle Bandinschrift der Maschine.

Bemerkung 2.3
Zu jedem Zeitpunkt sind höchstens endlich viele Symbole veschieden vom Blanksymbol. Damit beschreibt eine Konfiguration eine Momentansituation von M.

Beispiel 2.2
$K_1 = aqbba$ und $K_2 = aaq'ba$. Der Lese-Schreib-Kopf befindet sich auf der ersten Zelle des „rechten Teils“ von β.

Definition 2.3
Wir definieren auf der Menge der Konfigurationen eine zweistellige Relation

$$\vdash \subseteq (\Gamma^* \cdot Q \cdot \Gamma^*) \times (\Gamma^* \cdot Q \cdot \Gamma^*)$$

$$a_1 a_2 \dots a_m \underbrace{q}_{\in Q} b_1 b_2 \dots b_n \vdash \begin{cases} a_1 a_2 \dots a_m c q' b_2 \dots b_n & : \delta(q, b_1) = (q', c, R) \\ a_1 a_2 \dots a_m q' c b_2 \dots b_n & : \delta(q, b_1) = (q', c, 0) \\ a_1 a_2 \dots q' a_m c b_2 \dots b_n & : \delta(q, b_1) = (q', c, L) \end{cases}$$

Zwei Sonderfälle:

1. Der Lese-Schreib-Kopf steht am rechten Rand und bewegt sich nach rechts:

$$a_1 \dots a_m q b_1 \mapsto a_1 \dots a_m c q' \square$$

   falls $\delta(q, b_1) = (q', c, R)$.

2. Der Lese-Schreib-Kopf steht am linken Rand und bewegt sich nach links:

$$q b_1 b_2 \dots b_n \mapsto q' c \square b_2 \dots b_n$$

   falls $\delta(q, b_1) = (q', c, L)$.

Definition 2.4
Es sei M eine Turing-Maschine und es seien K_1, K_2 zwei Konfigurationen von M.

1. K_2 ist **unmittelbare Nachfolgekonfiguration** von K_1 gdw_{def} $K_1 \vdash K_2$.

2. K_2 ist eine **Folgekonfiguration** von K_1 gdw_{def} $K_2 = K_1$ oder es existiert eine endliche Folge $K_1', K_2', \dots, K_m'$ mit $K_1 = K_1' \vdash K_2', K_2' \vdash K_3', \dots, K_{m-1}' \vdash K_m' = K_2$.

   Schreibweise: $K_1 \vdash^* K_2$

Bemerkung 2.4
$\vdash^*$ ist eine reflexive und transitive Hülle von $\vdash$.

Definition 2.5
Die **Startkonfiguration** der Turing-Maschine M bei der Eingabe $w \in \Sigma^*$ ist $q_0 w = Start - Konf_M(w)$. Jede Konfiguration von M der Form $\alpha q_F \beta$ heißt **Endkonfiguration** (oder **Finalzustand**), falls $q_F \in F$.

2.2.1 Beispiele für Turing-Maschinen

Beispiel 2.3
$L = \{w \in \{a, b\}^* : w^R = w\}$ ist die Menge aller **Palindrome** über $\{a, b\}$.

Ziel: Konstruktion einer Turing-Maschine M, die L im folgenden Sinne erkennt:

1. $w \in L$, dann löscht M die Eingabe w, schreibt ein „a“ und stoppt im Finalzustand.

2. $w \in \{a, b\}^* \setminus L$, dann löscht M diese Eingabe w, schreibt ein „b“ und stoppt im Finalzustand.

Zustände:
q_0 ... Startzustand
q_a ... merke sich „a“ und Kopf läuft nach rechts
q_b ... merke sich „b“ und Kopf läuft nach rechts
q'_a ... testet „a“ und Kopf läuft nach links
q'_b ... testet „b“ und Kopf läuft nach links
q_+ ... Test positiv
q_- ... Test negativ
q_F ... Finalzustand
$\Sigma = \{a, b\}$
Zustandsüberführungsfunktion δ:

Q \ Γ	a	b	$\square$
q_0	$(q_a, \square, R)$	$(q_b, \square, R)$	$(q_F, a, 0)$
q_a	(q_a, a, R)	(q_b, b, R)	$(q'_a, \square, L)$
q_b	(q_b, a, R)	(q_b, b, R)	$(q'_b, \square, L)$
q'_a	$(q_+, \square, L)$	$(q_-, \square, L)$	$(q_F, a, 0)$
q'_b	$(q_-, \square, L)$	$(q_+, \square, L)$	$(q_F, a, 0)$
q_+	(q_+, a, L)	(q_+, b, L)	$(q_0, \square, R)$
q_-	$(q_-, \square, L)$	$(q_-, \square, L)$	$(q_F, b, 0)$
q_F	$(q_F, a, 0)$	$(q_F, b, 0)$	$(q_F, \square, 0)$

Die Überführungsfunktion ist stets eine **totaldefinierte Fuktion**! Das bedeutet, es gibt keine undefinierten Situationen oder bildlich gesprochen, keine der Tabellenzellen ist leer! Wenn ein Finalzustand erreicht wird, stoppt die Maschine (von außen betrachtet) bzw. ist in einer Endlosschleife (von innen betrachtet).

Beispiel 2.4
Wir konstruieren eine Maschine M, die die Sprache $L = \{a^n b^n : n \geq 1\} \subseteq \{a, b\}^*$ erkennt.

M arbeitet in zwei Etappen:

1. M testet, ob die Eingabe von der Form $a^i b^j$ ist.
2. M testet, ob $i = j$ ist.

q_0 ... falls „a“ gelesen wird: Bewegung nach rechts
falls „b“ gelesen wird: Übergang in q_1
falls „$\square$“ gelesen wird: „Fehler“
q_1 ... falls „a“ gelesen wird: „Fehler“
falls „b“ gelesen wird: Bewegung nach rechts
falls „$\square$“ gelesen wird: Übergang in q_2
q_2 ... löscht das rechteste „b“: Übergang in q_3
lese „a“ oder „$\square$“: „Fehler“
q_3 ... Bewegung nach links bis zum ersten $\square$: Übergang in q_4
q_4 ... streiche das „a“: Übergang zu q_5
falls „b“ oder „$\square$“: „Fehler“

q_5 ... testet, ob der „Rest“ das leere Wort ist: Fianlzustand q_7
lesse „a“ oder „b“: Übergang zu q_6
q_6 ... Bewegung nach rechts bis zum ersten $\square$: Übergang zu q_2
q_7 ... Finalzustand

	a	b	c
q_0	(q_0, a, R)	(q_1, b, R)	$(q_0, \square, 0)^\star$
q_1	$(q_1, a, 0)^\star$	(q_1, b, R)	$(q_2, \square, L)$
q_2	$(q_2, a, 0)^\star$	$(q_3, \square, L)$	$(q_2, \square, 0)^\star$
q_3	(q_3, a, L)	(q_3, b, L)	$(q_4, \square, R)$
q_4	$(q_5, \square, R)$	$(q_4, b, 0)^\star$	$(q_4, \square, 0)^\star$
q_5	(q_6, a, R)	(q_6, b, R)	$(q_7, \square, 0)$
q_6	(q_6, a, R)	(q_6, b, R)	$(q_2, \square, L)$
q_7	$(q_7, a, 0)$	$(q_7, b, 0)$	$(q_7, \square, 0)$

**: Für diese Zustände stoppt die Maschine mit „Fehler“.*

Es gilt: M erreicht bei Eingabe w den Finalzustand q_7 *gdw* $w \in L$.

2.3 Exkurs über Zahlenfunktionen

Es sei $b > 1 \in \mathbb{N}$ **Basis**

2.3.1 *b*-näre Zahlendarstellung

Ziffern $\{0, 1, \ldots, b-1\}$

Fakt: Jede natrüliche Zahl $n \geq 0$ besitzt eine eindeutige Darstellung der Form

$$m = x_m b^m + x_{m-1} b^{m-1} + \ldots + x_2 b^2 + x_1 b + x_0$$

wobei $x_m, \ldots, x_0 \in \{0, \ldots, b-1\}$ und $x_m \neq 0$ sind.

Schreibweise: $bnaer(n) = x_m x_{m-1} \ldots x_2 x_1 x_0$

$bnaer \colon \mathbb{N} \to \{0, 1, \ldots, b-1\}^*$ ist eine eindeutige, aber nicht bijektive Abbildung.

speziell: $b = 2$ mit den Ziffern $\{0, 1\}$ heißt binär; z. B.: $bin(17) = 10001$, aber $010001 \in \{0, 1\}^*$ ist kein Bild der Abbildung.

2.3.2 b-adische Darstellung

Ziffern $\{1, 2, \ldots, b\}$

Fakt
jede natürliche Zahl $n > 0$ besitzt eine eindeutige Darstellung der Form

$$n = y_n b^n + y_{n-1} b^{n-1} + \ldots + y_1 b^1 + y_0$$

wobei $y_n, y_{n-1}, \ldots, y_0 \in \{1, 2, \ldots, b\}$

Schreibweise: $bad(m) = y_n y_{n-1} \ldots y_0$, $bad(0) = \lambda$.

Die Abbildung $bad\colon \mathbb{N} \to \{1, 2, \ldots, b\}^*$ ist eine Bijektion.

speziell: $b = 2$ dyadisch; $dya\colon \mathbb{N} \to \{1, 2\}^*$ ist injektiv und surjektiv; z. B.: $dya(17) = 1121$

2.4 Turing-Berechenbarkeit

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine Turing-Maschine. Ferner seien $\square, \# \in \Gamma, \Delta \subseteq \Gamma \backslash \{\square\}, \Sigma \subseteq \Gamma \setminus \{\square, \#\}$ und $n > 0$.

Eine Turing-Maschine M **berechnet** eine (n-stellige) Funktion $f\colon (\Sigma^*)^n \to \Delta^*$, falls gilt:

1. Für $(w_1, w_2, \ldots, w_n) \in (\Sigma^*)^n$ sei f definiert und für die Startkonfiguration $q_0 w_1 \# w_2 \# \ldots \# w_n$ gilt:

   $$q_0 w_1 \# w_2 \# \ldots \# w_n \vdash^* q_F f(w_1, w_2, \ldots, w_n)$$

   mit $q_F \in F$.

2. Falls f für $(w_1, w_2, \ldots, w_n) \in (\Sigma^*)^n$ **nicht definiert** (Symbol: $\perp$) ist, dann ausgehend von $q_0 w_1 \# w_2 \# \ldots \# w_n$, stoppt M nicht oder M stoppt *nicht* in einem Finalzustand.

Definition 2.6
Eine **Wortfunktion** $f\colon (\Sigma^*)^n \to \Delta^*$ heißt **Turing-berechenbar** gdw_{def} es eine Turing-Maschine M gibt, die f berechnet.

Definition 2.7
Eine **Zahlenfunktion** $g\colon \mathbb{N}^n \to \mathbb{N}$ heißt **Turing-berechenbar** gdw_{def} g bei dyadischer Codierung f_g (f_g ist eine Wortfunktion $\{1, 2\}^* \to \{1, 2\}^*$) Turing-brechenbar ist, d. h.

$$f_g(y_1, y_2, \ldots, y_n) = dya\Big(g\big(dya^{-1}(y_1), \ldots, dya^{-1}(y_n)\big)\Big)$$

Definition 2.8
Eine Sprache $L \subseteq \Sigma^*$ heißt **Turing-entscheidbar** gdw_{def} die **charakteristische Funktion** χ_L Turing-berechenbar ist

$$\chi_L(w) = \begin{cases} 1 & : w \in L \\ 0 & : w \notin L \end{cases}$$

Definition 2.9
Eine Sprache $L \subseteq \Sigma^*$ heißt **Turing-semientscheidbar** gdw_{def} die **partielle charakteristische Funktion** χ_L^p Turing-berechenbar ist

$$\chi_L^p(w) = \begin{cases} 1 & : w \in L \\ \perp & : w \notin L \end{cases}$$

Definition 2.10
Eine Sprache $L \subseteq \Sigma^*$ heißt **Turing-aufzählbar** gdw_{def} eine total definierte Turing-berechenbare Funktion $f\colon \Sigma^* \to_t \Sigma^*$ existiert, so dass $f(\Sigma^*) = L$.

Bemerkung 2.5
f könnte z. B. die folgende Abbildung machen

$$f(w) = \begin{cases} w & : w \in L \\ u & : w \notin L \end{cases}$$

u ist dabei ein bekanntes festes Wort aus L. Dies könnte z. B. λ sein, falls $\lambda \in L$.

2.4.1 Normierte Startsituation

- Definition trifft keine Aussage über das Verhalten einer Turing-Maschine für Eingaben, die nicht von der erwarteten Form sind. Es ist nicht geklärt, was passiert, wenn Zeichen $\notin \Sigma \cup \Gamma$ auf dem Band stehen.
- Jede Turing-Maschine berechnet für jede natürliche Zahl $n > 0$ eine n-stellige Funktion.

2.4.2 partielle Funktionen

Als berechenbare Funktionen werden auch solche angesehen, die nicht überall definiert sind.
Beispiel 2.5
Wir definieren für $\Sigma = \{1, 2\}$ die Turing-Maschine $M = (\{q_0, q_F\}, \Sigma, \Sigma \cup \{\square\}, \delta, q_0, \{q_F\})$

$$\begin{aligned} \delta(q_0, 1) &= (q_0, 1, R) & \qquad \delta(q_F, 1) &= (q_F, 1, 0) \\ \delta(q_0, 2) &= (q_0, 2, R) & \qquad \delta(q_F, 2) &= (q_F, 2, 0) \\ \delta(q_0, \square) &= (q_0, \square, R) & \qquad \delta(q_F, \square) &= (q_F, \square, 0) \end{aligned}$$

$\delta(q_F, *)$ ist die Verfollständigung der Überführungsfunktion, obwohl q_F nie erreicht wird.

M berechnet die **„nirgends definierte“ Funktion** $f\colon \Sigma^* \to \Sigma^*$

$$f(w) = \perp \qquad \forall w \in \Sigma^n$$

Dies gilt für den Definitionsbereich $D \neq \emptyset$ und damit $f(D) = \emptyset$

2.4.3 Wortfunktionen

Die Abbildung $bad\colon \mathbb{N} \to \{1, 2, \ldots, b\}^*$ (für eine Basis $b > 1$) aus Abschnitt 2.3.2 ist eine Bijektion zwischen den Zahlen und den Wörten. Diese Bijektion erlaubt eine natürliche Zahl n mit dem Wort $bad(n)$ und schließlich die Menge $\mathbb{N}$ mit $\{1, 2, \ldots, b\}^*$ zu identifizieren.

Jedes Alphabet Σ mit k Buchstaben kann durch Umbennen in $\{1, 2, \ldots, k\}$ übersetzt werden und damit kann Σ^* mit $\mathbb{N}$ identifiziert werden.

Beispiel 2.6
Eine Turing-Maschine, die die Nachfolgerfunktion dyadischer Darstellung berechnet:
$\Sigma = \{1, 2\}, \Gamma = \{1, 2, \square\}$

q_0 ... Startzustand
q_{U1} ... Übertrag 1
q_{U0} ... Übertrag 0
q_F ... Finalzustand

δ	1	2	$\square$
q_0	$(q_0, 1, R)$	$(q_0, 2, R)$	$(q_{U1}, \square, L)$
q_{U1}	$(q_{U0}, 2, L)$	$(q_{U1}, 1, L)$	$(q_F, 1, 0)$
q_{U0}	$(q_{U0}, 1, L)$	$(q_{U0}, 2, L)$	$(q_F, \square, R)$
q_F	$(q_F, 1, 0)$	$(q_F, 2, 0)$	$(q_F, \square, 0)$

Wir haben in Definition 2.7 die Festlegung getroffen: Eine Zahlenfunktion $g\colon \mathbb{N}^n \to \mathbb{N}$ heißt Turing-berechenbar gdw_{def} die Wortunktion f Turing-berechenbar ist:

$$f\colon (\{1,2\}^*)^n \to \{1,2\}^*, f(y_1, \ldots, y_n) = dya\Big(g\big(dya^{-1}(y_1), \ldots, dya^{-1}(y_n)\big)\Big)$$

Eine „großzügigere“ Definition der Berechenbarkeit wäre die folgende:

Definition 2.11
Eine Zahlenfunktion $g\colon \mathbb{N}^n \to \mathbb{N}$ heißt **Turing-berechenbar** gdw_{def} zwei natürliche Zahlen $b_1, b_2 > 1$ und eine Turing-berechenbare Funktion

$$h\colon (\{1, \ldots, b_1\}^*)^n \to \{1, \ldots, b_2\}^*, (z_1, \ldots, z_n) \mapsto b_2ad\Big(g\big(b_1ad^{-1}(z_1), \ldots, b_1ad^{-1}(z_n)\big)\Big)$$

existieren.

Frage: Führen diese zwei verschiedenen Definitionen auf zwei verschiedene Arten von Berechenbarkeit?

Wir zeigen: Falls h Turing-berechenbar ist, dann folgt daraus, dass auch f Turing-berechenbar ist. Beide Definitionen sind also gleichwertig.

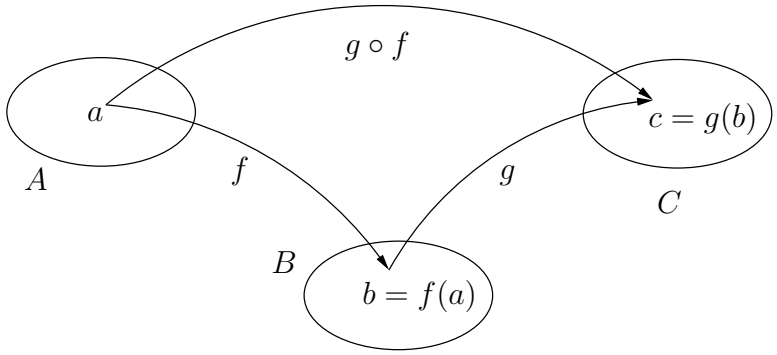
$$\begin{aligned} f(y_1,\dots,y_n) &= dya\Big(g(dya^{-1}(y_1),\dots,dya^{-1}(y_n))\Big) \\ &= dya\Bigg(b_2ad^{-1}\Big(h\big(b_1ad(dya^{-1}(\underbrace{\underbrace{\underbrace{\underbrace{y_1}_{\in\{1,2\}^*}}_{\in\mathbb{N}}}_{\in\{1,\dots,b_1\}^*}}_{\in\{1,\dots,b_2\}^*}}_{\in\mathbb{N}})),\dots,b_1ad(dya^{-1}(y_n))\big)\Big)\Bigg) \\ &= (b_2ad^{-1}\circ dya)h((dya^{-1}\circ b_1ad)(y_1),\dots,(dya^{-1}\circ b_1ad)(y_n)) \end{aligned}$$

Fakt

$(b_2ad^{-1}\circ dya)$	und	$(dya^{-1}\circ b_1ad)$
Umrechnung von b_2-adischer in dyadische Darstellung		Umrechnung von dyadischer in b_1-adishe Darstellung

sind „einfach“ durch Turing-Maschinen berechenbar.

Bemerkung 2.6 (Verkettung von Funktionen)
zur Verkettung von Funktionen:



$$c = g(b) = g(f(a)) = (f \circ g)(a)$$

Es gilt: $(a,c) \in f \circ g \;\; gdw_{def} \;\; \exists b \in B\colon (a,b) \in f, (b,c) \in g$

Fazit: Es gibt *einen* Begriff der Turing-Berechenbarkeit für Wort- und Zahlenfunktionen, unabhängig von der Darstellung.

Definition 2.12
$\mathcal{TM}$ ist die **Klasse aller Turing-berechenbarer Funktionen**.

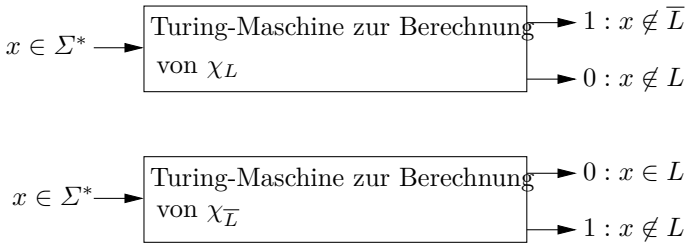
$$\mathcal{TM} = \{f\colon \mathbb{N}^n \to \mathbb{N} | f \text{ ist Turing-berechenbar}\}$$

2.4.4 Entscheidbarkeit, Semientscheidbarkeit und Aufzählbarkeit

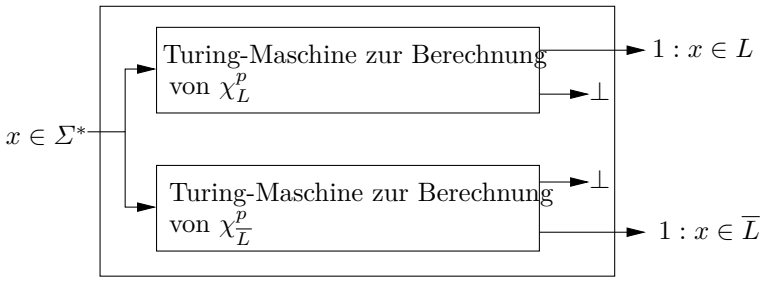
Alle diese Begriffen sind zurückgeführt auf die Berechenbarkeit gewisser Funktionen und damit spezielle Formulierungen der Berechenbarkeit.

Beziehungen zwischen den Begriffen

1. Beobachtung: Die Sprache $L \subseteq \Sigma^*$ ist Turing-entscheidbar *gdw* $\overline{L} = \Sigma^* \setminus L$ ist Turing-entscheidbar



2. Beobachtung: Die Sprache $L \subseteq \Sigma^*$ ist Turing-entscheidbar *gdw* die Sprache L und die Komplementsprache $\overline{L}$ Turing-semientscheidbar sind.



3. Beobachtung: Eine Sprache $L \subseteq \Sigma^*$ ist Turing-semientscheidbar *gdw* L ist aufzählbar.

   $\Leftarrow$ Wenn eine Sprache aufzählbar ist, dann gibt es eine Turing-Maschine M_{aufz}, die die Funktion $f(\Sigma^*) = L$ berechnet (Definition 2.10). Eine Turing-Maschine M_{semi} kann für eine Eingabe w alle Wörter der Sprache L von M_{aufz} erzeugen lassen und vergleicht, ob die Eingabe w dabei ist. Findet sie diese, stoppt M_{semi} mit 1, andernfalls ist die Ausgabe undefiniert. Damit berechnet M_{semi} genau die partielle charakteristische Funktion χ_L^p (Definition 2.9).

   $\Rightarrow$ Umgekehrt existiere eine Turing-Maschine M_{semi}, die für eine Eingabe w entscheidet, ob diese zu L gehört. M_{semi} kann zur Bestimmung, ob ein Wort $u \in L$ ist, 1,2,3,...viele Schritte benötigen. Man lässt aber M höchstens k Schritte machen und gibt ein festes (bekanntes) $a \in L$ aus, falls M nicht nach k Schritten gestoppt

hat. Einer Eingabe w wiederum lässt sich eineindeutig ein Paar $(u, k) \in \Sigma^* \times \mathbb{N}$ zuordnen[1].

Damit kann man eine Turing-Maschine M_{aufz} konstruieren, die für ein Wort $w \in \Sigma^*$ das Paar $(u, k) \in \Sigma^* \times \mathbb{N}$ bestimmt und prüft, ob $M_{semi}(u)$ nach k Takten stoppt. Ist dies der Fall, gibt M_{aufz} u aus, ansonsten a. M_{aufz} gibt also nur Wörter aus L aus – die Funktion, die M_{aufz} berechnet ist total.

Für ein Wort $u \in L$ stoppt M_{semi} nach einer endlichen Anzahl k an Takten, d. h. es gibt ein Paar (u, k) und damit ein $w \in \Sigma^*$, für das die Maschine M_{aufz} das Wort u ausgibt. M_{aufz} gibt also (irgendwann) jedes Wort aus L aus – Vollständigkeit.

2.4.5 Turing-Maschinen als Akzeptoren und Erkenner

Eine Sprache L ist Turing-entscheidbar *gdw* χ_L ist Turing-berechenbar (vgl. Definition 2.8), d. h. falls eine Eingabe $w \in \Sigma^*$ zu L gehört, endet die Berechnung einer Turing-Maschine in der Konfiguration $\square q_F 1 \square$ und falls $w \notin L$ endet die Berechnung in $\square q_F 0 \square$.

Wir definieren neue Finalzustände (einer neuen Turing-Maschine):

- aus der Konfiguration $\square q_F 1 \square \vdash \square q_+ \square\square$, q_+ heißt **akzeptierender Zustand** (wobei das Turing-Band leer ist)
- aus der Konfiguration $\square q_F 0 \square \vdash \square q_- \square\square$, q_- heißt **ablehnender Zustand** (wobei das Turing-Band leer ist)

Eine solche Turing-Maschine (ohne Ausgabe) heißt **Erkenner** der Sprache L mit dem akzeptierenden Finalzustand q_+ und dem ablehnenden Finalzustand q_-.

L ist Turing-semientscheidbar *gdw* χ_L^p ist berechenbar (vgl. Definition 2.9).

Gleichermaßen definieren wir einen akzeptierenden Zustand q_+ als denjenigen Finalzustand von $\square q_F 1 \square \vdash \square q_+ \square\square$. Es gibt jedoch keinen ablehnenden Zustand. Eine solche Turing-Maschine heißt **Akzeptor** der Sprache L.

[1] Ein u kann durch seine Stellung in Σ^* als Zahl $j(u)$ repräsentiert werden. Weiterhin lässt sich jedes $u_i \in \Sigma^*$ mit einer Zahl $i \in \mathbb{N}$ identifizieren und das Paar $(i, k) \in \mathbb{N} \times \mathbb{N}$ lässt sich durch Diagonalisierung mit einer Zahl $j(u)$ identifizieren – DML: Gleichmächtigkeit von $\mathbb{Z}$ und $\mathbb{N}$.

2.5 Techniken zur Programmierung von Turing-Maschinen

2.5.1 Endliche Sprache mit schnellem Zugriff

Die Daten einer endlichen Datenmenge D können durch folgenden Trick in das „Gedächtnis“ der Turing-Maschine eingeführt werden:

$$Q_{neu} = Q_{alt} \times D$$

Bemerkung 2.7
Dies ist keine Änderung des bisherigen Begriffs Turing-Maschine, sondern lediglich eine *Modifikation*, die eine Konstruktion wahrnimmt!

Beispiel 2.7
Für $w \in \Sigma^*$ bezeichnet $|w|$ die Länge von w, d. h. w hat $|w|$ viele Buchstaben. (vgl. Beginn von Abschnitt 2.1)

Wir betrachten die folgende Sprache

$$L = \{w = w_1 w_2 \dots w_{|w|} \colon w_1 \neq w_j, j = 2, \dots, |w|\} \subseteq \Sigma^*$$

Wir konstruieren folgende Turing-Maschine M: $\Sigma = \{a, b, c\}, \Gamma = \Sigma \cup \{\square\}$
$Q_{neu} = \{q_1, q_2, q_3\} \times \Gamma$
Startzustand: $[q_0, \square]$
Finalzustand: $[q_F, \square]$
Überführungsfunktion:

$$\begin{aligned}
\delta([q_0, X], X) &= ([q_1, X], \square, R) && X \in \{a, b, c\} \\
\delta([q_0, \square], \square) &= ([q_F, \square], \square, 0) && \text{(Finalzustand)} \\
\delta([q_1, X], Y) &= ([q_1, X], \square, R) && X, Y \in \{a, b, c\}, X \neq Y \\
\delta([q_1, X], \square) &= ([q_1, \square], \square, 0) && \text{(Finalzustand)} \\
\delta([q_1, X], X) &= ([q_2, \square], X, R) && X \in \{a, b, c\} \\
\delta([q_2, \square], X) &= ([q_2, \square], X, L) && (M \text{ stoppt nicht})
\end{aligned}$$

M ist ein Akzeptor für L.

2.5.2 Turing-Bänder mit mehreren Spuren – Mehrspurmaschinen

informell: Das Turing-Band ist in mehrere Zeilen bzw. Spuren unterteilt

formal:

$$\Gamma_{neu} = \Sigma \cup \Gamma_{alt}^{(k)} = \Sigma \cup (\Gamma_{alt} \times \ldots \times \Gamma_{alt})$$

Dabei ist $\square^k$ das neue Blanksymbol

Bemerkung 2.8
Analog zu Bemerkung 2.7 ist dies keine Änderung des Begriffs Turing-Maschine.

Beispiel 2.8 (Addition mehrstelliger Zahlen in dyadischer Darstellung)

$$\begin{array}{r} 1221 \\ 1221 \\ \hline 12122 \end{array}$$

Eingabe: $\square dya(x) \# dya(y) \square$
$\Sigma = \{1, 2, \#\}$

1. Initialisierung: $k = 3$ (3 Spuren)

$$\begin{aligned} b \in \Sigma &\mapsto (b, \square, \square) \\ \square &\mapsto (\square, \square, \square) \end{aligned}$$

wir erhalten als Bandinhalt:

$$\begin{array}{ccccc} \square & dya(x) & \# & dya(y) & \square \\ \square & \square \ldots \square & \square & \square \ldots \square & \square \\ \square & \square \ldots \square & \square & \square \ldots \square & \square \end{array}$$

2. Übertragung von $dya(y)$ in die 2. Spur:

$$\begin{array}{ccc} \square & dya(x) & \square \\ \square & dya(y) & \square \\ \square & \square \ldots \square & \square \end{array}$$

3. Stellenweise Addition:

$$\begin{array}{ccc} \square & dya(x) & \square \\ \square & dya(y) & \square \\ \square & dya(x+y) & \square \end{array}$$

4. Normierung auf eine Spur $(\gamma_1, \gamma_2, \gamma_3) \mapsto \gamma_3$

Beispiel 2.9 (Multipikation mehrstelliger Zahlen in dyadischer Darstellung)
Bei der Multiplikation zweier Faktoren x und y benötigt das herkömmliche Verfahren $\lfloor \log_2 y \rfloor$ viele Zeilen – also nicht konstant viele!

$1221 \cdot 1$	$1221 \cdot 2$	$1221 \cdot 11$	$1221 \cdot 12$	$1221 \cdot 21$	$1221 \cdot 111$
1221	12122	1221	12122	1221	1221
		1221	1221	12122	1221
		111111	121212	212121	1221
					1121211

Die Idee ist aber, den erste Faktor x mit jeder Stelle des zweiten Faktors y einzeln zu multiplizieren und die Ergebnisse aufzusummiert.

1. Initialisierung: $k = 4$ Spuren
2. Frage: $y = 0$? $\rightarrow$ ja: gehe zu 3.; nein: gehe zu 4.
3. Lösche den Bandinhalt und stoppe
4. 2. Spur Zwischenergebnis x mal letzte Ziffer von y.
5. Frage: hat y ein weiteres Bit? $\rightarrow$ ja: gehe zu 6.; nein: gehe zu 9.
6. 3. Spur Zwischenergebnis x mal aktuelles Bit von y.
7. 4. Spur Zwischenergebnis 2.+3. Spur
8. Übertrage das Zwischenergebnis Summe 2.+3. Spur in die 2. Spur und lösche die 3. und 4. Spur, gehe zu 5.
9. Normierung: Das Ergbenis $dya(x + y)$ steht in der 2. Spur und wird in eine Spur zurückübersetzt

2.5.3 Mehrbandmaschninen

k sei fixiert, k-Band-Turing-Maschine

informell: k Bänder mit *je* einem Kopf

formell: $\delta \colon Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, 0, R\}^k$

Bemerkung 2.9
Dies ist eine Erweiterung der bisherigen Definition der Turing-Maschine. help: Wieso das auf einmal? Weil sich die Abbildungsfunktion geändert hat. Vorher wurden die Mengen Q und Γ immer nur uminterpretiert, jetzt wird richtig was geändert.

Beispiel 2.10
$k = 2, L = \{w \in \{a, b\}^* : w = w^R\}$

Wir wissen bereits aus Beispiel 2.3, dass L Turing-entscheidbar durch eine gewöhnliche (1-Band-) Turing-Maschine ist.

Idee:

1. q_0 läuft an das rechte Ende der Eingabe, $\to q_1$
2. q_1 liest die Eingabe von rechts nach links und kopiert sie buchstabenweise auf das 2. Band, $\to q_2$
3. q_2 bewegt den Kopf auf Band 2 an das linke Ende, $\to q_3$
4. q_3 vergleicht beide Bandinhalte buchstabenweise; stets Übereinstimmung $\to$ akzeptiere; irgendwo Unterschied $\to$ ablehnen

Satz 2.1
Jede Sprache $L \subseteq \Sigma^*$, die von einer k-Band-Turing-Maschine entschieden werden kann, kann auch von einer gewöhnlichen (1-Band-) Turing-Maschine entschieden werden.

Beweis:
Es sei M eine k-Band-Turing-Maschine, die die Sprache $L \subseteq \Sigma^*$ entscheidet. (Schreibweise: $L(M) = L$: M entscheidet L)

Wir konstruieren eine 1-Band-Turing-Maschine M', die die Arbeit von M nachvollzieht, d. h. „simuliert“. (Beweis durch Simulation) M' vollzieht die Arbeit von M mit ihren Mitteln nach. Das Arbeitsband von M' ist unterteilt in $(2k + 1)$ Spuren. Die Spuren $1, 3, \ldots, 2k - 1$ enthalten Inhalte der Bänder $1, 2, \ldots, k$ von M. Die Spuren $2, 4, \ldots, 2k$ markieren die Kopfposition auf den Bändern $1, 2, 3, \ldots, k$ von M. Die Spur $2k + 1$ enthält eine linke und eine rechte Endmarke, die das am weitesten links bzw. rechts liegene Feld eines der Bänder markiert.

Ausgangssituation...

...von M:

$\square$	w_1	w_2	$\ldots$	w_n	$\square$	1. Band
	$\uparrow$					
$\square$	$\square$		$\ldots$	$\square$	$\square$	2. Band
	$\uparrow$					Startzustand: q_0
			$\vdots$			$\vdots$
$\square$	$\square$		$\ldots$	$\square$	$\square$	k. Band
	$\uparrow$					

...von M': $\square w_1 w_2 \ldots w_n \square$ einzigstes Band, Startzustand q_0'

1. **Initialisierung** (Einrichten der Spuren):

1.	□	w_1	w_2	...	w_n	□	
2.	□	▲	□	...	□	□	
3.	□	□		...	□	□	$(k = 2)$
4.	□	▲	□	...	□	□	
5.	▷	□		...		◁	
	↑						

2. **Simulation**: um einen Schritt der Arbeit von M nachzuvollziehen, führt M' folgende Schritte aus:

   a) der Kopf von M' bewegt sich von links nach rechts und merkt sich für jedes Band von M das aktuelle gelesene Symbol. Am rechten Rand kennt M' den aktuellen Zustand und die aktuell gelesenen Symbole und kann damit gemäß der Überführungsfunktion δ von M einen neuen aktuellen Zustand q_{t+1} von M und k neue Buchstaben (von M) und k Bewegungen der Köpfe bestimmen.

   b) Der Kopf von M' bewegt sich von rechts nach links und überschreibt die Buchstaben in den markierten Feldern gemäß δ, verschiebt die Kopfmarkierungen gemäß δ und erreicht die linke Endmarke. Falls hierbei eine Zelle, die eine der beiden Endmarken enthält, beschriftet wird, wird die Marke um ein Feld nach links bzw. rechts verschoben, so dass der Arbeitsbereich von M' stets zwischen den Endmarken liegt.

   Damit erreicht M' eine normierte Zwischensituation und der nächste Schritt von M kann simuliert werden.

   M' akzeptiert eine Eingabe w *gdw* M diese Eingabe akzeptiert.

3. **Normierung** (Band aufräumen) ■

2.5.4 Unterprogramme

Definition 2.13
Unterprogramme sind eigenständige Turing-Maschinen, die in größere Turing-Maschinen eingebaut sind.

- Ein Teil der Zustände von M und des Arbeitsspeichers von M sind für das Unterprogramm reserviert.
- In gewissen Situationen der Hauptprogramms werden Daten an ein Unterprogramm übergeben, dort behandelt und wieder zurück an das Hauptprogramm übergeben.

Beispiel 2.11 (Multiplikation von natürlichen Zahlen in unärer Darstellung)
Eingabe: der Form $1^n\#1^m$

$$\square \underbrace{1 \ldots 1}_{n-\text{mal}} \# \underbrace{1 \ldots 1}_{m-\text{mal}} \square$$

Resultat: $1^{n \cdot m}$

Idee: Kopiere ersten Block (1^n) so oft, wie es der zweite Block angibt. Konstruiere eine 3-Band-Turing-Maschine

1. Band: Eingabe
2. Band: 2. Block (1^m)
3. Band: Ergebnis

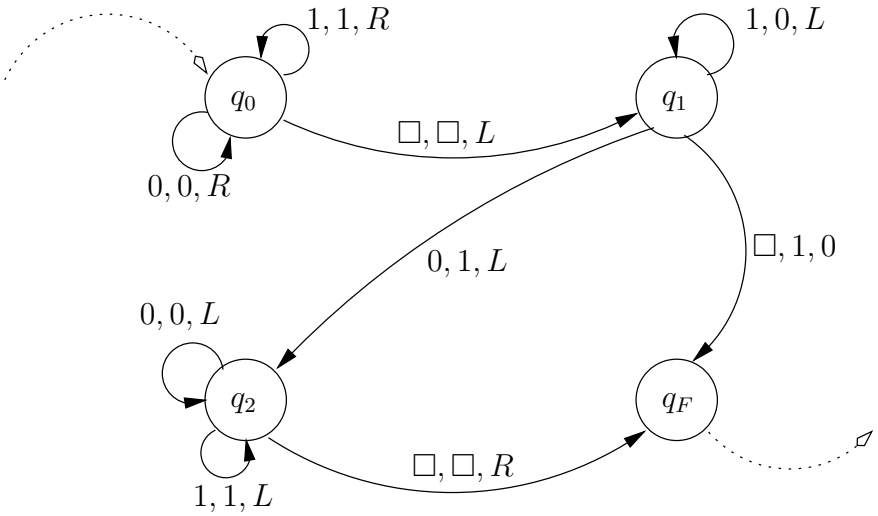
Wir beschreiben die Arbeit der Maschine:

- q_0 ... Überlaufen des ersten Blocks $\rightarrow q_1$
- q_1 ... Überlaufen des zweiten Blocks nach rechts und kopiere Inhalt auf das zweite Band (lösche ihn auf dem ersten Band) $\rightarrow q_2$
- q_2 ... falls auf dem zweiten Band ein Strich gelesen wird, dann gehe zu q_u, falls ein $\square$ gelesen wird, gehe zu q_3
- q_u ... kopiere den Inhalt des ersten Bandes auf das dritte Band $\rightarrow q'_u$
- q'_u ... gehe mit den Köpfen in die Ausgangsposition $\rightarrow q_2$
- q_3 ... Normierung – Resultat auf das erste Band schreiben und Aufräumen $\rightarrow q_F$
- q_0–q_3 ... Turing-Maschine mit Startzustand q_0, Finalzustand q_F
- q_u, q'_u ... Unterprogramm, Startzustand q_u, Finalzustand q'_u, eingebettet in große Turing-Maschine

2.5.5 Komposition von Turing-Maschinen

Transitionsdiagramme

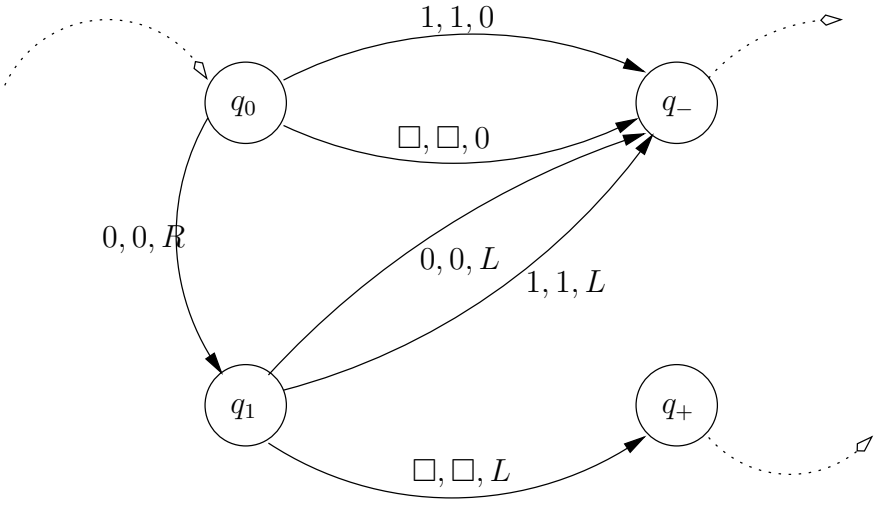
Beispiel 2.12
Am Beispiel einer Turing-Maschine, die binär eine Eins addiert. Zustände der Maschine werden als Knoten in einem Diagramm dargestellt. Die Überführungsfunktion mittels bewerteter Kanten.



Die Maschine „$i := i + 1$“

Dieser (Mega-)Knoten lässt sich in ein weiteres Transitionsdiagramm (einer größeren Turing-Maschine) einsetzen.

Beispiel 2.13
Eine Turing-Maschine, die testet, ob der Bandinhalt „0“ ist oder nicht (binäre Darstellung)



Die Maschine „$i = 0$“

Die Anschlüsse für dieses Unterprogramm (Knoten) führen in den Startzustand und verlassen die Endzustände.

Hintereinanderschalten von Turing-Maschinen

Es seien $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_{0_i}, F_i)$ $(i = 1, 2)$ zwei Turing-Maschinen. o. B. d. A. $Q_1 \cap Q_2 = \emptyset$. Die Turing-Maschine $M = (Q_1 \cup Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_{0_1}, F_2)$ realisiert ein hintereinanderschalten von M_1 und M_2. Dabei gilt:

$$\delta = \delta_1 \cup \delta_2 \cup \{(q_1, a) \mapsto (q_{0_2}, a', 0) \colon q_1 \in F_1, a \in \Gamma_1, a' \in \Gamma_2\}$$

Notation: $start \to M_1 \to M_2 \to stopp$ oder $M := M_1 \,;\, M_2$

Beispiel 2.14

$$\underbrace{start \to \boxed{\text{i:=i+1}} \to \boxed{\text{i:=i+1}} \to stopp}_{i:=i+2}$$

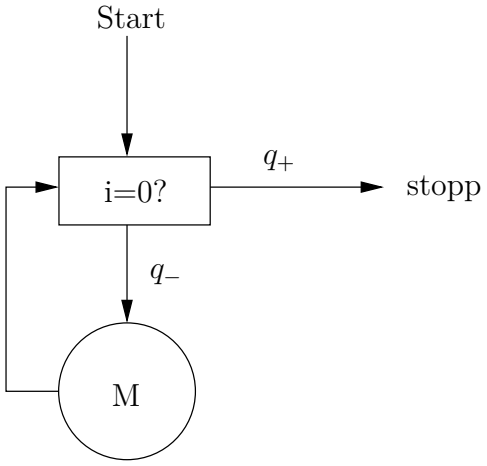
Beispiel 2.15 (Fallunterscheidung)

$$\delta = \delta_1 \cup \delta_2 \cup \{(q_+, a) \mapsto (q_{0_1}, a', 0)\} \cup \{(q_-, a) \mapsto (q_{0_2}, a', 0)\}$$

$$start \to M_0 \to \boxed{\text{i=0?}} \begin{cases} q_+ \to M_1 \to \ldots \to stopp \\ q_- \to M_2 \to \ldots \to stopp \end{cases}$$

$$M := M_0;\ \underline{\texttt{if}}\ i = 0\ \underline{\texttt{then}}\ M_1\ \underline{\texttt{else}}\ M_2$$

Beispiel 2.16 (while-Schleifen)



Maschine „$\underline{\texttt{while}}\ i \neq 0\ \underline{\texttt{do}}\ M$*“*

2.6 Rechenzeit und Speichersplatz

Definition 2.14
Es sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine Turing-Maschine und $w \in \Sigma^*$ eine Eingabe.

$$time_M(w) := \begin{cases} \text{Anzahl der Rechenschritte von } M \\ \text{bei der Eingabe } w & \text{falls } M \text{ bei } w \text{ stoppt} \\ \bot & \text{M stoppt nicht} \end{cases}$$

$$space_M(w) := \begin{cases} \text{Anzahl der vom Kopf besuchten Zel-} \\ \text{len des Arbeitsbandes} & \text{falls } M \text{ bei } w \text{ stoppt} \\ \bot & \text{M stoppt nicht} \end{cases}$$

Damit ist $time_M \colon \Sigma^* \to \mathbb{N}, space_M \colon \Sigma^* \to \mathbb{N}$.

$$TIME_M(n) := \begin{cases} \max\limits_{|w| \leq n} time_M(w) & time_M(w) \text{ definiert} \\ \bot & \text{Msonst} \end{cases}$$

$$SPACE_M(n) := \begin{cases} \max\limits_{|w| \leq n} space_M(w) & space_M(w) \text{ ist definiert} \\ \bot & \text{Msonst} \end{cases}$$

$TIME_M : \mathbb{N} \to \mathbb{N}$ ist die maximale Anzahl an Rechenschritten, die benötigt wird, um ein Wort ein Länge n zu berechnen. Analog ist $SPACE_M : \mathbb{N} \to \mathbb{N}$ der maximale Speicherplatz, der für die Berechnung eines Wortes der Länge n benötigt wird.

Definition 2.15
Es sei $t \colon \mathbb{N} \to_t \mathbb{N}$ und $s \colon \mathbb{N} \to_t \mathbb{N}$ zwei (total definierte) Zahlenfunktionen. Eine Funktion $f \colon \mathbb{N} \to \mathbb{N}$ ist **Turing-berechenbar mit der Zeitschranke** t und der **Speicherplatzbeschränkung** s, falls es eine Turing-Maschine M_t und eine Turing-Maschine M_s gibt, so dass $TIME_{M_t} \leq t$ und $SPACE_{M_s} \leq s$.

Es sei $L \subseteq \Sigma^*$ eine formale Sprache. L ist **Turing-erkennbar mit der Zeitbeschränkung** t und der **Speicherplatzbeschränkung** s *gdw* ein Erkenner M_t (Turing-Maschine ohne Ausgabe; s. Abschnitt 2.4.5) und ein Erkenner M_s existieren, die folgende Bedingungen erfüllen:

$$M_t(w \in L) = q_+, M_t(w \notin L) = q_-, TIME_{M_t} \leq t$$
$$M_s(w \in L) = q_+, M_s(w \notin L) = q_-, SPACE_{M_s} \leq s$$

Satz 2.2
Jede Sprache $L \subseteq \Sigma^*$, die von einer Mehrband-Turing-Maschine M mit der Zeitbeschränkung t und der Speicherplatzbeschränkung s enschieden werden kann, kann auch von einer 1-Band-Turing-Maschine M_1 mit der Zeitbeschränkung $O(s \cdot t)$ und der Platzbeschränkung $O(s)$ entschieden werden.

$$\mathcal{P} = \{L \colon \exists p, p \text{ ist Polynom}, TIME_M \leq p\}$$

$\mathcal{P}$ sind alle die Sprachen, die sich von einer Turing-Maschine in Polynomialzeit entscheiden lassen.

3 Partiell-rekursive Funktionen

Bisher haben wir zur Charakterisierung von Berechenbarkeit (Turing-)Maschinen, die Funktionen mit Hilfe eines Algorithmus' berechnen, betrachtet. Jetzt wollen wir einen algebraischen Ansatz betrachten, der auf einer Menge von Gundfunktionen, die (indutuitiv) berechenbar sind, und Schemata, die aus „einfachen" Funktionen neue „komplexe" Funktionen konstruieren, aufbaut.

Beispiel 3.1 (Modulo-Funktion)
$\mathrm{mod}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$\mathrm{mod}(a, b) =$ Rest bei der ganzzahligen Division von a und b
$\mathrm{mod}(a, 0) = \mathrm{mod}(0, b) = \mathrm{mod}(0, 0) = 0$

Für $b \neq 0$ lässt sich die Funktion rekursiv auf folgende Weise berechnen:

$$\mathrm{mod}(a, b) = \underline{\texttt{if}}\ a < b\ \underline{\texttt{then}}\ a\ \underline{\texttt{else}}\ \mathrm{mod}\ (a - b, b)$$

Vergleiche mit Definition 1.1

Das zugrundeliegende Schema ist folgendes:

- für eine endliche Menge $A \subseteq \mathbb{N}$ sind die Werte der Funktion bekannt (mittels anderer einfacherer Funktionen berechenbar)
- für den „großen" Rest werden die Funktionswerte durch Werte der Funktion an früheren Stellen B_n (mittels anderer einfacher Funktionen) bestimmt.

In unserem Beispiel 3.1 sind dies: $A = \{0, \ldots, b-1\}, B_n = \{n - b\}$.

Ein besonders einfacher Fall liegt vor, wenn $A = \{0\}$ und $B_n = \{n-1\}$.

3.1 Primitiv-rekursive Funktionen

Festlegung 3.1
Wir beschränken uns o. B. d. A. auf Zahlenfunktionen $f\colon \mathbb{N}^k \to \mathbb{N}$.

Bemerkung 3.1
Eine echte Einschränkung für diesen Abschnitt ist, dass wir nur total definierte Funktionen betrachten.

Definition 3.1 (Grundfunktionen)
Projektion

$$I_j^n(x_1, \ldots, x_n) = x_j \qquad ((x_1, \ldots, x_n) \in \mathbb{N}^n, n \geq 1, 1 \leq j \leq n)$$

Die Menge aller Projektionen $\{I_j^n : n > 0, 1 \leq j \leq n\}$

Konstanten

$$C_k^n(x_1, \ldots, x_n) = k \qquad ((x_1, \ldots, x_n) \in \mathbb{N}^n, n \geq 1, k \geq 0)$$

Die Menge aller Konstanten $\{C_k^n : n \geq 1, k \geq 0\}$

Nachfolgerfunktion

$$N(x) = x + 1 \qquad (x \in \mathbb{N})$$

$\mathcal{G}$...ist die Menge aller **Grundfunktionen**.

Bemerkung 3.2
Alle Grundfunktionen sind Turing-berechenbar!

Definition 3.2 (Einsetzungsprinzip)
Gegeben seien eine Funktion h: $\mathbb{N}^m \to \mathbb{N}$ und eine Menge von Funktionen g_i: $\mathbb{N}^n \to \mathbb{N}$ $(i = 1, \ldots, m)$. Die Funktion f: $\mathbb{N}^n \to \mathbb{N}$ geht aus den Funktionen h und g_i durch Einsetzung hervor gdw_{def}

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

Schreibweise: $f = SUB^m(h; g_1, \ldots, g_m)$

Es sei $F \subseteq \{f : \mathbb{N}^n \to \mathbb{N}, n \geq 0\}$ eine Menge von (Zahlen-)Funktionen. F heißt **abgeschlossen bezüglich des Schemas der Einsetzung** gdw_{def} jede Funktion, die durch Einsetzung von Funktionen aus F hervorgeht, gehört zu F.

Bemerkung 3.3
Die Klasse der Turing-berechenbaren Funktionen $\mathcal{TM}$ ist abgeschlossen bezüglich des Schemas der Einsetzung! Abbildung 3.1

Definition 3.3 (Schema der primitiven Rekursion)
Eine Funktion f: $\mathbb{N}^n \to \mathbb{N}$ geht aus den Funktionen g: $\mathbb{N}^{n-1} \to \mathbb{N}$ und h: $\mathbb{N}^{n+1} \to \mathbb{N}$ durch **primitive Rekursion** hervor gdw_{def}

$$\forall x_1, \ldots, x_{n-1}, y: f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1})$$
$$f(x_1, \ldots, x_{n-1}, y+1) = h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y))$$

Schreibweise: $f = PR(g, h)$

F heißt abgeschlossen bzgl. primitiver Rekursion gdw_{def} jede Funktion, die sich durch primitive Rekursion aus den Funktionen aus F ergibt, selbst zu F gehört.

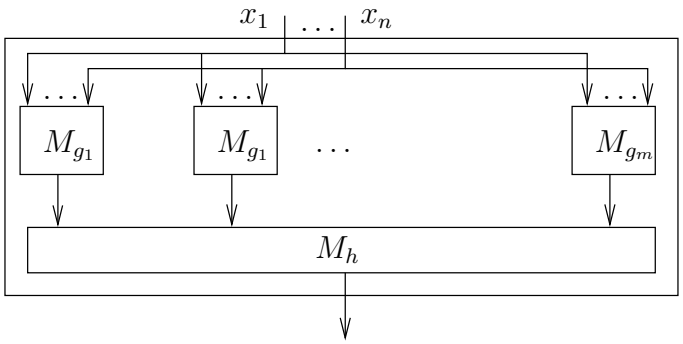


Abbildung 3.1: M_f als Komposition aus M_h und $M_{g_1}, \ldots, M_{g_m}$ zur Nachbildung des „Schemas der Einsetzung“

Bemerkung 3.4
Die Klasse der Turing-berechenbaren Funktionen $\mathcal{TM}$ ist abgeschlossen bzgl. primitiver Rekursion.

Beispiel 3.2
Die Addition $plus\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ geht aus den Grundfunktionen durch primitive Rekursion und Einsetzung hervor.

$$plus(x, 0) = x = I_1^1(x)$$
$$plus(x, y+1) = x + (y+1) = (x+y) + 1 = plus(x, y) + 1 = N\big(plus(x, y)\big)$$

Da h für PR eine dreistellige Funktion seien muss, wenden wir nochmal das Schema der Einsetzung an:

$$plus(x, y+1) = N\big(I_3^3(x, y, plus(x, y))\big)$$

$plus(x, y+1)$ lässt sich durch das Schema der Einsetzung $SUB^1(N; I_3^3)$ durch $h\colon \mathbb{N}^3 \to N$ ersetzen. Für die Funktion $plus$ gilt also: $plus = PR\big(I_1^1, SUB^1(N; I_3^3)\big)$.

Definition 3.4 (primitiv-rekursive Funktionen)
Die **Klasse $\mathbb{P}$r der primitiv-rekursiven Funktionen** ist die kleinste Klasse von Funktionen $\{f\colon \mathbb{N}^n \to \mathbb{N} | n \geq 0\}$ mit folgenden Eigenschaften:

1. Die Grundfunktionen $\mathcal{G}$ liegen in $\mathbb{P}$r. ($\mathcal{G} \subseteq \mathbb{P}$r)
2. $\mathbb{P}$r ist abgeschlossen bzgl. der Einsetzung.
3. $\mathbb{P}$r ist abgeschlossen bzgl. der primitiven Rekursion

Die Klasse der primitiv-rekursiven Funktionen lässt sich damit als algebraische Hülle schreiben:

$$\mathbb{P}\mathrm{r} = \Gamma_{\{SUB, PR\}}(G)$$

Dabei steht SUB für $SUB = \{SUB^m\colon m \in \mathbb{N}\}$

Bemerkung 3.5
Alle primitiv-rekursiven Funktionen sind Turing-berechenbar!

Frage: Sind alle Funktionen, die total definiert und berechenbar sind, primitiv-rekursiv?

3.2 Die Ackermann-Funktion

F. W. Ackermann definierte 1928 die folgende Funktion, die total definiert und berechenbar ist, aber von der gezeigt wurde, dass sie nicht primitiv-rekursiv ist!

Wir benutzen folgende Form: $A\colon \mathbb{N}^2 \to \mathbb{N}$

$$\begin{aligned} A(0,y) &:= y+1 \\ A(x+1,0) &:= A(x,1) \\ A(x+1,y+1) &:= A(x,A(x+1,y)) \end{aligned}$$

Bemerkung 3.6
Dieses Definitionsschema enthält eine doppelte Rekursion, die dazu führt, dass die Funktion *exorbitant* schnell wächst!

Beispiel 3.3

$$\begin{aligned} A(1,0) &= A(0,1) = 2 \\ A(1,1) &= A(0,A(1,0)) = A(0,2) = 3 \\ A(1,2) &= A(0,A(1,1)) = A(0,3) = 4 \\ \mathbf{A(1,y)} &\mathbf{= y+2} \\ A(1,y+1) &= A(0,A(1,y)) = A(0,y+2) = (y+2)+1 = (y+1)+2 \\ A(2,0) &= A(1,1) = 1+2 = 3 \\ A(2,1) &= A(1,A(2,0)) = A(1,3) = 3+2 = 5 \\ A(2,2) &= A(1,A(2,1)) = A(1,5) = 5+2 = 7 \\ \mathbf{A(2,y)} &\mathbf{= 2y+3} \\ A(2,y+1) &= A(1,A(2,y)) = A(1,2y+3) = 2y+3+2 = 2(y+1)+3 \\ A(3,0) &= A(2,1) = 2\cdot 1+3 = 5 \\ A(3,1) &= A(2,A(3,0)) = A(2,5) = 2\cdot 5+3 = 13 \\ A(3,2) &= A(2,A(3,1)) = A(2,13) = 2\cdot 13+3 = 29 \\ A(3,3) &= A(2,A(3,2)) = A(2,29) = 2\cdot 29+3 = 61 \\ \mathbf{A(3,y)} &\mathbf{= 2^{y+3}-3} \\ A(3,y+1) &= A(2,A(3,y)) = A(2,2^{y+3}-3) = 2\cdot(2^{y+3}-3)+3 = 2^{(y+1)+3}-3 \end{aligned}$$

$$A(4,0) = A(3,1) = 2^{1+3} - 3$$
$$A(4,1) = A(3, A(4,1)) = A(3, 2^{16} - 3) = 2^{2^{16}} - 3$$
$$A(4,2) = A(3, A(4,2)) = A(3, 2^{2^{16}} - 3) = 2^{2^{2^{16}}} - 3$$
$$\mathbf{A(4,y) = 2^{2^{\cdots^{2}}}\}y\text{-mal} - 3}$$
$$A(5,0) = A(4,1) = 2^{16} - 3$$
$$A(5,1) = A(4, A(5,0)) = A(4, 2^{16} - 3)$$

Satz 3.1
Die Ackermannfunktion ist (intuitiv) berechenbar, aber nicht primitiv-rekursiv.

Beweis:
Die Idee des Beweises ist folgende Beobachtung: Setzt man die Definition der Ackermann-Funktion wiederholt in die allgemeine Form ein (Gleichung 3.1), so sieht man, dass die Anzahl der ineinander geschachtelten Rekursionen von den Parametern x und y abhängt. Mit Primitiver-Rekursion kann man jedoch nur einfache Rekursion (mit konstanter Schachtelungstiefe) erreichen.

$$A(x+1, y+1) = A(x, A(x+1, y)) = A(x, A(x, A(x+1, y-1))) = \tag{3.1}$$
$$A(x, A(x, A(xA(x+1, y-2)))) \tag{3.2}$$

■

Haben $\mathbb{P}\mathrm{r} = \Gamma_{\{SUB,PR\}}(G)$, d. h. jede primitiv-rekursive Funktion lässt sich darstellen aus den Grundfunktionen und endlich oft nötiger Anwendung des Schemas der Einsetzung und des Schemas der primitiven Rekursion.

Für jede konkrete Funktion ergibt sich eine konkrete Darstellung mit einer festen Anzahl von Einsetzungen und primitiven Rekursionen.

Bemerkung 3.7
Jede primitiv-rekursive Funktion lässt sich durch einen Ausdruck endlicher Länge über einem gegebenen Alphabet beschreiben:

$$\Sigma = \{I, C, N, |, \#,), , , (, x, SUB, PR\}$$

Beispiel 3.4
Hatten: $plus = PR(I_1^1, SUB^1(N, I_3^3))$ es ergibt sich:

$$plus = PR(I\#|\#|\#, SUB\#|\#(N, I\#|||\#|||\#))$$

Dies ist ein Wort über dem Alphabet Σ.

Konsequenz:
Eine solche Darstellung zeigt:

1. $\mathbb{P}\mathrm{r}$ ist abzählbar unendlich
2. $\mathbb{P}\mathrm{r}$ ist effektiv nummerierbar (durch systematisches nummerieren der Wörter über Σ)

Dies hat folgende weitere Konsequenzen:
Es sei $\phi_0, \phi_1, \phi_2, \ldots$ eine Nummerierung aller einstelligen primitiv-rekursiven Funktionen. Wir definieren durch *Diagonalisierung* eine Funktion ψ

$$\psi(n) = \phi_n(n) + 1 \qquad (n \in \mathbb{N})$$

Es ist leicht einzusehen, dass ψ zwar (intuitiv) berechenbar, aber nicht primitiv-rekursiv ist.

Ann.: $\psi \in \mathbb{P}\mathrm{r}$ Dann müsste ψ zu ϕ_n gehören und es existiert ein $k \in \mathbb{N}$ mit $\psi = \phi_k$. Dies führt aber zu einem Widerspruch: $\phi_k(k) = \psi(k) = \phi_k(k) + 1$.

3.3 μ-rekursive Funktionen

Programmiersprachen mit Schleifen:

`while` *Bedingung* `do` *Anweisung* `end`

wollen solche Schleifen simulieren

Definition 3.5
Es sei $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ eine Funktion mit $n > 0$. Die Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$ entsteht aus g durch das Schema der **μ-Rekursion** gdw_{def} für alle $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt:

$$f(x_1, \ldots, x_n) = \begin{cases} \min\{t \in \mathbb{N}\colon g(x_1, \ldots, x_n, t) = 0\} & \text{falls ein solches } t \text{ existiert und } \forall 0 \le y \le t\ g(x_1, \ldots, x_n, y) \text{ definiert ist} \\ \bot & \text{sonst} \end{cases}$$

Schreibweise: $f = \mu R(g)$

Damit gilt: Wenn g berechenbar ist, dann ist auch f berechenbar.

als Schleife:
`t := 0`
`while` $g(x_1, \ldots, x_n, t) \neq 0$ `do t := t+1 end`

Definition 3.6
1. Eine Klasse $\mathbb{P}$ von Funktionen heißt abgeschlossen bzgl. μ-Rekursion gdw_{def} für alle $g \in \mathbb{P}$ ist $f = \mu R(g) \in \mathbb{P}$.
2. Die **Klasse $\mathbb{P}$ der partiell-rekursiven Funktionen** ist die kleinste Klasse, die $\mathbb{P}\mathrm{r}$ umfasst und abgeschlossen bzgl. μ-Rekursion ist. Anders dargestellt: $\mathbb{P} = \Gamma_{\{\mu R\}}(\mathbb{P}\mathrm{r})$

Äquivalent hierzu ist die folgende Definition:

Definition 3.7
Die Klasse $\mathbb{P}$ ist die kleinste Klasse, die $\mathcal{G}$ umfasst und die abgeschlossen bzgl. Einsetzung, primitiver-Rekursion und μ-Rekursion ist. Anders dargestellt: $\mathbb{P} = \Gamma_{\{SUB,PR,\mu R\}}(\mathcal{G})$

Bemerkung 3.8
Dabei werden das Schema der Einsetzung und das Schema der primitiven Rekurision für partiell-definierte Funktionen erweitert.

Beispiel 3.5

$$d(m,n) = \begin{cases} \frac{n}{m} & \text{falls Teiler existiert} \\ \perp & \text{sonst} \end{cases}$$

Zunächst zeigen wir, dass die Signumfunktion, Potenzfunktion und die Betragsfunktion primitiv-rekursiv sind.

$$\begin{aligned} sgn\colon \mathbb{N} \to \mathbb{N}; n \mapsto \begin{cases} 0 & : n = 0 \\ 1 & : n > 0 \end{cases} \\ exp\colon \mathbb{N}^2 \to \mathbb{N}; (m,n) \mapsto m^n; (0,0) \mapsto 1 \\ ab\colon \mathbb{N}^2 \to \mathbb{N}; (m,n) \mapsto |m-n| \end{aligned}$$

$$\begin{aligned} sgn(0) &= 0 = C_0^0 \\ sgn(n+1) &= 1 = h(n, sgn(n)) = C_1^2(n, sgn(n)) \\ sgn &= PR(C_0^0, C_1^2) \end{aligned}$$

Die Darstellung ist jedoch nicht eindeutig und so wäre auch folgende möglich:

$$\begin{aligned} sgn(0) &= I_1^1(C_0^0) \\ sgn(n+1) &= C_1^1(I_1^2(n,n)) \\ sgn &= PR(SUB^1(I_1^1, C_0^0), SUB^1(C_1^1, I_1^2)) \end{aligned}$$

Die Abstandsfunktion lässt sich aus anderen primitiv-rekursive Funktionen zusammensetzen und ist somit selbst primitiv-rekursiv

$$ab(x,y) = |x - (y+1)| = sgn(a-b) \cdot (a-b) + sgn(b-a) \cdot (b-a)$$

Damit ist $f\colon \mathbb{N}^3 \to \mathbb{N}$ primitiv-rekursiv

$$f(k,m,n) = |k \cdot m - n|^{sgn(m)}$$

setzen f in $\mu R(f)$ ein:

$$\begin{aligned}
[\mu R(f)](m,n) &= \begin{cases} \mu k(|k \cdot m - n| = 0) & : \text{falls ein solches } k \text{ existiert und } m \neq 0 \\ \bot & : \text{sonst} \end{cases} \\
&= \begin{cases} \mu k(k \cdot m = n) & : \text{falls ein solches } k \text{ existiert} \\ \bot & : \text{sonst} \end{cases} \\
&= \begin{cases} \frac{n}{m} & : \text{falls } m \text{ ein Teiler von } n \text{ ist, } m \neq 0 \\ \bot & : \text{sonst} \end{cases} \\
&= d(m,n)
\end{aligned}$$

Beispiel 3.6
Die Funktion $wurz\colon \mathbb{N} \to \mathbb{N}$ ist partiell-rekursiv

$$wurz(n) := \begin{cases} \sqrt{n} & : \sqrt{n} \in \mathbb{N} \\ \bot & : \text{sonst} \end{cases}$$

Wir betrachten folgende zweistellige Funktion h

$$h(m,n) = |m - n^2| \qquad\qquad h \in \mathbb{P}\mathrm{r}$$

Wir bestimmen $\mu R(h)$:

$$\begin{aligned}
[\mu R(h)](m) &= \begin{cases} \mu n(|m - n^2| = 0) & : \exists n \\ \bot & : \text{sonst} \end{cases} \\
&= \begin{cases} \sqrt{m} & : \sqrt{m} \in \mathbb{N} \\ \bot & : \text{sonst} \end{cases} \\
&= wurz(m)
\end{aligned}$$

Also ist $wurz = \mu R(h)$

Beispiel 3.7
Die Funktion $ganzwurz\colon \mathbb{N} \to \mathbb{N}$ ist partiell-rekursiv

$$ganzwurz(m) = gw(m) = \lfloor \sqrt{m} \rfloor \qquad\qquad (m \in \mathbb{N})$$

Es gilt:

$$\begin{aligned}
gw(m) = n &\Leftrightarrow n \leq \sqrt{m} < n+1 \\
&\Leftrightarrow n^2 \leq m < (n+1)^2
\end{aligned}$$

Also gilt:

$$\begin{aligned} gw(m) &= \mu n(m < (n+1)^2) \\ &= \mu n(|(m+1) - (n+1)^2| = 0) \end{aligned}$$

Weiter gilt: $gw(m) = wurz(m)$ falls $wurz(m)$ definiert ist und gw ist total definiert. Damit ist gw eine **Fortsetung** von $wurz$ und gw ist partiell-rekursiv (sogar primitiv-rekursiv)

Satz 3.2
Jede partiell-rekursive Funktion lässt sich darstellen als ein Term, der die Symbole für die Grundfunktion, die Schemata $SUB, PR, \mu R$, sowie weitere notwendige Hilfssymbole enthält.

Vergleiche das Alphabet Σ für primitiv-rekursive Funktionen in Bemerkung 3.7 und füge „μR“ als neues Symbol hinzu!

Die kanonische (quasilexikographische) Auflistung aller dieser Terme liefert eine *effektive Nummerierung* aller partiell-rekursiven Funktionen. $\mathbb{P}$ ist also aufzählbar und abzählbar unendlich.

3.4 Allgemein-rekursive Funktionen

Die Funktion $wurz \in \mathbb{P}$ aus Beispiel 3.6 und ist partiell (definiert). Die Funktion $gw \in \mathbb{P}$ aus Beispiel 3.7, total (definiert). Die Fortsetzung von $wurz$ ist gw: $gw|_{Dwurz} = wurz$

Definition 3.8
Eine partiell-rekursive Funktion $f \in \mathbb{P}$ heißt **allgemein-rekursiv** gdw_{def} f total definiert ist.

$\mathbb{P}$a bezeichnet die **Klasse aller allgemein-rekursiven Funktionen**.

Wenn $\mathbb{F}$ die Menge aller Zahlenfunktionen bezeichnet, dann gilt:

$$\emptyset \subseteq \mathbb{P}\text{r} \subseteq \mathbb{P}\text{a} \subseteq \mathbb{P} \subseteq \mathbb{F}$$

Satz 3.3
Alle Inklusionen sind echt:

$$\begin{array}{ccccccccc} \emptyset & \subsetneq & \mathbb{P}\text{r} & \subsetneq & \mathbb{P}\text{a} & \subsetneq & \mathbb{P} & \subsetneq & \mathbb{F} \\ & 1. & & 2. & & 3. & & 4. & \end{array}$$

Beweis:

1. z. B. ist $nachfolger \in \mathbb{P}\text{r}$ — Definition 3.1
2. Ackermann-Funktion — Abschnitt 3.2

3. $wurz \in \mathbb{P} \setminus \mathbb{P}\mathrm{a}$ — Beispiel 3.6
4. Es gibt Funktionen aus $\mathbb{F} \setminus \mathbb{P}$, die nicht berechenbar sind:

   Wir wissen aus Satz 3.2, dass $\mathbb{P}$ eine effektive Numerierung besitzt, d. h. es gibt höchstens abzählbar unendlich viele partiell-rekursive Funktionen. Aber es gibt überabzählbar viele Zahlenfunktionen, z. B. ist bereits $\{0,1\}^{\mathbb{N}}$ überabzählbar. ■

Bemerkung 3.9
Da nicht jede partiell-rekursive Funktion partiell ist, ist μ-rekursive Funktion eine alternative Bezeichnund.

Da nicht jede primitiv-rekursive Funktion primitiv ist, ist induktiv-rekursive Funktion eine alternative Bezeichnung.

Dennoch bleibt die Frage offen, ob *jede* partiell-rekursive Funktion zu einer allgemein-rekursiven Funktion fortgesetzt werden kann.

Dazu betrachten wir die folgende Diagonalisierung der partiell-rekursiven Funktionen: $\phi_0, \phi_1, \phi_2, \ldots$ ist eine Nummerierung der partiell-rekursiven Funktionen, die sich aus einer Auflistung aller zugehörigen Terme ergibt. Wir definieren nun die folgende Funktion

$$\psi(n) := \phi_n(n) + 1 = \begin{cases} \phi_n(n) + 1 & : \text{falls } \phi_n(n) \text{ definiert ist} \\ \bot & : \text{sonst} \end{cases}$$

ψ ist eine berechenbare Funktion, d. h. ψ kommt in der Liste ϕ_n vor. Nehmen wir an ψ hat den Index k, also $\psi = \phi_k$. Dies führt uns jedoch zu dem Problem:

$$\phi_k(k) = \psi(k) = \phi_k(k) + 1 = \begin{cases} \phi_k(k) + 1 & : \text{falls } \phi_k(k) \text{ definiert ist} \\ \bot & : \text{sonst} \end{cases}$$

als dessen Konsequenz nur gelten kann, dass ψ an der Stelle k nicht definiert seien kann. $\psi(n) \in \mathbb{P}$ ist also an min. einer Stelle k nicht definiert und somit nicht allgemein-rekursiv ($\notin \mathbb{P}\mathrm{a}$). help: Damit ist doch keine Aussage über die Fortsetzung gemacht

Die Eigenschaft partiell-rekursiv sein zu können, ist unverzichtbar für die Definition von Berechenbarkeit. Dies führt zu einer negativen Antwort unserer Frage.

Es sei ϕ_l eine Nummer irgendeiner allgemein-rekursiven Funktion. Dann gilt: $\psi(l)$ ist definiert und hat den Wert $\psi(l) = \phi_l(l) + 1$ und es gilt $\psi(l) \neq \phi_l(l)$. Also gilt $\psi \neq \phi_l$ und ϕ_l ist *keine* Fortsetzung von ψ.

Fakt
Nicht jede partiell-rekursive Funktion kann zu einer allgemein-rekursiven Funktion fortgesetzt werden!

Satz 3.4
Die Menge $\mathbb{P}\mathrm{a}$ der allgemein-rekursiven Funktionen besitzt keine effektive Nummerierung!

Beweis:
(durch Diagonalisierung) Annahme: es existiert eine Liste $f_0, f_1, f_2, \ldots$ über allgemein-rekursive Funktionen. Dann wäre auch die total berechenbare Funktion $g(n) = f(n) + 1$ in der Liste enthalten und es gäbe einen Index m mit $g = f_m$. Dies führt jedoch zu einem Widerspruch (Gleichung 3.3), als dessen Konsequenz nur gelten kann, dass es eine solche Liste – eine effektive Nummerierung – nicht gibt.

$$f_m(m) = g(m) = f_m(m) + 1 \tag{3.3}$$

■

Frage: Kann man eine nicht berechenbare Funktion aus $\mathbb{F}\backslash\mathbb{P}$ angeben?

Erinnern wir uns zurück: Eine Sprache $L \subseteq \Sigma^*$ heißt entscheidbar *gdw* die charakteristische Funktion χ_L berechenbar ist. (Definition 2.8) Die Menge $\mathbb{P}\mathrm{a}$ der allgemein-rekursiven Funktionen ist nicht aufzählbar und damit ist nach Abschnitt 2.4.4 die Menge nicht entscheidbar. Die charakteristisch Funktion der allgemein-rekursiven Funktionen ist also nicht berechenbar

(3.4)

$$\chi_{\mathbb{P}\mathrm{a}} \colon \mathbb{N} \to \mathbb{N} \qquad \chi_{\mathbb{P}\mathrm{a}}(n) = \begin{cases} 1 \colon \text{die Turing-Maschine } TM_n \text{ aus } \mathbb{P} \text{ ist total } (\in \mathbb{P}\mathrm{a}) \\ 0 \colon \text{sonst} \end{cases}$$

Gibt es ein anschauliches Beispiel einer partiell-rekursiven (d. h. berechenbaren) Funktion, von der es nicht offensichtlich ist, dass sie selbst oder eine ihrer Fortsetzungen nicht primitv-rekursiv ist?

> Für jede natürliche Zahl $n \geq 1$ führt die wiederholte Anwendung folgender Regel nach endlich vielen Schritten auf die Zahl 1:
>
> $$n \mapsto \begin{cases} \frac{n}{2} & n \text{ ist gerade} \\ 3n + 1 & n \text{ ist ungerade} \end{cases}$$

Ein Beispiel hierzu:

$$
\begin{aligned}
&1\\
&2 \mapsto 1\\
&3 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1\\
&4 \mapsto 2 \mapsto 1\\
&5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1\\
&6 \mapsto 3 \mapsto \dots\\
&7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto \dots\\
&8 \mapsto 4 \mapsto \dots\\
&9 \mapsto 28 \mapsto 14 \mapsto 7 \mapsto \dots\\
&10 \mapsto 5 \mapsto \dots
\end{aligned}
$$

Wir fassen alle Zahlen, für die diese Vermutung zutrifft als Menge $B = \{n \in \mathbb{N}_+ :$ Vermutung trifft für n zusammen. Nun ist die Frage: Ist $B = \mathbb{N}_+$?

Die einmalige Anwendung der Regel entspricht folgender Funktion

$$f(n) = \begin{cases} \frac{n}{2} & n \text{ ist gerade} \\ 3n+1 & n \text{ ist ungerade} \end{cases}$$

f ist definiert durch primitiv-rekursive Funktionen mittels Fallunterscheidung! Deshalb ist auch f primitiv-rekursiv.

Die t-malige Anwendung der Regel wird durch die folgende Funktion beschrieben

$$
\begin{aligned}
g(n,0) &= n && I_1^1(n)\\
g(n,t+1) &= f(g(n,t)) = f'(n,t,g(n,t)) && SUB(f', I_3^3)
\end{aligned}
$$

g ist also eine primitv-rekursive Funktion mit $PR(I_1^1, SUB(f', I_3^3))$.

Wir definieren nun

$$h(n) = \begin{cases} \mu t(g(n,t) = 1) & : \text{ein solches } t \text{ existiert} \\ \perp & : \text{sonst} \end{cases}$$

h ist partiell-rekursiv und für den Definitionsbereich gilt: $D_n = B$. Aber es gibt keine Beschreibung von h (oder einer Fortsetzung) als primitv-rekursive Funktion help: Woran sieht man das?

$$g(n,t) = 1 \ \ gdw \ \ g(n,t) - 1 = 0$$

3.4.1 Eigenschaften primitiv-rekursiver Funktionen

Satz 3.5 (stückweise definierte Funktionen)
Es seien folgende Funktionen primitiv-rekursiv

$$\begin{aligned} f_1, \ldots, f_k, f_{k+1} &\colon \mathbb{N}^n \to \mathbb{N} \\ \text{und } g_1, \ldots, g_k &\colon \mathbb{N}^n \to \mathbb{N} \end{aligned}$$

und es sei die folgende Bedingung erfüllt: Es gibt *keine* Tupel $(x_1, \ldots, x_n) \in \mathbb{N}^n$ mit der Eigenschaft, dass für jedes $j \neq i$ $g_i(x_1, \ldots, x_n) = 0$ *und* $g_j(x_1, \ldots, x_n) = 0$ (d. h. für jedes Tupel ist höchstens eine Funktion $g_i = 0$).

Dann ist die Funktion $h \colon \mathbb{N}^n \to \mathbb{N}$ mit

$$h(x_1, \ldots, x_n) = \begin{cases} f_1(x_1, \ldots, x_n) & : g_1(x_1, \ldots, x_n) = 0 \\ f_2(x_1, \ldots, x_n) & : g_2(x_1, \ldots, x_n) = 0 \\ \vdots & \\ f_k(x_1, \ldots, x_n) & : g_k(x_1, \ldots, x_n) = 0 \\ f_{k+1}(x_1, \ldots, x_n) & : \text{sonst} \end{cases}$$

primitiv-rekursiv.

Satz 3.6 (obere Schranken)
Es seien $g \colon \mathbb{N}^{n+1} \to_t \mathbb{N}$ und $h \colon \mathbb{N}^n \to \mathbb{N}$ primitiv-rekursiv. Ferner sei $f = \mu R(g)$ die durch μ-Rekursion aus g entstehende Funktion mit der Eigenschaft, dass für alle $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt:

$$f(x_1, \ldots, x_n) \leq h(x_1, \ldots, x_n)$$

(d. h. f ist für alle Tupel $(x_1, \ldots, x_n)$ definiert und es gilt die Ungleichung).

Dann ist f primitiv-rekursiv!

Satz 3.7 (Simultane Rekursion)
Es seien $g_1, g_2 \colon \mathbb{N}^{n-1} \to_t \mathbb{N}$ und $h_1, h_2 \colon \mathbb{N}^{n+2} \to_t \mathbb{N}$ primitiv-rekursiv.

Dann sind auch die Funktionen $f_1, f_2 \colon \mathbb{N}^n \to \mathbb{N}$ mit

$$\begin{aligned} f_1(x_1, \ldots, x_{n-1}, 0) &= g_1(x_1, \ldots, x_n) \\ f_2(x_1, \ldots, x_{n-1}, 0) &= g_2(x_1, \ldots, x_n) \\ \text{und } f_1(x_1, \ldots, x_{n-1}, y+1) &= h_1(x_1, \ldots, x_{n-1}, f_1(x_1, \ldots, x_{n-1}, y), f_2(x_1, \ldots, x_{n-1}, y)) \\ f_2(x_1, \ldots, x_{n-1}, y+1) &= h_2(x_1, \ldots, x_{n-1}, f_1(x_1, \ldots, x_{n-1}, y), f_2(x_1, \ldots, x_{n-1}, y)) \end{aligned}$$

primitv-rekursiv.

Satz 3.8 (Wertverlaufsfunktionen)
Es sei $A = \{0, 1, \ldots, l\}$ und ferner sei $B_n \subseteq \{0, \ldots, n-1\}$ mit der Eigenschaft $|B_n| = k$. Ferner sei $g_0, \ldots, g_l \colon \mathbb{N}^{n-1} \to_t \mathbb{N}$ und $h \colon \mathbb{N}^{n+k} \to_t \mathbb{N}$ primitv-rekursiv.

Dann ist auch die Funktion $f \colon \mathbb{N}^n \to_t \mathbb{N}$ mit

$$\begin{aligned} f(x_1, \ldots, x_{n-1}, 0) &= g_0(x_1, \ldots, x_{n-1}) \\ &\vdots \\ f(x_1, \ldots, x_{n-1}, l) &= g_l(x_1, \ldots, x_{n-1}) \\ f(x_1, \ldots, x_{n-1}, y+1) &= h(x_1, \ldots, x_{n-1}, y, z_1, \ldots, z_k) \\ z_1 &= f(x_1, \ldots, x_{n-1}, y_1) \\ &\vdots \\ z_k &= f(x_1, \ldots, x_{n-1}, y_k) \\ B_{y+1} &= \{y_1, \ldots, y_k\} \end{aligned}$$

primitiv-rekursiv.

Beispiel 3.8 (Fibonacci-Zahlen)

$$\begin{aligned} F(0) &= F(1) = 1 \\ F(n+1) &= F(n) + F(n-1) \qquad\qquad n \geq 2 \end{aligned}$$

$$\begin{aligned} A &= \{0, 1\}, B_n = \{n-2, n-1 \colon n \geq 2\} \\ F(0) &= F(1) = C_1^0 \\ F(n+1) &= h(n, F(n), F(n-1)) \end{aligned}$$

Für h ist noch zu zeigen, dass h primitiv-rekursiv ist. Der Beweis ist analog dem für *plus* in Beispiel 3.2:

$$\begin{aligned} h(a, b, c) &= b + c \\ h(a, b, 0) &= b \\ h(a, b, c+1) &= N\big(I_4^4(a, b, c, h(a, b, c))\big) \end{aligned}$$

F ist primitiv-rekursiv!

3.5 Partiell-rekursive und Turing-berechenbare Funktionen

Satz 3.9
Jede partiell-rekursive Funktion ist Turing-berechenbar, d. h. $\mathbb{P} \subseteq \mathcal{TM}$.

BEWEIS:
$\mathbb{P}$ ist laut Definition 3.6 die kleinste Klasse, die die Grundfunktionen umfasst und abgeschlossen bzgl. SUB, PR und μR ist. Zeigen wir also, dass diese auch Turing-berechenbar sind:

1. Nach Bemerkung 3.2 ist jede Grundfunktion Turing-berechenbar: $\mathcal{G} \subseteq \mathcal{TM}$.
2. Nach Bemerkung 3.3 ist $\mathcal{TM}$ abgeschlossen bzgl. der Einsetzung: falls $g_1, \ldots, g_m \colon \mathbb{N}^n \to_t \mathbb{N}$ und $h \colon \mathbb{N}^m \to_t \mathbb{N}$ Turing-berechenbar sind, dann ist auch $f \colon \mathbb{N}^n \to_t \mathbb{N}$ mit $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ Turing-berechenbar!
3. Nach Bemerkung 3.4 ist $\mathcal{TM}$ abgeschlossen bzgl. Primitiver-Rekursion, d. h. falls $g \colon \mathbb{N}^{n-1} \to_t \mathbb{N}$ und $h \colon \mathbb{N}^{n+1} \to_t \mathbb{N}$ Turing-berechenbar sind, dann ist auch $f \colon \mathbb{N}^n \to \mathbb{N}$ mit

$$\begin{aligned} f(x_1, \ldots, x_{n-1}, 0) &= g(x_1, \ldots, x_{n-1}) \\ f(x_1, \ldots, x_{n-1}, y+1) &= h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y)) \end{aligned}$$

   Turing-berechenbar.

   Es sei M_g (bzw. M_h) eine Turing-Maschine, die die Funktion g (bzw. h) berechnet. $M_g(x_1, \ldots, x_{n-1})$ bezeichnet im Folgenden das Resultat der Berechnung von M_g bei Eingabe $(x_1, \ldots, x_{n-1})$.

   Wir beschreiben folgende Turing-Maschine für die Eingabe $(x_1, \ldots, x_{n-1}, y)$:
   begin (Eingabe: $x_1, \ldots, x_{n-1}, y$)

   a) $t := 0$

   b) $z := M_g(x_1, \ldots, x_{n-1})$

   c) while $t \neq y$ do

   d) $\quad z := M_h(x_1, \ldots, x_{n-1}, t, z)$

   e) $\quad t := t + 1$

   f) end

   g) return z

   end

   Die so beschriebene Turing-Maschine berechnet f!
4. $\mathcal{TM}$ist abgeschlossen bzgl. μ-Rekursion, d. h. falls $g \colon \mathbb{N}^{n+1} \to \mathbb{N}$ Turing-berechenbar ist, dann ist auch $f \colon \mathbb{N}^n \to \mathbb{N}$ Turing-berechenbar, wobei

$$f(x_1, \ldots, x_n) = \mu t(g(x_1, \ldots, x_n) = 0)$$

   begin (Eingabe: $x_1, \ldots, x_n$)

a) $t := 0$

b) $z := M_g(x_1, \ldots, x_n, t)$

c) <u>while</u> $z \neq 0$ <u>do</u>

d) $\quad t := t + 1$

e) $\quad z := M_g(x_1, \ldots, x_n, t)$

f) <u>end</u>

g) <u>return</u> t

<u>end</u> ∎

Satz 3.10
Jede Turing-berechenbare Funktion ist partiell-rekursiv. $\mathcal{TM} \subseteq \mathbb{P}$

BEWEIS:
Die Idee dabei ist, die Arbeitsweise von Turing-Maschinen durch primitiv- bzw. partiell-rekursive Funktionen „zu beschreiben“.

Es sei $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine Turing-Maschine.

Das Arbeitsalphabet $\Gamma = \{\square, a_1, a_2, \ldots, a_{m-1}\}$ wird interpretiert (indetifiziert) mit der Zahlenmenge $\{0, 1, 2, \ldots, m-1\}$ des m-nären Systems.

<u>Schreibweise:</u> $Konf_M(w, t)$ bezeichnet die Konfiguration (Bandinschrift, Kopfposition, Zustand) von M bei Eingabe w nach dem Takt t und wird beschrieben durch

$$\underbrace{y_l \ldots y_2 y_1}_{\text{linker Teil}} \quad \underbrace{x_0 q}_{\text{Zustand}} \quad \underbrace{x_0 x_1 \ldots x_k}_{\text{rechter Teil}}$$

Die Berechnung von M bei Eingabe w wird vollständig durch die Folge von Konfigurationen beschrieben.

$$Konf_M(w, 0) \vdash Konf_M(w, 1) \vdash Konf_M(w, 2) \vdash \ldots \vdash Konf_M(w, t)$$

Dabei ist $K_0 = Start_M(w)$ die Startkonfiguration, K_t eine Finalkonfiguration und $K_i \vdash K_{i+1}$ gemäß δ.

Wir definieren:

$$Z(w, t) = q$$

$$R(w, t) = \sum_0^k x_i m^i$$

$$L(w, t) = \sum_0^l y_i m^i$$

Damit ist die Konfiguration $Konf_M(w,t)$ mit Hilfe der drei Funktionen Z, R und L vollständig beschrieben. Die Beschreibung der Bandinschrift ist auf diese Weise eindeutig!

Mithilfe von simultaner Rekursion zeigen wir jetzt noch, dass die definierten Funktionen (sogar) primitiv-rekursiv sind.

Anfang:

$$\begin{aligned} Z(w,0) &= q_0 \\ R(w,0) &= \sum_0^n w_i m^i \qquad\qquad w = w_1 w_2 \dots w_n \\ L(w,0) &= 0 \end{aligned}$$

Schritt an der Stelle $t \to t+1$:
es sei $Z(w,t) = q$ der aktuelle Zustand und das aktuelle Symbol $x = R(w,t) \bmod m$.
Ferner sei

$$\delta(q,x) = q'x' \begin{cases} \text{nach rechts} \\ \text{stehen bleiben} \\ \text{nach links} \end{cases}$$

Damit lässt sich ein Schritt der Überführungsfunktion auf die folgende Weise nachbilden:

$$Z(w,t+1) = q'$$

$$R(w,t+1) = \begin{cases} R(w,t) \div m & : \text{nach rechts} \\ R(w,t) - x + x' & : \text{stehenbleiben} \\ (R(w,t) - x + x')m + y_0 & : \text{nach links} \end{cases}$$

$$L(w,t+1) = \begin{cases} L(w,t)m + x' & : \text{nach rechts} \\ L(w,t) & : \text{stehenbleiben} \\ L(w,t) \div m & : \text{nach links} \end{cases}$$

Wir definieren:

$$stopp_M(w) := \mu t(Z(w,t) \in F)$$

1. Fall Ein solches t existiert nicht, d. h. M stoppt bei Eingabe w nicht.

   Dann ist $stopp_M(w) = \perp$.

2. Fall Ein t existiert, d. h. M stoppt und $stopp_M(w)$ ist definiert.

Die Funktion $stopp_M$ ist partiell-rekursiv.

Wir definieren weiterhin:

$$Res_M(w) = R(w, stopp(w))$$

Damit gilt: Das Ergebnis der Berechnung von M bei Eingabe w ist $Res_M(w)$ *gdw* ein Ergebnis existiert.

Damit haben wir gezeigt, dass jede Turing-berechenbare Funktion f eine Darstellung der folgenden Form mit den primitiv-rekursiven Funktionen Z, L und R und einer μ-Rekursion besitzt:

$$f(w) = Res_M(w) = R(w, \mu t(Z(w,t) \in F))$$ ∎

Bemerkung 3.10
Der Beweis hängt wesentlich davon ab, dass $\div$ und mod primitiv-rekursiv sind.

Dazu die nun folgende Herleitung.

Es seien $f\colon \mathbb{N}^{n+1} \to_t \mathbb{N}$ und $g\colon \mathbb{N}^n \to_t \mathbb{N}$ primitiv-rekusiv. Ferner gelte

$$\forall(x_1, \ldots, x_n) \in \mathbb{N}^n \exists y \leq q(x_1, \ldots, x_n) \qquad (f(x_1, \ldots, x_n) = 0)$$

Dann ist eine Funktion $h\colon \mathbb{N}^n \to \mathbb{N}$ primitiv-rekursiv, wobei

$$h(x_1, \ldots, x_n) = \mu y(f(x_1, \ldots, x_n, y) = 0)$$

Beweis:
Hilfsfunktion $h'\colon \mathbb{N}^{n+1} \to \mathbb{N}$

$$\begin{aligned} h'(x_1, \ldots, x_n, z) &= \mu y(y \leq z \wedge f(x_0, \ldots, x_n, y) = 0) \\ &:= \sum_{i=0}^{z} sgn(\prod_{j=0}^{i} f(x_1, \ldots, x_n, j)) \end{aligned}$$

h' ist primitiv-rekursiv und es gilt:

$$h(x_1, \ldots, x_n) = h'(x_1, \ldots, x_n, g(x_1, \ldots, x_n))$$

Also ist auch h primitiv-rekursiv! ∎

Anwendung:

$$x \div y = \mu z((x+1) - (z+1)y = 0)$$

Ausnahme: $x \div 0 := 0$

x	y	z
11	4	2
12	4	3
13	4	3

und damit

$$x \bmod y = x - y(x \div y)$$

3.5.1 Eigenschaften partiell-rekursiver Funktionen

Satz 3.11 (Kleene'scher Normalformsatz)
Jede (n-stellige) partiell-rekursive Funktion f besitzt die Darstellung der Form

$$f(x_1, \ldots, x_n) = g(\mu y(h(x_1, \ldots, x_n, y) = 0))$$

mit primitiv-rekursiven Funktionen $g\colon \mathbb{N} \to_t \mathbb{N}$ und $h\colon \mathbb{N}^{n+1} \to_t \mathbb{N}$

Satz 3.12 (Parametrisierungssatz)
Jede (n-stellige) partiell-rekursive Funktion f besitzt eine Darstellung der Form

$$f = \{(g_1(t), g_2(t), \ldots, g_n(t), g_{n+1}(t)) : t \in \mathbb{N}\}$$

mit (einstelligen) primitiv-rekursiven Funktionen $g_1, \ldots, g_{n+1}\colon \mathbb{N} \to_t \mathbb{N}$ oder f ist nirgends definitert.

4 Formale Sprachen und formale Grammatiken

Unverzichtbare Bestandteile höherer Programmiersprachen sind:

- Wertzuweisung
- Bedingungen testen
- Fallunterscheidung
- Schleifen

Rein von der Intuition her, können all diese Bestandteile (von Programmen) durch spezielle Turing-Maschinen realisiert werden. Das bedeutet: alles was sich mit solchen **Programmiersprachen** berechnen lässt, kann auch von Turing-Maschinen berechnet werden und ist damit auch partiell-rekursiv.

Programmiersprachen sind „künstliche“ Sprachen (formale Sprachen) mit der gemeinsamen Eigenschaft, dass ihre Syntax durch exakte Regeln definiert ist. Das heißt für jede Zeichenkette ist es wohl definiert, ob sie ein Wort (Programm) in der gegebenen formalen Sprache darstellt oder nicht.

Solche formalen Sprachen werden durch formale Grammatiken definiert. Damit wird der Begriff der formalen Grammatik zu einer weiteren mathematischen Präzesierung des Begriffs Algorithmus.

Grammatiken erzeugen Wörter. Wir wollen aber Funktionen berechnen! Eine Grammatik berechnet eine Funktion, indem sie ihren Graphen erzeugt.

Es sei $f\colon \Sigma^* \to \Delta^*$ eine Funktion (die durch eine Turing-Maschine berechnet wird). Dann lässt sich eine Grammatik konstruieren, die die folgende Sprache erzeugt:

$$\{(u, v)\colon u \in \Sigma^*, v \in \Delta^* \wedge f(u) = v\}$$

Dies ist gerade der Graph von f.

4.1 Grammatiken und Sprachen vom Typ-0

Definition 4.1
Eine „**Typ-0-Grammatik**“ oder einfach nur Grammatik genannt ist ein Viertupel $G = (N, T, S, P)$ mit

N ... ist ein endliches (nichtleeres) Alphabet – Menge der **Nichtterminale** oder **Variablen**

T ... ist ein endliches (nichtleeres) Alphabet mit $N \cap T = \emptyset$ – Menge der **Terminale**

S ... $S \in N$ ist das **Startsymbol**

P ... $P \subseteq (N \cup T)^* \times (N \cup T)^*$ ist eine endliche Menge von Paaren der Form (p, q), wobei $p \notin T^*$ (d. h. p enthält mindestens ein Nichterminal).
P ist die Menge der **Produktionen** oder **Regeln** von G.

Schreibweise: $p \to q$

Nun stellt sich die Frage, wie mit Hilfe dieser Regeln Wörter erzeugt werden können. Ein Ansatz ist die folgende Idee: Falls ein Wort $w = w_l p w_r$ bereits (irgendwie) erzeugt ist und es eine Regel $p \to q$ gibt, dann wird $w' = w_l q w_r$ erzeugt. Beginnen werden wir mit dem Startsymbol S.

Definition 4.2
Es sei $G = (N, T, S, P)$ eine Grammatik.

1. Ein Wort w' heißt **in einem Schritt** aus einem Wort w **ableitbar** gdw_{def} es zwei Wörter w_l und w_r gibt, die das Wort w bilden als $w = w_l p w_r$, eine Produktion $p \to q$ exsistiert und es gilt $w' = w_l q w_r$

   Schreibweise: $w \to w'$

2. Ein Wort v heißt **ableitbar** aus einem Wort u gdw_{def} es eine Folge von Wörtern $w_0, w_1, \ldots, w_n$ mit $w_0 = u, w_n = v$ und entsprechende Produktionen $w_i \to w_{i+1}$ für $i = 0, \ldots, n-1$ gibt.

   Eine solche Folge heißt **Ableitung der Länge n**

   Schreibweise: $u \to^* v$

3. Die von dieser Typ-0-Grammatik G erzeugte **Typ-0-Sprache** ist gegeben durch

   $$L(G) := \{w \in T^* : S \to^* w\}$$

   Eine Sprache L heißt Typ-0-Sprache gdw_{def} eine Grammatik G mit $L(G) = L$ exsistiert.

4. $CH(0) = \{L : L \text{ ist Typ-0-Sprache}\}$ ist die **Klasse der Typ-0-Sprachen**.

Beispiel 4.1

$$
\begin{aligned}
G &= (N, T, S, P) \\
N &= \{S, R, L\} \\
T &= \{a, \#\} \\
P &= \{S \to \#aL\#, && \text{Regel 1} \\
&= \quad aL \to Laa, && \text{Regel 2} \\
&= \quad \#L \to \#R, && \text{Regel 3} \\
&= \quad \#L \to \#, && \text{Regel 4} \\
&= \quad Ra \to aaR, && \text{Regel 5} \\
&= \quad R\# \to L\#, && \text{Regel 6} \\
&= \quad R\# \to \#\} && \text{Regel 7}
\end{aligned}
$$

Konkrete Beispiele für Ableitungen:

$$
\begin{aligned}
S &\to \#aL\# && \text{Regel 1} \\
\#aL\# &\to \#Laa\# && \text{Regel 2} \\
\#Laa\# &\to \#aa\# && \text{Regel 4}
\end{aligned}
$$

$\Rightarrow \#aa\# \in L(G)$

$$
\begin{aligned}
\#Laa\# &\to \#Raa\# && \text{Regel 3} \\
\#Raa\# &\to \#aaRa\# && \text{Regel 5} \\
\#aaRa\# &\to \#aaaaR\# && \text{Regel 5} \\
\#aaaaR\# &\to \#aaaa\# && \text{Regel 7}
\end{aligned}
$$

$\Rightarrow \#aaaa\# \in L(G)$

$$
\begin{aligned}
\#aaaaR\# &\to \#aaaaL\# && \text{Regel 6} \\
\#aaaaL\# &\to \#aaaLaa\# && \text{Regel 2} \\
&\dots
\end{aligned}
$$

$$L(G) = \{\#\underbrace{a \dots a}_{2^i}\# : i \in \mathbb{N}, i \geq 1\}$$

Grammatiken erzeugen Sprachen, häufig werden die Eigenschaften von Sprachen als Eigenschaften von Grammatiken nachgewiesen! Diese Nachweise werden vereinfacht, falls es gelingt die Produktionen zu vereinfachen: durch **Normalformen**.

Definition 4.3
Eine Grammatik $G = (N, T, S, P)$ ist in **Chomski-Normalform vom Typ-0** gdw_{def}

1. P enthält höchstens eine Regel der Form $p \to \lambda$ (λ: leeres Wort)
2. alle Produktionen haben eine der folgenden Formen
   a) $X \to YZ$
   b) $XY \to Z$
   c) $X \to a$

   wobei $X, Y, Z \in N, a \in T \cup \{\lambda\}$.

Bemerkung 4.1
Diese Einschränkung ist drastisch! Auch unsere Beispielgrammatik aus Beispiel 4.1 erfüllt diese Einschränkung nicht.

Definition 4.4
Zwei Grammatiken G_1 und G_2 sind **äquivalent** gdw_{def} $L(G_1) = L(G_2)$

Satz 4.1
Zu jeder Grammatik G vom Typ-0 gibt es eine äquivalente Normalformgrammatik vom Typ-0.

Beweis:
Der Beweis ist konstruktiv und beschreibt einen Algorithmus, wie man aus einer gegebenen Grammatik G eine äquivalente Normalformgrammatik G' erzeugt. Der Algorithmus vollzieht sich in folgenden Globalschritten und wird in Beispiel 4.2 an der Grammatik aus Beispiel 4.1 illustriert.

1. Schritt: Alle Terminalsymbole sollen nur noch in Regeln der Form 2c auftreten. Dazu wird:

- für jedes Terminalsymbol $t \in T$ ein Nichtterminalsymbol $t' \in N_1$ definiert,
- jede Regel $(p \to q) \in P$ wird durch $(p' \to q') \in P_1$ ersetzt, indem alle Terminale t in p und q durch t' ersetzt werden und so die neuen p' und q' entstehen und
- eine neue Regel $t' \to t$ $(\forall t \in T)$ neu aufgenommen werden.

Die so beschriebene Grammatik bezeichnen wir mit $G_1 = (N_1, T, S, P_1)$

2. Schritt: Wenn es mehrere Regeln der Form $p \to \lambda$ gibt, wird ein neues Nichtterminal L eingeführt, alle diese Regeln durch $p \to L$ ersetzt und eine neue Regel $L \to \lambda$ hinzugefügt.

3. Schritt: Nun sollen alle Regeln $p \to q$, die aus mehr als einem Symbol bestehen, so umgewandelt werden, dass sie entweder von Form 2a oder Punkt 2b sind.

Dazu wird für jede Regel (die nicht schon von der Form 2a oder 2b ist) ein neues Nichtterminal $B \in N_2$ definiert und die alte Regel $p \to q$ durch zwei neue Regeln ersetzt: $p \to B, B \to q$.

Das Ergebnis ist die Grammatik $G_2 = (N_2, T, S, P_2)$.

4. Schritt: Als nächster Schritt soll die Längenbeschränkung realisiert werden. Dabei betrachten wir Regel $p \to q \in P_2$, für die ($|p| > 2$ und $|q| = 1$) oder ($|p| = 1$ und $|q| > 2$) ist. Es sei $X \to Y_1 Y_2 \dots Y_n$ und $n > 2$ die Form von $p \to q$.

Wir definieren neue Nichtterminale $D_1, D_2, \dots, D_{n-2}$ und ersetzen $p \to q$ durch die Regeln

$$X \to Y_1 D_1, D_1 \to Y_2 D_2, D_2 \to Y_3 D_3, \dots, D_{n-3} \to Y_{n-2} D_{n-2}, D_{n-2} \to Y_{n-1} Y_n$$

Falls $X_1 X_2 \dots X_n \to Y$ die Form von $p \to q$ ist, dann definieren wir neue Nichtterminale $E_1, E_2, \dots, E_{n-2}$ und ersetzten $p \to q$ durch

$$X_1 X_2 \to E_1, E_1 X_3 \to E_2, E_2 X_4 \to E_3, \dots, E_{n-3} X_{n-1} \to E_{n-2}, E_{n-2} X_n \to Y$$

5. Schritt: Alle Regeln der Form $X \to Y$, die also auf *beiden* Seiten nur ein Terminal oder Nichtterminal stehen haben, werden nun noch durch je zwei neue Regeln $X \to VW$ und $VW \to Y$ ersetzt, wobei $X, Y \in N_4$. ∎

Beispiel 4.2
An einem Beispiel wollen wir nun den Algorithmus aus Satz 4.1 an der Grammatik aus Beispiel 4.1 nachvollziehen.

1. Schritt:

$$\begin{aligned}
G_1 &= (N_1, T, S, P_1) \\
N_1 &= N \cup \{A, Z\} = \{S, R, L, A, Z\} \\
P_1 &= \{S \to ZALZ, && \text{Regel 1} \\
&= \quad AL \to LAA, && \text{Regel 2} \\
&= \quad ZL \to ZR, && \text{Regel 3} \\
&= \quad ZL \to Z, && \text{Regel 4} \\
&= \quad RA \to AAR, && \text{Regel 5} \\
&= \quad RZ \to LZ, && \text{Regel 6} \\
&= \quad RZ \to Z, && \text{Regel 7} \\
&= \quad Z \to \#, \\
&= \quad A \to a\} \\
L(G_1) &= \{\#a^m\# \colon m = 2^n, n \geq 1\} = L(G)
\end{aligned}$$

2. Schritt: Dieser Fall tritt bei unserer Beispielgrammatik nicht auf.

3. Schritt:

$$
\begin{aligned}
G_2 &= (N_2, T, S, P_2) \\
N_2 &= N_1 \cup \{B_1, B_2, B_3, B_4\} \\
P_2 &= \{S \to ZALZ, && && \text{Regel 1} \\
&= \quad AL \to B_1, && B_1 \to LAA, && \text{Regel 2} \\
&= \quad ZL \to B_2, && B_2 \to ZR, && \text{Regel 3} \\
&= \quad ZL \to Z, && && \text{Regel 4} \\
&= \quad RA \to B_3, && B_3 \to AAR, && \text{Regel 5} \\
&= \quad RZ \to B_4, && B_4 \to LZ, && \text{Regel 6} \\
&= \quad RZ \to Z, && && \text{Regel 7} \\
&= \quad Z \to \#, \\
&= \quad A \to a\}
\end{aligned}
$$

4. Schritt:

$$
\begin{aligned}
G_3 &= (N_3, T, S, P_3) \\
N_3 &= N_2 \cup \{D_1, B_2, E_1, E_2\} \\
P_3 &= \{S \to ZD_1, D_1 \to AD_2, && D_2 \to LZ, && && \text{Regel 1} \\
&= \quad AL \to B_1, && B_1 \to LE_1, && E_1 \to AA, && \text{Regel 2} \\
&= \quad ZL \to B_2, && B_2 \to ZR, && && \text{Regel 3} \\
&= \quad ZL \to Z, && && && \text{Regel 4} \\
&= \quad RA \to B_3, && B_3 \to AE_2, && E_2 \to AR, && \text{Regel 5} \\
&= \quad RZ \to B_4, && B_4 \to LZ, && && \text{Regel 6} \\
&= \quad RZ \to Z, && && && \text{Regel 7} \\
&= \quad Z \to \#, \\
&= \quad A \to a\}
\end{aligned}
$$

5. Schritt: Dieser Fall tritt bei unserer Beispielgrammatik nicht auf.

Wir haben jetzt zu der Grammatik G eine äquivalente Grammatik G_3 in Chomski-Normalform vom Typ-0 konstruiert.

Wie bereits erwähnt werden Grammatiken für höhere **Programmiersprachen** eingesetzt. Sie dienen der Definition der **Syntax** der Sprache. Zur Prüfung der syntaktischen Korrektheit eines Programms stellt sich die Frage, ob das gegebene Programm tatsächlich den Regeln der Grammatik entspricht. Gilt also:

$$P \in L(G)$$

Diese Frage bezeichnet man als das **Wortproblem**. Sie ist gleichwertig zu der Frage nach einem **Parser** [1] für die Grammatik G.

Für Grammatiken vom Typ-0 gibt es trotz der Einschränkungen keinen Parser, d. h. es gibt Grammatiken vom Typ-0, für die *kein* Algorithmus existiert, der entscheidet, ob für eine Eingabe w gilt: $w \in L(G)$ oder $w \notin L(G)$. Typ-0-Grammatiken sind sowohl aus theoretischer wie aus praktischer Sicht zu allgemein.

4.1.1 Beziehungen zwischen dem Leistungsvermögen von Typ-0-Grammatiken und Turing-Maschinen

Satz 4.2
Falls $L \subseteq \Sigma^*$ Turing-aufzählbar ist, dann existiert eine Typ-0-Grammatik G mit $L(G) = L$.

Beweis:
Wenn L Turing-aufzählbar ist, dann existiert (nach Definition 2.10) eine Turing-Maschine M mit $M(\Sigma^*) = L$. Dazu können wir eine Turing-Maschine M' konstruieren: $Acc(M') = L$. Ausgehend von M' konstruieren wir eine Grammatik G, die alle Wörtet erzeugt, die von M' akzeptiert werden.

Es sei $\Sigma^* = \{\lambda = w_0, w_1, \ldots\}$ die **kanonische** (**quasilexikographische**) Auflistung von Σ^*. Für eine Eingabe u simuliert die neue Turing-Maschine M' die Maschine M auf die folgende Weise:

M' vollzieht die Berechnungen von M für die Eingabe $w_0, w_1, w_2, \ldots$ nach und vergleicht jedes (Zwischen-)Ergebnis $M(w_0), M(w_1), M(w_2), \ldots$ mit der Eingabe u. Falls für ein i gilt: $M(w_i) = u$, dann löscht M' den gesamten Bandinhalt und geht in den einzigsten akzeptierenden Finalzustand q_+.

Dann gilt:

$$
\begin{aligned}
u \in L \;& gdw \;\; \exists i \colon M(w_i) = u \\
& gdw \;\; M' \text{ stoppt bei Eingabe } u \\
& gdw \;\; M' \text{ akzeptiert die Eingabe } u \text{ (mit leerem Band und Finalzustand } q_+) \\
& gdw \;\; u \in Acc(M')
\end{aligned}
$$

o. B. d. A. fordern wir:

- der Startzustand q_0 kommt nur in der Anfangskonfiguration vor,
- eine neues Symbol # wird an das rechte Ende der Eingabe gesschrieben und
- das Turingband rechts von # wird nicht benutzt.

[1] Parser: effizienter Algorithmus, der nachweist, ob P Programm ist ($\in L(G)$) oder nicht ($\notin L(G)$)

M'' sei eine so modifizierte Turing-Maschine.

$$Start - M''(u) = q_0 u_1 u_2 \dots u_k \# \qquad u = u_1 \dots u_k$$

$$Acc_{M''}(u) = \begin{cases} q_+ & : u \in L \\ \perp & : \text{sonst} \end{cases}$$

Ziel ist es nun eine Typ-0-Grammatik G zu konstruieren, die ausgehend von der akzeptierenden Konfiguration q_+ von M'' jede mögliche Startkonfiguration $q_0 u_1 \dots u_k \#$ und damit L durch „Rückwärtsrechnung“ erzeugt.

Wir beschreiben eine solche Grammatik in drei Schritten:

1. Schritt: Erzeugung der akzeptierenden Konfiguration mit ausreichend vielen Blanksymbolen $\square$:

$$S \to q_+, q_+ \to q_+\square, q_+ \to \square q_+$$

(erzeugen damit $\square^i q_+ \square^j, i, j \in \mathbb{N}$)

2. Schritt: „Rückwärtsrechnung“

1. falls $\delta(q, a) = (q', a', R)$, dann erstelle die Regel $a'q' \to qa$
2. falls $\delta(q, a) = (q', a', L)$, dann erstelle die Regel $q'ba' \to baq$ $(\forall b \in \Gamma)$
3. falls $\delta(q, a) = (q', a', 0)$, dann erstelle die Regel $q'a' \to qa$

Mithilfe dieser Regeln lässt sich für jede Konfiguration k von M'' eine Vorgängerkonfiguration k^- mit $k^- \vdash_{M''} k$ erzeugen. Als Zwischenergebnis erhalten wir Konfigurationen der Form $\square\square \dots \square q_0 u_1 \dots u_k \#$, wobei $u_1 \dots u_k \in L$

3. Schritt: Die Schlussregel soll noch die $\square, q_0$ und $\#$ löschen:

$$\square q_0 \to q_0, q_0 a \to a q_0 (\forall a \in \Sigma), q_0 \# \to \lambda$$

Als Endergebnis erhalten wir: $u_1 \dots u_n \in L$, also $L(G) = L$. ∎

Das Pendant zum vorherigen Satz liefert der nächste:

Satz 4.3
Jede Sprache $L \subseteq \Sigma^*$, die durch eine Typ-0-Grammatik erzeugt werden kann, ist auch Turing-aufzählbar.

Beweis:
Es sei $G = (N, T, S, P)$ die Grammatik vom Typ-0 (in Normalform) mit $L(G) = L$. Jede solche Grammatik erzeugt einen **markieren Arbeitsbaum** für $L(G)$. Es sei $S \to^* u$. Die direkten Nachfolgeableitungen $Succ$ für ein u $(S \to^* u)$ sind

$$Succ(u) = \{v \colon u \to v\}$$

Gründe für $v \in Succ(u)$:

- an mehreren Stellen in u ist eine Regel aus P anwendbar (z. B. Regel $a \to b$ bei aca kann zu bca oder acb führen) oder
- an ein und derselben Stelle sind verschiedene Regeln anwendbar (z. B. Regeln $a \to b$ und $a \to c$ bei da können zu db oder dc führen)

Deshalb:

- nummerieren wir die Regeln aus P – fixiert
- nummerieren wir von links nach rechts alle Stellen in u, an denen eine Regel anwendbar ist

Damit ergeben sich Paare (r, s) für einen Schritt $u \to v$, die besagen: v ergibt sich aus u (in einem Schritt), indem an der Stelle s die Regel r angewendet wird. Schreibweise: $u \xrightarrow{(r,s)} v$.

Definieren induktiv:

$$Succ^0(\{S\}) = \{S\}$$
$$Succ^{k+1}(\{S\}) = Succ(Succ^k(\{S\}))$$

Damit gilt: $u \in Succ^k(\{S\})$ *gdw* u ist in k Schritten aus S ableitbar ($S \to^k u$).

Und weiterhin gilt:

$$L = \Big(\bigcup_{k=0}^{\infty} Succ^k(\{S\}) \Big) \cap T^*$$

Das heißt $w \in L$ *gdw* $w \in T^* \wedge \exists k \colon w \in Succ^k(\{S\})$

markierter Ableitungsbaum von G:

Knotenmenge V:

$$V = \bigcup_{k=0}^{\infty} Succ^k(\{S\})$$

Kantenmenge E: für jedes $u \in V$ und alle $v \in Succ(\{u\})$ ziehen wir eine Kante $u \to v$ mit der Marke (r, s), falls $u \xrightarrow{(r,s)} v$.

Dieser Baum ist abzählbar und eingeteilt in Schichten $Succ^0(\{S\}), Succ^1(\{S\}), \ldots$ todo: Skizze einfügen

Es sei $w \in L(G)$ (vergleiche Skizze). Wir konstruieren eine Turing-Maschine, die eine Eingabe w genau dann akzeptiert, wenn $w \in L(G)$, und anderenfalls nicht stoppt.

Die Turing-Maschine M arbeitet bei Eingabe w prinzipell so: M durchmustert den Ableitungsbaum Schicht für Schicht (Breitensuche) solange, bis sie die Eingabe findet.

Die Markierung der Kanten im Baum ermöglicht die Orientierung der Maschine. Eine Eingabe wird genau dann gefunden, wenn $w \in L(G)$. In diesem Fall akzeptiert M die Eingabe.

$$w \in Acc(M)$$ ■

Definition 4.5
Da Typ-0-Sprachen von Turing-Maschinen aufgezählt werden können, bezeichnet man die **Klasse der Typ-0-Sprachen** auch mit RE (recursive enumerable). $RE = CH(0)$ (vgl. Definition 4.2)

4.2 Grammatiken und Sprachen vom Typ-1

Ausgangspunkt für unsere kommenden Überlegungen ist, dass das Wortproblem für Sprachen vom Typ-0 unlösbar ist.

Definition 4.6 (Typ-1-Grammatik)
Eine Grammatik $G = (N, T, S, P)$ heißt **Typ-1-Grammatik** oder vom **Erweiterungstyp** oder **nicht verkürzend** gdw_{def}

- für alle Regeln $(p \to q) \in P$ gilt:

$$|p| \leq |q| \text{ oder } p \to q \text{ ist } S \to \lambda$$

- Falls $S \to \lambda \in P$, dann kommt S auf keiner rechten Seite einer Regel vor.

Eine Sprache $L \subseteq \Sigma^*$ ist vom **Typ-1** gdw_{def} es eine Typ-1-Grammatik G mit $L(G) = L$ gibt.

$CH(1)$ bezeichnet die **Klasse aller Typ-1-Sprachen**.

Definition 4.7 (Typ-1-Normalform-Grammatik)
Eine Typ-1-Grammatik $G = (N, T, S, P)$ ist **Typ-1-Normalform** gdw_{def} in P kommen nur Regeln der folgenden Form vor (außer $S \to \lambda$, falls $\lambda \in L(G)$):

1. $X \to YZ$,
2. $XY \to UV$ oder
3. $X \to a$

für $U, V, X, Y, Z \in N$ und $a \in T$

Satz 4.4
Zu jeder Typ-1-Grammatik G gibt es eine äquivalente Typ-1-Normalform-Grammatik G' mit $L(G) = L(G')$.

BEWEIS:
(Beweis ist konstruktiv) todo: aus Übung ergänzen ∎

Eine weitere wichtige Grammatikklasse ist die folgende

Definition 4.8 (kontextsensitive Grammatik)
Eine Grammatik $G = (N, T, S, P)$ heißt **kontextsensitiv** gdw_{def} alle Regeln haben die Form:

- $uAv \to uwv$ oder
- $S \to \lambda$

wobei $A \in N$ und $w \in (N \cup T)^+$ und $u, v \in (N \cup T)^*$.

Falls $S \to \lambda$ eine Regel ist, dann kommt S auf keiner rechten Seite einer Regel vor.

Beispiel 4.3 (für eine kontextsensitive Grammatik)

$$
\begin{aligned}
G &= (N, T, S, P) \\
N &= \{S, S', A, A', B, B'\} \\
T &= \{a, b\} \\
P &= \{S \to \lambda, S \to S', S' \to ABS', AB \to A'B, \\
&\qquad A'B \to A'A, A'A \to BA, BA \to B'A, \\
&\qquad B'A \to B'B, B'B \to AB, A \to a, B \to b\}
\end{aligned}
$$

Einerseite ist G kontextsensitiv, andererseits ist G *beinahe* Typ-1-Normalform (eine Ausnahme).

Frage: Welche Sprache wird von dieser Grammatik überhaupt erzeugt? Ansatz: $S' \to ABS'$ ersetzen durch $S' \to AC$ und $C \to BS'$.

Satz 4.5
Zu jeder kontextsensitiven Grammatik G gibt es eine äquivalente Typ-1-Grammatik G' mit $L(G) = L(G')$ und umgekehrt

Definition 4.9
Man bezeichnet die **Klasse der Typ-1-Sprachen** auch mit $CS = CH(1)$.

Satz 4.6
Für Sprachen vom Typ-1 ist das **Wortproblem** lösbar! Das heißt, zu *jeder* Grammatik G (vom Typ-1 oder kontextsensitiv) gibt es einen Algorithmus, der entscheidet, ob für eine Eingaben w gilt:

$$w \in L(G) \text{ oder } w \notin L(G)$$

Die **kontextsensitiven Sprachen** sind für die Praxis die bedeutendste Sprachklasse! Alle Programmiersprachen sind kontextsensitiv!

Aber: Das **Wortproblem** ist für kontextsensitive Sprachen zwar lösbar, jedoch i. a. *nicht effizient*!

Deshalb werden Programmiersprachen i. a. nicht durch rein-kontextsensitive Grammatiken beschrieben, sondern durch weiter vereinfachte Grammatiken bis auf „Ausnahmen“: z. B. im „linken Kontext“ muss für jeden Bezeichner zuvor der Typ deklariert sein.

4.3 Grammatiken und Sprachen vom Typ-2

Da wir mit Typ-1-Grammatik das Wortproblem lösen können, steht nun die Frage nach einem effizienten Parser.

Definition 4.10
Eine Grammatik $G = (N, T, S, P)$ ist vom **Typ-2** oder **kontextfrei** gdw_{def} alle Regeln sind von der Form $A \to w$ wobei $A \in N$ und $w \in (T \cup N)^*$ – auf der linken Seite steht genau ein Nichtterminal.

Eine Sprache $L \subseteq \Sigma^*$ heißt **kontextfrei** oder vom **Typ-2**, falls eine kontextfreie Grammatik G mit $L(G) = L$ existiert.

$CH(2) = CF$ bezeichnet die **Klasse der kontextfreien Sprachen**.

Klar ist: $CH(2) \subseteq CH(0)$ und $CH(1) \subseteq CH(0)$, aber gilt auch $CH(2) \subseteq CH(1)$?

An zwei Beispielen zeigen wir, dass nicht jede kontextfreie Grammatik kontextsensitiv ist.

Beispiel 4.4
Eine kontextfreie Sprache ist die Menge alle korrekten Klammerausdrücke, die sogenannte **Dyck-Sprache** D_2.

$$(()()) \in D_2, \qquad ((())) \in D_2, \qquad ())() \notin D_2$$

$$\begin{aligned} T &= \{(,)\} \\ N &= \{S\} \\ P &= \{S \to (S), S \to SS, S \to \lambda\} \end{aligned}$$

Diese Grammatik ist nicht vom Typ-1, weil S auf einer rechten Seite auftaucht, obwohl es die Regel $S \to \lambda$ gibt!

Beispiel 4.5
Die Menge aller **Palindrome** $PAL = \{w \in \Sigma^* : w = w^R\}$.

$$\begin{aligned} T &= \{a, b\} \\ N &= \{S\} \\ P &= \{S \to aSa, S \to bSb, S \to \lambda, S \to a, S \to b\} \end{aligned}$$

Auch diese Grammatik ist nicht kontextsensitiv.

Satz 4.7
Zu jeder kontextfreien Grammatik gibt es eine **λ-freie Grammatik** [2] G', die kontextfrei ist und für die gilt: $L(G') = L(G) \setminus \{\lambda\}$.

Beweis:
Wir nehmen an, die Grammatik hat die Regeln

$$\begin{aligned} A &\to \lambda \\ C &\to A_1 A A_2 \\ C &\to A_1 A_2 \end{aligned}$$

Wir definieren M_i induktiv als die Menge der Nichtterminale, die nach i Ableitungsschritten in das leere Wort überführt werden. $A \in M_i \ gdw \ A \to^i \lambda$

$$\begin{aligned} M_1 &= \{A \in N : (A \to \lambda) \in P\} \\ M_{i+1} &= M_i \cup \{B \in N : (B \to u) \in P \text{ mit } u \in M_i\} \end{aligned}$$

Damit gilt: Für alle $i = 1, 2, \ldots$ ist stets $M_i \subseteq N$. Also existiert ein Grenzwert k mit $M_k = M_{k+1} = M_{k+2} = \ldots$ Es gilt dann $A \in M_k \ gdw \ A \to^* \lambda$.

Damit können wir folgende Regelmenge P' definieren:

$$P' = P \setminus \{A \to \lambda : (A \to \lambda) \in P\} \cup \{C \to u' : u' \neq \lambda \wedge \exists (C \to u) \in P \wedge u' \text{ entsteht durch weglassen min. eines Symbols aus } M_k\}$$ ∎

Beispiel 4.6
Nehmen wir an, wir haben die Regel $C \to A_1 A_2 A_3 A_4 A_5$ und es gilt

$$A_i \in N, \quad A_2, A_3, A_4 \in M_k, \quad A_1, A_5 \notin M_k$$

[2] „λ-frei“ bedeutet, es gibt keine Regel der Form $w \to \lambda$

Dann werden folgende Regeln zusätzlich aufgenommen:

$$C \to A_1 A_3 A_4 A_5$$
$$C \to A_1 A_2 A_4 A_5$$
$$C \to A_1 A_2 A_3 A_5$$
$$C \to A_1 A_4 A_5$$
$$C \to A_1 A_3 A_5$$
$$C \to A_1 A_2 A_5$$
$$C \to A_1 A_5$$

Fallunterscheidung:

1. Fall: $\lambda \notin L(G)$. Dann gilt: $L(G') = L(G) \setminus \{\lambda\} = L(G)$ und damit ist G' äquivalent (und λ-frei) zu G

2. Fall: $\lambda \in L(G)$. Wir konstruieren G'' durch folgende Angaben:

$$N'' = N \cup \{S''\}, \quad P'' = P' \cup \{S'' \to \lambda, S'' \to S\}$$

Insgesamt erhalten wir: $L(G') = L(G) \setminus \{\lambda\}$, $G' = (N, T, S, P')$.

Dann gilt: $G'' = (N'', T, S'', P'')$ und $L(G'') = L(G) \cup \{\lambda\} = L(G)$ und G'' ist kontextsensitiv.

Damit haben wir:

Satz 4.8
Jede **kontextfreie Sprache** ist **kontextsensitiv**. (obwohl nicht jede kontextfreie Grammatik kontextsensitiv ist.)

Dies bedeutet für die **Chomsky-Hierarchie**:

- $CH(0) = RE$...die Klasse der **rekursiv-aufzählbaren Sprachen**
- $CH(1) = CS$...die Klasse der **kontextsensitiven Sprachen**
- $CH(2) = CF$...die Klasse der **kontextfreien Sprachen**

$$CF \subseteq CS \subseteq RE$$

Definition 4.11
Eine kontextfreie Grammatik $G = (N, T, S, P)$ ist in **Normalform für Typ-2-Grammatiken** gdw_{def}

- falls $\lambda \in L(G)$, dann (und nur dann) gilt die Regel $S \to \lambda$,
- S tritt in keiner rechten Seite einer Regel auf und

- außer der λ-Regel haben alle Regeln die Form:

$$\begin{array}{ll} A \to BC \text{ oder} & A, B, C \in N \\ A \to a & a \in T \end{array}$$

Satz 4.9
Zu jeder kontextfreien Grammatik G gibt es eine äquivalente kontextfreie Grammatik G' in Normalform!

Beobachtung: Falls $w \in L(G')$ und $|w| > 1$ ist, dann gibt es in G' eine Ableitung von w der Länge $2n - 1$. Diese Beobachtung ist Grund dafür, dass das **Wortproblem** für kontextfreie Sprachen effizient lösbar ist!

Ausgangspunkt ist das Wort $w = w_1 \dots w_n \in L$

1. w_n muss aus einem Nichtterminal A_n entstanden sein; altes Wort: $w' = w_1 \dots w_{n-1} A_n$
2. w_{n-1} muss aus einem Nichtterminal A_{n-1} entstanden sein; altes Wort: $w'' = w_1 \dots w_{n-2} A_{n-1} A_n$
3. A_{n-1} und A_n müssen aus einem Nichtterminal A'_n entstanden sein; altes Worts: $w''' = w_1 \dots w_{n-2} A'_n$
4. Mit dem Wort w''' können wir wieder zu Punkt 2 gehen.

Um ein Wort der Form $w = w_1 \dots w_{n-1} A_n$ um einen Buchstaben zu verkürzen ($w' = w_1 \dots w'_{n-2} A_n$) braucht es zwei Schritte. Für das gesamte Worte also $2(n-1)$ Schritte um ein Nichtterminal – das Startsymbol – zu erreichen. Ein zusätzlicher Schritt (Punkt 1) ist noch notwendig, um das Ausgangswort in diese Form zu bringen.

Satz 4.10
Zu jeder kontextfreien Grammatik G gibt es einen *effizienten* Algorithmus, der entscheidet, ob eine Eingabe $w \in L(G)$ ist oder nicht.

Was heißt effizient?

- Laufzeit z. B. des Algorithmus' von Kasany/Younger: $O(n^3)$ für eine vollständige Sprachanalyse
- Laufzeitredurzierung durch „schnelle Matrixmultiplikation" von Solovay/Strassen: $O(n^{\log_2 7})$
- offene Frage: Gibt es einen Algorithmus in $O(n^2)$?

Das **Wortproblem** in der **Chomski-Hierarchie**

$CH(0)$	unlösbar
$CH(1)$	lösbar
$CH(2)$	effizient lösbar

4.4 Grammatiken und Sprachen vom Typ-3

Als Ziel steht die weitere Vereinfachung der Regeln, um das Wortproblem mit linearem Aufwand zu lösen.

Definition 4.12
Eine Grammatik $G = (N, T, S, P)$ ist eine **Typ-3-** oder **rechtslineare** oder **reguläre** Grammatik gdw_{def} alle Regeln sind von der Form

$A \to w$ oder	$w \in T^*, A \in N$
$A \to wB$	$B \in N$

Definition 4.13
Eine Grammatik $G = (N, T, S, P)$ heißt **linkslinear** gdw_{def} alle Regeln sind von der Form

$A \to w$ oder	$w \in T^*, A \in N$
$A \to Bw$	$B \in N$

Satz 4.11
Zu jeder **rechtslinearen Grammatik** G gibt es eine äquivalente **linkslineare Grammatik** G' und umgekehrt!

Definition 4.14
Eine Sprache $L \subseteq \Sigma^*$ ist eine **Typ-3-Sprache** oder **reguläre Sprache**, falls es eine reguläre Grammatik G mit $L(G) = L$ gibt.

$CH(3) = REG...$bezeichnet die **Klasse der Typ-3-Sprachen** (regulären Sprachen)

Es gilt:

$$REG \subseteq CF \subseteq CS \subseteq RE$$

4.5 Zusammenfassung

Grammatik	allgemeine Form	Chomski-Normalform
Typ-0	$(p \to q) \in (N \cup T)^+ \times (N \cup T)^*$ p enthält min. ein $X \in N, p \neq \lambda$	$X \to a$ $X \to YZ$ $XY \to Z$
Typ-1	$S \to \lambda$ und S in keiner rechten Seite $\lvert p\rvert \leq \lvert q\rvert$ oder $S \to \lambda$ und S in keiner rechten Seite $uAv \to uwv, w \in (N \cup T)^+$ kontextsensitiv	$X \to a$ $X \to YZ$ $XY \to UV$
Typ-2	$A \to w, w \in (N \cup T)^*$	$X \to a$ $X \to YZ$
Typ-3	$A \to w$ $A \to wB, w \in T^*$ rechtslinear $A \to w$ $A \to Bw, w \in T^*$ linkslinear	$X \to a$ $X \to aY$ $X \to a$ $X \to Ya$

$A, B \in N, \quad U, V, X, Y, Z \in (N \cup T)$

5 Die Hierarchie der Sprachklassen

Wir wissen bereits:

Typ-3		Typ-2		Typ-1		Typ-0
regulär		kontextfrei		kontextsensitiv		aufzählbar
REG	$\subseteq$	CF	$\subseteq$	CS	$\subseteq$	RE

Satz 5.1 (Hierarchiesatz für Klassen der Chomski-Hierarchie)

$$REG \subsetneq CF \subsetneq CS \subsetneq RE$$

Beweis:
1. $RE \setminus CS \neq \emptyset$ wegen Satz 5.2 und Satz 5.3

■

5.1 Das Wortproblem für Sprachen vom Typ-i ($i = 0, 1, 2, 3$)

Wiederholung: Als Wortproblem bezeichnet man die Frage, ob für eine Eingabe w und eine Sprache $L \subseteq \Sigma^*$ vom Typ-i ($i = 0, \ldots, 3$) gilt

$$w \overset{?}{\in} L$$

Satz 5.2
Das Wortproblem für Sprachen vom Typ-0 ist nicht lösbar. Das heißt, es gibt Sprachen vom Typ-0, die nicht entscheidbar sind.

Beweis:
Im Kapitel über Aufzählbarkeit und Entscheidbarkeit todo: Link ergänzen; Den Beweis gibt es nicht

■

Satz 5.3
Das Wortproblem für Sprachen vom Typ-1 ist lösbar. Das heißt, jede Sprache vom Typ-1 ist entscheibbar.

Es gilt: Das Wortproblem für Sprachen vom Typ-1 ist ***PSPACE*-vollständig**, d. h. dieses Problem ist praktisch *hochgradig ineffizient*.

Beweis:
Es sei $G = (N, T, S, P)$ eine (Normalform-)Grammatik vom Typ-1 und es sei $w \in \Sigma^*(= T^*)$.

1. Fall: $w = \lambda$, (triviale Fall) Frage: Gilt $\lambda \in L(G)$? Oder anders formuliert: Ist $(S \to \lambda) \in P$?

2. Fall: $w \neq \lambda$, es sei $|w| = n$, dann gilt $n \geq 1$

Idee: berechnen *alle* Wörter $x \in (N \cup T)^*$ mit $|x| \leq n$. Dies funktioniert, weil die Regeln nicht verkürzend sind; wir definieren dazu Wortmengen $Z_n(m)$.

Dabei ist $Z_n(m)$ die Menge alljener Wörter über $(N \cup T)$, die höchstens die Länge n haben und in höchstens m Schritten aus S ableitbar sind.

$$Z_n(m) = \{x \in (N \cup T)^* \colon |x| \leq n \wedge S \to^k x, k \leq m\}$$

Induktive Definition von $Z_n(m)$:

$$\begin{aligned} Z_n(0) &:= \{S\} \\ Z_n(m+1) &:= Z_n(m) \cup \{x \in (N \cup T)^* \colon |x| \leq n \wedge \exists y \in Z_n(m) \text{ mit } y \to x\} \end{aligned}$$

Dann gilt:

$$\{w \in L(G) \colon |w| \leq n\} \subseteq \bigcup_{m \geq 0} Z_n(m)$$

und es gilt:

$$Z_n(0) \subseteq Z_n(1) \subseteq \ldots \subseteq Z_n(m) \subseteq Z_n(m+1) \subseteq \ldots$$

Andererseits ist durch die Längenbeschränkung $|x| \leq n$ $(x \in Z_n(m))$ die Anzahl der möglichen Wörter in $Z_n(m)$ beschränkt und damit endlich.

$$\Big| \bigcup_{m \geq 0} Z_n(m) \Big| \leq \sum_{i=1}^{n} (|N| + |T|)^i \tag{5.1}$$

Deshalb existiert ein $k \geq 1$ mit $Z_n(k) = Z_n(k+1)$

Dies liefert den folgenden Algorithmus:
begin { Eingabe: w }

1. $n := |w|$
2. $m := 0$
3. $Z_n(0) = \{0\}$
4. repeat

5. berechne $Z_n(n+1)$
6. $m := m + 1$
7. until $w \in Z_n(m)$ oder $Z_n(m+1) = Z_n(m)$
8. if $w \in Z_n(m)$
9. then return „Ja, $w \in L(G)$“
10. else return „Nein, $w \neq L(G)$“ ∎

Bemerkung 5.1
zu Gleichung 5.1: Im allgemeinen sind für die Eingaben w der Länge n exponentiell viele Zwischenschritte $x \in \bigcup_{m \geq 0} Z_n(m)$ zu berechnen $\Rightarrow O(k^n)$.

5.2 Das Pumping-Lemma für kontextfreie Sprachen

Satz 5.4 (Das Pumping-Lemma für CF)
Es sei $L \in CF$. Dann existiert ein $n \in \mathbb{N}^+$, so dass für alle Wörter $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ mit folgenden Eigenschaften existiert:

1. $|vx| \geq 1$ ($v \neq \lambda$ oder $x \neq \lambda$)
2. $|vwx| \leq n$
3. $\forall k \geq 1 \colon uv^k wx^k y \in L$

Beweis:
Es sei G mit $L(G) = L$ eine (Chomski-)Normalform-Grammatik vom Typ-2. Es sei die Anzahl der Nichtterminale $|N| = n_1$ und es sei $n = 2^{n_1}$.

Wir beschreiben die Ableitung von z in der Grammatik G durch einen Ableitungsbaum. Ein solcher **Ableitungsbaum** ist ein Binärbaum mit einer Wurzel, die mit dem Startsymbol S markiert ist und mit $|z|$ vielen Blättern, die mit den Buchstaben aus z markiert sind. siehe Abbildung 5.1 Alle inneren Knoten sind mit Nichtterminalen markiert: Es gibt zwei Sorten von inneren Knoten:

- mit *einem* Sohn: die stehen für Regeln der Form $X \to a$ und haben alle die Höhe 1
- mit *zwei* Söhnen: die stehen für Regeln der Form $X \to YZ$

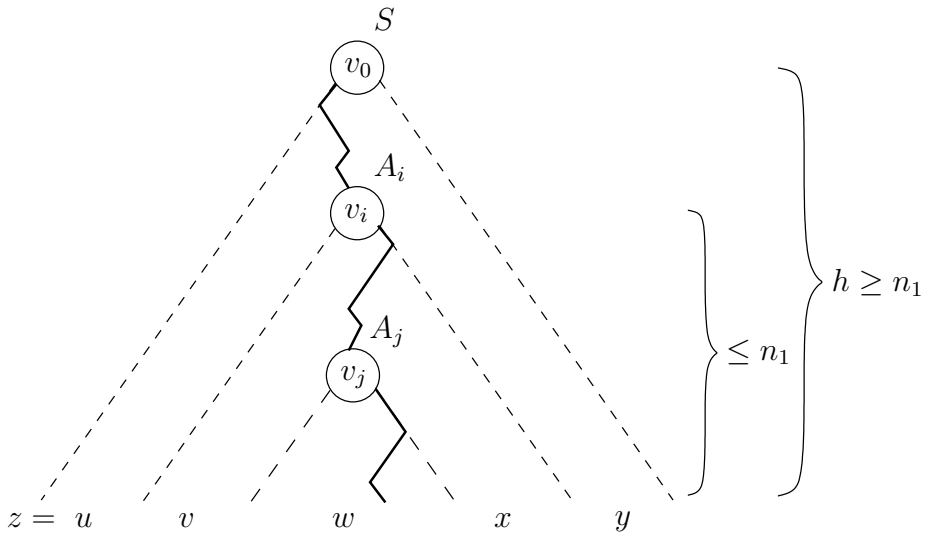


Abbildung 5.1: Allgemeiner Ableitungsbaum einer kontextfreien Grammatik.

Der gesamte Graph ist der Ableitungsbaum T für z. Die Höhe von T sei $h + 1$. Dann gilt: $h \geq \log_2|z| \geq \log_2 n = n_1$. Also ist $h \geq n_1 = |N|$.

Es gibt insgesamt n_1 verschiedene Nichtterminale. Für $h > n_1$ gibt es darum auf einem Pfad von der Wurzel S bis zu einem Blatt z_i mit dem Knoten $v_0 = S, v_1, v_2, \ldots, v_n, v_{n+1} = z_i$ zwei Knoten v_i und v_j $(i < j)$, die mit dem Nichtterminal A markiert sind, und für alle anderen v_k $(k \geq i + 1)$ sind die Markierungen verschieden. Die Höhe des Knotens v_i wird somit durch n_1 beschränkt.

Wir setzen $A = A_i = A_j$!

Die durch A_i und A_j bestimmten Teilbäume bewirken eine Zerlegung des Wortes z in $z = uvwxy$.

Eigenschaften dieser Zerlegung:

1. Die Knoten v_i stehen für die Anwendung der Regel $A_i \to BC$. o. B. d. A. sei v_j ein Nachfolger des Knotens, der mit C markiert ist.

   Dann ist das Wort, das aus B abgeleitet wird, ein Anfangswort von v.

   Deshalb ist $v \neq \lambda$ (andersnfalls $x \neq \lambda$) und damit gilt $|vx| \geq 1$.

2. Da die Höhe des Knotens v_i durch n_1 beschränkt ist, hat der durch A_i bestimmte Teilbaum höchstens $2^{n_1} = n$ Blätter!

   Das hierdurch erzeugte Wort ist vwx und es gilt $|vwx| \leq n$.

3.
   - Wird der Teilbaum bei A_j durch den Teilbaum A_i erzeugt (m. a. W. bei v_j wird nochmal der Teilbaum A_i verwendet), ergibt sich ein Ableitungsbaum T_2^2 für das Wort $z_2 = uv^2wx^2y$
   - dieser Vorgang lässt sich beliebig oft wiederholen
   - wird der Teilbaum bei A_i durch den Teilbaum bei A_j ersetzt, ergibt sich dein Ableitungsbaum T_2^0 für das Wort $z_0 = uwy$

Insgesamt: mit $z = uvwxy$ gehören alle Wörter uv^kwx^ky ($k \in \mathbb{N}$) zu L! ∎

Beispiel 5.1
Wir wissen bereits (aus der Übung), dass $L = \{a^nb^nc^n : n \in \mathbb{N}\}$ eine kontextsensitive Sprache ist. Nehmen wir nun an, L sei auch eine kontextfreie Sprache. Dann gilt das Pamping-Lemma (Satz 5.4) und es gibt eine natürliche Zahl n, so dass für alle $z \in L$ mit $|z| \geq n$ ein Zerlegung $z = uvwxy$ mit den Eigenschaften 1, 2 und 3 existiert.

Wir betrachten folgendes Wort: $z = a^nb^nc^n$ für dieses spezielle n. Dann ist $|z| = 3n \geq n$.

Nun untersuchen wir die entsprechende Zerlegung $z = uvwxy$. Nach Eigenschaft 2 gilt $|vwx| \leq n$. Damit kann das Teilwort vwx aber nicht gleichzeitig a's *und* b's *und* c's enthalten.

Entsprechendes gilt für das Teilwort vx. Es gibt also mindestens einen Buchstaben, der nicht in vx vorkommt.

Aus der Eigenschaft 1 folgt, dass $|vx| \geq 1$ ist, d. h. vx enthält mindestens einen Buchstaben.

Wegen Eigenschaft 3 gehört uwy zu L. Aber dieses Wort hat weniger a's als b's und damit nicht die Form $a^nb^nc^n$, womit es nicht Teil der Sprache L seien kann.

Aus diesem Widerspruch folgt, dass die Annahme $L \in CF$ falsch ist.

5.3 Das Pumping-Lemma für reguläre Sprachen

Satz 5.5 (Das Pumping-Lemma für *REG*)
Für jede Sprache $L \in REG$ gibt es eine natürliche Zahl n, so dass für alle Wörter $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvw$ existiert, die folgende Eigenschaften erfüllt:

1. $|v| \geq 1$
2. $|uv| \leq n$
3. für alle $k \in \mathbb{N}_0$ ist $uv^kw \in L$

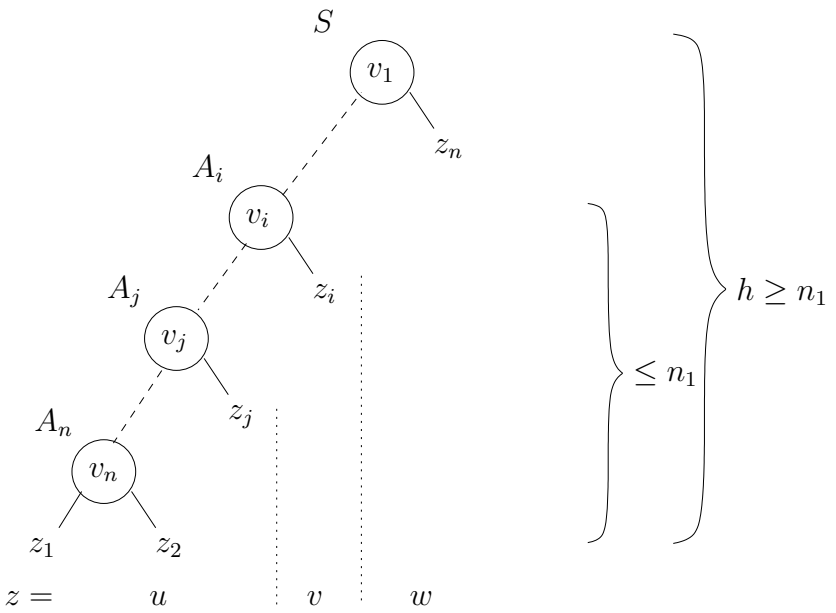


Abbildung 5.2: Ableitungsbäume für linkslineare Grammatiken sehen wie eine Raupe mit der Haarlänge 1 aus todo: Die Zeichnung könnte noch flacher sein

Beweis:
o. B. d. A. $\lambda \notin L$. Idee des Beweises: Es sei G eine linkslineare Grammatik in Normalform vom Typ-3. Alle Regeln sind von der Form $A \to Ba$ oder $A \to a$.

Wir setzen $n_1 = |N|$. Dabei ist $G = (N, T, S, P)$ und $n = n_1 + 1$.

Wie sehen in einer solchen Grammatik die Ableitungsbäume aus? Abbildung 5.2 ∎

Beispiel 5.2
Wir wissen, dass $L = \{a^n b^n : n \in \mathbb{N}\}$ eine kontextfreie Sprache ist. Treffen wir nun die Annahme, L sei regulär.

Dann würde das Pumping-Lemma für reguläre Sprachen Satz 5.5 gelten und wir könnten ein (spezielles) n (für diese Sprache) finden, so dass für alle Wörter $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvw$ mit den Eigenschaften 1, 2 und 3 existiert.

Wir betrachten jetzt das Wort $z = a^n b^n$ für dieses spezielle n. Dann folgt aus Eigenschaft 2, dass $|uv| \leq n$ und damit $|v| \leq n$ ist. Eigenschaft 1 liefert: $|v| \geq 1$. Diese beiden Sachen zusammen ergeben: $1 \leq |v| \leq n$.

Also kommen in v a's vor und keine b's. Also hat uw weniger a's als b's und damit nicht die Form $a^n b^n$, obwohl nach Eigenschaft 3 $uw \in L$ ist – Widerspruch.

Beispiel 5.3

(5.2) $L_1 = \{a^{m^2} : m \in \mathbb{N}\}$ $w \in L_1$ *gdw* $w \in \{a\}^* \wedge |w|$ ist Quadratzahl

Prüfen wir, ob diese Sprache regulär ist und nehmen dazu an, sie sei es.

Entsprechend gilt dann das Pumping-Lemma für reguläre Sprachen Satz 5.5 und wir finden eine natürliche Zahl n, so dass für alle Wörter $z \in L_1$ mit $|z| \geq n$ eine Zerlegung $z = uvw$ mit den Eigenschaften 1, 2 und 3 existiert.

Wir betrachten das Wort z^{n^2} mit der zugehörigen Zerlegung $z = uvw$.

Eigenschaft 1 und 2 liefern $1 \leq |v| \leq |uv| \leq n$.

Wegen Eigenschaft 3 ist auch $uv^2w \in L_1$ und es gilt:

$$n^2 = |uvw| < |uv^2w| = |uvw| + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$$

Die Länge des Wortes uv^2w ist aber demnach keine Quadratzahl ($n^2 < |uv^2w| < (n+1)^2$), weshalb es nicht zur Sprache L_1 gehört – Widerspruch. Die Annahme L_1 sei eine reguläre Sprache war also falsch.

Beispiel 5.4

(5.3) $\quad L_2 = \{a^p : p \text{ ist Primzahl}\} \qquad w \in L_2 \ gdw \ w \in \{a\}^* \wedge |w| \text{ ist Primzahl}$

Prüfen wir, ob diese Sprache regulär ist und nehmen dazu an, sie sei es.

Entsprechend gilt dann das Pumping-Lemma für reguläre Sprachen Satz 5.5 und wir finden eine natürliche Zahl n, so dass für alle Wörter $z \in L_2$ mit $|z| \geq n$ eine Zerlegung $z = uvw$ mit den Eigenschaften 1, 2 und 3 existiert.

Wir betrachten das Wort $z = a^p$ ($n + 2 \leq p$ ist prim) und die zugehörige Zerlegung $z = uvw$.

Eigenschaft 1 liefert wieder, dass v mindestens einen Buchstaben hat, und Eigenschaft 3 liefert $z_i = uv^iw \in L_2$ ($\forall i \in \mathbb{N}$)

$$|z_i| = |uw| + i|v|$$

Dabei muss $|uw| \geq 2$ sein, denn

$$\begin{aligned} |uw| &= |z| - |v| \geq n + 2 - |v| \\ &\geq n + 2 - n = 2 \qquad \text{wegen Eigenschaft 2 } |v| \leq n \end{aligned}$$

Setzen wir $i = |uw|$. Dann gilt für dieses i:

(5.4) $$|z_i| = |uw| + |uw||v| = |uw| \cdot (1 + |v|)$$

z_i ist aber nicht in L_2, da die Länge $|z_i|$ keine Primzahl ist (Faktorisierung in $|uw| \cdot (1+|v|)$).

5.4 Kontextfreie Sprachen über einelementigen Alphabeten

Ist die Sprache L_1 aus Gleichung 5.2 oder die Sprache L_2 aus Gleichung 5.3 kontextfrei?

Überprüfen wir L_1 und behaupten dazu L_1 sei kontextfrei.

Dann gilt das Pumping-Lemma für kontextfreie Sprachen Satz 5.4 und wir finden eine natürliche Zahl n_1, so dass für alle Wörter $z \in L_1$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ mit den Eigenschaften 1, 2 und 3 existiert.

Da nur ein Buchstabe vorhanden ist, ist die Position der Teilwörter unwesentlich und kann als $z = u'v'w'$ geschrieben werden. Für einbuchstabige Sprachen fallen die Aussagen der beiden Pumping-Lemma Satz 5.4 und Satz 5.5 also zusammen.

L_1 und L_2 sind also nicht kontextfrei und es gilt sogar der folgende Satz:

Satz 5.6
Jede kontextfreie Sprache $L \subset \{a\}^*$ über einem einbuchstabigen Alphabet ist regulär.

6 Die Hierarchie der Automaten

Ausgangspunkt ist der folgende Satz:

Satz 6.1
Eine Sprache $L \subseteq \Sigma^*$ ist vom **Typ-0** *gdw* es eine **gewöhnliche deterministische Turing-Maschine** M mit $L(M) = L$ gibt.

Dies erlaubt die folgende Sprechweise: **Gewöhnliche Turing-Maschinen** sind Turing-Maschinen vom Typ-0.help: Widerspruch zu Definition 6.1

Was sind dann Turing-Maschine vom Typ-i $(i = 1, 2, 3)$?

6.1 Das Phänomen „Nichtdeterminismus“

Wir hatten im Beweis von Satz 4.3 für eine Typ-0-Grammatik in Normalform $G = (N, T, S, P)$ den markierten Ableitungsbaum mit $Succ(u) = \{v \colon u \to v \text{ gemäß Regel } (p \to q) \in P\}$ definiert. Typisch dabei ist, dass aus einer Situation *mehrere* Nachfolgersituationen entstehen. Anders war es bisher mit Turing-Maschinen, bei denen aus einer Situation *genau eine* Nachfolgersituation entsteht.

Dies wirft einige Fragen auf:

1. Können Turing-Maschinen mit dieser Fähigkeit des nichtdeterminierens ausgestattet werden.
2. Was heißt es für eine solche Turing-Maschine, dass sie die Eingabe akzeptiert?
3. Haben nichtdeterministische Turing-Maschinen eine größere Berechnungskraft als gewöhnliche deterministische Turing-Maschinen?

Zur Frage 1 ist folgendes zu sagen: Bisher war die Überführungsfunktion δ für eine Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ als Funktion definiert. Die Berechnung von M bei einer Eingabe w ist eine eindeutig bestimmte Folge von Konfigurationen

$$Start - Konf_M(w) = K_0 \vdash K_1 \vdash K_2 \vdash \ldots \vdash \begin{cases} K_{acc} \\ \ldots \end{cases}$$

Jede Berechnung ist ein (endlicher oder unendlicher) Pfad. Das ist **Determinismus**! (Dies ist gerade der Unterschied zu Ableitungen in Grammatiken!)

todo: Skizze

Abbildung 6.1: caption

Ein Möglicher Ausweg wäre, wir erlauben für jede Konfiguration mehrere Nachfolger mit dem Effekt, dass die Berechnung für eine Eingabe w die Form eines todo: Skizze **Berechnungsbaumes** bekommt (**Nichtdeterminismus**) (Der Berechnungsbaum der Turing-Maschine entspricht dem markierten Ableitungsbaum der Grammatik.)

Formal gesprochen heißt das, die Überführungsfunktion

$$\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R, 0\}$$

wird durch eine **Überführungsrelation**

$$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R, 0\})$$

ersetzt. Diese Relation lässt sich ihrerseits als Funktion schreiben:

$$\delta \colon Q \times \Gamma \to \mathfrak{P}(Q \times \Gamma \times \{L, R, 0\})$$

Eine **nichtdeterministische Turing-Maschine** ist eine Turing-Maschine mit einer solchen Überführungsfunktion.

Zur Frage 2 folgendes: Wir können den Berechnungsbaum wie ein Labyrinth betrachten (Abbildung 6.1). Es gibt zwei Strategien, dieses Labyrinth abzulaufen:

1. **Obelix-Strategie**: an jeder Verzweigung klont sich Obelix, wobei einer der Klone den Schatz findet
2. **Asterix-Strategie**: ein Asterix durchläuft systematisch alle Gänge

Gemäß der Obelix-Stategie akzeptiert eine nichtdeterministische Turing-Maschine eine Eingabe w, wenn sie auf einem Pfad erfolgreich war:

$$L(M) = \{w \in \Sigma^* \colon Start - Konf_M(w) \vdash^* K_{acc}\}$$

Zur Frage 3 folgendes: Gemäß der Asterix-Strategie lässt sich jede nichtdeterministische Turing-Maschine (Algorithmus, Arbeitsweise) durch eine deterministische Turing-Maschine (Algorithmus, Arbeitsweise) simulieren! Daher sind beide Systeme gleichmächtig in Bezug auf Berechenbarkeit.

Jedoch besteht ein wesentlicher Unterschied im Zeitaufwand. Für ein Labyrinth der Tiefe t benötigt die Obelix-Strategie lineare Laufzeit $O(t)$, die Asterix-Strategie hingegen exponentiellen Zeitaufwand $O(2^t)$.

Turing-Maschine	Sprachklasse	Automaten		
		nichtdeterm.		determ.
Typ-0: gewöhnliche, nichtdeterministische Turing-Maschine	RE	RE	$?^a$	RE
		Turing-Maschinen		
Typ-1: nichtdeterministische Turing-Maschine mit endlichem Band	CS	LBA	$?^b$	$DLBA$
		linear beschränkte Autom.		
Type-2: nichtdeterministische Turing-Maschine mit einem Einwegeingabeband[c] und einem zweiten Turing-Band, auf dem der Kopf stehts am rechten Ende der Eingabe steht	CF	PDA	$\supsetneq$	$DPDA$
		Kellerautomaten		
Typ-3: nichtdeterministische Turing-Maschine, bei dem die Eingabe auf einem Einwegeingabeband steht	REG	NFA	$=$	DFA
		endliche Automaten		

[a] P-NP-Problem: Sind nichtdeterministische Turing-Maschinen schneller als ($NP \supsetneq P$) oder gleichschnell wie ($NP = P$) deterministische Turing-Maschinen
[b] LBA-Problem: $LBA \supsetneq DLBA$ oder $LBA = DLAB$
[c] **Einwegeingabeband**: jedes Zeichen kann nur einmal gelesen werden

Tabelle 6.1: Beziehungen zwischen Turing-Maschinen, Sprachen und Automaten

6.2 Turing-Maschinen vom Typ-0

Definition 6.1 (nichtdeterministische Turing-Maschinen)
Als eine **nichtdeterministische Turing-Maschine** $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ bezeichnet man eine Turing-Maschine mit der Überführungsfunktion

$$\delta \colon Q \times \Gamma \to \mathfrak{P}(Q \times \Gamma \times \{L, R, 0\})$$

Die von einer nichtdeterministischen Turing-Maschine akzeptierte Sprache beinhaltet alle Wörter, für die es einen akzeptierenden Berechnungspfad gibt.

$$L(M) = \{w \in \Sigma^* \colon Start - Konf_M(w) \vdash^* Konf_{acc}\}$$

$\mathcal{DTM}$ ... Klasse der Sprachen, die durch eine deterministische Turing-Maschine akzeptiert werden
$\mathcal{NTM}$ ... Klasse der Sprachen, die durch eine nichtdeterministische Turing-Maschine akzeptiert werden
$CH(0)$ ... Klasse der Sprachen, die durch eine Grammatik vom Typ-0 erzeugt werden

Wir wissen bereits aus Abschnitt 4.1.1, dass $\mathcal{DTM} = CH(0)$. Es gilt also $\mathcal{DTM} \subseteq CH(0)$ – für jede deterministische Turing-Maschine existiert eine Grammatik G vom Typ-0, so dass G die Maschine M durch „Rückwärtsberechnung“ simmuliert – Satz 4.2.

Dieser Beweis lässt sich wortwörtlich auf nichtdeterministische Turing-Maschine übertragen. Das heißt, $\mathcal{NTM} \subseteq CH(0)$. Andererseits gilt $\mathcal{DTM} \subseteq \mathcal{NTM}$ und damit $CH(0) \subseteq \mathcal{DTM} \subseteq \mathcal{NTM} \subseteq CH(0)$.

Wie lässt sich die Grammatik G unmittelbar durch eine nichtdeterministische Turing-Maschine M simulieren? Eingabe für die nichtdeterministische Turing-Maschine M sei ein Wort $w \in \Sigma^*$. $w \in L(G)$ *gdw* $S \to_G^* w$ Die nichtdeterministische Turing-Maschine M „rät“ rückwärts eine mögliche Ableitung für w. (Der Berechnungsbaum von M entspricht damit möglichen Ableitungsfolgen!)

Einschub: P-NP bisher: $\mathcal{DTM} = \mathcal{NTM}$ (Grundsätzlich die selbe Berechnungskraft, falls keine Einschränkung)

dagegen:

P ... Klasse aller Sprachen, die durch eine deterministische Turing-Maschine mit *polynomieller* Rechenzeit akzeptiert werden

NP ... Klasse aller Sprachen, die durch eine nichtdeterministische Turing-Maschine mit *polynomieller* Rechenzeit akzeptiert werden

Die **Rechenzeit** einer **deterministischen** Turing-Maschine ist die Länge der Berechnung als Folge von Konfigurationen oder mit anderen Worten: ist die Länge des Berechnungspfades.

Die **Rechenzeit** einer **nichtdeterministischen** Turing-Maschine ist die Tiefe des Berechnungsbaumes.

Diese Rechenzeit wird gemesssen bzgl. der Länge der Eingabe. Polynomielle **Rechenzeit** heißt also, es gibt ein Polynom p, so dass für alle Eingaben $w \in \Sigma^*$ gilt:

$$Zeit_M(w) \leq p(|w|)$$

Die Arbeit einer **nichtdeterministischen Turing-Maschine** lässt sich in „Rate-“ und „Testphasen“ aufteilen.

Beispiel 6.1 (Erfüllbarkeitsproblem)
Es sei F eine Formel mit den Atomen $A_1, \ldots, A_n$. F gehört zu SAT *gdw* es eine Belegung $\beta\colon \{A_1, \ldots, A_n\} \mapsto \{0, 1\}$ mit $I_\beta(F) = 1$ gibt.

$$SAT = \{F\colon F \text{ ist Formel in konjunktiver Normalform und } F \text{ ist erfüllbar}\} \in NP$$

Eine nichtdeterministische Turing-Maschine M_N rät eine Belegung β und berechnet I_β. Ist $I_\beta(F) = 1$ akzeptiert sie, wenn nicht rät sie die nächste Belegung. Existiert keine Belegung β mit $I_\beta(F) = 1$, stoppt die Turing-Maschine *nie*.

Die Frage, ob $SAT \in P$ ist bis jetzt ungeklärt. Es ist aber bereits gezeigt worden, dass $SAT \in DEXP$ (in exponenzieller Zeit mit deterministischen Turing-Maschinen entscheidbar).

Wäre $SAT \in P$, so wäre $P = NP$. Damit ist SAT ein ***NP*-schweres Problem**. SAT ist ***NP*-vollständig**.

Beispiel 6.2 (Hamiltonkreis-Problem)

$$HC = \{G : G = (E, V) \text{ einfacher ungerichter Graph und } G \text{ besitzt Hamiltonkreis}\}$$

Wir interpretieren $G = (E, V)$ als ein Straßennetz, in dem die Knoten $v \in V$ Städte und die Kanten $e \in E$ Straßen zwischen den Städten sind. Als **Hamiltonkreis** bezeichnet man eine Rundreise durch G, bei der jede Stadt *genau* einmal besucht wird und man in der Stadt, in der man begonnen hat, endet. Wenn es eine solche Rundreise gibt, besitzt G einen Hamiltonkreis.

Es gilt: $HC \in NP$. Eine nichtdeterministische Turing-Maschine M_N rät eine Folge von Kanten ($n!$ Möglichkeiten) und prüft, ob diese geratene Folge einen Hamiltonkreis bildet.

Die Frage, ob $HC \in P$, ist ebenfalls ungeklärt, aber man konnte – wie schon für SAT – zeigen, dass $HC \in DEXP$. Gleichfalls gilt: Könnte man zeigen, dass $HC \in P$, so wäre $NP = P$. Damit ist HC auch ein ***NP*-vollständiges** (***NP*-schweres**) Problem.

6.3 Turing-Maschinen vom Typ-1

Formal gesehen, ist eine Turing-Maschine vom Typ-1, eine Turing-Maschine mit dem Sonderzeichen $\triangleright$ am linken Ende und $\triangleleft$ am rechten Ende, die nicht überschrieben und nicht verrückt werden dürfen. Für die Überführungsfunktion heißt das

$$\delta(q, \triangleright) = (q', \triangleright, L)$$
$$\delta(q, \triangleleft) = (q', \triangleleft, L)$$

mit dem Effekt, dass sich der gesamte Bereich zwischen $\triangleright$ und $\triangleleft$ vollzieht.

Eine solche Turing-Maschine kann für eine Eingabe w (mit $\triangleright w \triangleleft$) rückwärts die Ableitung einer nichtverkürzenden Grammatik (vom Typ-1 – Abschnitt 4.2) nachvollziehen.

6.4 Turing-Maschinen vom Typ-2

Motivation: Wir betrachen Linksableitungen von kontextfreien Grammatiken (Abschnitt 4.3) in Normalform (Definition 4.11).

Beispiel 6.3

$$\begin{aligned} G &= (N, T, S, P) \\ N &= \{S, A, B\} \\ T &= \{a, b, c\} \\ P &= \{S \to SS, S \to aAa, S \to bBb, \\ &\qquad A \to aAa, A \to c \\ &\qquad B \to bBb, B \to c\} \end{aligned}$$

Eine mögliche Linksableitung wäre:

$$\begin{aligned} S \to SS \to bBbS \to bcbS \to bcbSS \to bcbaAaS \\ \to bcbaaAaaS \to bcbaacaaS \to bcbaacaaaAa \to bcbaacaaaca \in T^* \end{aligned}$$

Für jedes $w \in L(G)$ existiert stets eine Linksableitung. Es sei $S \to^* uAv \to^* w$, d. h. uAv ist ein Zwischenergebnis einer Linksableitung und hat die Form $u \in T^*, A \in N, v \in (N \cup T)^*$

6.4.1 Beschreibung von Linksableitungen durch Kellerautomaten

Einwegeingabeband: die Eingaben w steht zwischen zwei Endmarken $\triangleright$ und $\triangleleft$ und kann (nur) von links nach rechts gelesen werden. Das **Kellerband** ist ein Turing-Arbeitsband mit einer linken Endmarke $*$ (**Kellersymbol**) und der Einschränkung, dass der Kopf stehts am rechten Ende des Bandinhalts steht.

Ein Zwischenergebnis uAv einer Linksableitung steht für:
u ... steht für den Teil der Eingabe w, der bereits gelesen wurde
v ... ist der Inhalt des Kellerbandes
A ... wird in der Zustandsmenge gespeichert

Die Überführungsfunktion der entsprechenden Turing-Maschine:

$$\delta \colon Q \times \underbrace{(\Sigma \cup \{\triangleright, \triangleleft\})}_{=\Sigma'} \times \underbrace{(\Gamma \cup \{*\})}_{=\Gamma'} \to Q \times \Gamma' \times \{R, 0\}$$

Für einen Kellerautomaten sieht die Überführungsfunktion etwas anders aus:

$$\delta \colon Q \times (\Sigma' \cup \{\lambda\}) \times \Gamma' \to Q \times \Gamma'$$

zwei Regeltypen der Grammatik:

1. $A \to a$: Buchstabe wird verglichen mit dem nächsten Buchstaben der Eingabe. Falls Übereinstimmung $\to$ weiterrechnen, falls nicht $\to$ Abbrechen.

   „Leseschritt“

2. $A \to BC$: kein Test von Buchstaben der Eingabe, sondern schreiben C auf das Kellerband und nehmen B als neuen Zustand an.

Definition 6.2 (Formale Definition des Kellerautomaten)
Ein nichtdeterministischer **Kellerautomat** K ist ein Tupel $K = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

Q ... endliche Menge der Zustände
Σ ... endliche Menge, Eingabealphabet, $\triangleright, \triangleleft \notin \Sigma$
Γ ... endliche Menge, Kelleralphabet, $* \notin \Gamma$
$q_0 \in Q$ ... Startzustand
$F \in Q$ ... Finalzustände

die Überführungsfunktion

$$\delta \colon Q \times (\Sigma \cup \{\triangleright, \triangleleft\} \cup \{\lambda\}) \times \underbrace{(\Gamma \cup \{*\})}_{=\Gamma'} \to \mathfrak{P}(Q \times \Gamma')$$

6.4.2 Akzeptanzverhalten eines Kellerautomaten

Ein Konfiguration (q, w', γ) ist eine Momentanbeschreibung des Automaten, die gegeben ist durch den aktuellen Zustand q, den noch nicht gelesenen Teil w' der Eingabe und den kellerinhalt γ.

Es sei $w' = aw''$ und $\gamma = \gamma' A$ (Topsymbol des Kellers)

1. (Lese-Schritt), $(q', \beta) \in \delta(q, a, A)$, Nachfolgekonfiguration: $(q', w'', \gamma'\beta)$

   a) $\gamma \in \Gamma^+$ – Schreib- oder Push-Operation

   b) $\gamma = \lambda$ – Lösch- oder Pop-Opteration

2. (λ-Schritt), $(q', \beta) \in \delta(q, \lambda, A)$, Nachfolgekonfiguration: $(q', w'', \gamma'\beta)$

   a) $\beta \in \Gamma^*$ – Push-Operation (von links nach rechts bzw. von unten nach oben)

   b) $\beta = \lambda$ – Pop-Operation (von rechts nach links bzw. oben nach unten)

   Dies ist gerade das **Kellerprinzip** oder auch **Push-Down-Prinzip** genannt.

Es gibt zwei Möglichkeiten, wie ein Kellerautomat eine Eingabe w akzeptieren kann:

- entweder mit Finalzuständen

$$L(K) = \{w \in \Sigma^* \colon (q_0, w\triangleleft, *) \vdash^* (q, \triangleleft, *\gamma) \wedge q \in F\}$$

- oder mit einem leeren Keller

$$L_\lambda(K) = \{w \in \Sigma^* \colon (q_0, w\triangleleft, *) \vdash^* (q, \triangleleft, *) \wedge q \in F\}$$

Satz 6.2
Zu jedem Kellerautomaten K, der mit Finalzuständen akzeptiert, gibt es einen äquivalenten Kellerautomat K', der mit einem leeren Keller akzeptiert, so dass $L(K) = L_\lambda(K')$

Beispiel 6.4

$$L = \{ww^R \colon w \in \{0,1\}^*\}$$

Idee: Wir lesen die Eingabe und schreiben w auf das Kellerband. Aufgrund des Kellerprinzips ist w von rechts nach links, also in umgekehrter Reihenfolge zu lesen und wird mit der Eingabe w^R vergleichen.

Dabei haben wir ständig die Wahl zwischen „weiter w einlesen“ und „beginnen w^R zu testen“. Dies ist gerade der Nichtdeterminismus des Kellerautomaten.

$$\begin{aligned}
\Sigma &= \{0,1\} \\
\Gamma &= \{0,1\} \\
Q &= \{q_0, q_1, q_+, q_-\} \\
F &= \{q_+\}
\end{aligned}$$

Seien $a, b \in \{0,1\}$, wobei $a \neq b$

$$\begin{aligned}
\delta(q_0, a, *) &= \{(q_0, *a)\} \\
\delta(q_0, \triangleleft, *) &= \{(q_+, *)\} \\
\delta(q_0, a, b) &= \{(q_0, ba)\} \\
\delta(q_0, a, a) &= \{(q_0, aa), (q_1, \lambda)\} \\
\delta(q_1, a, a) &= \{(q_1, \lambda)\} \\
\delta(q_1, \triangleleft, *) &= \{(q_+, *)\} \\
\delta(q_1, a, b) &= \{(q_-, b)\}
\end{aligned}$$

Satz 6.3
Eine Sprache $L \subseteq \Sigma^*$ ist von Typ-2 *gdw* L wird von einem nichtdeterministischen Kellerautomat akzeptiert.

6.5 Turing-Maschinen vom Typ-3

Definition 6.3
Ein nichtdeterministischer Automat M ist ein Tupel $M = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ wobei

Q ... endliche Menge von Zuständen
$Q_0 \subseteq Q$ ... endliche Menge von Anfangszuständen
$F \subseteq Q$ ... endliche Menge von Finalzuständen

und der Überführungsfunktion

$$\delta \colon Q \times \Sigma \to \mathfrak{P}(Q)$$

Diese Überführungsfunktion δ wird fortgesetzt zu einer **erweiterten Überführungsfunktion** δ'

$$\delta' \colon \mathfrak{P}(Q) \times \Sigma^* \to \mathfrak{P}(Q)$$
$$\delta'(Q', wa) = \bigcup_{q \in Q'} \delta(\delta(q, w), a)$$

$L \subseteq \Sigma^*$ wird von M akzeptiert *gdw* $L = \{w \in \Sigma^* \colon \delta'(Q_0, w) \cap F \neq \emptyset\}$

7 Ausblick in die Theorie der Entscheidbarkeit

7.1 Entscheidbarkeit und Aufzählbarkeit

Widerholung: todo: Links finden

- $L \subseteq \Sigma^*$ heißt entscheidbar *gdw* χ_L ist berechenbar
- $L \subseteq \Sigma^*$ heißt semientscheidbar *gdw* χ_L^P ist berechenbar

$$\chi_L(w) = \begin{cases} 1 & : w \in L \\ 0 & : w \notin L \end{cases} \qquad \chi_L^P(w) = \begin{cases} 1 & : w \in L \\ \perp & : \text{sonst} \end{cases}$$

- $L \subseteq \Sigma^*$ heißt aufzählbar *gdw* es eine total definierte und berechenbare Funktion f mit $f(\Sigma^*) = L$ gibt oder $L = \emptyset$.

These von Church (Satz 1.3): Jede im intuitiven Sinne mit Hilfe eines Algorithmus' berechenbare Funktion ist berechenbar.

7.2 Charakterisierung der aufzählbaren Mengen

Definition 7.1
Es sei M eine Turing-Maschine, die die Funktion $Res_M \colon \Sigma^* \to \Delta^*$ berechnet.

Die Menge $D_M = \{u \in \Sigma^* \colon \text{die Maschine } M \text{ stoppt bei Eingabe } u\}$ heißt **Haltebereich** der Turing-Maschine M. (D_M ist gerade der Definitionsbereich der Funktion Res_M)

Die Menge $W_M = \{v \in \Delta^* \colon \exists u \in \Sigma^* \colon v = Res_M(u)\}$ heißt **Ergebnisbereich** der Turing-Maschine M. (W_M ist gerade der Wertebereich der Funktion Res_M)

Satz 7.1
Für eine Menge $L \subseteq \Sigma^*$ sind folgende Aussagen äquivalent:

1. L ist aufzählbar
2. L ist Ergebnisbereich einer Turing-Maschine

3. L ist Haltebereich einer Turing-Maschine
4. χ_L^P ist berechenbar
5. L wird von einer Typ-0-Grammatik erzeugt

Beweis:

1. $\Rightarrow$ 2.
   1. Wenn $L = \emptyset$ ist, dann ist L der Ergebnisbereich einer Turing-Maschine, die nirgends stoppt.
   2. Wenn $L \neq \emptyset$ ist, Dann ist $L = f(\Sigma^*)$ und jede Turing-Maschine, die f berechnet, hat L als Ergebnisbereich

2. $\Rightarrow$ 3. L sei der Ergebnisbereich einer Turing-Maschine. $D_M^* = \{v_1, v_2, v_3, \ldots\}$ sei eine Aufzählung von D_M in quasilexikographischer Reihenfolge. Wir konstruieren eine Turing-Maschine M', die für eine Eingabe $w \in \Sigma^*$ genau dann stoppt, wenn M das Ergebnis w produziert.

   M' arbeitet wie folgt: für Eingabe w wird (nacheinander) erzeugt:

   $$w\#\#Konf_i^M(v_0)\#Konf_{i-1}^M(v_1)\#\ldots\#Konf_{i-j}^M(v_j)\#\ldots\#Konf_0^M(v_i)$$

   Prüfe, ob ein Resultat von M vorliegt:

   Ja: vergleiche die Eingabe w mit dem Resultat $Res_M(v_j) = u$

      Ja: dann (lösches alles überflüssige und) stoppe

      Nein: dann künfig ignorieren und weiterrechen

   Nein: Bestimme für jedes $Konf_{i-j}^M(v_i)$ die Nachfolgekonfiguration $Konf_{i-j+1}^M(v_i)$ sowie $Konf_0^M(v_{i+1})$

   Es gilt: $w \in W_M$ *gdw* $w \in D_M$

3. $\Rightarrow$ 4. Es sei M eine Turing-Maschine mit $D_M = L$. M' arbeitet wie M, wobei auf einer zusätzlichen Spur alle Felder markiert werden, die der Arbeitskopf gelesen hat. Falls M bei einer Eingabe w stoppt, löscht M' alle markierten Bandinhalte und schreibt „1“. Falls M bei der Eingabe w nicht stoppt, dann stoppt auch M' nicht.

   Damit berechnet M' gerade χ_L^P.

4. $\Rightarrow$ 5.
   1. $L = \emptyset$ klar
   2. $L \neq \emptyset$: Es sei w_{fix} ein Wort aus L. Wir schreiben folgenden Algorithmus „**dovetailing**“ (*engl. Schwalbenschwanz*)

      Eingabe: $w \in \Sigma^*$

      a) Bestimme die Nummer n von w in der quasilexikographischen Aufzählung (d. h. bestimme $D_M(n) = w$)

b) Verstehen n als Cantornummer und bestimmen $n = c(l(n), r(n))$

c) setzen $u = v_{l(n)}$ und $t = r(n)$

d) M sei eine Turing-Maschine, die χ_L^P berechnet. Simuliere die Arbeit von M auf der Eingabe u mit insgesamt t Takten! Falls M das Resultat „1“ liefert ($u \in L$), liefert der Algorithmus die Ausgabe u. Sollte M nicht das Resultat „1“ liefern, gibt der Algorithmus die Ausgabe w_{fix} aus.

Das bedeutet also:

- für jede Eingabe stoppt der Algorithmus
- alle Ausgaben gehören zu L
- jedes Wort aus L kommt als Ausgabe vor

$$\begin{aligned} w \in L &\Rightarrow \chi_L^P(w) = 1 \\ &\Rightarrow Res_M(w) = 1 \\ &\Rightarrow \exists t_0 \in \mathbb{N}\colon M \text{ liefert } Res_M(w) = 1 \text{ in } t_0 \text{ Takten} \end{aligned}$$

n_0 sei die Nummer von w ($w = v_{n_0}$). Weiter sei $n = c(n_0, t_0)$. Bei Eingabe v_n liefert der Algorithmus gerade w.

5. $\Rightarrow$ 1. Satz 4.3 ∎

7.3 Codierung von Turing-Maschinen

Wir kennen aus Satz 3.2 bereits ein „Wörterbuch“ der partiell-rekursiven Funktionen und wollen jetzt ein „Wörterbuch“ der Turing-berechenbaren Funktionen. Die Idee ist, wir konstruieren die folgende Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

$$\begin{aligned} Q &= \{0, 1, \ldots, n\} \\ \Sigma \subseteq \Gamma &= \{a_0 = \square, a_1, a_2, \ldots, a_{k-1}\} \\ F &= \{1\} \\ q_0 &= 1 \end{aligned}$$

δ besteht aus $k{\cdot}n$ Zeilen zu je fünf Werten $(i, a_j) \mapsto (i, a_k, bew)$. Wir betrachten

$$A = \{a_0, \ldots, a_{k-1}\} \cup \{0, 1, \ldots, n\} \cup \{;, ,\}$$

Die Turing-Maschine kann umkehrbar eindeutig durch folgendes Wort $w_{TM} \in A^*$ dargestellt werden:

$$
\begin{aligned}
w_{TM} := n, a_0, \ldots, a_{k-1}, a_{i_1}, \ldots, a_{i_r}, \\
&0, \quad a_0, \quad \delta(0,a_0), \quad 0, \quad a_1, \quad \delta(0,a_1), \quad \ldots, \quad 0, \quad a_{k-1}, \quad \delta(0,a_{k-1}), \\
&1, \quad \ldots \\
&2, \quad \ldots \\
&\vdots \\
&n, \quad a_0, \quad \delta(n,a_0), \quad \ldots\ldots\ldots\ldots\ldots\ldots, \quad n, \quad a_{k-1}, \quad \delta(n,a_{k-1})
\end{aligned}
$$

Definition 7.2
w_{TM} heißt **Standardcodierung** der Turing-Maschine TM.

Für ein beliebiges $w \in A^*$ ist es entscheidbar, ob eine Turing-Maschine TM mit $w = w_{TM}$ existiert. Das Alphabet A hängt offensichtlich von TM ab (da $\Gamma \subseteq A$). Es sei $A = \{b_1, \ldots, b_j\}$. Verwenden deshalb $B = \{0,1\}$ und definieren die folgende Abbildung $\beta\colon A^* \to B^*$ durch

$$
\begin{aligned}
\beta(\lambda) &= \lambda \\
\beta(b_i) &= 10^i && (1 \le i \le j) \\
\beta(uv) &= \beta(u) \cdot \beta(v) && (u, v \in A)
\end{aligned}
$$

β ist ein Homomorphismus von A^* nach B^* und eindeutig bestimmt!

Definition 7.3
$bw_{TM} = \beta(w_{TM})$ heißt **binäre Standardcodierung** von TM. ($bw_{TM} \in \{0,1\}^*$ und für jedes $w \in \{0,1\}^*$ ist entscheidbar, ob es eine binäre Standardcodierung ist.)

Die binäre Standardcodierung liefert eine *effektive Nummerierung* aller Turing-Maschinen:
Eingabe: n
Erzeuge: das n-te Wort über $\{0,1\}^*$, das binäre Standardcodierung einer Turing-Maschine ist. Dies sei bw_n.

Nach der These von Church (Satz 1.3) gibt es eine Turing-Maschine, die als Ergebnisbereich gerade $\{bw_0, bw_1, bw_2, \ldots\}$ hat.

Definition 7.4
T_n sei diejenige Turing-Maschine TM mit $bw_{TM} = bw_n$. n heißt **Gödelnummer** der Turing-Maschine.

Eine **Gödelisierung** ist also eine *effektive Nummerierung* der Turing-Maschinen!

Wir haben damit auch zwei Gödelisierungen der aufzählbaren Mengen:
$D_n = D_{T_n}$ ... ist Haltebereich von T_n
$W_n = W_{T_n}$ ... ist Ergebnisbereich von T_n

Wir betrachten Turing-Maschinen mit $\{0,1\} \subseteq \Sigma$.

Definition 7.5
$H := \{bw_{TM}11n\colon$ die Turing-Maschine TM stoppt bei Eingabe $w\}$ $(H \subseteq \{0,1\}^*)$ Die Wortmenge H heißt **Halteproblem** für Turing-Maschinen.

Satz 7.2 (Anhaltesatz)
1. H ist aufzählbar
2. H ist nicht entscheidbar
3. $\bar{H} = \{0,1\}^* \setminus H$ ist nicht aufzählbar

Definition 7.6
Eine Turing-Maschine, die H aufzählt, heißt **universelle Turing-Maschine**.

Bemerkung 7.1
$H \in CH(0) \setminus CH(1)$

Index

$PSPACE$-vollständig, 71
$\mathbb{F}$, 43

abgeschlossen bezüglich des Schemas der Einsetzung, 36
ablehnender Zustand, 24
ableitbar, 55
  in einem Schritt, 55
Ableitung der Länge n, 55
Ableitungsbaum, 73
akzeptierender Zustand, 24
Akzeptor, 24
allgemein-rekursiv, 43
Alphabet, 12
Asterix-Strategie, 80

Basis, 18
berechnet, 19
Berechnungsbaumes, 80
Binärdarstellung, 14
Blanksymbol, 15
Buchstabe, 12

charakteristische Funktion, 20
Chomski-Hierarchie, 68, 69
Chomski-NF
  Typ-0, 57
Chomsky-Hierarchie, 67

D_2, *siehe* Dyck-Sprache
Determinismus, 79
$DEXP$, 82
Dezimaldarstellung, 14
$\div$, 52
dovetailing, 89
$\mathcal{DTM}$, 81
Dyck-Sprache, 65

Einsetzungsprinzip, 36
Einwegeingabeband, 81, 84
Endkonfiguration, 16
ε, 12
Ergebnisbereich, 88
Erkenner, 24
Euklidischer Algorithmus, 9, 10

Finalzustand, 16
Folgekonfiguration, 16
formale Sprache, 13
Fortsetung, 43
Funktion
  nirgends definierte, 21

Grammatik
  λ-freie, 66
  Erweiterungstyp, 63
  kontextfrei, 65
  kontextsensitiv, 64
  linkslinear, 69
  nicht verkürzend, 63
  rechtslinear, 69
  Typ-0, 55
  Typ-1, 63
  Typ-1-Normalform, 63
  Typ-2, 65
  Typ-3, 69
Grundfunktionen, 36
Gödelisierung, 91
Gödelnummer, 91

Haltebereich, 88
Halteproblem, 92
Hamiltonkreis, 83
Homomorphismus, 14

kanonische, 60
kanonische Ordnung, 12

Kellerautomat, 85
Kellerautomaten, 84
Kellerband, 84
Kellerprinzip, 85
Kellersymbol, 84
Klasse
    allg.-rek. Fkt., 43
    kontextfreie Sprachen, 65
    part.-rek. Fkt, 40
    Sprachen
        Typ-3, 69
    Turing-ber. Fkt., 22
    Typ-0-Sprachen, 55
    Typ-1-Sprachen, 63
Klasse $\mathbb{P}$r der primitiv-rekursiven Funktionen, 37
Klasse der Typ-0-Sprachen, 63
Klasse der Typ-1-Sprachen, 64
Kleene-Hülle, 13
Konfiguration, 15
Konkatenation, 12
Konkaternation, 13

λ, 12
leere Wort, 12
Länge des Wortes, 12

markieren Arbeitsbaum, 61
markierter Ableitungsbaum, 62
Menge aller endlichen Wörter, 12
mod, 9, 53
μ-Rekursion, 40

nicht definiert, 19
Nichtdeterminismus, 79, 80
Nichtterminale, 55
Normalform
    Grammatik, *siehe* Chomski-NF
    Typ-2-Grammatik, 67
Normalformen, 56
NP-schwer, 82, 83
NP-vollständig, 82, 83
$\mathcal{NTM}$, 81

Obelix-Strategie, 80

Palindrome, 16, 66
Parser, 60
partielle charakteristische Funktion, 20
Potenzen, 13
primitive Rekursion, 36
Produktionen, 55
Programmiersprachen, 54, 59
Projektion, 36
Push-Down-Prinzip, 85

quasilexikographische, 60
quasilexikographische Reihenfolge, 12

Rechenzeit
    deterministische, 82
    nichtdeterministische, 82
    polynomielle, 82
Regeln, 55
Rest bei der ganzzahligen Division, 9

SAT, 82
Speicherplatzbeschränkung, 33
Spiegelsprache, 13
Spiegelwort, 12
Sprache
    kontextfrei, 65, 67
    kontextsensitiv, 67
    rek.-aufzählb., 67
    Typ-0, 55, 79
    Typ-1, 63
    Typ-2, 65
    Typ-3, 69
Sprachen
    kontextsensitiv, 65
Standardcodierung, 91
    binäre, 91
Startkonfiguration, 16
Startsymbol, 55
$Succ$, 61
Syntax, 59

Takt, 15
Terminale, 55
$\mathcal{TM}$, 22
totaldefinierte Fuktion, 17

Turing-aufzählbar, 20
Turing-Band, 14
Turing-berechenbar, 19, 21
Turing-berechenbar mit der Zeitschranke, 33
Turing-entscheidbar, 20
Turing-erkennbar mit der Zeitbeschränkung, 33
Turing-Maschine, 15
    deterministische, 79
    gewöhnliche, 79
    nichtdeterministische, 80–82
    universelle, 92
Turing-semientscheidbar, 20

unmittelbare Nachfolgekonfiguration, 16

Variablen, 55

Wort, 12
Wortfunktion, 19
Wortproblem, 60, 64, 65, 68
    kontextfreie Sprachen, 68

Zahlenfunktion, 19

Überführungsfunktion
    erweiterte, 87
Überführungsrelation, 80

äquivalent, 57