

Theoretische Informatik

Kurzsript zu den Vorlesungen von

Prof. Van Bang Le Prof. Karsten Wolf

18. Februar 2018

Inhaltsverzeichnis

0	Wichtige Begriffe aus der Berechenbarkeitstheorie	3
1	Komplexität	4
1.1	Graphen	4
1.2	Zeitaufwand und Komplexitätsklassen	4
1.3	Polynomielle Reduktion und NP-Vollständigkeit	6
1.4	Platzkomplexität	7
1.5	Umgang mit schwierigen Problemen	7
1.5.1	Betrachtung von Spezialfällen	7
1.5.2	Annäherungsverfahren	7
1.5.3	Parametrisierte Algorithmen	8
1.5.4	Randomisierte Algorithmen	8
2	Formale Sprachen	9
2.1	Grundbegriffe	9
2.2	Chomsky Hierarchie	10
2.3	Endliche Automaten	10
2.4	Reguläre Ausdrücke	12
2.5	Reguläre Sprachen	12
2.6	Kontextfreie Sprachen	14
3	Semantik von Programmiersprachen	16
3.1	Eine einfache Programmiersprache: W	16
3.2	Operationelle Semantik	16
3.2.1	Natural Semantics (Big Step)	17
3.2.2	Structural Operational Semantics (Small Step)	17
3.3	Denotationelle Semantik	18
3.3.1	Direct-Style-Semantik	18
3.3.2	Flussgraph-Analyse	19
3.4	Axiomatische Semantik	19
3.4.1	Hoare-Kalkül	19
4	Formale Systeme	21
4.1	Prozessalgebra	21
4.2	Statecharts	22
4.3	Petri-Netze	23

0 Wichtige Begriffe aus der Berechenbarkeitstheorie

Ist ein Problem algorithmisch lösbar?

Definition 0.0.1 (Turingmaschine). Eine **deterministische Turingmaschine** (DTM) (oder ein det. Turingprogramm) M ist ein 5-Tupel $M = (Z, \Sigma, \delta, z_a, z_e)$ mit:

- Z ist eine endliche Menge von Zuständen
- z_a ist der ausgezeichnete Anfangszustand
- z_e ist der ausgezeichnete Endzustand
- Σ ist eine endliche Menge, das **Bandalphabet**
 \square ist ein ausgezeichnetes Symbol in Σ : es heißt **Leersymbol** (Blank) und zeigt an, dass die Bandzelle leer ist
- $Z \cap \Sigma = \emptyset$
- $\delta : Z \times \Sigma \rightarrow Z \times \Sigma \times \{+1, -1, 0\}$ ist eine nicht notwendig überall definierte Funktion, die **Übergangsfunktion**

Definition 0.0.2 (Turing-berechenbar). Eine (partielle) Funktion $f : \Sigma^* \rightarrow \Sigma^*$ heißt **Turingprogramm-berechenbar**, falls eine DTM M existiert mit $f = f_M$.

Definition 0.0.3 (Entscheidbarkeit). Eine Menge $A \subseteq \mathbb{N}$ (bzw. $A \subseteq \mathbb{N}^k$ oder $A \subseteq \Sigma^*$) heißt **entscheidbar**, wenn die charakteristische Funktion $\chi_A : \mathbb{N} \rightarrow \{0, 1\}$ (bzw. $\chi_A : \mathbb{N}^k \rightarrow \{0, 1\}$ oder $\chi_A : \Sigma^* \rightarrow \{0, 1\}$) von A ,

$$\chi_A(x) = \begin{cases} 1 & \text{falls } x \in A \\ 0 & \text{sonst, also falls } x \notin A \end{cases}$$

berechenbar ist.

1 Komplexität

Wie schwierig ist ein lösbares Problem?

1.1 Graphen

Definition 1.1.1. Ein (ungerichteter, einfacher) Graph G ist ein Paar $G = (V, E)$ bestehend aus **Knotenmenge** V und **Kantenmenge** $E \subseteq \binom{V}{2}$. $\binom{V}{2}$ steht hierbei für die Menge aller 2-elementigen Teilmengen von V .

Definition 1.1.2. Sei $G = (V, E)$ ein Graph. Dann gilt:

1. Zwei Knoten $x, y \in V$ sind **verbunden**, wenn $\{x, y\} \in E$ ist.
2. Eine Menge $Q \subseteq V$ von Knoten ist eine **Clique**, wenn je zwei Knoten in Q verbunden sind.
3. Eine Menge $U \subseteq V$ von Knoten ist eine **unabhängige Menge** (independent set), wenn je zwei Knoten in U unverbunden sind.

1.2 Zeitaufwand und Komplexitätsklassen

Definition 1.2.1 (\mathcal{O} -Notation). Für Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}$ schreiben wir $f \in \mathcal{O}(g)$ oder auch $f = \mathcal{O}(g)$, falls es eine Konstante $c > 0$ gibt mit: Es existiert ein n_0 , sodass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$ gilt. Man sagt, dass f asymptotisch höchstens so stark wächst wie g .

Formal: $f \in \mathcal{O}(g) \Leftrightarrow \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

Definition 1.2.2 (Zeitaufwand von DTM-Programmen). Sei M eine DTM über dem Alphabet Σ . Der **Zeitaufwand** $t_M(w)$ von M bei Eingabe $w \in \Sigma^*$ ist

$$t_M(w) = \begin{cases} \text{Zahl der Konfigurationsübergänge der Be-} & \text{falls Berechnung abbricht} \\ \text{rechnung von } M \text{ bei Eingabe } w & \\ \infty & \text{sonst} \end{cases}$$

Der **Zeitaufwand** $t_M(n)$ von M bei Eingaben der Codierungslänge n ist $t_M(n) = \max\{t_M(w) \mid w \in \Sigma^n\}$

Definition 1.2.3. Es sei $f : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion. Mit **DTIME(f)** bezeichnen wir die Menge aller Entscheidungsprobleme, die sich durch eine DTM M mit Zeitaufwand $t_m = \mathcal{O}(f)$ entscheiden lassen:

DTIME(f) = $\{L \subseteq \Sigma^* \mid L \text{ ist entscheidbar durch eine DTM } M \text{ mit } t_m = \mathcal{O}(f)\}$

Definition 1.2.4 (Komplexitätsklasse P). $P = \bigcup_{k=0}^{\infty} \text{DTIME}(n^k)$ ist die Klasse aller in (deterministisch) polynomiell Zeit aufwand lösbaren (Entscheidungs-) Probleme. P ist also die Klasse von Problemen, die effizient gelöst werden können.

Definition 1.2.5 (Komplexitätsklasse EXPTIME). $\text{EXPTIME} = \bigcup_{k=0}^{\infty} \text{DTIME}(2^{n^k})$ ist die Klasse aller in (deterministisch) exponentiellem Zeit aufwand lösbaren (Entscheidungs-) Probleme. (Es gilt: $P \subseteq \text{EXPTIME}$)

Definition 1.2.6 (Nichtdeterministische Turingmaschine). Eine nichtdeterministische Turingmaschine (NTM) (oder ein nichtdet. Turingprogramm) M ist ein 5-Tupel $M = (Z, \Sigma, \delta, z_a, z_e)$ mit:

- Z, z_a, z_e, Σ sind definiert wie bei einer deterministischen Turingmaschine
- $\delta \subseteq (Z \times \Sigma) \times (Z \times \Sigma \times \{+1, -1, 0\})$ ist eine Relation, die Übergangsrelation

In einem nichtdeterministischen Turingprogramm δ kann es zu einem Paar $(z, x) \in Z \times \Sigma$ mehr als einen Befehl mit der linken Seite z, x geben.

Definition 1.2.7 (Nichtdeterministische Berechnung). Jedem Eingabewort $w \in \Sigma^*$ kann man einen „Berechnungsbaum“ zuordnen, dessen maximale Pfade den möglichen Berechnungen entsprechen:

- Die Wurzel des Berechnungsbaums ist mit der Anfangskonfiguration $z_a w$ beschriftet
- Ist v ein Knoten des Baums, der mit der Konfiguration $K = \alpha z x \beta$ markiert ist und ist $\delta(z, x) = \{(z_1, x_1, \lambda_1), \dots, (z_r, x_r, \lambda_r)\}$, so hat v genau r Söhne, die jeweils mit den Nachfolgekonfigurationen K_i von K bezüglich (z_i, x_i, λ_i) beschriftet sind, $i = 1, \dots, r$.

Definition 1.2.8 (Nichtdeterministische Entscheidbarkeit). Eine NTM M entscheidet die Menge $L \subseteq \Sigma^*$, falls der Berechnungsbaum jedes Eingabewortes $w \in \Sigma^*$ endlich ist und für $w \in L$ mindestens einen erfolgreichen Berechnungspfad enthält.

Satz 1.2.1. Sei $L \subseteq \Sigma^*$. Dann ist L genau dann (deterministisch) entscheidbar, wenn L nichtdeterministisch entscheidbar ist.

Definition 1.2.9 (Zeitaufwand von NTM-Programmen). Sei M eine NTM über dem Alphabet Σ . Der Zeitaufwand $t_M(w)$ von M bei Eingabe $w \in \Sigma^*$ ist

$$t_M(w) = \begin{cases} \text{Tiefe des Berechnungsbaumes von } M \text{ bei } w & \text{falls Baum endlich} \\ \infty & \text{sonst} \end{cases}$$

Der Zeitaufwand $t_M(n)$ von M bei Eingaben der Codierungslänge n ist $t_M(n) = \max\{t_M(w) \mid w \in \Sigma^n\}$

Definition 1.2.10. Es sei $f : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion. Mit $\text{NTIME}(f)$ bezeichnen wir die Menge aller Entscheidungsprobleme, die sich durch eine NTM M mit Zeitaufwand $t_m = O(f)$ entscheiden lassen:

$$\text{NTIME}(f) = \{L \subseteq \Sigma^* \mid L \text{ ist entscheidbar durch eine NTM } M \text{ mit } t_m = \mathcal{O}(f)\}$$

Definition 1.2.11 (Komplexitätsklasse NP). $\text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$ ist die Klasse aller in nichtdeterministisch polynomiell Zeit aufzulösenden (Entscheidungs-) Probleme.

Satz 1.2.2. $P \subseteq \text{NP} \subseteq \text{EXPTIME}$

1.3 Polynomielle Reduktion und NP-Vollständigkeit

Definition 1.3.1 (Polynomielle Reduktion). Seien $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ zwei Entscheidungsprobleme. L_1 ist auf L_2 **polynomiell reduzierbar**, wenn es eine überall definierte, polynomiell berechenbare Funktion $f : \Sigma_1^* \rightarrow \Sigma_2^*$ gibt, so dass für alle $x \in \Sigma_1^*$ gilt:

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$

Ist L_1 polynomiell reduzierbar auf L_2 via f , so schreiben wir: $L_1 \leq_p L_2$

Definition 1.3.2 (NP-Vollständigkeit). Es sei $L \subseteq \Sigma^*$. L heißt **NP-vollständig**, falls gilt:

- (i) $L \in \text{NP}$
- (ii) $\forall M \in \text{NP} : M \leq_p L$

Lemma 1.3.1. Ist L NP-vollständig, so gilt: $L \in P \Leftrightarrow P = \text{NP}$

Satz 1.3.1 (Satz von Cook und Levin). SAT ist NP-vollständig.

Satz 1.3.2. Ist A NP-vollständig, $A \leq_p B$ und $B \in \text{NP}$, so ist B NP-vollständig.

Satz 1.3.3. 3-SAT ist NP-vollständig.

Satz 1.3.4. CLIQUE ist NP-vollständig.

Satz 1.3.5. INDSET und VERTEX COVER sind NP-vollständig.

Satz 1.3.6. 3-Färbbarkeit ist NP-vollständig.

Satz 1.3.7. SUBSET-SUM ist NP-vollständig.

Definition 1.3.3 (Komplexitätsklasse coK). Sei K eine Komplexitätsklasse über dem Alphabet Σ . Dann heißt

$$\text{co}K := \{L \subseteq \Sigma^* \mid \bar{L} \in K\}$$

die Klasse der Sprachen (Entscheidungsprobleme) L , deren Komplement \bar{L} in K liegt.

Satz 1.3.8. $\text{co}P = P$

1.4 Platzkomplexität

Definition 1.4.1 (Speicherbedarf von DTM-Programmen). Sei M eine DTM über dem Alphabet Σ . Der **Speicherbedarf** $s_M(w)$ von M bei Eingabe $w \in \Sigma^*$ ist

$$s_M(w) = \begin{cases} \text{Zahl der Bandzellen, die } M \text{ bei Bearbeitung} & \text{falls Berechnung abbricht} \\ \text{von Eingabe } w \text{ besucht} & \\ \infty & \text{sonst} \end{cases}$$

Der **Speicherbedarf** $s_M(n)$ von M bei Eingaben der Codierungslänge n ist $s_M(n) = \max\{s_M(w) \mid w \in \Sigma^n\}$

Definition 1.4.2. Sei $f : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion.

- **DSPACE**(f) ist die Menge aller Entscheidungsprobleme, die sich durch eine DTM M mit Speicherbedarf $s_M = O(f)$ entscheiden lassen.
- **NSPACE**(f) ist die Menge aller Entscheidungsprobleme, die sich durch eine NTM M mit Speicherbedarf $s_M = O(f)$ entscheiden lassen.

Definition 1.4.3 (Platzkomplexitätsklassen PSPACE und NSPACE).

$$\text{PSPACE} := \bigcup_{k=0}^{\infty} \text{DSPACE}(n^k) \quad \text{NSPACE} := \bigcup_{k=0}^{\infty} \text{NSPACE}(n^k)$$

Satz 1.4.1 (Satz von Savitch). $\text{PSPACE} = \text{NSPACE}$

Satz 1.4.2. $P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

1.5 Umgang mit schwierigen Problemen

1.5.1 Betrachtung von Spezialfällen

Manchmal sind nur spezielle Fälle eines schwierigen Problems praktisch relevant. Für diese kann es möglich sein, sie effizient zu lösen.

Beispiel 1.5.1.

SUBSET SUM ist für „super-wachsende“ Eingaben polynomiell lösbar.

1.5.2 Annäherungsverfahren

Man betrachtet die **Optimierungsversion** des zugehörigen Entscheidungsproblems und probiert die Lösung zu approximieren.

Definition 1.5.1. Ein **c-Approximationsalgorithmus** A (mit Güte $c > 1$) ist ein Algorithmus mit:

- Die Laufzeit von A ist polynomiell zur Eingabelänge.

- Die Ausgabe v von A zur Eingabe w ist eine zulässige Lösung.
- $c = \begin{cases} \frac{v}{v^*} & \text{bei Minimierungsproblemen} \\ \frac{v^*}{v} & \text{bei Maximierungsproblemen} \end{cases}$, wobei v^* eine optimale Lösung ist.

1.5.3 Parametrisierte Algorithmen

Komplexe Parameter des Problems werden von der Eingabe getrennt.

1.5.4 Randomisierte Algorithmen

Ein randomisierter Algorithmus hat in seinem Ablauf Zugriff auf eine Quelle von Zufallszahlen.

2 Formale Sprachen

Wie können Sprachen beschrieben werden?

2.1 Grundbegriffe

Definition 2.1.1 (Alphabete, Wörter, Sprachen).

- Ein **Alphabet** Σ ist eine nichtleere, endliche Menge.
- Die Elemente von Σ heißen **Zeichen** (Symbole, Buchstaben) des Alphabets.
- Ein **Wort** w über dem Alphabet Σ ist eine endliche Folge von Zeichen aus Σ .
- Die **Länge** $|w|$ des Wortes w ist die Anzahl der Zeichen in w .
- Das **leere Wort** ε bezeichnet das Wort der Länge 0.
- Σ^* ist die Menge aller Wörter über Σ .
- Eine **formale Sprache** über dem Alphabet Σ ist eine Teilmenge von Σ^* .

Definition 2.1.2 (Grammatik). Eine Grammatik ist ein 4-Tupel $G = (N, \Sigma, R, S)$ mit

- einem endlichem Alphabet N von **Nichtterminalen**,
- einem endlichen Alphabet Σ von **Terminalen**, $N \cap \Sigma = \emptyset$,
- einer endlichen Menge $R \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ von (Produktions-) **Regeln**,
- und einem **Startsymbol** $S \in N$.

Definition 2.1.3. Die von einer Grammatik $G = (N, \Sigma, R, S)$ **erzeugte Sprache** ist $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Definition 2.1.4. Zwei Grammatiken $G_1 = (N_1, \Sigma_1, R_1, S_1)$ und $G_2 = (N_2, \Sigma_2, R_2, S_2)$ heißen **äquivalent**, wenn $L(G_1) = L(G_2)$ gilt.

Satz 2.1.1. $L \subseteq \Sigma^*$ ist aufzählbar \Leftrightarrow Es gibt Grammatik $G = (N, \Sigma, R, S)$ mit $L = L(G)$.

2.2 Chomsky Hierarchie

Definition 2.2.1 (Typ 0: Unbeschränkte Grammatiken).

- Eine Grammatik $G = (N, \Sigma, R, S)$ heißt **unbeschränkt**.
- Eine Sprache L ist **aufzählbar**, falls es eine unbeschränkte Grammatik G gibt mit $L(G) = L$.

Definition 2.2.2 (Typ 1: Kontextsensitive Grammatiken).

- Eine Grammatik $G = (N, \Sigma, R, S)$ heißt **kontextsensitiv**, wenn für jede Regel $u \rightarrow v$ in R gilt: $|u| \leq |v|$, mit der Ausnahme $S \rightarrow \varepsilon$, falls das Startsymbol S auf keiner rechten Seite einer Regel vorkommt.
- Eine Sprache L heißt **kontextsensitiv**, falls es eine kontextsensitive Grammatik G gibt mit $L(G) = L$.

Satz 2.2.1. Kontextsensitive Sprachen sind entscheidbar.

Definition 2.2.3 (Typ 2: Kontextfreie Grammatiken).

- Eine Grammatik $G = (N, \Sigma, R, S)$ heißt **kontextfrei**, wenn für jede Regel $u \rightarrow v$ in R gilt: $u \in N$.
- Eine Sprache L heißt **kontextfrei**, falls es eine kontextfreie Grammatik G gibt mit $L(G) = L$.

Definition 2.2.4 (Typ 3: Reguläre Grammatiken).

- Eine Grammatik $G = (N, \Sigma, R, S)$ heißt **regulär** (oder rechtslinear), wenn für jede Regel $u \rightarrow v$ in R gilt: $u \in N$ und $v \in \{\varepsilon\} \cup \Sigma \cup \Sigma N$.
- Eine Sprache L heißt **regulär**, falls es eine reguläre Grammatik G gibt mit $L(G)$.

Satz 2.2.2 (Chomsky-Hierarchie). $\text{Typ } 3 \subset \text{Typ } 2 \subset \text{Typ } 1 \subset \text{Typ } 0$

2.3 Endliche Automaten

Definition 2.3.1 (Endliche Automaten). Ein **nichtdeterministischer endlicher Automat** (NEA) ist ein 5-Tupel $A = (Z, \Sigma, \delta, z_0, F)$ mit:

- Z endliche Zustandsmenge
- $z_0 \in Z$ **Anfangszustand**
- $F \subseteq Z$ Menge der akzeptierenden Zustände (**Endzustände**)
- Σ endlichens **Eingabealphabet**

- $\delta : Z \times \Sigma \rightarrow 2^Z$ **Überföhrungsrelation**

Ist die Überföhrungsrelation eine Funktion, also $\delta : Z \times \Sigma \rightarrow Z$, so ist der endliche Automat **deterministisch** (DEA).

Definition 2.3.2 (Sprache eines NEA). Sei $A = (Z, \Sigma, \delta, z_0, F)$ ein NEA.

- Ein **Lauf** von A , gesteuert durch Eingabefolge $w = x_0x_1 \dots x_n \in \Sigma^*$ ist **eine** Folge von Zuständen z_0, z_1, \dots, z_{n+1} mit $z_{i+1} \in \delta(z_i, x_i)$, $0 \leq i \leq n$.

Ist $w = \varepsilon$, so besteht der Lauf nur aus dem Anfangszustand z_0 .

- Die von A **akzeptierte Sprache** ist

$$L(A) = \{w \in \Sigma^* \mid \text{es gibt einen Lauf von } A, \text{ gesteuert durch } w, \text{ der zu einem Endzustand föhrt}\}$$

Definition 2.3.3 (Sprache eines DEA). Sei $A = (Z, \Sigma, \delta, z_0, F)$ ein DEA.

- Ein **Lauf** von A , gesteuert durch Eingabefolge $w = x_0x_1 \dots x_n \in \Sigma^*$ ist **die** Folge von Zuständen z_0, z_1, \dots, z_{n+1} mit $z_{i+1} = \delta(z_i, x_i)$, $0 \leq i \leq n$.

Ist $w = \varepsilon$, so besteht der Lauf nur aus dem Anfangszustand z_0 .

- Die von A **akzeptierte Sprache** ist

$$L(A) = \{w \in \Sigma^* \mid \text{der Lauf von } A, \text{ gesteuert durch } w, \text{ föhrt zu einem Endzustand}\}$$

Definition 2.3.4. Zwei endliche Automaten sind **äquivalent**, wenn sie die gleiche Sprache akzeptieren.

Satz 2.3.1. Zu jedem NEA gibt es einen äquivalenten DEA.

Ist $A = (Z, \Sigma, \delta, z_0, F)$ ein NEA, dann ist der Potenzmengenautomat $A' = (2^Z, \Sigma, \delta', \{z_0\}, F')$ mit:

$$\triangleright F' = \{M \subseteq Z \mid M \cap F \neq \emptyset\}$$

$$\triangleright \delta'(M, x) = \bigcup_{z \in M} \delta(z, x)$$

ein DEA mit $L(A) = L(A')$.

Satz 2.3.2. Zu jedem DEA A gibt es eine reguläre Grammatik G mit $L(G) = L(A)$.

Ist $A = (Z, \Sigma, \delta, z_0, F)$ ein DEA, dann ist $G = (Z, \Sigma, R, z_0)$ mit

$$\triangleright \text{ist } z_0 \in F, \text{ so ist } z_0 \rightarrow \varepsilon \in R$$

$$\triangleright \text{ist } \delta(z, x) = z', \text{ so ist } z \rightarrow xz' \in R$$

$$\text{ist } z' \in F, \text{ so ist außerdem } z' \rightarrow \varepsilon \in R$$

eine reguläre Grammatik mit $L(G) = L(A)$.

Satz 2.3.3. Zu jeder regulären Grammatik G gibt es einen NEA A mit $L(A) = L(G)$.

2.4 Reguläre Ausdrücke

Definition 2.4.1 (Syntax regulärer Ausdrücke). Sei Σ ein Alphabet.

1. \emptyset und ε sind reguläre Ausdrücke.
2. Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
3. Sind x und y reguläre Ausdrücke, so sind auch $(x) + (y)$, $(x)(y)$ und $(x)^*$ reguläre Ausdrücke.
4. Weitere reguläre Ausdrücke gibt es nicht.

Definition 2.4.2 (Semantik regulärer Ausdrücke). Sei x ein regulärer Ausdruck über Σ . Die von x beschriebene Sprache $L(x)$ ist:

1. $L(\emptyset) = \emptyset$ und $L(\varepsilon) = \{\varepsilon\}$.
2. Für jedes $a \in \Sigma$ ist $L(a) = \{a\}$.
3. $L(x + y) = L(x) \cup L(y)$, $L(xy) = L(x)L(y)$ und $L(x^*) = L(x)^*$.

Definition 2.4.3. Zwei reguläre Ausdrücke x, y über Σ heißen **äquivalent**, wenn $L(x) = L(y)$ gilt, dass heißt wenn sie die gleiche Sprache beschreiben. Schreibweise: $x \equiv y$.

Satz 2.4.1. Zu jedem regulären Ausdruck x gibt es eine reguläre Grammatik G mit $L(G) = L(x)$.

Satz 2.4.2. Zu jedem DEA A gibt es einen regulären Ausdruck r mit $L(r) = L(A)$.

Satz 2.4.3. $L \subseteq \Sigma^*$ ist regulär genau dann, wenn L aus den Sprachen \emptyset , $\{\varepsilon\}$ und $\{a\}$, $a \in \Sigma$, durch Vereinigung \cup , Konkatenation (Produkt) und Iteration $*$ konstruiert werden kann.

2.5 Reguläre Sprachen

Definition 2.5.1. Ein vollständig definierter DEA A ist **minimal**, falls für alle zu A äquivalenten vollständig definierten DEA A' gilt: A' enthält mindestens soviele Zustände wie A .

Definition 2.5.2 (Nerode-Relation). Sei $L \subseteq \Sigma^*$. Auf der Wortmenge Σ^* ist die Relation \sim_L wie folgt definiert:

$$v \sim_L w \iff (\forall u \in \Sigma^* : vu \in L \iff wu \in L)$$

Die Relation \sim_L ist eine Äquivalenzrelation auf Σ^* . Die Anzahl der Äquivalenzklassen heißt der **Myhill-Nerode-Index** von \sim_L . Schreibweise: $\text{INDEX}(\sim_L)$

Satz 2.5.1 (Myhill & Nerode). L ist regulär $\iff \text{INDEX}(\sim_L)$ ist endlich.

Algorithmus 2.5.1 (Minimierung endlicher Automaten).

Sei auf der Zustandsmenge Z eine Äquivalenzrelation wie folgt definiert:

$$z \equiv z' \iff (\forall u \in \Sigma^* : \delta^*(z, u) \in F \iff \delta^*(z', u) \in F)$$

Dann gilt $z \not\equiv z'$ genau dann, wenn:

- ▷ $z \in F, z' \notin F$ oder umgekehrt, oder
- ▷ es gibt ein $a \in \Sigma$ mit $\delta(z, a) \not\equiv \delta(z', a)$.

Sei $A = (Z, \Sigma, \delta, z_0, F)$ ein DEA ohne überflüssige Zustände.

- (1) Markiere alle Paare $\{z, z'\}$ mit: $z \in F, z' \notin F$ oder umgekehrt
- (2) Für unmarkierte Paare $\{z, z'\}$ prüfe, ob es ein $a \in \Sigma$ gibt mit markiertem Paar $\{\delta(z, a), \delta(z', a)\}$.
Wenn ja, dann markiere auch $\{z, z'\}$.
- (3) Wiederhole Schritt (2) bis kein neues markiertes Paar mehr entsteht.

Dann gilt $z \not\equiv z' \iff \{z, z'\}$ ist markiert. Den gesuchten Minimalautomat $A' = (Z', \Sigma, \delta', z'_0, F')$ erhält man aus A , indem man die \equiv -äquivalenten Zustände zusammen zieht.

- ▷ $Z' = \{[z] : z \in Z\}, z'_0 = [z_0], F' = \{[z] : z \in F\}$
- ▷ $\delta'([z], a) = [\delta(z, a)]$ für alle $[z] \in Z'$ und $a \in \Sigma$

Lemma 2.5.1 (Pumping-Lemma für reguläre Sprachen). Sei L regulär. Dann existiert eine Zahl $n \in \mathbb{N}$, so dass jedes Wort $w \in L$ mit $|w| \geq n$ eine Zerlegung $w = xyz$ mit folgenden Eigenschaften besitzt:

- (1) $|y| \geq 1$
- (2) $|xy| \leq n$
- (3) $xy^kz \in L$ für alle $k = 0, 1, 2, \dots$

Satz 2.5.2 (Abschlusseigenschaften). Die Klasse aller regulären Sprachen über demselben Alphabet ist abgeschlossen unter Vereinigung, Produkt, Stern, Komplement und Schnitt: Sind L und M regulär, so auch $L \cup M, LM, L^*, \bar{L} (= \Sigma^* \setminus L)$ und $L \cap M$ regulär.

Beispiel 2.5.1 (Wichtige Entscheidungsprobleme regulärer Sprachen).

Das Wortproblem

Gegeben: DEA $A = (Z, \Sigma, \delta, z_0, F)$ und $w \in \Sigma^*$.

Frage: Ist $w \in L(A)$?

Äquivalenztest

Gegeben: DEA's $A = (Z, \Sigma, \delta, z_0, F)$ und $A' = (Z', \Sigma, \delta', z'_0, F')$.

Frage: Ist $L(A) = L(A')$?

Leerheitstest

Gegeben: DEA $A = (Z, \Sigma, \delta, z_0, F)$.

Frage: Ist $L(A) = \emptyset$?

Endlichkeitstest

Gegeben: DEA $A = (Z, \Sigma, \delta, z_0, F)$.

Frage: Ist $L(A)$ endlich?

2.6 Kontextfreie Sprachen

Definition 2.6.1. Eine kontextfreie Grammatik ist in **Chomsky-Normalform** (CNF), wenn alle Regeln von der Form $X \rightarrow YZ$ oder $X \rightarrow a$ für $X, Y, Z \in N$ und $a \in \Sigma$ sind.

Lemma 2.6.1 (Pumping-Lemma für kontextfreie Sprachen). Sei L kontextfrei. Dann existiert eine Zahl $n \in \mathbb{N}$, so dass jedes Wort $w \in L$ mit $|w| \geq n$ eine Zerlegung $w = uvxyz$ mit folgenden Eigenschaften besitzt:

- (1) $|vy| \geq 1$
- (2) $|vxy| \leq n$
- (3) $uv^kxy^kz \in L$ für alle $k = 0, 1, 2, \dots$

Satz 2.6.1 (Abschlusseigenschaften). Die Klasse aller kontextfreien Sprachen über demselben Alphabet ist abgeschlossen unter Vereinigung, Produkt und Iteration: Sind L und M kontextfrei, so auch $L \cup M$, LM und L^* kontextfrei.

Die Klasse aller kontextfreien Sprachen ist jedoch nicht abgeschlossen unter Schnitt und Komplement.

Satz 2.6.2. Das Wortproblem für Kontextfreie Sprachen ist in $\mathcal{O}(|w|^3)$ Zeit lösbar, wobei eine kontextfreie Grammatik in CNF gegeben und w das Eingabewort ist.

Definition 2.6.2 (Kellerautomat). Ein **nichtdeterministischer Kellerautomat** (NKA) ist ein 7-Tupel $K = (Z, \Sigma, \Gamma, \delta, z_0, \$, F)$ mit:

- Z endliche Zustandsmenge
- $z_0 \in Z$ Anfangszustand
- $F \subseteq Z$ Menge der akzeptierenden Zustände (Endzustände)
- Σ endliches Eingabealphabet
- Γ endliches Kelleralphabet
- $\$ \in \Gamma$ das Kellerbodensymbol

- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{(Z \cup \Gamma^*)}$ Überführungsrelation

Satz 2.6.3 (Sprache von NKA's).

- 1) Jede von einem NKA akzeptierte Sprache ist kontextfrei.
- 2) Zu jeder kontextfreien Grammatik G existiert ein NKA K mit $L(G) = L_\varepsilon(K)$.

3 Semantik von Programmiersprachen

Welche (rechnerische) Bedeutung hat ein Programm?

3.1 Eine einfache Programmiersprache: W

```
 $\langle cmd\_seq \rangle ::= \langle command \rangle$   
                  |  $\langle cmd\_seq \rangle \text{ ';' } \langle command \rangle$   
  
 $\langle command \rangle ::= \langle identifier \rangle \text{ '=' } \langle expression \rangle$   
                  | 'skip'  
                  | 'while'  $\langle expression \rangle$  'do'  $\langle cmd\_seq \rangle$  'end'  
                  | 'if'  $\langle expression \rangle$  'then'  $\langle cmd\_seq \rangle$  'end'  
                  | 'if'  $\langle expression \rangle$  'then'  $\langle cmd\_seq \rangle$  'else'  $\langle cmd\_seq \rangle$  'end'  
  
 $\langle expression \rangle ::= \langle term \rangle$   
                  |  $\langle expression \rangle \text{ ('+' | '-' | 'OR')} \langle term \rangle$   
  
 $\langle term \rangle ::= \langle factor \rangle$   
                  |  $\langle term \rangle \text{ ('*' | '/' | 'AND')} \langle factor \rangle$   
  
 $\langle factor \rangle ::= \langle number \rangle \mid \langle identifier \rangle$   
                  | 'true' | 'false' | 'NOT'  $\langle factor \rangle$   
                  | '('  $\langle expression \rangle$  ')'  
                  |  $\langle expression \rangle \text{ ('<' | '>' | '<=' | '>=' | '<>' | '=')} \langle expression \rangle$ 
```

3.2 Operationelle Semantik

Wie entsteht der Effekt eines Programms?

Die Bedeutung eines Programms wird angegeben als Sequenz von Schritten, die Zustandstransformationen eines abstrakten Maschinenmodells beschreiben. Über Axiome und Schlussregeln kann so das Verhalten des Programms aus dem Verhalten seiner Komponenten hergeleitet werden. Formale operationelle Semantik ermöglicht die Argumentation über Terminierung, semantische Äquivalenz, Determiniertheit und Korrektheit einer Übersetzung.

Anwendung: Vergleich von Systemen, Model Checking

3.2.1 Natural Semantics (Big Step)

Jedes Programm ist eine Zustandstransformation.

$\langle C, s \rangle \rightarrow s'$ Gestartet in Zustand s terminiert Programm C und führt zu Zustand s'
Geeignet für Blockstrukturen.

$$\begin{array}{l}
[\text{skip}_{\text{NatS}}] \quad \langle \text{skip}, s \rangle \rightarrow s \\
[\text{ass}_{\text{NatS}}] \quad \frac{\langle x := E, s \rangle \rightarrow s[x \rightarrow \mathbb{A}[\![E]\!](s)]}{\langle C_1, s \rangle \rightarrow s' \quad \langle C_2, s' \rangle \rightarrow s''} \\
[\text{comp}_{\text{NatS}}] \quad \frac{}{\langle C_1; C_2, s \rangle \rightarrow s''} \\
[\text{if}_{\text{NatS}}^1] \quad \frac{\langle C_1, s \rangle \rightarrow s' \quad \mathbb{B}[\![b]\!](s) = T}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ end}, s \rangle \rightarrow s'} \\
[\text{if}_{\text{NatS}}^2] \quad \frac{\langle C_2, s \rangle \rightarrow s' \quad \mathbb{B}[\![b]\!](s) = F}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ end}, s \rangle \rightarrow s'} \\
[\text{while}_{\text{NatS}}^1] \quad \frac{\langle C, s \rangle \rightarrow s' \quad \langle \text{while } b \text{ do } C \text{ end}, s' \rangle \rightarrow s'' \quad \mathbb{B}[\![b]\!](s) = T}{\langle \text{while } b \text{ do } C \text{ end}, s \rangle \rightarrow s''} \\
[\text{while}_{\text{NatS}}^2] \quad \frac{\mathbb{B}[\![b]\!](s) = F}{\langle \text{while } b \text{ do } C \text{ end}, s \rangle \rightarrow s}
\end{array}$$

$$\mathbb{S}_{\text{NatS}}[\![C]\!](s) := \begin{cases} s' & \langle C, s \rangle \rightarrow s' \\ \perp & \text{sonst} \end{cases}$$

3.2.2 Structural Operational Semantics (Small Step)

Jedes Programm ist eine Folge elementarer Zustandstransformation.

$\langle C, s \rangle \rightarrow s'$ Gestartet in Zustand s terminiert Programm C und führt zu Zustand s'

$\langle C, s \rangle \rightarrow \langle C', s' \rangle$ Die Ausführung von Programm C im Zustand s kann auf die Ausführung von Programm C' im Zustand s' reduziert werden

Geeignet für Nichtdeterminismus und Parallelität.

$$\begin{array}{l}
[\text{skip}_{\text{SOS}}] \quad \langle \text{skip}, s \rangle \rightarrow s \\
[\text{ass}_{\text{SOS}}] \quad \langle x := E, s \rangle \rightarrow s[x \rightarrow \mathbb{A}[[E]](s)] \\
[\text{comp}_{\text{SOS}}^1] \quad \frac{\langle C_1, s \rangle \rightarrow s'}{\langle C_1; C_2, s \rangle \rightarrow \langle C_2, s' \rangle} \\
[\text{comp}_{\text{SOS}}^2] \quad \frac{\langle C_1, s \rangle \rightarrow \langle C'_1, s' \rangle}{\langle C_1; C_2, s \rangle \rightarrow \langle C'_1; C_2, s' \rangle} \\
[\text{if}_{\text{SOS}}^1] \quad \frac{\mathbb{B}[[b]](s) = T}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ end}, s \rangle \rightarrow \langle C_1, s \rangle} \\
[\text{if}_{\text{SOS}}^2] \quad \frac{\mathbb{B}[[b]](s) = F}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ end}, s \rangle \rightarrow \langle C_2, s \rangle} \\
[\text{while}_{\text{SOS}}] \quad \frac{\langle \text{while } b \text{ do } C \text{ end}, s \rangle \rightarrow \langle \text{if } b \text{ then } C; \text{ while } b \text{ do } C \text{ end else skip end}, s \rangle}{\langle \text{while } b \text{ do } C \text{ end}, s \rangle \rightarrow \langle \text{if } b \text{ then } C; \text{ while } b \text{ do } C \text{ end else skip end}, s \rangle}
\end{array}$$

$$\mathbb{S}_{\text{SOS}}[[C]](s) := \begin{cases} s' & \langle C, s \rangle \rightarrow s' \\ \perp & \text{sonst} \end{cases}$$

3.3 Denotationelle Semantik

Was ist der Effekt eines Programms?

Die Bedeutung eines Programms wird angegeben durch mathematische Formalismen wie Funktionen, Relationen und Gleichungen.

Anwendung: Statische Analyse (erhalten von Informationen über die Semantik eines Programms ohne dessen Ausführung)

3.3.1 Direct-Style-Semantik

- **kompositional:** Semantik eines Konstrukts allein auf der Basis seiner Teilkonstrukte definiert
- Kleinste **Fixpunkte** werden zur Definition der Semantik von Schleifen verwendet
- Fixpunkttheorie liefert Existenz solcher Fixpunkte

$$\begin{aligned}
\mathbb{S}_{\text{DS}}[x := E](s) &= s[x \rightarrow \mathbb{A}[E](s)] \\
\mathbb{S}_{\text{DS}}[\text{skip}] &= \text{id} \\
\mathbb{S}_{\text{DS}}[C_1; C_2] &= \mathbb{S}_{\text{DS}}[C_2] \circ \mathbb{S}_{\text{DS}}[C_1] \\
\mathbb{S}_{\text{DS}}[\text{if } b \text{ then } C_1 \text{ else } C_2 \text{ end}] &= \text{cond}(\mathbb{B}[b], \mathbb{S}_{\text{DS}}[C_1], \mathbb{S}_{\text{DS}}[C_2]) \\
\mathbb{S}_{\text{DS}}[\text{while } b \text{ do } C \text{ end}] &= \text{FIX}(F) \\
&\text{wobei } F(g) = \text{cond}(\mathbb{B}[b], g \circ \mathbb{S}_{\text{DS}}[C], \text{id})
\end{aligned}$$

3.3.2 Flussgraph-Analyse

Definition 3.3.1 (Available Expressions). Zu einem Knoten des Flussgraphen, bestimme alle Expressions, die auf allen Pfaden bereits berechnet sind und seitdem nicht modifiziert wurden.

Definition 3.3.2 (Reaching Definitions). Für einen Knoten k : Welche Zuweisungen sind auf mindestens einem Pfad zu k noch nicht überschrieben?

Definition 3.3.3 (Very busy expressions). Eine Expression ist very busy an einem Knoten, falls ihr Wert auf jedem Kontrollpfad noch einmal benutzt wird (ohne dass vorkommende Variablen ihren Wert ändern). Very busy expressions können gleich berechnet werden.

Definition 3.3.4 (Live Variables). Variable ist live an einem Knoten, falls ihr Wert auf mindestens einem Kontrollpfad noch einmal benutzt wird (ohne dass ihr Wert überschrieben wurde). Zuweisungen an Variablen, die nicht live sind, können gestrichen werden.

3.4 Axiomatische Semantik

Was lässt sich über den Effekt eines Programms aussagen?

Die Bedeutung eines Programms wird angegeben durch Logische Aussagen und Beweiskalküle.

Anwendung: Programmverifikation

3.4.1 Hoare-Kalkül

Definition 3.4.1 (Hoare-Tripel). $\{P\} S \{Q\}$ heißt Hoare-Tripel, wobei P die Vorbedingungen, S ein Programmsegment und Q die Nachbedingungen bezeichnet.

- Partielle Korrektheit
Wenn S terminiert und vor Abarbeitung von S die Aussage P gilt, so gilt nach der Abarbeitung von S die Aussage Q .

- Totale Korrektheit

Wenn vor Abarbeitung von S die Aussage P gilt, dann terminiert S und nach der Abarbeitung von S gilt die Aussage Q .

$[\text{skip}_{\text{par}}]$	$\{P\} \text{ skip } \{P\}$
$[\text{ass}_{\text{par}}]$	$\{P[x \rightarrow \mathbb{A}[\![E]\!]]\} x := E \{P\}$
$[\text{comp}_{\text{par}}]$	$\frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$
$[\text{if}_{\text{par}}]$	$\frac{\{P \wedge \mathbb{B}[\![b]\!]\} S_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}[\![b]\!]\} S_1 \{Q\}}{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \text{ end } \{Q\}}$
$[\text{while}_{\text{par}}]$	$\frac{\{I \wedge \mathbb{B}[\![b]\!]\} S \{I\}}{\{I\} \text{ while } b \text{ do } S \text{ end } \{I \wedge \neg \mathbb{B}[\![b]\!]\}}$
$[\text{cons}_{\text{par}}]$	$\frac{P' \Rightarrow P \quad \{P\} S \{Q\} \quad Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$

4 Formale Systeme

Wie kann man Systeme modellieren?

4.1 Prozessalgebra

Definition 4.1.1 (Prozessalgebra). Prozessalgebren sind Formalismen zur Beschreibung und zum Studium nebenläufiger/interaktiver Systeme.

1. Es gibt Prozesse und Aktionen.
2. Ein Prozess kann Aktionen ausführen.
3. Eine Aktion ändert den ausführenden Prozess.
4. τ bezeichnet eine interne, nach außen nicht sichtbare Aktion.

<i>definition</i>	$P := Q$	P verhält sich wie Q
<i>action</i>	$P := a.Q$	In P kann Aktion a stattfinden, danach verhält sich P wie Q
<i>choice</i>	$P := Q + R$	P kann sich wie Q oder wie R verhalten
<i>termination</i>	$P := 0$	In P findet keine Aktion mehr statt
<i>parallel</i>	$P := Q \mid R$	R und Q arbeiten parallel
<i>restriction</i>	$P := Q \setminus \{a_1, \dots, a_n\}$	P verhält sich wie Q ohne die Aktionen a_1, \dots, a_n
<i>relabeling</i>	$P := Q[a_1 \leftarrow a'_1, \dots, a_n \leftarrow a'_n]$	P verhält sich wie Q , wobei einige Aktionen umbenannt wurden

- **CSP (Communicating Sequential Processes)**

$?a.P$ Lies ein a aus der Umgebung und verhalte Dich weiter wie P

$!a.P$ Schreibe ein a an die Umgebung und verhalte Dich weiter wie P

- **CCS (Calculus of Communicating Systems)**

Zu jeder Aktion a wird die Existenz einer Komplementäraktion \bar{a} angenommen

Kommunikation ist Synchronisation von a und \bar{a}

Definition 4.1.2 (Trace Semantik). Eine [Trace](#) vom Prozess P bezeichnet ein Wort w von Aktionen mit $P - w \rightarrow Q$.

$$T(P) := \text{Menge aller Traces von } P \qquad P =_T Q \Leftrightarrow T(P) = T(Q)$$

Definition 4.1.3 (Completed Trace Semantik). Eine Trace w vom Prozess P heißt **vollständig**, falls $P - w \rightarrow Q$ und $Q = 0$.

$$\begin{aligned} \text{CT}(P) &:= \text{Menge aller vollständigen Traces von } P \\ P =_{\text{CT}} Q &\Leftrightarrow T(P) = T(Q) \wedge \text{CT}(P) = \text{CT}(Q) \end{aligned}$$

Definition 4.1.4 (Failure Semantik). (w, M) heißt **failure pair** vom Prozess P , falls $P - w \rightarrow Q$ und keine Aktion $a \in M$ ist in Q ausführbar.

$$\text{FT}(P) := \text{Menge aller failure pairs von } P$$

Definition 4.1.5 (Simulation Semantik). P simuliert Q , falls Q für jeden Schritt, den P macht, einen passenden Schritt ausführen kann.

$$P - a \rightarrow P' \wedge P s Q \Rightarrow \exists Q' : Q - a \rightarrow Q' \wedge P' s Q'$$

4.2 Statecharts

Definition 4.2.1 (Mealy-Automat). Ein Mealy-Automat ist ein 6-Tupel $A = (Z, \Sigma, \Omega, \delta, \lambda, z_0)$ mit:

- Z endliche Zustandsmenge
- Σ endliches Eingabealphabet
- Ω endliches Ausgabealphabet
- $\delta : Z \times \Sigma \rightarrow Z$ Übergangsfunktion
- $\lambda : Z \times \Sigma \rightarrow \Omega$ Ausgabefunktion
- z_0 Startzustand

Definition 4.2.2 (Statecharts). Statecharts oder Harel-Automaten zur Modellierung reaktiver Systeme erweitern Mealy-Automaten um

1. bedingte Zustandsübergänge: Ereignis[Bedingung]/Aktivität
2. hierarchische Zustände
3. Gedächtnis
4. Nebenläufigkeit

4.3 Temporal Logic of Actions (TLA)

4.4 Petri-Netze

Definition 4.4.1 (Petri-Netz). Ein **Petri-Netz** ist ein 5-Tupel $N = (P, T, F, W, m_0)$ mit:

- P ist eine endliche Menge von **Stellen** (*Places*)
- T ist eine endliche Menge von **Transitionen**
- $P \cap T = \emptyset$
- $P \cup T$ ist die Menge der **Knoten**
- $F \subseteq (P \times T) \cup (T \times P)$ ist eine endliche Menge von **Bögen** (*flow relations*)
- $W : F \rightarrow \mathbb{N} \setminus \{0\}$ sind die **Bogengewichte** (*Weights*)
- $m_0 : P \rightarrow \mathbb{N}$ ist die **Anfangsmarkierung**

Definition 4.4.2 (Bereiche von Knoten).

- (a) Der **Vorbereich** eines Knotens x ist $\bullet x = \{y \mid (y, x) \in F\}$.
- (b) Der **Nachbereich** eines Knotens x ist $x\bullet = \{y \mid (x, y) \in F\}$.