Theoretische Informatik

Vorlesungsskript

Mitschrift von Falk-Jonatan Strube

Vorlesung von Dr. Boris Hollas

10. Mai 2016

Inhaltsverzeichnis

Inhalte

Grundlage: Grundkurs Theoretische Informatik [1]

- Formale Sprachen
  - Reguläre Sprachen
    - Endliche Automaten
    - Reguläre Ausdrücke
  - Nichtreguläre Sprachen
  - Kontextfreie Sprachen
    - Kellerautomaten
    - Grammatiken
- Berechenbarkeit
  - Halteproblem
- Komplexitätsklassen
  - P
  - NP
  - NP-vollständige Probleme

1 Automaten und Formale Sprachen

Def.: Ein Alphabet ist eine Menge $\Sigma \neq \emptyset$ (Symbole in Σ – müssen nicht einzelne Buchstaben sein, auch Wörter usw. [bspw. „if“ oder „else“ im Alphabet der Programmiersprache C]).

Def.: Für $w_1, ..., w_n \in \Sigma$ ist $w = w_1...w_n$ ein Wort der Länge n.
Σ^n beschreibt alle Worte mit der Länge genau n
Das Wort ε ist das *leere Wort*.
Die Menge aller Wörter bezeichnen wir mit Σ^* (einschließlich dem leeren Wort).

Bsp.: $\Sigma = \{a, b, c\} \quad \rightarrow \Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, aaa, ...\}$

Def.: Für Wörter $a, b \in \Sigma^*$ ist ab die Konkatenation dieser Wörter.
Für ein Wort w ist w^n die n-fache Konkatenation von w, wobei $w^0 = \varepsilon$.

Bemerkung: Für alle $w \in \Sigma^*$ gilt $\varepsilon w = w = w\varepsilon$. ε ist also das neutrale Element der Konkatenation.

Def.: Eine *formale Sprache* ist eine Teilmenge von σ^*.

Def.: Für Sprachen A, B ist $AB = \{ab \mid a \in A,\ b \in B\}$ sowie $A^n = \prod_{i=1}^{n} A$, wobei $A^0 = \{\varepsilon\}$.

Bemerkung: $\emptyset, \varepsilon, \{\varepsilon\}$ sind unterschiedliche Dinge (leere Menge, leeres Wort, Menge mit leerem Wort).

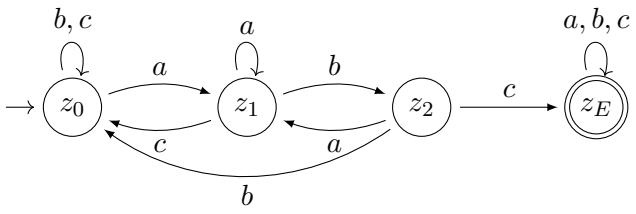
Bemerkung: Σ^* lässt sich ebenfalls definieren durch $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$.

Ferner ist $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

1.1 Reguläre Sprachen

1.1.1 Deterministische endliche Automaten (DFA)

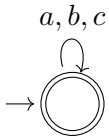
Bsp.:



(Pfeil zeigt auf Startzustand, Endzustand ist doppelt umrandet)

Dieser DFA akzeptiert alle Wörter über $\Sigma = \{a, b, c\}$, die abc enthalten.
Deterministisch: Es gibt genau ein Folgezustand. Von jedem Knoten aus gibt es genau eine Kante für jedes Zeichen, nicht mehrere und nicht keine.

Bsp.:



Dieser DFA erkennt die Sprache $\{a, b, c\}^*$.

Def.: Ein DFA ist ein Tupel $\mathcal{M} = (Z, \Sigma, \delta, z_0, E)$

- Z: Menge der Zustände
- Σ: Eingabealphabet
- δ: Überführungsfunktion $Z \times \Sigma \to Z$. Dabei bedeutet $\delta(z, a) = z'$, dass $\mathcal{M}$ im Zustand z für das Zeichen a in den Zustand z' wechselt.
- $z_0 \in Z$: Startzustand
- E: Menge der Endzustände

δ:

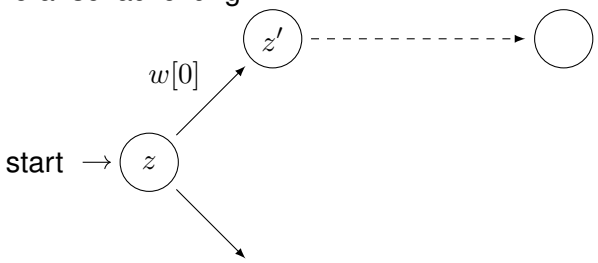
Def.: Die erweiterte Überführungsfunktion $\hat{\delta} : Z \times \Sigma^* \to Z$ ist definiert durch

$$\hat{\delta}(z, w) = \begin{cases} z & \text{für } w = \varepsilon \\ \hat{\delta}(\delta(z, a), x) & \text{für } w = ax \text{ mit } a \in \Sigma, x \in \Sigma^* \end{cases}$$

Dazu vergleichbarer C-Code:

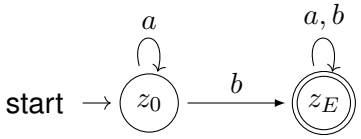
```
int δ̂(int z, char* w){
  if ( strlen(w) == 0 )
    return z;
  else
    return δ̂(δ(z, w[0]), w[1]);
```

Veranschaulichung:



Die erweiterte Überführungsfunktion bestimmt den Zustand nach dem vollständingen Lesen eines Wortes.

Bsp.:



$$\hat{\delta}(z_0, aaba) = \hat{\delta}(\delta(z_0, a), aba) =$$
$$\hat{\delta}(z_0, aba) = \hat{\delta}(\delta(z_0, a), ba) =$$
$$\hat{\delta}(z_0, ba) = \hat{\delta}(\delta(z_0, b), a) =$$
$$\hat{\delta}(z_E, a) = \hat{\delta}(\delta(z_E, a), \varepsilon) =$$
$$\hat{\delta}(z_E, \varepsilon) = z_E$$

Die von $\mathcal{M}$ *akzeptierte Sprache* ist $L(\mathcal{M}) = \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$

1.1.2 Nichtdeterministischer endliche Automaten (NFA)

ABB23
NFA, der alles akzeptiert, was abc enthält:
ABB24
Beispiel: Wort $abaabcab$

Beispiel: NFA, der alle Wörter akzeptiert, die auf 001 enden:
ABB25
Akzeptierte Worte unter anderem: 01011001, 001001
Ein Wort wird vom NFA akzeptiert, wenn es einen Weg, ausgehend von einem Startzustand, gibt, mit den ein End-Zustand erreicht wird.
Der NFA „weiß“ nicht, welcher Pfad zu durchlaufen ist; diesen muss der Benutzer ermitteln (wie bei einer Straßenkarte).
Ein NFA lässt sich formalisieren durch ein Tupel $\mathcal{M} = (Z, \Sigma, \delta, S, E)$

- Z: Zustände
- Σ: Eingabealphabet
- δ: $Z \times \Sigma \to \mathcal{P}(Z)$ Überführungsfunktion (bildet ab in Potenzmenge von Z)
- S: Menge der Startzustände
- E: Menge der Endzustände

Dabei bedeutet $\delta(z, a) \ni z'$, dass der NEA im Zustand z für die Eingabe a die Möglichkeit besitzt, in den Zustand z' zu wechseln.

1.1.3 Umwandlung eines NFA in einen DFA

Wir wollen den NFA
ABB 27
in einen DFA umwandeln. Der Startzustand des DFA besteht aus den Startzuständen des NFA:
ABB 28
Betrachten die Folgezustände für $a \in \Sigma$:
ABB 29

nächster Schritt:
ABB 30
nächster Schritt:
ABB 31
weitere Schritte:
$z_0, b : \{z_0\}$
$z_1, b : \{z_2\}$
ABB 32
$z_0, c : \{z_0\}$
$z_1, c : \{\}$
ABB 33
usw.:
ABB 34
Wenn ein Zustand des DFA einen Endzustand des NFA enthält, so ist es ein Endzustand.
Der auf diese Weise erhaltene DFA kann Zustände enthalten, die sich zu einem Zustand zusammen fassen lassen. Mit dem Algorithmus Minimalautomat lässt sich ein DFA konstruieren, der minimal bezüglich der Anzahl seiner Zustände ist. Der Minimalautomat ist eindeutig, d.h. Minimalautomaten unterscheiden sich höchstens in der Benennung der Zustände.

1.1.4 Reguläre Ausdrücke

Def.: Sei Σ ein Alphabet. Ein *regulärer Ausdruck* E sowie die durch E *erzeugte Sprache* $L(E)$ sind induktiv definiert:

- $\emptyset$ ist ein regulärer Ausdruck und $L(\emptyset) = \emptyset$.
  Bsp.:
  ABB35
- Für $a \in \Sigma \cup \{\varepsilon\}$ ist a ein regulärer Ausdruck und $L(a) = \{a\}$.
- Für reguläre Ausdrücke E_1, E_2 sind $(E_1|E_2),\ (E_1E_2),\ (E_1^*)$ reguläre Ausdrücke (hier: $|$ =„oder“) und $L(E_1|E_2) = L(E_1) \cup L(E_2),\ L(E_1E_2) = L(E_1)L(E_2),\ L(E_1^*) = L(E_1)^*$ die davon erzeugten Sprachen:

Ausdruck	Sprache	
$E_1\|E_2$	$L(E_1\|E_2)$	$= L(E_1) \cup L(E_2)$
E_1E_2	$L(E_1E_2)$	$= L(E_1)L(E_2)$
E_1^*	$L(E_1^*)$	$= L(E_1)^*$

  Hinweis: $E^+ = EE^*$, $E? = \varepsilon|E$

  Wenn E_1, E_2 regulär, dann auch $(E_1|E_2),\ (E_1E_2),\ (E_1^*)$ regulär

Bsp.:

- $L((0|1)^*) = (L(0|1))^* = (L(0) \cup L(1))^* = (\{0\} \cup \{1\})^* = \{0,1\}^*$
- Regulärer Ausdruck über $\Sigma = \{a, b, c\}$, der die gleiche Sprache erzeugt wie der DFA aus dem letzten Automaten-Beispiel:
  $L((a|b|c)^*abc(a|b|c)^*) = \{a, b, c\}^*\{abc\}\{a, b, c\}^*$

Satz: Reguläre Ausdrücke erzeugen genau die regulären Sprachen.

Mitschrift von
Falk-Jonatan Strube

Skizze: Umwandlung eines regulären Ausdrucks in einen endlichen Automaten.

- $\emptyset$: ABB 40
- $a \in \Sigma$: ABB 41 1.
  ε: ABB 41 2.
- Seien E_1, E_2 reguläre Ausdrücke und $\mathcal{M}_1, \mathcal{M}_2$ DFAs mit $L(E_1) = L(\mathcal{M}_1), L(E_2) = L(\mathcal{M}_2)$.
  - $E_1|E_2$: $\mathcal{M}_1, \mathcal{M}_2$ sind zusammen ein NFA, der $L(\mathcal{M}_1) \vee L(\mathcal{M}_2)$ erkennt.
    Bsp.: $E_1 = a, E_2 = b$
    ABB 42
  - E_1E_2: $\mathcal{M}_1, \mathcal{M}_2$ müssen hintereinander geschaltet werden, wobei ggf. neue Kanten eingefügt werden müssen. Dazu betrachtet man die Kante nach der neuen Verbindung und erzeugt dem entsprechend die Übergangskanten.
    ABB 43
  - E_1^*: Es müssen Kanten zurück zum Startzustand eingefügt werden
    ABB 44

Der Beweis für die umgekehrte Richtung (DFA $\rightarrow$ reg. Ausdruck) ist schwierig.

Bsp.:

- $E = 0(0|1)^*$
  ABB 45
- ABB 46
  Beobachtungen:
  - um zum Endzustand zu kommen, braucht man eine 1.
  - vor der 1 kann ε stehen, oder beliebig viele 0en der 1en.

  $\Rightarrow E = (0|1)^*1$

1.1.5 Das Pumping-Lemma

Wenn ein DFA ein Wort akzeptiert, das mindestens so lang ist wie die Anzahl seiner Zustände, dann muss er einen Zustand zweimal durchlaufen (Schubfachprinzip). Daraus folgt, dass der DFA dabei eine Schleife durchläuft.

Bsp.:
ABB 47
Für $x = abcdecfg$ durchläuft der Automat eine Schleife: $x = ab\boxed{cde}cfg$. Daher akzeptiert der DFA auch alle Wärter $ab(cde)^kcfg$ für $k \geq 0$.

Satz: (Pumping Lemma)
Für jede reguläre Sprache L gibt es ein $n > 0$ (n: Anzahl Zustände des Minimalautomaten), so dass es für alle Wörter $x \in L$ mit $|x| \geq n$ eine Zerlegung $x = uvw$ gibt (in vorherigem Bsp.: $u = ab,\ v = cde\ w = cfg$), so dass gilt:

1.) $|v| \geq 1$

2.) $|uv| \leq n$ (u, w können auch ε sein)

3.) $uv^kw \in L$ für alle $k \geq 0$.

Ohne Einschränkung ist n die Anzahl Zustände des Minimalautomaten.
$\Rightarrow \forall$ regulären Sprachen $L \quad \exists\, n > 0 \quad \forall\, x \in L,\ |x| \geq n \quad \exists\, u, v, w$ mit $x = uvw$ und $|v| \geq 1, |uv| \geq n \quad \forall\, k \geq 0\ uv^k w \in L$.
Das Pumping-Lemma lässt sich nutzen, um zu zeigen, dass eine Sprache nicht regulär ist.

Bsp.: Wir zeigen, dass $L = \{a^n b^n | n \in \mathbb{N}\}$ nicht regulär ist.
Problemstellung: Der Automat kann sich das n nicht „merken“, um nach n as wieder n bs zu erzeugen.
Beweis (Widerspruch):

- Angenommen, L sei regulär.
- Nach Pumping-Lemma gibt es dann ein $n > 0$, so dass sich alle $x \in L$ mit $|x| \geq n$ gemäß Pumping-Lemma zerlegen lassen.
- Sei $x = a^n b^n$.
- Angenommen v enthalte ein b, dann wäre $|uv| > n$.
  Aus $|uv| \leq n$ folgt aber, dass v kein b enthält. aus $|v| \geq 1$ folgt, dass v mindestens ein a enthält.
  ABB 48
- Das Wort uw enthält daher weniger as als bs und kann somit nicht in L enthalten sein (denn w enthält b^n, da v mindestens ein a enthält, ist durch uw mindestens ein a „verloren gegangen“: $uw = a^{n-|v|}b^n$) und ist deshalb nicht in L enthälten, Widerspruch ↯ #

Vorgehen:

- ist regulär
- Def. Pumping Lemma
- x finden (gilt für alle x, also ein günstiges x aussuchen, mit dem sich Beweis führen lässt)
- durch 1.) und/oder 2.) einschränken
- durch 3.) zum Widerspruch führen

Bsp.: Wir zeigen, dass $L = \{zz | z \in \{a, b\}^*\}$ nicht regulär ist.
Intuitiver Hinweis: Kann nicht regulär sein, da sich der Automat nicht merken kann, wie viele as und bs im ersten z gelesen wurden, um dann das gleiche im zweiten z zu fabrizieren.
Beweis:

- Angenommen, L ist regulär.
- Nach Pumping-Lemma gibt es ein $n > 0$, so dass sich alle $x \in L$ mit $|x| \geq n$ zerlegen lassen gemäß Pumping-Lemma.
- Sei $x = a^n b a^n b$.
- Wegen $|uv| \leq n$ und $|v| \geq 1$ besteht v aus mindestens einem a.
  ABB 61
- Dann enthält $uw = a^{n-|v|}ba^n b$ (für $k = 0$) weniger as in der vorderen Hälfte als in der hinteren Hälfte. Da sich uw deshalb nicht in die Form zz mit $z \in \{a, b\}^*$ brigen lässt, ist $uw \notin L$, Widerspruch!

Satz: Seien L regulär und n die Anzahl Zustände des Minimalautomaten zu L. Dann gilt $|L| = \infty$ genau dann, wenn es ein $x \in L$ gibt mit $n \leq |x| < 2n$.
Beweis:
($\Leftarrow$):
Gemäß Pumping Lemma gibt es eine Zerlegung $x = uvw$ mit $|v| \geq 1$ und $uv^k w \in L$ für alle $k \in \mathbb{N}_0$ ($\mathbb{N}$ ist unendlich).
Daraus folg $|L| = \infty$.
($\Rightarrow$):
Da es nur endlich viele Wörter x mit $|x| < n$ gibt, gibt es ein $x \in L$ mit $|x| \geq n$.
Sei daher $x \in L$ mit $|x| \geq n$ und $|x|$ minimal.
Gemäß PL lässt sich x zerlegen in $x = uvw$.
Da $uw \in L$ und $|x|$ minimal ist, gilt $|uw| < n$.
Wegen $|x| \geq \underbrace{|uv|}_{<n \text{ gemäß PL}} + \underbrace{|uw|}_{<n \text{ Satz zuvor}} < n + n = 2n$ folgt die Behauptung $n \leq |x| \leq 2n$.
Regulärer Ausdruck: Generator
Automat: Validator

1.2 Kontextfreie Sprachen

1.2.1 Kellerautomaten (PDA)

Ein Kellerautomat (Pushdown Automaton, PDA) besitzt gegenüber einem NFA zwei zusätzliche Eigenschaften:

- Es gibt ε-Übergänge.
- Er besitzt einen Stack, auf dem Zeichen abgelegt oder von dem Zeichen gelesen werden können.

Zur graphischen Darstellung von PDAs verwenden wir eine erweiterte Automatennotation:
ABB62
Unten auf dem Stack liegt das Symbol #. Dies ist das einzige Symbol, das sich zu Beginn einer Rechnung auf dem Stack befindet.

Bsp.: PDA, der $\{a^n b^n | n \in \mathbb{N}\}$ akzeptiert.
ABB63
Wir erlauben nun, dass der PDA in einem Schritt auch mehrere Zeichen auf den Stack schreibt. Dazu erweitern wir die graphische Notation wie folgt:
ABB 68

Def.: Ein PDA ist ein Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$

- Z: Zustände
- Σ: Eingabealphabet
- Γ: Stackalphabet
- δ: $Z \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Z \times \Gamma_\varepsilon)$, wobei $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$
- $z_0 \in Z$: Startzustand
- $\# \in \Gamma$: Unterstes Stackzeichen

- $E \in Z$: Endzustände

ABB 69
$a \in \Sigma \cup \{\varepsilon\}$
$\gamma \in \Gamma \cup \{\varepsilon\}$
$\gamma' \in \Gamma \cup \{\varepsilon\}$

Def.: Die von einem PDA M akzeptierte Sprache $L(M)$ ist die Menge aller $x \in \Sigma^*$, für die gilt: Der PDA M kann, ausgehend vom Startzustand und dem initialen Stackzustand $\#$, durch das Lesen des Wortes x einen Endzustand erreichen.

1.2.2 Kontextfreie Grammatiken

Eine kontextfreie Grammatik beschreibt, wie durch das Ersetzen von variablen Wörter der Sprache erzeugt werden können. Jede Ersetzungsregel hat die Form „linke Seite $\rightarrow$ rechte Seite“ (linke Seite der Regel kann ersetzt werden durch die rechte Seite), wobei „linke Seite“ eine Variable ist.

Beginnend mit dem Startsymbol werden solange Ersetzungsregeln angewendet, bis alle Variablen durch Terminalsymbole (Elemente aus Σ) ersetzt wurden.

Bsp.:

- Satz $\rightarrow$ NP VP*
- NP $\rightarrow$ Artikel Nomen
- Artikel $\rightarrow$ die
- Nomen $\rightarrow$ Katze
- Nomen $\rightarrow$ Maus
- VP $\rightarrow$ Verb NP
- Verb $\rightarrow$ jagt

Satz $\Rightarrow$ NP VP $\Rightarrow$ Artikel Nomen VP $\Rightarrow$ Artikel Nomen Verb NP $\Rightarrow$ Artikel Nomen Verb Artikel Nomen $\Rightarrow \ldots \Rightarrow$ die Katze jagt die Maus
Syntax dazu:
ABB 71

Def.: Eine kontextfreie Grammatik ist ein Tupel $\sigma = (V, \Sigma, P, S)$

- V: Endliche Menge der Variablen oder Nonterminalzeichen
- Σ: Alphabet oder Terminalzeichen $V \cap \Sigma = \emptyset$
- P: Regeln oder Produktionen der Form $u \rightarrow v$ mit $u \in V$ und $v \in (V \cup \Sigma)^*$
- $S \in V$

Für $x, y \in (V \cup \Sigma)^*$ schreiben wir $x \Rightarrow y$, wenn sich durch das Ersetzen einer Variablen in x die Satzform y erzeugen lässt.

*Nominalphase, Verbalphase

Bsp.: die Nomen Verb $\Rightarrow$ die Katze Verb

Die reflexive und transitive Hülle der Relation $\Rightarrow$ bezeichnen wir mit $\Rightarrow^*$. Umgangssprachlich: durch $\Rightarrow^*$ werden nicht alle $\Rightarrow$-Umformungen dargestellt, sondern teils übersprungen.

Bsp.:
Satz $\Rightarrow^*$ die Katze VP,
Satz $\Rightarrow^*$ die Katze jagt die Maus.

Def.: Die von einer Grammatik erzeugte Sprache ist $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$.

Abkürzende Notation:
$S \to \varepsilon | 0S1$ für $S \to \varepsilon,\ S \to 0S1$

1.2.2.1 Konstruktionsprinzipien für kontextfreie Grammatiken

Regeln der Form $X \to aXb$ führen zu:
ABB 83
Dies lässt sich für balancierte Strukturen nutzen.
Mit der Regel $X \to XX$ wächst der Syntaxbaum in die Breite.
ABB 84
Beispiel, die beide Prinzipien anwendet: $S \to [S]|SS|\varepsilon$
ABB 85
Damit lässt sich bspw. auch folgendes als Grammatik darstellen: $(3 * (4 + 5) - 1) * 2 + 1$.

Bsp.: Grammatik für arithmetische Ausdrücke.

- Zahlen: Lassen sich darstellen durch die Grammatik mit den Regeln:
  $S_N \to 0S_N \,|\, \ldots \,|\, 0S_N \,|\, 0 \,|\, ... \,|\, 9$
  Beispiel für 123:
  ABB 86

- Zeichen: Diese Grammatik verwenden wir, um Ausdrücke darzustellen mit folgender Grammatik:
  $S_E \to S_E + S_E \,|\, S_E - S_E \,|\, S_E * S_E \,|\, S_E / S_E \,|\, (S_E) \,|\, S_N$
  Damit lassen sich Ausdrücke erstellen, z.B.:
  ABB 87
  $(1 + 2) * 3 * 4$

1.2.3 Kellerautomaten und kontextfreie Sprachen

Satz: Kellerautomaten akzeptieren genau die kontextfreien Sprachen.
Beweis: Wir zeigen nur: Für jede kontextfreie Grammatik G gibt es einen PDA $\mathcal{M}$ mit $L(G) = L(\mathcal{M})$.
Skizze:
Wir konstruieren einen PDA mit drei Zuständen:
ABB 88
Im Startzustand wird das Startsymbol S der Grammatik auf den Stack abgelegt und der PDA wechselt in Zustand Z.
Wir unterscheiden 3 Fälle: Das oberste Stackzeichen ist...

- eine Variable A.
  Wenn es eine Regel $A \to \gamma$ der Grammatik gibt, dann kann der PDA A vom Stack entfernen und γ auf den Stack schreiben.
- ein Symbol $a \in \Sigma$.
  Wenn das nächste Zeichen der Eingabe mit a übereinstimmt, wird a vom Stack entfernt.
- das Zeichen $\#$.
  Dann geht der PDA in den Endzustand über.

Richtung PDA→kontextfreie Grammatik: Ohne Beweis.

Bsp.: Wir betrachten die Sprache $L = \{a^n b^n | n \geq 0\}$, die erzeugt wird von der Grammatik mit den Regeln:
$S \to aSb \,|\, \varepsilon$
Aus obiger Konstruktion erhalten wir folgenden PDA:
ABB 89
Verhalten für die Eingabe $aabb$:
ABB 90

1.2.3.1 Der CYK-Algorithmus

Def.: Eine Grammatik $G = (V, \Sigma, P, S)$ liegt in *Chomsky-Normalform* (CNF), wenn alle Regeln die Form $A \to BC$ oder $A \to a$ für $A, B, C \in V$, $a \in \Sigma$ haben.
Jede kontextfreie Grammatik G mit $\varepsilon \notin L(G)$ kann in CNF umgeformt werden.

Bsp.: Wir formen die Grammatik mit den Regeln $S \to SS \,|\, (S) \,|\, ()$ in CNF um.

1. Schritt:
   Terminalsymbole ersetzen durch neue Variablen:
   $S \to SS \,|\, LSR \,|\, LR$
   $L \to (,\ R \to)$
2. Schritt:
   Mehrfache Variablen auf der rechten Seite ersetzen:
   $S \to SS \,|\, LA \,|\, LR$
   $A \to SR$
   $L \to ($
   $R \to)$

Der Ableitungsbaum eines Wortes aus einer Grammatik in CNF ist – bis auf die unterste Ebene – ein binärer Wurzelbaum.
ABB 91
Wenn ein Wort $x = x_1 x_2 \dots x_n$ aus S ableitbar ist ($S \Rightarrow^* x$), dann gibt es ein k und A, B, sodass $S \Rightarrow AB$ und $A \Rightarrow *x_1 \dots x_k,\ B \Rightarrow^* x_{k+1} \dots x_n$
ABB 92 / 96
Der CYK-Algorithmus entscheidet das Wortproblem, indem eine Tabelle konstruiert wird. Der Eintrag T_{ij} ist die Menge der Variablen X mit $X \Rightarrow^* x_i \dots x_j$:
ABB 97
Für $i < j$ wird geprüft, ob sich $x_i \dots x_j$ zerlegen lässt in $x_i \dots x_k,\ x_{k+1} \dots x_j$, so dass gilt:

(i) es gibt eine Regel $X \to AB$

(ii) $A \Rightarrow^* x_i \dots x_k$, $B \Rightarrow^* x_{k+1} \dots x_j$.

Die Menge T_{ij} enthält alle Variablen x mit dieser Eigenschaft. Da (ii) nach Definition von T_{ij} äquivalent ist zu $A \in T_{ik},\ B \in T_{k+1\,j}$, erhalten wir $T_{(ii)} = \{x|$ es gibt eine Regel $X \to x_i\}$.
$T_{ij} = \bigcup_{i \le k < j} \{x|$ es gibt eine Regel $X \to AB \wedge A \in T_{ik} \wedge B \in T_{k+1\,j}\}$
Wenn die Menge T_{ij} in aufsteigender Reihenfolge von $j - i$ berechnet werden, dann können die Einträge der Tabelle aus bereits berechneten Einträgen bestimmt werden (dynamisches Programmieren). Für den Eintrag T_{ij} müssen dabei alle Kombinationen geprüft werden, die den Zerlegungen $x_i \dots x_j = x_i \dots x_k x_{k+1} \dots x_j$ für $i \le k < j$ entsprechen. Das Wort x liegt genau dann in der Sprache, wenn $S \in T_{1n}$.
ABB 99
$T_{26} = \bigcup_{2 \le k < 5} \{x|x \to AB \wedge \underbrace{A \Rightarrow^* x_2 \dots x_k}_{\Leftrightarrow A \in T_{2k}},\ \underbrace{B \Rightarrow^* x_{k+1} \dots x_6}_{\Leftrightarrow B \in T_{k+1\,6}}\}$

- $k = 2$: $A \in T_{22},\ B \in T_{36}$ (lila)
- $k = 3$: $A \in T_{23},\ B \in T_{46}$ (grün)
- $k = 4$: $A \in T_{24},\ B \in T_{56}$ (braun)
- $k = 5$: $A \in T_{25},\ B \in T_{66}$ (hellgrün)

Bsp.: Wir prüfen $(()()) \in L(G')$ für die Grammatik G' in CNF, die die Sprache der korrekten Klammerung erzeugt.
ABB100
Die Laufzeit des CYK-Algorithmus ergibt sich aus
(Größe der Tabelle)·(Aufwand pro Tabelleneintrag)$= O(n^2) \cdot O(n) = O(n^3)$.

```
1 T_ij := ∅
2 for (k=i; k<j; k++) {
3   if (Regel X → AB ∧ A ∈ T_ik ∧ B ∈ T_{k+1 j})
4     T_ij += {x}
5 }
```

1.2.4 Mehrdeutigkeit

Grammatik für arithmetische Ausdrücke:
$E \to E + E \,|\, E - E \,|\, E * E \,|\, E/E \,|\, x \,|\, y \,|\, z$
Daraus lässt sich abbilden:
ABB 111
Diese Grammatik ist auch mehrdeutig, wenn man sich auf nur einen Operator beschränkt:
ABB 112
Da $-$ links-assoziativ ist, stellt nur der linke Ableitungsbaum die korrekte Interpretation des Ausdrucks $x - y - z$ dar.
$\Rightarrow$ Ableitungsbäume dürfen nicht verdreht werden!
Eine Grammatik G heißt *eindeutig*, wenn es für alle Wörter $w \in L(G)$ *genau einen* Ableitungsbaum gibt.
Um eine eindeutige Grammatik zu erhalten, müssen zwei Probleme gelöst werden:

1.) Die Priorität der Operatoren

2.) Die Assoziativität der Operatoren

... müssen beachtet werden.
Lösung für:

1.) Die Grammatik muss so konstruiert werden, dass die Strichoperatoren nur auf der obersten Ebene, die Punktoperatoren nur auf der untersten Ebene erzeugt werden können.
$E \to E + E \,|\, E - E \,|\, F$
$F \to F * F \,|\, F/F \,|\, x \,|\, y \,|\, z$

2.) Die Grammatik muss so beschaffen sein, dass der Ableitungsbaum, gemäß der Richtung der Assoziativität, bei einem links-assoziativen Operator nur nach links wachsen kann.
Basierend auf der Lösung für 1.) (mit T: Term, F: Faktor):
$E \to E + T \,|\, E - T \,|\, T$
$T \to T * F \,|\, T/F \,|\, F$
$F \to x \,|\, y \,|\, z$

Diese Grammatik ist eindeutig.
Der Ableitungsbaum für $x * y + z$ ist:
ABB 113
Der Ableitungsbaum für $x - y - z$ ist:
ABB 114

1.2.5 Syntaxanalyse

```
if ( x<0 || y<0 ) {
  ...
} else if
  ...
```

↓
Lexer
ABB 115
↓
Parser
↓
Syntaxbaum

Ziel: Aus einem Wort einer kontextfreien Sprache soll ein soll ein Syntaxbaum erzeugt werden. Der CYK-Algorithmus ist dafür geeignet, besitzt jedoch eine Laufzeit in $O(n^3)$. Für deterministische kontextfreie Sprachen lässt sich das Wortproblem in Zeit $O(n)$ entscheiden.
Wir erlauben dem Parser, die nächsten k Zeichen der Eingabe (*lookahead*) zu sehen, um abhängig davon Entscheidungen zu treffen.

1.2.5.1 Top-Down-Parser

Ein Top-Down-Parser baut den Syntaxbaum von oben nach unten auf. Ein *Recursive Descent Parser* ist ein Top-Down-Parser, der die Regeln der kontextfreien Grammatik als rekursive Funktionen implementiert.

```
public class Parser {
  String input;
  int pos;

  boolean parse (String inptu0) {
    input = input0 + "#";
    pos = 0;
    return S() && match('#');
  }
```

```
10
11   boolean S() {
12     if( next() == 'a' )
13       return match('a') && S() && match('b'); // entspricht S→aSb
14     else return true; // entsrpicht S→ε
15   }
16
17   char next() {
18     return input.charAt(pos);
19   }
20
21   boolean match(char c) { // entspricht Schleife im Kellerautomat: a,a/ε und b,b/ε bzw. S
       ,ε/aSb
22     if( next() == c ){
23       pos++;
24       return true;
25     }
26     else return false;
27   }
28
29   public static void main(String[] args){
30     Parser p = new Parser();
31     System.out.println(p.parse("aabb"));  // true
32     System.out.println(p.parse("aaabb")); // false
33     System.out.println(p.parse("aabbb")); // false
34   }
35 }
```

Ablauf des Programms für *aabb*:
ABB 116
Achtung: nicht alle Grammatiken lassen sich mit einem rekursiven Abstiegsparser darstellen. Bsp.:
$E \to E + T$:

```
1 boolean E() {
2   return E() && match('+') && T();  // Endlosschleife durch Selbstaufruf
3 }
```

Aus der bereits behandelten Grammatik für arithmetische Ausdrücke kann kein Recursive Descent Parser erzeugt werden, weil die Grammatik linksrekursiv ist. Mögliche Abhilfe: Beseitigung der linksrekursion durch Umbau der Grammtik.
Ansatz:
$E \to T(+E) \,|\, T(-E) \,|\, T$
$T \to F(*T) \,|\, F(/T) \,|\, F(\varepsilon)$
$\Rightarrow$
$E \to TE'$
$E' \to \varepsilon \,|\, +E \,|\, -E$
$T \to FT'$
$T' \to \varepsilon \,|\, *T \,|\, /T$
$F \to x \,|\, y \,|\, z$
Diese Grammatik erzeugt die gleiche Sprache wie die vorherige Grammatik und ist nicht linksrekursiv. Aber die Ableitungsbäume wachsen nach rechts, d.h. alle Operatoren sind rechts-assoziativ.
ABB 117
Das Problem lässt sich für Recursive Descent Parser nicht befriedigend lösen, Man kann linksassoziative Operatoren durch Schleifen verarbeiten. Dazu: EBNF (Extended Backus-Naur-Form).
Obige Grammatik in EBNF:
$E \to T\{+T\}$
$T \to F\{(+|-)F\}$
$F \to x \,|\, y \,|\, z$

Dabei bedeutet $\{+T\}$, dass beliebig viele $+T$ folgen können. $\{...\}$ entspricht $(...)^*$.
Problem: Aus dieser Grammatik geht die Assoziativität der Operatoren nicht hervor. Diese muss festgelegt werden. Links-assoziative Operatoren können mit einer Schleife verarbeitet werden:

```
E() {
 T();
 while (next() == '+') {   // while-Schleife entspricht {+T}
  match ('+');
  T();
 }
}
```

1.2.5.2 Bottom-Up-Parser

Ein *Bottom-Up-Parser* baut einen Ableitungsbaum von unten nach oben auf und kontrolliert dabei die Rechtsableitung der Eingabe. Bottom-Up-Parser lassen sich effizient durch LR-Parser implementieren. Ein LR-Parser liest die Eingabe von links nach rechts und erzeugt den Ableitungsbaum der Rechtsableitung. Ein LR-Parser führt in jedem Schritt eine von vier möglichen Aktionen aus:

- **Shift**: Das nächste Zeichen der Eingabe wird auf den Stack geschoben.
- **Reduce**: Ein oder mehrere Symbole von der Spitze des Stack entsprechen der rechten Seite $A \to \gamma$ einer Regel und werden durch A ersetzt.
- **Accept**: Die Eingabe wurde verarbeitet, der Stock enthält nur das Startsymbol.
- **Error**: Ein Syntaxfehler wird gemeldet.

Bsp.: Wir betrachten die Grammatik für arithmetische Ausdrücke:
$E \to E + T \,|\, E - T \,|\, T$
$T \to T * F \,|\, T/F \,|\, F$
$F \to x \,|\, y \,|\, z$
Für die Eingabe $x + y * z$ führt ein LR-Parser folgende Schritte aus:

Stack	restl. Eingabe	Aktion
	$x + y * z$	shift
x	$+y * z$	reduce
F	$+y * z$	reduce
T	$+y * z$	reduce
E	$+y * z$	shift
$E+$	$y * z$	shift
$E + y$	$*z$	reduce
$E + F$	$*z$	reduce
$E + T$	$*z$	shift*
$E + T*$	z	shift
$E + T * z$		reduce
$E + T * F$		reduce
$E + T$		reduce
E		accept

*hier würde reduce stecken bleiben, weil es keine Regel für $E + E$ geben würde. Der Parser „weiß“ das aus Ableitungstabellen.

Der vom Parser erzeugte Ableitungsbaum der Rechtsableitung (Rechtsableitung ersichtlich dadurch, dass sich rechts alle Änderungen passieren, und die linke Seite unberührt bleibt) ergibt sich aus den ersten beiden Spalten, von unten nach oben gelesen.

```
%{
#include "calc.tab.h"
%}

integer [0-9]+
real {integer}("."{integer})?([eE][+-]?{integer})?

%%

{real}      {yylval.number = atof(yytext); return NUM;}
[ \t]+      ;
\n          {return NL;}
...

// Mit diesem Code kann mit Bison ein C-Programm erzeugt werden, dass diesen Parser
   implementiert
```

Literatur

[1] Boris Hollas. *Grundkurs Theoretische Informatik: Mit Aufgaben und Anwendungen*. Springer Berlin, 2015.