Fakultät Informatik/Mathematik

HOCHSCHULE FÜR
TECHNIK UND WIRTSCHAFT
DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Lehrstuhl für Künstliche Intelligenz / Theoretische Informatik

Vorlesungsmitschrift

THEORETISCHE INFORMATIK

Mitschrift von

Falk-Jonatan Strube

Vorlesung von

Prof. Dr. Boris Hollas

5. Mai 2017

INHALTSVERZEICHNIS

INHALTE

Grundlage: Grundkurs Theoretische Informatik

- 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 $\Sigma \neq \varnothing$ (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: $\varnothing, \varepsilon, \{\varepsilon\}$ sind unterschiedliche Dinge (leere Menge, leeres Wort, Menge mit leerem Wort).

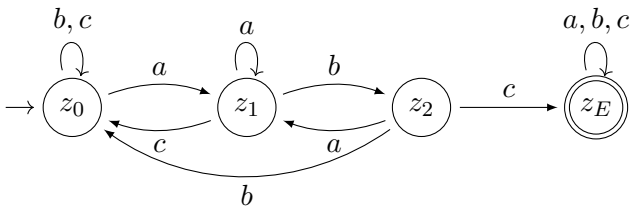
Bemerkung: Σ^* lässt sich ebenfalls definieren durch $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$.

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

1.1 REGULÄRE SPRACHEN

1.1.1 DETERMINISTISCHE ENDLICHE AUTOMATEN (DFA)

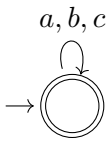
Bsp.:



(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.:

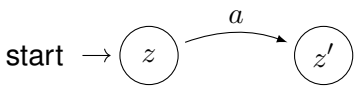


Dieser DFA erkennt die Sprache $\{a, b, c\}^*$.

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:



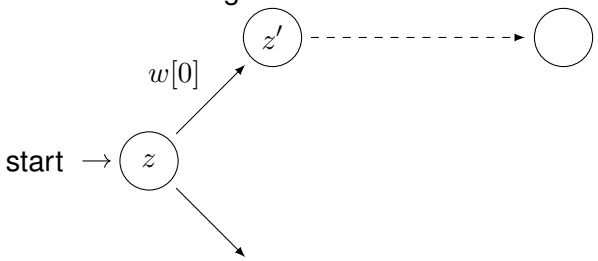
Def.: Die erweiterte Überführungsfunktion $\hat{\delta} : Z \times \Sigma^* \to Z$ ist definiert durch

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

Dazu vergleichbarer C-Code:

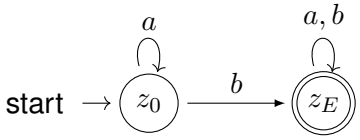
```
int $\hat{\delta}$(int z, char* w){
  if ( strlen(w) == 0 )
    return z;
  else
    return $\hat{\delta}$($\delta$(z, w[0]), w[1]);
```

Veranschaulichung:



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

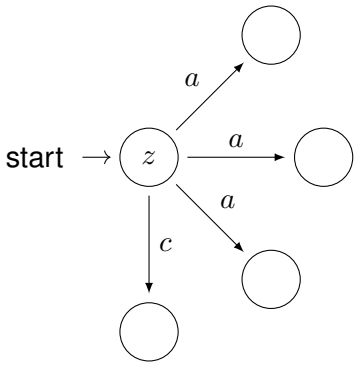
Bsp.:



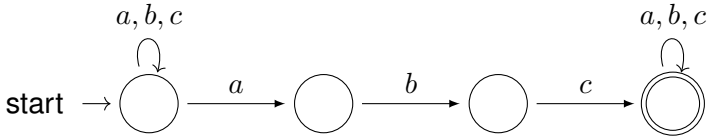
$$\hat{\delta}(z_0, aaba) = \hat{\delta}(\delta(z_0, a), aba) =$$
$$\hat{\delta}(z_0, aba) = \hat{\delta}(\delta(z_0, a), ba) =$$
$$\hat{\delta}(z_0, ba) = \hat{\delta}(\delta(z_0, b), a) =$$
$$\hat{\delta}(z_E, a) = \hat{\delta}(\delta(z_E, a), \varepsilon) =$$
$$\hat{\delta}(z_E, \varepsilon) = z_E$$

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)

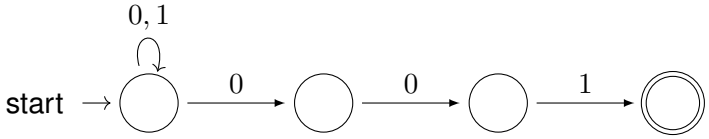


NFA, der alles akzeptiert, was abc enthält:



Beispiel: Wort $abaabcab$

Beispiel: NFA, der alle Wörter akzeptiert, die auf 001 enden:



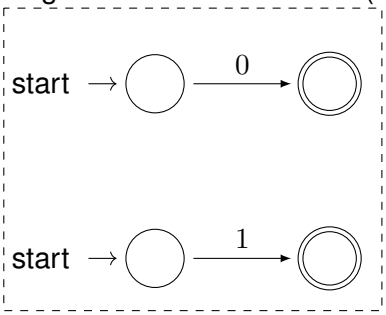
Akzeptierte Worte unter anderem: 01011001, 001001
Ein Wort wird vom NFA akzeptiert, wenn es einen Weg, ausgehend von einem Startzustand, gibt, mit den ein End-Zustand erreicht wird.
Der NFA „weiß“ nicht, welcher Pfad zu durchlaufen ist; diesen muss der Benutzer ermitteln (wie bei einer Straßenkarte).
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

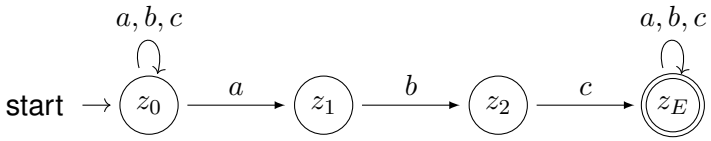
Dabei bedeutet $\delta(z, a) \ni z'$, dass der NEA im Zustand z für die Eingabe a die Möglichkeit besitzt, in den Zustand z' zu wechseln.

Folgendes ist auch ein NFA (mit $L(\mathcal{M}) = \{0, 1\}$):

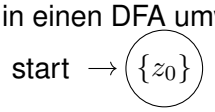


1.1.3 UMWANDLUNG EINES NFA IN EINEN DFA

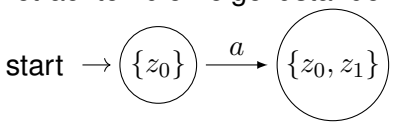
Wir wollen den NFA



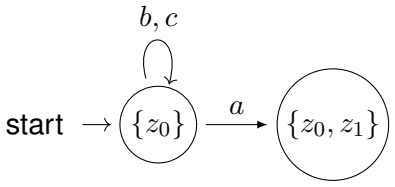
in einen DFA umwandeln. Der Startzustand des DFA besteht aus den Startzuständen des NFA:



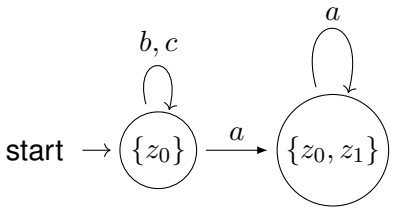
Betrachten die Folgezustände für $a \in \Sigma$:



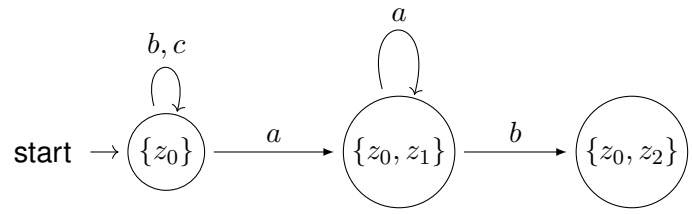
nächster Schritt:



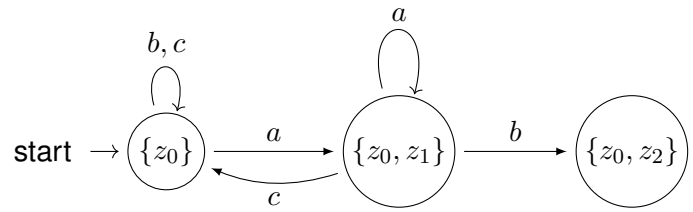
nächster Schritt:



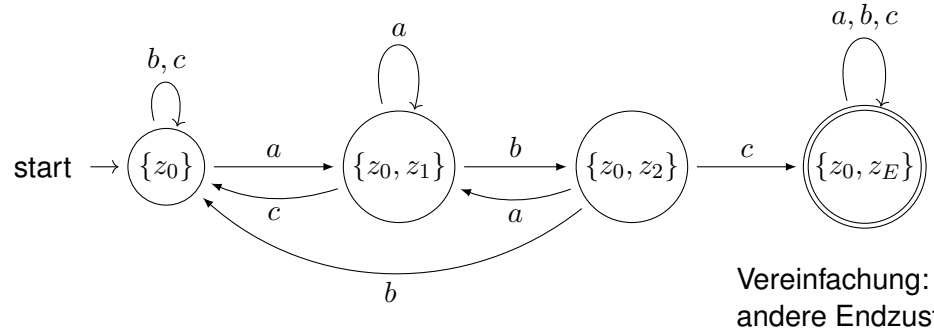
weitere Schritte:
$z_0, b : \{z_0\}$
$z_1, b : \{z_2\}$



$z_0, c : \{z_0\}$
$z_1, c : \{\}$



usw.:



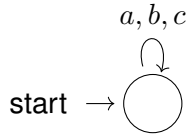
Vereinfachung:
andere Endzustände werden
weg gelassen.

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:

- $\varnothing$ ist ein regulärer Ausdruck und $L(\varnothing) = \varnothing$.
  Bsp.:

  $L(\mathcal{M}) = \varnothing$
- 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	
$E_1 \mid E_2$	$L(E_1 \mid E_2)$	$= L(E_1) \cup L(E_2)$
$E_1 E_2$	$L(E_1 E_2)$	$= L(E_1) L(E_2)$
E_1^*	$L(E_1^*)$	$= L(E_1)^*$

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

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

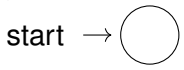
Bsp.:

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

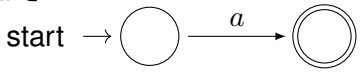
Satz: Reguläre Ausdrücke erzeugen genau die regulären Sprachen.

Skizze: Umwandlung eines regulären Ausdrucks in einen endlichen Automaten.

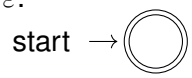
- $\varnothing$:



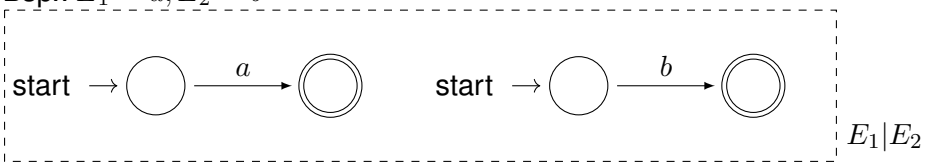
- $a \in \Sigma$:



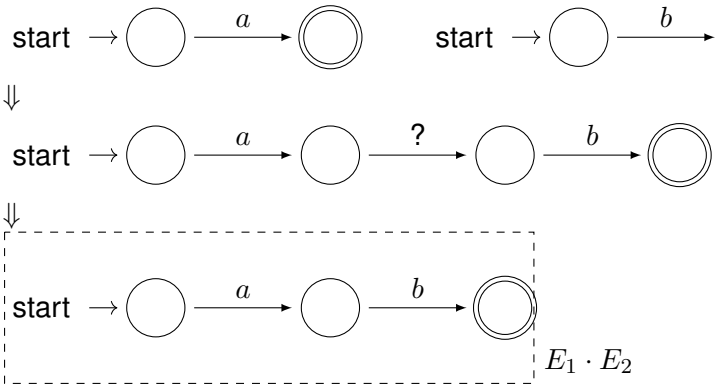
- ε:



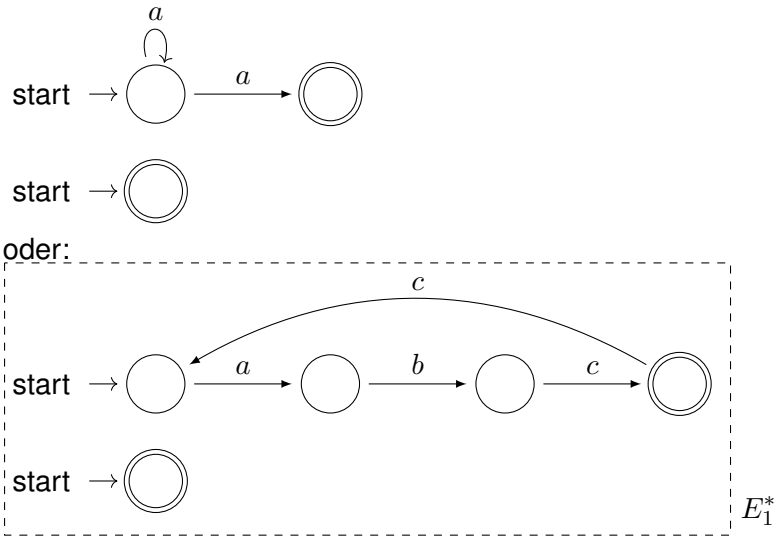
- Seien E_1, E_2 reguläre Ausdrücke und $\mathcal{M}_1, \mathcal{M}_2$ DFAs mit $L(E_1) = L(\mathcal{M}_1), L(E_2) = L(\mathcal{M}_2)$.
  - $E_1|E_2$: $\mathcal{M}_1, \mathcal{M}_2$ sind zusammen ein NFA, der $L(\mathcal{M}_1) \vee L(\mathcal{M}_2)$ erkennt.
    Bsp.: $E_1 = a, E_2 = b$



  - 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.



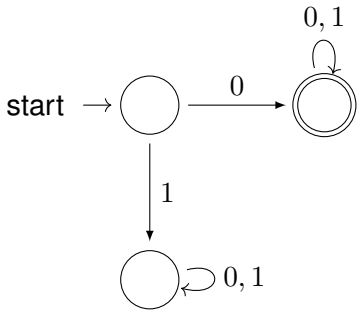
– E_1^*: Es müssen Kanten zurück zum Startzustand eingefügt werden
Beispiel:



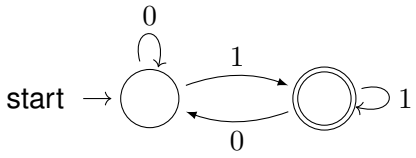
Der Beweis für die umgekehrte Richtung (DFA $\rightarrow$ reg. Ausdruck) ist schwierig.

Bsp.:

- $E = 0(0|1)^*$



-



Beobachtungen:

– um zum Endzustand zu kommen, braucht man eine 1.

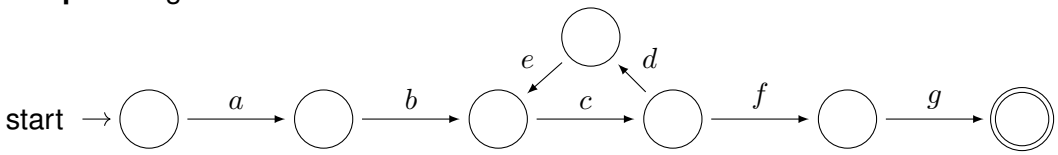
– vor der 1 kann ε stehen, oder beliebig viele 0en der 1en.

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

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.: Gegeben ist der NFA:



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

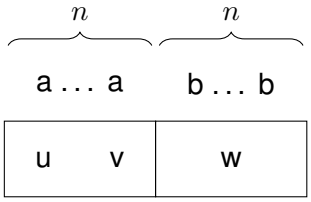
Satz: (Pumping Lemma)
Für jede reguläre Sprache L gibt es ein $n > 0$ (n: Anzahl Zustände des Minimalautomaten), so dass es für alle Wörter $x \in L$ mit $|x| \geq n$ eine Zerlegung $x = uvw$ gibt (in vorherigem Bsp.: $u = ab,\ v = cde\ w = cfg$), so dass gilt:

1. $|v| \geq 1$
2. $|uv| \leq n$ (u, w können auch ε sein)
3. $uv^k w \in L$ 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.



- 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 ↯ #

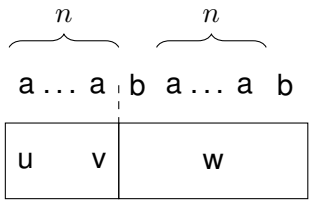
Vorgehen:

- 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.



- Dann enthält $uw = a^{n-|v|} b a^n b$ (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

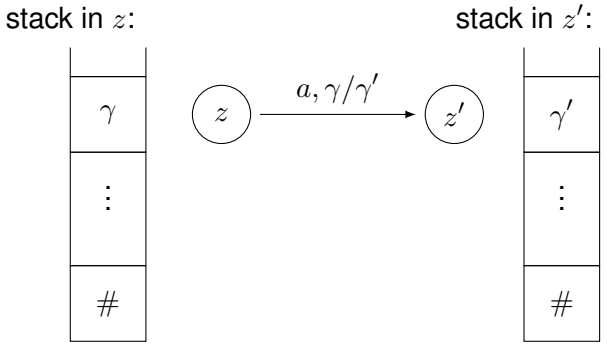
1.2 KONTEXTFREIE SPRACHEN

1.2.1 KELLERAUTOMATEN (PDA)

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

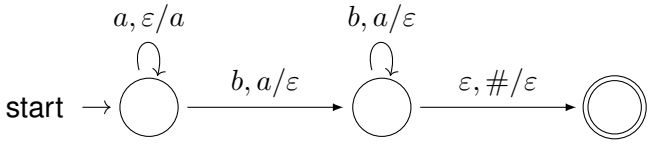
- Es gibt ε-Übergänge.
- Er besitzt einen Stack, auf dem Zeichen abgelegt oder von dem Zeichen gelesen werden können.

Zur graphischen Darstellung von PDAs verwenden wir eine erweiterte Automatennotation:

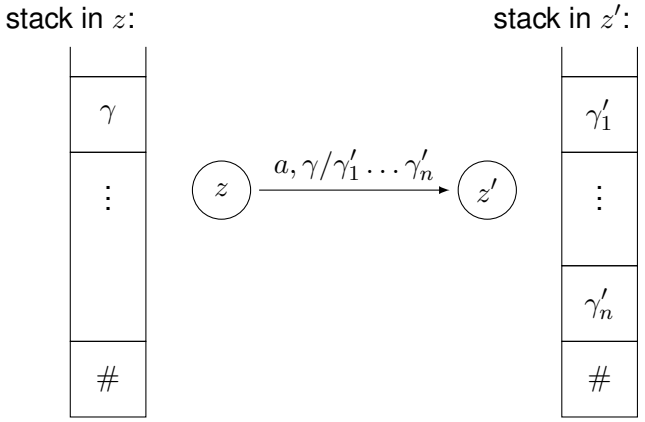


a: Eingabezeichen
γ: Top of Stack in z
γ': Top of Stack in z'
(wenn γ auf Stack liegt, wird γ runter geholt und γ' auf den Stack gelegt)
a, γ: Übergangsvoraussetzung
Stack unverändert lassen:$a, \varepsilon/\varepsilon$
Stack leeren: $a, \gamma/\varepsilon$
Stack füllen: $a, \varepsilon/\gamma$
Unten auf dem Stack liegt das Symbol #. Dies ist das einzige Symbol, das sich zu Beginn einer Rechnung auf dem Stack befindet.

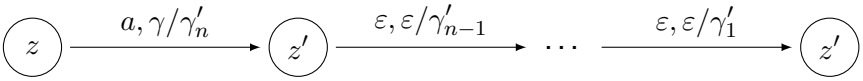
Bsp.: PDA, der $\{a^n b^n | n \in \mathbb{N}\}$ akzeptiert.



Wir erlauben nun, dass der PDA in einem Schritt auch mehrere Zeichen auf den Stack schreibt. Dazu erweitern wir die graphische Notation wie folgt:

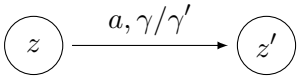


äquivalent:



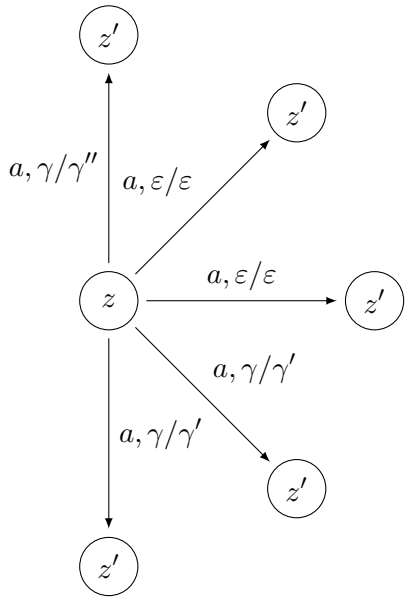
Def.: Ein PDA ist ein Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$

- 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



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

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.



(nicht deterministischer PDA)

1.2.2 KONTEXTFREIE GRAMMATIKEN

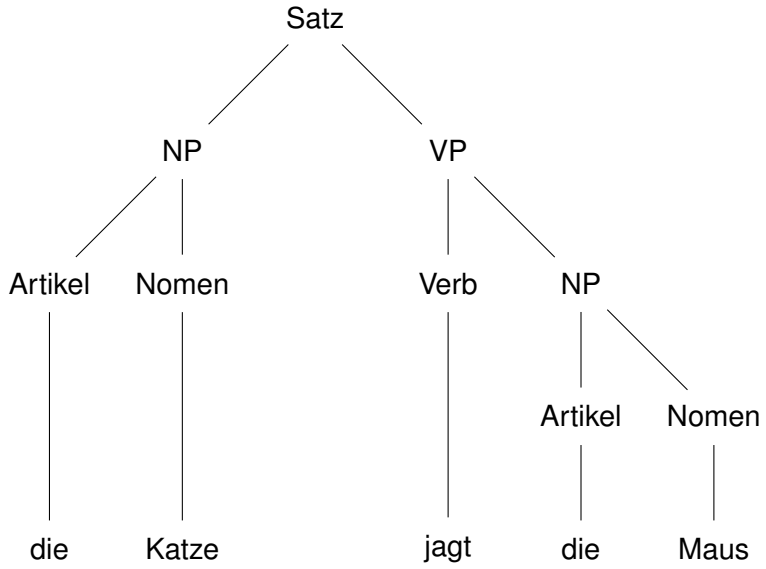
Eine kontextfreie Grammatik beschreibt, wie durch das Ersetzen von variablen Wörter der Sprache erzeugt werden können. Jede Ersetzungsregel hat die Form „linke Seite $\to$ rechte Seite“ (linke Seite der Regel kann ersetzt werden durch die rechte Seite), wobei „linke Seite“ eine Variable ist.

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

Bsp.:

- Satz $\rightarrow$ NP VP[1]
- 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:



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

- V: Endliche Menge der Variablen oder Nonterminalzeichen
- Σ: Alphabet oder Terminalzeichen $V \cap \Sigma = \varnothing$
- 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.

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.

[1] Nominalphase, Verbalphase

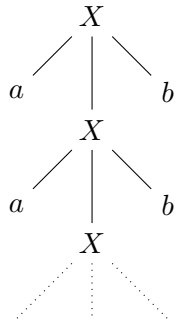
Bsp.:
Satz $\Rightarrow^*$ die Katze VP,
Satz $\Rightarrow^*$ die Katze jagt die Maus.

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

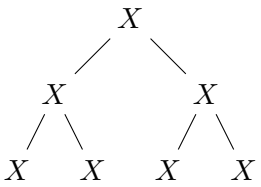
Abkürzende Notation:
$S \to \varepsilon | 0S1$ für $S \to \varepsilon,\ S \to 0S1$

1.2.2.1 KONSTRUKTIONSPRINZIPIEN FÜR KONTEXTFREIE GRAMMATIKEN

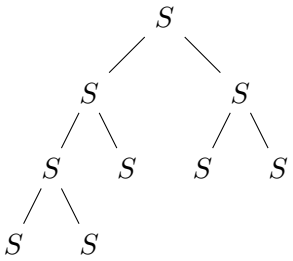
Regeln der Form $X \to aXb$ führen zu:



Dies lässt sich für balancierte Strukturen nutzen.
Mit der Regel $X \to XX$ wächst der Syntaxbaum in die Breite.



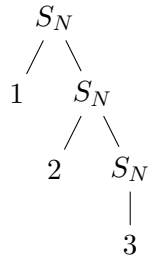
Beispiel, die beide Prinzipien anwendet: $S \to [S]|SS|\varepsilon$



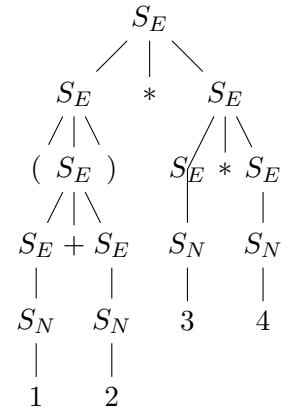
Damit lässt sich bspw. auch folgendes als Grammatik darstellen: $(3 * (4 + 5) - 1) * 2 + 1$.

Bsp.: Grammatik für arithmetische Ausdrücke.

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



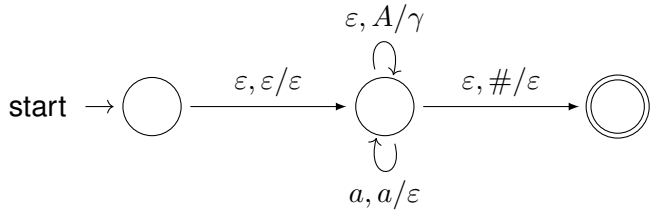
- Zeichen: Diese Grammatik verwenden wir, um Ausdrücke darzustellen mit folgender Grammatik:
  $S_E \to S_E + S_E \,|\, S_E - S_E \,|\, S_E * S_E \,|\, S_E / S_E \,|\, (S_E) \,|\, S_N$
  Damit lassen sich Ausdrücke erstellen, z.B.:



$(1 + 2) * 3 * 4$

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:

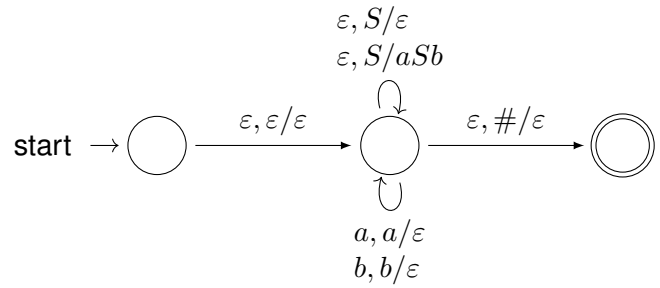


Im Startzustand wird das Startsymbol S der Grammatik auf den Stack abgelegt und der PDA wechselt in Zustand Z.
Wir unterscheiden 3 Fälle: Das oberste Stackzeichen ist...

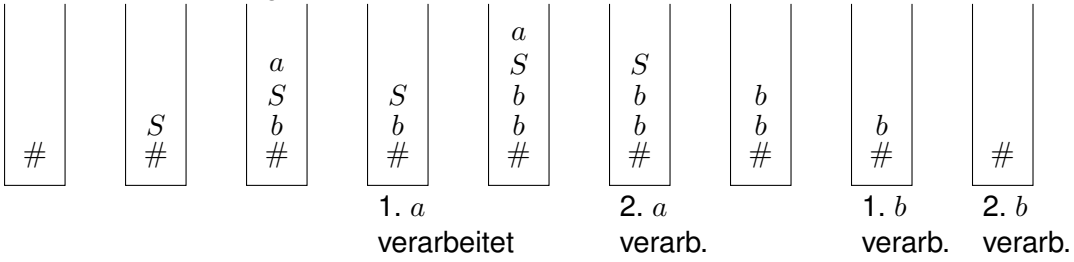
- eine Variable A.
  Wenn es eine Regel $A \to \gamma$ der Grammatik gibt, dann kann der PDA A vom Stack entfernen und γ auf den Stack schreiben.
- ein Symbol $a \in \Sigma$.
  Wenn das nächste Zeichen der Eingabe mit a übereinstimmt, wird a vom Stack entfernt.
- das Zeichen $\#$.
  Dann geht der PDA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



Verhalten für die Eingabe $aabb$:



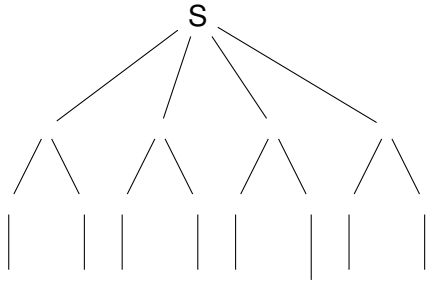
1.2.3.1 DER CYK-ALGORITHMUS

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.

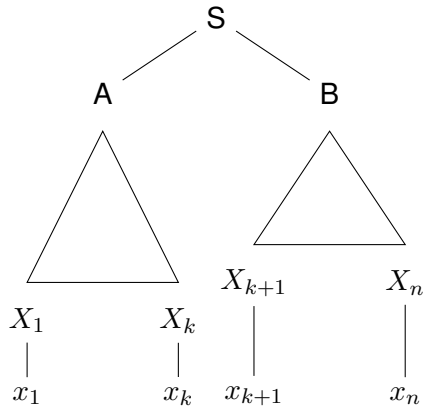
Bsp.: Wir formen die Grammatik mit den Regeln $S \to SS \,|\, (S) \,|\, ()$ in CNF um.

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:
   $S \to SS \,|\, LA \,|\, LR$
   $A \to SR$
   $L \to ($
   $R \to)$

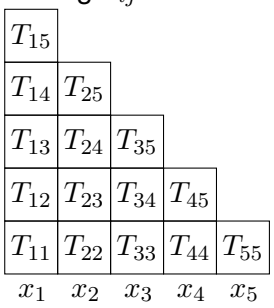
Der Ableitungsbaum eines Wortes aus einer Grammatik in CNF ist – bis auf die unterste Ebene – ein binärer Wurzelbaum.

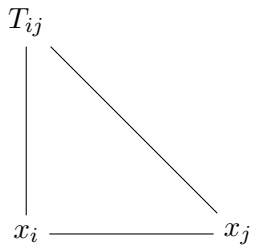


Wenn ein Wort $x = x_1 x_2 \dots x_n$ aus S ableitbar ist ($S \Rightarrow^* x$), dann gibt es ein k und A, B, sodass $S \Rightarrow AB$ und $A \Rightarrow *x_1 \dots x_k,\ B \Rightarrow^* x_{k+1} \dots x_n$



Der CYK-Algorithmus entscheidet das Wortproblem, indem eine Tabelle konstruiert wird. Der Eintrag T_{ij} ist die Menge der Variablen X mit $X \Rightarrow^* x_i \dots x_j$:





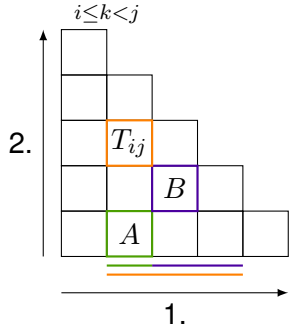
Für $i < j$ wird geprüft, ob sich $x_i \dots x_j$ zerlegen lässt in $x_i \dots x_k,\ x_{k+1} \dots x_j$, so dass gilt:

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

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

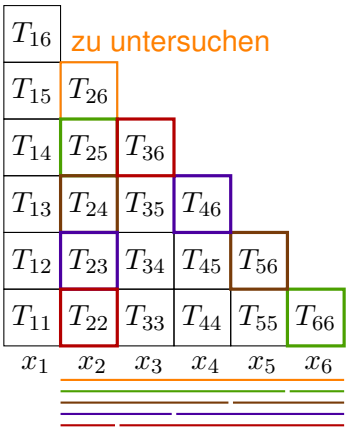
Die Menge T_{ij} enthält alle Variablen x mit dieser Eigenschaft. Da (ii) nach Definition von T_{ij} äquivalent ist zu $A \in T_{ik},\ B \in T_{k+1\,j}$, erhalten wir $T_{(ii)} = \{x|$ es gibt eine Regel $X \to x_i\}$.

$T_{ij} = \bigcup_{i \le k < j} \{x|$ es gibt eine Regel $X \to AB \wedge A \in T_{ik} \wedge B \in T_{k+1\,j}\}$



Vorgehensweise beim befüllen:
1. unterste Zeile
2. obere Zeilen auf Basis der unteren

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 $S \in T_{1n}$.

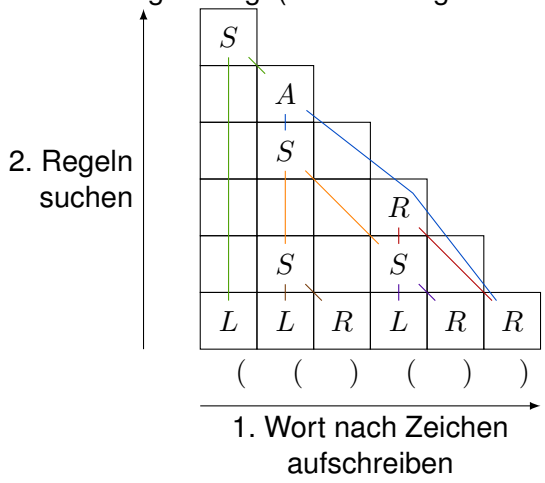


$\leftarrow T_{26}$ kann potentiell durch diese Kombinationen gebildet werden

$$T_{26} = \bigcup_{2 \le k < 5} \{x | x \to AB \wedge \underbrace{A \Rightarrow^* x_2 \dots x_k}_{\Leftrightarrow A \in T_{2k}}, \underbrace{B \Rightarrow^* x_{k+1} \dots x_6}_{\Leftrightarrow B \in T_{k+1\,6}}\}$$

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

Bsp.: Wir prüfen $(()()) \in L(G')$ für die Grammatik G' in CNF, die die Sprache der korrekten Klammerung erzeugt (siehe Übung: $S \to SS,\ S \to LA,\ A \to SR$).



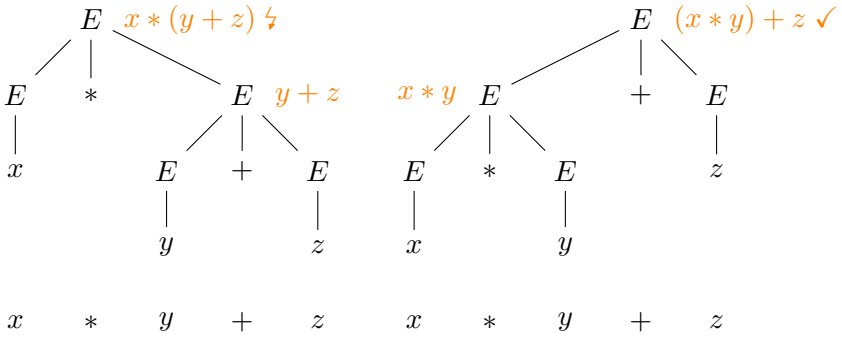
Die Laufzeit des CYK-Algorithmus ergibt sich aus
(Größe der Tabelle)·(Aufwand pro Tabelleneintrag)$= O(n^2) \cdot O(n) = O(n^3)$.

```
$T_{ij}:= \emptyset$
for (k=i; k<j; k++) {
  if (Regel $X\to AB$ $\wedge$ $A \in T_{ik}$ $\wedge$ $B \in T_{k
     +1\,j}$)
    $T_{ij}$ += {x}
}
```

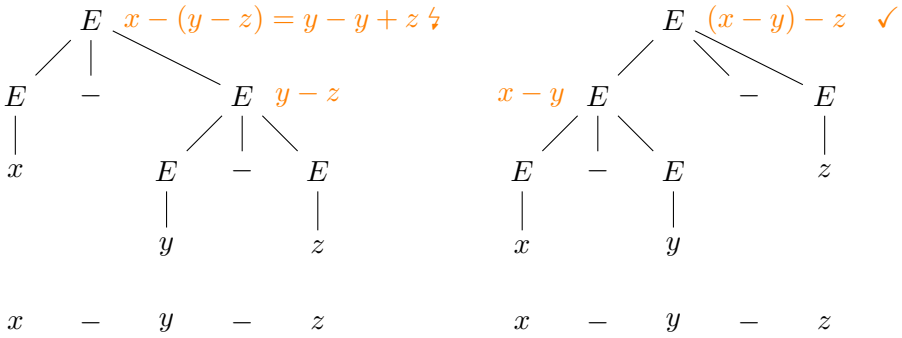
1.2.4 MEHRDEUTIGKEIT

Grammatik für arithmetische Ausdrücke:
$E \to E + E \,|\, E - E \,|\, E * E \,|\, E/E \,|\, x \,|\, y \,|\, z$

Daraus lässt sich abbilden:



Diese Grammatik ist auch mehrdeutig, wenn man sich auf nur einen Operator beschränkt:



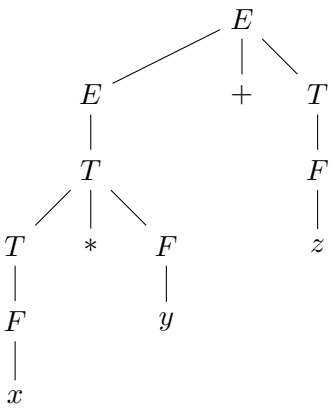
Da $-$ links-assoziativ ist, stellt nur der rechte Ableitungsbaum die korrekte Interpretation des Ausdrucks $x - y - z$ dar.
$\Rightarrow$ Ableitungsbäume dürfen nicht verdreht werden!
Eine Grammatik G heißt EINDEUTIG, wenn es für alle Wörter $w \in L(G)$ GENAU EINEN Ableitungsbaum gibt.
Um eine eindeutige Grammatik zu erhalten, müssen zwei Probleme gelöst werden:

1. Die Priorität der Operatoren
2. Die Assoziativität der Operatoren

... müssen beachtet werden.
Lösung für:

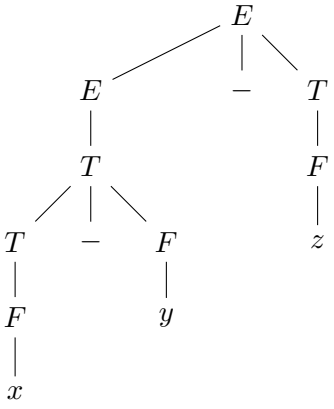
1. Die Grammatik muss so konstruiert werden, dass die Strichoperatoren nur auf der obersten Ebene, die Punktoperatoren nur auf der untersten Ebene erzeugt werden können.
   $E \to E + E \mid E - E \mid F$
   $F \to F * F \mid F/F \mid x \mid y \mid z$
2. Die Grammatik muss so beschaffen sein, dass der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

Diese Grammatik ist eindeutig.
Der Ableitungsbaum für $x * y + z$ ist:



$x \quad * \quad y \quad + \quad z$

Der Ableitungsbaum für $x - y - z$ ist:



$x \quad - \quad y \quad - \quad z$

1.2.5 SYNTAXANALYSE

```
1 if ( x<0 || y<0 ) {
2   ...
3 } else if
4   ...
```

↓
Lexer

IF	(	ID	<	NUM	OR	ID	<	NUM	)	···

↓
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.

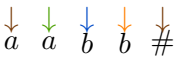
1.2.5.1 TOP-DOWN-PARSER

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.

Beispiel für Sprache $a^n b^n$:

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

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

  boolean S() {
    if( next() == 'a' )
      return match('a') && S() && match('b'); // entspricht S$\
         to$aSb
    else return true; // entsrpicht S$\to$$\varepsilon$
  }

  char next() {
    return input.charAt(pos);
  }

  boolean match(char c) { // entspricht Schleife im Kellerautomat:
     a,a/$\varepsilon$ und b,b/$\varepsilon$ bzw. S,$\varepsilon$/
     aSb
    if( next() == c ){
      pos++;
      return true;
    }
    else return false;
  }

  public static void main(String[] args){
    Parser p = new Parser();
    System.out.println(p.parse("aabb"));  // true
    System.out.println(p.parse("aaabb")); // false
    System.out.println(p.parse("aabbb")); // false
  }
}
```

Ablauf des Programms für AABB:

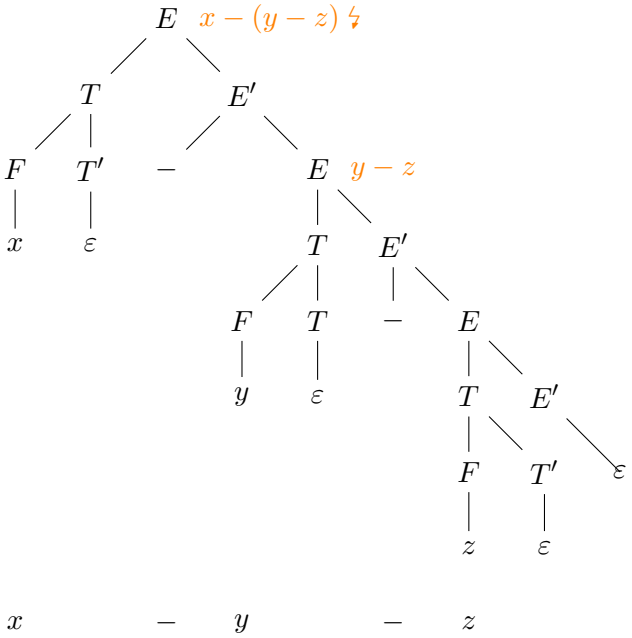


1.
2. S() && match('b')
3. S() && match('b')
4.
5.
6. parse ✓

Achtung: nicht alle Grammatiken lassen sich mit einem rekursiven Abstiegsparser darstellen. Bsp.: $E \to E + T$:

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

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 rechtsassoziativ.



$x \quad - \quad y \quad - \quad z$

Das Problem lässt sich für Recursive Descent Parser nicht befriedigend lösen, Man kann links-assoziative Operatoren durch Schleifen verarbeiten. Dazu: EBNF (Extended Backus-Naur-Form).
Obige Grammatik in EBNF:
$E \to T\{\{+|-\}T\}$

$T \rightarrow F\{(*|/)F\}$
$F \rightarrow x \,|\, y \,|\, z$
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();
 }
}
```

(SimpleInfixCalc.java)

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 $A \rightarrow \gamma$ 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:
$E \rightarrow E + T \,|\, E - T \,|\, T$
$T \rightarrow T * F \,|\, T/F \,|\, F$
$F \rightarrow x \,|\, y \,|\, z$
Für die Eingabe $x + y * z$ führt ein LR-Parser folgende Schritte aus:

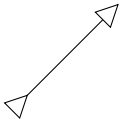
Stack	restl. Eingabe	Aktion
	$x + y * z$	shift
x	$+y * z$	reduce
F	$+y * z$	reduce
T	$+y * z$	reduce
E	$+y * z$	shift
$E+$	$y * z$	shift
$E + y$	$*z$	reduce
$E + F$	$*z$	reduce
$E + T$	$*z$	shift[2]
$E + T*$	z	shift
$E + T * z$		reduce
$E + T * F$		reduce
$E + T$		reduce
E		accept

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
```

1.2.6 OL-SYSTEME

Zur Darstellung benötigen wir Turtle-Grafik:



Befehle:

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

$forward(l)$
$left(\alpha)$ bzw. $right(\alpha)$

Im Unterschied zu einer kontextfreien Grammatik wird in einem OL-System in jedem Ableitungsschritt jede Variable ersetzt. Jede Variable wird dabei durch die gleiche Regel ersetzt. Anstelle eines Startsymbols gibt es eine initiale Satzform.

Bsp.:
Variable: F
Symbole: $+, -$
Regel: $F \to F + F - -F + F$

$$F \Rightarrow F + F - -F + F$$
$$\Rightarrow \underbrace{F + F - -F + F}_{1.\,F} + \underbrace{F + F - -F + F}_{2.\,F} - - \underbrace{F + F - -F + F}_{usw.} + \underbrace{F + F - -F + F}$$

Graphische Darstellung:
$F = forward$
$+ = left(60°)$
$- = right(60°)$

n=0:

n=1:

n=2:

1.3 DIE CHOMSKY HIERARCHIE

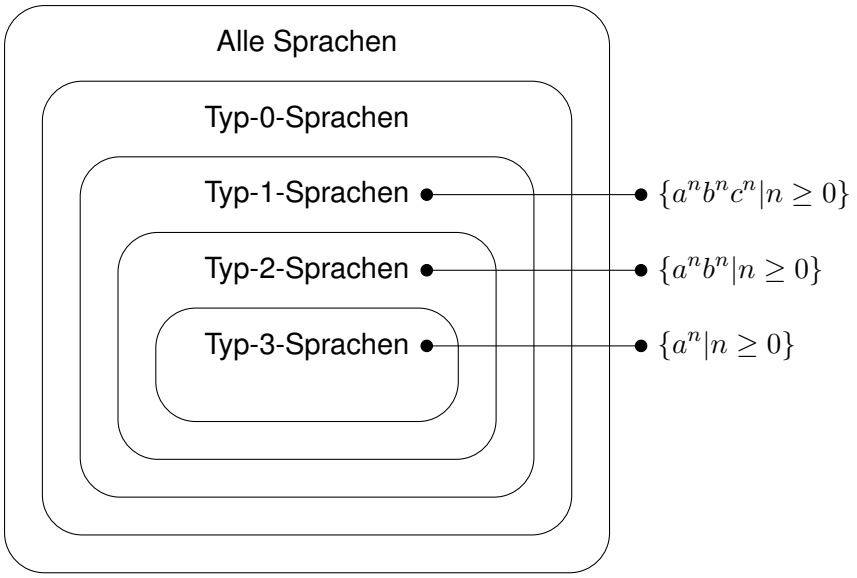
Für Grammatiken:

Typ	in aller Regeln $u \to r$ gilt	Beispiele
0 (rekursiv aufzählbar)	u, r beliebig	$ab \to c$
1 (kontext sensitiv)	$u = \alpha X \beta,\ v = \alpha\gamma\beta$ mit $\alpha, \beta \in (V \cup \Sigma)^*,\ x \in V,\ \gamma \in (V \cup \Sigma)^+$	$aAb \to aBb$
2 (kontextfrei)	$u \in V,\ v \in (V \cup \Sigma)^*$	$A \to aBb$
3 (regulär)	$u \in V,\ v \in \Sigma \cup \{\varepsilon\} \cup \Sigma V$	$A \to a$, $A \to aB$

weitere Beispiele:

0. $(a + b) \cdot c \to a \cdot c + b \cdot c$
1. Artikel$_m$ Nomen $\to$ Artikel$_m$ Kater
   Artikel$_f$ Nomen $\to$ Artikel$_f$ Katze
2. -
3. (ist linear im rechten Teilbaum entartet)

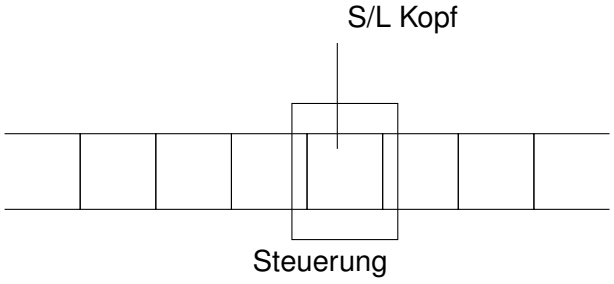
Chomsky-Hierarchie mit Beispielsprachen:



Eine Sprache hat mindestens die gleiche Klasse wie seine Grammatik, kann aber auch eine höhere haben. Für eine reguläre Sprache muss ein regulärer Ausdruck konstruierbar sein.

2 BERECHENBARKEIT UND KOMPLEXITÄT

Frage: Was können Computer berechnen, was können Computer effizient berechnen?



(Turing Maschine)

2.1 ENTSCHEIDBARKEIT

Entscheidungsproblem: Gegeben eine Sprache L und ein Wort $w \in \Sigma^*$. Gehört w zu L ($w \in L$)?
Wir kennen bereits entscheidbare Probleme:

- Wenn L als regulärer Ausdruck gegeben ist, dann ist $w \in L$ entscheidbar durch folgendes Verfahren:
  - Regulären Ausdruck in DFA umwandeln.
  - DFA die Eingabe w übergeben.
  - Wenn DFA einen Endzustand erreicht gilt $w \in L$, sonst $w \notin L$.
- $L = \varnothing$ ist entscheidbar, durch ein Entscheidungsverfahren, das immer „falsch“ liefert.
- $L = \Sigma^*$ ist entscheidbar, durch ein Entscheidungsverfahren, das immer „richtig“ liefert.
- Wenn L als kontextfreie Grammatik gegeben ist, ist L entscheidbar über:
  - Umformung in CNF
  - Anwendung des CYK-Algorithmus

Def.: Eine Sprache L heißt ENTSCHEIDBAR, wenn es ein Programm P_L (Entscheidungsverfahren) gibt, wenn gilt

- für $w \in L$ liefert P_L die Ausgabe $true$
- für $w \notin L$ liefert P_L die Ausgabe $false$.

Bsp.:

- Wenn L eine kontextfreie Sprache, ist, dann ist L entscheidbar durch den CYK-Algorithmus.
- Die Sprache $L = \{(M, w) \mid M \text{ ist ein DFA mit } w \in L(M)\}$ ist entscheidbar durch ein Programm des aus der Eingabe (M, w) eine Repräsentation des DFA erstellt und damit M für die Eingabe w simuliert (vgl. Darstellung eines Interpreters in Zusammenhang mit der erweiterten Überführungsfunktion $\tilde{\delta}$).

- Die Sprache $\{M \mid M$ ist ein DFA mit $L(M) = \Sigma^*\}$ ist entscheidbar (siehe Übung).
- Allgemein: $\{(P, w) \mid P$ hält für $w\}$. Im Beispiel: Collatz-Problem.

2.2 HALTEPROBLEM

Das Halteproblem ist die Sprache $H = \{(P, w) \mid$ Das Programm P hält für die Eingabe $w\}$. Die Frage, ob ein Programm P für eine gegebene Eingabe w hält, ist damit gleichwertig zur Frage $(P, w) \in H$. Wir beweisen zunächst die Unentscheidbarkeit eines Spezialfalls.

Satz: Das spezielle Halteproblem $K = \{P \mid$ das Programm P hält für die Eingabe $P\}$ ist unentscheidbar.

Bsp.:

- für ein Programm $P \in K$:

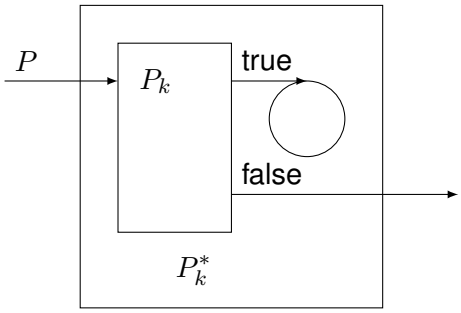
```
void P( Input w ) {
  return;
}
```

- für ein Programm $P \notin K$:

```
void P( Input w ) {
  while(true);
}
```

Beweis (Widerspruch):
Angenommen, K sei entscheidbar durch ein Entscheidungsverfahren P_K. Daraus konstruieren wir ein Programm P_K^*, das P_K als Unterprogramm verwendet und das

- in eine Endlosschleife übergeht, wenn P_K $true$ liefert
- hält, wenn P_K $false$ liefert.



Nach Konstruktion gilt dann:

- Für die Eingabe P hält P_K^* genau dann, wenn P_k false enthält.

Da nach Annahme P_K ein Entscheidungsverfahren für K ist, bedeutet das:

- Für die Eingabe P hält P_K^* genau dann, wenn P für die Eingabe P nicht hält.

Für $P = P_K^*$ folgt:

- Für die Eingabe P_K^* hält P_K^* genau dann, wenn P_K^* für die Eingabe P_K^* nicht hält.

Widerspruch!

Folgerung: Das Halteproblem H ist unentscheidbar.
Beweis: Angenommen, H wäre entscheidbar durch P_H. Dann können wir folgendes Entscheidungsverfahren für K konstruieren:

```
bool P_K ( Programm P ) {
  return P_H(P,P);
}
```

Widerspruch!

2.2.1 WEITERE UNENTSCHEIDBARE PROBLEME

Um die Unentscheidbarkeit weiterer Probleme zu zeigen, verwenden wir einen Beweis durch Widerspruch nach folgender Bauart:

- Um die Unentscheidbarkeit einer Formalen Sprache B zu zeigen, verwenden wir eine unentscheidbare Sprache A.
- Wir nehmen an, dass die Sprache B entscheidbar ist. Folglich gibt es ein Entscheidungsverfahren für B.
- Wir zeigen, dass sich damit ein Entscheidungsverfahren für A konstruieren lässt. Widerspruch!

Anwendung:

Satz: Das Halteproblem H ist nicht entscheidbar.
$H = \{(P, w) \mid P \text{ hält für } w\}$
$K = \{P \mid P \text{ hält für } P\}$

Beweis. Angenommen H ist entscheidbar.
Dann gibt es ein Entscheidungsverfahren P_H für H.
Dann können wir folgendes Programm konstruieren:

```
boolean $P_K$ (Program P) {
  return $P_H$(P,P);
}
```

Dann gilt: $\underline{P \in K} \Leftrightarrow P \text{ hält für } P \Leftrightarrow (P, P) \in H \Leftrightarrow \underline{P_H(P, P) \text{ ist true}}$
Widerspruch, da K unentscheidbar ist. □

Satz: $H_\varepsilon = \{P \mid P \text{ hält für die Eingabe } \varepsilon\}$

Beweis. Angenommen, H_ε ist entscheidbar.
Dann gibt es Entscheidungsverfahren P_{H_ε} für H_ε.
Damit können wir folgendes Programm konstruieren:

```
boolean $P_H$ (Program P, Input w){
  void F() {
    P(w);
  }
  return $P_{H_\varepsilon}$(F);
}
```

Dann gilt: $(P, w) \in H \Leftrightarrow P \text{ hält für } w \Leftrightarrow F \text{ hält für } \varepsilon \Leftrightarrow F \in H_\varepsilon$
Widerspruch, da H unentscheidbar ist. □

Satz: $H^* = \{P \mid P \text{ hält für jede Eingabe}\}$ ist nicht entscheidbar.

Beweis. Angenommen, H^* ist entscheidbar durch ein Entscheidungsverfahren P_{H^*}.
Dann können wir folgendes Programm konstruieren:

```
boolean $P_{H_\varepsilon}$(Program P) {
  void F(Input w){
    P($\varepsilon$);
  }
  return $P_{H^*}$(F);
}
```

Dann gilt: $P \in H_\varepsilon \Leftarrow P$ hält für die Eingabe $\varepsilon \Leftrightarrow F$ hält für jede Eingabe $w \Leftrightarrow F \in H^*$
Widerspruch, da H_ε nicht entscheidbar. □

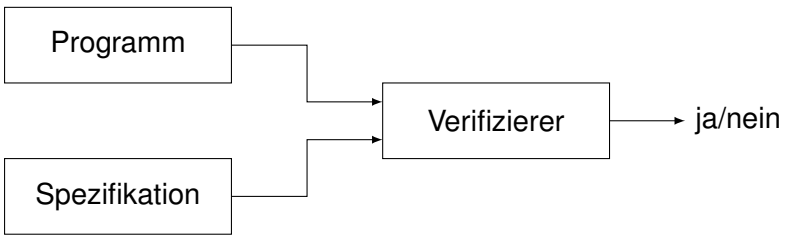
Satz: $\ddot{A} = \{(P_1, P_2) \mid P_1, P_2 \text{ berechnen die gleichen Funktionen}\}$ ist nicht entscheidbar.

Beweis. Angenommen $\ddot{A}$ ist entscheidbar durch $P_{\ddot{A}}$.
Dann konstruieren wir das Programm:

```
boolean $P_{H^*}$(Program P){
  int F(Input w){
    return 0;
  }
  int G(Input w){
    P(w); // hält P(w) immer? Nur dann ist G 0.
    return 0;
  }
  return $P_{\ddot{A}}$(F,G);
}
```

Dann gilt: $P \in H^* \Leftrightarrow P$ hält für jede Eingabe $w \Leftrightarrow G$ berechnet $w \mapsto 0$
$\Leftrightarrow F, G$ berechnen die gleiche Funktion $\Leftrightarrow (F, g) \in \ddot{A}$ (genau dann, wenn ich H^* entscheiden kann, kann ich $\ddot{A}$ entscheiden)
Widerspruch, da H^* unentscheidbar. □

2.2.1.1 UNENTSCHEIDBARKEIT DER PROGRAMMVERIFIKATION



Aus der Unentscheidbarkeit des Halteproblems folgt:
$\{(P, S) \mid$ Das Programm P erfüllt die Spezifikation $S\}$ ist unentscheidbar.
Auch unentscheidbar:

- $\{P \mid P \text{ verursacht keine Division durch } 0\}$
- $\{P \mid P \text{ verursacht keine Array-out-Bound-Fehler}\}$
- $\{P \mid P \text{ dereferenziert keine Nullpointer}\}$

Möglicher Ausweg:
Verifizierer liefert ja, nein ODER unbekannt.

3 KOMPLEXITÄT

Frage: Gibt es Probleme, die zwar lösbar (entscheidbar) sind, aber dazu einen sehr großen Rechenaufwand erfordern?

Kostenmaße:

- Uniforme Kostenmaß:
  Alle Operationen erfordern konstanten Aufwand (LZ in $O(1)$)
- Logarithmisches Kostenmaß:
  Alle Operationen erfordern logarithmischen Aufwand (LZ in $O(\log n)$)

Im folgenden verwenden wir das uniforme Kostenmaß, wo angemessen.

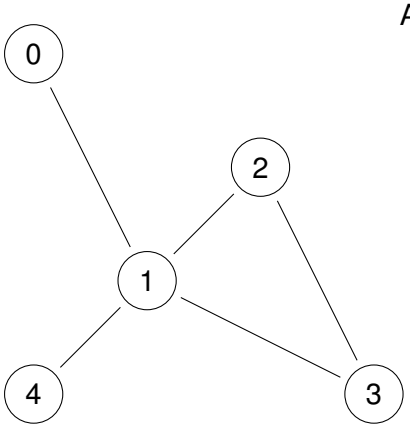
3.1 DIE KLASSE P

Def.: $P = \bigcup_{k>0} \{L \mid L \text{ ist entscheidbar durch ein Programm LZ in } O(n^k)\}$

Bsp.: Folgende Sprachen bzw. Probleme liegen in P:

- $\varnothing$, Σ^* (LZ $O(1)$)
- $\{a^n \mid n \geq 0\}$ (muss maximal n Zeichen überprüfen: LZ $O(n)$)
- Jede kontextfreie Sprache L, da L duch den CYK-Algorithmus in Zeit $O(n^3)$ entschieden werden kann.

Bsp.: Das Problem $PFAD = \{(G, n_1, n_2) \mid G$ ist ein Graph, in dem es einen Pfad von n_1 nach n_2 gibt $\}$ liegt in P.
Veranschaulichung:



Array(List(1),
List(0,2,3,4),
. . .)

Der Graph G sei dabei als Adjazenzliste gegeben.

Beweis. Da die Adjazenzliste von $G = (V, E)$ mindestens $|V| + |E|$ Elemente enthält, gilt für die Länge n der Eingabe: $n \geq |V| + |E|$.

Ein Entscheidungsverfahren für $PFAD$ ist eine in n_1 gestartete Breitensuche nach n_2, die die LZ $O(|V| + |E|)$ besitzt. Folglich ist die LZ des Entscheidungsverfahrens $\leq c \cdot (|V| + |E|) \leq c \cdot n \in O(n)$. □

Das gleiche Ergebnis erhalten wir, wenn die LZ nur in $|V|$ gemessen wird: Denn es gilt $n \geq |V|$. Für die LZ der Breitensuche gilt: LZ$\in O(|V| + |E|) \subseteq O(|V| + |V|^2) = O(|V|^2) \subseteq O(n^2)$. Auch damit folgt $PFAD \in P$.

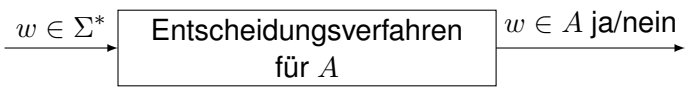
Die Klasse P wird betrachtet als Klasse der effizient lösbaren Probleme.

3.2 DIE KLASSE NP

Die Klasse NP (nicht deterministisch polynomiell) enthält alle Probleme, die in polynomieller Zeit verifizierbar sind.

Bsp.: $PFAD \in NP$, denn mit Hilfe eines Zertifikates lässt sich prüfen, dass es in G einen Pfad von n_1 nach n_2 gibt und die LZ dafür ist polynomiell:
Das Zertifikat ist der Pfad selbst, die LZ liegt in $O(|V|^2)$.

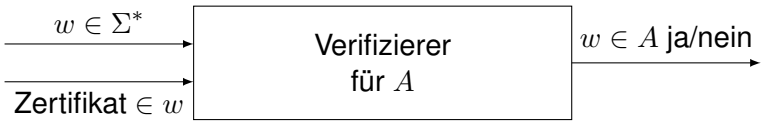
Bsp.: $SAT = \{F \mid F$ ist eine erfüllbare Formel der Aussagenlogik$\}$. Zum Beispiel gilt $x \vee y \in SAT$, $x \wedge \neg x \notin SAT$.
SAT ist entscheidbar durch die Konstruktion einer Wahrheitstabelle. Wenn F n Variablen enthält, benötigt dies die LZ $O(2^n)$ (liegt also nicht in P).
SAT lässt sich jedoch effizient (d.h. in polynomieller Zeit) verifizieren, wenn als Zertifikat eine erfüllende Belegung für F gegeben ist. Die LZ für die Verifikation liegt dann in $O(|F|)$. Deshalb gilt $SAT \in NP$.



$A \in P$

LZ des Entscheidungsverfahrens polynomiell $\Rightarrow A \in P$

$A \in NP$



LZ des Verifizierungsverfahrens polynomiell $\Rightarrow A \in NP$

Def.: $NP = \bigcup_{k \geq 1} \{L \mid L$ ist verifizierbar in Zeit $O(n^k)\}$, wobei n die Länge der Eingabe ist.

Satz: $P \subseteq NP$.

Beweis. (Skizze) Sei $L \in P$. Dann kann L in polynomieller Zeit entschieden werden. Ein Entscheidungsverfahren ist aber auch ein Verifizierungsverfahren, das kein Zertifikat (oder das Zertifikat ε) verwendet. Folglich gilt $P \in NP$. □

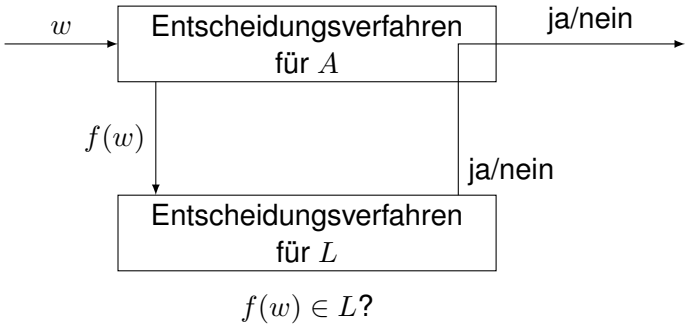
Unbekannt ist, ob $P = NP$ gilt.
Weiterhin gibt es Sprachen, von denen nicht bekannt ist, ob sie in NP liegen. Ein Beispiel ist $TAUT = \{F \mid F$ ist eine Tautologie$\}$. Es wird $TAUT \notin NP$ vermutet.

3.2.1 NP-VOLLSTÄNDIGKEIT

Die NP-vollständigen Probleme sind eine Klasse von Problemen innerhalb von NP, für die keine effizienten (polynomielle) Entscheidungsverfahren bekannt sind. Die NP-vollständigen Probleme sind mindestens so schwierig wie alle anderen Probleme in NP.

„Halbformal“ bedeutet das:
Eine Sprache L ist NP-vollständig, wenn gilt:

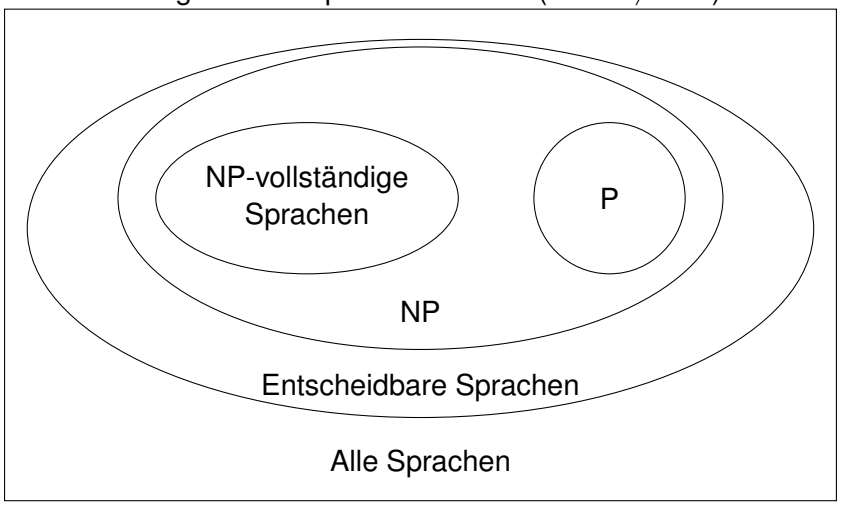
1. $L \in NP$
2. Für jede Sprache $A \in NP$ lässt sich das Entscheidungsproblem für A effizient übersetzen in ein Entscheidungsproblem für L.



Folgerung: Aus $L \in P$ folgt dann auch $A \in P$.

Weitere Folgerung: Wenn es eine NP-vollständige Sprache L gibt mit $L \in P$, dann gilt $P = NP$.

Trotz jahrzehnte währender Suche wurde bisher kein NP-vollständiges Problem $L \in P$ gefunden. Deswegen wird $P \neq NP$ vermutet (nur vermutet, nicht bewiesen).
Vermutete Lage der Komplexitätsklassen (Fall $P \neq NP$):



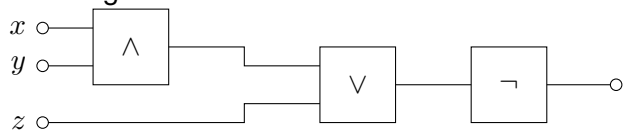
Satz: Die Sprache $SAT = \{F \mid F$ ist eine erfüllbare Formel der Aussagenlogik$\}$ ist NP-vollständig.
Dies bedeutet, dass kein effizientes (polynomielles) Entscheidungsverfahren bekannt ist. Alle bekannten Entscheidungsverfahren besitzen exponentielle Laufzeit. Das naive Verfahren besitzt eine Laufzeit in $O(p(n) \cdot 2^n)$. Das beste bekannte Verfahren besitzt eine Laufzeit in $O(1,308^n \cdot p(n))$ für $3SAT$ (Formel in KNF mit 3 Literalen pro Klausel, z.B. $(x \vee y \vee \neg z) \wedge (\neg x \vee a \vee z) \wedge (\neg y \vee z \vee b) \wedge \ldots$).
Nicht alle Elemente in SAT sind schwierige Instanzen. Z.B. ist $2SAT = \{F \mid F$ ist eine Formel

in KNF mit 2 Literalen pro Klausel$\}$ effizient entscheidbar (d.h. $2SAT \in P$).

$$\underbrace{(\dots) \wedge (\dots)}_{2^{\frac{n}{2}}} \wedge \underbrace{(\dots) \wedge (\dots)}_{2^{\frac{n}{2}}} = 2^{\frac{n}{2}+1}$$

Anwendungen von SAT

- Äquivalenz von Schaltkreisen
  Da Schaltkreise durch logische Gatter ($\vee$, $\wedge$, $\neg$) dargestellt werden, lassen sich Schaltkreise durch logische Formeln beschreiben.



  Zwei Schaltkreise S_1, S_2 sind äquivalent genau dann, wenn $S_1 \leftrightarrow S_2$ eine Tautologie ist genau dann, wenn $\neg(S_1 \leftrightarrow S_2)$ unerfüllbar genau dann, wenn $\neg(S_1 \leftrightarrow S_2) \notin SAT$.

- Software-Paketverwaltung
  Installationsproblem (lässt sich ein Paket installieren? Geht bspw. nicht bei Ringabhängigkeiten): Angenommen, ein Paket A benötigt die Pakete B, C sowie eins der Pakete D, E und ist unverträglich mit F, dann kann dies dargestellt werden durch $A \rightarrow B \wedge C \wedge (D \vee E) \wedge \neg F$ ($\rightarrow$ bedeutet „benötigt“).
  Jedes der Pakete B, C, D, E besitzt wiederum Abhängigkeiten, die entsprechend durch Formeln dargestellt werden können. Sei G die $\wedge$-Verknüpfung aller dadurch entstehenden Formeln mit A und entsprechenden Atomformeln für die bereits auf dem System installierten Paketen. A ist genau dann installierbar, wenn G erfüllbar ist. Die erfüllende Belegung liefert die zu installierenden Pakete.

- Software Model Checking

```
1 int a[2];
2 if(i== 0) j = 1;
3 else j = 2;
4 x = a[j];
```

  Es besteht die Gefahr eines Index-out-of-Bounds Fehlers. Dieser muss abgefangen werden.
  ⇓

```
1 int a[2];
2 if(i==0) j=1;
3 else j = 2;
4 assert ( 0<=j && j<2);
5 x=a[j];
```

  Aus dem Programmcode wird die Formel $C = (i = 0 \rightarrow j = 1) \wedge (i \neq 0 \rightarrow j = 2)$ erzeugt, aus der Spezifikation die Formel $P = 0 \leq j \wedge j < 2$.

Hinweis: Travelling-Salesman-Problem (möglichst kurzer Rundweg durch Graph) (vgl. Hammilton Kreis) hat LZ $O(n^2 \cdot 2^n)$.

3.3 WIEDERHOLUNG (KLAUSUR)

3.3.1 CYK

$\{a^n b^n | n > 0\}$
$S \to aSb \mid ab$
Umformen in CNF:

1. $S \to ASB \mid AB$
   $A \to a$
   $B \to b$

2. $S \to XB \mid AB$
   $X \to AS$
   $A \to a$
   $B \to b$

Prüfen, ob $aabb$ drin liegt:

S			
X			
	S		
A	A	B	B
a	a	b	b

3.3.2 ENTSCHEIDBARKEIT

Die Frage ist, ob es für die Menge ein oder mehrere Entscheidungsverfahren gibt. Die Frage ist nicht, welches das richtige ist, sondern nur, dass es umfassend entscheidbar ist (Vergleich: Frage nach Gott – es gibt Entscheidungsverfahren, ober wir wissen nicht, welches das richtige ist, weil wir nicht wissen, ob Gott existiert).
NP-Vollständigkeit!

3.3.3 AUTOMATEN

In der Klausur müssen bei bspw. DFA keine Tupel angegeben werden, der vollständig beschriftete Graph reicht.
Reguläre Ausdrücke (Definition!)
DFA Konstruieren!
Kontextfreie Grammatiken (konstruieren)
Pumping-Lemma!

3.3.4 RECURSIVE DESCENT PARSER

Grammatik zu Problem ausdenken und in rekursive Prozeduren umsetzen.

3.3.5 OL-SYSTEME

Voraussichtlich keine Aufgabe dazu.