Theoretische Informatik Script

Markus Söhnel

Inhaltsverzeichnis

1 Automaten und formale Sprachen

Definition
Ein Alphabet $\sum$ ist eine endliche Menge die nicht leer ist. Mit $\sum^*$ bezeichnen wir alle Elemente (Wörter), die sich durch zusammenfügen von Symbolen aus $\sum$ bilden lassen. Die Länge eines Wortes ist die Anzahl der Symbole, aus denen es besteht. Das leere Wort bezeichnen wir mit ϵ

Beispiel
Alphabet $\sum = \{a, b, c\}$ $\sum^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, bbb, ccc...\}$
Alphabet $\sum = \{if, then, else, x, =, 0, 1, 5\}$ **Wörter:** if x = 0 then x = 1 else x = 5

Definition Eine formale Sprache über einem Alphabet $\sum$ ist eine Teilmenge von $\sum^*$.

Beispiel
Die Menge der Schlüsselwörter der Programmiersprache C (if, else, while, for, ...) ist eine formale Sprache über dem Alphabet {a..z}.
Die Menge der syntaktisch korrekten C Programme, ist eine formale Sprache. Diese lässt sich darstellen über dem Alphabet {a, ..., z, (,), {, }, =, !, &, ;, 0, ..., 9}
oder über {if, else, for, do, while, goto, ==, !=, <, >, <=, >=, «, », &, &&, ||, &, 0, ..., 9}

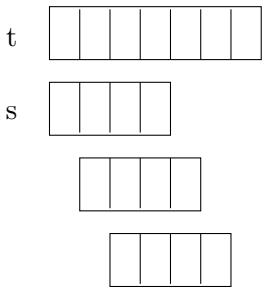
Definition
Für Wörter v, w ist vw die Konkatenation der Wörter v, w.
Das Wort w^n ist die n-fache Konkatenation von w. Dabei ist $w^0 = \epsilon$

Beispiel
$(a, b, c)^3 = abcabcabc$. Für ein Wort w gilt: $\sum w = w = w\epsilon$

Bemerkung
Es gilt $\epsilon \in \sum^*$ für alle $\sum$, da $\sum^*$ die Menge aller Wörter ist.

1.1 Reguläre Sprachen

Motivation
Suche nach einem Wort s in einem Text t.
Naiver Algorithmus:
Das Wort s Zeichen für Zeichen mit t vergleichen bei Mismatch eine Stelle weiterschieben.

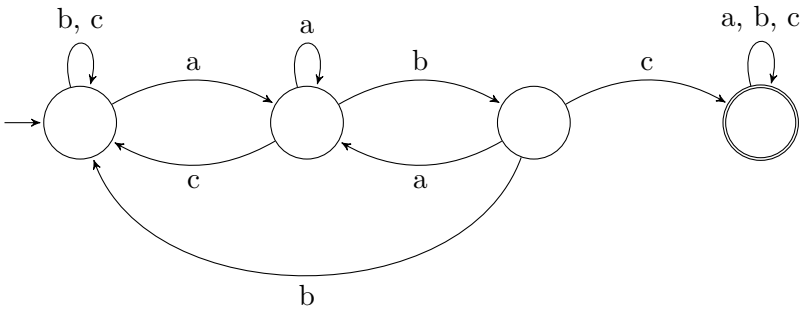


Laufzeit:
O(|s| * |t|) (schlechte Laufzeit)

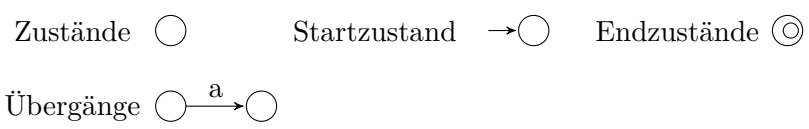
1.1.1 Deterministische Endliche Automaten (Deterministic Finite Automata, DFA)

Ein Automat ist ein formales Modell, um formale Sprachen zu verarbeiten. Ein DFA besteht aus endlich vielen Zuständen und Übergängen zwischen Zuständen. In jedem Schritt verarbeitet der DFA ein Zeichen und wechselt dabei den Zustand. Wenn sich der DFA dabei in einem Endzustand befindet, dann akzeptiert der DFA die Folge der bereits verarbeiteten Zeichen (Wort).

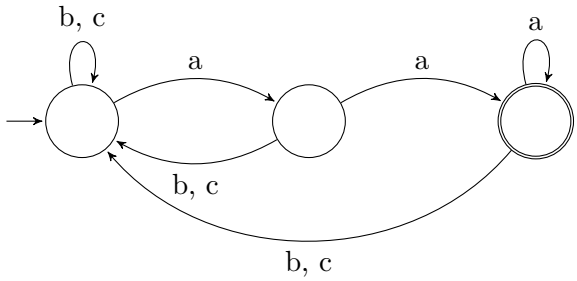
Beispiel



Dieser Automat akzeptiert alle Wörter über dem Alphabet $\sum = \{a, b, c\}$, die „abc“ enthalten. Dieser Automat beschreibt damit die formale Sprache aller Wörter, die äbc” enthalten. Zur Darstellung von DFAs verwenden wir folgende graphische Notation:



Weiteres Beispiel: DFA, der alle Wörter akzeptiert, die auf „aa“ enden.



Definition
Ein DFA ist ein Tupel

$$M = (Z, \sum, \delta, z_0, E)$$

- Z: Menge der Zustände
- $\sum$: Eingabe Alphabet
- δ: $Z \times \sum \to Z$ Überführungsfunktion
  Dabei bedeutet $\delta(z, a) = z'$. Der DFA geht mit $a \in \sum$ von z nach z' über.
- $z_0 \in Z$: Startzustand
- $E \in Z$: Endzustand

Beispiel
Der DFA M, der alle Wörter akzeptiert, die *abc* enthalten, lässt sich formal beschreiben durch

$$M = (\{z_0, z_1, z_2, z_E\}, \{a, b, c\}, \delta, z_0, \{z_E\})$$

wobei δ durch folgende Tabelle gegeben ist

δ	z_0	z_1	z_2	z_E
a	z_1	z_1	z_1	z_E
b	z_0	z_2	z_0	z_E
c	z_0	z_0	z_E	z_E

Definition
Sei $M = (z, \sum, \delta, z_0, z_E)$ ein DFA.

- Die erweiterte Überführungsfunktion $\hat{\delta} * Z \times \sum^* \Rightarrow Z$ von M ist definiert durch

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

- Die von M akzeptierte Sprache ist $L(M) = \{w \in \sum^* | \hat{\delta}(z_0, w) \in E\}$

Beispiel (Fortsetzung)
Mit δ wie oben erhalten wir

$\hat{\delta}(z_0, aababcc) = \hat{\delta}(\delta(z_0, a), ababcc)$

$= \hat{\delta}(z_1, ababcc) = \hat{\delta} = (\delta(z_1, a), babcc)$

$= \hat{\delta}(z_1, babcc) = \hat{\delta}(\delta(z_1, b), abcc)$

$= \hat{\delta}(z_1, abcc) = \hat{\delta}(\delta(z_2, a), bcc)$

$= \hat{\delta}(z_2, bcc) = \hat{\delta}(\delta(z_2, b), cc)$

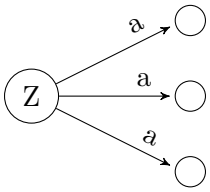
$= \hat{\delta}(z_0, cc) = \hat{\delta}(\delta(z_0, c), c)$

$= \hat{\delta}(z_E, \epsilon) = \hat{\delta}(\delta(z_E, \epsilon)) = \underline{\underline{z_E}}$

Wegen $z_E \in E$ gilt $aabacc \in L(M)$

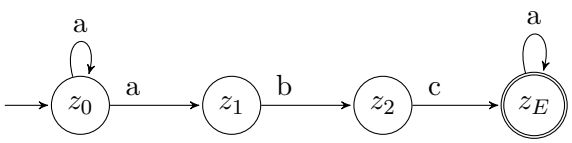
1.1.2 Nicht deterministische endliche Automaten (NFA)

Ein NFA ist eine Verallgemeinerung eines DFA. Während ein DFA für jedes Paar aus Zustand und gelesenem Zeichen genau einen Folgezustand besitzt, besitzt der NFA beliebig viele Folgezustände



Eine Möglichkeit, diesen Nichtdeterminismus zu verstehen, besteht darin, einen NFA als Modell zulässiger Zustandsfolgen zu betrachten.

Beispiel



So wie eine Straßenkarte mögliche Wege beschreibt, beschreibt auch ein NFA mögliche Zustandsfolgen bei der Verarbeitung eines Wortes. Insbesondere "weiß" der NFA nicht, welchen Folgezustand er aus-
wählen muss . Ein NFA lässt sich daher nicht unmittelbar als Programm implementieren.

Die von einem NFA M akzeptierte Sprache $L(M)$ besteht aus allen Wörtern $w \in \sum^*$, für die M einen Endzustand erreichen kann. Dabei müssen die Kanten entsprechend der Zeichen der Eingabe durchlaufen werden.

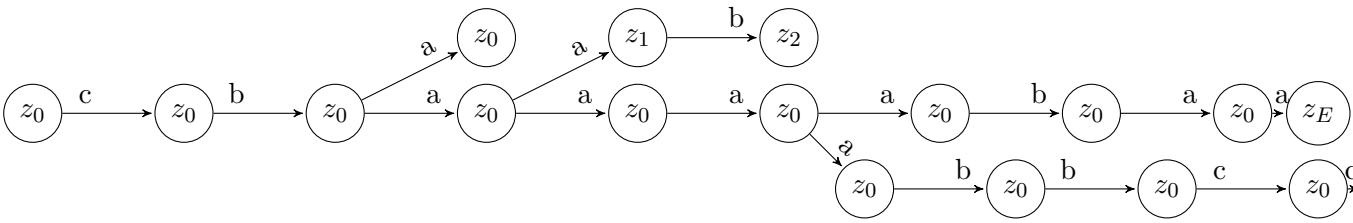
Beispiel (Fortsetzung)
Für die Eingabe $w = cbacbabcc$ kann der NFA die Zustandsfolge
$z_0, z_0, z_0, z_0, z_0, z_0, z_1, z_2, z_E, z_E$ durchlaufen. Da der letzte Zustand ein Endzustand ist, wird w akzeptiert, das heißt, $w \in L(M)$.

Für die Eingabe $abcabc$ gibt es zwei Zustandsfolgen, die zu z_E
führen($z_0, z_0, z_0, z_0, z_1, z_2, z_E$ sowie $z_0, z_1, z_2, z_E, z_E, z_E, z_E$). Daher gilt, $abcabc \in L(M)$

Für die Eingabe *abba* gibt es dagegen keine Zustandsfolge mit der ein Endzustand erreicht werden kann. Daraus folgt, dass $abba \notin L(M)$.

Eine weitere Möglichkeit, den Nichtdeterminismus eines NFA zu verstehend besteht darin, einen NFA als Modell zu betrachten, das parallele Berechnungen beschreibt. Die durch einem NFA beschriebenen
möglichen Zustandsübergänge lassen sich dann durch einen Berechnungsbaum darstellen.

Beispiel (Fortsetzung)
Berechnungsbaum für die Eingabe *cbaababcc*.

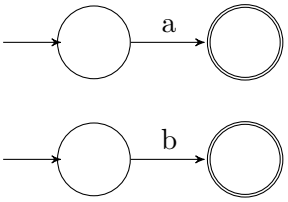


Für einen NFA lässt sich die Überführungsfunktion definieren durch

$$\delta * Z \times \sum \to P(Z)$$

wobei $z' \in \delta(z, a)$ bedeutet. Wenn der NFA sich im Zustand z befindet und das Zeichen a erhält, dann kann er den Zustand z' wechseln.
Ferner besitzt ein NFA einen oder mehrere Startzustände.

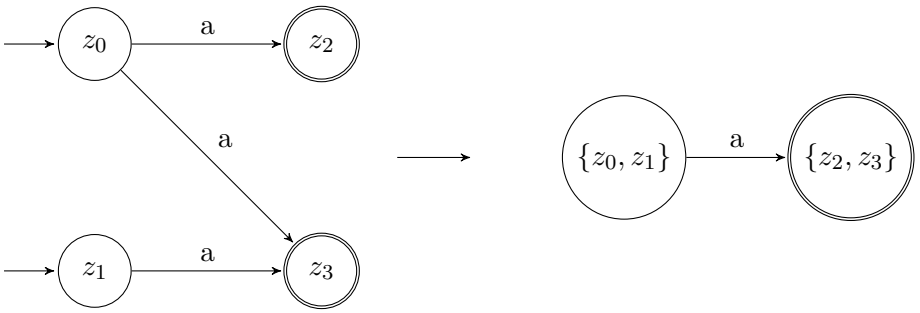
Beispiel



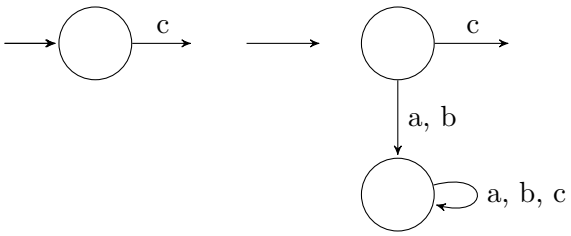
NFA, der die Sprache $\{a, b\}$ akzeptiert.

Umwandlung eines NFA in einen DFA

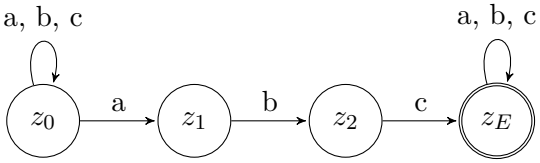
Es gilt: Für jeden NFA gibt es einen DFA, der die gleiche Sprache erkennt.
Idee zur Umwandlung: Wir vereinigen mögliche Folgezustände des NFA zu einem Zustand des DFA



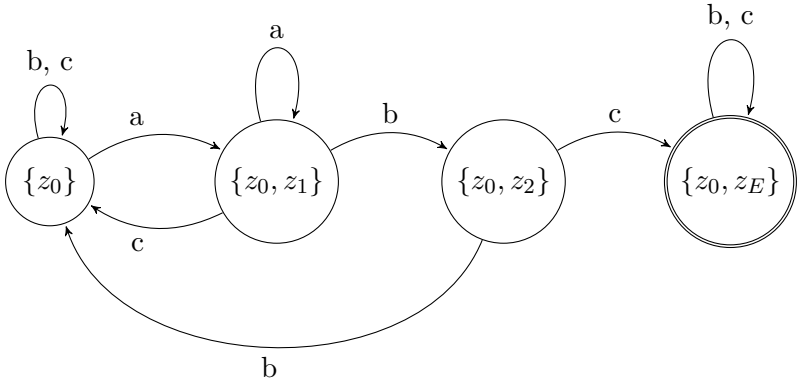
Falls es für einen Zustand z und ein Zeichen a keinen Folgezustand gibt (d.h. $\delta(z, a) = 0$), führen wir einen Fehlerzustand ein, der nicht mehr verlassen werden kann



Beispiel
Umwandlung des NFA



In einen DFA



$\{z_0, z_1\} \cup 0 = \{z_0, z_1\}$
$\{z_0\} \cup 0 = \{z - 0\}$
$\{z_0\} \cup \{z_2\} = \{z_0, z_2\}$

Der Zustand $\{z_0, z_E\}$ ist ein Endzustand, weil er einen Endzustand des NFA enthält. Alle
Zustände, die von $\{z_0, z_E\}$ ausgehen, enthalten z_E und sind damit ebenfalls Endzustände.

Diese können vereinigt werden, ohne die vom Automaten erkannte Sprache zu verändern.

Da die Potenzmenge $P(Z)$ der Zustandsmenge des NFA $2^{|z|}$ Elemente enthält, kann der aus einem NFA umgewandelte DFA im schlimmsten Fall $2^{|z|}$ viele Zustände enthalten. Es ist möglich,
dass sich darunter gleichwertige Zustände befinden, die zusammengefasst werden können.

Mit dem Algorithmus Minimalautomat kann aus einem DFA ein Automat erzeugt werden, der die gleiche Sprache erkennt und der minimal bezüglich der Anzahl Zustände ist (Minimalautomat).

Je zwei Minimalautomaten unterscheiden sich höchstens in der Benennung der Zustände.

1.1.3 Reguläre Ausdrücke

Definition
Sei $\sum$ ein Alphabet. Ein regulärer Ausdruck E über $\sum$ sowie die durch E erzeugte Sprache $L(E)$ sind induktiv definiert.

- $\emptyset$ ist ein regulärer Ausdruck $L(\emptyset) = \emptyset$
- Für jedes $a \in \sum \cup \{\epsilon\}$ 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 Ausrücke und
  - $L(E_1|E_2) = L(E_1) \cup L(E_2)$,
  - $L(E_1E_2) = L(E_1)L(E_2)$ dabei ist
    $L(E_1)L(E_2) = \{w_1w_2 | w_1 \in L(E_1), w_2 \in L(E_2)\}$
  - $L(E_1^*) = (L(E_1))^*$

  Um Klammern zu sparen, legen wir folgende Regeln für die Priorität der Operatoren fest. Die höchste Priorität besitzt der Operator "$*$", gefolgt von Konkatenation, gefolgt vom Operator "$|$"

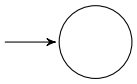
Beispiel
$(a|b)^*$ ist ein regulärer Ausdruck und $L((a,b)^*)$
$= (L(a|b))^*$ (3. Regel)
$= (L(a) \cup L(b))^*$ (3. Regel)
$= (\{a\} \cup \{b\})^*$ (2. Regel)
$= \{a,b\}^*$

Satz
Für jeden regulären Ausdruck E gibt es einen NFA M mit $L(E) = L(M)$

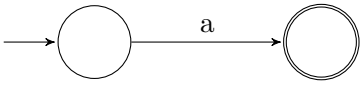
Beweis
Wir indizieren über den Aufbau regulärer Ausdrücke.

Induktionsanfang

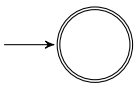
- Für $E = 0$ ist M folgender NFA



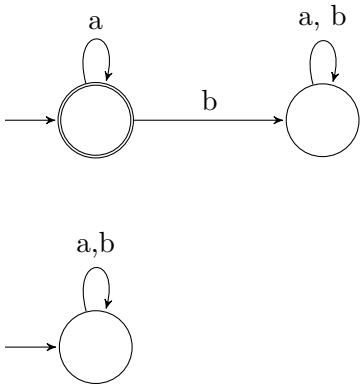
- Für $E = a$ ist M der NFA



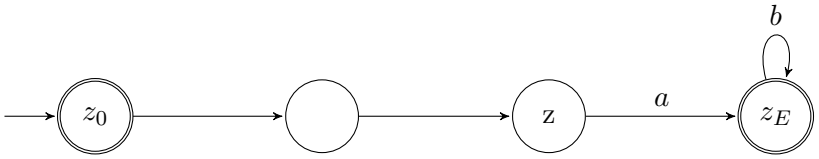
Für $E = \epsilon$ ist M der NFA



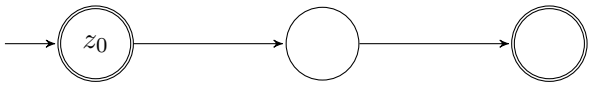
Induktionsschritt
Seien E_1, E_2 reguläre Ausdrücke und nach Induktionsvoraussetzung M_1', M_2' NFAs mit $L(E_1) = L(M_1'), L(E_2) = L(M_2')$. Ferner seien M_1, M_2 DFAs mit
$L(M_1) = L(M_1'), L(M_2) = L(M_2')$



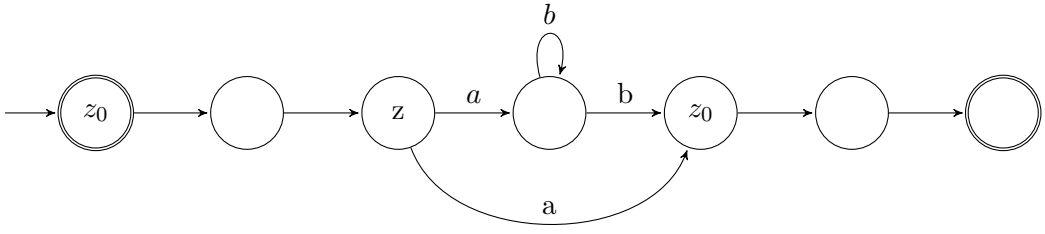
- $E_1|E_2$: Der NFA für $E_1|E_2$ ist die Vereinigung von M_1, M_2, da ein NFA mehrere Startzustände haben darf
- E_1E_2: Skizze zur Idee



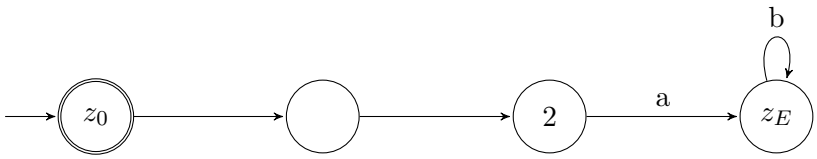
M_1



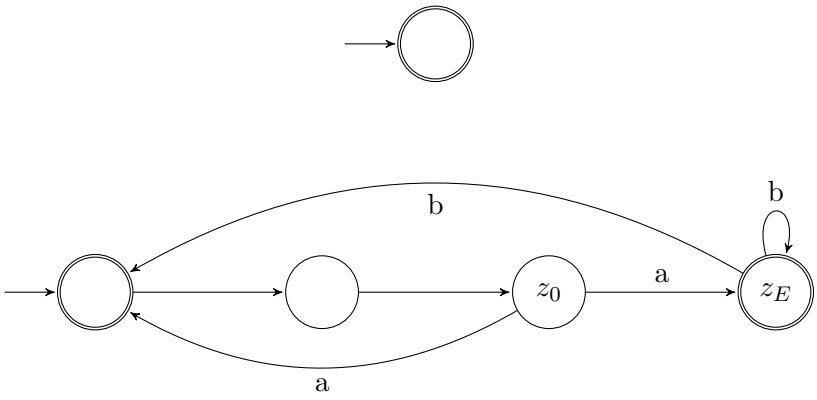
M_2



Kein Formaler Beweis
Skizze zum Beweis

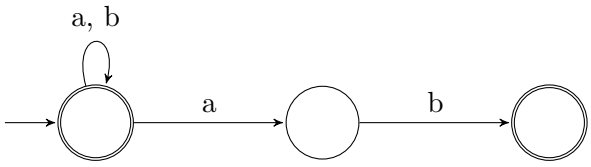


NFA:



Ebenso gilt:
Satz
Für jeden NFA M gibt es einen regulären Ausdruck E mit $L(E) = L(M)$
Ohne Beweis

Beispiel
Sei M der NFA

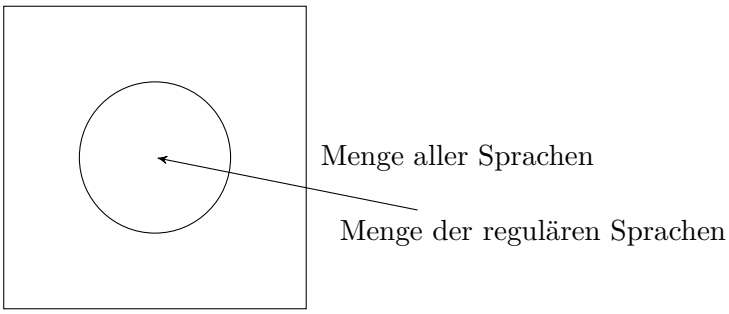


Ein regulärer Ausdruck E mit $L(E) = L(M)$ ist

$$\underline{\underline{(a|b)^*ab}}$$

Definition
Die Menge der regulären Sprachen ist die Menge der Sprachen, die von einem DFA erkannt werden.

Folgerung aus dem bisher aufgeschriebenen Satzes. Die NFAs erkennen genau die regulären Sprachen. Die regulären Ausdrücke erzeugen genau die regulären Sprachen.



Lexer
Ein Lexer ist ein Tool, das aus einem regulärem Ausdruck einen DFA erzeugt, der die gleiche Sprache erkennt. Damit können Akzeptoren für Wörter oder Muster konstruiert werden.

Arbeitsweise eines Lexers:
Regulärer Ausdruck $\longrightarrow$ NFA $\longrightarrow$ DFA

1.2 Nicht regulärer Sprachen

Satz
Die Sprache $L\{a^n b^n | n \geq 0\}$ ist nicht regulär. (Der Automat hat nur eine endliche Anzahl von Zuständen)

Beweis
Angenommen L sei regulär. Dann gibt es einen DFA M mit $L(M) = L$. Nach dem lesen von a^n befindet sich der DFA M in einem von $|Z|$ Zuständen.

Da es mehr Präfixe a^n als Zustände gibt, folgt aus dem Schubfachprinzip: Es gibt zwei verschiedene Wörter
a^{n_1}, a^{n_2}, so dass sich M nach dem lesen von a^{n_1} bzw. a^{n_2} im gleichen Zustand z befindet.

Da M nach Annahme $a^{n_1}b^{n_1}$ akzeptiert, gelangt M von z aus durch das Lesen von b^{n_1} in den Endzustand. Dann akzeptiert M jedoch auch das Wort $a^{n_2}b^{n_1}$ - WIDERSPRUCH, da $n_1 \neq n_2$

1.3 Kontextfreie Sprachen

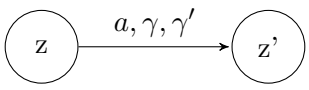
1.3.1 Kellerautomaten (PDAs)

Ein Kellerautomat (Pushdown Automat PDA) besitzt gegenüber einem NFA zwei zusätzliche Eigenschaften

- Ein Kellerautomat kann den Zustand wechseln, ohne dabei ein Eingabezeichen zu lesen (ϵ-Übergänge)
- Ein Kellerautomat besitzt ein Stack (oder Keller), in dem er eine unbegrenzte Anzahl von Zeichen speichern kann. Der Stack ist eine LIFO Datenstruktur

Ferner gibt es nun einen Startzustand, was wegen der ϵ-Übergänge keine Einschränkung ist.

Graphische Darstellung von PDAs



Ein Übergang der Beschriftung a, γ, γ' bedeutet:

- Der PDA ließt das Zeichen a der Eingabe, entfernt das oberste Stackzeichen γ und schreibt γ' auf den Stack

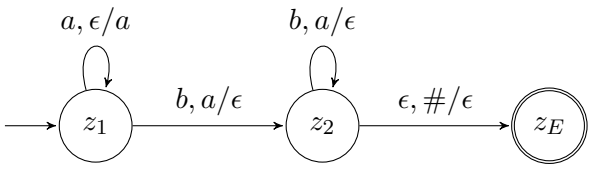
$\boxed{\#}$

Jedes der Zeichen a, γ, γ' kann ϵ sein. Für

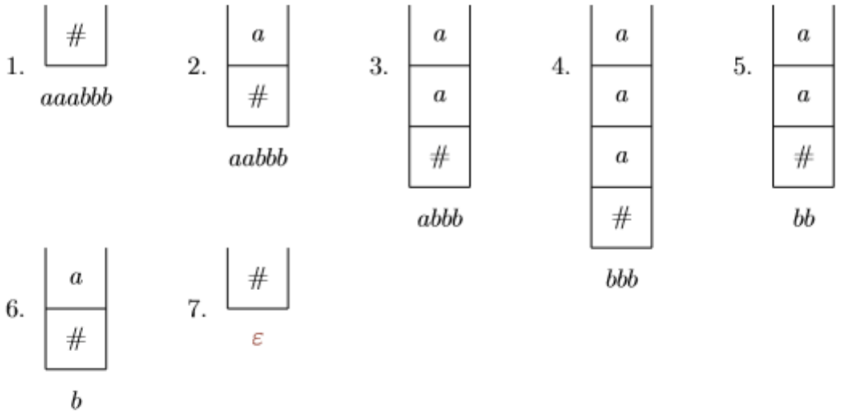
- $a = \epsilon$ kann der PDA diesen Übergang ausführen, ohne ein Zeichen der Eingabe zu lesen
- $\gamma = \epsilon$ kann der PDA diesen Übergang ausführen, ohne das oberste Stackzeichen zu entfernen
- $\gamma' = \epsilon$ schreibt der PDA kein Zeichen auf den Stack

Damit ein PDA einen leeren Stack erkennen kann, wird dieser das Ende des Stacks mit dem untersten Stackzeichen # markieren. Zu Beginn jeder Rechnung enthält der Stack nur das Zeichen #.
Ein PDA akzeptiert eine Eingabe, wenn er mit dieser einen Endzustand erreicht

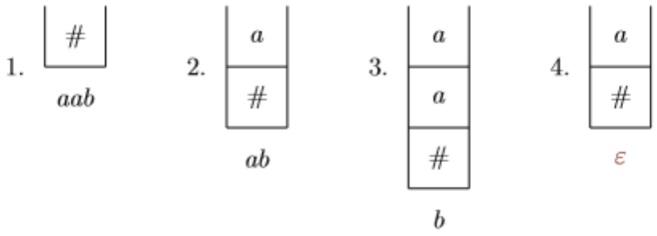
Beispiel
Ein PDA, der die Sprache $\{a^n b^n | n \geq 1\}$ akzeptiert. Für jedes gelesene a wird dazu ein a auf den Stack geschoben, für jedes gelesenes b ein a vom Stack entfernt.
Danach muss der Stack leer sein (da es genauso viele a wie b gibt)



Verhalten für die Eingabe *aaabbb*



Verhalten für aab



Verhalten für abb

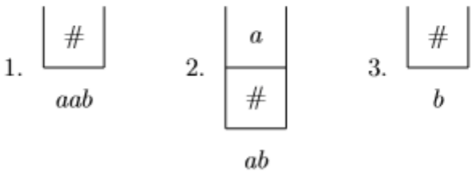
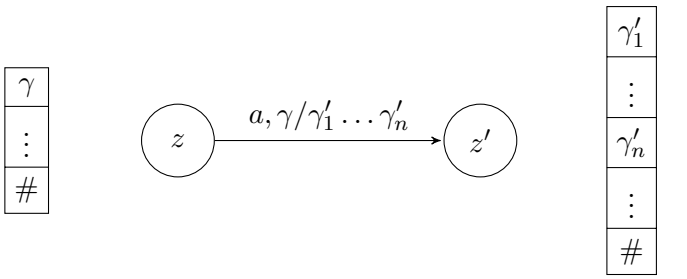


abb wird nicht akzeptiert, weil die Eingabe nicht bis zum Ende gelesen werden kann (ein b ist noch übrig)

Im folgenden erlauben wir, dass ein PDA mehrere Zeichen auf den Stack schreibt. Wir schreiben $a, \gamma/\gamma'_1 \dots \gamma'_n$, wenn der PDA zuerst γ_n und zuletzt γ_1 auf den Stack schreibt.



Vorher Nachher

Definition
Die von einem PDA M akzeptierte Sprache $L(M)$ ist die Menge aller $w \in \sum^*$ für die gilt:
Der PDA M kann, ausgehend vom Startzustand
und den initialen Stackinhalt #, durch das lesen des Wortes w einen Endzustand erreicht

Bemerkung
Die deterministischen PDAs (DPDAs) sind weniger leistungsfähig als nicht deterministische PDAs. Insbesondere lassen sich PDAs nicht umwandeln in DPDAs.

1.3.2 Kontextfreie Grammatiken

Jede Grammatik besitzt ein Startsymbol und Ersetzungsregeln der Form

$$\text{linke Seite} \quad \rightarrow \quad \text{rechte Seite}$$

Beginnend mit dem Startsymbol können diese Regeln solange angewendet werden, bis keine Regel mehr anwendbar ist.

Bei einer Kontextfreien Grammatik muss die linke Seite eine Variable sein.

Beispiel
Wir betrachten eine Grammatik, die aus den Variablen

- Satz
- Norminalphase
- Verbalphase
- Artikel
- Nomen
- Verb
- sowie die Terminalzeichen *die, Katze, Maus, jagt*

besteht. Das Startsymbol ist Satz, die Regeln sind

$$\text{Satz} \rightarrow \text{Norminalphase} \quad \text{Verbalphase}$$

$$\text{Artikel} \rightarrow \text{die}$$

$$\text{Nomen} \rightarrow \text{Katze}$$

$$\text{Nomen} \rightarrow \text{Maus}$$

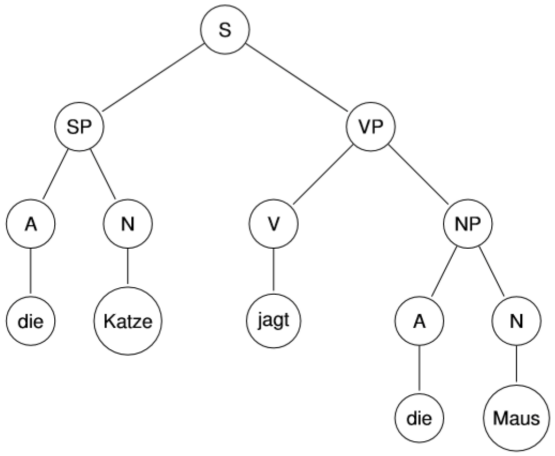
$$\text{Verbalphase} \rightarrow \text{Verb}$$

$$\text{Verbalphase} \quad \rightarrow \quad \text{Verb Nominalphase}$$

Mögliche Ableitung:

$$\begin{gathered}
\text{Satz} \quad \Rightarrow \quad \text{Nominalphase Verbalphase} \Rightarrow \\
\text{Artikel Nomen Verbalphase} \quad \Rightarrow \\
\text{die Nomen Verbalphase} \quad \Rightarrow \\
\text{die Katze Verbalphase} \quad \Rightarrow \\
\text{die Katze Verb Nominalphase} \quad \Rightarrow \\
\text{die Katze jagt Nominalphase} \quad \Rightarrow \\
\text{die Katze jagt Artikel Nomen} \quad \Rightarrow \\
\text{die Katze jagt die Nomen} \quad \Rightarrow \\
\text{die Katze jagt die Maus} \quad \Rightarrow
\end{gathered}$$

Syntaxbaum dazu



Definition
Eine kontextfreie Grammatik ist ein Tupel $G = (V, \sum, P, S)$, wobei gilt

- V ist die Menge der Variablen oder Nonterminalzeichen
- $\sum$ ist das Alphabet mit $V \cap \sum = \emptyset$. Die Elemente aus $\sum$ heißen auch Terminalzeichen

- P ist die Menge der Regeln (oder Produktionen) der Form $u \to v$, wobei $u \in V, v \in (V \cup \sum)^*$
- $S \in V$ ist das Startsymbol

Beispiel (Fortsetzung)
Die Grammatik lässt sich als Tupel $G = (V, \sum, P, S)$ darstellen mit
$V = \{$Satz, Nominalphase, Verbalphase, Verb, Artikel, Nomen$\}$,
$\sum = \{$die, Katze, Maus, jagt$\}$,
$P =$ wie V,
$S =$ Satz

Wir schreiben $x \Rightarrow y$, wenn sich aus $x \in (V \cup \sum)^*$ durch die Anwendung genau einer Regel $y \in (V \cup \sum)^*$ erzeugen lässt.

Beispiel
Es gilt

$$\text{Satz} \Rightarrow \text{Nominalphase Verbalphase} \Rightarrow \text{Artikel Name Verbalphase}$$

Definition
Die Relation $\Rightarrow^*$ ist die reflexive und transitive Hülle der Relation $\Rightarrow$ (auf $(V \cup \sum)^*$)

Beispiel
Es gilt

$$\text{Satz} \Rightarrow^* \text{Satz}$$

$$\text{Satz} \Rightarrow^* \text{Artikel Nomen Verbalphase}$$

$$\text{Satz} \Rightarrow^* \text{die Katze jagt die Maus}$$

Definition
Die von einer kontextfreien Grammatik G erzeugte Sprache ist

$$L(E) = \{w \in \Sigma^* | S \Rightarrow^* w\}$$

Wenn in einer kontextfreien Grammatik eine linke Seite durch verschiedene rechte Seiten ersetzt werden kann, verwenden wir das Zeichen | (oder) um Alternativen anzugeben

Beispiel
Die Grammatik mit den Regeln

$$S \to \epsilon$$
$$S \to SS$$
$$S \to [S]$$

können wir damit kürzer darstellen durch

$$S \to \epsilon | SS | [S]$$

Beispiel
Durch die Grammatik mit den Regeln

$$S_N \to 0|1|\dots|9|0S_N|1S_N|\dots|9S_N$$

und dem Startsymbol S_N können wir Zahlen aus $\mathbb{N}_0$ darstellen. Die Zahl 120 lässt sich ableiten durch

$$S_N \Rightarrow 1S_N12S_N \Rightarrow 120$$

Diese Regeln verwenden wir in einer weiteren Grammatik, um arithmetische Ausdrücke zu erzeugen.

$$\underbrace{\underbrace{2*3}_{\text{Arithmetischer Ausdruck}} + \underbrace{3*14}_{\text{Arithmetischer Ausdruck}}}_{\text{Arithmetischer Ausdruck}}$$

Vorüberlungen

- Die Operatoren $+$, $-$, $*$, / sind binär, weshalb vor und nach jedem Operator ein arithmetischer Ausdruck stehen muss
- Auf jede öffnende Klammer muss eine schließende Klammer folgen

Damit erhalten wir die Grammatik mit den Regeln

$$S_E \to S_N|(S_E)|S_E + S_E|S_E - S_E|S_E * S_E|S_E/S_E$$

Der Ausdruck $2 * (3 + 4)$ lässt sich ableiten durch

$S_E \Rightarrow S_E{*}S_E \Rightarrow S_N{*}S_E \Rightarrow 2{*}S_E \Rightarrow 2{*}(S_E) \Rightarrow 2{*}(S_E{+}S_E) \Rightarrow 2{*}(S_N{+}S_N) \Rightarrow 2{*}(3{+}4)$

Definition
Eine Sprache L heißt kontextfrei, wenn es eine kontextfreie Grammatik G gibt mit $L(G) = L$.

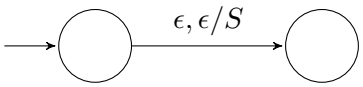
1.3.3 PDAs und kontextfreie Grammatiken

Satz
Kellerautomaten (PDAs) akzeptieren genau die kontextfreien Sprachen.

Beweis

- Für jeden PDA M gibt es eine kontextfreie Grammatik G mit $L(M) = L(G)$
  - Ohne Beweis -
- Für jede kontextfreie Grammatik G gibt es einen PDA M mit $L(M) = L(G)$

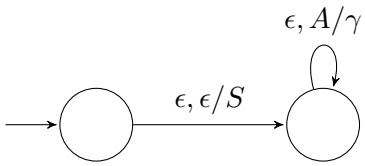
  Idee: M simuliert auf seinem Stack Ableitungen
  aus der Grammatik G. Wir konstruieren einen PDA M mit drei Zuständen wie folgt



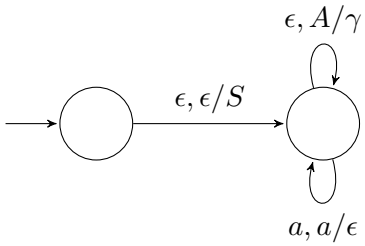
  – Zuerst schreibt M das Startsymbol S auf den Stack und wechselt in einen weiteren Zustand.

  In diesem Zustand unterscheiden wie drei Fälle: Das oberste Stackzeichen ist

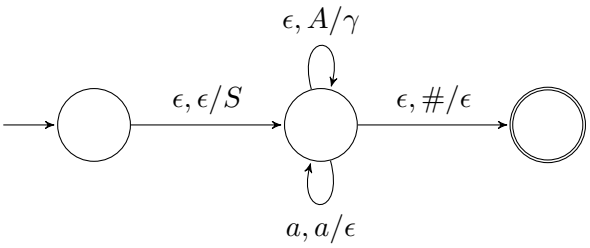
  – eine Variable A. Wenn es eine Regel $A \to \gamma$ in G gibt, kann M das oberste Stackzeichen A durch γ ersetzen



  – ein Zeichen $\in \sum$, das mit den nächsten Zeichen der Eingabe übereinstimmt. Dann wird a vom Stack entfernt.



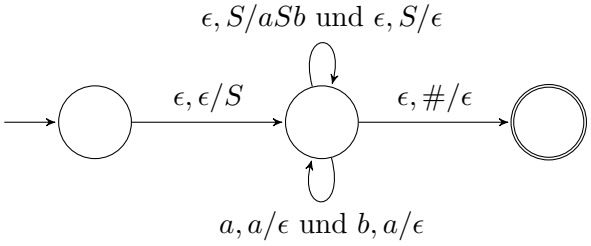
– #: Dann geht M in den Endzustand über



Beispiel
Für die Sprache $L = \{a^n b^n | n > 0\}$ konstruieren wir einen PDA M mit $L(M) = L$.
L wird erzeugt vor der Grammatik mit folgenden Regeln

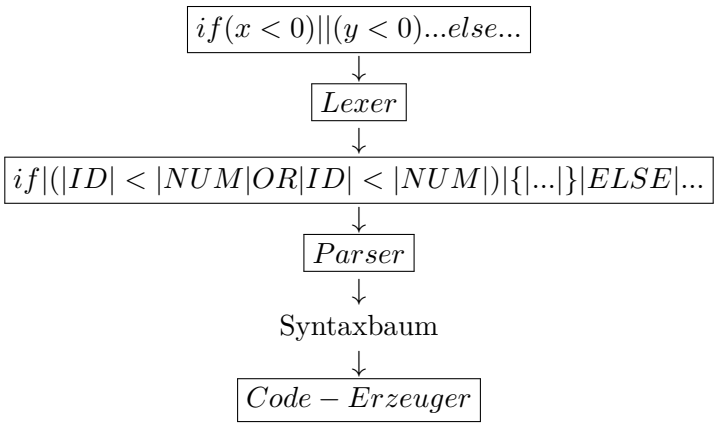
$$S \to aSb|\epsilon$$

Aus obigen Beweis erhalten wird den PDA



1.3.4 Syntaxanalyse

Arbeitsweise eines Compilers:

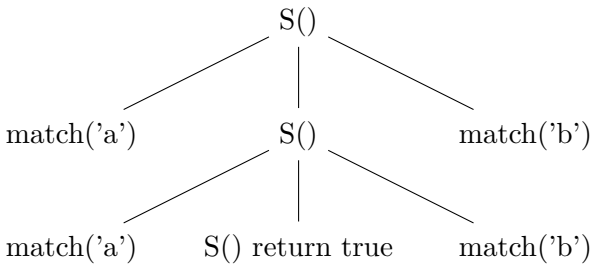


Parser lassen sich in zwei Klassen unterteilen

Top-Down Parser
Ein Top-Down Parser erzeugt einen Syntaxbaum von oben nach unten.
Ein Top-Down Parser arbeitet wie das Verfahren aus dem Beweis zu Umwandlung einer kontextfreien Grammatik in einem PDA. Ein Top-Down Parser lässt sich durch die rekursive Prozeduren implementieren von denen jede eine Regel der Grammatik entspricht. Der Callstack übernimmt die Funktion des Stack des PDA.

Wir erlauben dem Parser die k nächsten Zeichen der Eingabe zu sehen (lookahead von k), um davon abhängig eine Regel auszuwählen. Beim Verarbeiten der Eingabe baut ein Top-Down Parser implizit den Syntaxbaum von oben nach unten auf.

Beispiel
Parser für $\{a^n b^n | n \leq 0\}$, Eingabe = $aabb$



Der Callgraph
Der Callgraph besitzt die gleiche Struktur wie der Syntaxbaum.

Ein Parser, der durch rekursiv Funktionen einen Syntaxbaum von oben nach unten aufgebaut hat, heißt Rekursiv Descent Parser.

1.3.5 Mehrdeutigkeit

Definition
Eine Grammatik
G heißt mehrdeutig, wenn es ein $w \in L(G)$ gibt, für das zwei Ableitungsbäume existieren.

Beispiel
Die Grammatik für arithmetische Ausdrücke mit den Regeln

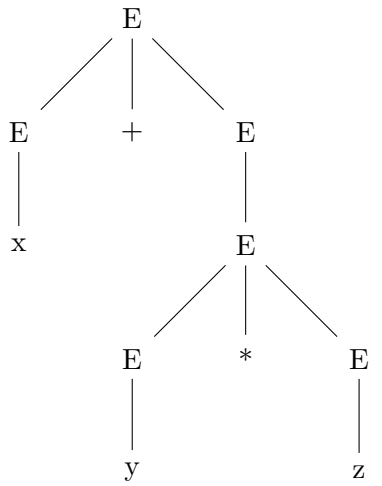
$$E \to E + E | E - E | E * E | E / E | (E) | x | y | z$$

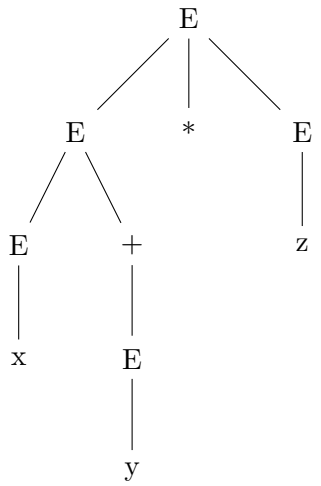
ist mehrdeutig, denn der Ausdruck $x + y * z$ besitzt zwei Ableitungsbäume

Auch wenn nur der Operator „-“ betrachtet wird,
ist die Grammatik mehrdeutig, denn der Ausdruck $x - y - z$ besitzt die Ableitungsbäume.

Mit obiger Grammatik gibt es mehrere Probleme:

- Die Grammatik berücksichtigt nicht die Priorität der Operatoren (Punkt vor Strich!)



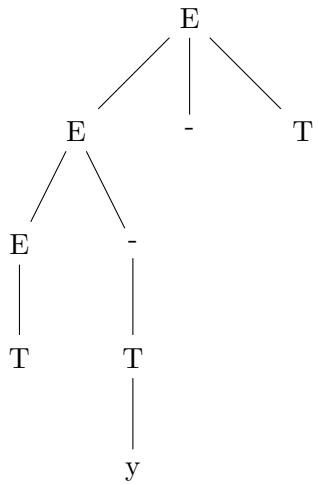


- sie berücksichtigt nicht die Assoziativität der Operatoren (alle Operatoren müssen links-assoziativ sein)

Um das Problem der Priorität zu lösen, definieren wir eine Grammatik, bei der sich Operatoren niedrigere Priorität oben im Ableitungsbaum und Operatoren höhere Priorität weiter unten im Ableitungsbaum befinden müssen.
Wir führen dazu eine Variable T (Term) ein, aus der Produkte abgeleitet werden können. Für die Funktionen aus E gibt es folgende Möglichkeiten

1. $E \to E - T | E + T | T$
   oder
2. $E \to T - E | T + E | T$

Sowohl mit (1) als auch mit (2) ist gewährleistet, dass die Operationen $+, -$ oben im Ableitungsbaum vorkommen. Mit (1) ist folgende Ableitung möglich



entspricht
$(T - T) - T$, d.h.
$-$ ist links-assoziativ

Nur Grammatik (1) berücksichtigt sowohl die Priorität als auch die Assoziativität der Operatoren $+, -$ in korrekter Weise. Entsprechend definieren wir Regeln für T und erhalten

$$E \to E - T|E + T|T$$

$$T \to T * F|T/F|F$$

$$F \to (E)|x|y|z$$

Diese Grammatik ist eindeutig und berücksichtigt Priorität und Assoziativität der Operatoren. Der eindeutige Ableitungsbaum für $x + y * z$ ist

Bottom-up Parser
Ein Bottom-up Parser baut einen Ableitungsbaum von unten nach oben auf. Diese lassen sich effizient realisieren durch LR-Parser. Ein LR-Parser ließt die Eingabe von links nach rechts und fährt in jedem Schritt eine von vier möglichen Aktionen aus:

- Shift: Das nächste Zeichen der Eingabe wird auf den Stack geschoben
- Ein oder mehrere Symbole an der Spitze des Stacks entsprechend der rechten Seite γ einer Regel $A \to \gamma$ und werde durch A ersetzt.
- Accept: Die Eingabe wurde verarbeitet und der Stack enthält nur das Startsymbol
- Error: Ein Syntaxfehler wurde gemeldet

Um zu entscheiden,
welche Aktion (Shift oder Reduce) auszuführen ist, verwendet der Parser eine Parsertabelle.

Beispiel
Mit der eindeutigen Grammatik von oben und der Eingabe $x + y + z$ wird ein LR-Parser folgende Aktionen aus

Stack Top	Restliche Eingabe	Aktion (s = shift, r = reduce)
	$x + y * z$	s
x	$+y * z$	r
F	$+y * z$	r
T	$+y * z$	r
E	$+y * z$	s
$E+$	$y * z$	s
$E + y$	$*z$	r
$E + F$	$*z$	r
$E + T$	$*z$	s
$E + T*$	z	s
$E + T * z$		r
$E + T * F$		r
$E + T$		r
E		accept

Die vom Parser konstruierte Rechtsableitung lässt sich aus den ersten beiden Spalten von unten nach oben ablesen. Ebenso kann der Parser den Wert eines Ausdrucks berechnen, wenn Zwischenwerte in den Symbolen gespeichert.
Zur Konstruktion von LR-Parsern werden Tools wie Yacc, Bison, CUP verwendet.

Beispiel
$E \to E + T|T|T - E$
$T \to T * F|T/F|F$

```
1           boolean E()
2           {
3               return E() && match('+') && T();
4           }
```

Wenn wir die eindeutige Grammatik für arithmetische Ausdrücke in einen Recursive Descent Parser überführen wollen, stellen sich zwei Probleme

- Es ist nicht klar erkennbar, welche Regel angewendet werden soll
- Die Regel $E \to E + T$ ist linksrekursiv. Diese führt zu einer endlosen Rekursion.

Wir brauchen daher eine andere Grammatik.

Mögliche Lösung
$E \to T + E|T - E|T$
Dann sind die
Operatoren jedoch rechtsassoziativ, also würde diese Grammatik falsche Ergebnisse berechnen.

EBNF (Erweiterte Backus-Naur-Form

Wir betrachten Ableitungen aus E.

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \cdots \Rightarrow T + \cdots + T$$

In EBNF lässt sich dies darstellen durch

$$E \to T\{+T\} \qquad \text{Alternative Notation: } E \to T(+T)^*$$

Dabei bedeutet $\{x\}$ beliebig viele Vorkommen von x.
Die Grammatik für arithmetische Ausdrücke lässt sich in EBNF wie folgt darstellen

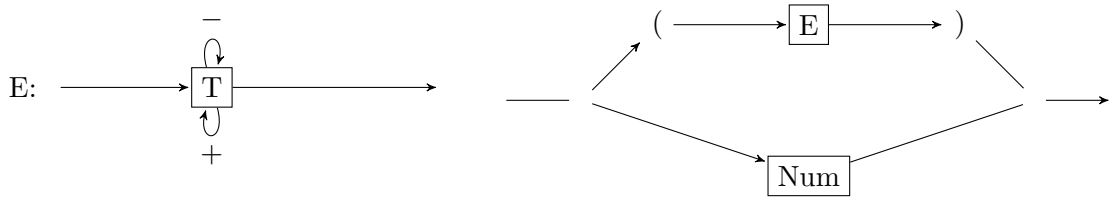
$$E \to T\{+T\}|T\{-T\}$$

$$T \to F\{*F\}|F\{/F\}$$

$$F \to (E)|Num$$

$$Num \to Z\{Z\} \qquad Z = 0|\ldots|9$$

Aus der Darstellung in EBNF lassen sich Syntaxdiagramme ableiten



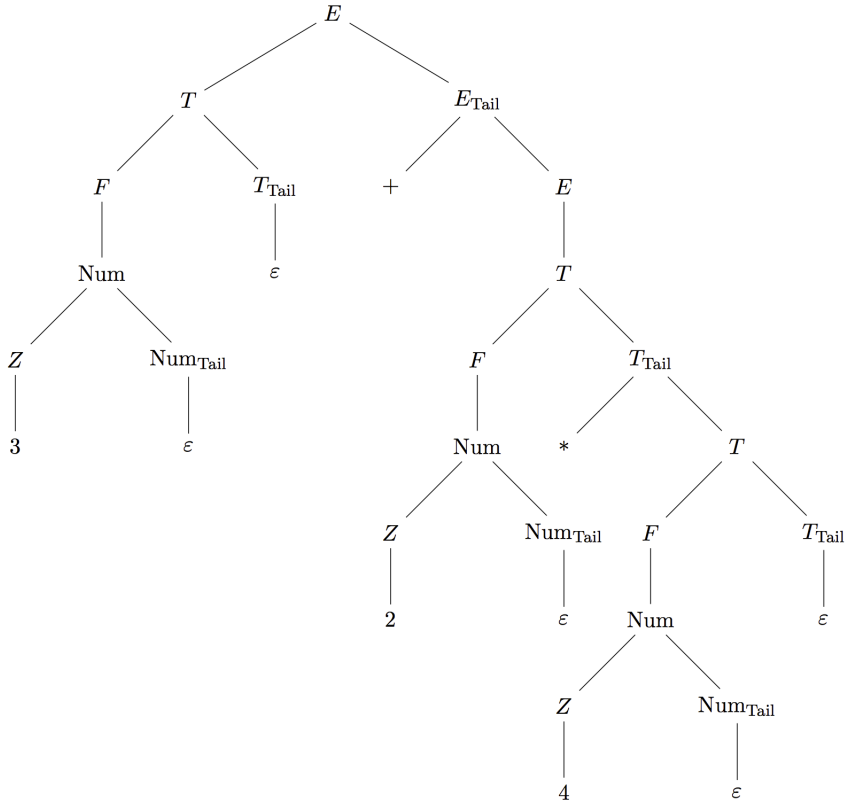
Problem: Aus der Darstellung in EBNF oder den Syntaxdiagrammen ist nicht ersichtlich, welche Assoziativität die Operatoren haben. Ohne weitere Annahme (z.B.

alle Operatoren links assoziativ) ist diese Grammatikbeschreibung nicht eindeutig.

Wir wollen nun eine eindeutige Grammatik für Ausdrücke in Infix-Notation konstruieren, die eindeutig und nicht linksrekursiv ist.

$E \to TE_{Tail}$	$F \to (E) \vert Num$
$E_{Tail} \to \epsilon \vert + E \vert - E$	$Num \to ZNum_{Tail}$
$T \to FT_{Tail}$	$Num_{Tail} \to \epsilon \vert Num$
$F_{Tail} \to \epsilon \vert * T \vert / T$	$Z \to 0 \vert \ldots \vert 9$

Beispiel
Ableitung des Ausdrucks $3 + 2 * 4$



OL-Systeme

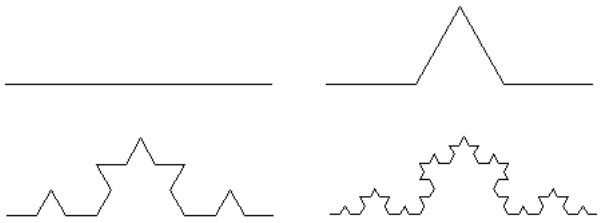
Eine Turtle besitzt eine Position in der Ebene und eine Orientierung. Zeichenbefehle sind

- forward(l)

- left(a) bzw. right(a). Direkt die Turtle

Ein OL-System besteht, wie eine kontextfreie Grammatik, aus Variablen, Terminalzeichen und Regeln. In jedem Ableitungsschritt müssen jedoch alle Variablen ersetzt werden durch eine Regel

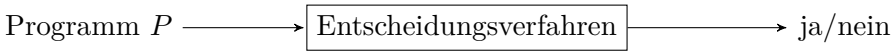
Beispiel Koch Kurve
Variable: F
Terminalzeichen: $+, -$
Regeln: $F \to F + F - -F + F$



2 Berechenbarkeit und Komplexität

2.1 Entscheidbarkeit

Ist es möglich, durch einen Algorithmus festzustellen, ob es ein Programm P eine bestimmte Eigenschaft besitzt?



Eigenschaften können sein

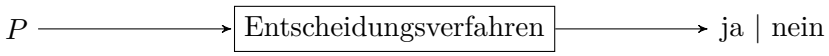
- Das Programm P stürzt ab
- Das Programm P liefert immer eine Antwort
- Das Programm P terminiert immer

Beispiel

```
1       for(k >= 3)
2           for(x, y, z := 1 to k)
3               for(n = 3 to k)
4                   if(x^n + y^n = z^n) stop;
```

Dieses Programm sucht nach einem Gegenbeispiel zu der von Fermat aufgestellten Behauptung (Fermats letzter Satz), dass die Gleichung $x^2 + y^2 = z^2$ keine Lösung in $x, y, z \in \mathbb{N}$ für $n \geq 3$ besitzt. Dies war für über Jahrhunderte ein ungelöstes Problem. Offenbar gilt: Das Programm hält genau dann wenn Fermats letzter Satz falsch ist.

Weiteres Problem: Ist es entscheidbar, ob ein Programm P „Hello World“ ausgibt?



Betrachtet dazu das Programm

```
1      void P()
2      {
3       fermat();
4       printf("Hello World");
5      }
```

Da P() genau dann „Hello World“ ausgibt, wenn fermat() terminisiert, ist dieses Problem mindestens so schwierig wie das Halteproblem.

Um Berechenbarkeit zu untersuchen, verwenden wir Programme auf einem abstraktem Computer mit unbegrenztem Speicher und Variabler, die beliebig große Werte annehmen können, als Berechnungsmodell.

Eine Sprache L heißt entscheidbar, wenn es ein Programm P_L gibt mit der Eigenschaft

- Für die Eingabe $w \in L$ liefert P_L true
- Für die Eingabe $w \notin L$ false

In Pseudocode:

```
1      boolean P_L(w)
2      {
3       if (w in L) return true;
4       else return false;
5      }
```

2.2 Das Halteproblem

Das Halteproblem ist formal die Sprache

$$H = \{(P, w) | \text{ Das Programm P hält für die Eingabe } w\}$$

DIe Frage, ob ein Programm P, gegeben als Text, für eine Eingabe w hält, ist damit gleichberechtigt zu der Frage, ob $(P, w) \in H$ gilt.
Um zu zeigen, dass H unentscheidbar ist, zeigen wir zunächst

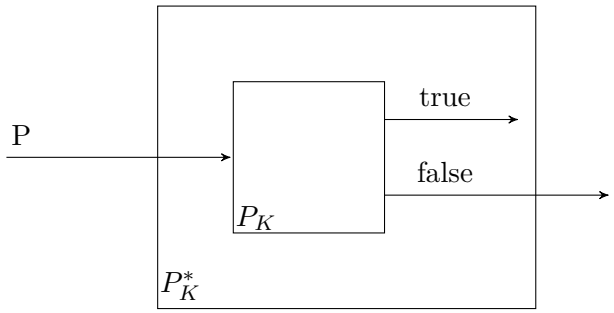
Satz
Das spezielle Halteproblem

$$K = \{P | \text{ Das Programm P hält für die Eingabe P}\}$$

ist unentscheidbar.

Beweis (durch Widerspruch)
Angenommen, K ist entscheidbar durch ein Programm P_K^*, das P_K als Unterprogramm benutzt und das

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



Nach Konstruktion gilt damit:

- Für die Eingabe P hält P_K^* genau dann, wenn P_K false liefert. Da P_K nach Annahme die Sprache K entscheidet, bedeutet dies:
- Für die Eingabe P hält P_K^* genau dann, wenn P nicht hält. Da dies für beliebige P gilt, können wir $P = P_K^*$ wählen. Damit folgt:
- Für die Eingabe P_K^* hält P_K^* genau dann, wenn P_K^* nicht hält und damit ein Widerspruch ist.

2.2.1 Weitere unentscheidbare Probleme

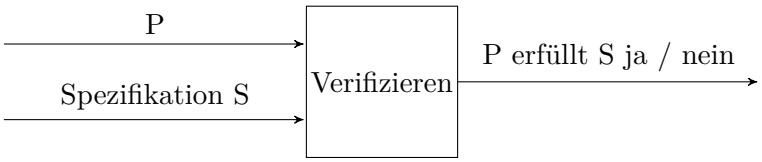
Mit der Unentscheidbarkeit des speziellen Halteproblems können wir die Unentscheidbar weitere Probleme zeigen. Um zu zeigen, dass eine Sprache B nicht entscheidbar ist, verwenden wir eine unentscheidbare Sprache A. Wir zeigen, dass mit der Annahme B sei entscheidbar, ein Entscheidungsverfahren für A konstruieren. Aus diesem Widerspruch folgt die Unentscheidbarkeit von B.

Satz
H ist nicht entscheidbar.

Beweis (durch Widerspruch)
Angenommen H sei entscheidbar. Dann können wir folgendes Programm konstruieren, dass das angenommene Entscheidungsverfahren P_H für H als Unterprogramm verwendet.

```
1       boolean P_K (Programm P)
2       {
3           return P_H (P, P)
4       }
```

Damit wäre aber P_K ein Entscheidungsverfahren für K, Widerspruch.



Damit folgt, dass auch die Programmverifikation unentscheidbar ist.

Satz Verify $= \{(P, S)|$ Das Programm P erfüllt die Spezifikation $S\}$ ist nicht entscheidbar.

Beweis
Angenommen Verify ist entscheidbar durch ein Programm P_{Verify}. Dann können wir das Programm konstruieren.

```
1       boolean P_H(Programm P, input w)
2       {
3         return  P_Verify (P, P haelt fuer die Eingabe w)
4       }
```

das dann ein Entscheidungsverfahren für H ist, Widerspruch!

Satz
$H_\epsilon = \{P|P$ hält für die Eingabe $\epsilon\}$ ist nicht entscheidbar

Beweis
Sei wieder angenommen, H_ϵ sei entscheidbar durch ein Programm P_{H_ϵ}. Damit können wir ein Programm P_H konstruieren, das zunächst ein Programm F konstruiert, das P mit der Eingabe w aufruft.

```
1       boolean P_H (Programm P, input w)
2       {
3         void F()
4         {
5          P(w)
6         }
7
8         return P_Hε(F())
9       }
```

Dann ist P_H aber ein Entscheidungsverfahren für das Halteproblem, denn:

$$(P, w) \in H \Leftrightarrow P \text{ hält für } w \Leftrightarrow F \text{ hält für } \epsilon \Leftrightarrow F \in H_\epsilon$$

Und damit Widerspruch.

2.2.2 Weitere unentscheidbare Probleme

Fast alle Probleme in Zusammenhang mit Programmverifikation, z.B.

$$\{P|P \text{ verursacht eine Division durch 0 für irgendeine Eingabe}\}$$

$$\{P|P \text{ verursacht einen Bufferoverflow für irgendeine Eingabe}\}$$

$$\{(P_1, P_2)|P_1 \text{ verhält sich wie } P_2\}$$

Mathematische Probleme

$$\{F|\text{ Die durch die Formel F ausgedrückte Funktion ist } x \to 0\}$$

$$\{S|S \text{ ist eine wahre mathematische Aussage}\}$$

(Es gibt keinen Algorithmus der prüft, ob eine Formel wahr ist)

2.3 Komplexitätstheorie

Wir beschränken uns nun auf entscheidbare Probleme und betrachten den Aufwand zur Lösung dieser Probleme.

2.3.1 Die Klasse P und NP

Für die O-Notation gelten folgende Rechenregeln

- $O(f+g) = O(f) + O(g)$
- $O(f+g) = O(max(f,g))$
- $O(c*f) = O(f)$ für eine Konstante $c > 0$
- $O(f*g) = O(f) * O(g)$

Bei der Laufzeitmessung von Algorithmen verwenden wir das uniforme Kostenmaß, bei dem die Laufzeit aller einzel Operationen (wie Zuweisung, Vergleich, Addition) in $O(1)$ liegt.

Definition
Die Komplexitätsklasse P ist definiert durch

$$P = \bigcup_{k \geq 1} \{L | L \text{ ist entscheidbar durch ein Programm mit Laufzeit in } O(n^k)\}$$

wobei n die Länge der Eingabe ist.

Beispiele
Folgende Sprachen liegen in P:

- Die Sprache aller Palindrome: Das Programm

```
1       boolean P (String w)
2       {
3               return w = reverse(w)
4       }
```

  stellt für alle $w \in \Sigma^*$ in der Zeit $O(|w|)$ fest, ob w ein Palindrom ist. Dabei seien reverse sowie -"Funktionen mit linearer Laufzeit.

- Die Sprache $\{a^n b^n | n \geq 0\}$, da diese durch einen Recursive Descent Parser in der Zeit $O(n)$ entschieden werden kann.

- Jede kontextfreie Sprache. Es gibt einen Algorithmus, der jede kontextfreie Sprache in Zahl $O(n^3)$ entscheidet.
- PFAD $= \{(G, n_1, n_2) | G$ ist ein Graph, in dem es einen Pfad von n_1 nach n_2 gibt$\}$ wobei G durch eine Adjanzenzliste gegeben sei.

  Da die Adjazensliste eines Graphen $G = (V, E)$ 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 mit Ziel n_2. Deren Laufzeit liegt in $O(|V| + |E|)$. Aus $|V| + |E| \leq n$ folgt

  $$O(|V| + |E|) \subseteq O(n)$$

  Damit liegt die Laufzeit des Entscheidungsverfahrens in $O(n)$, woraus PFAD $\in P$ folgt.

Ferner gibt es Sprachen, die nicht offensichtlich in P liegen.
Wichtiges Beispiel:

$$SAT = \{F | F \text{ ist eine erfüllbare Formel der Aussagenlogik}\}$$

z.B. gilt $(x \vee y) \wedge z \in SAT \qquad \overline{x} \wedge x \neq SAT$
Um für eine Formel F zu prüfen, ob $F \in SAT$ gilt, können wir alle 2^n Belegungen erzeugen und jeweils den Wahrheitswert berechnen (n sei die Anzahl der Variablen F)

Die Laufzeit dieses Entscheidungsverfahrens liegt in $O(2^n * f(n))$, wobei $f(n)$ die Zeit ist, um den Wahrheitswert zu berechnen.
(Anzahl Variablen $\leq$ Länge der Eingabe)

Wir können jedoch in polynomieller Zeit verifizieren, dass $F \in SAT$ gilt, wenn eine erfüllende Belegung C_F für F bekannt ist. Denn dazu muss das Entscheidungsverfahren lediglich den Wahrheitswert von F unter der Belegung C_F berechnen, was in der Zeit $O(|F|)$ möglich ist.

Definition
Die Komplexitätsklasse NP ist definiert durch

$$NP = \bigcup_{k \geq 1} \{L | L \text{ ist verifizierbar in Zeit } O(n^k)\}$$

wobei n die Länge der Eingabe ist.

Beispiel
Es gilt $SAT \in$ NP, da mit einer erfüllenden Belegung für F in Zeit $O(|F|)$ verifiziert werden kann, ob $F \in SAT$ gilt.

Es gilt: $P \subseteq$ NP, denn für jede Sprache $L \in P$ gibt es ein Entscheidungsverfahren mit einer Laufzeit in $O(n^k)$. Dies ist ebenfalls ein Verifizierungsverfahren, das kein Zertifikat verwendet.

Unbekannt ist, ob $P =$ NP gilt.
Die Klasse P wird als betrachtet als Klasse der effizient lösbaren Probleme. Gründe dazu

- Die meisten Probleme in P lassen sich in der Zeit $O(n^k)$ mit einem kleinen k lösen. Diese Probleme sind damit auch praktisch lösbar

NP-vollständige Probleme

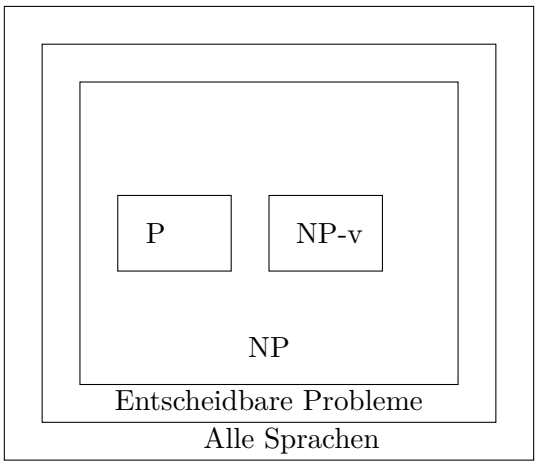
Die NP-vollständigen Probleme sind eine Klasse von Problemen in NP, für die keine effizienten Entscheidungsverfahren bekannt sind. Alle bekannten Verfahren besitzen eine exponentielle Laufzeit. Die NP-vollständigen Probleme sind mindestens so schwierig wie jedes andere Problem in NP.

Es gilt: Wenn ein effizientes Entscheidungsverfahren für ein NP-vollständiges Problem gefunden wurde wird, dann ist für jedes Problem in NP ein effizientes Entscheidungsverfahren bekannt.
Genauer:

Satz
Es gilt P=NP genau dann, wenn es ein NP-vollständiges Problem L mit $L \in P$ gibt.

Man vermutet, dass P $\neq$ NP gilt, weil bisher kein vollständiges Problem P gefunden wurde.

Situation für P $\neq$ NP



Satz
Das Erfüllbarkeitsprinzip der Aussagenlogik

$$SAT = \{F | F \text{ ist eine erfüllbare Formel der Aussagenlogik}\}$$

ist NP-vollständig.

Bemerkung
Das schnellste Verfahren für SAT hat die Laufzeit $O(1,308^n * p(n))$, wobei p ein Polynom ist.

Anwendungen

- **Autokonfiguration**
  Die bestellbaren Konfigurationen kann man durch Regeln in Form von aussagenlogischen Formel beschreiben, z.B.

  $$\text{Sportfahrwerk} \rightarrow \text{Leichtmetallfelgen} \wedge (\text{Motor 2.0} \vee \text{Motor 2.5} \vee \text{Motor 3.0})$$

  Alle derartigen Formel werden mit $\wedge$ verknüpft zu einer Formel F_1. Die Wünsche des Kunden lassen sich als Formel F_2 beschreiben. Die Kundenwünsche sind erfüllbar, wenn $F_1 \wedge F_2$ erfüllbar sind.

- **Vereinfachen von Schaltkreisen**
  Angenommen, es ist ein Referenzwurf für einen Schaltkreis vorhanden. Dieser Entwurf soll nun vereinfacht werden (weniger Gatter), um die Herstellungskosten zu senken. Um zu prüfen, ob dieser Schaltkreis gleichwertig zum Referenzentwurf ist, wird der Referenzentwurf durch eine Formel F_R der Aussagenlogik

dargestellt, der vereinfachte Schaltkreis durch die Formel F_S. Dann muss $F_R \leftrightarrow F_S$ eine Tautologie sein. Diese ist gleichwertig damit, dass

$$\overline{(F_R \leftrightarrow F_S)} \text{ unerfüllbar ist}$$

Das heißt, beide Schaltkreise sind gleichartig, genau dann, wenn

$$\overline{(F_R \leftrightarrow F_S)} \notin SAT$$

Satz
Das Problem HAMILTON-Kreis

$$\{G|G \text{ besitzt einen Hamilton Kreis}\}$$

ist NP-vollständig.

Verallgemeinerung davon
Traveling Salesman Problem (TSP).

Gegeben n Städte und Verbindungen zwischen den Städten, gesucht ist eine kürzeste Rundreise durch die Städte.

Satz
Das Problem TSP $= \{(M, k)|M$ ist eine Entfernungsmatrix und es gibt eine Rundreise über die Länge $\leq k\}$ ist NP-vollständig.

Bemerkung
Das Problem eine kürzeste Rundreise zu berechnen, ist mindestens so schwer, wie obiges Entscheidungsproblem.

Anwendungen

- **Platine bohren**
  Um Bohrzeit für Platinen zu minimieren, muss ein TSP für die Bohrlöcher gelöst werden
- **Optimieren einer Fertigungsstraße**
  Auf einer Fertigungsstraße sollen Produkte $P_1, \ldots, P_n$ hergestellt werden. Dazu muss die Fertigungsstraße jeweils umgerüstet werden. Sei d_{ij} der Zeitaufwand, um eine Fertigungsstraße die Produkt i herstellt, für Produkt j umzurüsten. Um eine Reihenfolge festzulegen, die die Summe der Rüstzeiten minimiert, muss ein TSP für die Matrix (d_{ij}) gelöst werden.

Weiteres, ähnliches Problem: Kürzester Hamiltonpfad
Dieses Problem ist ebenfalls NP-vollständig.

Anwendung

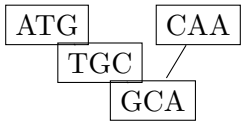
- DNA-Sequenzierung
  Eine DNA Sequenz ist ein Wort über dem Alphabet {A, C, G, T}. Um ein DNA zu sequenzieren, wird diese in kleine Bruchstücke geschnitten, diese sequenziert und aus diesen Bruchstücken die ursprüngliche Sequenz rekonstruiert.

  Ansatz dazu
  Kürzeste, gemeinsame Obersequenz (Shortest Common Supersequenz) bestimmen, d.h. eine kürzeste Sequenz finden, die alle Bruchstücke als Teilwort enthält. Dazu wird anhand der Überlappung zweier Bruchstücke ein Abstand berechnet und damit ein Shortest Common Superstring- bzw. kürzester Hamiltonpfad Problem gelöst.

Beispiel
Ursprüngliche Sequenz

$$ATGCAA$$

Bruchstücke: ATG, CAA, GCA, TGC



Damit ist eine kürzeste gemeinsame Oberfrequenz $ATGCAA$ gefunden.

Algorithmen für TSP

- Nearest Neighbor Greedy Algorithmus

```
1          Starte in einem beliebigen Knoten
2
3          Solange nicht alle Knoten besucht,
4            wähle einen nächsten Nachbarn des aktuellen Knoten aus
5            und verlängere den aktuellen Pfad
```

  Greedy-Algorithmen wählen in jedem Schritt eine lokal optimale Teillösung aus. Greedy-Algorithmen können optimal sein. Für das TSP ist der Greedy

Algorithmus nicht optimal, er kann beliebig schlechte Lösungen liefern.

- Es gibt einen Algorithmus, der optimale Lösung liefert, mit Laufzeit in $O(n^2 * 2^n)$, wobei $n = |V|$
- Approximation: Wir versuchen einen Rundweg zu finden dessen Länge etwa so groß ist, wie die kürzeste Länge.
  Das metrische TSP ist ein TSP, bei dem der Abstand d_{ij} zweier Knoten eine Metrik ist.

  Metrik
  - $d_{ij} > 0$
  - Muss Symmetrisch sein
  - Muss die Dreiecksungleichung beinhalten

  Für das metrische TSP gibt es einen Approximationsalgorithmus mit der Laufzeit in $O(n^2 \log n)$, der eine Lösung liefert, die eine Länge $\leq 2*$ *optimale Länge* besitzt.

Rucksackproblem

Ein Rucksack der Größe $S > 0$ soll mit einer Auswahl von Gegenständen $1, \ldots, n$ der Größe $s_1, \ldots, s_n > 0$ maximal bepackt werden. Gesucht ist also eine Menge $C \subseteq \{1, \ldots, n\}$ mit

$$\sum_{k \in C} s_k \leq S \quad \text{und} \quad \sum_{k \in C} s_k \quad \textit{maximal}$$

Anwendungen

- CD mit mp3 maximal befüllen
- Budget maximal verbrauchen

Das Rucksack-Problem ist NP-vollständig.

Algorithmen für Rucksack
Sei $r(k, s)$ die maximale Füllmenge eines Rucksacks de Größe s $(0 \leq s \leq S)$, der mit einer Auswahl von Gegenständen $1, \ldots, k$ bepackt ist. Gesucht ist $r(n, S)$. Wir berechnen $r(k, s)$ für $1 \leq k_s \leq n, 0 \leq s \leq S$ mit Hilfe einer Tabelle.

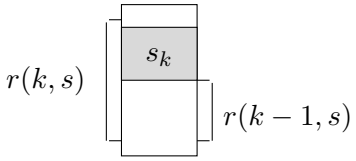
Feststellung
Wenn für alle $0 \leq l \leq s \quad r(k-1,l)$ bereits bekannt ist, lässt sich daraus $r(k,s)$ wie folgt berechnen:

- Wenn der Gegenstand k nicht eingepackt wird, gilt

$$r(k,s) = r(k-1,s)$$

- Wenn der Gegenstand k eingepackt wird, gilt

$$r(k,s) = r(k-1,s-s_k) + s_k$$



Daraus ergibt sich

$$r(k,s) = \begin{cases} 0 & \text{für } k = 0 \\ r(k-1,s) & \text{für } s \leq s_k \\ max\left(r(k-1,s),s\right), r(k-1,s-s_k) + s_k & \text{sonst} \end{cases}$$

$r(k,s)$ lässt sich mit einer Tabelle berechnen

Laufzeit

$$O(n*S)* \underbrace{O(1)}_{\text{pro Schritt}} = O(n*S)$$

Dies ist der Widerspruch dazu, dass Rucksack NP-vollständig ist, denn die Länge der Eingabe ist

$$\log_s S + \sum_{k=1}^{n} \log_2 s_k$$