Theoretische Informatik

Markus Klemm.net

SS 2014

Inhaltsverzeichnis

1 Automaten und formale Sprachen

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

Beispiel:

- Alphabet $\Sigma = \{a, b, c\},\ \Sigma^* = \{\epsilon, a, b, c, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}$

- Alphabet $\Sigma = \{\text{if}, \text{then}, \text{else}, x, =, 0, 1, \ldots, 9\}$ Wörter: if $x = 0$ else $x = 5$, if else than x

Definition Eine formale Sprache, über ein Alphabet Σ ist eine Teilmenge von Σ^*.
Beispiel:

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

Konkatenation von Wörtern

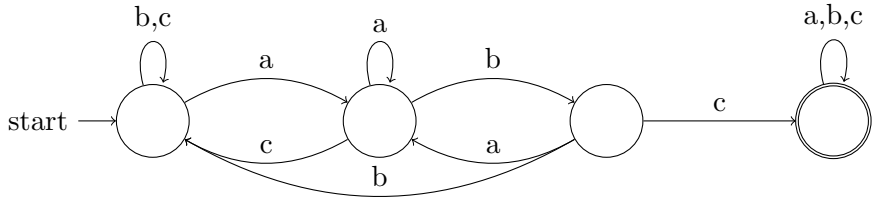
Definition Für Wörter v, w ist vw die Konkatenation. Das Wort w^n ist die n–fache Konkatenation von w. Dabei ist $w^0 = \epsilon$.
Beispiel: $(abc)^3 = abc\,abc\,abc$
Bemerkung: Es gilt $\epsilon \in \Sigma^*$, da Σ^* die Menge aller Wörter ist.

1.1 Reguläre Sprachen

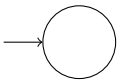
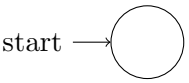
Motivation: Suche nach einem Wort s in einem Text t.
Naiver Algorithmus: Wort s Zeichen für Zeichen mit t vergleichen, bei Mismatch eine Stelle weiterschieben.
Laufzeit: $\mathcal{O}(|s| \cdot |t|)$ (schlechte Laufzeit)

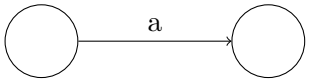
1.1.1 Deterministische Endliche Automaten (Deterministic Finite Automate, DFA)

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).



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

- Zustände: 
- Startzustände  [Hier im Script:] 

- Endzustände 
- Übergänge 

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

Defnition Ein DFA ist ein Tupel $M = (Z, \Sigma, \delta, z_0, E)$

- Z: Menge der Zustände
- Σ: Eingabealphabet
- $\delta : Z \times \Sigma \to Z$: Überführungsfunktion
- $z_0 \in Z$: Startzustand
- $E \subseteq Z$: Endzustände

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 $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA.

- Die erweiterte Überführungsfunktion $\hat{\delta} : Z \times \Sigma^* \to Z$ von M ist definiert durch:

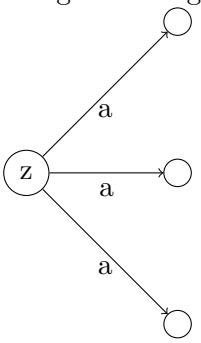
$$\hat{\delta}(z, w) = \begin{cases} z & \text{für} \quad w = \epsilon \\ \hat{\delta}(\delta(z, a), x) & \text{für} \quad w = ax \text{ mit } a \in \Sigma, x \in \Sigma^* \end{cases}$$

- Die von M akzeptierte Sprache ist:

$$L(M) = \{w \in \Sigma^* | \hat{\delta}(z_0, w) \in E\}$$

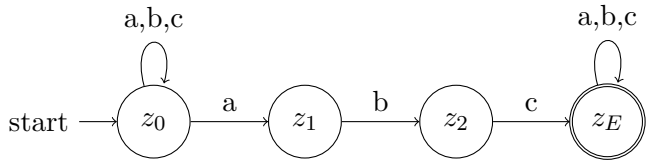
1.1.2 Nichtdeterministische endliche Automaten (NFA)

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: NFA, der alle Wörter akzeptiert, die abc enthalten:



So wie eine Straßenkarte mögliche Wege beschreibt, beschreibt auch ein NFA mögliche Zustandsfolgen bei der Verarbeitung eines Wortes. Inbesondere „weiß“ der NFA nicht, welchen Folgezustand er auswählen muss. Ein NFA lässt sich daher nicht unmittelbar als Programm implementieren.

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

Beispiel (Fortsetzung) Für die Eingabe $w = cbaab\underline{abc}c$ 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, d.h. $w \in L(M)$.
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 $abba \notin L(M)$.

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

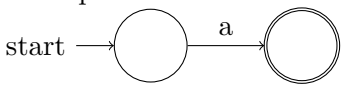
Da es eine Folge von Berechnungen gibt, die zu einem Endzustand führt, wird die Eingabe akzeptiert.

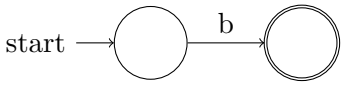
Für einen NFA lässt sich die Überführungsfunktion definieren durch

$$\delta : Z \times \Sigma \to \mathcal{P}(Z)$$

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

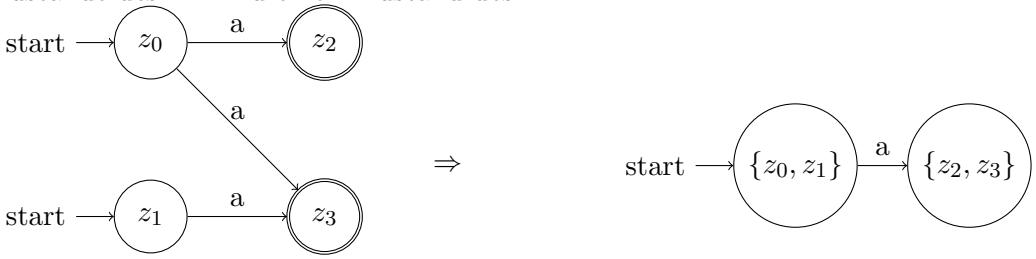
Beispiel





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

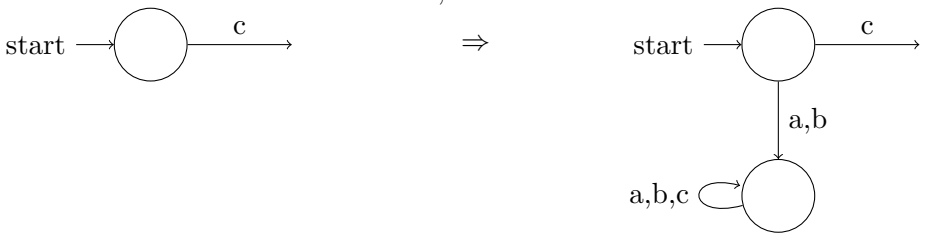
Umwandlung eines NFA in einen DFA Es gilt: Für jeden NFA gibt es einen DFA, der die gleiche Sprache erkennt. Idee zu Umwandlung: Wir vereinigen mögliche Folgezustände des NFA zu einem Zustand des DFA.



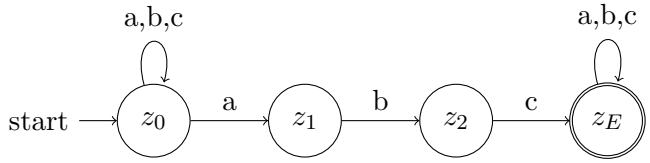
NFA

DFA

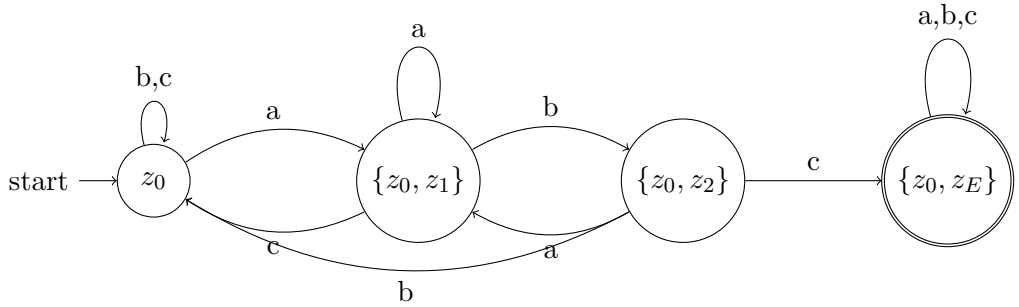
Falls es für einen Zustand z und ein Zeichen a keinen Folgezustand gibt (d.h. $\delta(z, a) = \varnothing$), führen wir einen Fehlerzustand ein, der nicht mehr verlassen werden kann.



Beispiel: Umwandlung des NFA



in einen DFA.



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 $\mathcal{P}(Z)$ 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 der Zustände ist (Minimalautomat).
Je zwei Minimalautomaten unterscheiden sich höchstens in der Benennung der Zustände.

1.2 Reguläre Ausdrücke

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

1. $\varnothing$ ist ein regulärer Ausdruck $L(\varnothing) = \varnothing$.
2. Für jedes $a \in \Sigma \cup \{\varepsilon\}$ ist a ein regulärer Ausdruck und $L(a) = \{a\}$.
3. Für reguläre Ausdrücke E_1, E_2 sind $(E_1|E_2)$, (E_1E_2), (E_1^*) reguläre Ausdrücke 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 $L(E_1)L(E_2) = \{w_1w_2 | w_1 \in L(E_1), w_2 \in L(E_2)\}$)
   $L(E_1^*) = (L(E_1))^*$

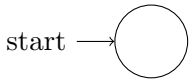
Um Klammern zu sparen, legen wir folgende Regeln für die Priorität der Operatoren fest: Die höchste Priorität besitzt der Operator $*$, gefolgt von Konkatenation, gefolgt vom Operator $|$.

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

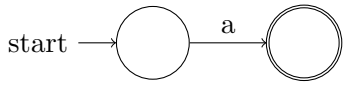
Satz Für jeden regulären Ausdruck E gibt es einen NFA M mit $L(E) = L(M)$

Beweis: Wir induzieren über den Aufbau regulärer Ausdrücke:
Induktionsanfang:

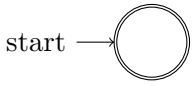
- Für $E = \varnothing$ ist M folgender NFA:



- Für $E = a$ ist M der NFA:

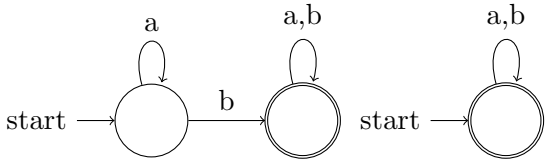


- Für $E = \varepsilon$ ist M der NFA:

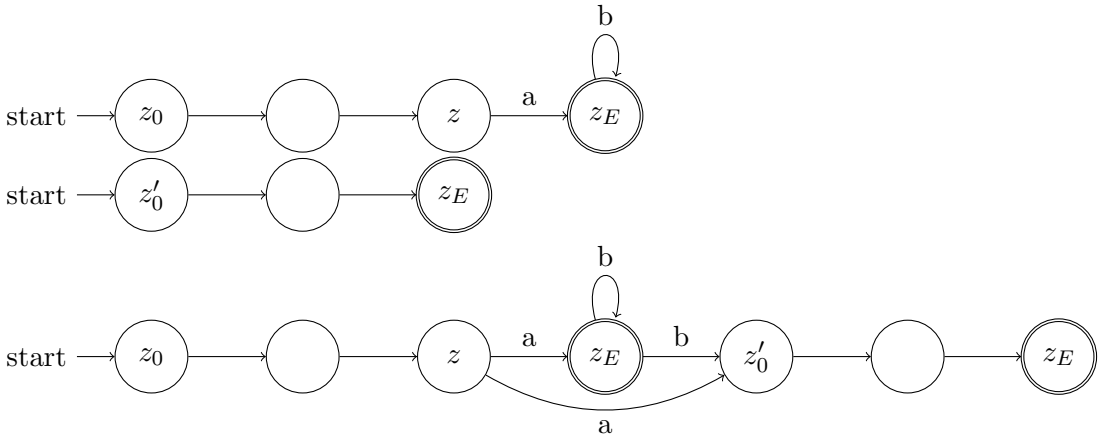


Induktionsschritt:
Seien E_1, E_2 reguläre Ausdrücke und nach Induktionsvorraussetzung M_1', M_2' NFAs mit $L(E_1) = L(M_1'), L(E_2) = L(M_2')$. Ferner seien M_1, M_2 DFAs mit $L(M_1) = L(M_1'), L(M_2) = L(M_2')$

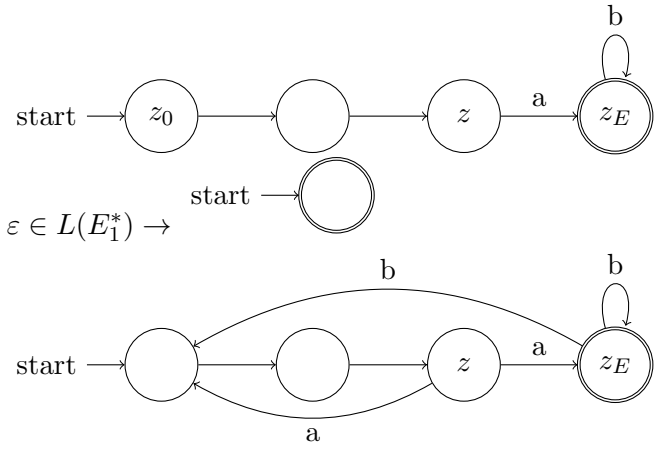
- $E1|E_2$: Der NFA für $E_1|E_2$ ist die „Vereinigung“ von M_1, M_2, da ein NFA mehrere Startzustände haben darf.



- E_1E_2: Skizze zur Idee:



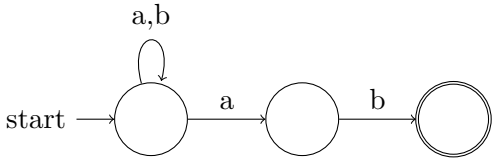
- E_1^*



- $E^+ = EE^*$
- $[E? = \varepsilon|E]$

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 $(a|b)^*ab$.

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

Folgerung aus den bisher aufgeschriebenen Sätzen:

- Die NFAs erkennen genau die regulären Sprachen.
- Die regulären Ausdrücke erzeugen genau die regulären Sprachen.

Menge der regulären Sprachen = Sprachen, die von DFAs, NFAs oder von regulären Ausdrücken erkannt werden $\subset$ Menge aller Sprachen

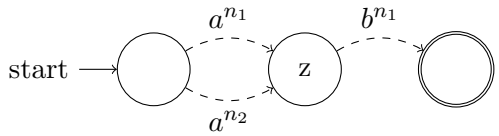
Lexer Ein Lexer ist ein Werkzeug, das aus einem regulären Ausdruck einen DFA erzeugt, der die gleiche Sprache erkennt. Damit können Akzeptoren für Wörter aller Muster konstruiert werden.
Arbeitsweise eines Lexers: regulärer Ausdruck $\rightarrow$ NFA $\rightarrow$ DFA

1.3 Nichtreguläre Sprachen

Satz Die Sprache $L = \{a^n b^n | n \geq 0\}$ ist nicht regulär.

Beweis:

Angenommen, L sei regulär. Dann gibt es einen DFA M mit $L(M) = L$. Nach dem Lesen von a^n befindet sich 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 einen Endzustand. Danach akzeptiert M jedoch auch das Wort $a^{n_2}b^1$, Widerspruch, da $n_1 \neq n_2$.

1.4 Kontextfreie Sprachen

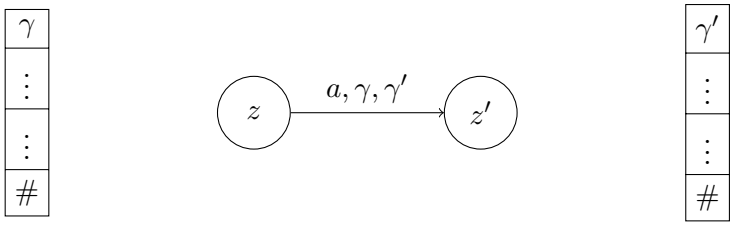
1.4.1 Kellerautomaten (PDAs)

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

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

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

Grapische Darstellung von PDAs: Wir erweitern die Darstellung von NFAs um Stackoperationen.



Stack vorher Stack nachher

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

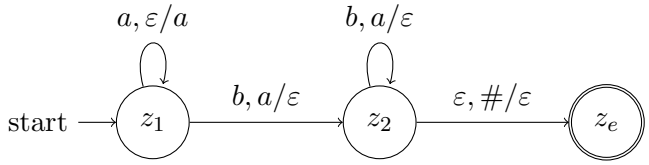
- Der PDA liest das Zeichen a der Eingabe, entfernt das oberste Stackzeichen γ und schreibt γ' auf den Stack.

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

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

Damit ein PDA einen leeren Stack erkennen kann, wird das Ende des Stacks mit dem unterstem Stackzeichen # markiert. 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 gelesene b ein a vom Stack entfernt. Danach muss der Stack leer sein.



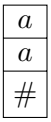
Verhalten für die Eingabe *aaabbb*:

#

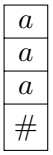
1. *aaabbb*

a
#

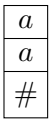
2. aabbb



3. abbb



4. bbb



5. bb

6. b

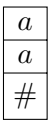
7.

8.

Verhalten für *aab*

1. aab

2. ab



3. b

4. ε

Sackgasse in z_2

Verhalten für *abb*

#

1. abb

a
#

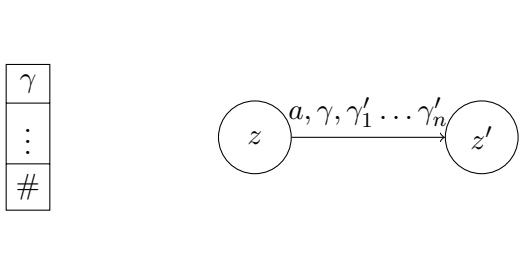
2. bb

#

3. b

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.



Stack vorher Stack nachher

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

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

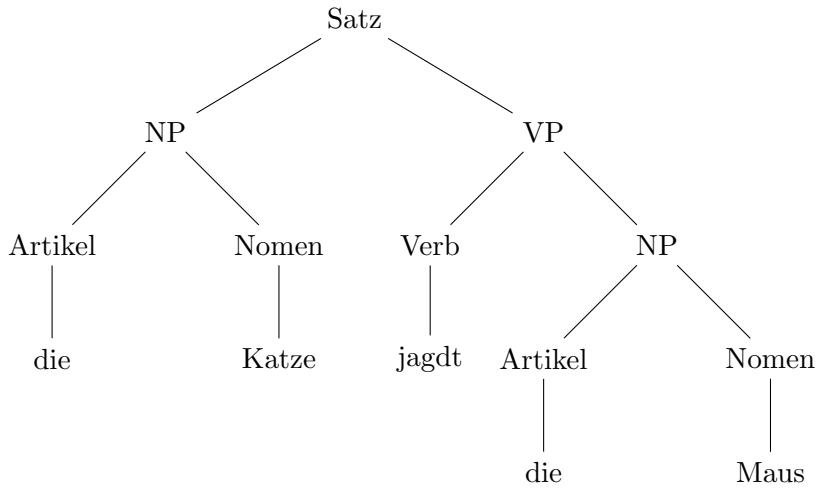
1.5 Kontextfreie Grammatiken

Jede Grammatik besitzt ein Startsymbol und Ersetzungsregeln der Form linke Seite → 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, Nominalphrase, Verbalphrase, Artikel, Nomen, Verb
sowie den Terminalzeichen:
die, Katze, Maus, jagdt
besteht. Das Startsymbol ist Satz, die Regeln sind
Satz → Nominalphrase Verbalphrase
Nominalphrase → Artikel Nomen
Verb → jagt
Artikel → die
Nomen → Katze
Nomen → Maus
Verbalphrase → Verb
Verbalphrase → Verb Nominalphrase

Mögliche Ableitung:
Satz ⇒ Nominalphrase Verbalphrase ⇒ Artikel Nomen Verbalphrase ⇒ die Nomen Verbalphrase ⇒ die Katze Verbalphrase ⇒ die Katze Verb Nominalphrase ⇒ die Katze jagt Nominalphrase ⇒ die Katze jagt Artikel Nomen ⇒ die Katze jagt die Nomen ⇒ die Katze jagt die Maus

Syntaxbaum dazu:



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

- V ist die Menge der Variablen oder Nonterminalzeichen
- Σ ist das Alphabet mit $V \cap \Sigma = \varnothing$. Die Elemente aus Σ heißen auch Terminalzeichen
- P ist die Menge der Regeln (oder Produktionen) der Form $u \to v$, wobei $u \in V, v \in (V \cup \Sigma)^*$.
- $S \in V$ ist das Startsymbol

Beispiel (Forts) Die Grammatik lässt sich als Tupel $G = (V, \Sigma, P, S)$ darstellen, mit
$V = \{\text{Satz}, \text{Nominalphrase}, \text{Verbalphrase}, \text{Verb}, \text{Artikel}, \text{Nomen}\}$,
$\Sigma = \{$die,Katze,Maus,jagt$\}$,
$S =$ Satz und
P wie oben.

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

Beispiel Es gilt Satz $\Rightarrow$ Nominalphrase Verbalphrase $\Rightarrow$ Artikel Nomen Verbalphrase

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

Beispiel Es gilt

- Satz $\Rightarrow^*$ Satz
- Satz $\Rightarrow^*$ Artikel, Nomen, Verbalphrase
- Satz $\Rightarrow^*$ die Katze jagt die Maus

Definition Die von einer kontextfreien Grammatik G erzeugte Sprache ist

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

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

- $S \to \varepsilon$
- $S \to SS$
- $S \to [S]$

können wir damit kürzer darstellen durch
$S \to \varepsilon | SS | [S]$

Beispiel Durch die Grammatik mit den Regeln

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

und dem Startsymbol S_N können wir Zahlen aus $\mathbb{N}_0$ darstellen. Die Zahl 120 lässt sich ableiten durch $S_N \Rightarrow 1S_N \Rightarrow 12S_N \Rightarrow 120$. Diese Regeln verwenden wir in einer weiteren Grammatik, um arithmetische Ausdrücke zu erzeugen.

- 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.

$$S_E \to S_N | (S_E) | S_E \text{ Op } S_E$$

und dem Startsymbol S_E.
Der Ausdruck $2 * (3 + 4)$ lässt sich ableiten durch
$S_E \Rightarrow S_E * S_E \Rightarrow S_N * S_E \Rightarrow 2 * S_E \Rightarrow 2 * (S_E) \Rightarrow 2 * (S_E + S_E) \Rightarrow 2 * (S_N + S_E) \Rightarrow 2 * (S_N + S_n) \Rightarrow 2 * (3 + S_N) \Rightarrow 2 * (3 + 4)$

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

1.5.1 PDAs und kontextfreie Grammatiken

Satz Kellerautomaten (PDAs) akzeptieren genau die kontextfreien Sprachen.
Beweis:

1. Für jeden PDA M gibt es eine kontextfreie Grammatik G mit $L(M) = L(G)$. Ohne Beweis.

2. Für jede kontextfreie Grammatik G gibt es einen PDA M mit $L(M) = L(G)$

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

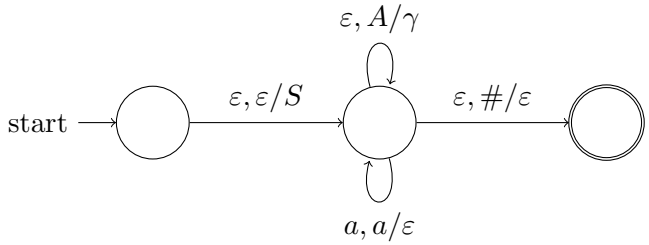
- Zuerst schreibt M das Startsymbol S auf den Stack und wechselt in einen weiteren Zustand.

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

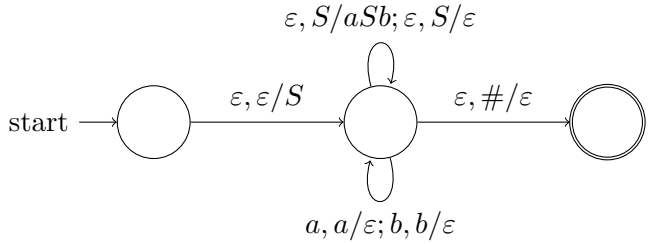
- eine Variable A. Wenn es eine Regel $A \to \gamma$ in G gibt, kann M das oberste Stackzeichen A durch γ ersetzen.

- ein Zeichen $a \in \Sigma$, 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 $L = \{a^n b^n | n \geq 0\}$ konstruieren wir einen PDA M mit $L(M) = L$. L wird erzeugt von der Grammatik mit den Regeln $S \to aSb|\varepsilon$.
Aus obigen Beweis erhalten wir den PDA.



Verhalten für *aabb*:

1. | S |
 | $\#$ | aabb

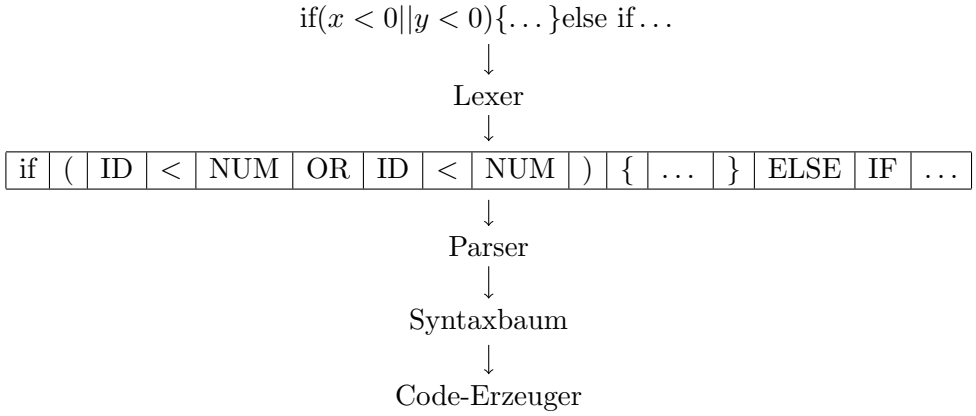
2. | a |
 | S |
 | b |
 | $\#$ | aabb

3. | S |
 | b |
 | $\#$ | abb

4. | a |
 | S |
 | b |
 | b |
 | $\#$ | abb

5. | S |
 | b |
 | b |
 | $\#$ | bb

Syntaxanalyse Arbeitsweise eines Compilers

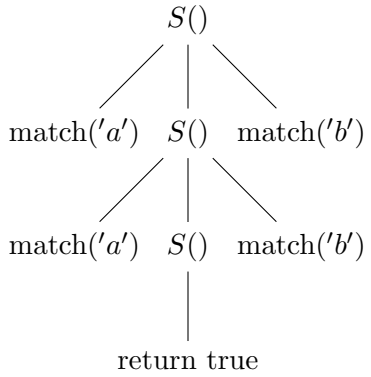


Parser lassen sich in zwei Klassen unterteilen:

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 zur Umwandlung einer kontextfreien Grammatik in einen PDA. Ein Top-Down-Parser lässt sich durch rekursive

Prozeduren implementieren, von denen jede einer 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 $\{a^n b^n | n \geq 0\}$, Eingabe $aabb$



Der Callgraph besitzt die gleiche Struktur wie der Syntaxbaum.

Ein Parser, der durch rekursive Funktionen einen Syntaxbaum von oben nach unten aufbaut, heißt Recursive Descent Parser.

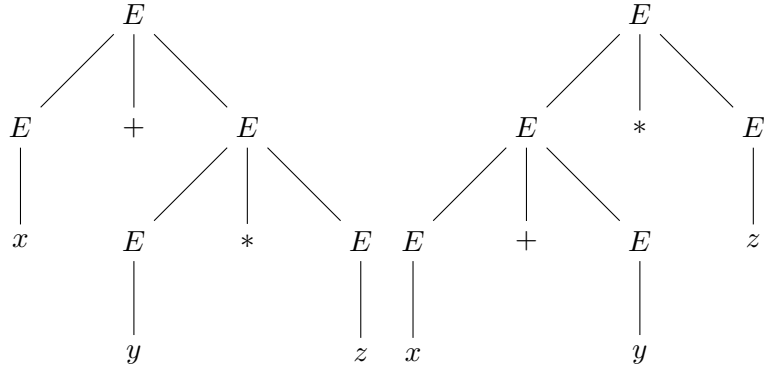
Mehrdeutigkeit

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

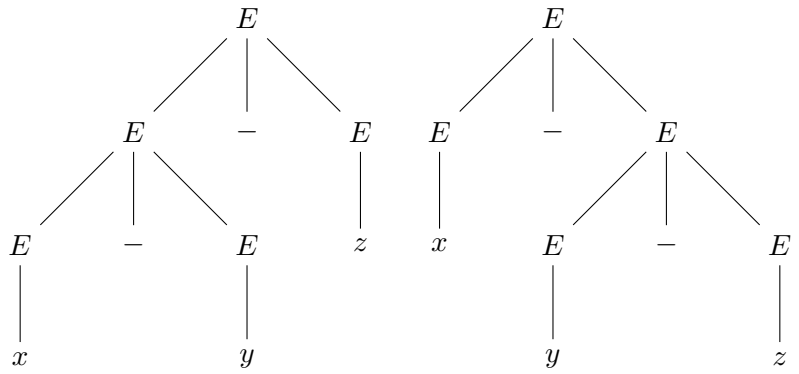
Beispiel: Die Grammatik für arithmetische Ausdrücke mit den Regeln

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

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



Auch wenn wir nur den Operator „-“ betrachten, ist die Grammatik mehrdeutig, denn der Ausdruck $x - y - z$ besitzt die Ableitungsbäume:



$(x - y) - z$ $x - (y - z) \Rightarrow x - y + z$

Mit obiger Grammatik gibt es mehrere Probleme

- Sie berücksichtigt nicht die Priorität der Operatoren (Punkt vor Strich)
- Sie berücksichtigt nicht die Assoziativität der Operatoren

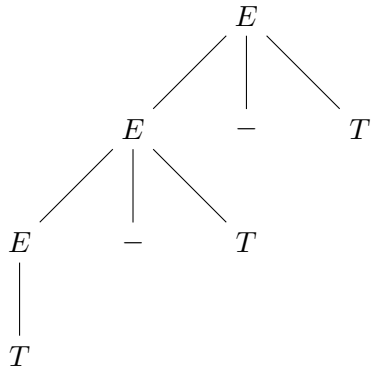
Um das Problem der Priorität zu lösen, definieren wir eine Grammatik, bei der sich Operatoren niedriger Priorität oben im Ableitungsbaum und Operatoren höherer Priorität weiter unten im Ableitungsbaum befinden müssen.
Wir führen dazu eine Variable T (Term) ein, aus der Produkte abgeleitet werden können.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

$$E \to E - T | E + T | T \tag{1}$$

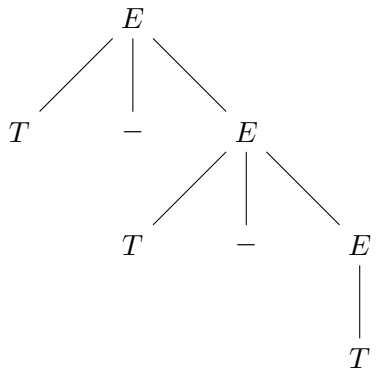
oder

$$E \to T - E | T + E | T \tag{2}$$

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



entspricht $(T - T) - T$, d.h. $-$ ist links-assoziativ.
Mit (2) ist folgende Ableitung möglich



entspricht $T - (T - T)$, d.h. $-$ wäre rechtsassoziativ.

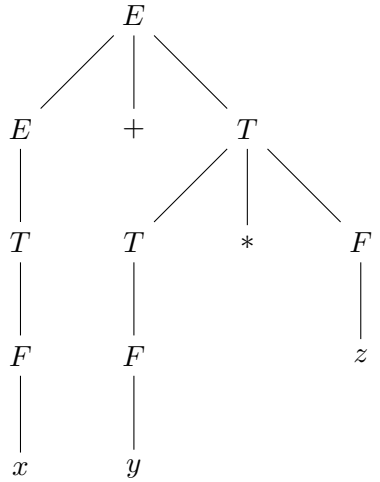
Nur Grammatik (1) berücksichtigt sowohl die Priorität als auch die Assoziativität der Operatoren $+, -$ in korrekter Weise. Entsprechend definieren wir Regeln für T und erhalten:

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

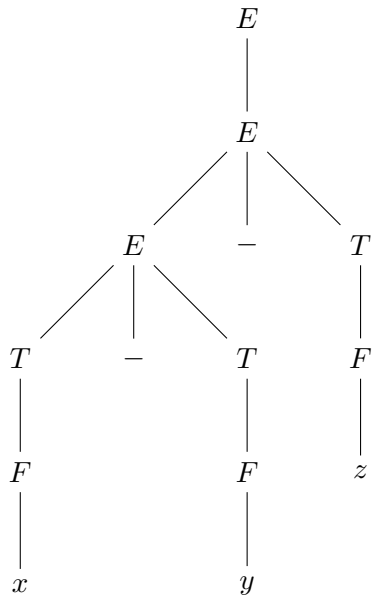
$$T \to T * F|T/F|F$$

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

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



Der Ableitungsbaum für $x - y - z$ ist



Bottom-up Parser Ein Buttom-up Parser baut einen Ableitungsbaum von unten nach oben auf. Diese lassen sich effizient realisieren durch LR-Parser. Ein LR-Parser liest die Eingabe von links nach rechts und 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 an der Spitze des Stacks entsprechen der rechten Seite γ einer Regel $A \to \gamma$ und werden durch A ersetzt.
- Accept: Die Eingabe wurde verarbeitet und der Stack enthält nur das Startsymbol.
- Error: Ein Syntaxfehler wird gemeldet.

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

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

Stack bottom top	Restliche Eingabe	Aktion
	$x + y * z$	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 des Ausdrucks berechnen, wenn Zwischenwerte in den Symbolen gespeichert werden.

Zur Konstruktion von LR-Parsern werden Tools wie Yacc,Bison, CUP verwendet.

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

- Es ist nicht erkennbar, welche Regel ausgewählt 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 die 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

$$E \to T\{+T\} \text{ (alternat. Notation: } E \to T(+T)^* \text{)}$$

Dabei bedeutet $\{x\}$: beliebig viele Vorkommen von x.

Die Grammatik für arithmetische Ausdrücke lässt sich in EBNF wie folgt darstellen:

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

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

$$F \to (E)|\text{Num}$$

$$\text{Num} \to Z\{Z\}$$

$$Z \to 0|\ldots|9$$

Aus der Darstellung in EBNF lassen sich Syntaxdiagramme ableiten:
E :
F :

Problem: Aus der Darstellung in EBNF oder den Syntaxdiagrammen ist nicht ersichtlich, welche Assoziativität die Operatoren haben. Ohne weitere Annahmen (z.B: alle Operatoren linksassoziativ) ist diese Grammatikbeschreibung nicht eindeutig.

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

$$E \to TE_{\text{Tail}}$$

$$E_{\text{Tail}} \to \varepsilon| + E| - E$$

$$T \to FT_{\text{Tail}}$$

$$T_{\text{Tail}} \to \varepsilon| * T|/T$$

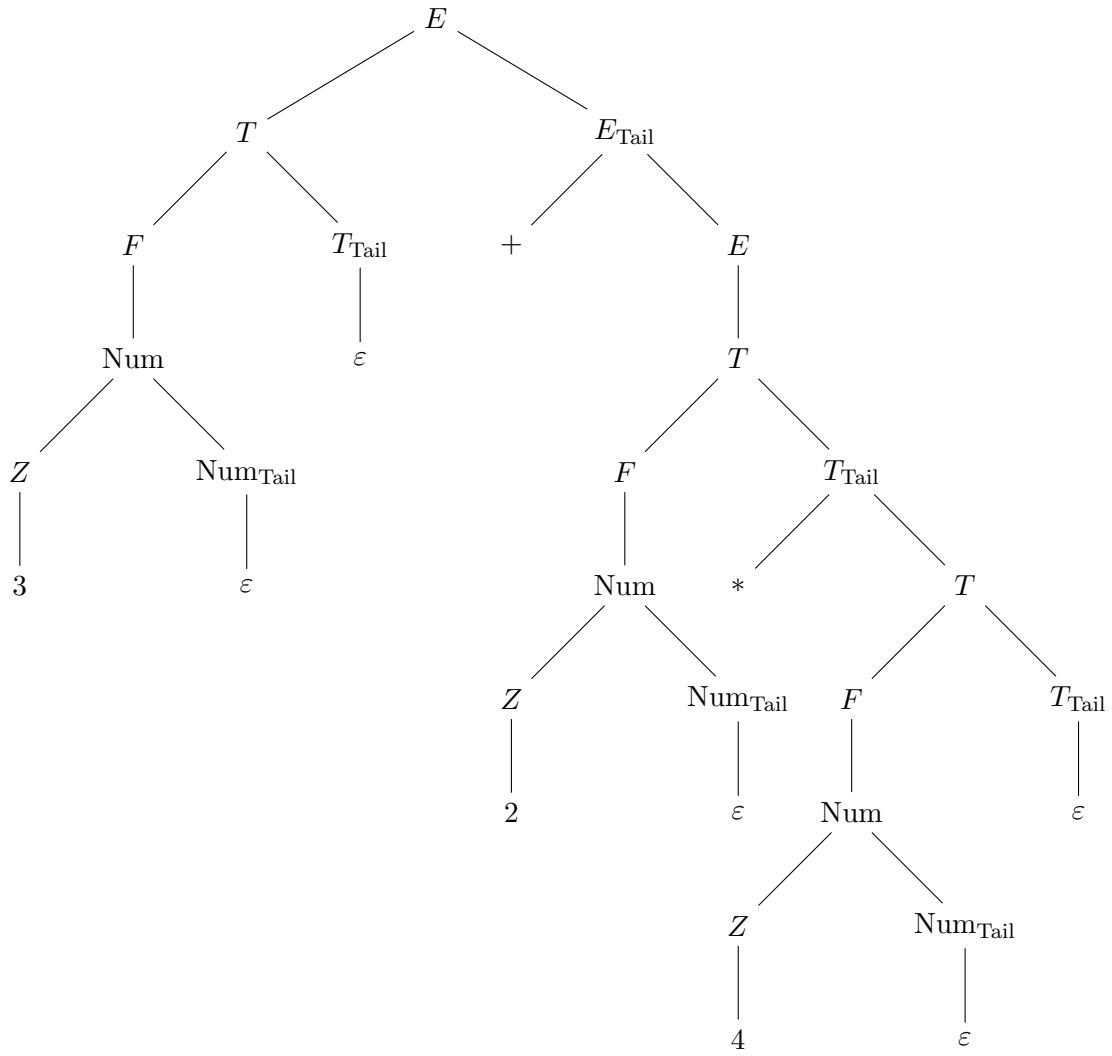
$$F \to (E)|\text{Num}$$

$$\text{Num} \to Z\text{Num}_{\text{Tail}}$$

$$\text{Num}_{\text{Tail}} \to \varepsilon|\text{Num}$$

$$Z \to 0|\ldots|9$$

Beispiel: Ableitung des Ausdrucks $3 + 2 * 4$



1.5.2 0L-Systeme

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

- forward(l)
- left(α) bzw. right(α). Dreht die Turtle

Ein 0L-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
Variablen: F

Terminalzeichen: $+, -$
Regeln: $F \to F + F - -F + F$

2 Teil 2

2.1 Berechenbarkeit und Komplexität

2.1.1 Entscheidbarkeit

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

Programm P ⟶ Entscheidungsverfahren ⟶ ja/nein

Eigenschaften können sein:

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

```
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 einen Gegenbeispiel zu der von Fermat aufgestellten Behauptung (Fermats letzter Satz), dass die Gleichung $x^n + y^n = z^n$ keine Lösung in $x, y, z \in \mathbb{N}$ für $n \geq 3$ besitzt. Dies war 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?

Programm P ⟶ Entscheidungsverfahren ⟶ ja, gibt HW aus/ nein

Man betrachte dazu das Programm

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

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

Um Berechenbarkeit zu untersuchen, verwenden wir Programme auf einen abstrakten mit unbegrenzten Speicher und Variablen, die beliebig große Werte annehmen können, als Berechnungsmodell.

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

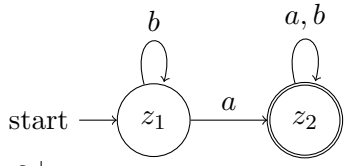
- Für die Eingabe $w \in L$ liefert P_L true
- Für die Eingabe $w \notin L$ liefert P_L false

In Pseudocode

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

Übung: Zeigen Sie, dass folgende Sprachen entscheidbar sind:

- $\varnothing$
- Σ^*
- Die Sprache aller Palindrome
- $\{(M, w) | M \text{ ist ein DFA mit } w \in L(M))\}$
- $\{M | M \text{ ist ein DFA mit } L(M) = \Sigma^*\}$



δ	z_1	z_2
a	z_2	z_2
b	z_1	z_2

$(\{z_1, z_2\}, z_1, \{\{z_2, z_2\}, \{z_1, z_2\}\}, \{z_2\})$

Die Srpache $\varnothing$ ist entscheidbar durch das Programm

```
1 boolean P_∅ (w) {
2     return false;
3 }
```

Σ^* ist entscheidbar durch das Programm

```
1 boolean P_Σ* (w) {
2     return true;
3 }
```

Die Sprache aller Palindrome ist entscheidbar durch

```
1 boolean P_Palindrom (w) {
2     for (i=1 to length(w) /2 )
3         if (w[i] ≠ w[length(w) -i])
4             return false;
5     return true;
6 }
```

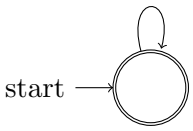
Die Sprache in $\{(M,w)|M$ ist ein DFA mit $w \in L(M))\}$ ist entscheidbar durch das Programm

```
1 boolean P_d (DFA M,word w) {
2     return simmulate(M,w);
3 }
```

wobei simmulate(M,w) den DFA M für die Eingabe w simuliert. Dies ist möglich (vgl. HA).

$\{M|M$ ist ein DFA mit $L(M) = \Sigma^*\}$ ist entscheidbar durch

```
1 boolean P_e (DFA M) {
2     return Minimalautomat(M) == M_Σ*;
3 }
```

wobei M_{Σ^*} der DFA 

HA: Ist $\{(M_1, M_2)|L(M_1) = L(M_2)\}$ entscheidbar

Das Halteproblem Das Halteproblem ist formal die Sprache

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

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

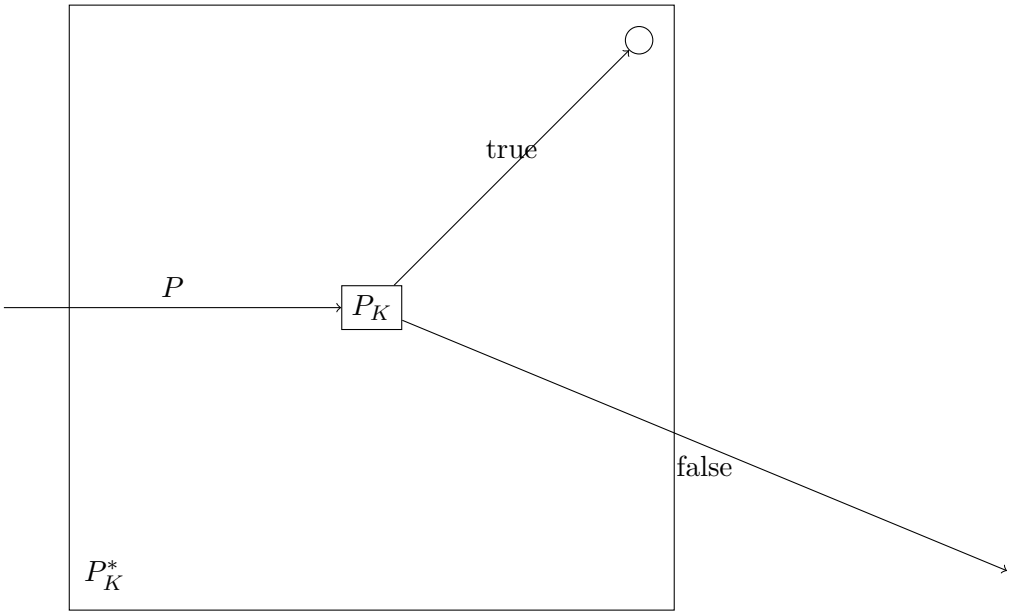
Satz Das spezielle Halteproblem

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

ist unentscheidbar.

Beweis: Angenommen, K ist entscheidbar durch ein Programm P_K.
Daraus konstruieren wir 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 entscheid, 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.

Weitere unentscheidbare Probleme Mit der Untentscheidbarkeit des speziellen Halteproblems können wir die Unentscheidbarkeit weiterer 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 lässt. Aus diesem Widerspruch folgt die Unentscheidbarkeit von B.

Satz H ist nicht entscheidbar.

Beweis: Angenommen H sei entscheidbar. Dann können wir folgendes Programm konstruieren, das das angenommene Enteschеidungsverfahren P_H für H als Unterprogramm verwendet:

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

Damit wäre aber P_K ein Entscheidungsverfahren für K, Widerspruch.
Damit folgt, dass auch die Programmverifikation unentscheidbar ist.

Satz

$$\text{Verify } = \{(P, S) | \text{Das Programm erfüllt die Spezifikation } S\}$$

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     return P_verify(P, P hält für die Eingabe w)
3 }
```

das dann ein Entscheidungsverfahren für H ist, Widerspruch.

Satz

$$H_\varepsilon = \{P | P \text{ hält für die Eingabe } \varepsilon\}$$

ist nicht entscheidbar.

Beweis: Sei wieder angenommen, H_ε sei entscheidbar, durch ein Programm P_{H_ε}.

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

Dann ist P_H aber ein Entscheidungsverfahren für das Halteproblem, denn $(P, w) \in H \Leftrightarrow P$ hält für $w \Leftrightarrow F$, hält für $\varepsilon \Leftrightarrow F \in H_\varepsilon$ und damit Widerspruch.

Weitere unentscheidbare Probleme Fast alle Probleme im Zusammenhang mit Programmverifikation, z.B.

- $\{P | P$ verursacht eine Division durch 0 für eine Eingabe $\}$
- $\{P | P$ verursacht einen Buffer-Overflow für eine Eingabe $\}$
- $\{(P_1, P_2) | P_1$ verhält sich wie $P_2\}$

Mathematische Probleme

- $\{F|$ Die durch die Formel F ausgedrückte Funktion ist $x \to 0\}$
- $\{S | S$ ist eine wahre mathematische Aussage $\}$

2.2 Komplexitätstheorie

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

2.2.1 Die Klassen P und NP

Für die $\mathcal{O}$-Notation gelten folgende Rechenregeln:

- $\mathcal{O}(f+g) = \mathcal{O}(f) + \mathcal{O}(g)$
- $\mathcal{O}(f+g) = \mathcal{O}(\max(f,g))$
- $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$ für eine Konstante $c > 0$
- $\mathcal{O}(f \cdot g) = \mathcal{O}(f) \cdot \mathcal{O}(g)$

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

Definition Die Komplexitätklasse P ist definiert durch

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

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     return w= reverse(w)
3 }
```

  stellt für alle $w \in \Sigma^*$ in der Zeit $\mathcal{O}(|w|)$ 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 Rekursive Descent Parser in Zeit $\mathcal{O}(n)$ entschieden werden kann.

- Jede kontextfreie Sprache. Es gibt einen Algorithmus der jede kontextfreie Sprache in Zeit $\mathcal{O}(n^3)$ 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 Adjazenzliste gegeben sei. Denn da die Adjazenzliste 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 $\mathcal{O}(|V| + |E|)$. Aus $|V| + |E| \leq n$ folgt:
  $\mathcal{O}(|V| + |E|) \subseteq \mathcal{O}(n)$. Damit liegt die Laufzeit in des Entscheidungsverfahren in $\mathcal{O}(n)$, woraus PFAD $\in P$ folgt.

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

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

z.B. gilt $(x \vee y) \wedge z \in$ SAT, $\neg x \wedge x \notin$ 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 in F).
Die Laufzeit dieses Entscheidungsverfahren liegt in $\mathcal{O}(2^n \cdot f(n))$, wobei $f(n)$ die Zeit ist, um den Wahrheitswert zu berechnen.

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 $\mathcal{O}(|F|)$ möglich ist.

Definition Die Komplexitätsklasse NP ist definiert durch

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

wobei n die Länge der Eingabe ist.

Beispiel: Es gilt SAT $\in$ NP, da mit einer erfüllenden Belegung für F in Zeit $\mathcal{O}(|F|)$ 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 $\mathcal{O}(n^k)$. Dies ist ebenfalls ein Verifizierungsverfahren das kein Zertifikat verwendet.

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

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

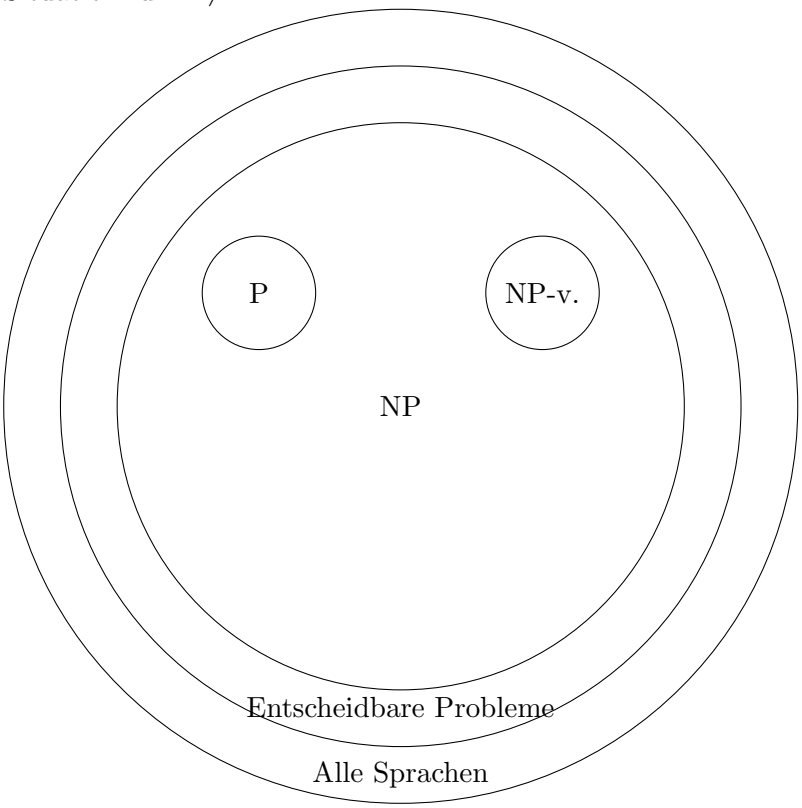
2.2.2 NP-vollständige Probleme

Die NP-vollständigen Probleme sind eine Klasse von Problemen in NP, für die keine effizienten Entscheidungsverfahren bekannt sind. Alle bekannten Verfahren besitzen 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 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 NP-vollständiges Problem in P gefunden wurde.

Situation für P $\neq$ NP:



Satz Das Erfüllbarkeitsproblem der Aussagenlogik

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

ist NP-vollständig.

Bemerkung: Das schnellste bekannte Verfahren für SAT hat die Laufzeit $\mathcal{O}(1,308^n \cdot p(n))$, wobei p ein Polynom ist.

Anwendungen:

- Autokonfiguration: Die bestellbaren Konfigurationen kann man durch Regeln in Form von aussagenlogischen Formelen beschreiben, z.B:
  Sportfahrwerk $\rightarrow$ Leichtm $\wedge$ (Motor 2.0 $\vee$ Motor 2.5 $\vee$ 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 ist.

- Vereinfachen von Schaltkreisen
  Angenommen, es ist ein Referenzentwurf 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, der vereinfachte Schaltkreis durch eine Formel F_S.
  Dann muss $F_R \leftrightarrow F_S$ eine Tautologie sein. Dies ist gleichwertig damit, dass $\neg(F_R \leftrightarrow F_S)$ unerfüllbar ist. D.h., beide Schaltkreise sind gleichwertig genau dann, wenn $\neq (F_R \leftrightarrow F_S) \notin \text{SAT}$.

Satz Das Problem HAMILTON-KREIS $= \{G|$ Der Graph G besitzt einen Hamiltonkreis $\}$ ist NP-vollständig.

Verallgemeinerung davon: Traveling Salesman Problem (TSP). Gegeben n Städte und Verbindungen zwischen diesen Städten, gesucht ist eine kürzeste Rundreise durch alle Städte.

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

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

- Platine bohren: Um die Bohrzeit für eine Platine zu minimieren, muss ein TSP- für die Bohrlöcher gelöst werden.
- 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 eine 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 Supersequence) 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 Short Common Superstring bzw. kürzester Hamiltonpfad-Pfad-Problem gelöst.

Beispiel Ursprüngliche Sequenz: $ATGCAA$
Bruchstücke: ATG, CAA, GCA, TGC

Algorithmen für TSP:

- Nearest Neighbor Greedy Algorithmus:
  Starte in einem beliebigen Knoten
  Solange noch nicht alle Knoten besucht wurden: Wähle einen nächsten unbesuchten Nachbarn des aktuellen Knoten aus und verlängere den aktuellen Pfad

  Greedy-Algorithmen wählen in jeden 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 eine optimale Lösung liefert, mit Laufzeit in $\mathcal{O}(n^2 2^n)$, 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. Für das metrische TSP gibt es einen Approximationsalgorithmus mit Laufzeit in $\mathcal{O}(n^2 \log n)$, der eine Lösung liefert, die eine Länge $\leq 2 \cdot$ optimale Länge besitzt.

Das 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

$$\sum_{k \in C} s_k \leq S$$

und

$$\sum_{k \in C} s_k \text{ maximal}$$

Anwendungen:

- CD mit mp3 optimal befüllen
- Budget maximal verbrauchen

Das Rucksackproblem ist NP-vollständig.

Algorithmus für Rücksack:
Sei $r(k, s)$ die maximale Füllmenge eines Rucksackes der 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 $1 \leq k \leq n, 0 \leq s \leq S$ mit Hilfe einer

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

- Wenn der Gegenstand k nicht eingepackt wird, gilt

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

- Wenn der Gegenstand k eingepackt wird, gilt

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

Daraus ergibt sich

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

Laufzeit: $\mathcal{O}(nS)$.
Dies ist kein Widerspruch dazu, dass Rucksack NP-vollständig ist, denn die Länge der Eingabe ist $\log_2 S + \sum_{k=1}^{n} \log_2 s_k$
$S = 2^{\log_2 S}$