Theoretische Informatik

Einführung in die Theorie der formalen Sprachen

Technische Hochschule Rosenheim

SS 2019

Prof. Dr. J. Schmidt

Inhalt

- Definition von formalen Sprachen
- Die Chomsky-Hierarchie
- Das Pumping-Theorem
- Die Analyse von Wörtern
- Anwendung: Compiler

DEFINITION
FORMALE SPRACHEN

Einführung

- Für die Programmierung von Rechnern ist *natürliche Sprache* oder mathematische Formelsprache nicht geeignet
- daher: Entwicklung an Rechner angepasster *Programmiersprachen*
- Auffälligster Unterschied zu natürlichen Sprachen:
  - streng formalisierten Sprachregeln der Programmiersprachen
  - sowie deren geringer Sprachumfang
    - Wortschatz und
    - Regeln
- hier: Behandlung einiger grundlegender Eigenschaften von *formalen Sprachen*
- diese sind die theoretischen Grundlagen von Programmiersprachen und Compilern

Beispiel

- Sprache $L = \{10^n1 \mid n \in \mathbb{N}_0\}$

- Grammatik:
  Terminale: $\{0, 1\}$
  Nichtterminale: $\{Z, A\}$
  Startsymbol: Z

  $Z \to 1A1, \quad Z \to 11$
  $A \to 0A, \quad A \to 0$

  üblich: BNF
  $Z \to 11 \mid 1A1$
  $A \to 0A \mid 0$

- Ableitung von 100001:
  $Z \to 1A1 \to 10A1 \to 100A1 \to 1000A1 \to 100001$

Definition: Formale Sprache

- **Vokabular** $V = T \cup S$ mit
  - endliches Alphabet T von *terminalen Zeichen*
    üblich: Verwendung von Kleinbuchstaben
  - endliche Menge S aus *nicht-terminalen Zeichen* (Variablen), wozu mindestens das *Startsymbol* Z gehört
    üblich: Verwendung von Großbuchstaben
- endliche Menge P von ***Produktionen***,
  d.h. *Ableitungsregeln* $u \rightarrow v$ mit $u \in V^+$, $v \in V^*$

- **Syntax** oder Ableitungsstruktur:
  - Gesamtheit Ableitungsregeln
- **Grammatik**: Ableitungsstruktur bei expliziter Unterteilung des Vokabulars V in
  - terminale Zeichen T und
  - nicht-terminale Zeichen S

Weitere Begriffe

- Sprache L
  - die aus dem Startsymbol Z in endlich vielen Schritten ableitbaren, nur aus terminalen Zeichen bestehenden Wörter
- zu L komplementäre Sprache: T*\L
- Formale Sprache besteht also aus
  - Grammatik
  - zugehörige Sprache

Backus Naur Form (BNF)

- für Produktionen mit gleicher linker Seite
  $A \rightarrow v_1$, $A \rightarrow v_2$, …, $A \rightarrow v_n$
- wird oft abkürzend die BNF verwendet:
  $A \rightarrow v_1 \mid v_2 \mid \dots \mid v_n$

- enger Zusammenhang zwischen formalen Sprachen und Automaten
- Zustandsübergang: eine Produktion
- Eingabealphabet: Menge der Terminalsymbole
- Zustandsmenge: Menge der nicht-Terminalsymbole
- Anfangszustand: Startsymbol Z
- Endzustand: Produktion, die zu leerem Wort ε oder einem Terminalsymbol führt
- akzeptierte Sprache: Wörter $x \in T^*$ akzeptiert, die sich aus Z ableiten lassen

Technische Hochschule Rosenheim

DIE CHOMSKY-HIERARCHIE

Überblick

Wesentliche Beiträge zur Klassifizierung formaler Sprachen:

- dem norwegischen Mathematiker A. Thue (1863 - 1922)
- und seit ca. 1955 von dem amerikanischen Linguisten Noam Chomsky (*1928)

- Einteilung von Grammatiken und Sprachen in die sog. **Chomsky-Hierarchie**
- Klassifikation der Grammatiken nach Art der zulässigen Produktionen von Typ 0 (am allgemeinsten) bis Typ 3 (am weitesten eingeschränkt)

Chomsky-Hierarchie (Grammatiken)

Technische Hochschule **Rosenheim**

- Typ 0 (**allgemeine** Grammatik)
  - keine Einschränkung, außer dass sich aus einem nur aus terminalen Zeichen bestehenden Wort kein anderes Wort mehr ableiten lässt
  - Produktionen haben also die Form:
    $xAy \rightarrow u$ mit $A \in S^+$ und $x,y,u \in V^*$

- Typ 1 (**kontextsensitive** Grammatik)
  - Produktionen haben die Form:
    $xAy \rightarrow xuy$ mit $x,y \in V^*$, $A \in S$ und $u \in V^+$
  - zusätzlich: $Z \rightarrow \varepsilon$, (dann darf Z nicht auf rechter Seite auftreten)
  - Anmerkung:
    - bis auf $Z \rightarrow \varepsilon$ sind diese Regeln monoton (nicht verkürzend)
    - umgekehrt gilt: alle monotonen Grammatiken definieren die gleiche Sprache wie kontextsensitive Grammatiken: $u \rightarrow v$, $|u| \leq |v|$ $u,v \in V^*$
    - die Regeln sind aber dann nicht mehr notwendigerweise in der obigen Form
    - manche Autoren (z.B. Schöning) verwenden diese als Definition von Typ 1 Grammatiken

Chomsky-Hierarchie (Grammatiken)

Technische Hochschule Rosenheim

- Typ 2 (**kontextfreie** Grammatik)
  - Produktionen haben die Form:
    $A \to u$ mit $A \in S$ und $u \in V^+$
  - zusätzlich: $Z \to \varepsilon$, (dann darf Z nicht auf rechter Seite auftreten)

- Typ 3 (**reguläre** Grammatik)
  - Produktionen haben die Form:
    $A \to u$ mit $A \in S$ und entweder $u \in T \cup TS$ oder $u \in T \cup ST$
    - rechtslineare Produktion:
      $A \to uB$ mit $A,B \in S$ und $u \in T^+$
    - linkslineare Produktion:
      $A \to Bu$ mit $A,B \in S$ und $u \in T^+$
    - terminale Produktion:
      $A \to u$ mit $A \in S$ und $u \in T^+$
  - zusätzlich: $Z \to \varepsilon$, (dann darf Z nicht auf rechter Seite auftreten)
  - Produktionen müssen entweder **alle** *rechtslinear* oder **alle** *linkslinear* sein

Definierte Sprache

- Eine Sprache L heißt vom Typ i (i = 0, 1, 2, 3), falls eine Chomsky Grammatik G vom Typ i existiert, mit L(G) = L

- Es gilt: $L_3 \subset L_2 \subset L_1 \subset L_0$
- Anmerkung:
  - eine Sprache bleibt auch dann vom Typ i, wenn man dafür eine Grammatik vom Typ j mit j < i angibt
  - z.B. kann man für eine reguläre Sprache durchaus eine kontextsensitive Grammatik angeben

Definierte Sprache

Es gibt Sprachen, die allgemeiner sind als Typ 0

- Typ 0 Sprachen heißen auch (rekursiv) *aufzählbare Sprachen*
- die dazu gehörigen Wörter können durch einen Algorithmus nacheinander erzeugt werden
- es existiert also eine Abbildung der natürlichen Zahlen auf die Menge der Wörter einer aufzählbaren Sprache, die durch eine berechenbare Funktion vermittelt wird
- prinzipiell kann jede Teilmenge $L \in T^*$ als Sprache aufgefasst werden
- die Menge aller Teilmengen (Potenzmenge) einer abzählbar unendlichen Menge (wie T^*) ist überabzählbar
- es muss folglich auch Sprachen geben, die nicht durch eine Grammatik erzeugt werden
- und die auch nicht durch eine TM (und damit einen Computer) akzeptiert werden

Beispiel

$L = \{10^n1 \mid n \in \mathbb{N}_0\}$ (siehe auch Kap. 1)

- Startsymbol: S

- Grammatik, Typ 0:
  $S \rightarrow 1A1,\ A \rightarrow 0A,\ A \rightarrow \varepsilon$
  - Ableitung von 100001:
    $S \rightarrow 1A1 \rightarrow 10A1 \rightarrow 100A1 \rightarrow 1000A1 \rightarrow 10000A1 \rightarrow 100001$

- Grammatik, Typ 2:
  $S \rightarrow 11,\ S \rightarrow 1A1,\ A \rightarrow 0A,\ A \rightarrow 0$
  - Ableitung von 100001:
    $S \rightarrow 1A1 \rightarrow 10A1 \rightarrow 100A1 \rightarrow 1000A1 \rightarrow 100001$

- Grammatik, Typ 3:
  $S \rightarrow 11,\ S \rightarrow 1A$
  $A \rightarrow 0A,\ A \rightarrow 0B$
  $B \rightarrow 1$
  - Ableitung von 100001:
    $S \rightarrow 1A \rightarrow 10A \rightarrow 100A \rightarrow 1000A \rightarrow 10000B \rightarrow 100001$

- Die Sprache L ist also regulär

Äquivalenz Grammatik – Automat

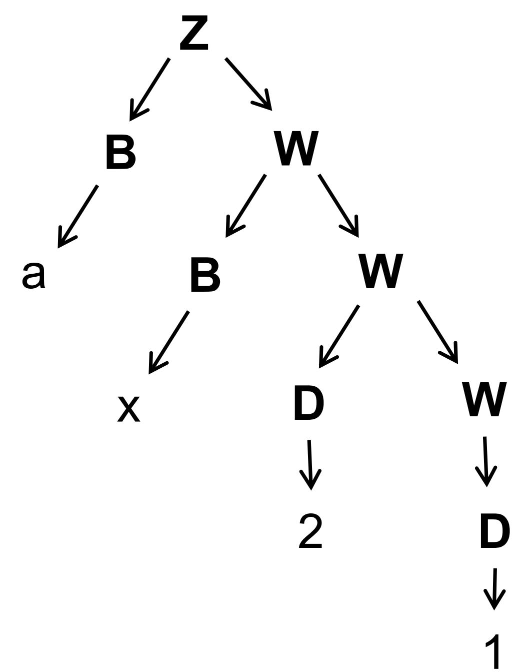
Sprache	Grammatik	Automat
Menge aller Sprachen ohne Typ 0	-	-
Typ 0	allgemeine Grammatik	Turingmaschine
Typ 1	kontextsensitive / monotone Grammatik	nichtdeterministischer linear beschränkter Automat
Typ 2	kontextfreie Grammatik	nichtdeterministischer Kellerautomat
det. kontextfrei	LR(k) Grammatik	deterministischer Kellerautomat
Typ 3	reguläre Grammatik	endlicher Automat

Beispiel: Bezeichner

- Bezeichner sind in den meisten Programmiersprachen nach gewissen Regeln frei wählbar
- C: Zeichenkette aus
  - Buchstaben
  - Unterstrich (_)
  - Dezimalziffern
  - als erstes Zeichen ist nur Buchstabe oder Unterstrich zugelassen
- Formale Sprache:
  S = {**Z**, **W**, **B**, **D**}
  T = {a, b, ..., z, A, B, ..., Z, _, 0, 1, ..., 9}
  P = { **Z** → **B**, **Z** → **BW**,
  W → **D**, **W** → **B**, **W** → **DW**, **W** → **BW**,
  B → a | b | ... | Z | _, **D** → 0 | 1 | ... | 9}
- Zu Vermeidung von Verwechslungen: Variablen fett, terminale Zeichen normal gedruckt

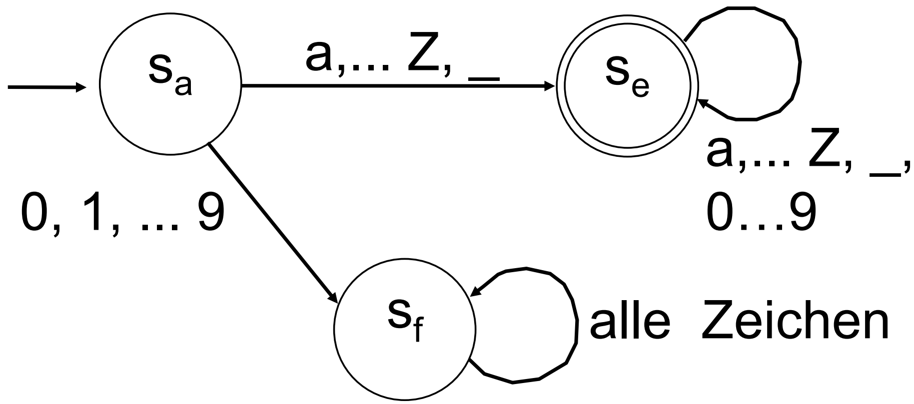
Beispiel: Bezeichner

- Ableitung des Wortes ax21
  Z → **BW** → a**W** → a**BW** → ax**W** → ax**DW** → ax2**W** → ax2**D** → ax21
- Darstellung als Syntaxbaum:
  Anzahl Nachfolger eines Knotens = Wortlänge der rechten Seite der angewendeten Produktion



Beispiel: Bezeichner

- Darstellung als deterministischer endlicher Automat



- offensichtlich ist die Sprache also regulär (Typ 3)
- die angegebene Grammatik ist aber kontextfrei (Typ 2)

Beispiel: Bezeichner

- Typ 3 Grammatik, nur rechtslineare Produktionen

  Z → a | b | … | Z | _
  Z → a**W** | b**W** | … | Z**W** | _**W**
  W → a | b | … | Z | _ | 0 | 1 | … | 9
  W → a**W** | b**W** | … | Z**W** | _**W** | 0**W** | 1**W** |… 9**W**

- Anmerkungen
  - es wird hier keine Beschränkung der Länge der Bezeichners vorgenommen
  - Längenbeschränkung ist mit endlichem Automaten nur realisierbar, wenn für jede zulässige Länge ein eigener Endzustand eingeführt wird
  - Mit Hilfe eines Kellerautomaten wäre dieses Problem aber ohne weiteres lösbar, indem man in einer zusätzlichen Kellervariablen über die Länge des Namens Buch führt

Beispiel: Typ 1 Sprache

$L = \{a^n\, b^n\, a^n \mid n \in \mathbb{N}\}$

- $S = \{Z, A, B\}$
  $T = \{a, b\}$
  $P = \{ \quad Z \rightarrow aba \mid aZA \mid a^2bBa$
  $BA \rightarrow bBa$
  $aA \rightarrow Aa$
  $B \rightarrow ba \quad \}$
- Grammatik ist monoton und definiert damit eine Typ 1 Sprache
- die Regel $aA \rightarrow Aa$ ist nicht kontextsensitiv, daher ist die Grammatik Typ 0

Beispiel: Typ 1 Sprache

- Ableitung für das Wort $a^4b^4a^4$:
  $Z \rightarrow aZA \rightarrow aaZAA \rightarrow a^2a^2bBaAA \rightarrow a^4bBAaA \rightarrow a^4b^2BaaA \rightarrow a^4b^2BaAa \rightarrow a^4b^2BAa^2 \rightarrow a^4b^2bBaa^2 \rightarrow a^4b^3baa^3 \rightarrow a^4b^4a^4$
- oder alternativ:
  $Z \rightarrow aZA \rightarrow aaZAA \rightarrow a^2a^2bBaAA \rightarrow a^4bBAaA \rightarrow a^4bBAAa \rightarrow a^4bbBaAa \rightarrow a^4b^2BAa^2 \rightarrow a^4b^2bBaa^2 \rightarrow a^4b^3baa^3 \rightarrow a^4b^4a^4$
- Ableitung ist also nicht eindeutig
- Bezeichnung:
  - Wörter mit eindeutiger Ableitung: eindeutige Wörter
  - Sprachen, die nur aus eindeutigen Wörtern bestehen: eindeutige Sprachen

Beispiel: Sackgasse

- Folge von Produktionen stoppt, bevor ein Wort bestehend aus rein terminalen Zeichen erreicht ist
- Beispiel:
  $Z \rightarrow aZA \rightarrow a^3bBaA \rightarrow a^3bbaaA \rightarrow a^3b^2aAa \rightarrow a^3b^2Aa^2$
- Die Kette führt in eine Sackgasse

Abgeschlossenheit

- Regeln für die Verknüpfung von Sprachen (Operationen)
- seien L_1, L_2, L Sprachen
- Mögliche Operationen:
  - Vereinigung: $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ oder } w \in L_2\}$
  - Durchschnitt: $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ und } w \in L_2\}$
  - Komplement: $\bar{L} = \{w \mid w \in T^* \text{ ohne } L\}$
  - Konkatenation: $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ und } w_2 \in L_2\}$
  - Kleenesche Hülle: $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

- Eine Klasse von formalen Sprachen heißt *abgeschlossen* unter einer Operation, wenn die resultierende Sprache zur selben Klasse gehört wie die Ausgangssprache(n).

Abgeschlossenheit

Sprache	Durchschnitt	Vereinigung	Komplement	Konkatenation	Kleenesche Hülle
Typ 3	ja	ja	ja	ja	ja
det.kf.	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 0	ja	ja	nein	ja	ja

Beispiel

- Typ 2 Sprachen sind unter Durchschnittsbildung nicht abgeschlossen
- betrachte die Typ 2 Sprachen
  - $L_1 = \{ a^i b^k c^k \mid i, k > 0\}$
  - $L_2 = \{ a^i b^i c^k \mid i, k > 0\}$
- Schnittmenge:
  - $L_1 \cap L_2 = \{ a^i b^i c^i \mid i > 0\}$
  - diese Sprache ist, wie im vorherigen Beispiel erläutert, Typ 1

REGULÄRE AUSDRÜCKE

Reguläre Ausdrücke

Technische Hochschule Rosenheim

- Mittel zur formalen Beschreibung von Zeichenketten bzw. Wörtern, die zu einer bestimmten Sprache gehören
- **Metasprache**, eng verwandt mit BNF und EBNF
  - deutlich flexibler und anschaulicher
  - aber nicht so universell.
- durch reguläre Ausdrücke beschreibbare Sprachen sind die regulären Sprachen
  - damit sind diese äquivalent zu endlichen Automaten

Reguläre Ausdrücke – Definition

Technische Hochschule **Rosenheim**

- Gegeben: Alphabet T
- Syntax
  - jedes Zeichen des Alphabets ist ein regulärer Ausdruck
  - die leere Menge $\emptyset$ ist ein regulärer Ausdruck
  - das leere Wort ε ist ein regulärer Ausdruck
  - wenn a und b reguläre Ausdrücke sind, dann auch
    - (a) (Klammerung)
    - ab (Konkatenation)
    - (a | b) (a oder b)
    - a* (Kleensche Hülle)
    - a^+ (positive Hülle – eigentlich unnötig, da $a^+ = aa^*$)
- Semantik
  - $L(\emptyset) = \emptyset$
  - $L(\varepsilon) = \{\varepsilon\}$
  - $L(x \in T) = \{x\}$
  - $L((a)) = L(a)$
  - $L(ab) = \{uv \mid u \in L(a) \text{ und } v \in L(b)\}$
  - $L((a \mid b)) = L(a) \cup L(b)$
  - $L(a^*) = L(a)^*$

Aufgabe

- Konstruieren Sie einen endlichen Automaten, der die Sprache akzeptiert, die durch folgenden regulären Ausdruck definiert ist:

  $a^+ (ba \mid b)^* c \mid b$

Reguläre Ausdrücke – Verwendung

Technische Hochschule Rosenheim

- dienen in Compilern zur Überprüfung, ob eine Zeichenkette syntaktisch korrekt gebildet ist
- Beschreibung oder Prüfung von semantischen Eigenschaften von Zeichenketten ist mit regulären Ausdrücken nicht möglich

- andere Einsatzgebiete
  - Textverarbeitung: Suchen, Ersetzen und Modifizieren von Mustern
  - Unix/Windows Shell
  - Bestandteil einiger Programmiersprachen: PHP, Perl, Python

- Achtung:
  - die im Folgenden gezeigten Konstrukte sind eine exemplarische Auswahl
    - sie unterscheiden sich je nach Sprache/Tool
    - manche sind evtl. nicht verfügbar
    - meist sind zusätzliche verfügbar
  - oft sind Konstrukte verfügbar, die über die Mächtigkeit regulärer Ausdrücke hinausgehen
    - das sind dann keine regulären Ausdrücke im Sinne der theoretische Informatik mehr
    - trotzdem werden sie in der Praxis leider so bezeichnet
  - Bezeichnung oft als *Regex* oder *Regexp* (von regular expression)

Regex – Metazeichen

- reservierte Zeichen in regulären Ausdrücken, mindestens:
  + ? . * ^ $ ( ) [ ] { } | \
- ist eines dieser Zeichen Bestandteil der beschriebenen Sprache, muss es maskiert werden
  - üblich: voranstellen von \, z.B.: \? für ?
  - andere Varianten existent
- jetzt: Tabelle der wichtigsten metasprachlichen Konstrukte
  - diese sind bei weitem nicht vollständig

RegEx – Konstrukte

Zeichen	Bedeutung
^	am Anfang eines Strings
$	am Ende eines Strings
.	beliebiges Zeichen
n?	optional vorhandenes n
n*	kein oder mehrfaches Vorkommen von n
n+	ein oder mehrere Vorkommen von n
n{2}	genau zweimaliges Vorkommen von n
n{3,}	mindestens 3 oder mehrere Vorkommen von n
n{4,11}	mindestens 4, höchstens 11 Vorkommen von n

Zeichen	Bedeutung
()	Klammern für Ausdrücke
(n|a)	Entweder n oder a
[1-6]	eine Ziffer zwischen 1 und 6
[d-g]	ein Kleinbuchstabe zwischen d und g
[E-H]	ein Großbuchstabe zwischen E und H
[^a-z]	kein Vorkommen von Kleinbuchstaben zwischen a und z
[_a-zA-Z]	ein Unterstrich und ein beliebiger Buchstabe des Alphabets
[:space:]	Leerzeichen
\	Escape-Zeichen, z.B. \? für ?, \r für neue Zeile

Beispiel: Dezimalzahlen

- Verbale Beschreibung
  - erstes Zeichen: Minuszeichen (optional): [-]?
  - es folgen beliebig viele Ziffern, mindestens aber eine: [0-9]+
  - danach kann ein Dezimalpunkt stehen: \.
  - wenn dies der Fall ist, können noch beliebig viele Ziffern folgen, mindestens aber eine: [0-9]+
- regulärer Ausdruck: [-]?[0-9]+(\ .[0-9]+)?

Beispiel: Wörter und natürliche Zahlen

Technische Hochschule **Rosenheim**

- Beschreibung eines Strings
  - aus natürlichen Zahlen und
  - aus Wörtern mit einer beliebigen Anzahl von Buchstaben
  - wobei die Zahlen bzw. Wörter durch Leerzeichen getrennt sind
- regulärer Ausdruck:
  ^ [a-zA-Z]+ | ([1-9][0-9]*) ([:space:][a-zA-Z]+ | ([1-9][0-9]*))* $
- Hinweis:
  - ^ bzw. $ markieren **nicht** Beginn und Ende des Strings im Sinne von z.B. Anführungszeichen „…“
  - sondern:
    - ^ bedeutet: Übereinstimmung nur, wenn der nachfolgende Ausdruck am **Anfang einer Zeichenkette** vorkommt
    - $ bedeutet: Übereinstimmung nur, wenn der nachfolgende Ausdruck am **Ende einer Zeichenkette** vorkommt
  - Hier:
    - Abcd generiert eine Übereinstimmung
    - öä:Abcd generiert keine Übereinstimmung
      (würde man ^ weglassen, dann hätte man auch hier eine Übereinstimmung)

DAS PUMPING THEOREM

Einführung

- Pumping Theorem
  - wichtiger Satz für reguläre Grammatiken
    (und damit für endliche Automaten)
  - kann für sehr viele weiterführende Aussagen und Beweise über reguläre Sprachen genutzt werden
  - insbesondere nützlich, wenn für eine Sprache gezeigt werden soll, dass sie **nicht** regulär ist
  - oft auch: Pumping Lemma

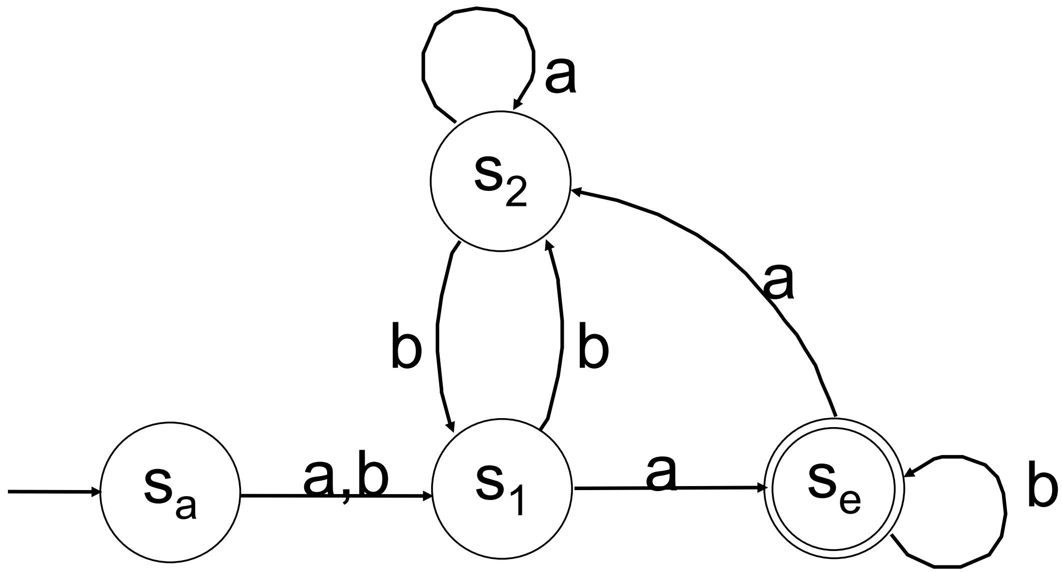
- ein ähnlicher Satz existiert für kontextfreie Sprachen

Pumpen von Wörtern

Technische Hochschule Rosenheim

- sei w ein Wort aus der Sprache einer regulären Grammatik
- ist w lang genug, kann es immer aus drei Teilen zusammengesetzt werden: w = xyz
- „Pumpen" bedeutet: Vervielfachung von y, z.B.
  w' = xyyz, w'' = xyyyz, …
- dies muss möglich sein, weil
  - jeder endliche Automat mit unendlich großer Sprache Zyklen durchlaufen muss
  - daher müssen auch in den Wörtern Wiederholungen auftreten

Beispiel



- Wort w = aa
  - gehört zur Sprache
  - ist aber zu kurz zum Pumpen
- Wort w = abba
  - gehört auch zur Sprache
  - kann gepumpt werden
  - mit x = a, y = bb, z = a erhält man w‘ = abbbba, w“ = abbbbbba, …

Pumping Theorem für reguläre Sprachen

Technische Hochschule Rosenheim

- sei L eine **reguläre** Sprache
- dann gibt es eine Konstante n, so dass sich jedes Wort $w \in L$, mit $|w| \geq n$ zerlegen lässt in $w = xyz$ mit
  - $|xy| \leq n$
  - $|y| \geq 1$
  - $|z|$ beliebig (also auch 0)
- Es gilt dann: $x\,y^i\,z \in L$, für alle $i = 0, 1, 2, \ldots$

Beispiel: Palindrome

- Palindrom: Wort, das vorwärts und rückwärts gelesen gleich ist, z.B. abba, otto, reittier
- Behauptung: Die Sprache L = {w | w ist ein Palindrom auf T} mit T = {a, b} ist nicht regulär
- Beweis durch Widerspruch

Beispiel: Palindrome Beweis

- Annahme: es gibt eine reguläre Grammatik, die nur Palindrome auf T erzeugt
- dann muss das Pumping Theorem gelten
- $a^n b a^n$ ist ein Palindrom aus L
- $w = a^n b a^n = xyz$
- xy kann dann nur aus a‘s bestehen, da $|xy| \leq n$
- hat xy maximale Länge, dann gilt: $xy = a^n$
- insbesondere enthält y dann mindestens ein a
- Pumping Theorem: auch xyyz muss zu L gehören
- xyy enthält aber mindestens ein a mehr als z, d.h.
  $xyyz = a^m b a^n$ mit $m > n$
- xyyz ist also kein Palindrom: Widerspruch!
- L kann also nicht regulär sein
- und es kann auch keinen endlichen Automaten geben, der nur Palindrome über T akzeptiert

weitere nicht reguläre Sprachen

Technische Hochschule **Rosenheim**

Mit Hilfe des Pumping Theorems kann z.B. von folgenden Sprachen nachgewiesen werden, dass sie nicht regulär sind:

- $L = \{a^n\, b^n \mid n \in \mathbb{N}\}$
  - diese Sprache ist kontextfrei
  - damit ist auch bewiesen, dass die Menge der kontextfreien Sprachen tatsächlich umfassender ist als die Menge der darin enthaltenen regulären Sprachen
- $L = \{0^q \mid q$ ist eine Quadratzahl$\}$
- $L = \{0^p \mid p$ ist eine Primzahl$\}$
  - und damit: es gibt keinen endlichen Automaten, der für eine gegebene Zahl entscheiden kann, ob sie prim ist.

Pumping Theorem für kontextfreie Sprachen

Technische Hochschule **Rosenheim**

- sei L eine **kontextfreie** Sprache
- dann gibt es eine Konstante n, so dass sich jedes Wort $w \in L$, mit $|w| \geq n$ zerlegen lässt in $w = uvxyz$ mit
  - $|vxy| \leq n$
  - $|vy| \geq 1$
  - $|u|, |x|, |z|$ beliebig (also auch 0)
- Es gilt dann: $u\, v^i\, x\, y^i\, z \in L$, für alle $i = 0, 1, 2, \ldots$

Anwendung

- Nachweis, dass eine Sprache nicht kontextfrei ist
- dies kann z.B. von folgenden Sprachen nachgewiesen werden:
- $L = \{a^n\, b^n\, c^n \mid n \in \mathbb{N}\}$
  - diese Sprache ist kontextsensitiv
  - damit ist auch bewiesen, dass die Menge der kontextsensitiven Sprachen tatsächlich umfassender ist als die Menge der darin enthaltenen kontextfreien Sprachen
- $L = \{0^q \mid q$ ist eine Quadratzahl$\}$
- $L = \{0^p \mid p$ ist eine Primzahl$\}$
  - und damit: es gibt keinen Kellerautomaten, der für eine gegebene Zahl entscheiden kann, ob sie prim ist.

Anmerkung

- mit dem Pumping Theorem kann nur gezeigt werden, dass eine Sprache **nicht** regulär/kontextfrei ist
- es kann **nicht** gezeigt werden, dass sie regulär/kontextfrei ist
- es gibt Sprachen, die das Pumping Theorem erfüllen, die aber nicht regulär/kontextfrei sind

- Beispiel:
  - $L = \{ a^i b^k c^k \mid i, k > 0 \} \cup \{ b^j c^k \mid j,k \geq 0 \}$
  - erfüllt das Pumping Theorem für reguläre Sprachen
  - ist aber nicht regulär

ANALYSE VON WÖRTERN

Wortproblem und Parsing Problem

Technische Hochschule Rosenheim

- Wortproblem: entscheide von einem Wort x, ob es **wohlgeformt** ist (konform zu den Regeln)

- Parsing Problem (Zerteilungsproblem)
  - vollständige Analyse des Wortes x
  - Ableitung von x wird bis zum Startsymbol Z zurückverfolgt
  - es müssen alle Schritte von $Z \rightarrow x$ bestimmt werden

- Fragen:
  - sind diese Probleme für alle Sprachklassen lösbar?
  - wie schwierig/zeitaufwändig ist die Entscheidung?

Wortproblem für reguläre Sprachen (Typ 3)

Technische Hochschule Rosenheim

- lösbar durch deterministischen endlichen Automaten
- stoppt die Verarbeitung in einem Endzustand, so ist das analysierte Wort Teil der Sprache
- Zeitaufwand: **linear** bzgl. der Wortlänge

Wortproblem für kontextfreie Sprachen (Typ 2)

Technische Hochschule **Rosenheim**

- lösbar, wenn Grammatik in Chomsky Normalform ist
- CYK-Algorithmus (Cocke, Younger, Kasami)
- Zeitaufwand: $O(n^3)$ bzgl. der Wortlänge
  - für praktische Zwecke (z.B. Syntaxanalyse) zu langsam
  - deshalb: typischerweise Beschränkung auf LR(k) Grammatiken
  - diese haben lineares Laufzeitverhalten

Chomsky Normalform

- Für jede kontextfreie Grammatik G mit $\varepsilon \notin$ G kann man eine Grammatik G‘ angeben mit L(G) = L(G‘), die in Chomsky Normalform ist
  - $\varepsilon \in$ G: $Z \rightarrow \varepsilon$, Z darf nicht auf rechter Seite auftreten

- Eine Grammatik ist in Chomsky Normalform, wenn alle Regeln die Form haben
  (A, B, C Variablen, a Terminalsymbol)
  - $A \rightarrow BC$
  - $A \rightarrow a$

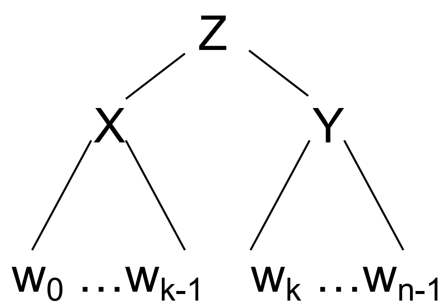
Chomsky Normalform Konstruktion

Technische Hochschule **Rosenheim**

- Ordne jedem Terminalsymbol a eine neue Variable V_a zu
  - füge der Grammatik alle Produktionen $V_a \rightarrow a$ hinzu
  - ersetze auf allen rechten Seiten Terminalsymbole durch V_a
- Ersetze Regeln mit mehr als 2 Variablen auf rechter Seite
  - aus $A \rightarrow B_1B_2\ldots B_k$, $k \geq 3$ werden folgende Regeln:
  - $A \rightarrow B_1V_2$
$V_2 \rightarrow B_2V_3$
…
$V_{k-1} \rightarrow B_{k-1}B_k$
- Ersetze Produktionen der Form $A \rightarrow B$
  - entferne alle Regeln $A \rightarrow B$
  - Füge für jede Regel $B \rightarrow b$ eine Regel $A \rightarrow b$ hinzu

CYK-Algorithmus

- gegeben: Wort der Länge n, $w = w_0 w_1 \ldots w_{n-1} \in T^*$
- Fall n = 1, d.h. $w = w_0$
  - da Grammatik in CNF: Regel $Z \rightarrow w_0$ muss existieren
- Fall n > 1
  - da Grammatik in CNF: das Wort muss aus 2 Teilwörtern bestehen
  - diese lassen sich über eine Produktion $Z \rightarrow XY$ ableiten

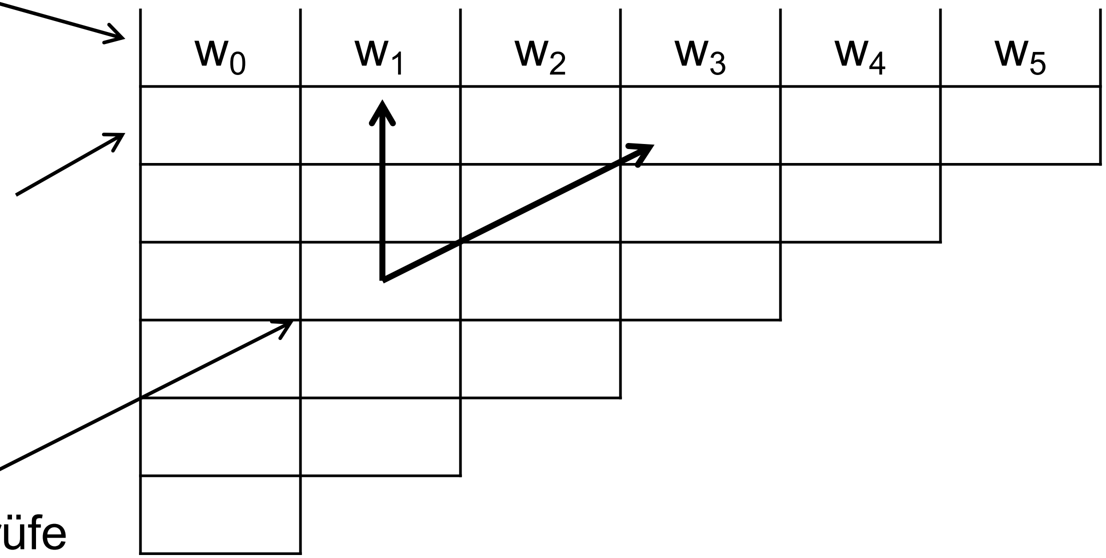


  - Reduktion des Problems auf 2 Teilwörter der Länge k und n – k
  - k steht am Anfang nicht fest, d.h. man muss alle möglichen Trennungen betrachten
- benötigt wird eine Tabelle der Größe n x n, von der aber nur die Hälfte der Einträge besetzt ist

CYK-Algorithmus Prinzip

Beginne mit Wort in oberster Zeile (nicht in Tabelle gespeichert)

Trage in jedes Feld der 1. Zeile alle Variablen ein, aus denen sich w_i erzeugen lässt



Für jeden Eintrag: Prüfe senkrecht nach oben und diagonal, ob es in der richtigen Entfernung eine Regel gibt, die das Teilwort erzeugt; falls ja: trage die Variable(n) ein

CYK-Algorithmus Prinzip

Welche Variablenmenge im Eintrag T_{ij} erzeugt welches Teilwort:

$i \rightarrow$

$j \downarrow$

w_0	w_1	w_2	w_3	w_4	w_5
T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
T_{02}	T_{12}	T_{22}	T_{32}		
T_{03}	T_{13}	T_{23}			
T_{04}	T_{14}				
T_{05}					

CYK-Algorithmus Prinzip

Welche Regeln werden betrachtet? Suche senkrecht und diagonal, aber im richtigen Abstand!

$i \longrightarrow$

$j \downarrow$

w_0	w_1	w_2	w_3	w_4	w_5
T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
T_{02}	T_{12}	T_{22}	T_{32}		
T_{03}	T_{13}	T_{23}			
T_{04}	T_{14}				
T_{05}					

CYK-Algorithmus Prinzip

Welche Regeln werden betrachtet? Suche senkrecht und diagonal, aber im richtigen Abstand!

i

	w_0	w_1	w_2	w_3	w_4	w_5
	T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
	T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
j	T_{02}	T_{12}	T_{22}	T_{32}		
	T_{03}	T_{13}	T_{23}			
	T_{04}	T_{14}				
	T_{05}					

CYK-Algorithmus Prinzip

Technische Hochschule Rosenheim

Welche Regeln werden betrachtet? Suche senkrecht und diagonal, aber im richtigen Abstand!

$i \longrightarrow$

$j \downarrow$

w_0	w_1	w_2	w_3	w_4	w_5
T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
T_{02}	T_{12}	T_{22}	T_{32}		
T_{03}	T_{13}	T_{23}			
T_{04}	T_{14}				
T_{05}					

CYK-Algorithmus Prinzip

Welche Regeln werden betrachtet? Suche senkrecht und diagonal, aber im richtigen Abstand!

i

j

w_0	w_1	w_2	w_3	w_4	w_5
T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
T_{02}	T_{12}	T_{22}	T_{32}		
T_{03}	T_{13}	T_{23}			
T_{04}	T_{14}				
T_{05}					

CYK-Algorithmus Prinzip

Welche Regeln werden betrachtet? Suche senkrecht und diagonal, aber im richtigen Abstand!

i

j

w_0	w_1	w_2	w_3	w_4	w_5
T_{00}	T_{10}	T_{20}	T_{30}	T_{40}	T_{50}
T_{01}	T_{11}	T_{21}	T_{31}	T_{41}	
T_{02}	T_{12}	T_{22}	T_{32}		
T_{03}	T_{13}	T_{23}			
T_{04}	T_{14}				
T_{05}					

CYK-Algorithmus
Beispiel

- geg.: Grammatik G in CNF
  $Z \to AB$, $A \to CD \mid CF$, $B \to c \mid EB$
  $C \to a$, $D \to b$, $E \to c$, $F \to AD$
- ist $w = aaabbbcc \in L(G)$?

a	a	a	b	b	b	c	c
C	C	C	D	D	D	B, E	B, E

CYK-Algorithmus
Beispiel

Z → AB, A → CD | CF, B → c | EB

C → a, D → b, E → c, F → AD

a	a	a	b	b	b	c	c
C	C	C	D	D	D	B, E	B, E
		A				B	

CYK-Algorithmus
Beispiel

$Z \rightarrow AB,\ A \rightarrow CD \mid CF,\ B \rightarrow c \mid EB$
$C \rightarrow a,\ D \rightarrow b,\ E \rightarrow c,\ F \rightarrow AD$

a	a	a	b	b	b	c	c
C	C	C	D	D	D	B, E	B, E
		A				B	
		F					

CYK-Algorithmus
Beispiel

Technische Hochschule Rosenheim

$Z \rightarrow AB$, $A \rightarrow CD \mid CF$, $B \rightarrow c \mid EB$
$C \rightarrow a$, $D \rightarrow b$, $E \rightarrow c$, $F \rightarrow AD$

a	a	a	b	b	b	c	c
C	C	C	D	D	D	B, E	B, E
		A				B	
		F					
	A						
	F						
A							
Z							
Z							

Wortproblem für kontextsensitive Sprachen (Typ 1)

Technische Hochschule **Rosenheim**

- lösbar, da Grammatik monoton sein muss
- für ein Wort der Länge n dürfen also alle Zwischenergebnisse höchstens n Zeichen lang sein
- die Anzahl der Wörter mit Länge n über einem endlichen Alphabet ist endlich
- daher muss es einen Algorithmus geben, der das Wortproblem löst:
- es müssen alle möglichen Ableitungen durchprobiert werden
- Zeitaufwand: $O(a^n)$ bzgl. der Wortlänge
  - für praktische Zwecke nicht verwendbar
  - (a = Anzahl Zeichen des Alphabets)

Wortproblem für Typ 0 Sprachen

- in Grammatiken können bei Ableitungen Sackgassen auftreten (auch für andere Sprachklassen als Typ 0)
  - die Ableitung muss nicht eindeutig sein
  - bei der Rückverfolgung kann man auf einen Weg gelangen, der gar nicht zum Startsymbol Z führt
  - bei **Typ 1** Sprachen (und damit auch Typ 2, 3) ist garantiert, dass eine **Sackgasse endliche** Länge hat
  - bei **Typ 0** Sprachen kann eine Sackgasse auch **unendlich** lang sein

- Das Wortproblem für Typ 0 Sprachen ist unlösbar!
  - Es gibt **keinen** Algorithmus, der für alle Typ 0 Sprachen entscheiden kann, ob ein Wort w von einer gegebenen Typ 0 Grammatik erzeugt wird oder nicht
  - das Problem ist **unentscheidbar**

Leerheits-/Schnitt-/ Äquivalenzproblem

- Leerheitsproblem
  - gegeben: Grammatik G (oder äquivalenter Automat)
  - Frage: ist $L(G) = \varnothing$
- Schnittproblem
  - gegeben: zwei Grammatiken G_1 und G_2 (oder äquivalente Automaten)
  - Frage: ist $L(G_1) \cap L(G_2) = \varnothing$
- Äquivalenzproblem
  - gegeben: zwei Grammatiken G_1 und G_2 (oder äquivalente Automaten)
  - Frage: Definieren G_1 und G_2 die gleiche Sprache, d.h. ist $L(G_1) = L(G_2)$?

Wort-/Leerheits-/Schnitt-/Äquivalenzproblem

Technische Hochschule Rosenheim

- für welche Sprachklassen/Automatenmodelle ist das Problem entscheidbar (lösbar)?
- Einträge
  - ja: es gibt einen Algorithmus, der das Problem löst
  - nein: das Problem ist unlösbar, es gibt keinen Algorithmus dafür

Sprache	Wortproblem	Leerheitsproblem	Äquivalenzproblem	Schnittproblem
Typ 3	ja	ja	ja	ja
det.kf.	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

Zusammenfassung – formale Sprachen

Technische Hochschule Rosenheim

- Definition formaler Sprachen
  - terminale/nichtterminale Symbole, Produktionen (Regeln)
  - Sprache L(G): alle aus Startsymbol ableitbaren Wörter
- Chomsky-Hierarchie
  - Typ 0 bis 3 Grammatik/Sprache, $L_3 \subset L_2 \subset L_1 \subset L_0$
  - Äquivalenz Grammatiken/Automatenmodelle
  - Abschlusseigenschaften (Durchschnitt, Vereinigung, Komplement, Konkatenation, Stern/Kleene'sche Hülle)
  - Reguläre Ausdrücke zur Beschreibung regulärer Sprachen
- Pumping Theorem
  - für reguläre/kontextfreie Sprachen
  - Nachweis, dass eine Sprache nicht regulär/kontextfrei ist
- Analyse von Wörtern
  - Wortproblem
    - reguläre Sprache: endlicher Automat
    - kontextfreie Sprache: CYK-Algorithmus
    - kontextsensitive Sprache: alle Varianten durchtesten
    - Typ 0 Sprache: unlösbar
  - Leerheits-, Schnitt- und Äquivalenzproblem

ANWENDUNG: COMPILER

Definition

- Compiler (Übersetzer):
  - Programm, das die Anweisungen eines in einer Programmiersprache P1 (Quellsprache) geschriebenen Programms in Anweisungen einer anderen Programmiersprache P2 (Zielsprache) überträgt
- Compiler muss einem **Quellprogramm** a $\in$ P1 genau ein semantisch äquivalentes (bedeutungsgleiches) **Zielprogramm** b $\in$ P2 zuordnen
- hierfür werden formale Sprachen verwendet
- Zielprogramm soll möglichst effizient ablaufen
  - optimierende Compiler
  - Laufzeit und/oder der Speicherbedarf von b werden minimiert
  - Zielprogramm b $\in$ P2 modifiziert
  - Beibehaltung der **semantischen Äquivalenz** zwischen a und b

Arten von Compilern

- Compiler im engeren Sinn
  - Quellsprache P1 ist eine höhere Sprache als Zielsprache P2
- Assembler (Assemblierer)
  - Compiler zur Übertragung von ASSEMBLER-Quellprogrammen in Maschinensprache
- Cross-Compiler
  - Compiler erzeugt Zielcode, der auf einer anderen Plattform läuft als der Compiler selbst
    - anderes Betriebssystem und/oder CPU
- Präprozessor/Präcompiler
  - Übersetzung von Spracherweiterungen vor eigentlicher Compilierung
- Compiler-Compiler
  - Programm zur Generierung eines Compilers aus einer formalisierten Sprachbeschreibung
  - z.B.: YACC (Yet Another Compiler Compiler)

Arten von Compilern

- Interpreter (Interpretierer)
  - Anweisungen des Quellprogramms werden übersetzt und sofort ausgeführt (während des Programmablaufs)
  - Vorteil:
    - Test während Entwicklung sehr schnell möglich, ohne separate Compilierung
    - wichtiges Instrument, wenn Programm ohne Änderung auf Rechnern mit unterschiedlichen Betriebssystemen und unterschiedlicher Hardware laufen sollen
  - Nachteil: Zur Ausführungszeit kommt immer die Übersetzungszeit hinzu
    - kostet besonders bei Schleifendurchläufen viel Zeit
  - Beispiele: BASIC, LISP, PROLOG, Python,
    mit Einschränkungen: Java

Schritte

- **Lexikalische Analyse**
  - Umwandlung des Quellprogramm $a \in P1$ mit **Scanner** in Zwischencode (**Token**)
  - Objekte der Sprache (z.B. Kommentare, Operatoren, Schlüsselwörter, Namen) werden als solche erkannt und in Token verwandelt
  - Es können hier bereits einfache Regelverletzungen gemeldet werden
    - z.B. Verwendung eines nicht zugelassenen Zeichens in einem Bezeichner
  - Beschreibung durch reguläre Grammatik/reguläre Ausdrücke
  - Realisierung durch (deterministischen) **endlichen Automat**

Schritte

- **Syntaktische Analyse**
  - **Parser** erzeugt aus Token entsprechend der Syntax von P1 den **Ableitungsbaum** (**Syntaxbaum**) des Programms $a \in P1$
  - Verwendung von deterministisch kontextfreien Grammatiken
    - Top-Down: LL(k) Grammatik, meist LL(1)
    - Bottom-Up: LR(k) Grammatik, meist LR(1)
  - realisiert als **deterministischer Kellerautomat**

- **Semantische Analyse**
  - Analyse des Ableitungsbaums von $a \in P1$
  - gleichzeitig: Code-Generator überträgt a in die Zielsprache P2
  - Ergebnis: Zielprogramm $b \in P2$
  - Prüfung der Semantik des Programms, z.B.
    - wurden alle verwendeten Variablen definiert/deklariert
    - werden sie typgerecht verwendet
    - gibt es evtl. Bereichsüberschreitungen
  - Verwendung von **(kontextfreien) Attributgrammatiken**

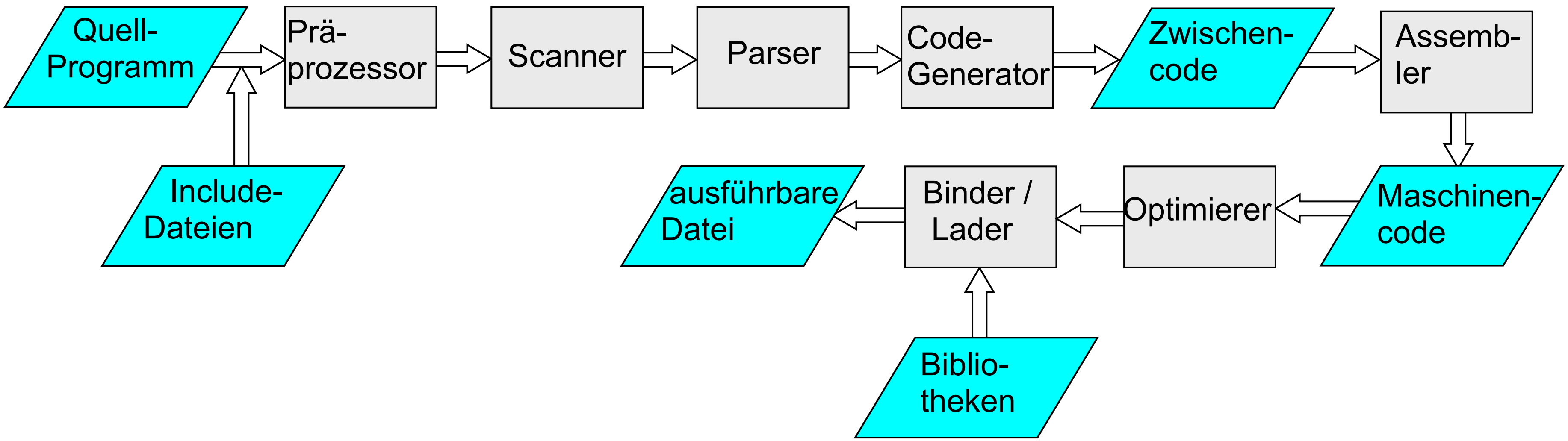
Schritte

- **Code-Optimierung**
  - Steigerung der Effizienz des Zielprogramms b $\in$ P2
  - Zeit- und/oder Speicherbedarf optimieren; oft Widerspruch
  - Programmcode b verändert
    - Semantik muss natürlich unverändert bleiben
  - Code-Optimierung ist zeitaufwändig
  - vollständiger Erhalt der Semantik von b kann nicht in jedem Fall garantiert werden kann
  - daher: optional

Linken (Binden)

- Ergebnis der Übersetzung ist **Objekt-Code** **(**kein lauffähiges Programm)
- wird durch ein separates Hilfsprogramm (**Binder**/**Linker**) in ein ausführbares Programm übertragen
- Grund:
  - aufgerufene Funktionen liegen oft nicht als Code vor, sondern sind in unabhängig erstellte Module oder in Standard-Bibliotheken ausgelagert
- Binder fügt die Objekt-Codes aller benötigten Module und Bibliotheken zu einem lauffähigen Programm zusammen

Schritte im Detail



Tools

- lex / flex: Lexikalische Analyse
  - lex: 1975
- yacc / bison: syntaktische/semantische Analyse
  - yacc: 1979
- erzeugen C-Code Dateien
- Links:
  - lex & yacc: http://dinosaur.compilertools.net/
  - flex: http://flex.sourceforge.net/
  - bison: http://www.gnu.org/software/bison/

“The asteroid to kill this dinosaur is still in orbit.”
(Lex Manual Page)

Zusammenfassung – Compiler

- Compiler: übersetzt Quell- in Zielprogramm
- Arten von Compilern
  - Compiler im engeren Sinn, Assembler, Cross-Compiler, Präprozessor, Compiler-Compiler, Interpreter
- (Haupt-)Schritte beim Übersetzungsvorgang
  - lexikalische Analyse
    - Scanner
    - reguläre Grammatik
    - Umwandlung in Token
  - syntaktische Analyse
    - Parser
    - Generierung eines Syntaxbaums
    - deterministisch kontextfreie Grammatik
  - semantische Analyse
    - Analyse des Syntaxbaums
  - Code-Generierung und –Optimierung
  - Linken
- Reguläre Ausdrücke
  - zur Beschreibung der Grammatiken für lexikalische und (in erweiterter Form) syntaktische Analyse

Quellen

Die Folien entstanden auf Basis folgender Literatur

- H. Ernst, J. Schmidt und G. Beneken: Grundkurs Informatik. Springer Vieweg, 6. Aufl., 2016.
- Schöning, U.: *Theoretische Informatik - kurz gefasst*. Spektrum Akad. Verlag (2008)
- Sander P., Stucky W., Herschel, R.: *Automaten, Sprachen, Berechenbarkeit*, B.G. Teubner, 1992