

# Einführung in die Informatik

## Grammars & Parsers

---

Grammatiken, Parsen von Texten

Wolfram Burgard  
Cyrill Stachniss

# Einleitung

---

- Wir haben in den vorangehenden Kapiteln meistens vollständige Java-Programme als mögliche Beschreibungen von Algorithmen angegeben.
- Einzelne Befehle einer Programmiersprache stellen **abstrakte Befehle** zur **Manipulation der Daten** und zur **Kontrolle der Ausführung** zur Verfügung.
- Damit werden sie **unabhängig von dem konkreten Rechner**.
- Der **Compiler** sorgt dann dafür, dass sie in die Sprache des entsprechenden Rechners übersetzt werden.
- Darüber hinaus erlauben sie eine **Strukturierung des Programms**, was die Wartung und Entwicklung deutlich vereinfacht.
- In diesem Kapitel beschäftigen wir uns mit der Beschreibung von künstlichen Sprachen (z.B. Programmiersprachen).

# Techniken zur Beschreibung der Syntax von Programmiersprachen

---

- Um Programme übersetzen zu können, muss der Rechner die Programmiersprache kennen, d.h. er muss wissen, was **gültige Programme** sind.
- Programmiersprachen sind **künstliche Sprachen**.
- Eine **Sprache** ist nichts anderes als eine **(im allgemeinen unendliche) Menge von Sätzen**, die jeweils aus **einzelnen Symbolen** bestehen.
- Die einzelnen, **nicht weiter unterteilbaren Symbole** nennt man auch **Tokens** der Sprache.
- Bei der Analyse eines Programms wird es in der so genannten **lexikalischen Analyse** zunächst in die Folge seiner Tokens zerlegt.

# Kategorien von Tokens

---

**Reservierte Wörter:** Dies sind **Schlüsselwörter**, wie z.B. `boolean`, `int`, `class`, `static` usw.

**Konstanten:** Dazu gehören **Literale**, wie z.B. `4711L` oder `"Mehrwertsteuer"`.

**Sonderzeichen:** Z.B. `+`, `-`, `=`, `;`, `...` für Operatoren und Begrenzer.

**Bezeichner:** Alle **benutzerdefinierten Namen** zur Benennung von Variablen, Methoden, etc

**Kommentare:** Besonders gekennzeichnete Zeichenfolgen, die **vom Compiler übersprungen** werden.

...

# Syntax von Programmiersprachen

---

- Unter der **Syntax einer Programmiersprache** versteht man die Regeln, die **festlegen, was gültige Programme sind**.
- Eine typische Form zur Formulierung dieser Regeln sind so genannte **kontextfreien Grammatiken**.
- Obwohl manche Teile der Syntax-Definition nicht durch kontextfreie Grammatiken erfasst werden können, hat diese Form der Syntaxbeschreibung große Vorteile.
- Dazu gehört insbesondere, dass **Analyseprogramme für Programme** (so genannte **Parser**) **automatisch konstruiert** werden können.

# Arithmetische Ausdrücke

---

**Arithmetische Ausdrücke** können induktiv folgendermaßen definiert werden:

1. Jede Zahl ist ein arithmetischer Ausdruck.
2. Ist  $E$  ein arithmetischer Ausdruck, so ist auch  $(E)$  ein arithmetischer Ausdruck.
3. Sind  $E_1$  und  $E_2$  arithmetische Ausdrücke, so sind auch  $E_1 + E_2$ ,  $E_1 - E_2$  und  $E_1 * E_2$ ,  $E_1 / E_2$  arithmetische Ausdrücke.
4. Nur die auf diese Weise erhältlichen Zeichenreihen sind syntaktisch korrekt gebildete, arithmetische Ausdrücke.

# Eine kontextfreie Grammatik für arithmetische Ausdrücke

---

$\langle \text{Ziffer} \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

$\langle \text{Zahl} \rangle \rightarrow \langle \text{Ziffer} \rangle | \langle \text{Ziffer} \rangle \langle \text{Zahl} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Zahl} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow (\langle \text{Ausdruck} \rangle)$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle + \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle - \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle * \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle / \langle \text{Ausdruck} \rangle$

# Typen von Symbolen in Grammatiken

---

Bei den einzelnen Symbolen unterscheidet man:

**Metasymbole:** Zeichen wie  $\rightarrow$  und  $|$ , die **zur Formulierung der Regeln benötigt** werden.

**Terminalsymbole:** Sie entsprechen den **Tokens der Sprache** und sind die einzigen Zeichen, die in Sätzen der Sprache auftreten können. In unserem Beispiel sind dies 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (, ), +, - und \*.

**Nichtterminalsymbole:** Dies sind die in spitze Klammern eingeschlossenen Zeichenreihen. Sie werden manchmal auch **Variablen oder syntaktische Kategorien** genannt.



# Bedeutung der Komponenten

---

- Die oben angegebene Grammatik besteht aus einer Menge von **Regeln** oder **Produktionen**.
- Mit Hilfe der Regeln lassen sich, ausgehend von einem Startsymbol (hier:  $\langle \text{Ausdruck} \rangle$ ), Zeichenreihen erzeugen, indem man ein Vorkommen **eines Nichtterminalsymbols**, das **auf der linken Seite einer Regel** vorkommt, in der Zeichenreihe **durch die rechte Seite ersetzt**.
- Tritt auf der rechten Seite das **Metasymbol |** auf, so fasst man die dadurch abgetrennten Zeichenreihen als **alternative Ersetzungsmöglichkeiten** auf.
- Die erste Regel ist somit als Abkürzung für zehn einzelne Regeln der Form:

$$\langle \text{Ziffer} \rangle \rightarrow n$$

für  $n = 0 \dots 9$  aufzufassen

# Anwendungsbeispiele

---

$\langle \text{Ziffer} \rangle \rightarrow 0|1|2|3|4|5|6|7|8|9$

$\langle \text{Zahl} \rangle \rightarrow \langle \text{Ziffer} \rangle | \langle \text{Ziffer} \rangle \langle \text{Zahl} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Zahl} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow (\langle \text{Ausdruck} \rangle)$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle + \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle - \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle * \langle \text{Ausdruck} \rangle$

$\langle \text{Ausdruck} \rangle \rightarrow \langle \text{Ausdruck} \rangle / \langle \text{Ausdruck} \rangle$

# Anwendungsbeispiel

---

$\langle \text{Ausdruck} \rangle \Rightarrow \langle \text{Ausdruck} \rangle * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (\langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (\langle \text{Ausdruck} \rangle + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (\langle \text{Zahl} \rangle + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (\langle \text{Zahl} \rangle \langle \text{Ziffer} \rangle + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (\langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (1 \langle \text{Ziffer} \rangle + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (17 + \langle \text{Ausdruck} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (17 + \langle \text{Zahl} \rangle) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (17 + 4) * \langle \text{Ausdruck} \rangle$   
 $\Rightarrow (17 + 4) * \langle \text{Zahl} \rangle \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * \langle \text{Zahl} \rangle \langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * \langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * 3 \langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * 3 \langle \text{Ziffer} \rangle \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * 37 \langle \text{Ziffer} \rangle$   
 $\Rightarrow (17 + 4) * 372$

# Eine formale Definition von Grammatiken

---

Eine **Grammatik** ist ein 4-Tupel  $G = (V, T, P, S)$  bestehend aus:

1. Einer Menge  $V$  von **Nichtterminalsymbolen** (Variablen),
2. einer Menge  $T$  von **Terminalsymbolen** (Tokens) mit  $V \cap T = \emptyset$ ,
3. einer Menge  $P$  von **Produktionen** (Regeln) der Form  $p \rightarrow q$  mit  $p \in (V \cup T)^*$  und  $q \in (V \cup T)^*$  sowie
4. einem **Startsymbol**  $S \in V$ .

Hierbei bedeutet  $v \in V^*$ , dass  $v$  eine **Zeichenkette** ist, die aus den Variablen in  $V$  gebildet wird.

# Ableitbarkeit

---

- Sei  $x=x_1\dots p\dots x_n$  ein Wort aus  $(V \cup T)^*$  das die linke Seite einer Regel  $p \rightarrow q \in P$  enthält.
- Dann kann man durch Ersetzen von  $p$  durch  $q$  in  $x$  ein Wort  $y=y_1\dots q\dots y_n$  erhalten und schreibt dafür:  $x \Rightarrow y$  oder, falls erforderlich  $x \underset{G}{\Rightarrow} y$ , oder  $x \underset{p \rightarrow q}{\Rightarrow} y$ .

Wir sagen, dass  **$y$  in einem Schritt aus  $x$  ableitbar** ist.

- **$y$  heißt ableitbar aus  $x$**  mithilfe von  $G$ , kurz  $x \xrightarrow[G]{*} y$ , wenn entweder  $x=y$  ist oder es Worte  $x_1, x_2, \dots, x_n$  gibt mit  $x=x_1$  und  $y=x_n$ , und  $x_i \underset{G}{\Rightarrow} x_{i+1}$ , für  $i=1, \dots, n-1$ .

# Von einer Grammatik erzeugte Sprache

---

- Die **von einer Grammatik erzeugte Sprache** ist die Menge der aus dem Startsymbol ableitbaren Worte, die nur aus Terminalzeichen bestehen:

$$L(G) = \{ W \mid S \xRightarrow{*} W \text{ und } W \in T^* \}.$$

- $L(G)$  ist also die **Menge aller aus dem Startsymbol in endlich vielen Schritten ableitbaren Terminalzeichenreihen**.
- Meistens gibt man nur die Produktionen von  $G$  an und legt damit  $V$  und  $T$  implizit fest.
- Für unsere Grammatik  $G$  zur Definition arithmetischer Ausdrücke ist also  $L(G)$  die Menge aller syntaktisch korrekt gebildeten, arithmetischen Ausdrücke.

# Beispiel (1)

---

$G=(V,T,P,S)$  sei eine Grammatik mit

$$V = \{S, A, B\}$$

$$T = \{a, b\}$$

$$P = \{S \rightarrow AB, A \rightarrow AA, A \rightarrow a, B \rightarrow BB, B \rightarrow b\}$$

Dann sieht man leicht, dass gilt:

$$L(G) = \{a^n b^m \mid n, m \geq 1 \text{ und } m, n \in \mathbb{N}\}.$$

## Beispiel (2)

---

$G=(V,T,P,S)$  sei eine Grammatik mit

$$V = \{S\}$$

$$T = \{a,b\}$$

$$P = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$$

wobei  $\varepsilon$  das leere Wort bezeichnet. Dann ist

$$L(G) = \{a^n b^n; n \in \mathbb{N}, n \geq 0\}.$$

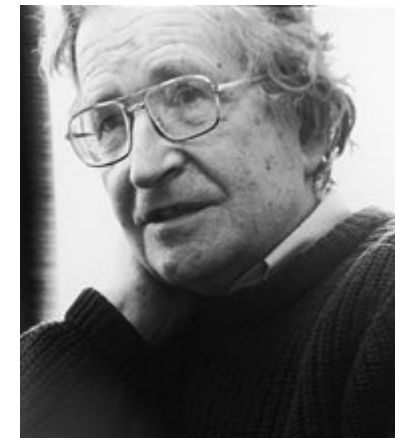
- Offensichtlich kann man mit Regeln dieser Art **Klammerstrukturen** erzeugen.
- In Java kommen solche Strukturen z.B. in Form von `{ . . . }` oder `if . . . then . . . else` vor.



# Typen von Grammatiken

---

- Nach der Art der zugelassenen Regeln klassifiziert man Grammatiken in verschiedene Typen (Chomsky-Hierarchie: Typ 0 bis Typ 3).
- Dies geht zurück auf Noam Chomsky, 1959, und wird in der theoretischen Informatik genauer studiert.
- Noam Chomsky ist eigentlich Linguist und kein Informatiker
- Prof. (Emeritus) am MIT in Boston
- Er ist einer der bedeutenden politischen Denker in den USA



# Die Chomsky-Hierarchie

Typ	Bezeichnung	Regelart $p \rightarrow q$
0	<b>unbeschränkte Grammatiken</b>	$p \in (V \cup T)^* \setminus \{\epsilon\}$ $q \in (V \cup T)^*$
1	<b>kontextsensitive Grammatiken</b> oder monotone Grammatiken	$p \in (V \cup T)^* \setminus \{\epsilon\}$ $q \in (V \cup T)^*$ $\text{Länge}(p) \leq \text{Länge}(q)$
2	<b>kontextfreie Grammatiken</b>	$p \in V$ $q \in (V \cup T)^*$
3	<b>reguläre Grammatiken</b> d.h. linkslineare Grammatiken oder rechtslineare Grammatiken mit	$p \rightarrow qt \mid \epsilon$ $p \rightarrow tq \mid \epsilon$ $p, q \in V, t \in T$

# Ein Beispiel für eine Typ-0-Grammatik

---

- Beide oben angegebenen Sprachen sind kontextfrei!
- Betrachten wir die folgende Typ-0-Grammatik:

$V = \{S, a, b, c, d, n\}, T = \{0, 1\},$

$P = \{S \rightarrow anb, d \rightarrow 0, d \rightarrow 1, n \rightarrow d, n \rightarrow dn, 1b \rightarrow b0, 0b \rightarrow c1, ab \rightarrow 1, 1c \rightarrow c1, 0c \rightarrow c0, ac \rightarrow \epsilon \}$

Anwendungen:

# Vergleich dieser Sprachtypen

---

- Eine Sprache  $L$  heißt vom Typ  $i$ ,  $0 \leq i \leq 3$ , wenn es eine Grammatik vom Typ  $i$  gibt mit  $L=L(G)$ .
- Bezeichnet  $L_i$  die Familie der Sprachen vom Typ  $i$ , so gilt der (hier nicht bewiesene) **Hierarchiesatz**:

$$L_0 \supset L_1 \supset L_2 \supset L_3$$

- Somit ist jede Sprache vom Typ  $i$  echt mächtiger als eine Sprache vom Typ  $i+1$ .

# Existiert eine Grammatik für Java?

---

- Es ist möglich, eine Grammatik  $G$  anzugeben, so dass  $L(G)$  die Menge der korrekt gebildeten Java-Programme ist (bis auf die semantischen Nebenbedingungen).
- Aufgrund der semantischen Nebenbedingungen kann ein der Grammatik entsprechendes Java-Programm eventuell nicht übersetzbar sein.
- Beispielsweise kann man aus der Grammatik nicht ableiten, dass eine Variable erst deklariert werden muss, bevor sie benutzt wird.
- Eine Grammatik für Java findet man z.B. in: R. Kühnel, Java Fibel, Addison Wesley, 1996, Anhang A.

# Backus-Naur-Form (BNF)

---

- Für die Spezifikation der Syntax von Programmiersprachen verwendet man häufig kontextfreie Grammatiken.
- Allerdings werden die Regeln meist in der **Backus-Naur-Form** (BNF) dargestellt:
  - Anstelle von  $\rightarrow$  wird  $::=$  (oder  $=$ ) verwendet.
  - Mit dem Metasymbol  $|$  werden Regeln zusammengefasst.
  - Nichtterminalsymbole werden als in spitzen Klammern eingeschlossene Zeichenreihen repräsentiert.
- Beispiel:

$$\begin{aligned} \langle \text{Zahl} \rangle & ::= \langle \text{Ziffer} \rangle \mid \langle \text{Zahl} \rangle \langle \text{Ziffer} \rangle \\ \langle \text{Ziffer} \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

# Erweiterte Backus-Naur-Form (EBNF)

---

- Hierbei stehen Terminalzeichen in Anführungszeichen oder werden anderweitig hervorgehoben.
- Für  $n \geq 0$  Wiederholungen verwendet man die Notation  $\{ \dots \}$ .
- Optionale Teile hingegen werden in eckige Klammern  $[ \dots ]$  eingeschlossen.
- Lässt man führende Nullen zu, so kann man eine Integer-Zahl folgendermaßen beschreiben:

$\text{Integer} ::= [+|-]\text{Ziffer}\{\text{Ziffer}\}$

# Techniken zum Parsen

---

Chomsky	Technik
0	<b>Turing-Maschine</b>
1	<b>Linear beschränkte Turing-Maschine</b>
2	<b>Kellerautomat (Stack Machine)</b>
3	<b>Endlicher Automat</b>



# Wie funktioniert ein Parser

---

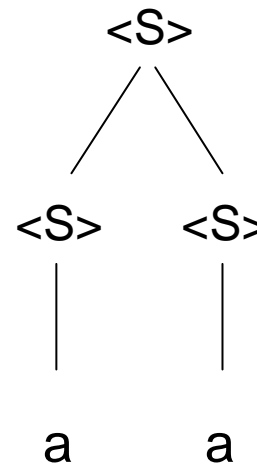
- Top-Down oder Bottom-Up Realisierung
- Meist wird ein so genannter Kellerautomat (Stack Machine) verwendet, da oft kontextfreie Grammatiken verwendet werden
- Idee:
  1. Verarbeitung der Eingabe von rechts nach links
  2. Abarbeiten (Ersetzen) eines Strings sobald dies möglich ist
  3. Ansonsten auf dem Stack abspeichern und diesen Ausdruck abarbeiten sobald dies möglich ist

# Parser-Baum

---

- Die Anwendung einer Grammatik kann auch als Baum dargestellt werden
- Die Blattelemente beschreiben dabei den zu parsenden String
- Die inneren Elemente bezeichnen die Regel nach der der String ersetzt wird

- Regeln  
 $\langle S \rangle ::= \langle S \rangle \langle S \rangle$   
 $\langle S \rangle ::= a$
- String  
aa



# Beispiel HTML-Tabelle parsen

---

- Regeln

<TAB> ::= “<table>” <ROWS> “</table>”

<ROWS> ::= <ROWS><ROWS> | “<tr>” <COLS> ”</tr>”

<COLS> ::= <COLS><COLS> | “<td>” <TEXT> “</td>” |  
“<td>” <TAB> “</td>”

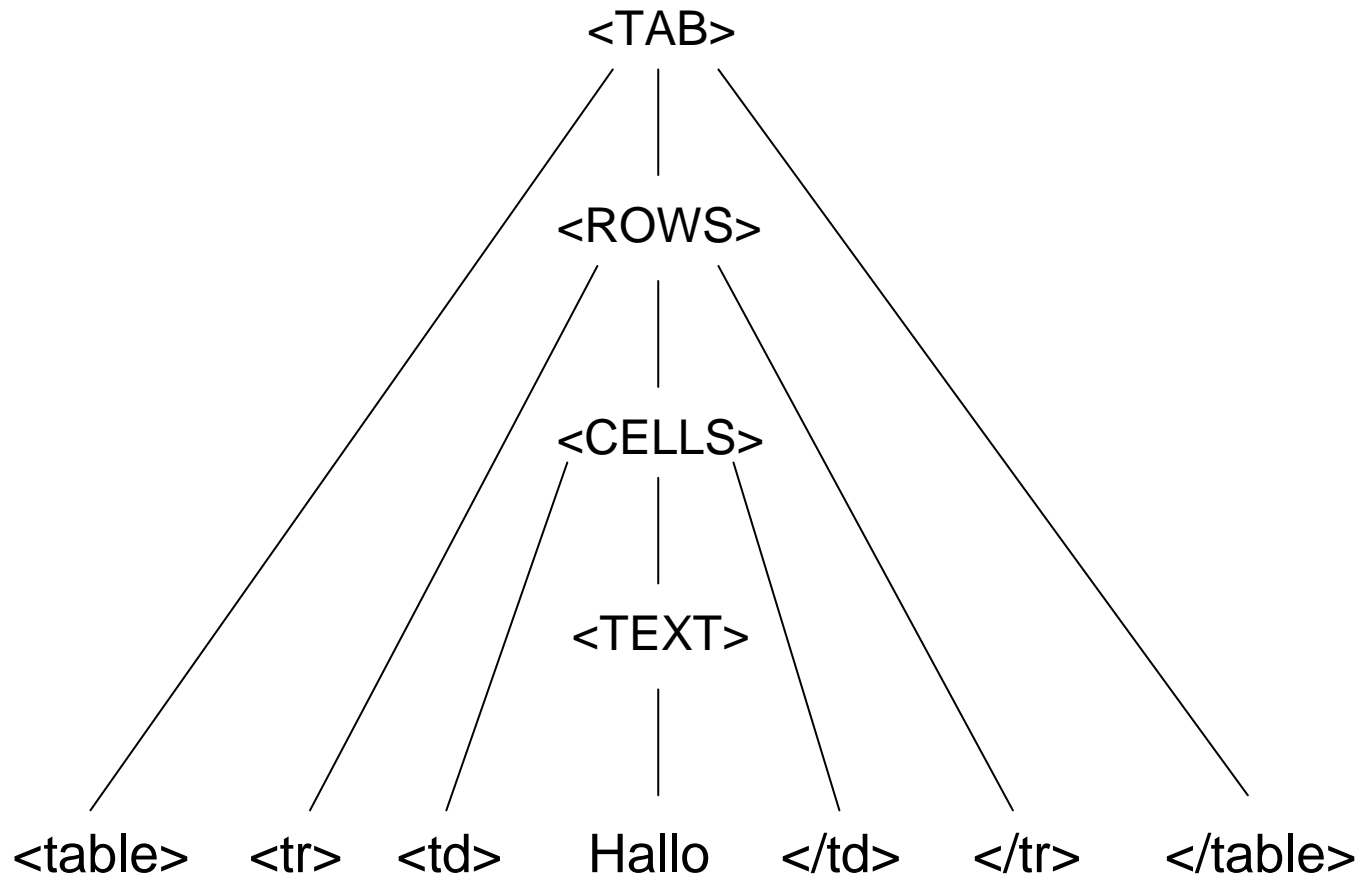
<TEXT> ::= .....

- Eingabe

<table><tr><td>Hallo</td></tr></table>

# Parser-Baum

---



# Lex und Yacc

---

- Tools zum einfachen Bauen von Parsern
- Lex
  - Lexikalische Analyse
- Yacc
  - Parsegenerator für kontextfreie Grammatiken
  - Nutzt einen Kellerautomat

# Zusammenfassung

---

- **(Imperative) Programmiersprachen** stellen **abstrakte Anweisungen zur Manipulation von Daten** zur Verfügung.
- Zur Beschreibung von Programmiersprachen verwendet man **Grammatiken**.
- Die von einer **Grammatik erzeugte Sprache** ist die Menge aller in **endlich vielen Schritten ableitbaren, variablenfreie Zeichenketten**.
- Je nach **Struktur der Regeln** sind **Grammatiken unterschiedlich mächtig**.
- Die **Chomsky-Hierarchie klassifiziert** verschiedene Arten von **Grammatiken**.
- Die **(erweiterte) Backus-Naur-Form** ist eine typische Art **Grammatiken von Programmiersprachen zu notieren**.