Universität Bremen

Praktische Informatik 1
Grammatik und Parsen

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

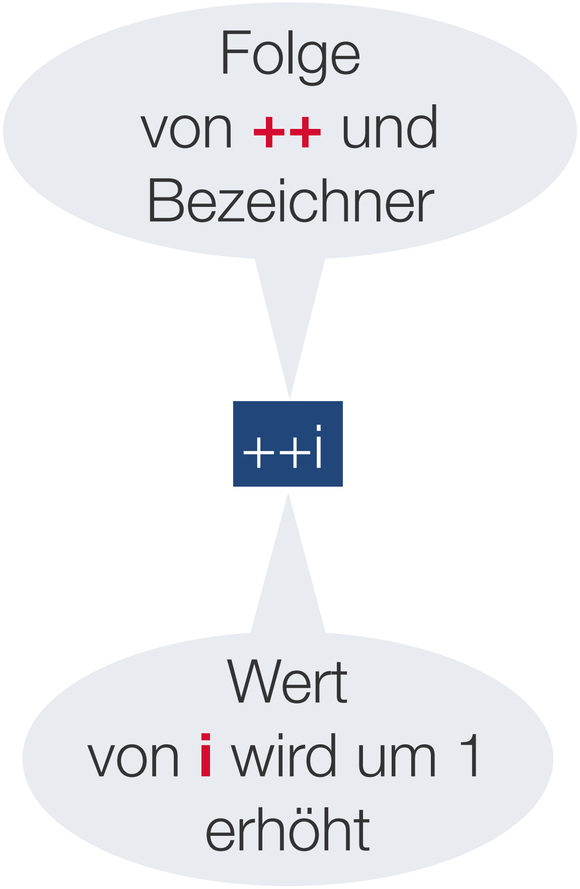
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

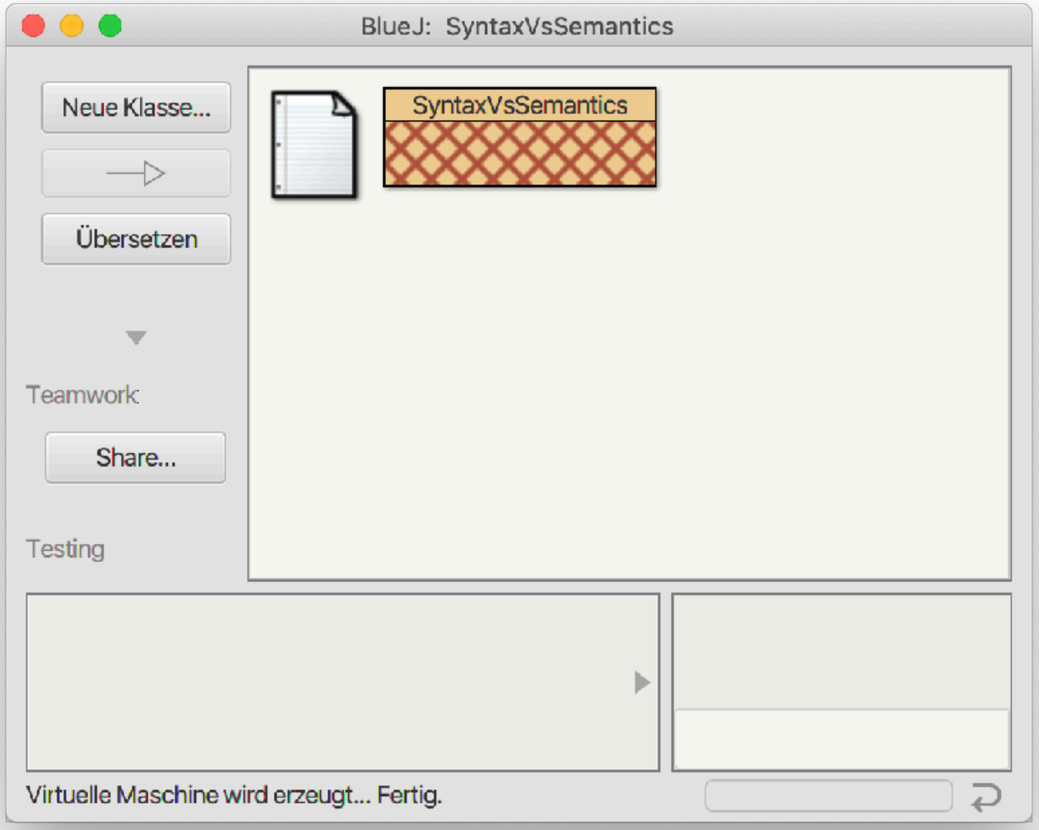
Syntax und Semantik

- **Syntax**: Korrekte Bildung von Sätzen aus sprachlichen Elementen
  - Ist durch formale **Grammatik** eindeutig beschrieben
  - **Lexikalische Analyse**: Zerlegung in **Tokens** (Bezeichner, Literale, Schlüsselwörter, Operatoren…)
- **Semantik** legt die Bedeutung der Sprachkonstrukte fest
  - Formale Beschreibung der Semantik aufwendig (aber möglich)
  - Daher oft informelle Beschreibung der Semantik



Universität Bremen

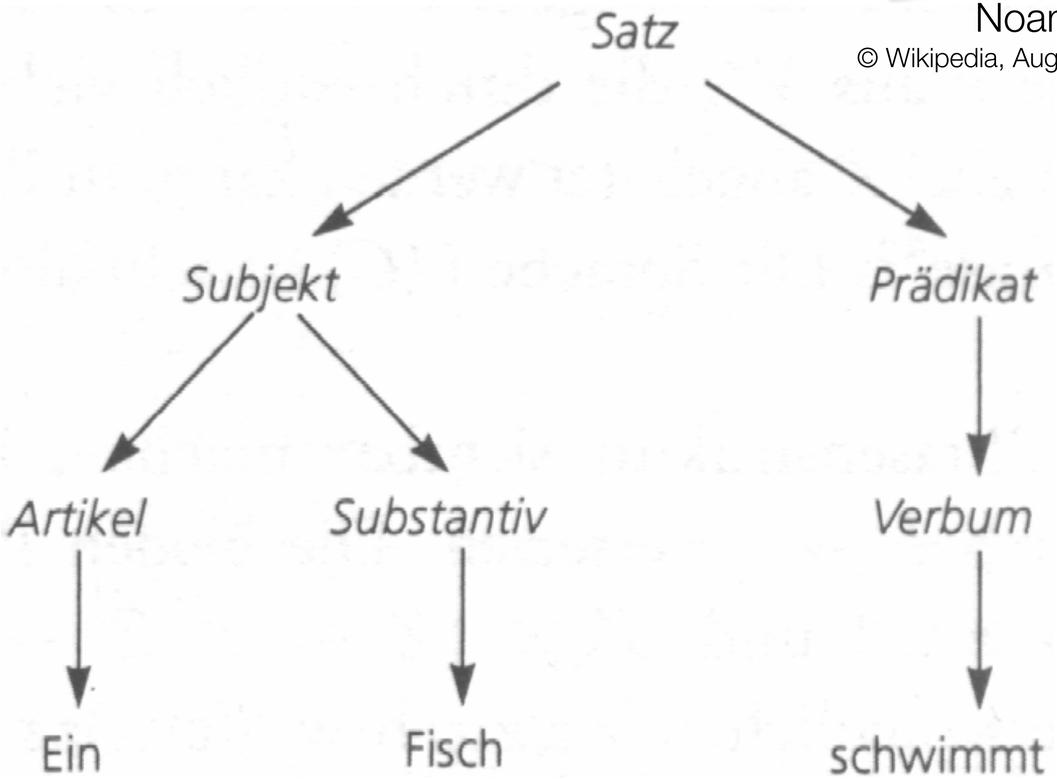
Syntax und Semantik in Java: Demo



Chomsky-Grammatiken: Produktionen

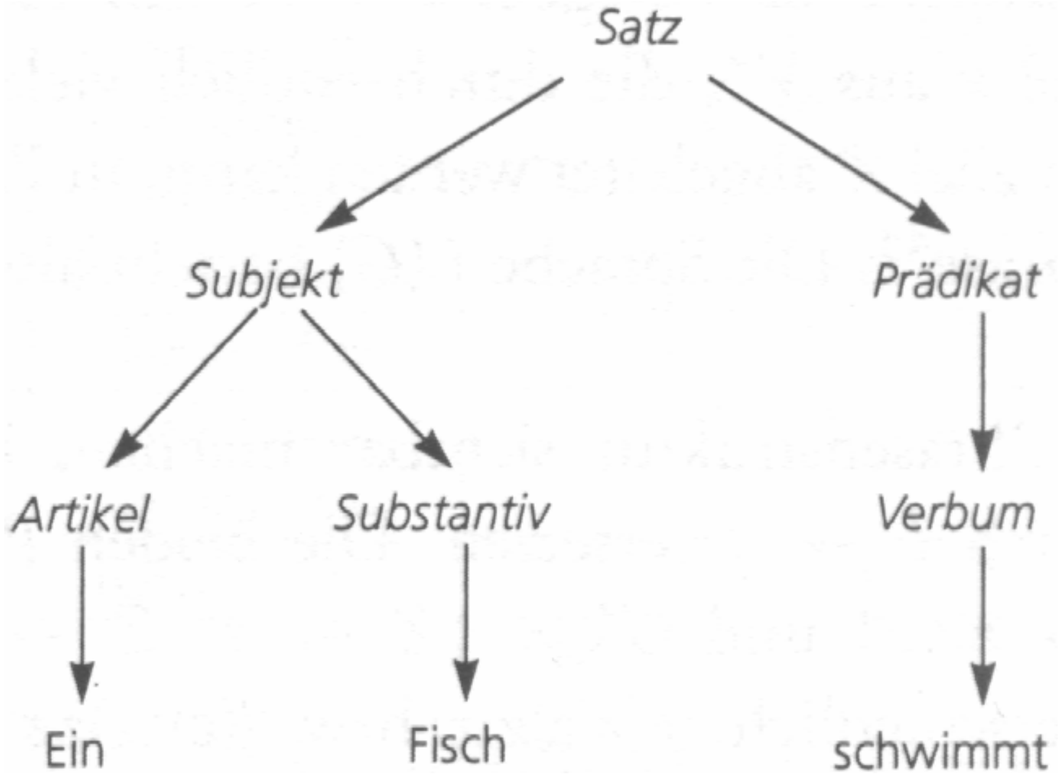
Noam Chomsky

Satz	→	*Subjekt Prädikat*
Subjekt	→	*Artikel Substantiv*
Subjekt	→	*Substantiv*
Prädikat	→	*Verbum*
Artikel	→	Ein
Substantiv	→	Fisch
Substantiv	→	Fische
Verbum	→	schwimmt
Verbum	→	schwimmen



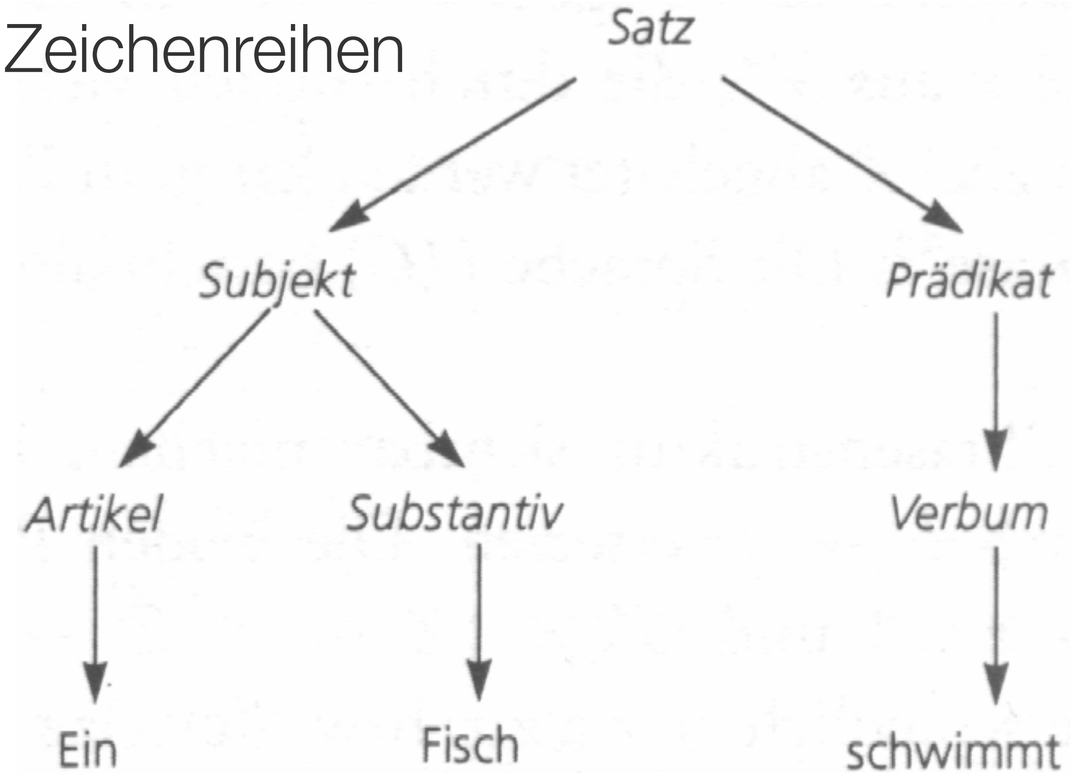
Chomsky-Grammatiken: Begriffe

- Eine Chomsky-Grammatik besteht aus **Terminalen**, **Nichtterminalen**, **Produktionen** und einem **Ziel**
- **Terminale**: Ein, Fisch, schwimmt, …
- **Nichtterminale**: *Satz*, *Subjekt*, *Prädikat*, ..
- **Startsymbol**/**Ziel**: *Satz*



Chomsky-Grammatiken: Begriffe

- **Vokabular**: Nichtterminale und Terminale
- **Sprachschatz**: Alle möglichen terminalen Zeichenreihen ohne Grammatik
- **Satzform**/**Phrase**:
  Ein Fisch schwimmt,
  Ein Fische schwimmt,
  Fisch schwimmt,
  Fische schwimmt …

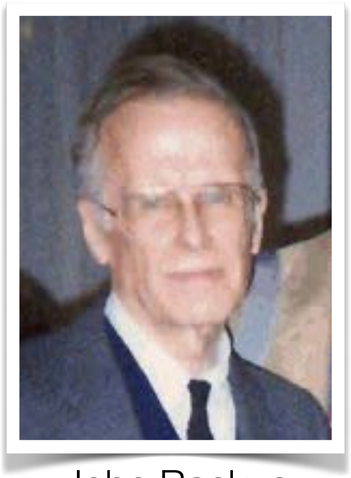


Universität Bremen

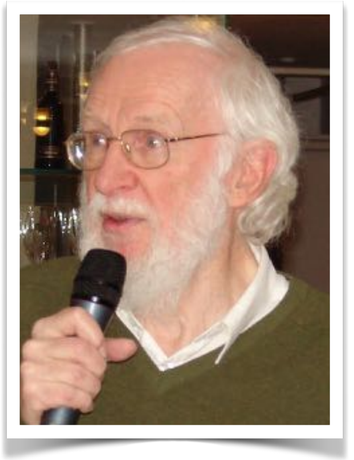
Chomsky-Grammatiken: Parsen

- Zerteilung (Parsing): Ist eine Zeichenreihe eine Phrase (ist sie **wohlgeformt**)?
- Umkehr des Ableitungssystems: Reduktions-/Zerteilungssystem
- Wie wird ein Text in linearer Zeit geparst?
  - Ohne Ausprobieren
- Es gibt **Top-Down-Parser** und **Bottom-Up-Parser**

Backus-Naur-Form (BNF)



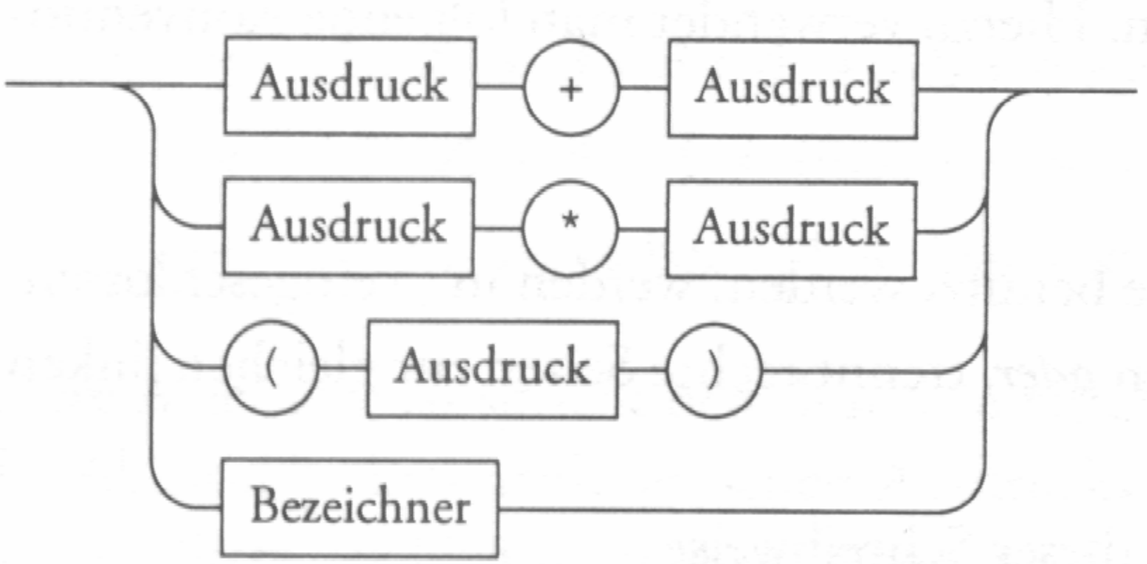
John Backus



Peter Naur


```
Ausdruck    ::= Ausdruck + Ausdruck
              | Ausdruck * Ausdruck
              | ( Ausdruck )
              | Bezeichner
Bezeichner  ::= a | b | ... | z
```

Grammatik



Syntaxdiagramm

Backus-Naur-Form: 1. Zerlegung

Ausdruck ::= (a + b) * c + d

Ausdruck ::= (a + b) * c

Ausdruck ::= d

Ausdruck ::= (a + b)

Ausdruck ::= c

Bezeichner ::= d

Ausdruck ::= a + b

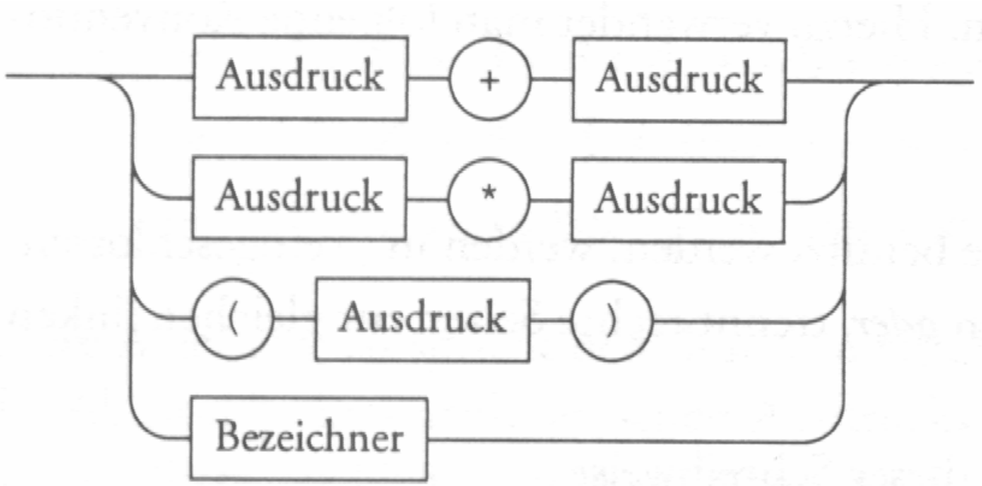
Bezeichner ::= c

Ausdruck ::= a

Ausdruck ::= b

Bezeichner ::= a

Bezeichner ::= b



Backus-Naur-Form: 2. Zerlegung

Ausdruck :: = (a + b) * c + d

Ausdruck ::= (a + b)

Ausdruck ::= a + b

Ausdruck ::= a

Bezeichner ::= a

Ausdruck ::= b

Bezeichner ::= b

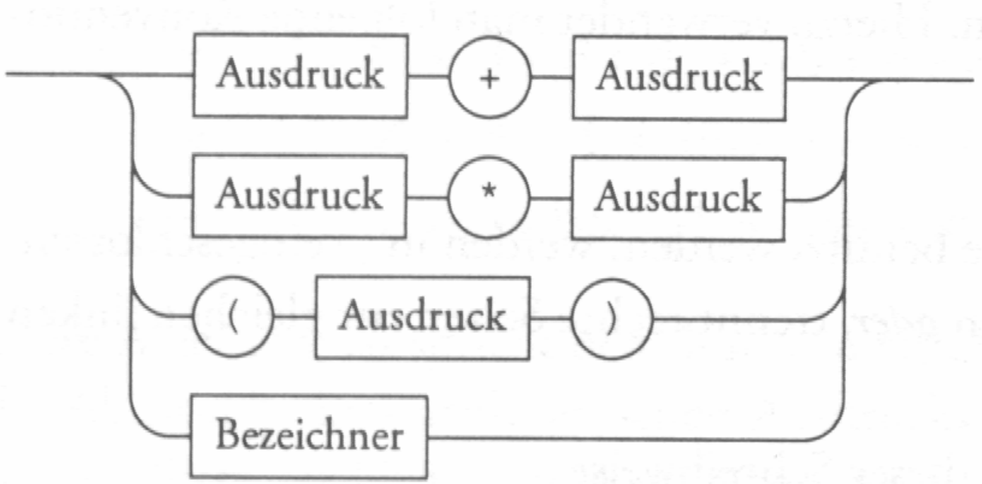
Ausdruck ::= c + d

Ausdruck ::= c

Bezeichner ::= c

Ausdruck ::= d

Bezeichner ::= d



Universität Bremen

Backus-Naur-Form: Eindeutige Grammatiken

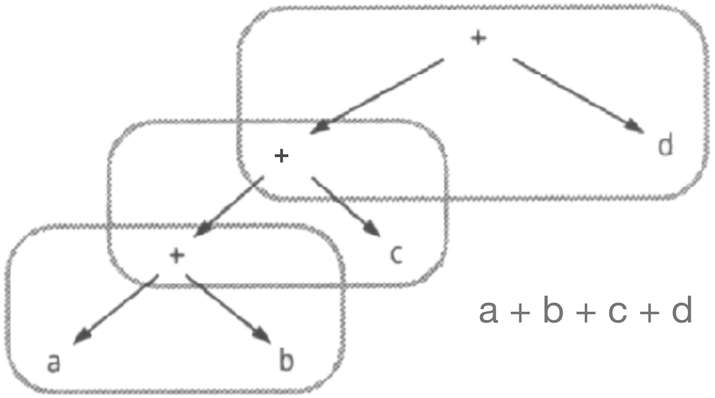
- Linksrekursiv

```
Ausdruck ::= Term | Ausdruck + Term
Term     ::= Faktor | Term * Faktor
Faktor   ::= ( Ausdruck ) | Bezeichner
```

- Rechtsrekursiv

```
Ausdruck ::= Term | Term + Ausdruck
Term     ::= Faktor | Faktor * Term
Faktor   ::= ( Ausdruck ) | Bezeichner
```

- Linksrekursive Grammatiken können nicht top-down geparst werden
  - Aber viele Ausdrücke sind linksrekursiv

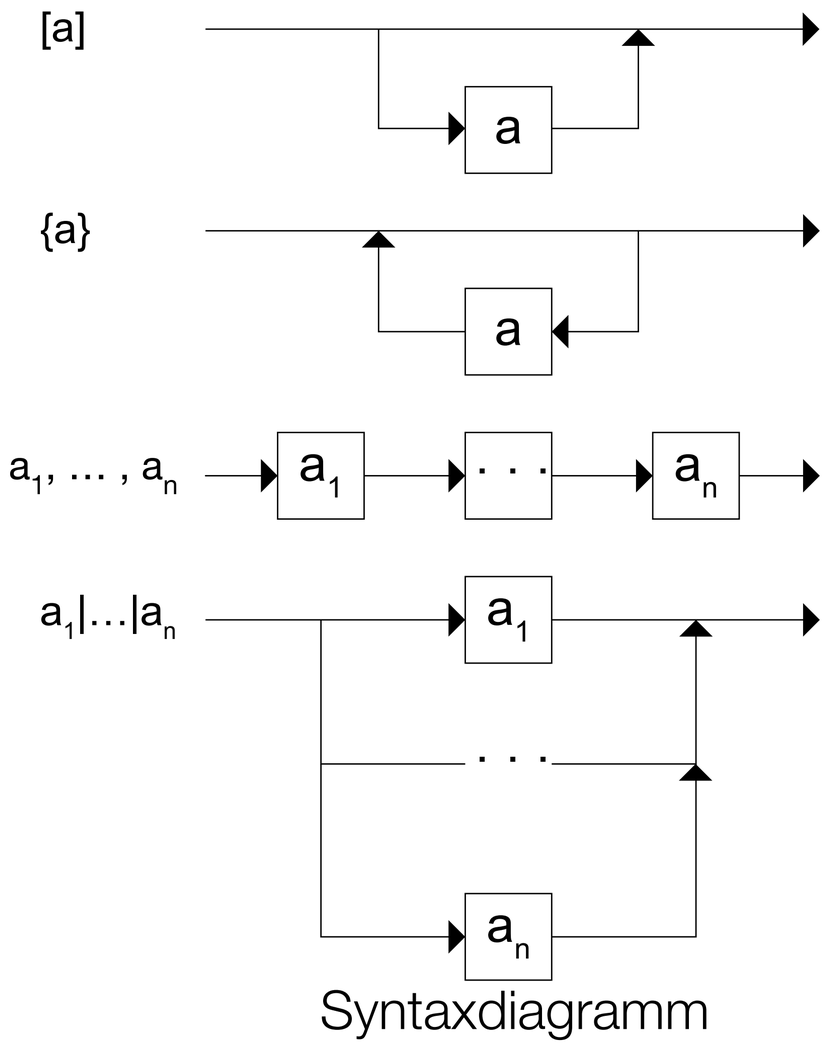


a + b + c + d

ISO/IEC 14977 : 1996(E)

EBNF: Erweiterte Backus-Naur-Form (unvollst.)

- **=** Definition einer Produktion
- **,** Verkettung von (Nicht-)Terminalen
- **;** Beendet Definition einer Produktion
- **'...'** Schließt Terminale ein (alternativ: **"..."**)
- **|** Trennt Alternativen (wie bei BNF)
- **(…)** umschließt Folge von (Nicht-)Terminalen
- **[…]** umschließt optionale Folge von (Nicht-)Terminalen
- **{…}** Inhalt der Klammer beliebig oft wiederholen (auch 0-mal)



Syntaxdiagramm

Universität Bremen

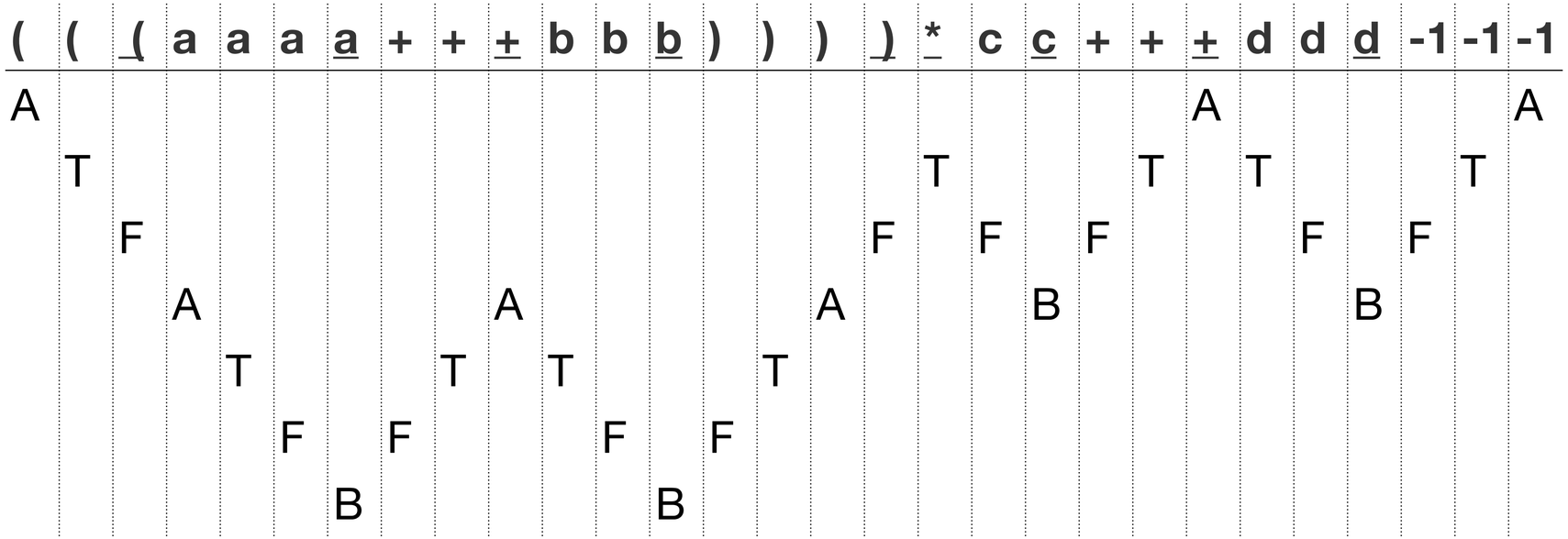
Erweiterte Backus-Naur-Form in EBNF (unvollst.)

```
Produktion    = Bezeichner , '=' , Ausdruck , ';' ;
Ausdruck      = Alternative , { '|' , Alternative } ;
Alternative   = Produkt , { ',' , Produkt } ;
Produkt       = Nichtterminal
                | "'" , Zeichen - "'" , { Zeichen - "'" } , "'"
                | '"' , Zeichen - '"' , { Zeichen - '"' } , '"'
                | '(' , Ausdruck , ')'
                | '[' , Ausdruck , ']'
                | '{' , Ausdruck , '}' ;
Nichtterminal = Buchstabe , { Buchstabe | Ziffer } ;
Ziffer        = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
                                        :
```

Zerlegung mit EBNF

- Ausdruck = **(a + b) * c + d**

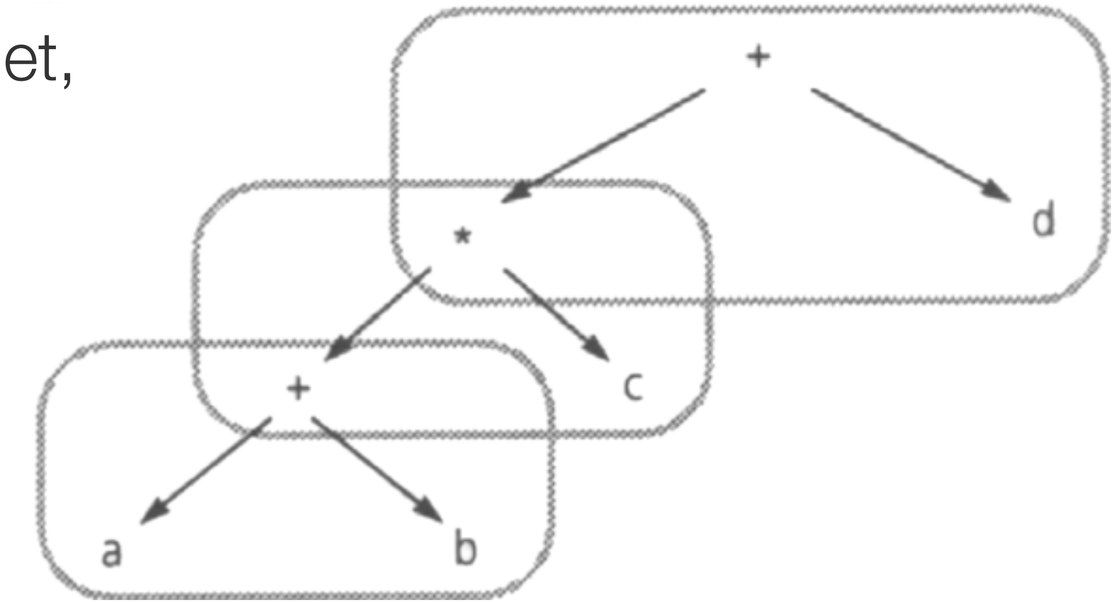
```
Ausdruck   = Term , { '+' , Term } ;
Term       = Faktor , { '*' , Faktor } ;
Faktor     = '(' , Ausdruck , ')'
           | Bezeichner ;
Bezeichner = 'a' | 'b' | … | 'z' ;
```



Universität Bremen

Rekursiv absteigender Parser

- Jede Produktion wird in eine Methode umgesetzt
- Methoden rufen sich gegenseitig auf
- Jeweils nächstes Zeichen/Token entscheidet, wie es weitergeht
- Normalerweise wird während des Parsens ein **Syntax-Baum** aufgebaut



Universität Bremen

Ausdrücke parsen und auswerten: Demo

- Natürliche Zahlen
- Rechenarten: **+**, **-**, ***, **/**, **%**
- Vorzeichen: **-**
- Klammern: **( )**
- Kein Leerraum erlaubt

```
-(4*8/15-(16+23))%42
```

Universität Bremen

Zusammenfassung der Konzepte

- **Syntax** und **Semantik**
- **Parsen** und **rekursiv absteigender Parser**
- **BNF** und **EBNF**

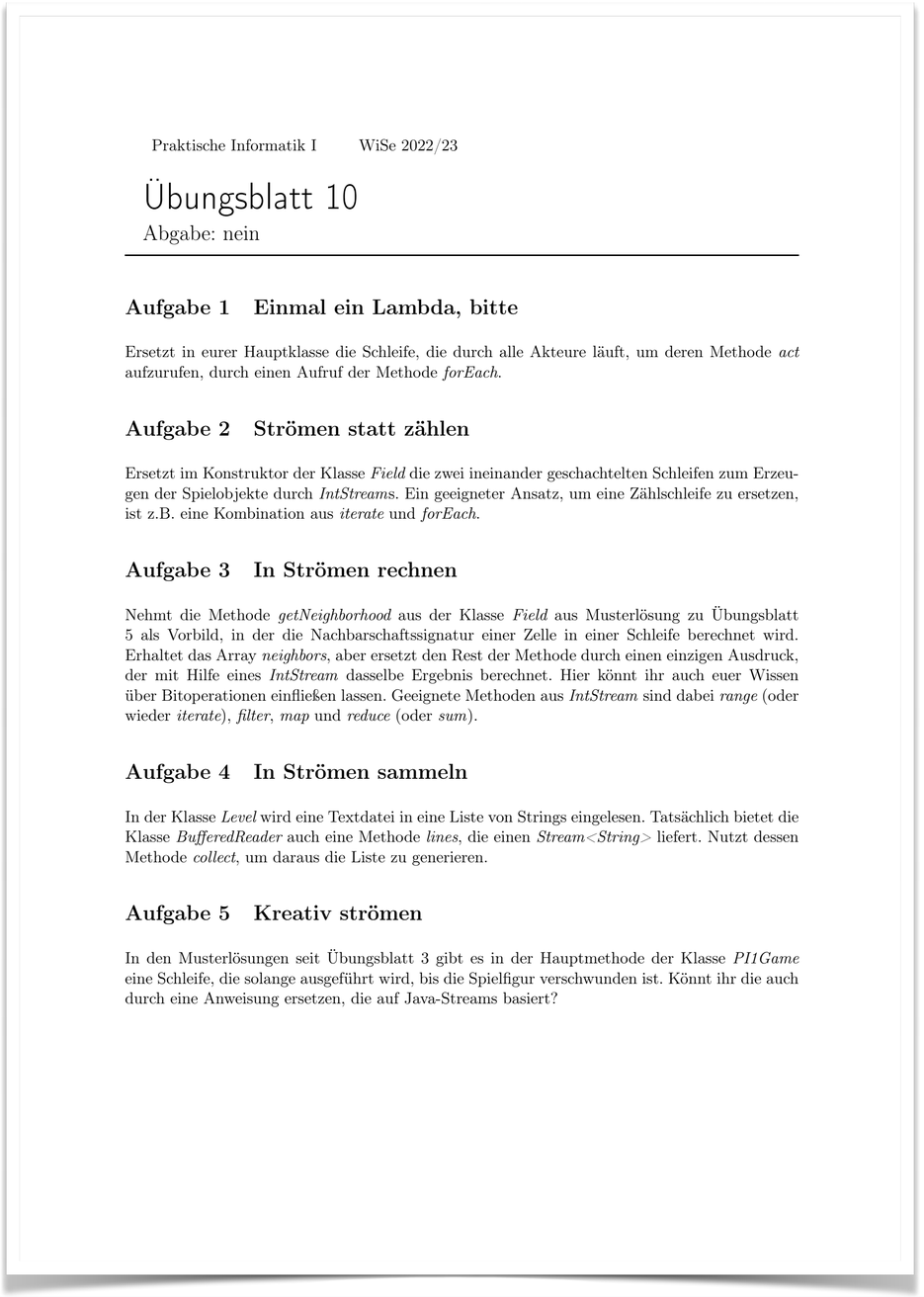
Google: „Java Grammar“

Vertiefung:
Vorlesung „Übersetzerbau“

Universität Bremen

Übungsblatt 10

- Aufgabe 1: Aufzählungsschleife ersetzen
- Aufgabe 2: Zählschleifen ersetzen
- Aufgabe 3: Nachbarschaftssignatur ausrechnen
- Aufgabe 4: Dateiinhalt einlesen
- Aufgabe 5: **while**-Schleife ersetzen



Praktische Informatik I WiSe 2022/23

Übungsblatt 10

Abgabe: nein

Aufgabe 1 Einmal ein Lambda, bitte

Ersetzt in eurer Hauptklasse die Schleife, die durch alle Akteure läuft, um deren Methode *act* aufzurufen, durch einen Aufruf der Methode *forEach*.

Aufgabe 2 Strömen statt zählen

Ersetzt im Konstruktor der Klasse *Field* die zwei ineinander geschachtelten Schleifen zum Erzeugen der Spielobjekte durch *IntStreams*. Ein geeigneter Ansatz, um eine Zählschleife zu ersetzen, ist z.B. eine Kombination aus *iterate* und *forEach*.

Aufgabe 3 In Strömen rechnen

Nehmt die Methode *getNeighborhood* aus der Klasse *Field* aus Musterlösung zu Übungsblatt 5 als Vorbild, in der die Nachbarschaftssignatur einer Zelle in einer Schleife berechnet wird. Erhaltet das Array *neighbors*, aber ersetzt den Rest der Methode durch einen einzigen Ausdruck, der mit Hilfe eines *IntStream* dasselbe Ergebnis berechnet. Hier könnt ihr auch euer Wissen über Bitoperationen einfließen lassen. Geeignete Methoden aus *IntStream* sind dabei *range* (oder wieder *iterate*), *filter*, *map* und *reduce* (oder *sum*).

Aufgabe 4 In Strömen sammeln

In der Klasse *Level* wird eine Textdatei in eine Liste von Strings eingelesen. Tatsächlich bietet die Klasse *BufferedReader* auch eine Methode *lines*, die einen *Stream<String>* liefert. Nutzt dessen Methode *collect*, um daraus die Liste zu generieren.

Aufgabe 5 Kreativ strömen

In den Musterlösungen seit Übungsblatt 3 gibt es in der Hauptmethode der Klasse *PI1Game* eine Schleife, die solange ausgeführt wird, bis die Spielfigur verschwunden ist. Könnt ihr die auch durch eine Anweisung ersetzen, die auf Java-Streams basiert?