Syntax von Programmiersprachen

Programmiersprachen

Sprache = Menge von Wörtern, typischerweise unendlich

Programmiersprache: Wörter repräsentieren *Programme*

Programm kann auf einem Computer (evtl. nur abstrakte Maschine) ausgeführt werden und induziert somit eine Berechnung

Bsp.:

- Javaprogramme werden in JVM ausgeführt
- arithmetische Ausdrücke im Taschenrechner evaluieren
- Tastenkombinationen zur Steuerung des Handys
- ...

Begriffsbildung

typisches Problem: gegeben Programm P, potentiell aus der Sprache L, führe dies aus

- Syntax: beschreibt die Programmiersprache; welche Wörter sind korrekte Programme, welche nicht
- Semantik: beschreibt die Bedeutung eines Programms; wie ist dieses auszuführen
- Interpreter: Programm, welches einen Computer steuert, um andere Programme auszuführen
- Compiler: Programm, welches Programme einer Sprache L in äquivalente Programme einer anderen Sprache L' (typischerweise Maschinensprache) übersetzt
- Lexer/Parser: Programme, die aus einem Wort, gegeben als String, eine baumartige Struktur bauen; werden benutzt, um die Struktur in Programmen zu erkennen

Syntax

Syntax von Programmiersprachen häufig gegeben durch kontext-freie Grammatik (evtl. mit Nebenbedingungen), z.B.

$$\begin{array}{rcl} \langle \mathit{Prg} \rangle & ::= & \ldots \mid \texttt{while} \texttt{ (} \langle \mathit{BExp} \rangle \texttt{) \{} \langle \mathit{Prg} \rangle \texttt{ \}} \mid \ldots \\ \langle \mathit{BExp} \rangle & ::= & \ldots \mid \langle \mathit{BExp} \rangle < \langle \mathit{BExp} \rangle \mid \ldots \end{array}$$

in diesem Beispiel wichtig:

- Teil, der aus $\langle \mathit{Prg} \rangle$ abgeleitet wird, ist ein *Programm*
- Teil, der aus $\langle \mathit{BExp} \rangle$ abgeleitet wird, ist ein *boolescher Ausdruck*

warum ist solch eine Unterscheidung wichtig?

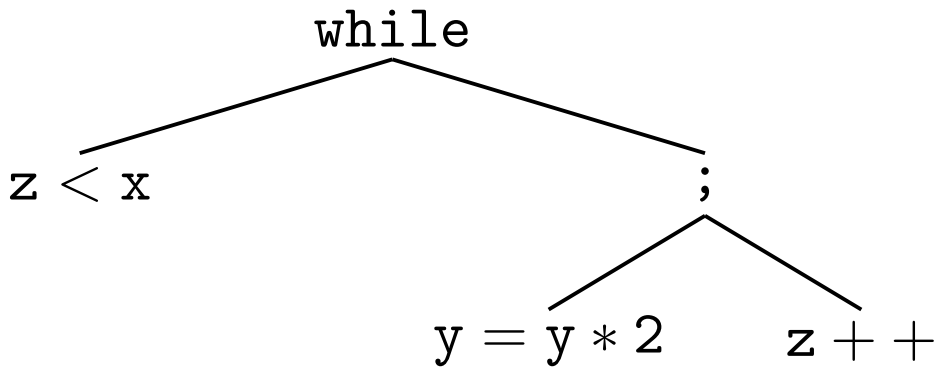
Semantik besagt sicherlich, dass boolesche Ausdrücke zu **true** oder **false** ausgewertet werden; Auswertung eines Programm ist jedoch anders

Notwendigkeit des Parsens

zum Verstehen und Ausführen des Programms

```
while (z < x) {
    y = y*2; z++
}
```

ist es notwendig, darin Struktur zu erkennen, z.B.



beachte:

- Links-Mitte-Rechts-Durchlauf des Baums liefert nicht originale Stringrepräsentation des Programms
- Ausführung des Programms abhängig von Art des Teilbaums (z.B. Programm vs. bool. Ausdruck)

Konkrete Syntax

konkrete Syntax einer Programmiersprache beschreibt genau die Zeichenketten, die gültige Programme beschreiben, z.B.

$$\begin{aligned} \langle Prg \rangle &::= \ldots \mid \langle Id \rangle = \langle Exp \rangle \\ \langle Id \rangle &::= \langle Alpha \rangle (\langle Alpha \rangle \cup \langle Num \rangle)^* \end{aligned}$$

beachte: in diesem Fall gar nicht so konkret, gewisser Grad an Abstraktion bereits vorhanden: Whitespaces evtl. implizit zugelassen

```
x3=x1        x3 = x1        x3 =
                                 x1
```

konkrete Syntax wird dazu verwendet, in Programmen Struktur zu erkennen, also für den *Parser*

Abstraktionsgrad bzgl. Whitespaces o.ä. wird vom *Lexer* gehandhabt

Abstrakte Syntax

konkrete Syntax stellt oft zuviel Information bereit, welche nur zum Erkennen der Programmstruktur notwendig ist, aber zu Redundanz in Ableitungsbäumen führt (Beispiel folgt)

abstrakte Syntax beschreibt ebenfalls Prgrammstruktur, aber auf abstrakterer Ebene, z.B.

$$\langle Prg \rangle \quad ::= \quad \ldots \mid Id{=}\langle Exp \rangle$$

hier also z.B. Bezeichner als Terminale aufgefasst, die zusätzlich einen Namen haben, z.B. $Id(x3)$

Grund: für Auswertung von Programmen oder Ausdrücken ist lexikalische Struktur von Bezeichnern uninteressant, wichtig ist nur zu wissen, ob es ein Bezeichner oder Ausdruck oder Konstante etc. ist

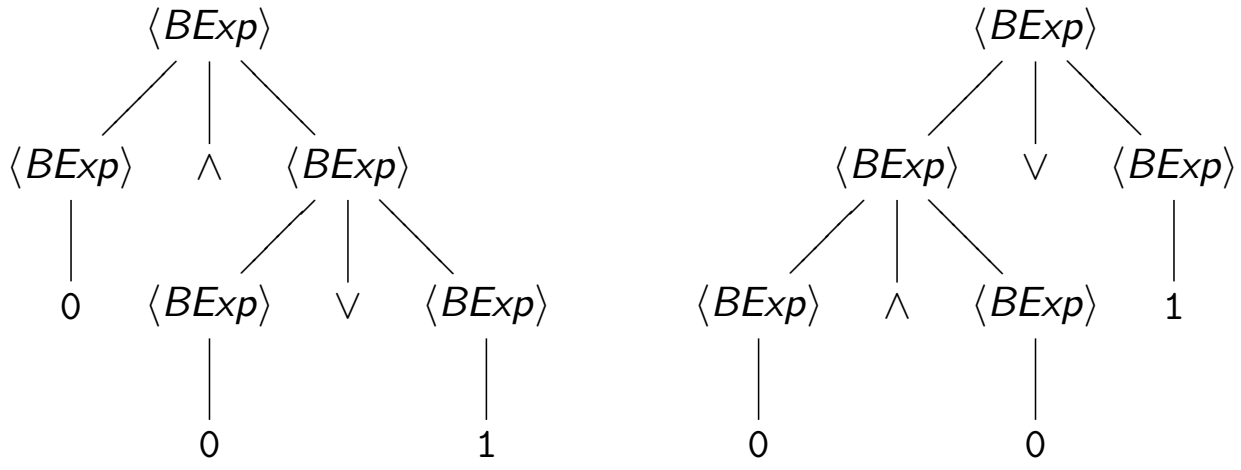
Konkrete vs. abstrakte Syntax

Bsp.: einfache Sprache boolescher Ausdrücke über 0,1 mit $\vee$, $\wedge$ mit üblicher Auswertung

$$\langle BExp \rangle \quad ::= 0 \mid 1 \mid \langle BExp \rangle \vee \langle BExp \rangle \mid \langle BExp \rangle \wedge \langle BExp \rangle$$

Ausdruck $0 \wedge 0 \vee 1$ is gültig bzgl. dieser Grammatik, aber was ist sein Wert?

Grammatik ist *mehrdeutig*



Mehrdeutigkeiten

Grammatik ist mehrdeutig, wenn es ableitbare Wörter mit mehr als einem Ableitungsbaum gibt

$\rightsquigarrow$ keine eindeutige Struktur; Auswerten nicht eindeutig

Mehrdeutigkeiten können oft durch *Präzedenzregeln* aufgelöst werden; hier: "$\wedge$ bindet stärker als $\vee$"

Umgang mit Präzedenzen:

- Parser so programmieren, dass er sie berücksichtigt, oder
- in Grammatik einbauen

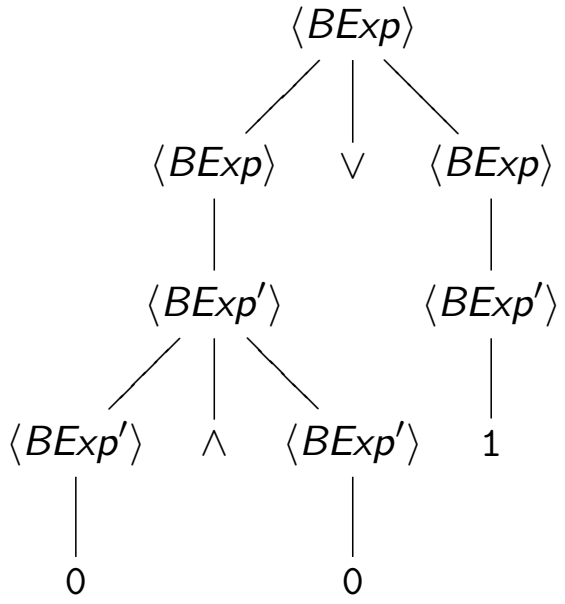
Mehrdeutigkeiten eliminieren

hier z.B.

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

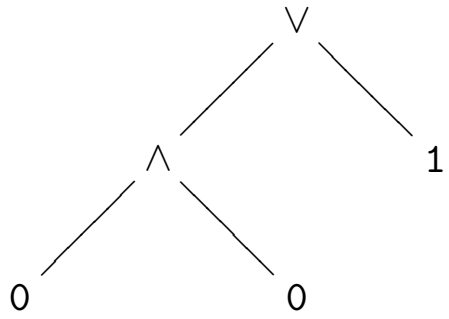
nur noch ein Ableitungsbaum für besagtes Wort mit richtiger Präzedenz



Vorteile abstrakter Syntax

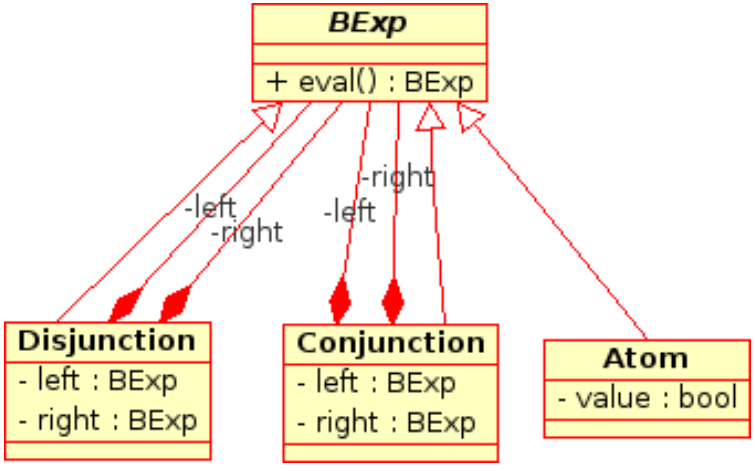
obiger Ableitungsbaum ist unnötig gross und Auswertung von $\langle BExp \rangle$ und $\langle BExp' \rangle$ unterscheidet sich nicht wesentlich

Struktur und nötige Information kann auch so gespeichert werden



hier benötigt: innere Knoten sind Operatoren, Blätter sind Konstanten

leicht abzubilden in Klassendiagramm



Vorgehensweise

hier: Programmiersprache AntBrain gegeben

- konkrete Syntax durch kontext-freie Grammatik
- abstrakte Syntax durch UML-Klassendiagramm

was ist zu tun?

1. aus Grammatik extrahieren: was sind die wichtigen Teile einer Eingabe? Nicht einzelne Zeichen, sondern Schlüsselwörter, Bezeichner, Konstanten, Befehle, . . . , genannt *Tokens*
   Lexer schreiben, der Zeichenkette in Liste von Tokens zerlegt / übersetzt
2. aus konkreter Syntax einen Parser bauen, der Listen von Tokens in Ableitungsbäume umwandelt; diese möglichst gleich als baumförmige Objektstrukturen der abstrakten Syntax erzeugen

Beispiel

gesamtes Vorgehen beispielhaft gezeigt anhand einfacher Sprache arithmetischer Ausdrücke

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp} \rangle + \langle \mathit{IExp} \rangle \mid \langle \mathit{IExp} \rangle * \langle \mathit{IExp} \rangle \mid (\langle \mathit{IExp} \rangle) \mid \\
&\quad \langle \mathit{Num} \rangle \mid \langle \mathit{Id} \rangle \mid \texttt{rand} \\
\langle \mathit{Num} \rangle &::= \langle \mathit{Digit} \rangle^{+} \\
\langle \mathit{Digit} \rangle &::= \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \mathit{Id} \rangle &::= \langle \mathit{Alpha} \rangle \big(\langle \mathit{Alpha} \rangle \cup \langle \mathit{Digit} \rangle \big)^{*} \\
\langle \mathit{Alpha} \rangle &::= \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}
\end{aligned}
$$

beachte: Grammatik ist mehrdeutig

Beispiel mit eindeutiger Grammatik

Präzedenzregel in Grammatik einbauen und zusätzlich Ende der Eingabe in Grammatik verlangen, damit der Parser später vollständige Eingaben verarbeitet

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \Big(+ \langle \mathit{IExp}_2 \rangle \Big)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \Big(* \langle \mathit{IExp}_3 \rangle \Big)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IExpAtom} \rangle \\
\langle \mathit{IExpAtom} \rangle &::= \langle \mathit{Num} \rangle \mid \langle \mathit{Id} \rangle \mid \texttt{rand} \\
&\vdots
\end{aligned}
$$

(in abstrakter Syntax würde man $\langle \mathit{Num} \rangle$ und $\langle \mathit{Id} \rangle$ als Tokens betrachten)

Token

sinnvolle Gruppierung von Teilen einer Zeichenkette in Tokens:
`EOF`, `Identifier`, `Keyword`, `Number`, `Symbol`

einige Tokens haben Werte:

- ein String bei `Identifier` – der Name
- ein Integer bei `Number` – sein Wert
- ein Name bei `Keyword` – um welches es sich handelt; hier gibt es jedoch lediglich `rand`
- ein Name bei `Symbol` – es gibt die Symbole `(`,`)`,`+`,`*`, die gut als Aufzählungstyp modelliert werden können

Lexer

nimmt Zeichenkette und wandelt diese in Liste von Tokens um, z.B.

```
47 + x7 + rand) * (016 +)rand34
```

wird zu

Number(47), Symbol("+"), Identifier("x7"), Symbol("+"), Keyword(rand), Symbol(")"), Symbol("*"), Symbol("("), Number(16), Symbol("+"), Symbol(")"), Identifier("rand34")

beachte:

- Lexer überprüft nicht, ob Zeichenkette herleitbar ist (Parser)
- Lexing kann auch fehlschlagen, hier z.B. bei `x - 3`
- Java-Klasse `Scanner` hier nicht geeignet, um Lexer zu konstruieren

Parser

zur Erinnerung: Parser erhält Liste von Tokens und konstuiert daraus Ableitungsbaum bzgl. (abstrakter) Syntax, der diese Liste als Blätter und Startsymbol der Grammatik als Wurzel enthält

es gibt allgemeines Verfahren zum Parsen kontext-freier Grammatiken

- läuft in Zeit $\mathcal{O}(n^3)$ bei Eingabelänge n
- konstruiert endliche Repräsentation aller Ableitungsbäume

beides ist nicht von Vorteil (bei eindeutigen Grammatiken)

Ausweg: Unterklassen der kontext-freien Sprachen, die einfacheres und damit evtl. auch schnelleres (z.B. in $\mathcal{O}(n)$) Parsen erlauben

Bottom-Up vs. Top-Down Parsen

im Prinzip gibt es zwei Arten von Parsern

- Bottom-Up
  - finde Teil der Eingabe der Form $B_1 \ldots B_k$, sodass es Regel $A ::= B_1 \ldots B_k$ gibt; ersetze diesen Teil durch A
  - endet, wenn gesamte Eingabe durch Startsymbol S ersetzt wurde
  - Problem: normalerweise keine eindeutige Ersetzung möglich
- Top-Down
  - interessant ist nur, ob Eingabe $t_1 \ldots t_n$ aus Startsymbol S herleitbar ist
  - beginnt sozusagen an der Wurzel S
  - versucht, sukzessive Teilbäume darunter durch Anwenden einer Regel zu konstruieren
  - Problem: normalerweise stehen mehrere Regeln zur Auswahl
  - endet, wenn Blätterfront der Eingabeliste entspricht

Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid \mathtt{0} \mid \mathtt{1}$$

Parsen:

$$0 \qquad \wedge \qquad 0 \qquad \vee \qquad 1$$

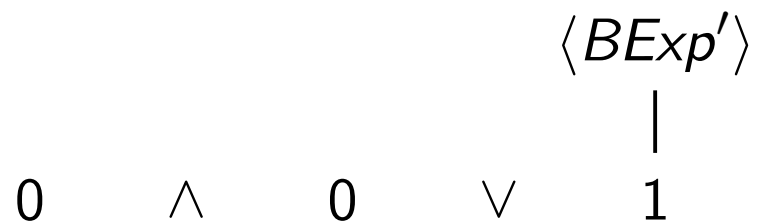
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:

$$\begin{array}{ccccccc} \langle BExp' \rangle & & & & & & \langle BExp' \rangle \\ | & & & & & & | \\ 0 & \wedge & 0 & \vee & 1 \end{array}$$

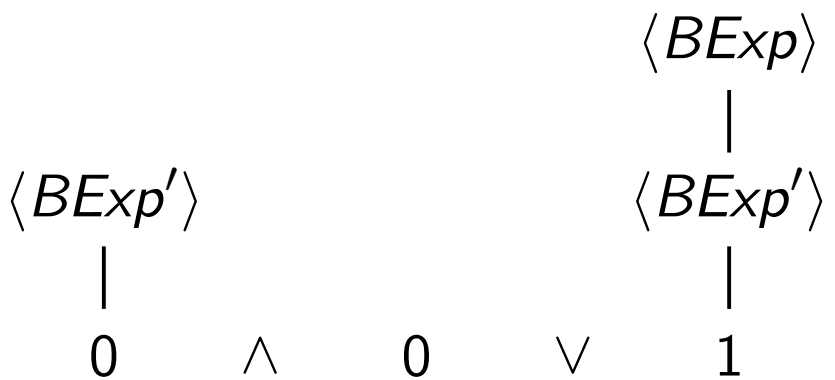
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



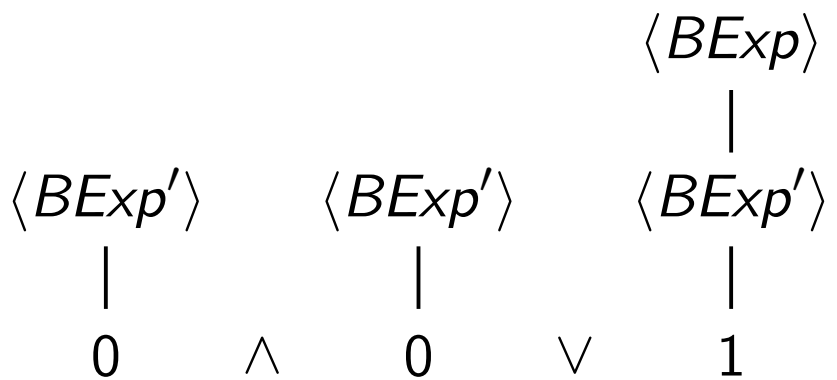
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



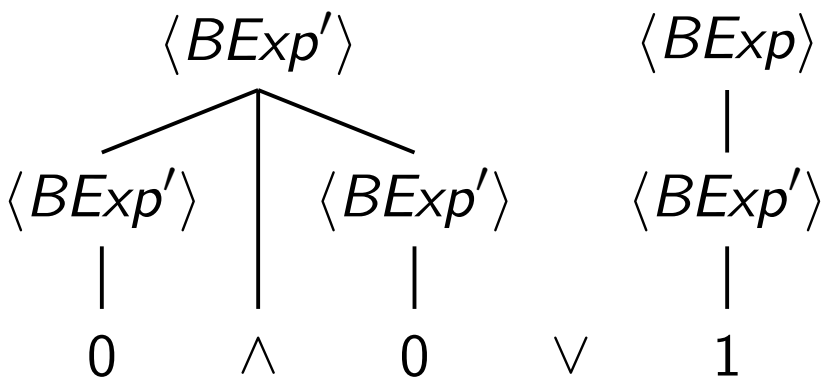
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



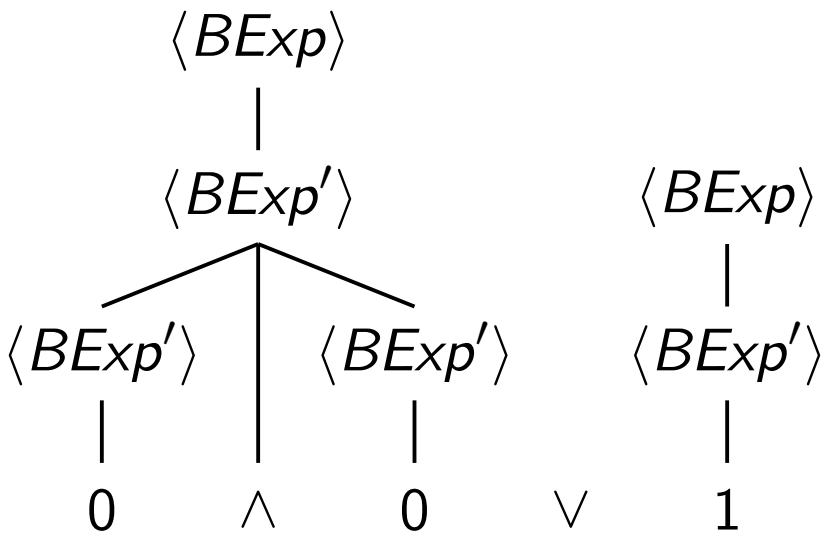
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



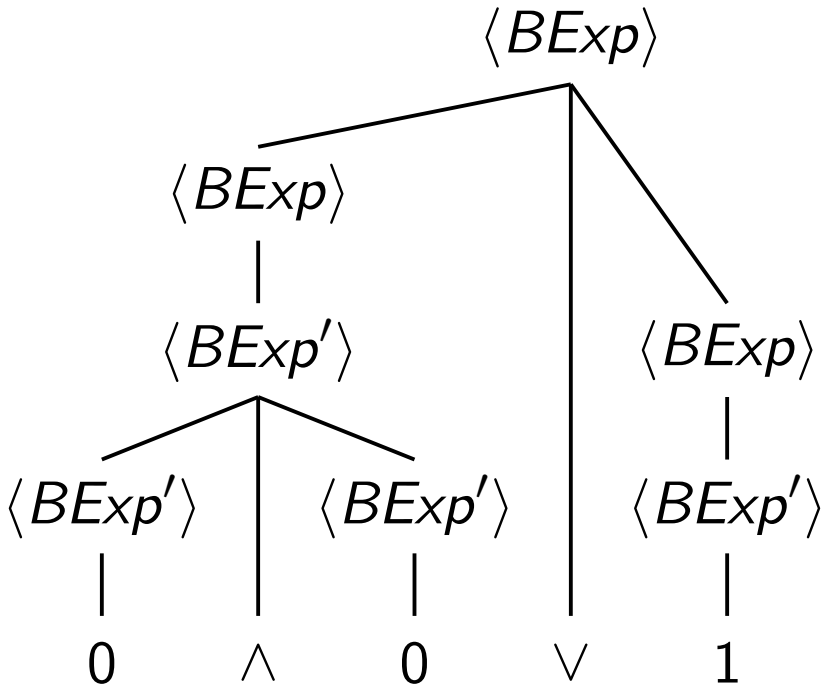
Beispiel Bottom-Up Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



Top-Down Parser

allgemeine Funktionsweise:

`Parse`(Nonterminal A, TokenList $[t_1, \ldots, t_n]$)

wähle Regel $A ::= B_1 \ldots B_m$
wähle $i_1 \leq i_2 \leq \ldots i_m \leq i_{m+1}$ mit $i_1 = 1$, $i_{m+1} = n + 1$
`for` $j = 1, \ldots, m$ `do`
`Parse`(B_i, $[t_{i_j}, \ldots, t_{i_{j+1}-1}]$)

hier zur Vereinfachung nur ein *Recognizer*, kein *Parser*

muss also noch beim Rücksprung konstruierten Ableitungsbaum liefern

Verfahren ist nichtdeterministisch, braucht also noch Strategien, um Wählen deterministisch zu machen; hängt von Sprachklasse ab

Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:

$$\langle BExp \rangle$$

$$0 \qquad \wedge \qquad 0 \qquad \vee \qquad 1$$

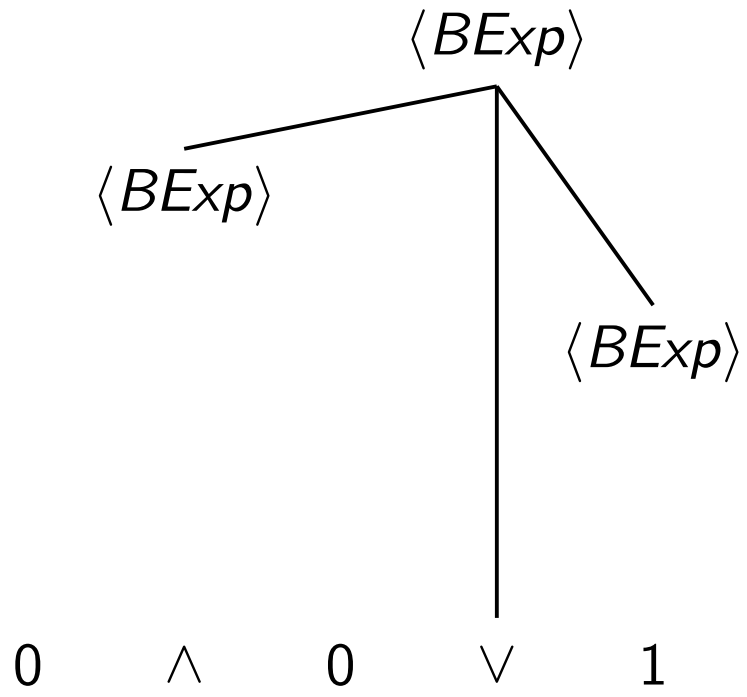
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



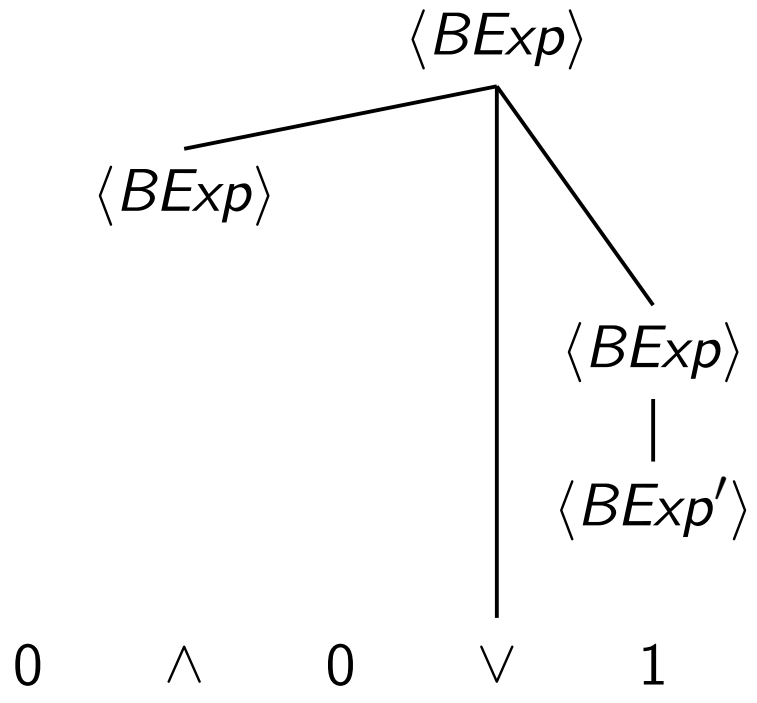
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



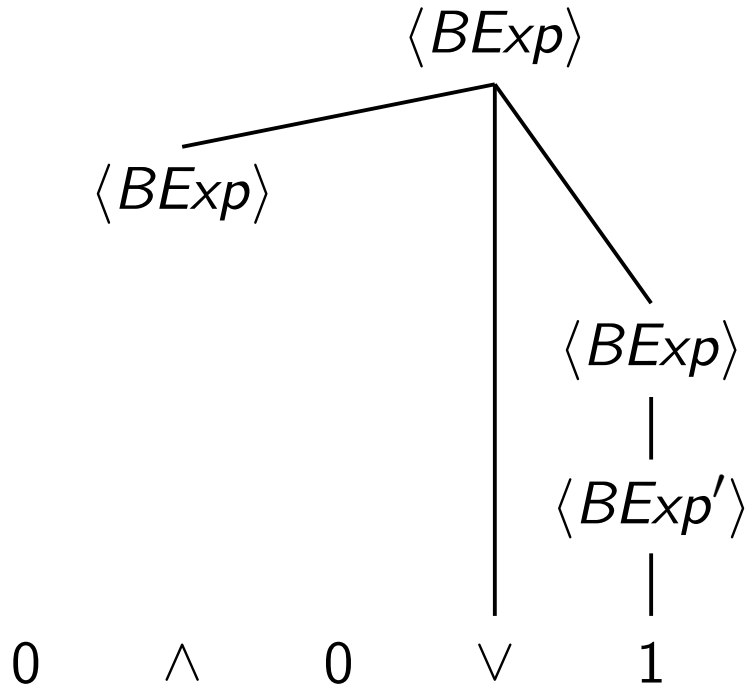
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



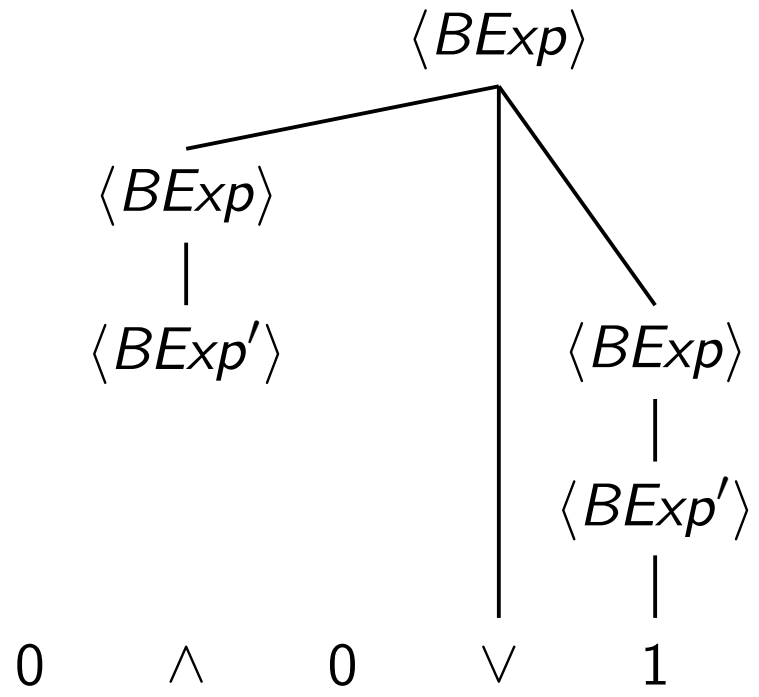
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



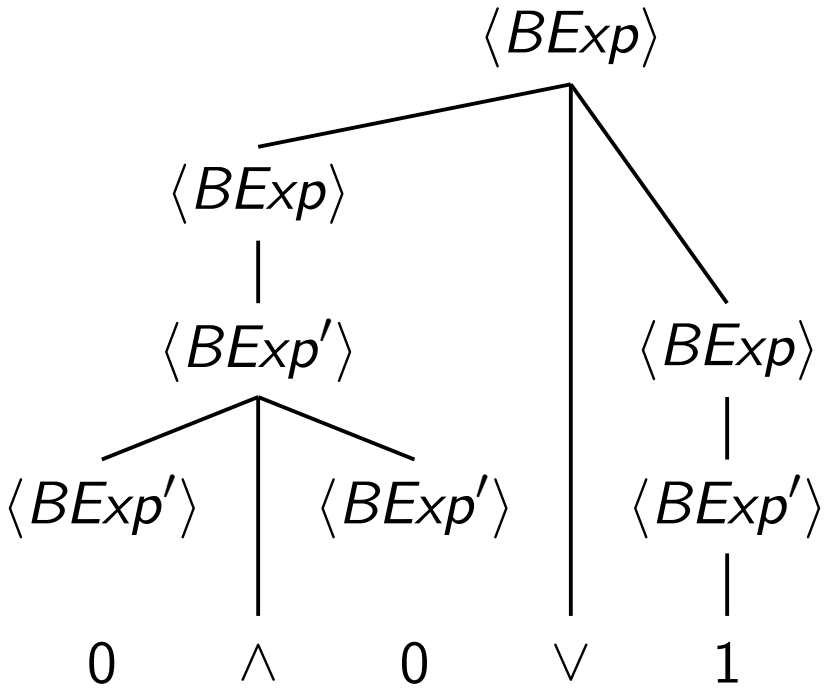
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \quad ::= \quad \langle BExp \rangle \left(\ \vee \ \langle BExp \rangle \ \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \quad ::= \quad \langle BExp' \rangle \left(\ \wedge \ \langle BExp' \rangle \ \right)^* \mid (\ \langle BExp \rangle \ ) \mid 0 \mid 1$$

Parsen:



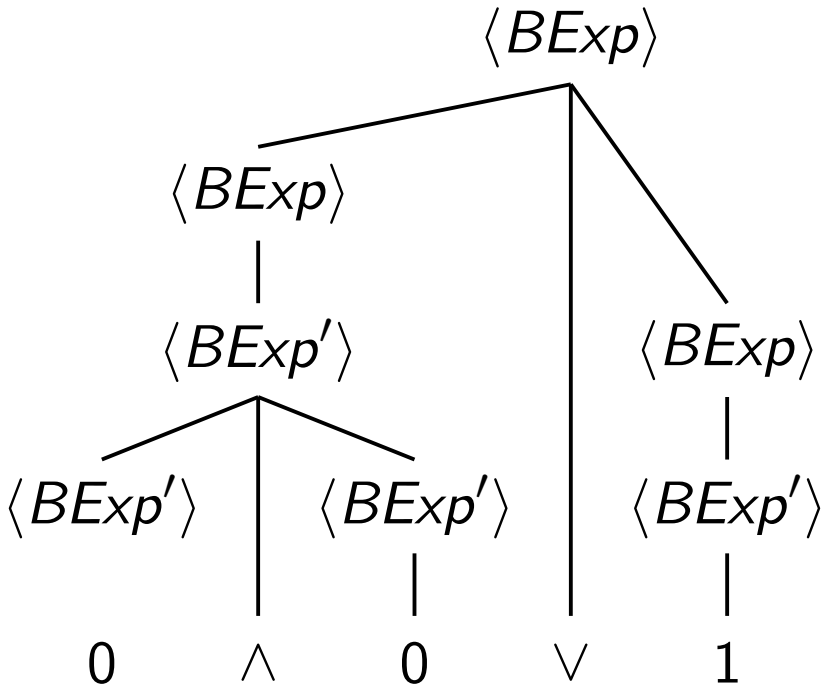
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



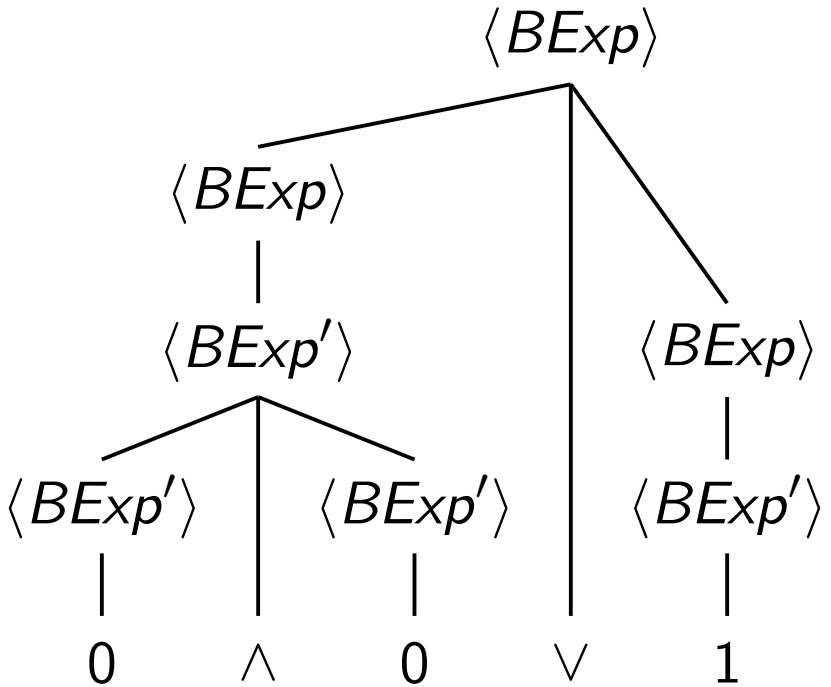
Beispiel Top-Down Parsen

Grammatik:

$$\langle BExp \rangle \;::=\; \langle BExp \rangle \left(\;\vee\; \langle BExp \rangle \right)^* \mid \langle BExp' \rangle$$

$$\langle BExp' \rangle \;::=\; \langle BExp' \rangle \left(\;\wedge\; \langle BExp' \rangle \right)^* \mid (\; \langle BExp \rangle \;) \mid 0 \mid 1$$

Parsen:



LL(k)-Sprachen

größeres Problem beim top-down Parsen: Wahl der Regel

LL(k)-Sprache ist Sprache, für die es eine LL(k)-Grammatik gibt

Def.: Sei G Grammatik über Alphabet Σ, α Satzform, $k \in \mathbb{N}$

$$first_k(\alpha) \;=\; \{v \in \Sigma^k \mid \exists \beta, \alpha \Rightarrow^* v\beta\}$$

Def.: G ist LL(k)-Grammatik, wenn für alle Nonterminals A und alle Regeln $A ::= \alpha$, $A ::= \beta$ mit $\alpha \neq \beta$ und alle Satzformen γ und Wörter v mit $S \Rightarrow^* vA\gamma$:

$$first_k(\alpha\gamma) \cap first_k(\beta\gamma) = \emptyset$$

soll heißen: bei einer LL(k)-Sprache erkennt man anhand von höchstens k der nächsten Tokens, welche Regel anzuwenden ist

LL(k)-Parsen

allgemeine Vorgehensweise:

`Parse`(Nonterminal A, TokenList $[t_1, \ldots, t_n]$)

wähle anhand von $t_1, \ldots, t_k$ eindeutige Regel $A ::= B_1 \ldots B_m$
wähle $i_1 \leq i_2 \leq \ldots i_m \leq i_{m+1}$ mit $i_1 = 1$, $i_{m+1} = n + 1$
`for` $j = 1, \ldots, m$ `do`
`Parse`(B_i, $[t_{i_j}, \ldots, t_{i_{j+1}-1}]$)

noch zu eliminieren: Nichtdeterminismus bzgl. Wahl der Zerlegung

- finde Zerlegung beim Parsen selbst
- parse erst $[t_1, \ldots, t_n]$ bzgl. B_1
- dies reduziert dann $[t_1, \ldots, t_{i-1}]$ für ein i zu einem Ableitungsbaum mit Wurzel B_1
- parse dann $[t_i, \ldots, t_n]$ bzgl. B_2, etc.

Beispiel

dies ist eine LL(1)-Grammatik

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IExpAtom} \rangle \\
\langle \mathit{IExpAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

betrachte Regeln für $\langle \mathit{IExp}_1 \rangle$ und $\langle \mathit{IExp}_2 \rangle$ z.B. als Abkürzung für

$$
\begin{aligned}
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \langle \mathit{IExpSeq}_1 \rangle \\
\langle \mathit{IExpSeq}_1 \rangle &::= + \langle \mathit{IExp}_2 \rangle \ \langle \mathit{IExpSeq}_1 \rangle \mid \epsilon
\end{aligned}
$$

beachte: Kleene-Sterne hier immer zu lesen als “soviel wie möglich mit dieser Regel parsen”

Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$

`(` `3` $+$ `rand` `)` $*$ `4` `EOF`

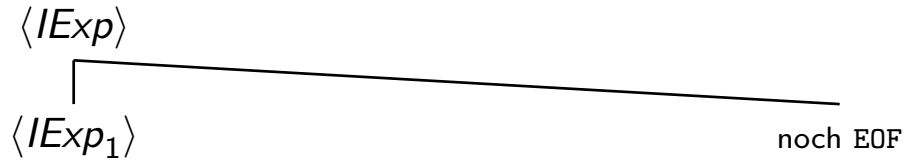
Beispiel LL(1)-Parsen

$\langle IExp \rangle$

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$

(3 + rand) * 4 EOF

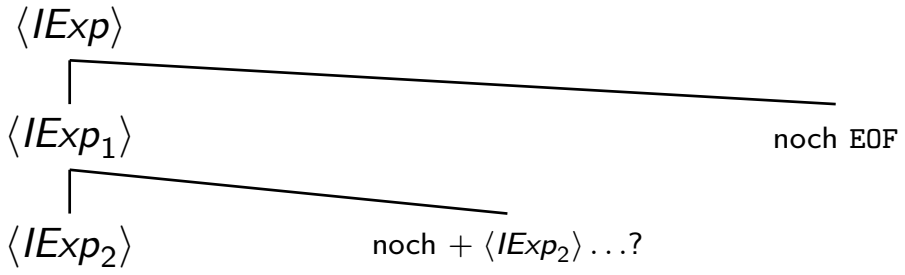
Beispiel LL(1)-Parsen



$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

`(  3  +  rand  )  *  4  EOF`

Beispiel LL(1)-Parsen

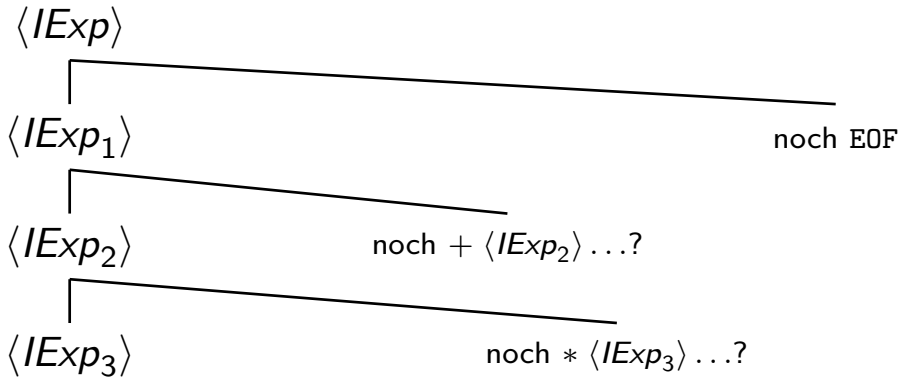


$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= \texttt{(} \ \langle \mathit{IExp}_1 \rangle \ \texttt{)} \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$

`(` `3` `+` `rand` `)` `*` `4` `EOF`

Beispiel LL(1)-Parsen

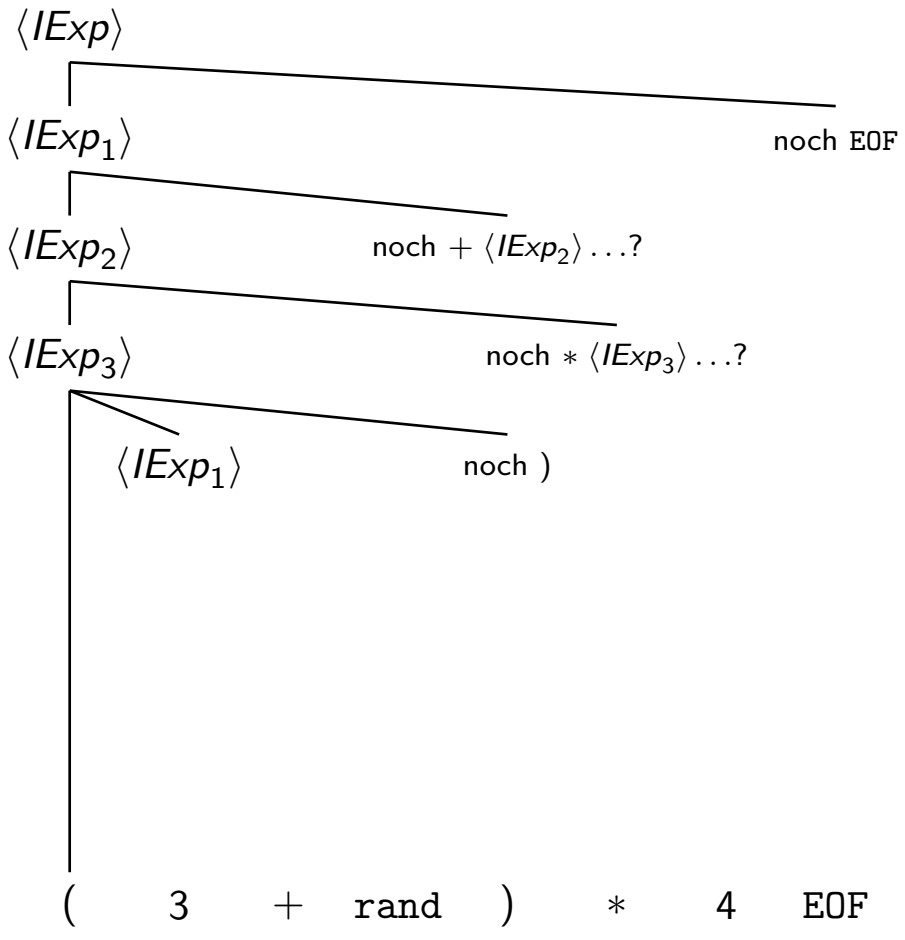
$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



`(` `3` `+` `rand` `)` `*` `4` `EOF`

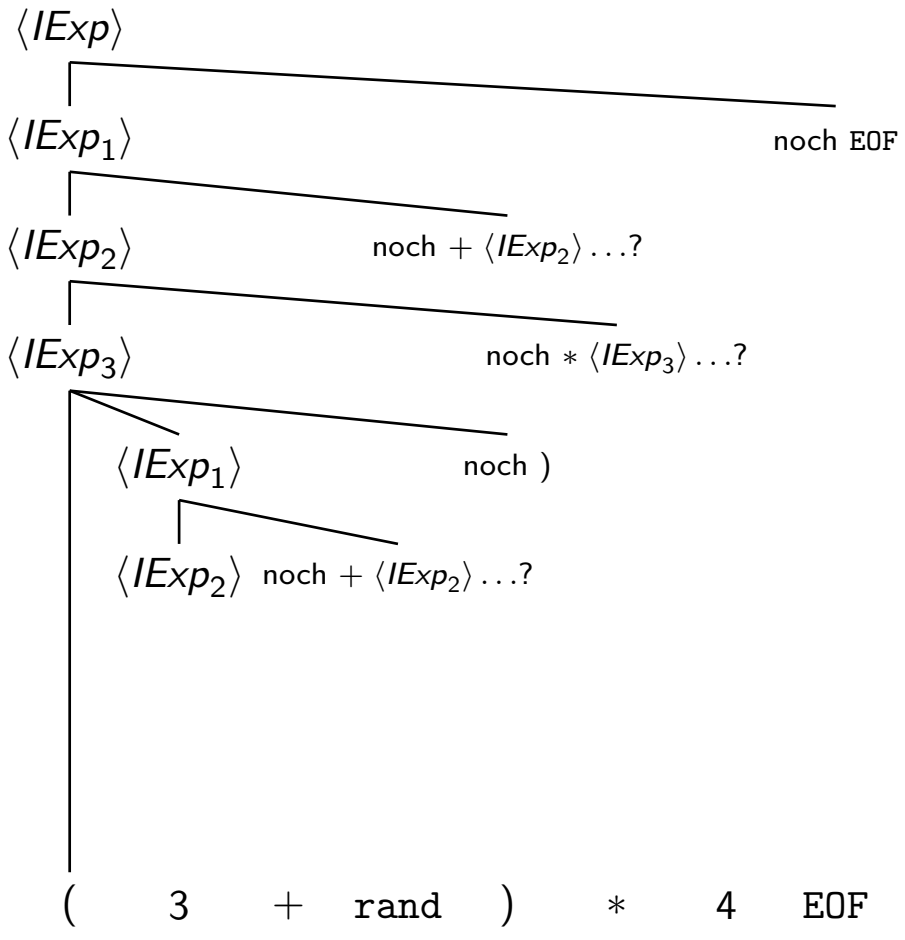
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



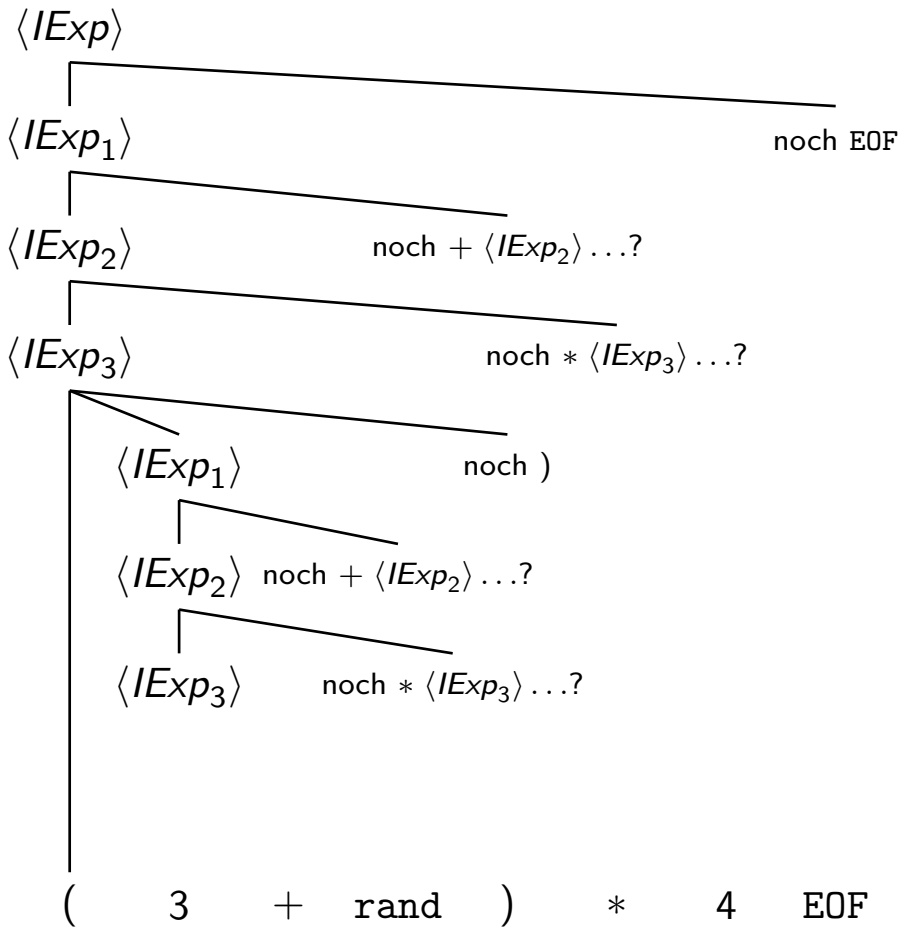
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



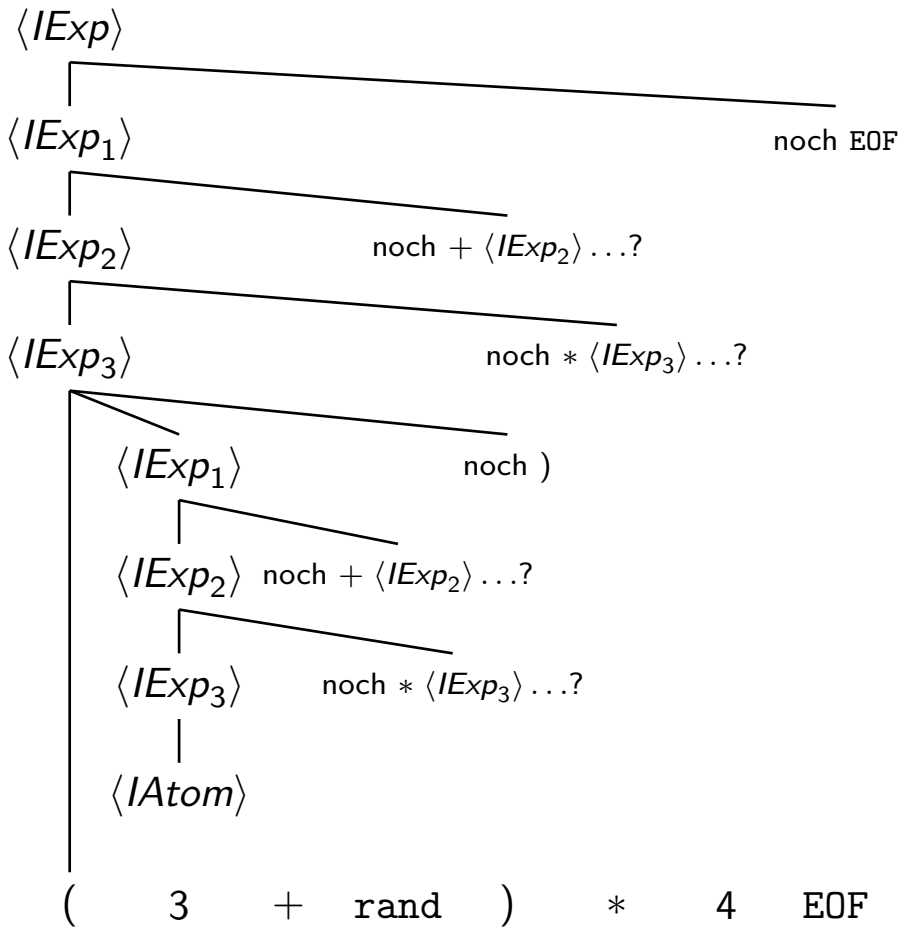
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



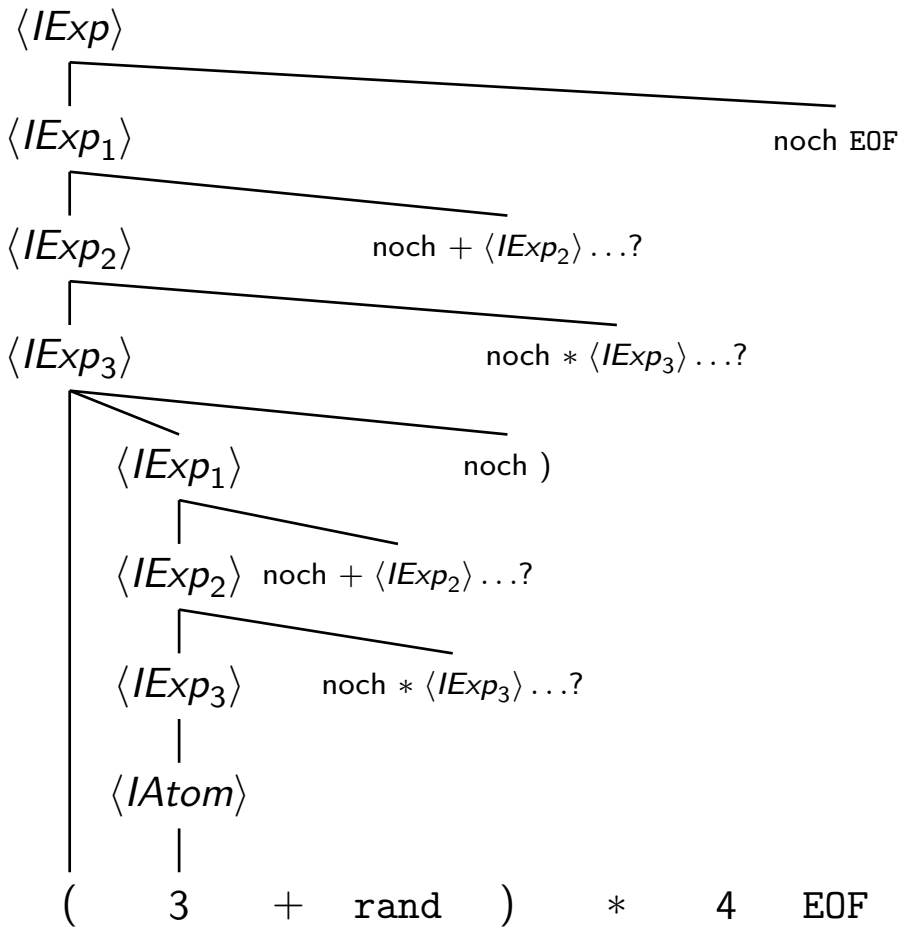
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



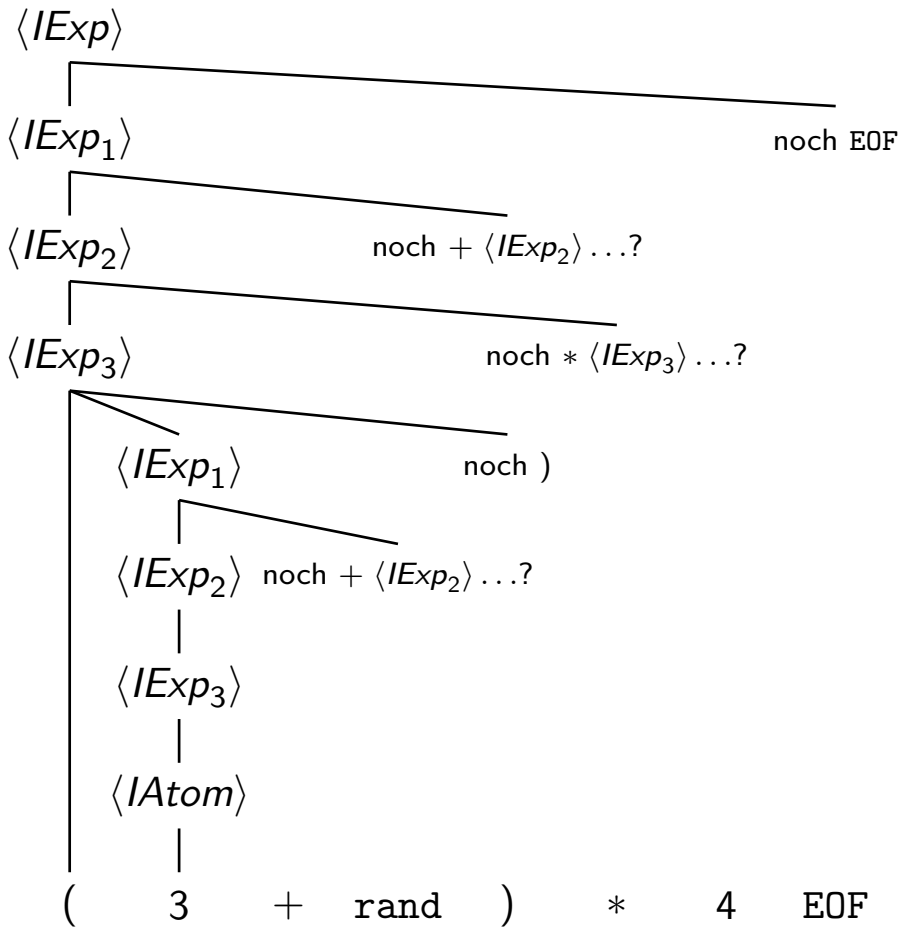
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



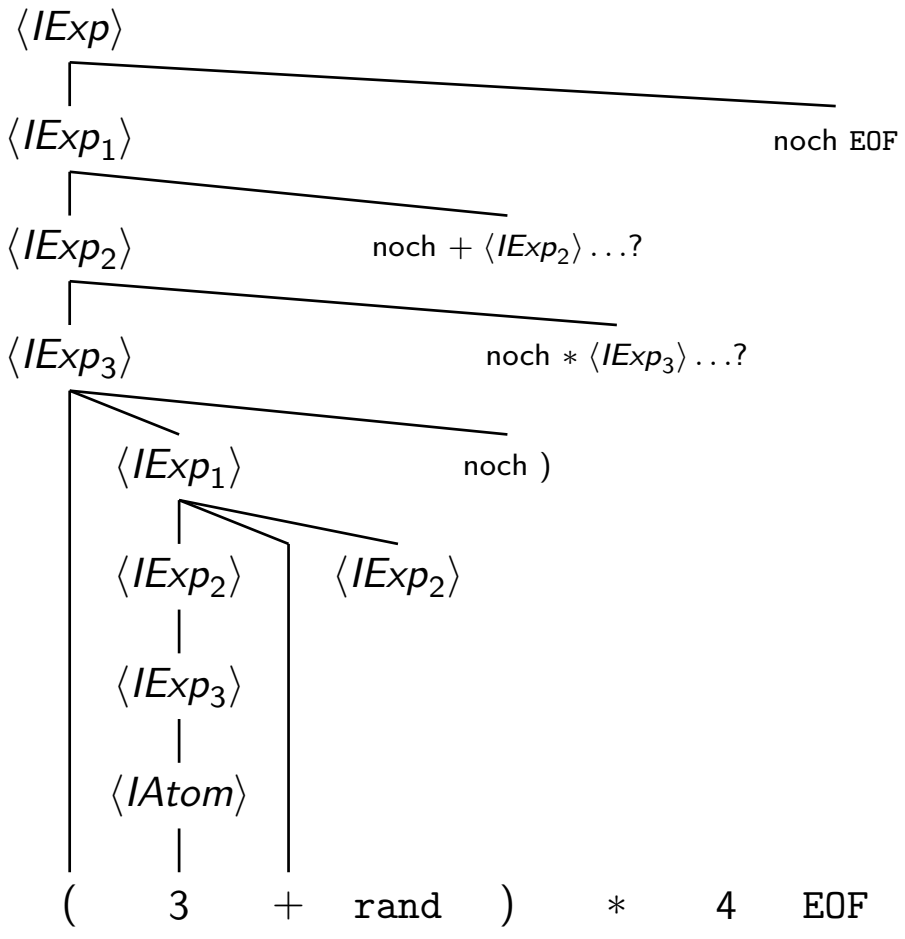
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\langle IExp_1 \rangle) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



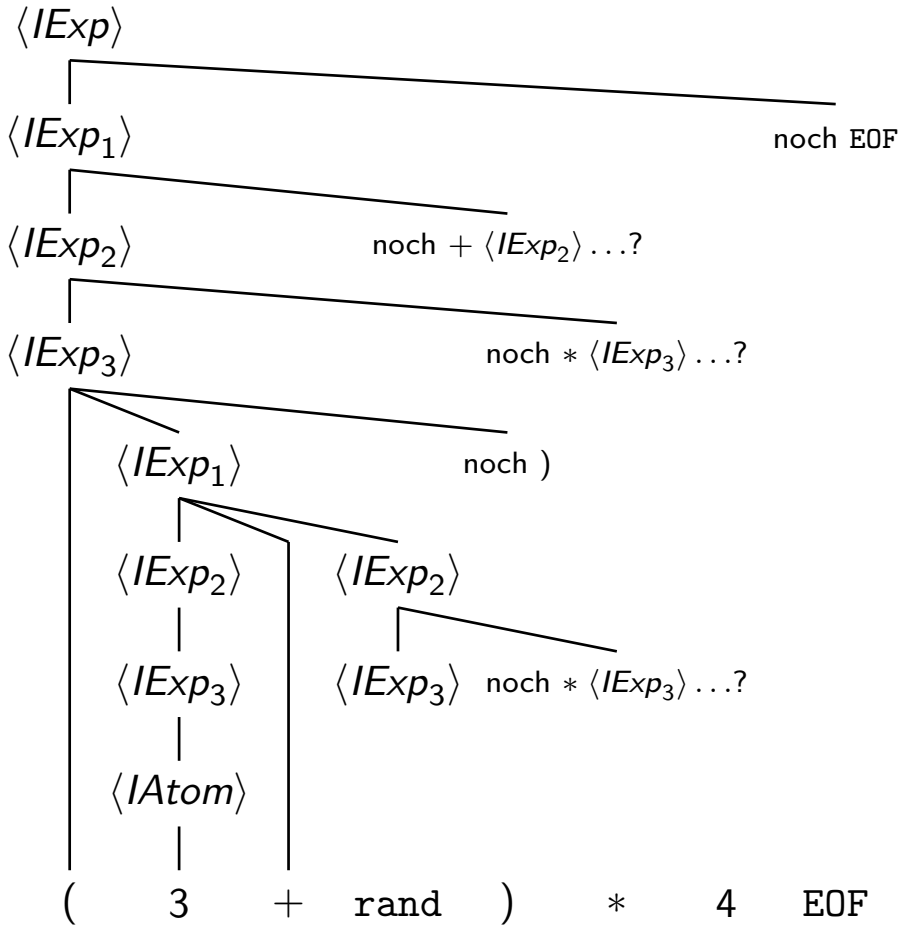
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \ \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$



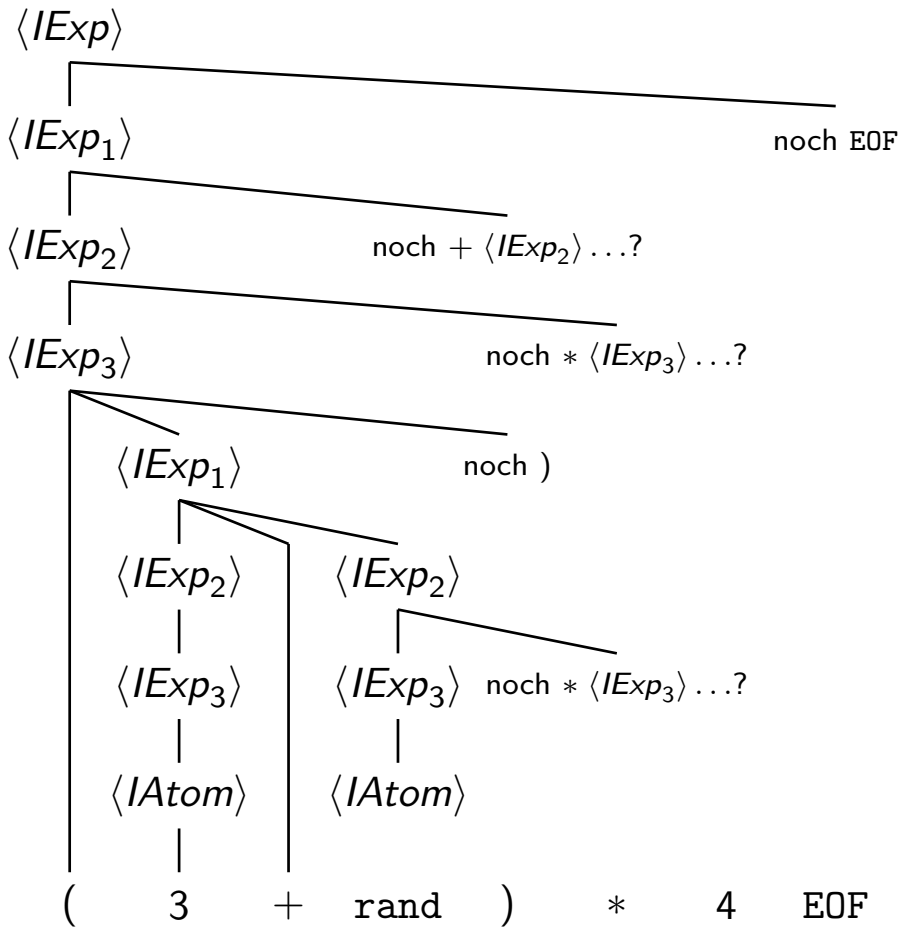
Beispiel LL(1)-Parsen

$$
\begin{array}{lcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



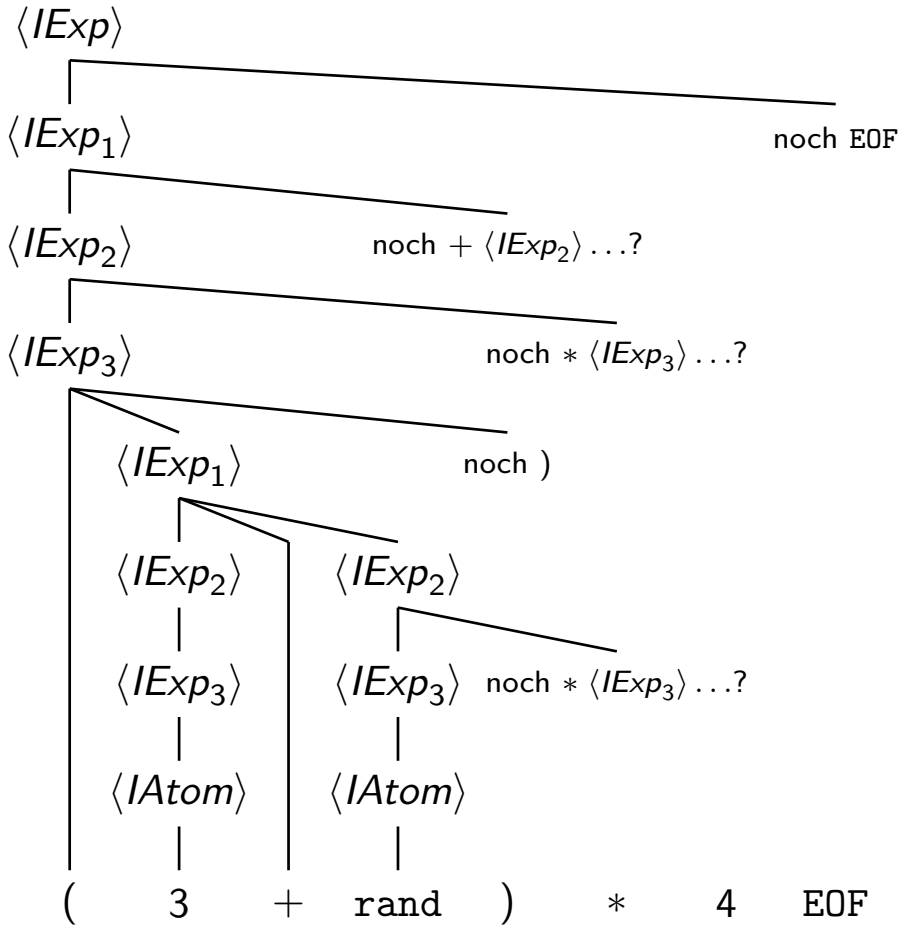
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



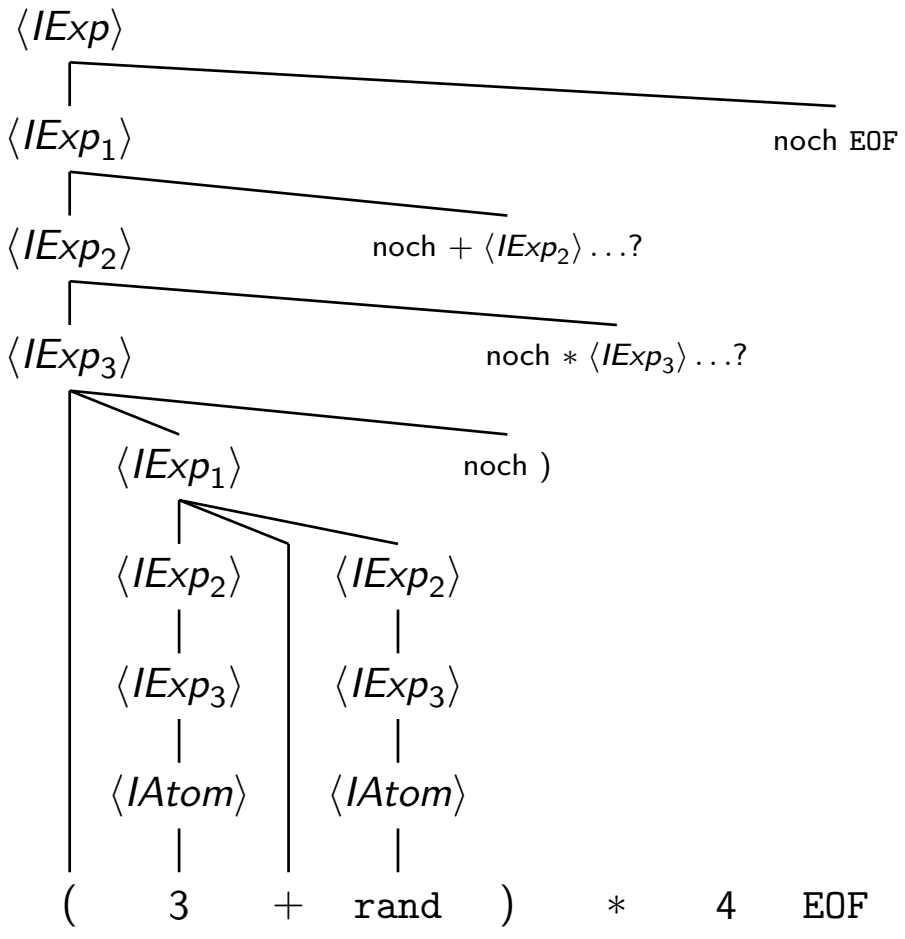
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



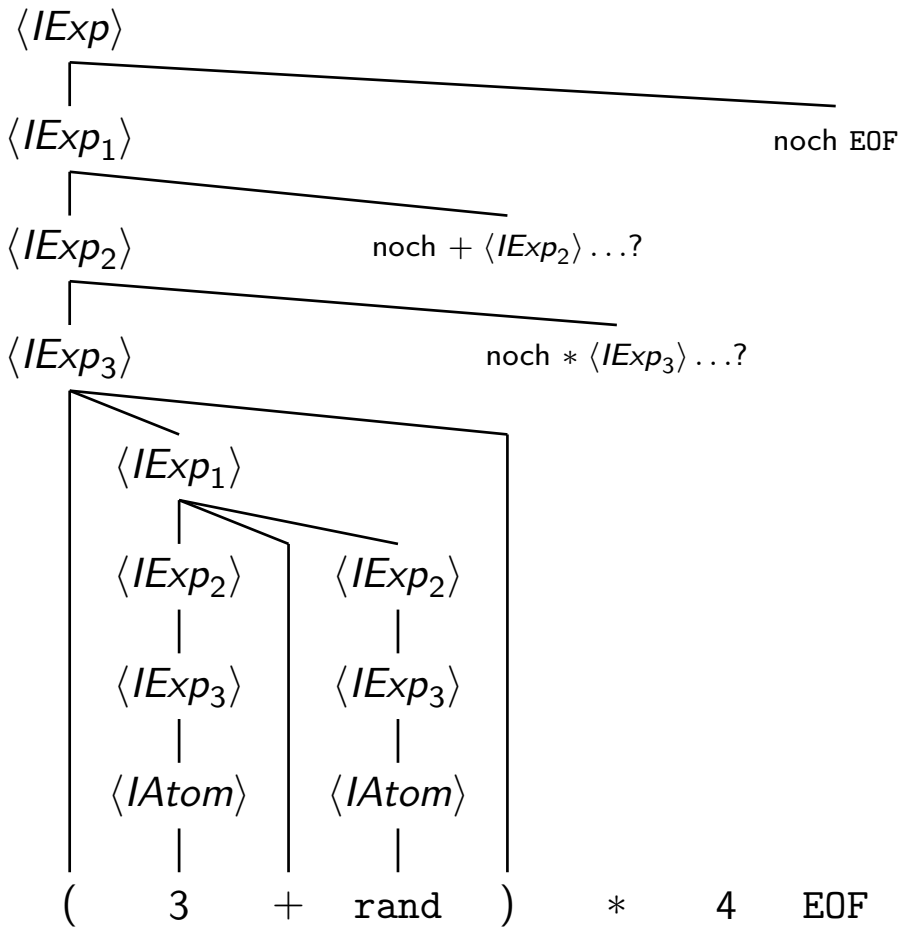
Beispiel LL(1)-Parsen

$$
\begin{array}{lcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \ \left(+ \ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \ \left(* \ \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



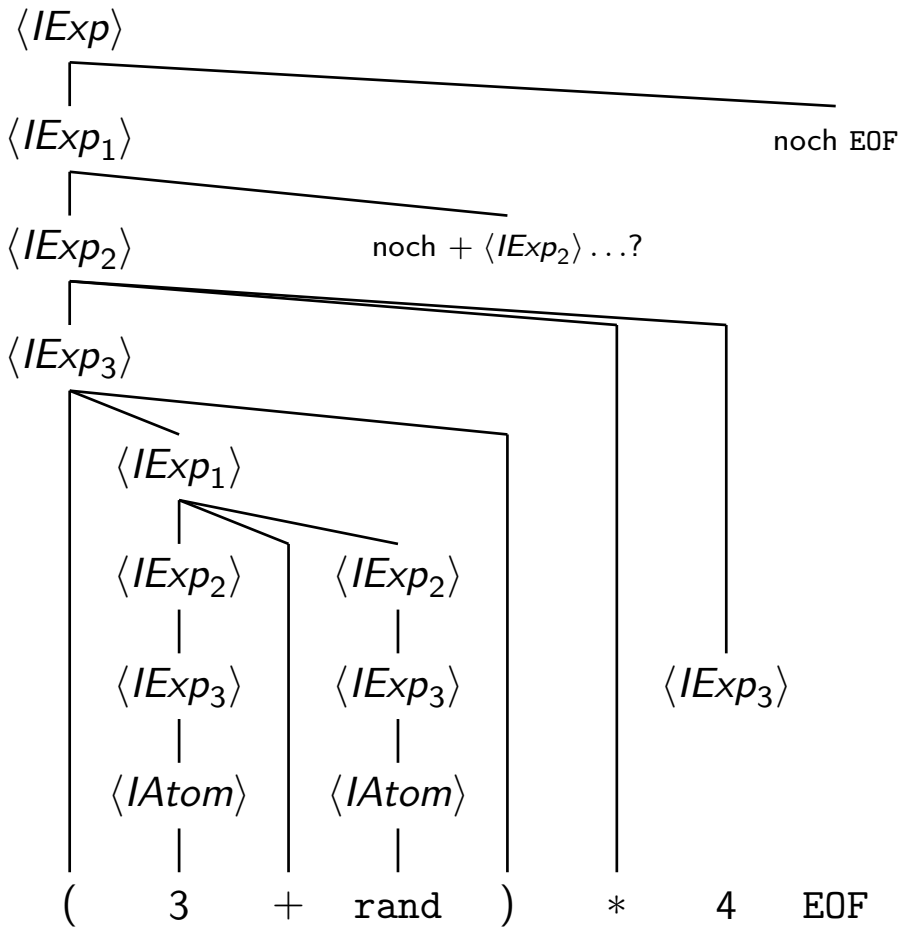
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle \mathit{IExp} \rangle &::= \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle &::= \langle \mathit{IExp}_2 \rangle \ \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle &::= \langle \mathit{IExp}_3 \rangle \ \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle &::= (\ \langle \mathit{IExp}_1 \rangle \ ) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle &::= \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{aligned}
$$



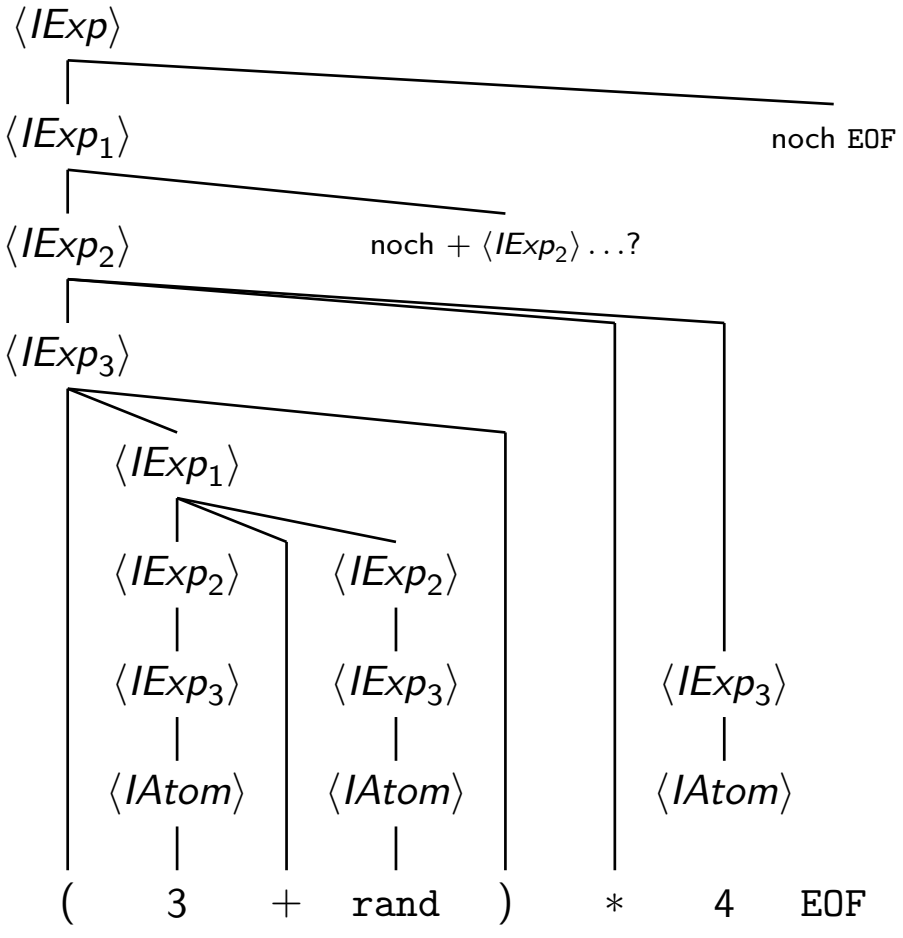
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \ \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \ \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



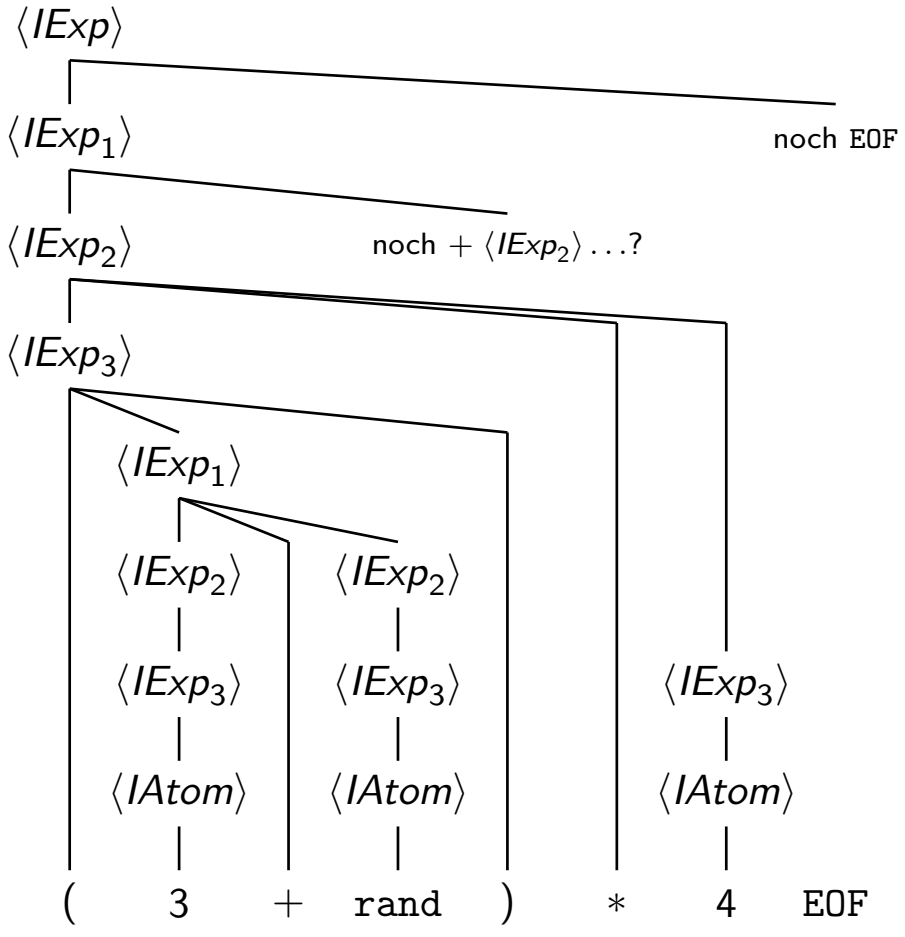
Beispiel LL(1)-Parsen

$$
\begin{aligned}
\langle IExp \rangle &::= \langle IExp_1 \rangle \texttt{ EOF} \\
\langle IExp_1 \rangle &::= \langle IExp_2 \rangle \left(+ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle &::= \langle IExp_3 \rangle \left(* \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle &::= \texttt{(} \langle IExp_1 \rangle \texttt{)} \mid \langle IAtom \rangle \\
\langle IAtom \rangle &::= Num \mid Id \mid \texttt{rand}
\end{aligned}
$$



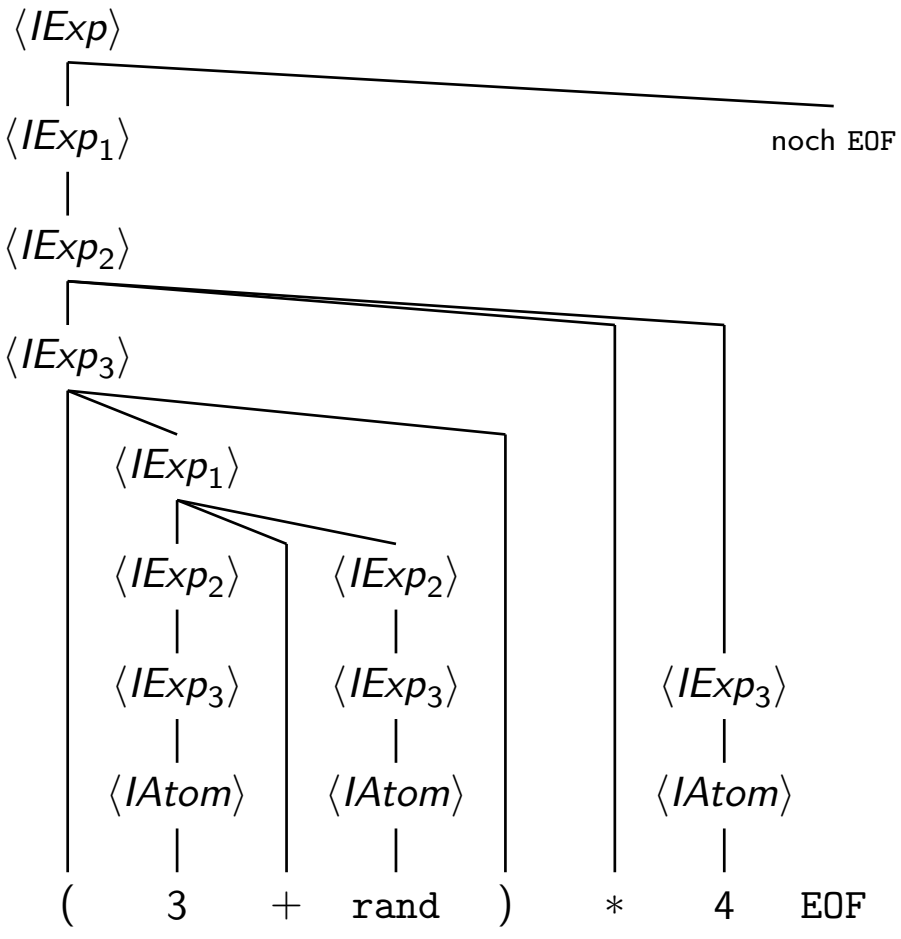
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \ \left(+ \ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \ \left(* \ \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$



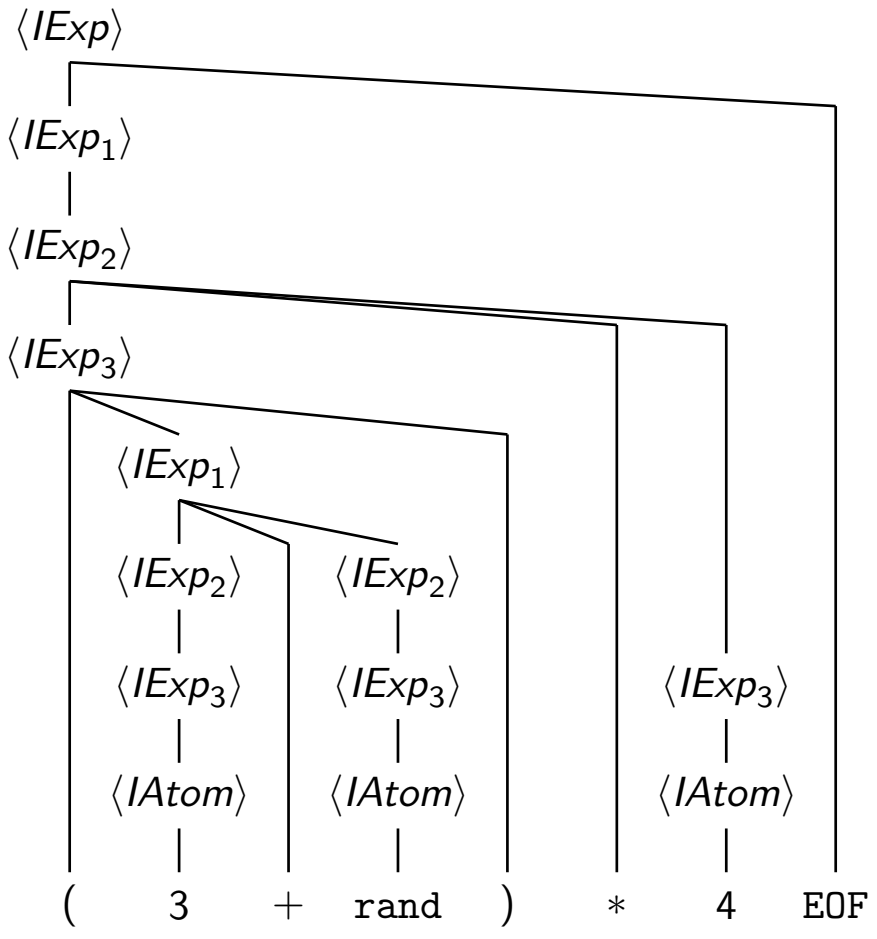
Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle IExp \rangle & ::= & \langle IExp_1 \rangle \ \texttt{EOF} \\
\langle IExp_1 \rangle & ::= & \langle IExp_2 \rangle \ \left(+ \ \langle IExp_2 \rangle \right)^* \\
\langle IExp_2 \rangle & ::= & \langle IExp_3 \rangle \ \left(* \ \langle IExp_3 \rangle \right)^* \\
\langle IExp_3 \rangle & ::= & (\ \langle IExp_1 \rangle \ ) \mid \langle IAtom \rangle \\
\langle IAtom \rangle & ::= & Num \mid Id \mid \texttt{rand}
\end{array}
$$



Beispiel LL(1)-Parsen

$$
\begin{array}{rcl}
\langle \mathit{IExp} \rangle & ::= & \langle \mathit{IExp}_1 \rangle \ \texttt{EOF} \\
\langle \mathit{IExp}_1 \rangle & ::= & \langle \mathit{IExp}_2 \rangle \left(+ \langle \mathit{IExp}_2 \rangle \right)^* \\
\langle \mathit{IExp}_2 \rangle & ::= & \langle \mathit{IExp}_3 \rangle \left(* \langle \mathit{IExp}_3 \rangle \right)^* \\
\langle \mathit{IExp}_3 \rangle & ::= & (\langle \mathit{IExp}_1 \rangle) \mid \langle \mathit{IAtom} \rangle \\
\langle \mathit{IAtom} \rangle & ::= & \mathit{Num} \mid \mathit{Id} \mid \texttt{rand}
\end{array}
$$



Zu beachten

- Ableitungsbaum bzgl. konkreter Syntax beim LL(k)-Parsen ist Baum der rekursiven Aufrufe, Rückgabe sollte aber Struktur bzgl. abstrakter Syntax sein (hier in Bsp. nicht gezeigt)

- Bsp.: "IntParser" auf Homepage

- Parser und Lexer kann man auch automatisch mit *Parsergeneratoren*, aus Grammatik erzeugen lassen – hier nicht weiter behandelt, in Compilern Parser normalerweise auch von Hand geschrieben

- 12-Variablen Regel in Spezifikation $\rightsquigarrow$ Menge der korrekten AntBrain-Programme nicht kontextfrei

  Regel beim Parsen zuerst einmal ignorieren, Anzahl vorkommender Variablen in Programm kann leicht danach an Baum in abstrakter Syntax bestimmt werden