Tutorium 13: Syntaktische Analyse

David Kaufmann

8. Februar 2023

Tutorium Programmierparadigmen am KIT

Compiler

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
- $\rightsquigarrow$ Programme in Maschinensprache sind schwer les-/schreibbar

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
- $\leadsto$ Programme in Maschinensprache sind schwer les-/schreibbar
- Also: Erfinde einfacher zu Schreibende ($\approx$ mächtigere) Sprache, die dann in die Sprache der Maschine übersetzt wird.
- Diesen Übersetzungsschritt sollte optimalerweise ein Programm erledigen, da wir sonst auch einfach direkt Maschinensprache-Programme schreiben können.

Compiler

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Scala, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript/WebAssembly

Compiler

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Scala, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript/WebAssembly
- Single-pass vs. Multi-pass
  - Single-pass: Eingabe wird einmal gelesen, Ausgabe währenddessen erzeugt (ältere Compiler)
  - Multi-pass: Eingabe wird in Zwischenschritten in verschiedene Repräsentationen umgewandelt
    - Quellsprache, Tokens, AST, Zwischensprache, Zielsprache

Lexikalische Analyse

```
int x1 = 123;
print("123");
```

```
int, id[x1], assign,
intlit[123], semi,
id[print], lp,
stringlit["123"], ...
```

- Lexikalische Analyse (Tokenisierung) verarbeitet eine Zeichensequenz in eine Liste von *Tokens*.
- Tokens sind Zeichengruppen, denen eine Semantik innewohnt:
  - `int` — Typ einer Ganzzahl
  - `=` — Zuweisungsoperator
  - `x1` — Variablen- oder Methodenname
  - `123` — Literal einer Ganzzahl
  - `"123"` — String-Literal
  - etc.
- Lösbar mit regulären Ausdrücken, Automaten

Syntaktische Analyse

- Syntaktische Analyse stellt die unterliegende (Baum-)Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header + Inhalt-Struktur von Mails
  - Verschachtelte arithmetische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.

Syntaktische Analyse

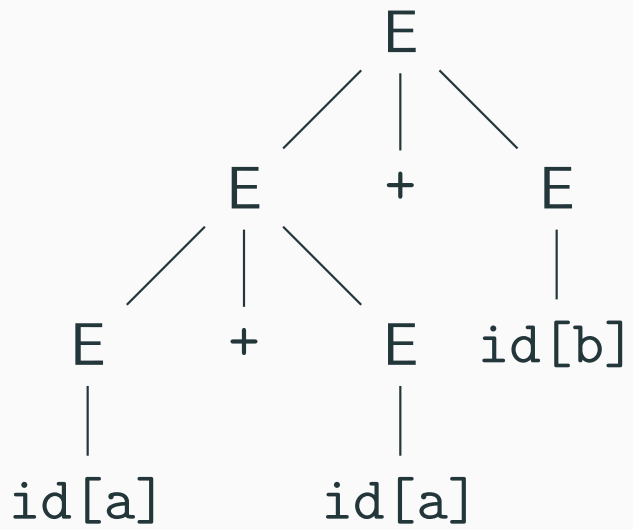
- Syntaktische Analyse stellt die unterliegende (Baum-)Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header + Inhalt-Struktur von Mails
  - Verschachtelte arithmetische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.
- Übliche Vorgehensweise (in PP):
  - Grammatik G erfinden
  - Ggf. G in andere Form G' bringen
  - Rekursiven Abstiegsparser für G' implementieren
- Alternativ: Parser-Kombinatoren, Yacc, etc.

Beispiel: Arithmetische Ausdrücke

`a+a+b` `a*c+b` `a*c+b*d`

- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- Beispielgrammatik:

$$
\begin{aligned}
E \rightarrow\ & E\ \texttt{+}\ E \\
\mid\ & E\ \texttt{*}\ E \\
\mid\ & \texttt{(}\ E\ \texttt{)} \\
\mid\ & \texttt{id}
\end{aligned}
$$

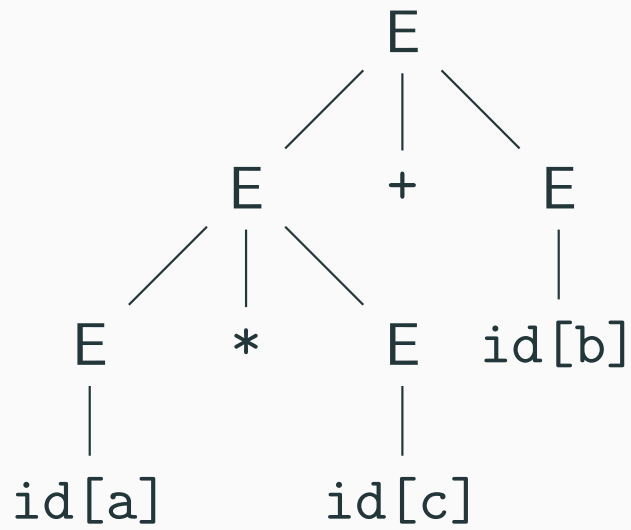


Beispiel: Arithmetische Ausdrücke

`a+a+b` `a*c+b` `a*c+b*d`

- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- Beispielgrammatik:

$$
\begin{aligned}
E \rightarrow\ & E\ \texttt{+}\ E \\
\mid\ & E\ \texttt{*}\ E \\
\mid\ & \texttt{(}\ E\ \texttt{)} \\
\mid\ & \texttt{id}
\end{aligned}
$$

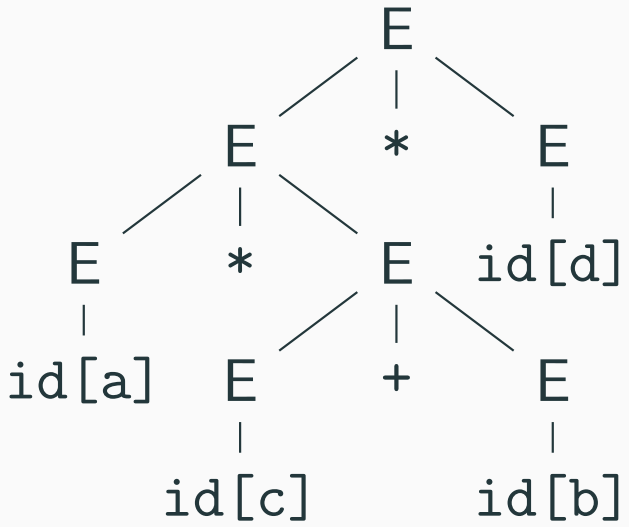


Beispiel: Arithmetische Ausdrücke

`a+a+b` `a*c+b` `a*c+b*d`

- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- Beispielgrammatik:

$$
\begin{aligned}
E \rightarrow\ & E\ \texttt{+}\ E \\
\mid\ & E\ \texttt{*}\ E \\
\mid\ & \texttt{(}\ E\ \texttt{)} \\
\mid\ & \texttt{id}
\end{aligned}
$$



Beispiel: Mathematische Ausdrücke

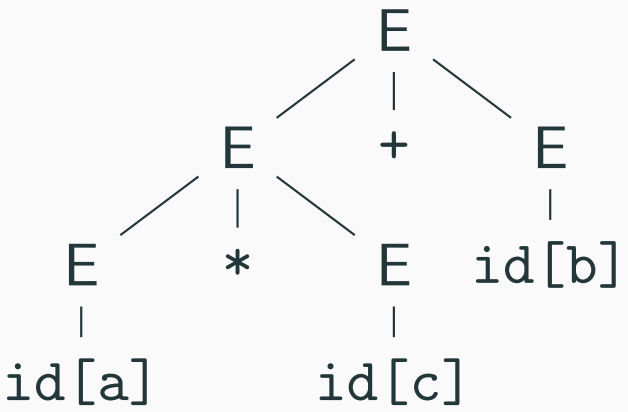
$$E \rightarrow E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

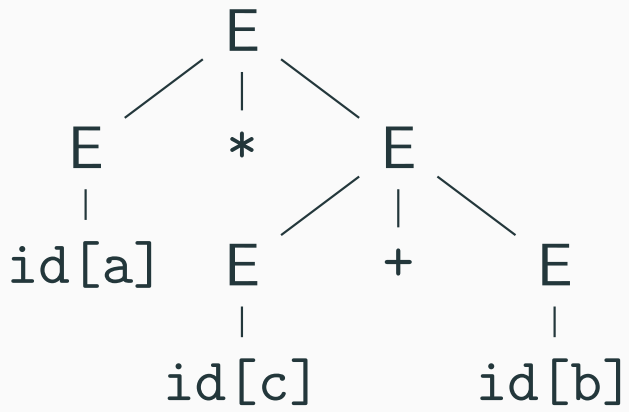
```
a*c+b
```

Beispiel: Mathematische Ausdrücke

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

a*c+b





- Grammatik nicht eindeutig $\rightsquigarrow$ schlecht
- Grammatik garantiert nicht Punkt-vor-Strich $\rightsquigarrow$ schlecht
- Grammatik ist linksrekursiv $\rightsquigarrow$ nicht einfach zu parsen $\rightsquigarrow$ schlecht

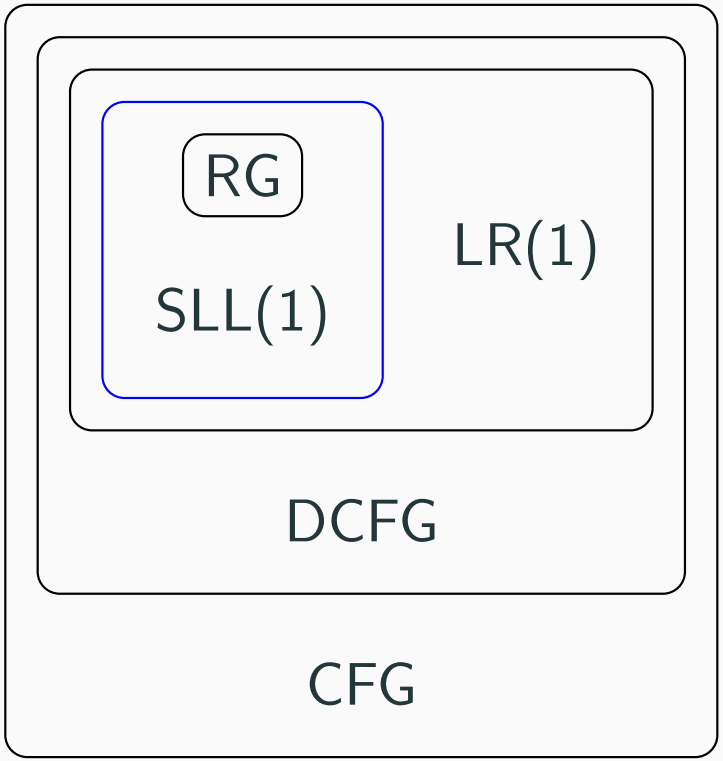
Grammar Engineering 1 — Präzedenz

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{id}$$

- Punkt-vor-Strich („Operatorpräzedenz“) wird von dieser naiven Grammatik nicht beachtet.
- Lösung: Ein Nichtterminal pro Präzedenzstufe:
  - *Summen* von *Produkten* von *Atomen*.
  - Herkömmliche Begriffe: Ausdruck, Term und Faktor.

$$\begin{aligned} \mathit{Expr} &\rightarrow \mathit{Expr} \texttt{ + } \mathit{Term} \\ &\mid \mathit{Term} \\ \mathit{Term} &\rightarrow \mathit{Term} \texttt{ * } \mathit{Factor} \\ &\mid \mathit{Factor} \\ \mathit{Factor} &\rightarrow \texttt{(} \mathit{Expr} \texttt{)} \mid \texttt{id} \end{aligned}$$

Welche Art von Grammatik wollen wir denn genau?

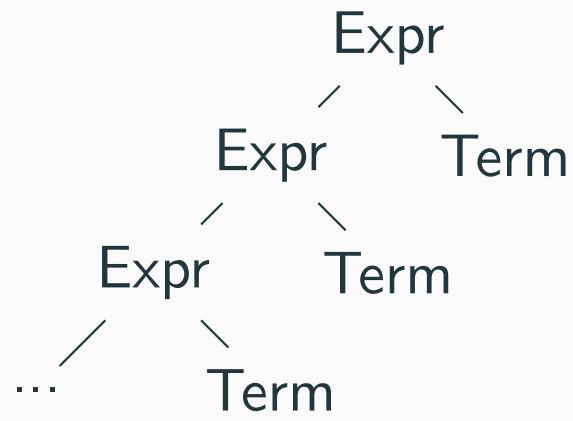


- CFG-Parsen ist i.A. in $O(n^3)$, bspw. Earley-Algorithmus.
- Reguläre Grammatiken ($\approx$ reg. Sprachen) sind uns nicht mächtig genug.
- LR: Left-to-right, Rightmost
- LL: Left-to-right, Leftmost
- SLL-Parsing $\in O(n)$

CFG: Context-Free Grammar/Kontextfreie Grammatik

Grammar Engineering 2 — Linksrekursion eliminieren

$$
\begin{aligned}
\mathit{Expr} &\to \mathit{Expr} + \mathit{Term} \\
&\mid \mathit{Term} \\
\mathit{Term} &\to \mathit{Term} * \mathit{Factor} \\
&\mid \mathit{Factor} \\
\mathit{Factor} &\to (\mathit{Expr}) \mid \texttt{id}
\end{aligned}
$$



Problem: Die Linksableitung des Symbols *Expr* in dieser Grammatik ist eine endlose Schleife.

Grammar Engineering 2 — Linksrekursion eliminieren

$$
\begin{aligned}
\mathit{Expr} &\to \mathit{Expr} \texttt{ + } \mathit{Term} \\
&\mid \mathit{Term} \\
\mathit{Term} &\to \mathit{Term} * \mathit{Factor} \\
&\mid \mathit{Factor} \\
\mathit{Factor} &\to \texttt{(} \mathit{Expr} \texttt{)} \mid \texttt{id}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &\to \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' &\to \texttt{+ } \mathit{Term}\ \mathit{Expr}' \\
&\mid \epsilon \\
\mathit{Term} &\to \mathit{Factor}\ \mathit{Term}' \\
\mathit{Term}' &\to * \mathit{Factor}\ \mathit{Term}' \\
&\mid \epsilon \\
\mathit{Factor} &\to \texttt{(} \mathit{Expr} \texttt{)} \mid \texttt{id}
\end{aligned}
$$

Lösung: Linksrekursion eliminieren, durch folgendes Umschreiben der Grammatik:

Grammar Engineering 2 — Linksrekursion eliminieren

$$
\begin{aligned}
\mathit{Expr} &\to \mathit{Expr}\ \texttt{+}\ \mathit{Term} \\
&\mid \mathit{Term} \\
\mathit{Term} &\to \mathit{Term} * \mathit{Factor} \\
&\mid \mathit{Factor} \\
\mathit{Factor} &\to \texttt{(}\ \mathit{Expr}\ \texttt{)} \mid \texttt{id}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &\to \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' &\to \texttt{+}\ \mathit{Term}\ \mathit{Expr}' \\
&\mid \epsilon \\
\mathit{Term} &\to \mathit{Factor}\ \mathit{Term}' \\
\mathit{Term}' &\to * \mathit{Factor}\ \mathit{Term}' \\
&\mid \epsilon \\
\mathit{Factor} &\to \texttt{(}\ \mathit{Expr}\ \texttt{)} \mid \texttt{id}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Sym} &\to \mathit{Sym}\ \alpha \\
&\mid \beta
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Sym} &\to \beta\ \mathit{Sym}' \\
\mathit{Sym}' &\to \alpha\ \mathit{Sym}' \\
&\mid \epsilon
\end{aligned}
$$

Verbesserte Grammatik

$$
\begin{aligned}
E &\to T\ EList \\
EList &\to \texttt{+}\ T\ EList \\
&\mid \epsilon \\
T &\to F\ TList \\
TList &\to \texttt{*}\ F\ TList \\
&\mid \epsilon \\
F &\to \texttt{(}\ E\ \texttt{)} \mid \texttt{id}
\end{aligned}
$$

- Grammatik ist eindeutig ✓
- Grammatik erzeugt nur korrekte Terme ✓
- Grammatik enthält keine Linksrekursion ✓

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \epsilon \mid \texttt{+}\ \ T\ \ \mathit{EList} \mid \texttt{-}\ \ T\ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \epsilon \mid \texttt{+}\ \ T\ \ \mathit{EList} \mid \texttt{-}\ \ T\ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere *Indizmenge* $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \epsilon \mid \texttt{+}\ \ T\ \ \mathit{EList} \mid \texttt{-}\ \ T\ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere *Indizmenge* $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- $IM_1(\mathit{EList} \rightarrow \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\mathit{EList})) = \{), \#\}$
- $IM_1(\mathit{EList} \rightarrow \texttt{+}\ \ T\ \ \mathit{EList}) = \mathrm{First}_1(\texttt{+}\ T\ \mathit{EList}\ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{+}\}$
- $IM_1(\mathit{EList} \rightarrow \texttt{-}\ \ T\ \ \mathit{EList}) = \mathrm{First}_1(\texttt{-}\ T\ \mathit{EList}\ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{-}\}$

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere *Indizmenge* $\mathit{IM}_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $\mathit{IM}_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- $\mathit{IM}_1(\mathit{EList} \rightarrow \ \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\mathit{EList})) = \{), \#\}$
- $\mathit{IM}_1(\mathit{EList} \rightarrow \ \texttt{+} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{+} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{+}\}$
- $\mathit{IM}_1(\mathit{EList} \rightarrow \ \texttt{-} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{-} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{-}\}$
- $\mathrm{First}_k(A)$: Menge an möglichen ersten k Token in A
- $\mathrm{Follow}_k(A)$: Menge an möglichen ersten k Token nach A

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form

$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form

$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

$$E \to E \texttt{ + } T \mid E \texttt{ - } T \mid T$$
$$T \to T \texttt{ * } F \mid T \texttt{ / } F \mid F$$
$$F \to \texttt{num} \mid \texttt{(} E \texttt{)}$$

- Begründet formal, dass obige Grammatik nicht $\mathrm{SLL}(1)$.
- Berechnet $\mathrm{Follow}_1(N)$ für $N \in \{E, T, F\}$.

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow T \;\; EList \\
EList &\rightarrow \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\rightarrow F \;\; TList \\
TList &\rightarrow \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\rightarrow \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow T \;\; EList \\
EList &\rightarrow \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\rightarrow F \;\; TList \\
TList &\rightarrow \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\rightarrow \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

- Was bringt uns das diese Grammatik in SLL(1)-Form ist?

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow T \;\; EList \\
EList &\rightarrow \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\rightarrow F \;\; TList \\
TList &\rightarrow \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\rightarrow \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

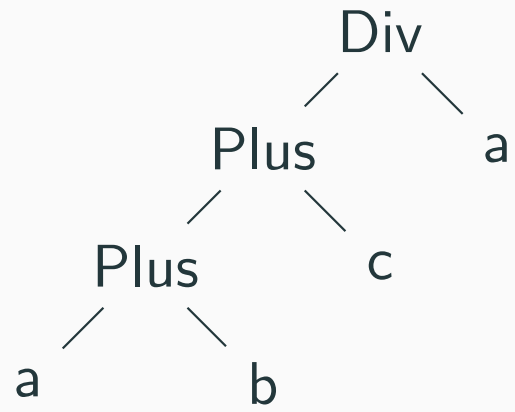
- Was bringt uns das diese Grammatik in SLL(1)-Form ist?
- G ist jetzt einfach ausprogrammierbar:
  - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
  - `lexer.lex()` konsumiert das aktuelle Token
  - `lexer.current` gibt nicht konsumierenden Zugriff auf das aktuelle Token

AST

- Der Parser soll einen AST zurückgeben
- Verwendet die Klassen aus `./demos/compiler/Exp.java`
- Am Ende soll der Parser einen Baum wie links zurückgeben

Beispiel:

$$(a + b + c)/a$$



WS2122