

# LL-Parser (Theorie)

---

BC George (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

# Wiederholung

---

- Warum reichen uns DFAs nicht zum Matchen von Eingabezeichen?
- Wie können wir sie minimal erweitern?
- Sind PDAs deterministisch?
- Wie sind kontextfreie Grammatiken definiert?
- Sind kontextfreie Grammatiken eindeutig?

# Motivation

---

# Was brauchen wir für die Syntaxanalyse von Programmen?

- einen Grammatiktypen, aus dem sich manuell oder automatisiert ein Programm zur deterministischen Syntaxanalyse erstellen lässt
- einen Algorithmus zum sog. Parsen von Programmen mit Hilfe einer solchen Grammatik

# Themen für heute

- Arten der Syntaxanalyse
- mehrdeutige Sprachen
- Top-down-Analyse
- LL(k)-Grammtiken

# Syntaxanalyse

---

Wir verstehen unter Syntax eine Menge von Regeln, die die Struktur von Daten (z. B. Programmen) bestimmen.

Syntaxanalyse ist die Bestimmung, ob Eingabedaten einer vorgegebenen Syntax entsprechen.

Diese vorgegebene Syntax wird im Compilerbau mit einer Grammatik beschrieben.



# Ziele der Syntaxanalyse

- aussagekräftige Fehlermeldungen, wenn ein Eingabeprogramm syntaktisch nicht korrekt ist
- evtl. Fehlerkorrektur
- Bestimmung der syntaktischen Struktur eines Programms
- Erstellung des AST (abstrakter Syntaxbaum): Der Parse Tree ohne Symbole, die nach der Syntaxanalyse inhaltlich irrelevant sind (z. B. Semikolons, manche Schlüsselwörter)
- die Symboltabelle(n) mit Informationen bzgl. Bezeichner (Variable, Funktionen und Methoden, Klassen, benutzerdefinierte Typen, Parameter, ...), aber auch die Gültigkeitsbereiche.

# Arten der Syntaxanalyse

Die Syntax bezieht sich auf die Struktur der zu analysierenden Eingabe, z. B. einem Computerprogramm in einer Hochsprache. Diese Struktur wird mit formalen Grammatiken beschrieben. Einsetzbar sind Grammatiken, die deterministisch kontextfreie Sprachen erzeugen.

- Top-Down-Analyse: Aufbau des Parse trees von oben
  - Parsen durch rekursiven Abstieg
  - LL-Parsing
- Bottom-Up-Analyse: LR-Parsing

# Mehrdeutigkeiten

Wir können nur mit eindeutigen Grammatiken arbeiten, aber:

**Def.:** Eine formale Sprache  $L$  heißt *inhärent mehrdeutige Sprache*, wenn jede formale Grammatik  $G$  mit  $L(G) = L$  mehrdeutig ist.

Das heißt, solche Grammatiken existieren.

⇒ Es gibt keinen generellen Algorithmus, um Grammatiken eindeutig zu machen.

## Bevor wir richtig anfangen...

**Def.:** Ein Nichtterminal  $A$  einer kontextfreien Grammatik  $G$  heißt *unerreichbar*, falls es kein  $a, b \in (N \cup T)^*$  gibt mit  $S \xRightarrow{*} aAb$ . Ein Nichtterminal  $A$  einer Grammatik  $G$  heißt *nutzlos*, wenn es kein Wort  $w \in T^*$  gibt mit  $A \xRightarrow{*} w$ .

**Def.:** Eine kontextfreie Grammatik  $G = (N, T, P, S)$  heißt *reduziert*, wenn es keine nutzlosen oder unerreichbaren Nichtterminale in  $N$  gibt.

Bevor mit einer Grammatik weitergearbeitet wird, müssen erst alle nutzlosen und dann alle unerreichbaren Symbole eliminiert werden. Wir betrachten ab jetzt nur reduzierte Grammatiken.

# Top-Down-Analyse

---

Wie würden Sie manuell parsen?

# Algorithmus: Rekursiver Abstieg

Hier ist ein einfacher Algorithmus, der (indeterministisch) einen Ableitungsbaum vom Nonterminal  $X$  von oben nach unten aufbaut:

**Eingabe:** Ein Nichtterminal  $X$  und das nächste zu verarbeitende Eingabezeichen  $a$ .

```
RecursiveDescent( $X, a$ ) //  $X \in N, a \in T$   
  for a production,  $X \rightarrow Y_1 Y_2 \dots Y_n$   
    for  $i = 1$  to  $n$   
      if  $Y_i \in N$   
        RecursiveDescent( $Y_i, a$ )  
      else if  $Y_i \neq a$   
        error
```

**Abbildung 1:** Recursive Descent-Algorithmus

Was ist mit

1)  $X \rightarrow a\alpha \mid b\beta$

2)  $X \rightarrow B\alpha \mid C\beta$

3)  $X \rightarrow B\alpha \mid B\beta$

4)  $X \rightarrow B\alpha \mid C\beta$  und  $C \rightarrow B$

5)  $X \rightarrow X\beta$

6)  $X \rightarrow B\alpha$  und  $B \rightarrow X\beta$

$$X, B, C, D \in N^*; a, b, c, d \in T^*; \beta, \alpha, \beta \in (N \cup T)^*$$



$$X \rightarrow BC \mid BD$$

# Algorithmus: Linksfaktorisierung

**Eingabe:** Eine Grammatik  $G = (N, T, P, S)$

**Ausgabe:** Eine äquivalente links-faktorierte Grammatik  $G'$

```
repeat
  for each  $X \in N$ 
    find the longest prefix  $\alpha \in (N \cup T)^*$  common to at least two
      of its productions

    if  $\alpha \neq \epsilon$  exists
      replace all  $X$ -productions

        
$$X \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$$

        (with  $\gamma \neq \alpha\delta$  for a  $\delta \in (N \cup T)^*$ )

        by the following rules with a new nonterminal  $Y$ :

          
$$X \rightarrow \alpha Y \mid \gamma$$

          
$$Y \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$


until there are no two  $X \in N$  with productions with common prefixes
```

**Abbildung 2:** Algorithmus zur Linksfaktorisierung

**Def.:** Eine Grammatik  $G = (N, T, P, S)$  heißt *linksrekursiv*, wenn sie ein Nichtterminal  $X$  hat, für das es eine Ableitung  $X \xRightarrow{+} X\alpha$  für ein  $\alpha \in (N \cup T)^*$  gibt.

Linksrekursion gibt es

*direkt:*  $X \rightarrow X\alpha$

und

*indirekt:*  $X \rightarrow \dots \rightarrow \dots \rightarrow X\alpha$

# Algorithmus: Entfernung von direkter Linksrekursion

**Eingabe:** Eine Grammatik  $G = (N, T, P, S)$

**Ausgabe:** Eine äquivalente Grammatik  $G'$  ohne direkte Linksrekursion

```
replace each production  $X \rightarrow X\alpha \mid \beta$  by  
     $X \rightarrow \beta Y$   
     $Y \rightarrow \alpha Y \mid \epsilon$ 
```

**Abbildung 3:** Algorithmus zur Entfernung direkter Linksrekursion

# Algorithmus: Entfernung von indirekter Linksrekursion

**Eingabe:** Eine Grammatik  $G = (N, T, P, S)$  mit  $N = \{X_1, X_2, \dots, X_n\}$  ohne  $\epsilon$ -Regeln oder Zyklen der Form  $X_1 \rightarrow X_2, X_2 \rightarrow X_3, \dots, X_{m-1} \rightarrow X_m, X_m \rightarrow X_1$

**Ausgabe:** Eine äquivalente Grammatik  $G'$  ohne Linksrekursion

```
for i = 1 to n
  for j = 1 to i-1
    replace each  $X \rightarrow X_j \alpha$  by the productions
       $X_i \rightarrow \beta_1 \alpha \mid \beta_2 \alpha \mid \dots \mid \beta_k \alpha$ 
    with  $X_j \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$  are all the
       $X_j$ -productions
  remove all  $X_i$  - productions with direct left recursion
```

**Abbildung 4:** Algorithmus zur Entfernung indirekter Linksrekursion

# Arbeiten mit generierten Parseern: LL(k)-Grammatiken

---

# First-Mengen

$$S \rightarrow A \mid B \mid C$$

Welche Produktion nehmen?

Wir brauchen die “terminalen  $k$ -Anfänge” von Ableitungen von Nichtterminalen, um eindeutig die nächste zu benutzende Produktion festzulegen.  $k$  ist dabei die Anzahl der Vorschautoken.

**Def.:** Wir definieren *First* - Mengen einer Grammatik wie folgt:

- $a \in T^*, |a| \leq k : First_k(a) = \{a\}$
- $a \in T^*, |a| > k : First_k(a) = \{v \in T^* \mid a = vw, |v| = k\}$
- $\alpha \in (N \cup T)^* \setminus T^* : First_k(\alpha) = \{v \in T^* \mid \alpha \xRightarrow{*} w, \text{ mit } w \in T^*, First_k(w) = \{v\}\}$

**Def.:** Bei einer kontextfreien Grammatik  $G$  ist die *Linksableitung* von  $\alpha \in (N \cup T)^*$  die Ableitung, die man erhält, wenn in jedem Schritt das am weitesten links stehende Nichtterminal in  $\alpha$  abgeleitet wird.

Man schreibt  $\alpha \Rightarrow_I^* \beta$ .



Manchmal müssen wir wissen, welche terminalen Zeichen hinter einem Nichtterminal stehen können.

**Def.** Wir definieren *Follow* - Mengen einer Grammatik wie folgt:

$\forall \beta \in (N \cup T)^* :$

$$Follow_k(\beta) = \{w \in T^* \mid \exists \alpha, \gamma \in (N \cup T)^* \text{ mit } S \xRightarrow{*}_I \alpha\beta\gamma \text{ und } w \in First_k(\gamma)\}$$

**Def.:** Eine kontextfreie Grammatik  $G = (N, T, P, S)$  ist genau dann eine  $LL(k)$ -Grammatik, wenn für alle Linksableitungen der Form:

$$S \xRightarrow{*}_I wA\gamma \Rightarrow_I w\alpha\gamma \xRightarrow{*}_I wx$$

und

$$S \xRightarrow{*}_I wA\gamma \Rightarrow_I w\beta\gamma \xRightarrow{*}_I wy$$

mit  $(w, x, y \in T^*, \alpha, \beta, \gamma \in (N \cup T)^*, A \in N)$  und  $First_k(x) = First_k(y)$  gilt:

$$\alpha = \beta$$

# LL(k)-Grammatiken

Das hilft manchmal:

Für  $k = 1$ :  $G$  ist  $LL(1)$  :  $\forall A \rightarrow \alpha, A \rightarrow \beta \in P, \alpha \neq \beta$  gilt:

1.  $\neg \exists a \in T : \alpha \xRightarrow{*}_I a\alpha_1$  und  $\beta \xRightarrow{*}_I a\beta_1$
2.  $((\alpha \xRightarrow{*}_I \epsilon) \Rightarrow (\neg(\beta \xRightarrow{*}_I \epsilon)))$  und  $((\beta \xRightarrow{*}_I \epsilon) \Rightarrow (\neg(\alpha \xRightarrow{*}_I \epsilon)))$
3.  $((\beta \xRightarrow{*}_I \epsilon) \text{ und } (\alpha \xRightarrow{*}_I a\alpha_1)) \Rightarrow a \notin \text{Follow}(A)$
4.  $((\alpha \xRightarrow{*}_I \epsilon) \text{ und } (\beta \xRightarrow{*}_I a\beta_1)) \Rightarrow a \notin \text{Follow}(A)$

Die ersten beiden Zeilen bedeuten:

$\alpha$  und  $\beta$  können nicht beide  $\epsilon$  ableiten,  $\text{First}_1(\alpha) \cap \text{First}_1(\beta) = \emptyset$

Die dritte und vierte Zeile bedeuten:

$(\epsilon \in \text{First}_1(\beta)) \Rightarrow (\text{First}_1(\alpha) \cap \text{Follow}_1(A) = \emptyset)$

$(\epsilon \in \text{First}_1(\alpha)) \Rightarrow (\text{First}_1(\beta) \cap \text{Follow}_1(A) = \emptyset)$



# LL(k)-Sprachen

Die von  $LL(k)$ -Grammatiken erzeugten Sprachen sind eine echte Teilmenge der deterministisch parsbaren Sprachen.

Die von  $LL(k)$ -Grammatiken erzeugten Sprachen sind eine echte Teilmenge der von  $LL(k+1)$ -Grammatiken erzeugten Sprachen.

Für eine kontextfreie Grammatik  $G$  ist nicht entscheidbar, ob es eine  $LL(1)$  - Grammatik  $G'$  gibt mit  $L(G) = L(G')$ .

In der Praxis reichen  $LL(1)$  - Grammatiken oft. Hier gibt es effiziente Parsergeneratoren, deren Eingabe eine  $LL(k)$ - (meist  $LL(1)$ -) Grammatik ist, und die als Ausgabe den Quellcode eines (effizienten) tabellengesteuerten Parsers generieren.

# Algorithmus: Konstruktion einer LL-Parsertabelle

**Eingabe:** Eine Grammatik  $G = (N, T, P, S)$

**Ausgabe:** Eine Parsertabelle  $P$

```
for each production  $X \rightarrow \alpha$ 
  for each  $a \in First(\alpha)$ 
    add  $X \rightarrow \alpha$  to  $P[X, a]$ 

  if  $\epsilon \in First(\alpha)$ 
    for each  $b \in Follow(\alpha)$ 
      add  $X \rightarrow \alpha$  to  $P[X, b]$ 
      if  $\epsilon \in First(\alpha)$  and  $\perp \in Follow(X)$ 
        add  $X \rightarrow \alpha$  to  $P[X, \perp]$ 
```

**Abbildung 5:** Algorithmus zur Generierung einer LL-Parsertabelle

Hier ist  $\perp$  das Endezeichen des Inputs. Statt  $First_1(\alpha)$  und  $Follow_1(\alpha)$  wird oft nur  $First(\alpha)$  und  $Follow(\alpha)$  geschrieben.



Rekursive Programmierung bedeutet, dass das Laufzeitsystem einen Stack benutzt (bei einem Recursive-Descent-Parser, aber auch bei der Parsertabelle). Diesen Stack kann man auch “selbst programmieren”, d. h. einen PDA implementieren. Dabei wird ebenfalls die oben genannte Tabelle zur Bestimmung der nächsten anzuwendenden Produktion benutzt. Der Stack enthält die zu erwartenden Eingabezeichen, wenn immer eine Linksableitung gebildet wird. Diese Zeichen im Stack werden mit dem Input gematcht.



# Algorithmus: Tabellengesteuertes LL-Parsen mit einem PDA

**Eingabe:** Eine Grammatik  $G = (N, T, P, S)$ , eine Parsertabelle  $P$  mit  $w \perp$  als initialem Kellerinhalt

**Ausgabe:** Wenn  $w \in L(G)$ , eine Linksableitung von  $w$ , Fehler sonst

```
a = next_token()
X = top of stack // entfernt X vom Stack

while X  $\neq$   $\perp$ 

    if X = a
        a = next_token()

    else if X  $\in T$ 
        error

    else if  $P[X, a]$  leer
        error

    else if  $P[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ 
        process_production( $X \rightarrow Y_1 Y_2 \dots Y_k$ )
        push( $Y_1 Y_2 \dots Y_k$ ) //  $Y_1$  = top of stack

X = top of stack
```

Abbildung 6: Algorithmus zum tabellengesteuerten LL-Parsen

- Syntaxanalyse wird mit deterministisch kontextfreien Grammatiken durchgeführt.
- Eine Teilmenge der dazu gehörigen Sprachen lässt sich top-down parsen.
- Ein einfacher Recursive-Descent-Parser arbeitet mit Backtracking.
- Ein effizienter  $LL(k)$ -Parser realisiert einen DPDA und kann automatisch aus einer  $LL(k)$ -Grammatik generiert werden.
- Der Parser liefert in der Regel einen abstrakten Syntaxbaum.

# LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.