

# LL-Parser

---

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

- Syntaxanalyse
- Top-down-Analyse
- rekursiver Abstieg
- LL(k)-Analyse

# Syntaxanalyse

---

# 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 nach unten
  - Parsen durch rekursiven Abstieg
  - tabellengesteuertes LL-Parsing
- Bottom-Up-Analyse: LR-Parsing



## 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.

## Algorithmus: Rekursiver Abstieg

---

# Algorithmus: Rekursiver Abstieg

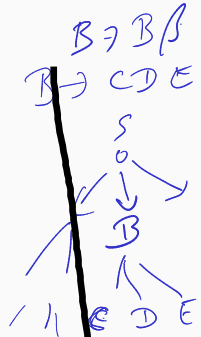
Hier ist ein einfacher Algorithmus, der (indeterministisch) top-down Ableitungen vom Nonterminal  $X$  aufbaut:

**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
```

$Y_i \in T$

Abbildung 2: Recursive Descent-Algorithmus



$Y_i \in N \cup T$

$C \rightarrow C\beta$   
Linkrekursion  
 $\Rightarrow$  Algorithmus

# **Tabellengesteuerte Parser: LL(k)-Grammatiken**

---

nach oben  
parieren

**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.

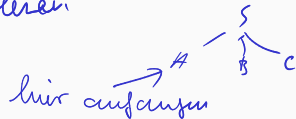
Man schreibt  $\alpha \Rightarrow_I^* \beta$ .

# First-Mengen

$\downarrow$   $\downarrow$   $\downarrow$ : Der Ableitungsbaum erhält um  
Eingabe von links nach rechts gelesen  
Linkschrittweise

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

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 sog. *Vorschautoken*.

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

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

# Follow-Mengen

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

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

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

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

$S \rightarrow AB \mid C$   
 $A \rightarrow \epsilon \mid \dots$  } hier muss zur Entscheidung, ob  $S$  nach  $A$  oder nach  $C$  abgeleitet werden muss, auch  $\text{First}(B)$  berücksichtigt werden

oder einem  $\beta \in (N \cup T)^*$

$k$  Symbole hinter  $B$

# LL(k)-Grammatiken

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

$$S \xRightarrow{*}_l wA\gamma \Rightarrow_l w\alpha\gamma \xRightarrow{*}_l wx$$

und

$$S \xRightarrow{*}_l wA\gamma \Rightarrow_l w\beta\gamma \xRightarrow{*}_l wy$$

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

$\alpha = \beta$ , d. h. die Ableitung von  $A$  ist eindeutig



# LL(1)-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.

$O(n)$

# 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  $P[A, \perp]$ 
```

ANTLR pddl  
LL\* gram analysieren

$a \in First(\alpha)$

$a$	
$S$	$S \rightarrow \alpha$

Endezeichen  
des Inputs

**Abbildung 3:** 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.

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  $\neq$   $\perp$ 

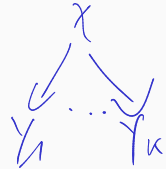
    if X = a
        a = next_token()

    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 4:** 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 (AST).



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