# LL-Parser

BC George (FH Bielefeld)

# Wiederholung

# PDAs und kontextfreie Grammatiken

- Warum reichen uns DFAs nicht zum Matchen von Eingabezeichen?
- Wie könnnen 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 Syntaxanlyse
- mehrdeutige Sprachen
- Top-down-Analyse
- LL(k)-Grammtiken

# Syntaxanalyse

# Syntax

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 Symboltablelle(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.

$\Rightarrow$ 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 $S \overset{*}{\Rightarrow} aAb$. Ein Nichtterminal $A$ einer Grammatik $G$ heißt *nutzlos*, wenn es kein Wort $w \in T^*$ gibt mit $A \overset{*}{\Rightarrow} w$.

**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?

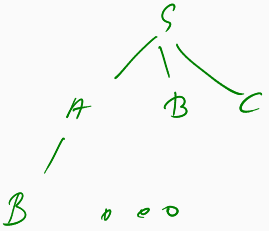
$S \to ABC$

$A \to aA \mid B$

$B \to b \mid C \mid c$

$C \to cC \mid \varepsilon$

Parsen von ccbcc




Manuelles Parsen findet oft top-down statt.

# 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 $a$.

vom Scanner

Nichtterminal, das weiter abgeleitet werden soll.

nichtdeterministisch

Backtracking

```
RecursiveDescent(X, a) //X ∈ N, a ∈ T
    for a production, X → Y_1 Y_2 ... Y_n
        for i = 1 to n
            if Y_i ∈ N
                RecursiveDescent(Y_i, a)
            else if Y_i ≠ a
                error
```

**Abbildung 1:** Recursive Descent-Algorithmus

# Grenzen des Algorithmus

Was ist mit

1) $X \to a\alpha \mid b\beta$ ok
2) $X \to B\alpha \mid C\beta$ Die Terminale Anfänge von B und C müssen bekannt sein
3) $X \to B\alpha \mid B\beta$ Linksfaktorisierung möglich
4) $X \to B\alpha \mid C\beta$ und $C \to B$ indirekte Linksfaktorisierung möglich
5) $X \to X\beta$ Linksrekursion entfernen
6) $X \to B\alpha$ und $B \to X\beta$ indirekte Linksrekursion entfernen

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

# Linksfaktorisierung

$X \to BC \mid BD$ Entscheidung verlagern

$X \to BE$ ($E$ neu)

$E \to C \mid D$

mit längstem Präfix durchführen

$X \to \underset{\text{ein}}{ABC} \mid \underset{\text{ein}}{AB}D$

# Algorithmus: Linksfaktorisierung

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

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

```
repeat
   for each X ∈ N
      find the longest prefix α ∈ (N ∪ T)* common to at least two
         of its productions

      if α ≠ ε exists
         replace all X-productions

               X → α β1 | α β2 | ... | α βn | γ
               (with γ ≠ αδ for a δ ∈ (N ∪ T)*)

                  by the following rules with a new nonterminal Y:

                     X → α Y | γ
                     Y → β1 | β2 | ... | βn

  until there are no two X ∈ N with productions with common prefixes
```

**Abbildung 2:** Algorithmus zur Linksfaktorisierung

# Linksrekursion

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

Linksrekursion gibt es

*direkt*: $X \to X\alpha$

und

*indirekt*: $X \to \ldots \to \ldots \to X\alpha$

# 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 → Xα | β by
        X → β Y
        Y → α Y | ε
```

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

# Algorithmus: Entfernung von indirekter Linksrekursion

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

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

```
for i = 1 to n
   for j = 1 to i-1

      replace each X → Xj α by the productions

         Xi → β1 α | β2 α | ... | βk α
         with Xj → β1 | β2 | ... | βk are all the
            Xj-productions

      remove all Xi - productions with direct left recursion
```

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

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

# First-Mengen

$S \to A \mid B \mid C$

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:

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

$A \to Bc \mid aaB$ $\quad First_1(A) = \{a, b\}$ $\quad First_1(B) = \{b\}$

$B \to bB \mid b$ $\quad First_2(A) = \{aa, bb, bc\}$ $\quad First_2(B) = \{bb, b\}$

# Linksableitungen


Die Reihenfolge der Ableitungen spielt für die Struktur des Ableitungsbaumes keine Rolle.

aber bei LL:

**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 $\alpha \overset{*}{\Rightarrow}_l \beta$.

# Follow-Mengen

$S \to AB$

$A \to \ldots \mid \varepsilon$

In $First_1(S)$ müssen auch die auf A folgenden Symbole enthalten sein.

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

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

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

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

# 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:

$S \overset{*}{\Rightarrow}_l wA\gamma \Rightarrow_l w\alpha\gamma \overset{*}{\Rightarrow}_l wx$ $\quad A \hookrightarrow \alpha$

und

$S \overset{*}{\Rightarrow}_l wA\gamma \Rightarrow_l w\beta\gamma \overset{*}{\Rightarrow}_l wy$ $\quad A \hookrightarrow \beta$

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 First-Menge bestimmt eindeutig die anzuwendende Produktion

# LL(k)-Grammatiken

Das hilft manchmal:

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

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

Die ersten beiden Zeilen bedeuten:

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

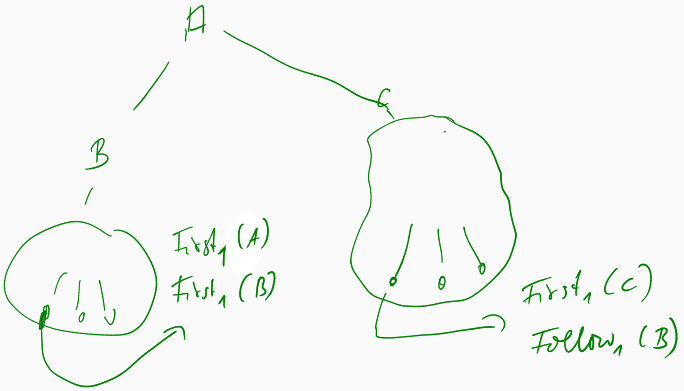
Die dritte und vierte Zeile bedeuten:

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

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

# LL(1)-Grammatiken

$A \to BC \qquad B \to \ldots \qquad C \to \ldots$



# 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)

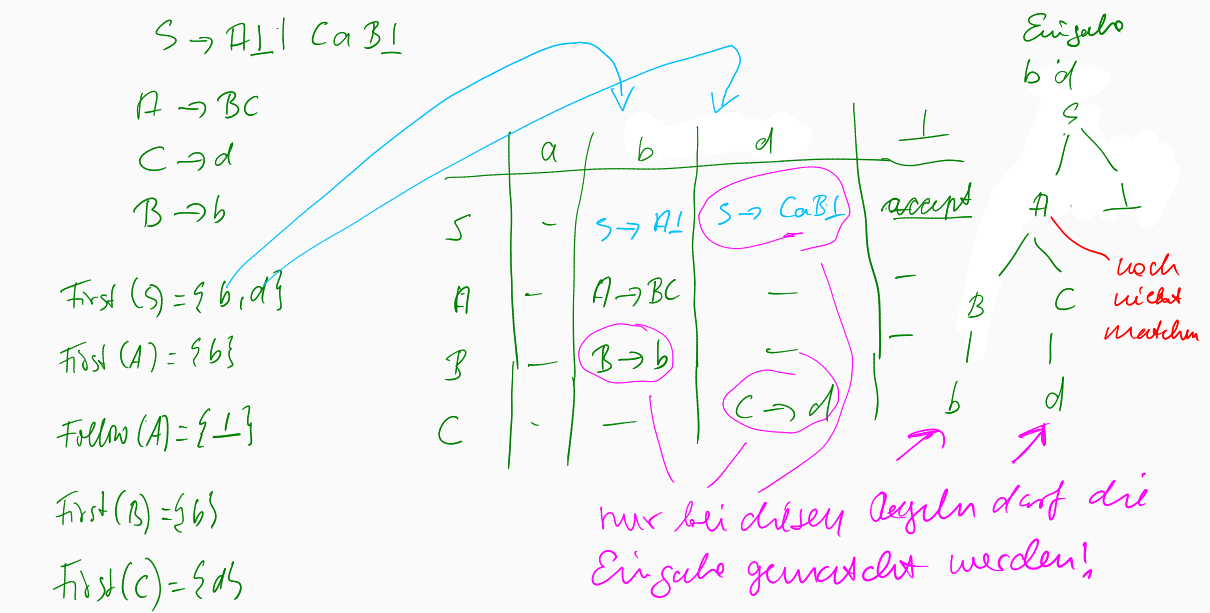
**Ausgabe:** Eine Parsertabelle *P*

```
for each production X → α
   for each a ∈ First(α)
      add X → α to P[X, a]

   if ϵ ∈ First(α)
      for each b ∈ Follow(α)
         add X → α to P[X, b]
         if ϵ ∈ First(α) and ⊥ ∈ Follow(X)
            add X → α to P[A, ⊥]
```

**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.

# LL-Parsertabellen



# LL-Parsertabellen

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

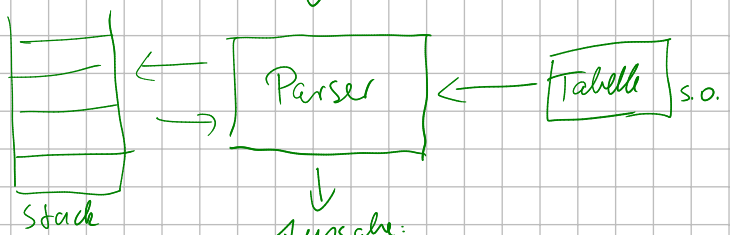
vom Scanner

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

while X ≠⊥

   if X = a
      a = next_token()

   else if X ∈ T
      error

   else if P[X, a] leer
      error

   else if P[X, a] = X → Y1Y2 ... Yk
      process_production(X → Y1Y2 ... Yk)
      push(Y1Y2 ... Yk) //Y1 = top of stack

   X = top of stack
```

**Abbildung 6:** Algorithmus zum tabellengesteuerten LL-Parsen

# LL-Parser:

Eingabe: Stream of tokens
Current_token()




Ausgabe:
AST, Symboltabellen-einträge, Fehlermeldungen

Anwenden der Regel
$S \Rightarrow AB$

# Wrap-Up

- 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