CFG, LL-Parser

BC George (FH Bielefeld)

Wiederholung

Endliche Automaten. reguläre Ausdrücke, reguläre Grammatiken, reguläre Sprachen

- Wie sind DFAs und NFAs definiert?
- Was sind reguläre Ausdrücke?
- Was sind formale und reguläre Grammatiken?
- In welchem Zusammenhang stehen all diese Begriffe?
- Wie werden DFAs und reguläre Ausdrücke im Compilerbau eingesetzt?

Motivation

Wofür reichen reguläre Sprachen nicht?

Für z. B. alle Sprachen, in deren Wörtern Zeichen über eine Konstante hinaus gezählt werden müssen. Diese Sprachen lassen sich oft mit Variablen im Exponenten beschreiben, die unendlich viele Werte annehmen können.

- $a^i b^{2*i}$ ist nicht regulär
- $a^i b^{2*i}$ für $0 \leq i \leq 3$ ist regulär
- Wo finden sich die oben genannten Konstanten bei einem DFA wieder?
- Warum ist die erste Sprache oben nicht regulär, die zweite aber?

Themen für heute

- PDAs: mächtiger als DFAs, NFAs
- kontextfreie Grammatiken und Sprachen: mächtiger als reguläre Grammatiken und Sprachen
- DPDAs und deterministisch kontextfreie Grammatiken: die Grundlage der Syntaxanalyse im Compilerbau
- Der Einsatz kontextfreier Grammatik zur Syntaxanalyse mittels Top-Down-Techniken

Einordnung: Erweiterung der Automatenklasse DFA, um komplexere Sprachen als die regulären akzeptieren zu können

Wir spendieren den DFAs einen möglichst einfachen, aber beliebig großen, Speicher, um zählen und matchen zu können. Wir suchen dabei konzeptionell die "kleinstmögliche" Erweiterung, die die akzeptierte Sprachklasse gegenüber DFAs vergrößert.

- Der konzeptionell einfachste Speicher ist ein Stack. Wir haben keinen wahlfreien Zugriff auf die gespeicherten Werte.
- Es soll eine deterministische und eine indeterministische Variante der neuen Automatenklasse geben.
- In diesem Zusammenhang wird der Stack auch Keller genannt.

Kellerautomaten (Push-Down-Automata, PDAs)

Def.: Ein Kellerautomat (PDA) $P = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ \perp,\ F)$ ist ein Quintupel mit:

Q:	eine endliche Menge von Zuständen
Σ:	eine endliche Menge von Eingabesymbolen
Γ:	ein endliches Kelleralphabet
δ:	die Übergangsfunktion
	$\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$
q_0:	der Startzustand
$\perp \in \Gamma$:	der anfängliche Kellerinhalt, symbolisiert den leeren Keller ($\perp$ = bottom)
$F \subseteq Q$:	die Menge von Endzuständen

Abbildung 1: Definition eines PDAs

Ein PDA ist per Definition nichtdeterministisch und kann spontane Zustandsübergänge durchführen.

Was kann man damit akzeptieren?

Strukturen mit paarweise zu matchenden Symbolen.

Bei jedem Zustandsübergang wird ein Zeichen (oder ϵ) aus der Eingabe gelesen, ein Symbol von Keller genommen. Diese und das Eingabezeichen bestimmen den Folgezustand und eine Zeichenfolge, die auf den Stack gepackt wird. Dabei wird ein Symbol, das später mit einem Eingabesymbol zu matchen ist, auf den Stack gepackt.

Beispiel

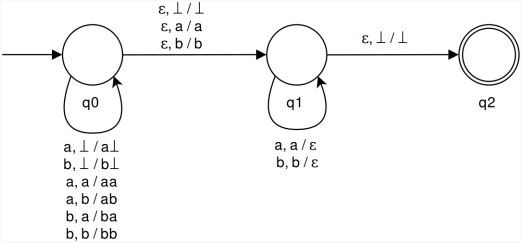


Abbildung 2: Ein PDA für $L = \{ww^R \mid w \in \{a, b\}^*\}$

Konfigurationen von PDAs

Def.: Eine Konfiguration (ID) eines PDAs 3-Tupel (q, w, γ) mit

- q ist ein Zustand
- w ist der verbleibende Input, $w \in \Sigma^*$
- γ ist der Kellerinhalt $\gamma \in \Gamma^*$

eines PDAs zu einem gegebenen Zeitpunkt.

Die Übergangsrelation eines PDAs

Def.: Die Relation $\vdash$ definiert Übergänge von einer Konfiguration zu einer anderen:

Sei $(p, \alpha) \in \delta(q, a, X)$, dann gilt $\forall w \ \epsilon \ \Sigma^*$ und $\beta \in \Gamma^*$:

$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$.

Def.: Wir definieren mit $\overset{*}{\vdash}$ 0 oder endlich viele Schritte des PDAs induktiv wie folgt:

- Basis: $I \overset{*}{\vdash} I$ für eine ID I.
- Induktion: $I \overset{*}{\vdash} J$, wenn $\exists$ ID K mit $I \vdash K$ und $K \overset{*}{\vdash} J$.

Eigenschaften der Konfigurationsübergänge

Satz: Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ ein PDA und $(q, x, \alpha) \overset{*}{\vdash} (p, y, \beta)$. Dann gilt für beliebige Strings $w \in \Sigma^*$, γ in Γ^*:

$(q, xw, \alpha\gamma) \overset{*}{\vdash} (p, yw, \beta\gamma)$

Satz: Sei $P = (Q, \Sigma, \Gamma, \gamma, q_0, \bot, F)$ ein PDA und $(q, xw, \alpha) \overset{*}{\vdash} (p, yw, \beta)$.

Dann gilt: $(q, x, a) \overset{*}{\vdash} (p, y, \beta)$

Akzeptierte Sprachen

Def.: Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ ein PDA. Dann ist die *über einen Endzustand* akzeptierte Sprache $L(P) = \{w \mid (q_0, w, \bot) \overset{*}{\vdash} (q, \epsilon, \alpha)\}$ für einen Zustand $q \in F, \alpha \in \Gamma^*$.

Def.: Für einen PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ definieren wir die über den *leeren Keller* akzeptierte Sprache $N(P) = \{(w \mid (q_0, w, \bot) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$.

Akzeptanzäquivalenzen

Satz: Wenn $L = N(P_N)$ für einen PDA P_N, dann gibt es einen PDA P_L mit $L = L(P_L)$.

Satz: Für einen PDA P mit ϵ-Transitionen existiert ein PDA Q ohne ϵ-Transitionen mit $L(P) = N(P) = L(Q) = N(Q)$.

Die Transitionsfunktion δ ist dann von der Form $\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma^*}$.

Deterministische PDAs

Def. Ein PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ ist *deterministisch* $:\Leftrightarrow$

- $\delta(q, a, X)$ hat höchstens ein Element für jedes $q \in Q, a \in \Sigma$ oder ($a = \epsilon$ und $X \in \Gamma$).
- Wenn $\delta(q, a, x)$ nicht leer ist für ein $a \in \Sigma$, dann muss $\delta(q, \epsilon, x)$ leer sein.

Deterministische PDAs werden auch *DPDAs* genannt.

Der kleine Unterschied

Satz: Die von DPDAs akzeptierten Sprachen sind eine echte Teilmenge der von PDAs akzeptierten Sprachen.

Die Sprachen, die von *regex* beschrieben werden, sind eine echte Teilmenge der von DPDAs akzeptierten Sprachen.

Kontextfreie Grammatiken und Sprachen

Kontextfreie Grammatiken

Def. Eine *kontextfreie (cf-)* Grammatik ist ein 4-Tupel $G = (N, T, P, S)$ mit N, T, S wie in (formalen) Grammatiken und P ist eine endliche Menge von Produktionen der Form:

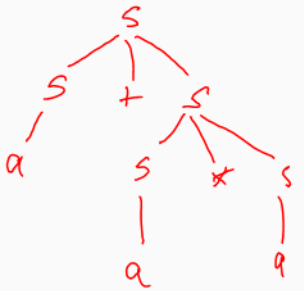
$X \to Y$ mit $X \in N, Y \in (N \cup T)^*$.

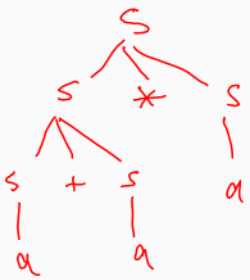
$\Rightarrow, \overset{*}{\Rightarrow}$ sind definiert wie bei regulären Sprachen. Bei cf-Grammatiken nennt man die Ableitungsbäume oft *Parse trees*.

Beispiel

$S \to a \mid S + S \mid S * S$

Ableitungsbäume für $a + a * a$:





$A \to ABC$

$B \to \ldots$

Die Reihenfolge der Ableitungen spielt keine Rolle für die Struktur des Baumes

Mehrdeutige Grammatiken

Def.: Gibt es in einer von einer kontextfreien Grammatik erzeugten Sprache ein Wort, für das mehr als ein Ableitungsbaum existiert, so heißt diese Grammatik *mehrdeutig*. Anderenfalls heißt sie *eindeutig*.

Satz: Es gibt kontextfreie Sprachen, für die keine eindeutige Grammatik existiert.

Kontextfreie Grammatiken und PDAs

Satz: Die kontextfreien Sprachen und die Sprachen, die von PDAs akzeptiert werden, sind dieselbe Sprachklasse.

Satz: Sei $L = N(P)$ für einen DPDA P, dann hat L eine eindeutige Grammatik.

Def.: Die Klasse der Sprachen, die von einem DPDA akzeptiert werden, heißt Klasse der *deterministisch kontextfreien (oder LR(k)-) Sprachen*

von links nach rechts lesen — Rückwärtsableitung

Das Pumping Lemma für kontextfreie Sprachen

Wenn wir beweisen müssen, dass eine Sprache nicht cf ist, hilft das Pumping Lemma für cf-Sprachen:

Satz: Sei L eine kontextfreie Sprache

$\Rightarrow \exists$ eine Konstante $p \in \mathbb{N}$:

$\underset{\substack{z \in L \\ |z| \geq p}}{\forall} \exists\ u, v, w, x, y \in \Sigma^*$ mit $z = uvwxy$ und

- $| vwx | \leq p$
- $vx \neq \epsilon$
- $\forall i \geq 0 : uv^i wx^i y \in L$

Abschlusseigenschaften von kontextfreien Sprachen

Satz: Die kontextfreien Sprachen sind abgeschlossen unter:

- Vereinigung
- Konkatenation
- Kleene-Hüllen L^* und L^+

Satz: Wenn L kontextfrei ist, dann ist L^R kontextfrei.

Entscheidbarkeit von kontextfreien Grammatiken und Sprachen

Satz: Es ist entscheidbar für eine kontextfreie Grammatik G,

- ob $L(G) = \emptyset$
- welche Symbole nach ϵ abgeleitet werden können
- welche Symbole erreichbar sind
- ob $w \in L(G)$ für ein gegebenes $w \in \Sigma^*$ Wortproblem

Satz: Es ist nicht entscheidbar,

- ob eine gegebene kontextfreie Grammatik eindeutig ist
- ob der Durchschnitt zweier kontextfreier Sprachen leer ist
- ob zwei kontextfreie Sprachen identisch sind
- ob eine gegebene kontextfreie Sprache gleich Σ^* ist

Abschlusseigenschaften deterministisch kontextfreier Sprachen

Satz: Deterministisch kontextfreie Sprachen sind abgeschlossen unter

- Durchschnitt mit regulären Sprachen
- Komplement

Sie sind nicht abgeschlossen unter

- Umkehrung
- Vereinigung
- Konkatenation

Syntaxanalyse

Syntax

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

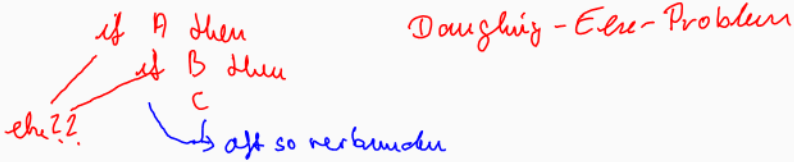
Syntaxanalyse ist die Bestimmung, ob Eingabedaten einer vorgegebenen Syntax entsprechen.

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

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 ∈ 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 3: Recursive Descent-Algorithmus

Grenzen des Algorithmus

Was ist mit

1) $X \to a\alpha \mid b\beta$ ✓
2) $X \to B\alpha \mid C\beta$ Die Terminalen Anfänge von B und C können helfen.
3) $X \to B\alpha \mid B\beta$ Linksfaktorisierung nötig
4) $X \to B\alpha \mid C\beta$ und $C \to B$ indirekte Linksfaktorisierung nötig
5) $X \to X\beta$ Linksrekursion entfernen
6) $X \to B\alpha$ und $B \to ~~X~~\beta$ indirekte Linksrekursion

} entfernen

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

Linksfaktorisierung

$X \to BC \mid BD$:

Entscheidung verschieben

$X \to BE$

$E \to C \mid D$

Ein neues Nichtterminal

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 4: 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 5: 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 ϵ-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 → X_j α by the productions

         X_i → β_1 α | β_2 α | ... | β_k α
         with X_j → β_1 | β_2 | ... | β_k are all the
            X_j-productions

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

Abbildung 6: Algorithmus zur Entfernung indirekter Linksrekursion

Arbeiten mit generierten Parsern: LL(k)-Grammatiken

First-Mengen

terminale Anfänge

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

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:

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

Linksableitungen

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 α abgeleitet wird.

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

$S \overset{*}{\Rightarrow} ABC \overset{*}{\Rightarrow} aaBC \overset{*}{\Rightarrow} aabC$

$\Rightarrow aabc$

Follow-Mengen

$D \to AB \mid B \mid AE \mid Aa$

$A \to \varepsilon$

$Follow_1(A) = First_1(B) \cup First_1(E) \cup \{a\}$

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

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

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

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

ε-Rückableitung

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:

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

und

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

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

$\alpha = \beta$

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 $((\beta \overset{*}{\Rightarrow}_l \epsilon \Rightarrow \alpha\neg \overset{*}{\Rightarrow}_l \epsilon))$
3) $((\beta \overset{*}{\Rightarrow}_l \epsilon)$ und $(\alpha \overset{*}{\Rightarrow}_l a\alpha_1)) \Rightarrow a \notin Follow(A)$
4) $((\alpha \overset{*}{\Rightarrow}_l \epsilon)$ und $(\beta \overset{*}{\Rightarrow}_l a\beta_1)) \Rightarrow a \notin Follow(A)$

¬ = nicht

1) und 2)bedeuten:

α und β können nicht beide ϵ ableiten, $First_1(\alpha) \cap First_1(\beta) = \emptyset$

3) und 4) bedeuten:

$(\epsilon \in First_1(\beta)) \Rightarrow (First_1(\alpha) \cap Follow_1(A) = \emptyset)$ $(\epsilon \in First_1(\alpha)) \Rightarrow (First_1(\beta) \cap Follow_1(A) = \emptyset)$

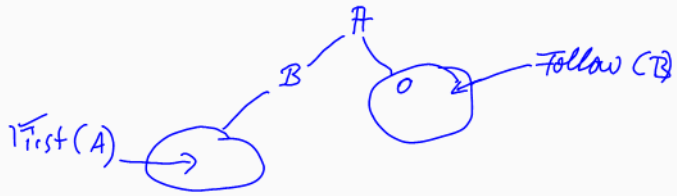
LL(1)-Grammatiken

$A \Rightarrow BC \quad B \Rightarrow a \mid b \mid c\,C \mid D \quad D \Rightarrow BF$

$First_1(B) = \{a, b, c\} \cup First_1(D)$

$Follow_1(B) = First_1(C) \cup First_1(F)$

$First_1(aB) = \{a\} \quad First_1(BC) = First_1(B)$ // bei $B \Rightarrow \varepsilon$ $First_1(C)$ dazunehmen



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 7: Algorithmus zur Generierung einer LL-Parsertabelle

Hier ist $\perp$ das Endezeichen des Inputs. Statt $\mathit{First}_1(\alpha)$ und $\mathit{Follow}_1(\alpha)$ wird oft nur $\mathit{First}(\alpha)$ und $\mathit{Follow}(\alpha)$ geschrieben.

LL-Parsertabellen

$S \to A \mid CaB$

$A \to BC$

$C \to d$

$B \to d$

$\vdots$

Terminale (= Token, Rückgabewerte des Lexers)

Nonterm.	a	b	c	d
S	$S \to A$	$S \to A$	—	$S \to CaB$
A	—	$A \to BC$	—	—

In der Tabelle steht welche Produktion genommen werden muss wenn das Nichtterminal in der ersten Spalte abgeleitet werden soll und das Terminal oben der Vorausschau gelesen wird

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.

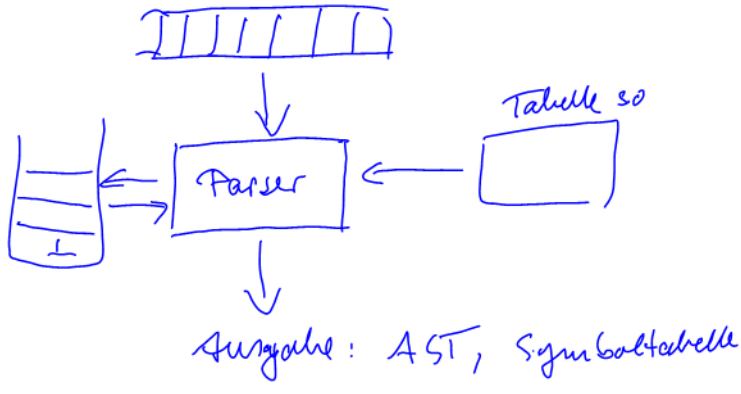
Algorithmus: Tabellengesteuertes LL-Parsen mit einem PDA

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

```
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 8: Algorithmus zum tabellengesteuerten LL-Parsen


Eingabe: Stream of Tokens
Tabelle so
Parser
Ausgabe: AST, Symboltabelle

Ergebnisse der Syntaxanalyse

while-Schleife → „w-Liste" ...

- eventuelle Syntaxfehler mit Angabe der Fehlerart und des -Ortes
- Fehlerkorrektur
- Format für die Weiterverarbeitung:
  - Syntaxbaum oder Parse Tree
  - abstrakter Syntaxbaum (AST): Der Parse Tree ohne Symbole, die nach der Syntaxanalyse inhaltlich irrelevant sind (z. B. ;, Klammern, manche Schlüsselwörter, ...)
- Symboltabelle

Wrap-Up

- Syntaxanalyse wird mit deterministisch kontextfreien Grammatiken durchgeführt.
- Eine Teilmenge der dazu gehörigen Sprachen lässt sich ~~bottom-up~~ 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

Symboltabelle

LICENSE