CFG kontextfreie Grammatiken

BC George (HSBI)

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 Variablen bei einem DFA wieder?
- Warum ist die erste Sprache oben nicht regulär, die zweite aber?

Themen für heute

Push down automata, Kellerautomaten

- 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 Grammatiken 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) $P = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ \bot,\ F)$ ist ein Septupel mit:

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

Abbildung 1: Definition eines PDAs

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

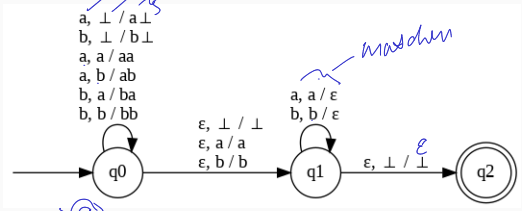
Was kann man damit akzeptieren?

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. Soll das automatisch vom Stack genommene Symbol auf dem Stack bleiben, muss es wieder gepusht werden.

Beispiel

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



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.

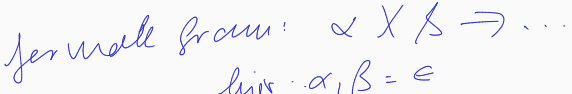
Der kleine Unterschied

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:

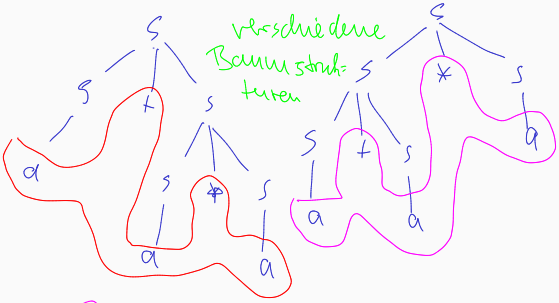
$X \to Y$ mit $X \in N, Y \in (N \cup T)^*$.

$\Rightarrow, \overset{*}{\Rightarrow}$ 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 $a + a * a$:



Das können wir nicht gebrauchen

Achtung:

$S \to A\,B$

$A \to a$

$B \to b$

egal ob ein A oder B ausgeleitet wird

(Links bzw. Rechtsableitung)

Nicht jede kontextfreie Grammatik ist eindeutig

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*.

es gibt keinen Algorithmus

Satz: Es ist nicht entscheidbar, ob eine gegebene kontextfreie Grammatik eindeutig ist.

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: Eine von einem DPDA akzeptierte Sprache hat eine eindeutige Grammatik.

Vorgehensweise im Compilerbau: Eine Grammatik für die gewünschte Sprache definieren und schauen, ob sich daraus ein DPDA generieren lässt (automatisch).

Syntaxanalyse

Was brauchen wir für die Syntaxanalyse von Programmen?

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

Syntax

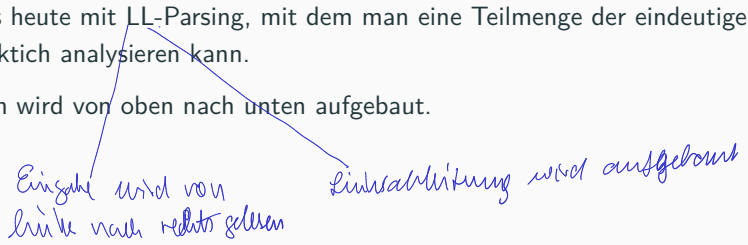
Wir verstehen unter Syntax eine Menge von Regeln, die die Struktur von Daten (z. B. Programmen) bestimmen.

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

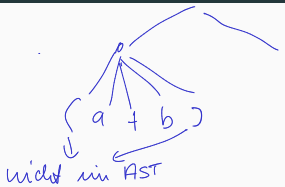
Diese vorgegebene Syntax wird im Compilerbau mit einer kontextfreien Grammatik beschrieben und mit einem sogenannten **Parser** analysiert.

Wir beschäftigen uns heute mit LL-Parsing, mit dem man eine Teilmenge der eindeutigen kontextfreien Grammatiken syntaktisch analysieren kann.

Der Ableitungsbaum wird von oben nach unten aufgebaut.



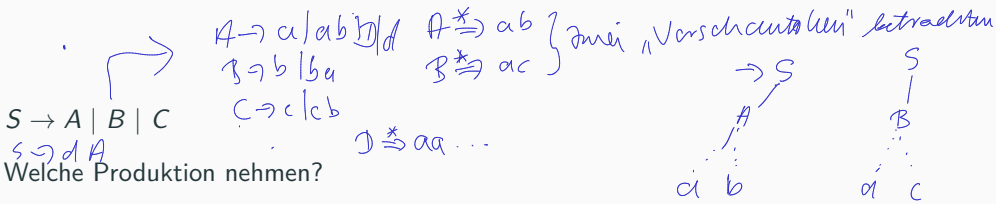
Ziele der Syntaxanalyse



- 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.

LL(k)-Grammatiken

First-Mengen



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

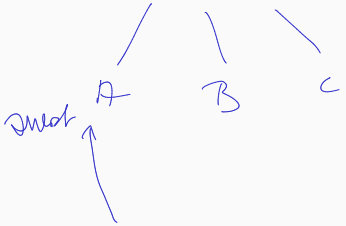
Welche Produktion nehmen?

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:

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

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

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:

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

und

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

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

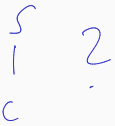
$\alpha = \beta$

LL(1)-Grammatiken

$S \to AB \mid c$

$A \to \varepsilon \mid b$

oder



?

Def: Follow-Mengen.

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

$\text{Follow}_k(\beta) = \{ w \in T^* \mid \exists \alpha, \gamma \in (N \cup T)^* \text{ mit } S \overset{*}{\Rightarrow}_\ell \alpha \beta \gamma \text{ und } w \in \overline{\text{First}}_k(\gamma) \}$

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 (hier: ANTLR), deren Eingabe eine LL(k)- (meist LL(1)-) Grammatik ist, und die als Ausgabe den Quellcode eines (effizienten) tabellengesteuerten Parsers generieren.

Was brauchen wir zur Erzeugung eines LL(k)-Parsers?

- eine *LL(k)*-Grammatik
- die $First_k$-Mengen der rechten Seiten aller Produktionsregeln
- die $Follow_k$-Mengen aller Nichtterminale und der rechten Seiten aller Produktionsregeln (s.u.)
- das Endezeichen $\perp$ hinter dem Eingabewort

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

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

Follow-Mengen geben an, welche Terminale im Parse Tree hinter einem Nichtterminal oder der rechten Seite einer Produktion kommen können.

Beispiel: First- und Follow-Mengen

P:
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to id \mid (id)$

$First_1(F) = First_1(T) = First_1(E) = \{(, id\}$
$First_1(E') = \{+, \underline{\varepsilon}\}$
$First_1(T') = \{*, \underline{\varepsilon}\}$

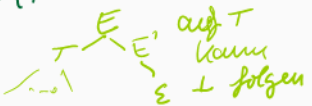
alle möglichen terminalen Anfänge der Nichtterminale

$G = (\{E, E', T, T', F\}, \{+, *, id, (,)\}, P, E)$

← "Aus dem Diagramm stabil" zum Parsen von mathematischen Ausdrücken

$Follow_1(E) = Follow_1(E') = Follow_1(TE') = Follow_1(+TE') = \{), \underline{\perp}\}$
$Follow_1(T) = Follow_1(T') = Follow_1(FT') = Follow_1(*FT) = \{+,), \underline{\perp}\}$
$Follow_1(F) = \{+, *,), \underline{\perp}\}$

alle möglichen terminalen Zeichen, die im Ableitungsbaum folgen können.



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[X, ⊥]
```

Abbildung 2: Algorithmus zur Generierung einer LL-Parsertabelle

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

LL-Parsertabellen

Tabelle für
$E \to TE'$ $E' \to +TE' \mid \varepsilon$ $T \to FT'$ $T' \to *FT' \mid \varepsilon$ $F \to (E) \mid id$

Nicht-terminale	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

LL-Parsertabellen

Rekursive Programmierung bedeutet, dass das Laufzeitsystem einen Stack benutzt. 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

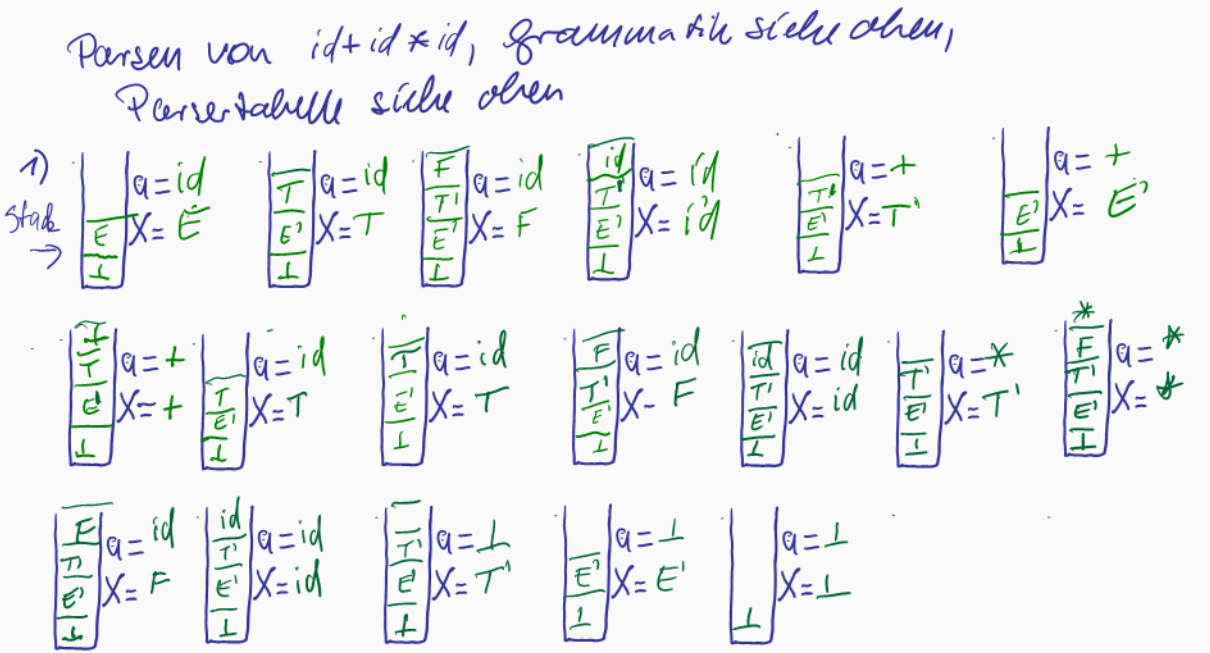
Eingabe: Eine Grammatik G = (N, T, P, S), eine Parsertabelle P mit "w" als initialem Kellerinhalt

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 → Y1 Y2 ... Yk
      process_production(X → Y1 Y2 ... Yk)
      push(Y1 Y2 ... Yk) //Y1 = top of stack

   X = top of stack
```

Abbildung 3: Algorithmus zum tabellengesteuerten LL-Parsen

Beispiel: LL-Parsen



Ergebnisse der Syntaxanalyse

- eventuelle Syntaxfehler mit Angabe der Fehlerart und des -Ortes
- Fehlerkorrektur
- Format für die Weiterverarbeitung:
  - Ableitungsbaum oder 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

Das sollen Sie mitnehmen

- Die Struktur von gängigen Programmiersprachen lässt sich nicht mit regulären Ausdrücken beschreiben und damit nicht mit DFAs akzeptieren.
- Das Automatenmodell der DFAs wird um einen endlosen Stack erweitert, das ergibt PDAs.
- Kontextfreie Grammatiken (CFGs) erweitern die regulären Grammatiken.
- Deterministisch parsebare Sprachen haben eine eindeutige kontextfreie Grammatik.
- Es ist nicht entscheidbar, ob eine gegebene kontextfreie Grammatik eindeutig ist.
- Syntaxanalyse wird mit deterministisch kontextfreien Grammatiken durchgeführt.
- Eine Teilmenge der dazu gehörigen Sprachen lässt sich top-down parsen.
- 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