LL-Parser

BC George (HSBI)

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?

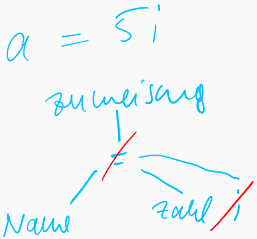
- einen Grammatiktypen, aus dem sich manuell oder automatisiert ein Programm zur deterministischen Syntaxanalyse erstellen lässt
- einen Algorithmus zum sog. Parsen von Programmen mit Hilfe einer solchen Grammatik

Themen für heute

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

Syntaxanalyse

Ziele der Syntaxanalyse



- Auffinden von Syntaxfehlern mit möglichst genauer Fehlerangabe !
- evtl. Vorschläge zur Fehlerbehebung
- Erstellen eines AST (abstract parse trees) zur semantischen Analyse = Ableitungsbaum ohne strukturell überflüssige Token (Semikolons, geschweifte Klammern, . . .)

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 $S \overset{*}{\Rightarrow} aAb$. 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.

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 ∈ N, a ∈ T
    for a production, X → Y1 Y2 ... Yn
        for i = 1 to n
            if Yi ∈ N
                RecursiveDescent(Yi, a)
            else if Yi ≠ a
                error
```

Abbildung 1: Recursive Descent-Algorithmus

Wann scheitert das Verfahren?

$A \to A\beta \mid \gamma$

Linksrekursion in der Grammatik führt zu einer unendlichen Rekursion

$\Rightarrow$ Es gibt Algorithmen zur Elimination von Linksrekursion

$\Rightarrow$ selbst herausfinden

$A \to B\beta \mid \gamma \ldots$

$B \to A\delta \mid \ldots$

indirekte Linksrekursion

$\Rightarrow$ Algorithmen

Tabellengesteuerte Parser: LL(k)-Grammatiken

First-Mengen

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

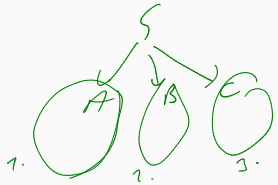
Welche Produktion nehmen?

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

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

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

Linksableitungen



Folge von Produktionen (Ableitungsregel)

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

1. b $S \overset{*}{\Rightarrow}_l abA$

Follow-Mengen

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

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

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

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

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:

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

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:

$\alpha = \beta$

w ist schon vorarbeitet beim Parsen, das nächste Token bestimmt eindeutig die Ableitung

LL(1)-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 $(\beta \overset{*}{\Rightarrow}_l a\beta_1)) \Rightarrow a \notin \mathit{Follow}(A)$

zum Verständnis

Die ersten beiden Zeilen bedeuten:

α und β können nicht beide ϵ 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(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.

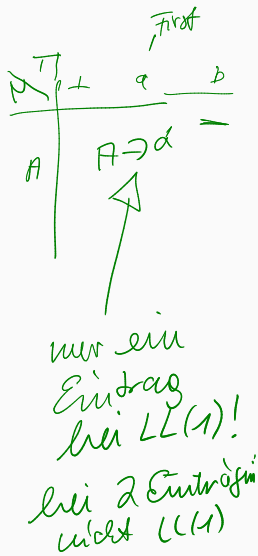
Algorithmus: Konstruktion einer LL-Parsertabelle

Eingabe: Eine Grammatik G = (N, T, P, S) mit $\perp \in T$ als Endezeichen

Ausgabe: Eine Parsertabelle P

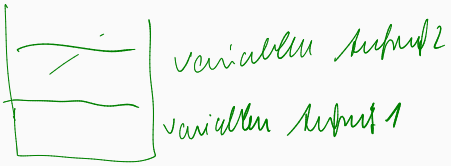
```
for each production X → α
    for each a ∈ First(α)
        add X → α to P[X, a]

    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



Statt $First_1(\alpha)$ und $Follow_1(\alpha)$ wird oft nur $First(\alpha)$ und $Follow(\alpha)$ geschrieben.

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
```

Produktion anwenden

z.B. AST bauen

Abbildung 3: Algorithmus zum tabellengesteuerten LL-Parsen

Wrap-Up

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 (AST).