Die Anwendung kontextfreier Grammatiken

Wiederholung: kontextfreie Grammatik

Kontextfreie Grammatiken sind **4-Tupel** der Form G=(∑, V, S, P) mit

∑ : Terminalsymbole

V : Nichtterminalsymbole (Variablen)

S ∈ V : Startvariable

P : Produktionen der Form : V x (V ∪∑)*

Sie erzeugen die Sprache L(G) = {w ∈ ∑* | *w lässt sich aus G ableiten* }

Unter dem Begriff „**Ableiten**“ versteht man die Erzeugung eines Wortes (Satzes) aus einer Grammatik. Während dieses Prozesses werden, beginnend beim Startsymbol, passende Produktionen ausgeführt. Solange noch Nichtterminalsymbole enthalten sind, spricht man von einer „**Satzform**“. Sind nur noch Terminalsymbole enthalten, von einem **Word** oder **Satz**.

Beim Ableiten wird zwischen ***Rechtsableitungen*** (in der Satzform wird immer das am weitesten rechts stehende Symbol ersetzt) und ***Linksableitungen*** (in der Satzform wird immer das am weitesten links stehende Symbol ersetzt) unterschieden.
Der Prozess des Ableitens ist nicht deterministisch.

Zwei Grammatiken heißen ***äquivalent***, wenn sie die gleiche Sprache erzeugen.

Eine Grammatik heißt ***eindeutig***, wenn es zu jedem Wort in L(G) genau eine Ableitung gibt, sonst ist die Grammatik ***mehrdeutig***. Existieren nur mehrdeutige Grammatiken für eine Sprache, so wird die Sprach ***inhärent mehrdeutig*** bezeichnet.

Kontextfreie Grammatiken im Compilerbau

Unter dem Begriff **Compiler** versteht man ein Computerprogramm, das in einer bestimmten Programmiersprache geschriebene Programme in eine vom Computer ausführbare Form, also in Assemblersprache übersetzt.

Während des Compilierungsprozesses werden verschiedene Phasen durchlaufen:

- In der ersten Phase wird die lexikalische Analyse durchgeführt. D.h. der Eingabestring oder Quelltext wird auf lexikalische Korrektheit geprüft und in sogenannte „**Tokens**“ übersetzt.
- In der zweiten Phase des Compilierungsprozesses, der Syntaxanalyse, wird mit Hilfe kontextfreier Grammatiken kontrolliert, ob es sich bei dem vom „**Lexer**“ gelieferten Tokenstrom um ein, in der Grammatik der Quellsprache, korrektes Programm handelt.

Es werden hierbei kontextfreie Grammatiken verwendet, da diese die wesentlichen programmiersprachlichen Konstrukte gut definieren können, wie zum Beispiel die inhärent rekursive Struktur der Programmiersprachen, die korrekten Klammerausdrücke und andere passende Anfangs- und Endmarkierungen.

In der Phase der Syntaxanalyse, welche auch als „***Parsing***“ bezeichnet wird, wird ein, zur Eingabe passender, ***Syntaxbaum*** erstellt. Man unterscheidet dabei im Wesentlichen zwischen zwei verschiedenen Methoden:

1. Top-Down-Syntaxanalyse

Bei dieser Methode wird der Syntaxbaum von der Wurzel zu den Blättern hin erzeugt. D.h. es wird versucht, beginnend beim Startsymbol eine Linksableitung für den Eingabestring zu finden.

Dies kann durch einen prädiktiven Parser zum einen mit einer Recursive-Descent-Methode oder zum anderen mit einem tabellengesteuerten Verfahren realisiert werden.

Bei der **Recursive-Descend-Methode** wird mit einem (manchmal aber auch mit mehreren) sogenannten *„**Lookahead-Symbol**(en)“* gearbeitet. Diese dienen dazu, die richtige Produktion auszuwählen, um vom Startsymbol zum Eingabestring zu kommen.

Es gibt dabei in der Phase der Erstellung des Syntaxbaums zwei Pointer:

- Der erste Pointer zeigt auf das aktuelle Eingabezeichen im Eingabestring, welches als Lookahead-Symbol verwendet wird.
- Der zweite Pointer zeigt auf den aktuellen Knoten im Baum.

Zeigt Pointer 2 auf ein Terminalsymbol, so wird dieses mit dem vergleichen, auf das Pointer 1 zeigt. Sind sie gleich, werden beide Pointer einen Knoten bzw. ein Zeichen weitergerückt, sonst wird ein „Error“ ausgegeben. Zeigt Pointer 2 auf ein Nichtterminalsymbol, so wird dieses unter Beachtung des Lookahead-Symbols, auf das Pointer 1 zeigt, durch eine Produktion ersetzt und Pointer 2 auf den ersten Kindknoten weiter versetzt.

Beim **tabellengesteuerten Verfahren**, wird ein Kellerautomat verwendet, in dessen Stack zu Beginn das Startsymbol liegt, das im Eingabepuffer den Eingabestring enthält und welcher eine Parsetabelle besitzt, die ihm zur Auswahl der Produktionen dient.

Solange der Stapel nicht leer ist, wird das oberste Stapelsymbol mit dem aktuellen Eingabezeichen verglichen:

- Ist es ein Terminalsymbol, dann wird es für den Fall, dass es mit dem Eingabezeichen übereinstimmt, vom Stapel gelöscht und im Eingabestring ein Zeichen weitergegangen, sonst wird ein Fehler ausgegeben.
- Handelt es sich um ein Nichtterminalsymbol, wird in der Parsetabelle beim Eintrag [Nichtterminal, aktuelles Eingabezeichen] die auszuführende Produktion nachgeschlagen, das Nichtterminal vom Stapel entfernt, die rechte Seite der Produktion auf den Stapel gelegt und die Produktion ausgegeben. Ist der Eintrag in der Parsetabelle leer, so liegt ebenfalls ein Fehler vor.

Beide Methoden sind für Grammatiken geeignet, die zur Klasse der ***LL(k)-Grammatiken*** gehören. LL(k) steht für das Lesen der Eingabe von **L**inks nach rechts, das Erstellen einer **Li**nksableitung und **k** verwendete Lookahead-Symbole.

> LL(k) Grammatiken sind im Wesentlichen eindeutige und nicht linksrekursive kontextfreie Grammatiken.

2. Bottom-Up-Syntaxanalyse

Bei der Bottom-Up-Syntaxanalyse, wird der Syntaxbaum beginnend bei den Blättern bis hin zur Wurzel erstellt.
Es handelt sich um eine weiter verbreitete Methode zur Syntaxanalyse als die Top-Down-Syntaxanalyse, da sich eine größere Klasse von Grammatiken damit parsen lässt.

Eine Form der Bottom-Up-Syntaxanalyse ist das **Shift-Reduce-Verfahren**, dessen Ziel darin besteht, den Eingabestring durch die Reduktion auf das Startsymbol zurückzuführen. Es entsteht dabei eine umgekehrte Rechtsableitung des Eingabestrings.

> Die **Schwierigkeit** des Verfahrens besteht darin, zu entscheiden, welcher Substring reduziert werden muss. Ein Substring, der der rechten Seite einer Produktion entspricht und dessen Reduktion zum Nichtterminalsymbol auf der linken Seite dieser Produktion einem Schritt der umgekehrten Rechtsableitung entspricht, wird als ***„Handle"*** bezeichnet. Die Reduktion der *Handles* von links nach rechts wird als ***Handle-Pruning*** (dt.: Beschneidung/Stutzen) bezeichnet.

Ein **Shift-Reduce-Parser**, der meist die Implementierung eines Kellerautomaten ist, kennt vier Aktionen: schieben, reduzieren, akzeptieren und Fehler erkennen. Zu Beginn der Analyse ist der Stack leer und im Eingabepuffer liegt der Eingabestring. Es werden dann so lange Symbole aus dem Eingabestring auf den Stack gelegt, bis ein Handle oben aufliegt, welches dann reduziert wird.

> **Probleme** die dabei auftreten können, sind der Shift-Reduce- und der Reduce-Reduce-Konflikt, welche auf den Problemen beruhen, dass manchmal nicht klar ist, ob reduziert oder geschoben werden soll, oder durch welche Produktion das Handle ersetzt werden soll.

Diese Probleme treten bei der ***LR(k)-Syntaxanalyse*** nicht auf. Bei der LR(k)-Analyse, wobei das L für das Lesen von **L**inks nach rechts, das R für das Erstellen einer **R**echtsableitung und das **k** für die Anzahl der Lookahead-Symbole steht, wird ähnlich wie bei der LL(k)-Syntaxanalyse ein Kellerautomat verwendet.
Der **Unterschied** besteht allerdings darin, dass in der LR(k)-Syntaxanalyse auch Zustände und nicht nur Grammatiksymbole auf dem Stack gespeichert werden und die Parsetabelle zweigeteilt ist - in einen Aktions- und einen Sprung- Teil. Die Zustände geben in diesem Fall Aufschluss darüber *wann*, *ob* und *wie reduziert* oder *geschoben* werden soll.

Da die Erstellung eines LR(k)-Parsers per Hand sehr aufwändig und mit einem hohen Fehlerrisiko verbunden ist, gibt es **Parsergeneratoren**, die bei Eingabe einer LR(k)-Grammatik für uns einen Parser automatisch generieren.
Ein Beispiel dafür ist **Yacc** (**y**et **a**nother **c**ompiler **c**ompiler). Ein Yacc-Programm ist dabei meistens wie folgt aufgebaut:

```
Deklaration                 //z.B. Variablen werden deklariert
%%
Übersetzungsregeln          //die Produktionen der Grammatik und damit assoziierte
%%                            Aktionen werden festgelegt
Unterstützende C-Routine    //eine C-Routine zur lexikalischen Analyse wird definiert
```

Kontextfreie Grammatiken in Markup-Sprachen

Unter dem Begriff „**Markup-Sprache**“ (dt.: Auszeichnungssprache) werden Sprachen, die für inhaltliche Gliederung und/oder Formatierung von Texten und anderen Daten verwendet werden zusammengefasst. So genannte ***„Tags“*** geben dabei an, wie das Erscheinungsbild eines Textes sein soll, oder welche Bedeutung der Text hat.

Eine der bekanntesten Markup-Sprachen ist ***HTML*** (**H**yper **T**ext **M**arkup **L**anguage). In HTML werden die Tags (dargestellt durch spitze Klammern) größtenteils verwendet, um die Formatierung des Textes festzulegen.

Bsp.:

 FU Berlin => FU Berlin (Link)

 Erstens () => 1. Erstens (Liste)

 Zweitens () => 2. Zweitens

Eine kontextfreie Grammatik, die einige Elemente von HTML darstellen kann, könnte wie folgt aussehen:

Char → a|A|…
Text → ε | Char Text
Doc → ε | Element Doc
Element → Text | Doc ( ) | List | …
ListItem → Doc ()
List → ε | ListItem List

Neben HTML gibt es noch **XML** (E**x**tensible **M**arkup **L**anguage), bei der es um die strukturelle Darstellung von Daten geht, also weniger das Format als die Semantik des Textes im Vordergrund steht.
In **DTD** (**D**ocument-**T**ype-**D**efinition) zum Beispiel, werden mit Hilfe eines Satzes an Regeln Dokumente eines bestimmten Typs deklariert, z.B. Telefonlisten oder Studentendatein.

Üblicherweise wird ein DTD-Dokument mit folgender Zeile begonnen:

<!DOCTYPE Name-der-DTD [Liste der Elementdefinitionen]>

Der Struktur eines DTD-Dokuments liegt dabei wieder eine kontextfreie Grammatik zugrunde, wie z.B. im Beispiel:

<!ELEMENT PCS (PC*)>	PCS → ε | PC PCS
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, DISK+)>	PC → Model Price Processor Disks
<!ELEMENT PROCESSOR (MODEL, SPEED)>	Processor → Model Speed
<!ELEMENT DISK (HARDDISK | CD)>	Disks → Disk | Disk Disks