Die Anwendungen von Kontextfreien Grammatiken (kfGs)

von Marco Träger (4130515) zum Proseminar Theoretische-Informatik WS0708

Die Analyse von Sprachen spielt vor allem im Compilerbau und sonstigen Computer-Anwendungen, die auf der Basis von Sprachen operieren eine wichtige Rolle.
Da Computersprachen und Beschreibung von Datenstrukturen in der Regel immer eindeutig sein sollen, damit kontextfrei, bietet es sich an dafür kfGs und PDAs zu verwenden. Es hat sich sogar gezeigt, das die meisten dieser Anwendungsgebiete nur det. PDAs benötigen. Dieser Vortrag soll als Einführung in dieses Thema dienen.

Vereinfachter Ausschnitt aus der C-Syntax in Backus-Naur-Form

```
<expression> ::= <assignment-expression>
               | <expression> , <assignment-expression>

<assignment-expression> ::= <additive-expression>
                          | <unary-expression> <assignment-operator> <assignment-expression>

<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>
                     | <unary-operator> <cast-expression>
                     | sizeof <unary-expression>
                     | sizeof <type-name>

<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>


<cast-expression> ::= <unary-expression>
                    | ( <type-name> ) <cast-expression>

<postfix-expression> ::= <primary-expression>
                       | <postfix-expression> [ <expression> ]
                       | <postfix-expression> ( {<assignment-expression>}* )
                       | <postfix-expression> . <identifier>
                       | <postfix-expression> -> <identifier>
                       | <postfix-expression> ++
                       | <postfix-expression> --


<primary-expression> ::= <identifier>
                       | <constant>
                       | <string>
                       | ( <expression> )

<constant> ::= <integer-constant>
             | <character-constant>
             | <floating-constant>
             | <enumeration-constant>


<unary-operator> ::= & | * | + | - | ~ | !

<assignment-operator> ::= = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=
```

- Fett gedruckte **Nichtterminale** werden während der lexikalischen Analyse vom C-Lexer produziert.
- Der Einfach halt halber, sind im Gegensatz zum korrekten C-Syntax wird bei der Regel <type-name> hier abgebrochen
- <assignment-expression> der Einfachheit halber geändert

•

Lexer - lexikalischer Analysator

Wie im oberen Beispiel gut zu erkennen gibt es in einer kfG viele Produktionen die auch äquivalent durch reguläre Ausdrücke ersetzt werden können.

BSP: Wir betrachten die Sprach die durch die kontextfreie Grammatik G gebildet wird.

G = ({<integer-constant>, <digit>}, {0,1,...,9}, P, <integer-constant>)

mit P (in EBNF):

```
<integer-constant> ::= <digit> <integer-constant> | <digit>
<digit>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

So ist die von G erzeugte Sprach L(G) äquivalent zur Sprache die vom regulären Ausdruck R erzeugt wird

R = {0,1,...,9} {0,1,...,9}*

Wir erkennen also das man einige Produktionen von kfGs mit regulären Ausdrücken erzeugen kann. Diese Verfahren eignet sich besonders gut um Produktionen zu sparen und so der PDA zu vereinfachen. Ebenfalls macht es auch für die weitere Verarbeitung (Semantische Analyse) Sinn, zB. eine Zahl als ein zusammengehöriges Symbol in der Sprache zu identifizieren.

Parser – syntaktischer Analysator

Def: Parser

Deutsch: Zerteiler. Hierbei handelt es sich um einen modifizierten Kellerautomaten, der für eine gegebenen Grammatik erstellt wird. Im Gegensatz zum Kellerautomaten arbeitet der Parser zusätzlich über eine Syntaxanalysetabelle die über die gegebene Grammatik erstellt wurde. Diese ermöglicht eine einfachere maschinellen Bearbeitung.
Als Ausgabe liefern Parser meist Syntaxbäume, in denen die Eingabe entsprechend der Grammatik angeordnet ist.

Def: k-Lookahead

Bei einem k-Lookahead handelt es sich um eine Datenstruktur die ermöglicht k Eingabesymbole voraus zu lesen um damit eindeutige(deterministische) Entscheidungen treffen zu können.

Def: LL(k)-Parser

LL(k) steht für scaning form left, using left productions, k symbols lookahead

Es handelt es sich um einen Top-Down-Parser der über die nächsten k-Eingabesymbole eindeutig entscheiden kann welche Produktion er anwenden muss.
Er ist in der Lage alle Sprachen zu erkennen die durch eine entsprechende LL(k)-Grammatik gebildet werden.

Def: LR(k)-Parser

LR(k) steht für scaning form left, using right productions, k symbols lookahead

Es handelt es sich um einen Bottom-Up-Parser der über die nächsten k-Eingabesymbole eindeutig entscheiden kann welche Produktion er anwenden muss.
Er ist in der Lage alle Sprachen zu erkennen die durch eine entsprechende LR(k)-Grammatik gebildet werden.

Sprach-Hierarchie von kfG

$$\mathcal{L}(\mathrm{LL}(1)) \subsetneq \mathcal{L}(\mathrm{LL}(2)) \subsetneq \cdots \subsetneq \mathcal{L}(\mathrm{LL}(k)) \subsetneq \mathcal{L}(\mathrm{LR}(1)) = \mathcal{L}(\mathrm{DPDA}) \subsetneq \mathcal{L}(\mathrm{PDA})$$
$$\mathcal{L}(\mathrm{LR}(0)) \subsetneq \mathcal{L}(\mathrm{LR}(1)) = \mathcal{L}(\mathrm{LR}(2)) = \cdots = \mathcal{L}(\mathrm{DPDA}) \subsetneq \mathcal{L}(\mathrm{PDA})$$

Wir erkennen also, das schon eine LR(1)-Grammatik genau die Sprachen beschreibt die ein det. PDA erkennen kann. Und daher schon ein Parser, der nur ein Symbol voraussehen kann, schon in der Lage ist alle det. kontextfreien Sprachen zu erkennen.
Es bietet sich also Aufgrund der Einfachheit des LR(1)-Parsers an, diesen für die verschiedensten Probleme zu verwenden.

Parsertabelle/Syntaxanalysetabelle

Wie schon erwähnt ist die Syntaxanalysetabelle ein wichtiger Bestandteil eines Parsers. Diese wird aus einer kfG erzeugt und bildet die Funktionsweise des Parsers ab.

1. E * B ← E
2. E + B ← E
3. B ← E
4. id ← B

Erzeugt die LR(1)-Syntaxanalysetabelle:

	Aktion				**GoTo**	
Zustand	*****	**+**	**id**	**$**	**E**	**B**
0			s1		2	3
1	r4	r4	r4	r4		
2	s4	s5		acc		
3	r3	r3	r3	r3		
4			s1			6
5			s1			7
6	r1	r1	r1	r1		
7	r2	r2	r2	r2		

Wobei in der Aktionstabelle:

s# : „gelesenes Zeichen auf den Stack und # auf den Zustandsstack

r# : „reduziere nach Regel #, bringe die diese Seite auf den Stack, lösche x Zustände vom Zustandsstack wobei x die Anzahl Nichtterminale auf der reduzierten Seite“

acc: „Akzeptieren“

Und in der Goto-Tabelle:

: springe in Zustand # wenn entsprechendes Symbol auf dem Stack liegt.

Parser- und Compilergeneratoren

Durch die gleiche Struktur denen alle Parser unterliegen, liegt es sehr nahe diese und ihrer Syntaxtabelle nicht selbst zu programmieren sondern sie von einem Parsergenerator anhand der von ihnen zu erkennen Grammatik generieren zu lassen.

Dabei wird die vom Parser zu erkennende Grammatik in einer Speziellen Sprache formuliert und vom Grammatik-Parser des Parsergenerators analysiert. Die Produktionsregeln werden erkannt um mit ihrer Hilfe die Syntaxtabelle generiert. Die restlichen Programmteile wie Ein- und Ausgabe werden ebenfalls automatisch erstellt.

Der Schritt vom Parsergenerator zum Compilergenerator gestaltet sich meist sehr einfach. Es werden gewöhnlich für jeder Produktion des Parsers semantische Aktionen gestimmt, also jeder Produktion wird eine Bedeutung zugewiesen. Damit kann dann zB. eine beliebiger Taschenrechner oder ein Programme zur Übersetzung von verschiedenen Dateiformaten generiert werden.

Beispiele für Compielergeneratoren sind zB: JavaCC, yacc

XML

XML bedeutet Extensible Markup Language, also „erweiterbare Auszeichnungssprache“.

XML wird genutzt um Daten eine Bedeutung und eine Hierarchie zuzuweisen, sie also ihren Kontext zu erhalten.

```
<ADDR>Takustraße 6</ADDR>
```

Einem XML-Dokument liegt immer eine Definition seiner Struktur zu Grunde, diese Struktur wird gewöhnlich in der Form einer DTD (Document-Type-Definition) getroffen.

```
<!DOCTYPE Book [
          <!ELEMENT BOOK (TITLE, AUTHOR+, CONTENTTABLE, CHAPTER+)
          <!ELEMENT AUTHOR \#PCDATA>
          <!ELEMENT CONTENTTABLE (TABLEENTRY*)>
          <!ELEMENT TABLEENTRY (TITLE, PAGE)>
          <!ELEMENT CHAPTER (TITLE, CONTENT)>
          <!ELEMENT TITLE \#PCDATA>
          <!ELEMENT NUMBER \#PCDATA>
          <!ELEMENT CONTENT \#PCDATA>
]>
```

Auf der Basis dieser Struktur kann können nun die Daten strukturiert werden.

```
<BOOK>
          <TITLE>Robinson Crusoe</TITLE>
          <AUTHOR>Daniel Defoe</AUTHOR>
          <CONTENTTABLE>
                    <TABLEENTRY>
                              <TITLE>Erstes Seeunglück</TITLE>
                              <PAGE>1</PAGE>
                    </TABLEENTRY>
                    ...
          </CONTENTTABLE>
          <CHAPTER>
                    <TITLE>Erstes Seeunglück</TITLE>
                    <CONTENT>Ich wurde...</CONTENT>
          </CHAPTER>
          <CHAPTER> ... </CHAPTER>
          <CHAPTER> ... </CHAPTER>
</BOOK>
```

Man erkennt an der Struktur der Daten also ihre Bedeutung und kann so leicht Daten zwischen Systemen und Programmen austauschen, da die Daten nicht ihren Kontext verlieren können.

DTD- und XML-Parser

An dem Beispiel XML kann man leicht erkennen wie Parser und Parsergeneratoren zusammenarbeiten.

Anhand der DTD wird über einen XML-Parsergenerator ein XML-Parser für das spezielle Problem generiert. Dieser XML-Parser ist dann in der Lage XML-Dokumente diese Dokumententyps syntaktisch zu analysieren.

Beispiele für XML-Parser(Generatoren): Xerces, Gnome XML-Parser, XPP, SimpleXML

Quellen:

Compilers – von A. V. Aho, R. Sethi, J. D. Ullman (ISBN 0-201-10088-6)
Einführung in die Automatentheorie – von Hopcroft, Motwani, Ullman; Auflage: 2, 2003.
Wikipedia – www.wikipedia.de
C-Syntax - http://www.cs.grinnell.edu/~stone/courses/languages/C-syntax.xhtml