1 BTRACC-Ein Parsergenerator auf der Basis eines Backtracking-Verfahrens

Uwe Erdmenger
pro et con, Innovative Informatikanwendungen GmbH, Annaberger Straße 240, 09125 Chemnitz
Uwe.Erdmenger@proetcon.de

Einige Besonderheiten der syntaktischen Analyse von Quellcode im Reengineering-Bereich sind mit den bekannten Parsergeneratoren nur schwer realisierbar. Insbesondere der Wunsch, sehr schnell brauchbare Analysewerkzeuge erstellen zu können, war der Anlaß für die Entwicklung des neuen Parsergenerators *BTRACC*.

Der folgende Beitrag beschreibt die grundlegende Arbeitsweise des Werkzeugs. Ausgehend von der bekannten Erweiterten Backus-Naur-Form (EBNF) wird die Generierung des Parsers und seine Funktionalität beschrieben, wobei auch auf die Frage der Attributierung eingegangen wird.

1.1 Einleitung und Motivation

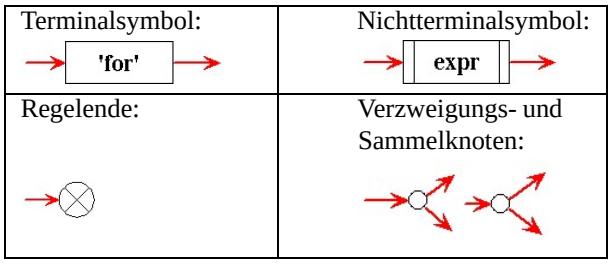
Bei der Erstellung von Analysewerkzeugen für Reengineering-Zwecke liegt die Grammatik meist in einer Form vor, welche für bekannte Parsergeneratoren (YACC, PCCTS, COCO/R; ...) nicht optimal geeignet ist. Insbesondere der große Sprachumfang und der Wunsch, mehrere Dialekte mit einem Parser bearbeiten zu können, sind die Ursache dafür. Außerdem stehen hier häufig Korrekturen und Erweiterungen an, welche bei oben genannten Werkzeugen zu größeren Umstellungen der vorhandenen Grammatik führen.

Um vorliegende Grammatiken ohne große Umstellung in Parsern verwenden und diese auch einfach pflegen zu können, wird bei der Erstellung von Parsern in der Firma *pro et con GmbH* ein eigener Parsergenerator **BTRACC** (BackTRACking Compiler Compiler) verwendet, dessen Funktionsweise im Weiteren beschrieben werden soll. Mit diesem Werkzeug wurden bereits ein SQL-, ein Natural- ein Java- und ein C-Parser realisiert, die auch in kommerziellen Produkten Verwendung finden.

1.2 Interne Darstellung der Grammatik

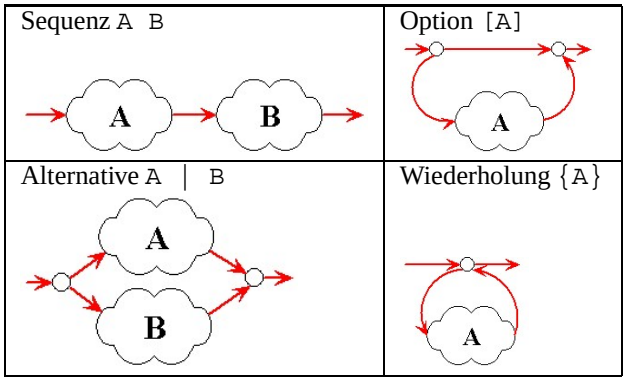
Ausgangspunkt ist die attributierte Grammatik der zu analysierenden Sprache in EBNF. Diese liest der Parsergenerator BTRACC, prüft ihre syntaktische Korrektheit und baut eine interne Darstellung auf.

Dabei handelt es sich um eine Datenstruktur, welche als Graph angesehen werden kann. Weil BTRACC in C++ geschrieben wurde, lag die Verwendung von dynamisch allokierten Strukturen als Knoten des Graphen und Zeigern als Kanten nahe. Dabei gibt es folgende Arten von Knoten:



Aus diesen Grundbausteinen kann die interne Repräsentation der Regeln zusammengesetzt werden. Dabei wird zu jeder Regel ein Graph aufgebaut, welcher die rechte Regelseite wiederspiegelt. Alle diese Regeln werden, sortiert nach ihrem Namen, in einer Hash-Liste verwaltet, welche einen schnellen Zugriff über den Regelnamen (Nichtterminalsymbol) gestattet.

Die Strukturen der Erweiterten Backus-Naur-Form werden wie folgt zusammengesetzt:



Dabei stehen `A` und `B` für Terminalsymbol-Knoten, Nichtterminalsymbol-Knoten oder wiederum komplette Strukturen (rekursive Definition).

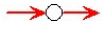
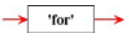
1.3 Generierung der Parser-Quellcodes

Die Graphen dürfen keine Zyklen ohne mindestens ein Terminalsymbol enthalten. Das führt zu einer Endlosschleife im Algorithmus. Dies wird vor der Generierung getestet. Praktisch bedeutet das, daß die Eingabegrammatik nicht linksrekursiv sein und keine Verschachtelung optionaler Konstrukte der Art `{ [ A ] }` enthalten darf.

Der zu generierende Parser-Quellcode besteht in der Hauptsache aus einem Feld von C-Strukturen, welche als Maschinenbefehle einer virtuellen Maschine angesehen werden können. Dabei wird zu jedem Knoten im Graphen ein solcher Befehl erzeugt. Er enthält auch die Kante zum nächsten/alternativen Knoten als Feldindex des ihm zugeordneten Befehls. Das ist möglich, da nach obiger Vorschrift ein Knoten maximal 2 Nachfolger haben kann. Folgende C-Struktur wird dabei verwendet:

```
struct BEFEHLE {
   int mnemonic; // Art des Befehls
   int next;     // naechster Befehl
   int alt;      // alternativer Befehl
   int token;    // bei Terminalen
};
```

Die folgenden Befehle werden generiert:

Befehl	Knoten	Befehl	Knoten
BRANCH		GOTO	
TEST		CALL	
RETURN			

1.4 Arbeitsweise der virtuellen Maschine

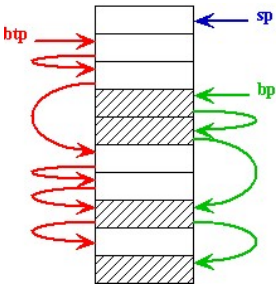
Die virtuelle Maschine verwaltet einen Stack mit zwei verschiedenen Arten von Stackframes:

1. **Entscheidungspunkte**
   Bei Alternativen (Befehl BRANCH) wird der erste Weg (Komponente `next` der Struktur `BEFEHLE`) weiter verfolgt, wohingegen der 2. Weg in diesen Entscheidungspunkten gemerkt wird. Sie beinhalten daher die folgenden Komponenten:
   - aktuelle Position des Scanners
   - Index des Stackframes des vorhergehenden Entscheidungspunktes
   - Index des beim Anlegen aktuellen Call-Frames
   - Index des ersten Befehls des alternativen Weges
2. **Call-Frames**
   Beim Auftreten von Nichtterminalsymbolen (Befehl CALL) müssen Informationen zum Weg nach dem Aufruf gespeichert werden (ähnlich wie beim Funktionsaufruf in Programmiersprachen). Dies sind:
   - Index des beim Aufruf aktuellen Call-Frames
   - Index des nachfolgenden Befehls im Feld der Befehle

Dazu werden 5 Variablen mit folgender Bedeutung verwaltet:

1. `ip` Index des aktuell abzuarbeitenden Befehls
2. `sp` Index des 1. freien Stackframes
3. `btp` Index des zuletzt angelegten Entscheidungspunktes
4. `bp` Index des zuletzt angelegten Call-Frames
5. `tok` Position des aktuellen Tokens als Index in eine lineare Liste aller Token

Die nebenstehende Skizze verdeutlicht diesen Sachverhalt.



Die Abarbeitung beginnt mit dem 1. Befehl der Regel, welche zum Startsymbol gehört. Dabei ist der Stack leer und `sp`, `btp`, `bp` und `tok` haben den Wert 0.

Die einzelnen Befehle bewirken folgendes:

1. **BRANCH**: Es wird ein neuer Entscheidungspunkt auf dem Stack abgelegt und dabei `btp` und `sp` aktualisiert. Es wird beim Befehl, auf den `next` verweist, weitergearbeitet (`ip=next`).
2. **GOTO**: Die Variablen und der Stack werden nicht geändert. Es wird beim Befehl, auf den `next` verweist, weitergearbeitet.
3. **CALL**: Es wird ein neuer Call-Frame auf dem Stack angelegt und dabei `sp` und `bp` aktualisiert. Dann wird beim 1. Befehl der gerufenen Regel weitergearbeitet, der in der `next`-Komponente gespeichert ist.
4. **RETURN**: Ist `bp!=0`, so wird `bp` aus dem aktuellen Call-frame zurückgestellt und beim dort angegebenen Befehl weitergearbeitet. Wichtig ist hier, daß dabei `sp` und `btp` nicht verändert werden. Der Stack wird also nicht verkleinert, damit die nach dem Aufruf der Regel eingerichteten Entscheidungspunkte im Backtracking-Fall auch erreicht werden können.
   Ist `bp==0`, so ist die Regel zum Startsymbol vollständig abgearbeitet und das Parsen erfolgreich beendet.
5. **TEST**: Falls das aktuelle Token gleich dem getesteten ist, so wird `tok` auf die Position des nächsten Tokens gestellt. Dann wird bei `next` weitergearbeitet.
   Sind beide Token nicht gleich und ist `btp!=0`, so wird Backtracking ausgelöst. Das heißt, `bp`, `btp` und `tok` werden aus dem stackobersten Entscheidungspunkt wieder hergestellt und `sp` wird auf `btp` gesetzt. Damit wird der Stack reduziert. Weitergearbeitet wird mit dem Befehl, welcher im stackobersten Entscheidungspunkt angegeben ist.
   Ist `btp==0`, so gibt es keine offene Alternative mehr und das Parsen wird ohne Erfolg beendet.

1.5 Berechnung der Attribute

Die Berechnung der Attribute (meist Syntaxbäume) ist im Backtracking-Fall nur schwer rückgängig zu machen. Daher erfolgt sie in einem separaten Durchlauf nach erfolgreichem Parsen. Die Besonderheit dabei ist, daß in diesem 2. Durchlauf noch der Stackinhalt zur Verfügung steht, der im 1. Durchlauf aufgebaut wurde. Dieser wird nun als eine Art „roter Faden“ verwendet, so daß der Durchlauf deterministisch erfolgen kann.

Zum Zwecke der Attributierung generiert BTRACC zu jeder Regel eine Funktion. Die Regelparameter sind die Parameter der Funktion und die Berechnung der Attribute geschieht innerhalb des Funktionsblocks. Diese Funktionen rufen sich gegenseitig auf. Dies ist analog zum Verfahren des rekursiven Abstiegs, welches auch von LL(1)-Parsergeneratoren, wie z.B. COCO/R verwendet wird.

Die Funktionen ändern die Variablen wie oben beschrieben, manipulieren aber den Stack nicht. Im Falle einer Verzweigung (wenn also ein alternativer Weg existiert), wird verglichen, ob dieser Weg im stackobersten Entscheidungspunkt steht und die dort angegebene Position in der Tokenliste der aktuellen Position entspricht. Ist dies der Fall, so führte in der Parsing-Phase der 1. Weg zum Ziel. Dieser ist nun auch hier zu wählen. Ist es nicht der Fall, so ist der alternative Weg der richtige.

Auf diese Weise kann in der Attributierungsphase der korrekte Weg deterministisch (ohne Backtracking) gefunden werden und eine aufwendige Rücknahme der Attributberechnung entfallen.