



Der Parsergenerator BTRACC2

Uwe Erdmenger

pro et con

Innovative Informatikanwendungen GmbH

11. Workshop Software-Reengineering
04.-06. Mai 2009, Bad Honnef



Eigenschaften von Legacy-Quellcode

- Sprachen mit umfangreicher Syntax (COBOL: 60 Seiten)
- Notation: Syntaxdiagramme, "Railroad-Diagramme" (gut lesbar, aber keine LL- bzw. LR-Eigenschaften, auch mehrdeutige Syntax)
- Mehrere Dialekte, verschiedene Versionen → häufige Anpassungen
- Eingebettete Systeme (SQL, CICS, DL/I, ...) – ebenfalls mit teilweise umfangreicher Syntax
- ➔ Analyse mit YACC, ANTLR, COCO, ... schwer realisierbar



Parsergenerator BTRACC2

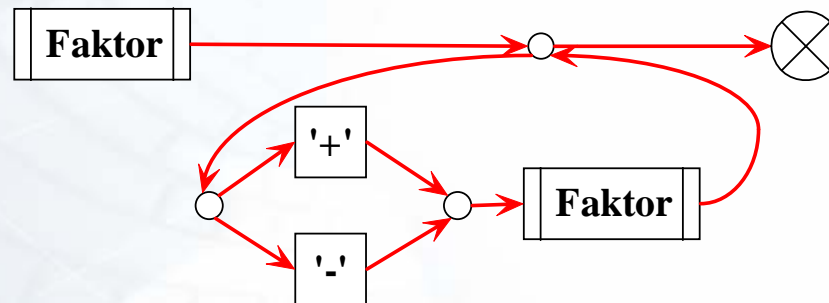
- **Ziel:**
 - Syntax möglichst ohne Umstellungen so verwenden, wie sie in der Dokumentation vorliegt

- **Grundlegende Eigenschaften:**
 - LL(1)-Parsergenerator
 - Backtracking zur Auflösung von LL(1)-Konflikten
 - Eingabe: EBNF mit Attributierung
 - Trennung von Parsing- und Attributberechnungsphase
 - Implementiert in C
 - Generiert C, C++, Java



Prinzipielle Arbeitsweise (1)

- Ausgangspunkt: Syntaxregeln in EBNF:
 $\text{Term} = \text{Faktor} \{ ('+' \mid '-') \text{Faktor} \}.$
- Einlesen und Aufbau eines Graphen:



- Knoten für Nichtterminale (Faktor), Terminalsymbole ('+'), Verzweigungs- und Sammelknoten
- Maximal zwei Nachfolger pro Knoten



Prinzipielle Arbeitsweise (2)

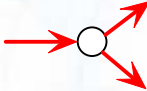



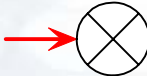
- Interne Verarbeitung des Graphen:
 - Test auf Zyklen ohne Terminalsymbol (auch über mehrere Regeln hinweg) – treten z.B. bei Linksrekursion auf; solche Grammatiken sind nicht verarbeitbar
 - Berechnung der FIRST- und FOLLOW-Mengen der Wege, die von Verzweigungsknoten ausgehen

- Generierung:
 - Befehle für eine virtuelle Maschine (in C, C++ oder Java):

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



Befehle der virtuellen Maschine

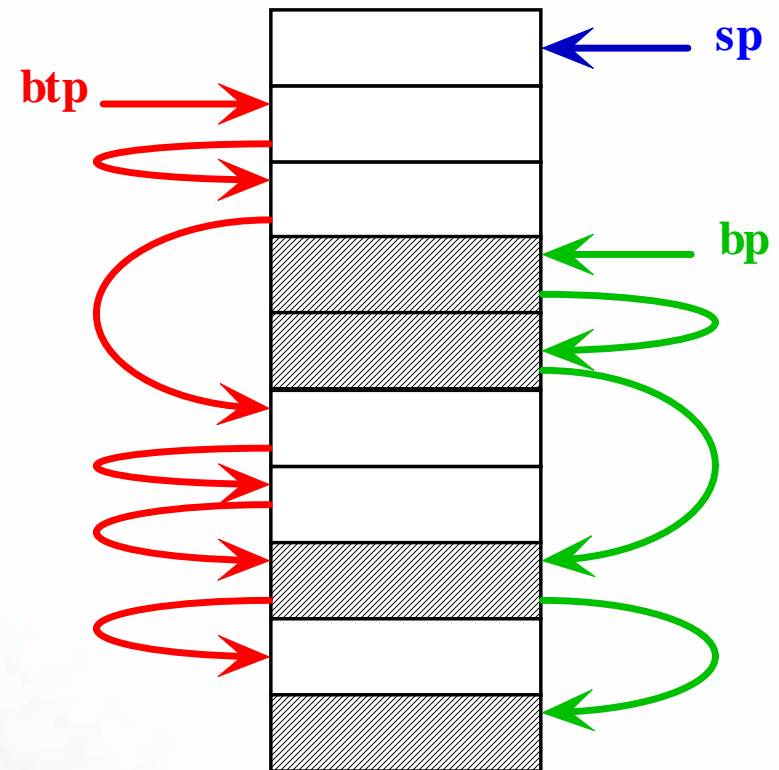
- **BRANCH**  (aus Verzweigungsknoten)
- **GOTO** 
- **TEST**  (aus Terminalsymbolen)
- **CALL**  (aus Nichtterminalen)
- **RETURN**  (aus dem Regelende)



Arbeitsweise der VM

➤ Stack mit zwei Frame-Arten:

- **Call-Frames:** Speichern Daten für die Weiterarbeit nach RETURN (Pos. des Call-Frame der rufenden Regel, Nr. des Befehls nach dem Call)
- **Entscheidungspunkte:** Speichern einen zurückgestellten alternativen "Weg" (Scannerposition, vorheriger Entscheidungspunkt, oberster Call-Frame und alternativer Weg)





Semantik der Befehle (1)

- **BRANCH:**
Neuer Entscheidungspunkt mit alt. Weg auf den Stack
Gehe zu Befehl mit Index **next**
- **GOTO:**
Ändere Stack nicht
Gehe zu Befehl mit Index **next**
- **CALL:**
Neuer Call-Frame mit **alt** als Fortsetzung auf den Stack
Gehe zu 1. Befehl der gerufenen Regel (in **next**)



Semantik der Befehle (2)

➤ RETURN:

Falls `bp != 0`

Gehe zu Befehl aus dem obersten Call-Frame

`bp := bp im obersten Call-Frame`

`/* Stack bleibt unverändert */`

Sonst

Fertig; Parsen war erfolgreich



Semantik der Befehle (3)

➤ TEST:

Falls aktuelles Token == getestetes Terminalsymbol

Ändere Stack nicht; Lese nächstes Token

Gehe zu Befehl mit dem Index **next**

Sonst Backtracking

Falls **bt_p != 0**

Stelle Zustand aus diesem Entscheidungspunkt her

Gehe zu Befehl aus dem Entscheidungspunkt

Sonst

Fertig; Parsen war nicht erfolgreich



Attributierung

- Separater deterministischer Durchlauf nach dem erfolgreichen Beenden des Parsens
- Stackinhalt dient als "roter Faden" (Ist der Entscheidungspunkt noch auf dem Stack, so wähle **next**, sonst wähle **alt**.)
- Für jede Regel gibt es eine Attributierungsfunktion (wie bei Methode des *Rekursiven Abstiegs*)
- Attribute = Parameter dieser Funktionen
- Attributberechnung = C-Blöcke in den Funktionen



Effektivierung mit dem CUT-Operator (!)

- Beispiel: Regel `Block = 'BEGIN' {Stmnt} 'END' .`
- Nach `'END'` wurde ein gültiger Block gefunden → Es ist (meist) sinnlos, eine Alternative zu suchen
- Alle Entscheidungspunkte, die nach `'BEGIN'` angelegt wurden, sollten nach `'END'` gelöscht werden
- Das realisiert der Operator `!` (CUT)
- Regel mit CUT: `Block = 'BEGIN' {Stmnt} 'END' ! .`
- Verkürzt hauptsächlich die Verarbeitungszeit bei fehlerhaften Eingaben



Semantische Prädikate

- Ausdrücke mit booleschen Werten
- Werden von der VM gerufen
 - Wert = **TRUE** → Fortsetzung mit dem nächsten Befehl
 - Wert = **FALSE** → Backtracking
- Verwendung z.B. um Regeln dialektabhängig an- und abzuschalten
- Beispiel: `<? Dialekt == MF_COBOL ?> ...`
- Kein Zugriff auf Attributwerte (werden erst später berechnet)



Test des Tokenwertes

- Normalerweise werden mit Befehl TEST Tokenarten (z.B. `NUMLIT`, `IDENT`) verglichen
- Diese werden als `enum` implementiert – effizienter Vergleich
- Manchmal Vergleich des Tokenwertes sinnvoll
- Beispiel: Erkennung der Stufennummer 78 in COBOL
- Notation in doppelten Hochkommas: `"78"`



Erweiterungen (1)

- Ziel der Version 2: Erweiterbarkeit für Generierung von Parsern in mehreren Zielsprachen gewährleisten
- Trennung:
 - Graphenaufbau, Tests und Mengenberechnung
→ sprachunabhängig
 - Generierung der Parser
→ sprachspezifisch
- Bisher: Generierung von Parsercode in C, C++ und Java
- Strukturierte Attributberechnungsfunktionen (kein GOTO)



Erweiterungen (2)

- Wiederholung 1 oder mehrmals: $\{\dots\}^+$
- Wiederholung m- bis n-fach: $\{\dots\}^m - n$
- Tokenmengen:
tokenset S1 = ANY - {ADD, SUB, MUL, DIV}.
 - **ANY** – vordefinierte Menge aller vorkommenden Token
 - Spezieller Befehl TEST_SET in der virtuellen Maschine
 - Wie Terminalsymbole verwendet
 - Verwendung: z.B. wenn Schlüsselworte an bestimmten Stellen wie Bezeichner verwendet werden dürfen
- Reduzierung der Backtracking-Fälle durch konsequente Verwendung der FIRST-/FOLLOW-Mengen



Vorführung





Anwendungsprojekte für die Version 2

- Java-Frontend
 - Neue Version (mit Unterstützung für Java 1.6)
 - 109 Syntaxregeln
 - Verwendung in JAVA FGM und im Rochade-Repository von ASG

- Verbundprojekt SOAMIG
 - Migration in serviceorientierte Architekturen
 - Partner: Institut für Softwaretechnik der Uni Koblenz
Amadeus Germany GmbH
pro et con
 - Ziel: Migration von Legacy-Systemen toolgestützt in SOA
 - Verwendung des Java-Frontends von pro et con
 - Reimplementation des COBOL-Frontends



Geplante Erweiterungen

- Generierung von Parsern in anderen Programmiersprachen
 - Perl – für Analyse von Script-Sprachen, Ablösung von **Parse::RecDescent**
- Visueller Debugger für die virtuelle Maschine
 - Anzeige der aktuellen Position in BNF, Eingabe und Stack
 - Einzelschrittverarbeitung, Breakpoints, ...
 - Schnittstelle zum generierten Parser: TCP (sprachunabhängig)
- Profiler zur Analyse von Optimierungsmöglichkeiten
 - Anzahl der Backtrackings für bestimmte Regeln und Eingaben



Zusammenfassung

- BTRACC2
 - Ausgetestet, praktisch anwendbar
 - Für C, C++ und Java → erweiterbar auf andere Sprachen
 - Auf die Belange der Analyse von Legacy-Quellcode "zugeschnitten"
 - In mehreren Projekten verwendet, andere folgen → SQL, CICS, DL/I
 - Entwicklung geht weiter → Perl, Debugger, ...



Vielen Dank

pro et con

Innovative Informatikanwendungen GmbH

Analyse · Reengineering · Migration

Uwe Erdmenger

Uwe.Erdmenger@proetcon.de

www.proetcon.de

Tel.: 0371/5347-236

Fax: 0371/5347-345