Technische Universität München
Fakultät für Informatik
Prof. Dr. Helmut Seidl
R. Höllerer

WS 2003/04

Übersetzung von Programmiersprachen

Merkblatt 2

Erzeugung eines Parsers mit CUP

1 Vorbereitungen

Der Parsergenerator CUP wurde von Scott E. Hudson am Georgia Institute of Technology implementiert und ist unter Andrew W. Appels[1] Anleitung erweitert und ergänzt worden. CUP generiert aus einer Parserspezifikation einen LALR-Parser in der Programmiersprache Java. CUP kann im gemeinsamen Einsatz mit Scannergeneratoren (wie z. B. JFlex oder JLex) verwendet werden.

CUP steht an den Rechnern der Informatik-Halle in /usr/proj/uebbau/CUP zur Verfügung. Wenn Sie die Vorbereitungen wie im Merkblatt 1 beschrieben gemacht haben, kann ein Parseraus der Spezifikation mimasyntax..cup folgendermaßen erstellt werden:

```
cup -interface -parser "Parser"< mimasyntax.cup
```

Zum Testen braucht man noch einen Scanner und ein Hauptprogramm, das den Parser aufruft.

Sei die Scannerspezifikation in mimalex.jflex und das Hauptprogramm in Main.java, so kann der Parser mit der Testeingabe test.mima folgendermaßen getestet werden:

```
jflex mimalex.jflex
javac Main.java
java Main test.mima
```

In der Vorgabe zum Übungsblatt 2 ist ein Makefile mit obigen Kommandos angegeben.

2 CUP-Eingabe

Die CUP-Eingabe besteht aus vier Teilen:

User Code
Terminale und Nonterminale
Operatorpräzedenzen
Grammatik

---

[1] Andrew W. Appel: Modern Compiler Implementation in Java. Cambridge University Press 1998. (Im Buch wird die Verwendung von CUP beschrieben)

User Code enthält package- und import-Anweisungen und eventuell Parser-Initialisierungscode sowie die Angabe der Scannerfunktion, mit der das jeweils nächste Symbol vom Scanner geliefert wird.
Terminale und Nonterminale enthält die Liste der Terminal- und Nonterminalsymbole der Grammatik. Die Liste kann auch Typangaben enthalten (s. dazu Abschnitt 3 Objekttyp Symbol).

Im Operatorpräzedenzen-Teil können für Operatoren Präzedenzen angegeben werden. Diese Angaben ermöglichen die Verwendung von mehrdeutigen Grammatiken, in denen die Konflikte durch Operatorpräzedenzen gelöst werden können.
Der Grammatik-Teil enthält die Syntax der Sprache. Die Grammatikregeln haben die Form

l ::= r ;

wobei l eine Nonterminalbezeichnung ist (bestehend aus Kleinbuchstaben) und r die verschiedenen Alternativen für l enthält.

In den rechten Seiten der Regeln können Aktionen, d.h. Java Code eingeschlossen in {: und :} angeben werden. Dieser Code wird vom Parser ausgeführt, wenn er den Teil der rechten Seite erkannt hat, der links vor der Aktion steht. In den Aktionen wird üblicherweise der abstrakte Syntaxbaum aufgebaut oder in einem sehr einfachen Compiler direkt der Code erzeugt.

Beispiel einer Parserspezifikation:

```
import java.io.*;
import java_cup.runtime.Symbol;

scan with {: return lexer.next_token(); :};

   /* Terminals (tokens returned by lexer). */
   terminal WHILE, DO, OD, ASSIGN;
   terminal Integer ADDOP, MULOP, RELOP;
   terminal Integer INTCONST;
   terminal String  IDENT;

   non terminal stat, statList;
   non terminal cond;
   non terminal expr, term, fact ;

   stat ::=  WHILE cond DO statList OD
               {:  System.out.println("while-stat"); :}
         |    IDENT ASSIGN expr
         ;

   statList::=     statList stat
          |       stat
          ;

   cond ::= expr RELOP expr
             {:  System.out.println("cond"); :}
           ;

   expr  ::=  expr ADDOP term
          |     ADDOP term
          |  term
          ;
```

```
term  ::= term MULOP fact
        | fact
        ;
fact  ::= INTCONST
        | IDENT
        ;
```

Aus dieser Parserspezifikation wird das Parserprogramm und eine Codierung der Grammatiksymbole als int-Konstanten erzeugt:

- Parser.java
- sym.java

3 Objekttyp Symbol

Von CUP erzeugte Parser verwenden als Typ für Terminale und Nonterminale den Objekttyp Symbol, d.h. der Scanner muß Objekte vom Typ Symbol liefern und der Parser liefert als Ergebnis der Analyse ein Objekt vom Typ Symbol. Die Definition von diesem Typ steht in java_cup.runtime.Symbol.java.

Der Objekttyp Symbol enthält folgende Komponenten:

```
int sym
Object value
int left
int right
```

Die erste Komponente sym gibt an, um welches Terminal- oder Nonterminalsymbol es sich handelt. Dafür erstellt CUP eine Integercodierung der Grammatiksymbole. Die Codierung der Terminalsymbole wird in der Datei sym.java ausgegeben, z.B. für obige Spezifikation:

```
public interface sym {
  /* terminals */
  public static final int EOF = 0;
  public static final int INTCONST = 9;
  public static final int DO = 3;
  public static final int error = 1;
  public static final int ADDOP = 6;
  public static final int OD = 4;
  public static final int ASSIGN = 5;
  public static final int RELOP = 8;
  public static final int MULOP = 7;
  public static final int WHILE = 2;
  public static final int IDENT = 10;
}
```

Die zweite Komponente value ist vom Typ Object und enthält die semantische Information, z.B. beim Terminalsymbol IDENTIFIER den Bezeichner (als String) oder bei INTCONST den Integerwert (als Objekt vom Typ Integer). Für Terminalsymbole, die keine semantische Bedeutung haben, ist Wert von value null.

Der Typ der value-Komponente wird auch als Typ des Grammatiksymbols verwendet, wie z.B.

```
terminal Integer INTCONST;
terminal String  IDENT;
```

Die dritte und vierte Komponente left und right geben die Position des Symbols in der Eingabe an. Bei Nonterminalsymbolen besetzt der Parser diese Komponenten. Wir haben in unserer JFlex-Spezifikation left und right bei Terminalsymbolen mit Zeilen- und Spaltenangabe besetzt, um adäquate Fehlermeldungen ausgeben zu können.

In den Aktionen hat man Zugriff auf die value-Komponente des Symbols durch sogenannte labels in der rechten Seite der Produktion. Ein label steht hinter dem Grammatiksymbol mit : davon getrennt, z. B.

```
stat ::= IDENT:i ASSIGN expr
            {:  System.out.println("Zuweisung an Identifier"+ i); :}
```

Hier bezieht sich i auf die value-Komponente des Terminalsymbols IDENT (also auf den Bezeichner-String). In der Aktion wird somit der Bezeichner ausgedruckt.

```
stat ::= WHILE cond:c DO statList:l OD
            {: RESULT = new WhileStat(c,l);  :}
```

Hier bezieht sich c auf die value-Komponente des Nonterminalsymbols cond und l auf die value-Komponente des Nonterminalsymbols statList. In der Aktion wird aus den Komponenten ein neuer WhileStat-Knoten erzeugt. Diese Aktion dient zum Aufbau des abstrakten Syntaxbaumes.

Dabei muß man in der Spezifikation für die Nonterminale geeignete Typen angegeben. Da man Statement als einen Obertypen vorstellen kann und die verschiedenen Statements als Ausprägungen davon, ist es naheliegend entsprechende Klassenhierarchie für den abstrakten Syntaxbaum zu definieren, etwa folgendermaßen:

```
public interface SyntaxNode {...}

public abstract class Stat implements SyntaxNode { ...}

public class WhileStat extends Stat{ ...}

public class StatList implements SyntaxNode { ...}
```

Die Typangaben in der Liste der Nonterminale wären dann:

```
non terminal StatList statList;
non terminal Stat stat;
```

Der Bezeichner RESULT ist von CUP vordefiniert und bezieht sich auf die value-Komponente des Symbols auf der linken Seite der Regel, d. h. als Ergebnis der Aktion wird ein neues Objekt vom Typ Symbol erzeugt, dessen sym-Komponente mit stat besetzt ist und die value-Komponente mit dem neuen WhileStat-Knoten.

4 CUP Dokumentation

Unter /usr/proj/uebbau/ CUP/manual.html ist ein ausführliches Manual über CUP zu finden.