Java Compiler Compiler (JavaCC)

Andreas Kunert

Überblick über den Vortrag

1. Überblick über JavaCC
2. einführendes Beispiel
3. lexikalische Analyse
4. syntaktische Analyse
5. semantische Aktionen
6. Syntaxbaumgenerierung (JJTree)
7. Fragen

1. Überblick über JavaCC

Eigenschaften von JavaCC (1)

- Scanner- und Parsergenerator in/für Java
- erzeugt LL(k)-Parser (rekursiver Abstieg)
- mittels Präprozessor Syntaxbaumkonstruktion möglich (JJTree)
- Dokumentationswerkzeug enthalten (JJDoc)

Eigenschaften von JavaCC (2)

- JavaCC ist selbst in JavaCC implementiert worden
- unterstützt Unicode
- Parsergenerator versteht EBNF
- sehr gute Debuggingmöglichkeiten
- exzellente Fehlerbehandlung
  (sowohl in JavaCC als auch in den generierten Compilern)
- „100% Pure Java“-zertifiziert
- bewegte Geschichte (Suntest, Metamata, Webgain, Sunlabs)

Informationen

- Webseite:
  `www.experimentalstuff.com/Technologies/JavaCC`
- Newsgroup:
  `comp.compilers.tools.javacc`
- Mailing List:
  `www.experimentalstuff.com/Technologies/JavaCC/mailinglist.html`
  `javacc-interest@experimentalstuff.com`
- FAQ:
  `http://www.engr.mun.ca/~theo/JavaCC-FAQ`
- Repository of JavaCC Grammars:
  `www.cobase.cs.ucla.edu/pub/javacc`

2. Einführendes Beispiel

Anwendung von JavaCC

1. `MeinCompiler.jj` erzeugen (vim ;-)
2. `javacc MeinCompiler.jj`
   (generiert Scanner, Parser und diverse Hilfsdateien)
3. `javac MeinCompiler.java`
4. `java MeinCompiler`
   (wenn Parser direkt startbar)

Aufbau einer .jj-Datei (formal)

```
javacc_input ::= [ "options" "{" (option)+ "}" ]
                 "PARSER_BEGIN" "(" <IDENTIFIER> ")"
                 java_compilation_unit
                 "PARSER_END" "(" <IDENTIFIER> ")"
                 ( production )*
                 <EOF>
```

Aufbau einer .jj-Datei

```
options{
   LOOKAHEAD = 1;
   ...
}
PARSER_BEGIN(MeinParser)
   public class MeinParser {
      public static void main(String argv[]) {...}
   }
PARSER_END(MeinParser)

... // Scanner- und Parserregeln
```

Beispiel

- `MyCalc.jj`
  „handcompiliert“

3. Die lexikalische Analyse

Tokenaktionen

- `TOKEN`
  Token werden ganz normal an Parser weitergegeben
- `SKIP`
  Token werden ignoriert
- `MORE`
  bisher gelesene Zeichen werden gepuffert und lexikalische Analyse fortgeführt
- `SPECIAL_TOKEN` Token werden „asynchron“ an Parser weitergegeben

Aufbau einer Lexikregel (formal)

```
regular_expr_production ::= [ lexical_state_list ]
                            (   "TOKEN"
                              | "SKIP"
                              | "SPECIAL_TOKEN"
                              | "MORE" )
                            [ "[" "IGNORE_CASE" "]" ]
                            ":"
                            "{" regexpr_spec ("|" regexpr_spec)* "}"
```

Aufbau einer Lexikregel (formal)

```
regexpr_spec       ::= regular_expression
                       [ java_block ]
                       [ ":" java_identifier ]
regular_expression ::= java_string_literal
                       | "<" [ [ "#" ] java_identifier ":" ]
                         complex_regular_expression_choices ">"
                       | "<" java_identifier ">"
                       | "<" "EOF" ">"
```

Beispiel

- `XScanner.jj`

4. Die syntaktische Analyse

Eigenschaften des Parsers

- LL(k) – mit k beliebig, aber fest
- *local-lookaheads* möglich
- Prinzip rekursiver Abstieg
- Erweiterte BNF

Rekursiver Abstieg

```
programm =
        "PROGRAM" bezeichner ";" block ".".
block =
        [ konstantendefinitionsteil ] [ typendefinitionsteil ]
        [ variablendeklarationsteil ] [ prozedurdeklarationsteil ]
        "BEGIN" verbundanweisung "END".
```

```
public void program() {
        nextSymbol();
        testFor(PROGRAMSY);
        testFor(IDENT);
        testFor(SEMICOLON);
        block();
        testFor(PERIOD);
}
private void block() {
        if ( testIf (CONSTSY)) constantdefinitionpart();
        if ( testIf (TYPESY)) typedefinitionpart();
        if ( testIf (VARSY)) vardeclarationpart();
        if ( testIf (PROCEDURESY)) proceduredeclaration();
        testFor(BEGINSY);
        statseq();
        testFor(ENDSY);
}
```

Aufbau einer Grammatikregel

```
void metasymbol()
{
   ... // Code der am Anfang der Methode steht
}
{
   <Terminal>                        { ... }
|  <Terminal1> <Terminal2>           { ... }
|  ( <Terminal1> | <Terminal2> )     { ... }
|  [ <Terminal> ]                    { ... }
|  ( <Terminal> )?                   { ... }
|  ( <Terminal> )+                   { ... }
|  ( <Terminal> )*                   { ... }
}
```

Beispiel

- `CopCow.jj`
  (mit verschiedenen Lookaheads)

5. Semantische Aktionen

Einbau von semantischen Aktionen

- an (nahezu) beliebiger Stelle in Lexik-/Grammatikregeln
- nur Java 1.2-Syntax möglich (keine Assertions)
- beliebige Rückgabetypen
- Methodenköpfe können frei definiert werden

Beispiel

- `MyCalc2.jj`

6. Syntaxbaumkonstruktion mit JJTree

Geänderter Erstellungsablauf

- `MeinCompiler.jjt` erzeugen (vim ;-)
- `jjtree MeinCompiler.jjt`
  (generiert Knotenklassen und JavaCC-Quelldatei)
- `javacc MeinCompiler.jj`
  (generiert Scanner, Parser und diverse Hilfsdateien)
- `javac MeinCompiler.java`
- `java MeinCompiler`
  (wenn Parser direkt startbar)

Beispiel

- `MyCalc3.jj`

- Minako

7. Fragen ???