

SableCC

Benjamin Daeumlich

06.07.2006

Gliederung

- 1 Einführung
- 2 Compilergenerierung
- 3 Bewertung
- 4 Referenzen

Einführung

Autor

- Autor: Étienne Gagnon



- SableCC: Teil seiner Masterarbeit an der McGill-Universität Montreal vom März 1998

Name

- Sable: Forschungsgruppe an der McGill-Universität
- Zielsetzung: Tools entwickeln, um Java schneller zu machen
- wie Java benannt nach einer Insel (an der kanadischen Ostküste)

Features

- generiert Lexer, Parser, AST-Generator und „Treewalker-Klassen“
- Lexer: basiert auf DFA, unterstützt Unicode
- Parser: LALR(1)
- EBNF wird unterstützt
- geschrieben in Java und erzeugt Java-Code (Zielsprache)
- Lizenz: LGPL
- aktuelle Version: 3.2 (Dezember 2005)

Compilergenerierung

Compilergenerierung in 5 Schritten

- 1 Erstellung der Grammatik-Datei
- 2 SableCC anwenden auf Grammatik-Datei („Framework erstellen“)
- 3 Klassen für semantische Aktionen erstellen („working-classes“)
- 4 Main-Klasse erstellen („driver-class“)
- 5 Compiler mit Java kompilieren

1. Schritt

Erstellung der Grammatik-Datei

Grammatik-Datei

- eingeteilt in fünf Abschnitte:
 - Package
 - Helpers
 - Tokens
 - Ignored Tokens
 - Productions

- keine Konvention für Dateierdung

Package und Helpers

- Package:
 - Ordner, in dem Dateien generiert werden
- Helpers:
 - **Syntax:** `<<Name>> = <<Definitionsteil in EBNF>>;`
 - für „Hilfstoken“, z. B. zum Definieren von Zahlen, Buchstaben oder Zeilenumbrüchen
 - im Definitionsteil entweder Angabe des:
 - Zeichens in einfachen Anführungszeichen
 - ASCII-Codes des Zeichens
 - Unicodes des Zeichens
 - Zeichenfolgen: `['0' .. '9']`
 - Ausschluss: `[1 .. 128] - 10`

Tokens und Ignored Tokens

- Tokens:
 - **Syntax:** $\langle\text{Tokenname}\rangle = \langle\text{Tokendefinition in EBNF}\rangle;$
 - bei Schlüsselwörtern: Wort in Definition in einfachen Anführungszeichen
- Ignored Tokens:
 - Angabe von zu überlesenden Tokens, durch Komma getrennt und mit Semikolon abgeschlossen

Productions

- **Syntax:** $\langle\langle\text{Regelname}\rangle\rangle = \{ \langle\langle\text{Alternativename}\rangle\rangle \langle\langle\text{Regeldefinition in EBNF}\rangle\rangle ;$
- für jede Alternative anderer Name
- Klammerungen nicht erlaubt
- bei mehrfacher Verwendung eines Nichtterminals in der Regeldefinition:
 - [$\langle\langle\text{Name}\rangle\rangle$]: vor die Nichtterminale (wobei $\langle\langle\text{Name}\rangle\rangle$ stets verschieden in einer Alternative)

2. Schritt

SableCC anwenden auf Grammatik-Datei

Framework erstellen

```
java -jar sablecc.jar «Grammatik-Datei»
```

```
Verifying identifiers.  
Generating token classes.  
Generating production classes.  
Generating alternative classes.  
Generating analysis classes.  
Generating utility classes.  
Generating the lexer.  
-Constructing NFA.  
.....  
-Constructing DFA.  
.....  
-resolving ACCEPT states.  
Generating the parser.  
.....  
.....  
.....  
..
```

generierte Dateien

- Lexer:
 - `Lexer.java`
 - `LexerException.java`
- Parser:
 - `Parser.java`
 - `ParserException.java`
- Analysis:
 - „Treewalker-Klassen“
- Node:
 - Klassen, die den AST definieren

Namensgebung für „Node-Klassen“

- Token:
T«**Tokenname**».java
- Produktion:
P«**Produktionsname**».java
- Alternative:
A«**Alternativname**»«**Produktionsname**».java
- Großschreibung am Anfang und nach Unterstrich im Namen
(Token kw_while generiert TKwWhile.java)

3. Schritt

„working-class“ erstellen

Aufbau einer „working-class“

```
import minako.node.*;
import minako.analysis.*;
import java.util.*;

public class Semantic extends DepthFirstAdapter
{

    /* Code */

}
```

DepthFirstAdapter.java

- DepthFirstAdapter.java enthält für jeden Knoten eine Methode namens:

```
case<<Knotenname>> (<<Knotenname>> node)
```

- darin wird zuerst die Methode

```
in<<Knotenname>> (<<Knotenname>> node)
```

und zuletzt

```
out<<Knotenname>> (<<Knotenname>> node)
```

ausgeführt

- kann jeweils neu definiert werden in „working-class“

Beispiel

- „returnstat = {empty} kw_return semicolon | ...;“
erzeugt:

```
public void inAEmptyReturnstat (AEmptyReturnstat node) {
    defaultIn(node);
}

public void outAEmptyReturnstat (AEmptyReturnstat node) {
    defaultOut(node);
}

@Override
public void caseAEmptyReturnstat (AEmptyReturnstat node) {
    inAEmptyReturnstat(node);
    if(node.getKwReturn() != null) {
        node.getKwReturn().apply(this);
    }
    if(node.getSemicolon() != null) {
        node.getSemicolon().apply(this);
    }
    outAEmptyReturnstat(node);
}
```

nützliche Methoden für „Nodes“

Methode	Funktion
apply()	„besucht“ den Knoten
clone()	kopiert den „Unterbaum“
parent()	gibt Elternknoten zurück (oder null)
removeChild()	entfernt einen Kinderknoten
replaceChild()	ersetzt einen Kinderknoten
replaceBy()	ersetzt „Unterbaum“
toString()	Textrepräsentation vom „Unterbaum“

4. Schritt

„driver-class“ erstellen

Aufbau der „driver-class“

```
import minako.parser.*;
import minako.lexer.*;
import minako.node.*;
import java.io.*;

public class Main {

    public static void main(String[] args) {

        Lexer lexer = new Lexer (new PushbackReader(new BufferedReader(new
            FileReader(args[0])), 1024));

        Parser parser = new Parser(lexer);

        Start ast = parser.parse();

        ast.apply(new Semantic());

    }

}
```

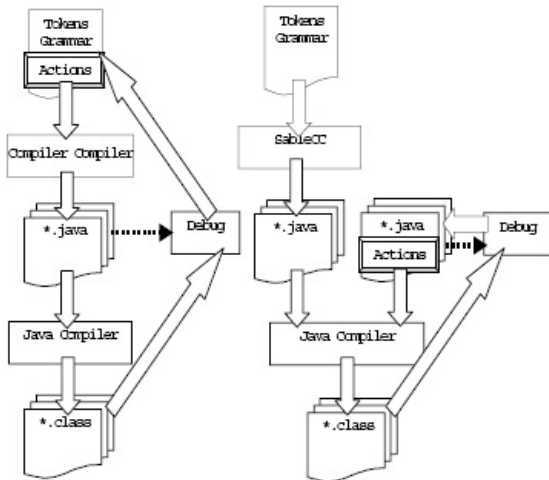

5. Schritt

Compilierung

```
javac Main.java
```

Bewertung

Vergleich zu anderen Java-Compilergeneratoren



Vorteile

- Trennung zwischen generiertem Code und „Benutzer-Code“
- Unicode-Unterstützung
- Objektorientierung durch Java
- automatische AST-Generierung
- wird noch weiterentwickelt (Version 4.0 angekündigt)

Nachteile

- keine Präzedenzen
- keine semantischen Attribute
- Code für sematische Aktionen lang und teilweise unübersichtlich
- ungenügende Dokumentation

Referenzen

Referenzen & Literatur



SableCC Homepage

<http://www.sablecc.org/>



Étienne Gagnon's Master thesis



Smallpascal-Beispiel

<http://www.brainycreatures.org/compiler/sablecc.asp>