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

SS 2004

Übersetzung von Programmiersprachen

Merkblatt 1

Erzeugung eines Scanners mit JFlex

1 Vorbereitungen

Der Scannergenerator JFlex wurde von Gerwin Klein[1] in seiner Studiumszeit an der TUM implementiert. JFlex generiert aus einer Scannerspezifikation einen Scanner in der Programmiersprache Java. JFlex kann entweder allein oder im gemeinsamen Einsatz mit dem Parsergenerator CUP verwendet werden.
JFlex steht an den Rechnern der Informatik-Halle in /usr/proj/compiler/compilerbau/bin zur Verfügung. Um mit JFlex zu arbeiten ist es günstig, einen Link auf jflex im eigenen bin-Verzeichnis zu erstellen. Geben Sie ein:

```
cd ~/bin
ln -s /usr/proj/compiler/compilerbau/bin/* .
```

dann kann ein Scanner aus der Spezifikation minispec.jflex folgendermaßen erstellt und mit der Testeingabe test.mini getestet werden:

```
jflex minispec.jflex
javac Lexer.java
java Lexer test.mini
```

2 JFlex-Eingabe

Die JFlex-Eingabe besteht aus drei Teilen, die mit %% voneinander getrennt werden.

```
User Code
%%
Optionen und Deklarationen
%%
Lexikalische Regeln
```

User Code wird zum Anfang des erzeugten Scanners kopiert. Üblicherweise stehen in diesem Teil Kommentare sowie package- und import-Anweisungen.

Optionen und Deklarationen enthält JFlex-Optionen, Java Code eingeschlossen in %{und %}, sowie sogenannte Makrodefinitionen. Mit einer Makrodefinition wird einem regulären Ausdruck ein Name gegeben, und dieser Name kann dann in den lexikalischen Regeln verwendet werden.

In den lexikalischen Regeln werden die zu erkennenden Symbole definiert.

---

1. Nun Mitarbeiter der Fakultät s. http://www.informatik.tu-muenchen.de/~kleing

Beispiel einer Scannerspezifikation:

```
// JFlex lexer specification for mini
//import java_cup.runtime.Symbol;

%%

%class Lexer
%standalone
%line
%column
%{
   private void out(String sym) {
   System.out.println( "( " + sym + ", irrelevant "+ ", \t" + (yyline+1) + " ) " );
 }
 private void out(String sym, String semval) {
   System.out.println( "( " + sym +", " + semval +  ", \t" + (yyline+1) + " ) " );
 }
%}
Ident = [a-zA-Z] [a-zA-Z0-9]*
IntLiteral = 0 | [1-9][0-9]*
RealLiteral = [0-9]+ \. [0-9]+
new_line = \r|\n|\r\n;
white_space = {new_line} | [ \t\f]

%%

"if"              { out("If-Symbol"); }
"then"            { out("Then-Symbol"); }
"else"            { out("Else-Symbol");}
{Ident}           { out("Identifikator", yytext()); }
{IntLiteral}      { out("Intkonst", yytext()); }
{RealLiteral}     { out("Realkonst", yytext()); }
":="              { out("Zuweisungsoperator"); }
{white_space}     { /* ignore */ }
.|\n              { out("!!! Illegales Zeichen", yytext() );  }
```

Erläuterungen zur Spezifikation:

%class Lexer	Erzeugter Scanner heißt Lexer und wird in die Datei Lexer.java geschrieben.
%standalone	Es wird eine main Funktion erzeugt.
%line	Der Scanner führt Zeilenzählung durch.
%column	Der Scanner führt Spaltenzählung durch.
%{ ... %}	Der in %{und %} eingeschlossene Java Code wird in den erzeugten Scanner hineinkopiert.

Ident = [a-zA-Z] [a-zA-Z0-9]* Eine Makrodefinition[2]: ein Identifikator fängt mit einem Buchstaben an, danach können weitere Buchstaben oder Ziffern folgen. Verwendet wird diese Makrodefinition in der lexikalischen Regel

2. Makrodefinitionen dürfen nur eine Zeile lang sein.

```
{Ident}        { out("Identifikator", yytext()); }
```

die angibt, wie das Symbol "Identifikator" auszusehen hat und was der Scanner als Ergebnis liefert. Die nach dem Erkennen eines Symbols auszuführende Aktion steht zwischen den Klammern { und }. In einer Testversion des Scanners wird man die erkannten Symbole, wie oben spezifiziert, einfach ausdrucken. In einem Scanner, der mit einem Parser zusammenarbeitet, wird man ein Symbol zurückliefern (s. hierzu Abschnitt Zusammenarbeit mit CUP).

Die letzte lexikalische Regel

```
.|\n           { out("!!! Illegales Zeichen", yytext() );  }
```

ist sehr wichtig. Der Punkt steht für alle Zeichen ausser \n. Obige Regel erfaßt also alle Zeichen (das Zeichen | bedeutet oder), die zu keiner vorangegangenen Symboldefinition gepaßt haben. In einem Scanner, der mit einem Parser zusammenarbeitet, würde hier als Aktion eine Fehlermeldung stehen.

3 Reguläre Ausdrücke

Die Symbole in den Makrodefinitionen und in den lexikalischen Regeln werden durch reguläre Ausdrücke definiert. Diese werden folgendermaßen notiert:

Seien a und b reguläre Ausdrücke, dann bedeutet

a | b	a oder b
a b	a gefolgt von b
a*	Null oder mehrere a's
a+	Ein oder mehrere a's
a?	Ein oder kein a
(a)	Klammerung dient zur Strukturierung des Ausdrucks und matcht a

Reguläre Ausdrücke, die aus einem Zeichen oder einer Zeichenklasse bestehen, werden folgendermaßen notiert:

Seien c, d und e einzelne Zeichen, dann bedeutet

c	das Zeichen c
[cde]	Zeichenklasse: das Zeichen c oder das Zeichen d oder das Zeichen e
[c-e]	Zeichenklasse: alle Zeichen zwischen c und e (z.B. [1-4] bedeutet 1, 2, 3 oder 4)
[^c]	Zeichenklasse: alle anderen Zeichen außer c
[^cde]	Zeichenklasse: alle anderen Zeichen außer c und d und e

Weitere reguläre Ausdrücke sind:

"begin"	String: steht für den Text begin (im Text dürfen die Zeichen \ und " nicht vorkommen)

{Name} Makroverwendung: vorher muß ein Makro, Name = ... , definiert sein

Die in den regulären Ausdrücken vorkommenden Metazeichen müssen meist durch \ vorangestellt werden (z.B. *+ steht für ein oder mehr Sterne; aber [^*/] für alle anderen Zeichen als * oder /).

4 Zusammenarbeit mit CUP

Soll JFlex mit dem Parsergenerator CUP verwendet werden, importiert man Symbol und ersetzt %standalone mit folgenden Optionen:

```
%cup
%implements sym
```

Der von CUP generierte Parser definiert den Typ Symbol und erzeugt aus der CUP-Eingabe die Integer-Kodierung für die verschiedenen Terminalsymbole (Datei sym.java). Wird bei der Parsergenerierung die Option -interface verwendet, kann man in der Scannerspezifikation direkt die Terminalsymbolnamen aus der CUP-Eingabe verwenden.

Wenn

```
stat ::= IDENT ASSIGN expr
```

eine Zeile aus der CUP-Eingabe ist, so verwendet man in der Scannerspezifikation IDENT und ASSIGN als Symbolkodierung in den Aktionen.

```
{Ident}          { return symbol(IDENT, yytext()); }
":="             { return symbol(ASSIGN); }
```

Im Optionen und Deklarationen-Teil könnte man als Java Code folgende Funktionen angeben, die die Symbolübergabe zum Parser (mit oder ohne semval) erledigen.

```
%{
private Symbol symbol(int sym) {
   return new Symbol(sym, yyline+1, yycolumn+1);
 }

 private Symbol symbol(int sym, Object semval) {
   return new Symbol(sym, yyline+1, yycolumn+1, semval);
 }
%}
```

5 JFlex Dokumentation

Unter /usr/proj/compiler/compilerbau/docs/jflex/manual.html ist ein ausführliches Manual über JFlex zu finden.