JLex
ein Scanner-Generator für Java

Bernd Kühl (bekuehl@uos.de)
Axel-Tobias Schreiner (axel@uos.de)

Fachbereich Mathematik/Informatik
Universität Osnabrück

Einleitung

Scanner dienen zur lexikalischen Analyse. Sie bearbeiten einen Strom von Zeichen und klassifizieren diese in sogenannte *Tokens*. Als Beispiel könnte ein Scanner in der Zeichenfolge

```
23.4 hello 20 world
```

eine Gleitkomma-Zahl `23.4`, die beiden Wörter `hello` und `world` und eine Integer-Zahl `20` als *Tokens* finden. Scanner dienen oft, aber nicht nur, als Frontend für einen Compiler.

Die bekanntesten Tools in der C-Branche in diesem Zusammenhang dürften *lex*[1] bzw. *flex*[2] sein. Die beiden Tools nehmen eine tabellarische Spezifikation eines Scanners als Eingabe und generieren eine C-Quelle, die den eigentlichen Scanner darstellt. Damit sind *lex* bzw. *flex* keine Scanner, sondern Scanner-Generatoren.

Java setzt sich mehr und mehr durch. Auch in Java ist das Entwickeln der Scanner von Hand mühsam und fehleranfällig. Eine erste Hilfe stellt die Klasse *java.io.StreamTokenizer* dar. *StreamTokenizer* erkennen ganze Zahlen, Identifier, Kommentare, Zwischenraum und zitierten Text. Für einfache lexikalische Analysen sind sie durchaus geeignet, aber schon Gleitkomma- und Integer-Zahlen zu untersuchen, sprengt die Fähigkeit der Klasse. Hier setzt *JLex*, als eine Umsetzung von *lex* bzw. *flex* nach Java, an. Auch *JLex* erhält eine Scanner-Beschreibung als Eingabe und erzeugt für die eigentliche Scanner-Klasse eine *.java*-Datei.

Ein Beispiel

Die Idee und Benutzung von *Jlex* soll im Folgenden an einem einfachen Beispiel vorgestellt werden. Listing 1 zeigt ein Beispiel zum Erkennen typischer *Tokens* einer Programmiersprache:

```
package scanner;

%%

%{
  public static void main(String argv[]) throws java.io.IOException {
    Scanner scanner = new Scanner(System.in);
    while (scanner.yylex()) ;
  }

  protected void comment(String text)  { System.out.println("\t#–comment: "+text); }
  protected void integer(String text)  { System.out.println("\tinteger: "+text); }
  protected void floating(String text) { System.out.println("\tfloating point: "+text); }
  protected void keyword(String text)  { System.out.println("\tkeyword: "+text); }
  protected void name(String text)     { System.out.println("\tidentifier: "+text); }
  protected void character(char ch)    { System.out.println("\tcharacter: "+ch); }
%}

%public
%class Scanner
```

```
%type boolean
%eofval{
  return false;
%eofval}

alpha   =       [a-zA-Z_]
alnum   =       [a-zA-Z_0-9]
dec     =       [0-9]
sign    =       [+-]?
exp     =       ([eE]{sign}{dec}+)
fp      =       ({dec}+\.{dec}*{exp}?|"."{dec}+{exp}?|{dec}+{exp})
whitespace =    [ \t\r\n\b\015]+
%%

#.*                     { comment(yytext().substring(1)); return true; }
{dec}+                  { integer(yytext()); return true; }
{fp}                    { floating(yytext()); return true; }
while                   { keyword(yytext()); return true; }
{alpha}{alnum}*         { name(yytext()); return true; }
{whitespace}            { }
.                       { character(yytext().charAt(0)); return true; }
```

JLex produziert aus dieser Eingabedatei die Java-Quelle der Scanner-Klasse *Scanner*. Wie es zum Namen der erzeugten Klasse kommt, wird später näher erläutert. Übersetzt man die Klasse und führt `main` aus, wird die Funktionalität des Scanners (siehe Listing 2) klar: Der Scanner erkennt in der Eingabe unabhängig voneinander Integer- und Gleitkomma-Zahlen, das Wort `while`, Identifier, Kommentare (alle Zeichen von # bis zum Ende der Zeile); Zwischenraum wird ignoriert, alle anderen Zeichen erhält man einzeln.

```
$ CLASSPATH=.. java scanner.Scanner
var1 = 2.3;
        identifier: var1
        character: =
        floating point: 2.3
        character: ;
while(23>0)
        keyword: while
        character: (
        integer: 23
        character: >
        integer: 0
        character: )
# hello world!
        #-comment:  hello world!
```

Der Aufbau von JLex-Dateien

Grundsätzlich bestehen *JLex*-Dateien aus drei durch %% getrennte Abschnitte (siehe Listing 1):

```
user code section
%%
JLex directives
%%
regular expression rules
```

Die %%-Anweisungen müssen am Beginn einer Zeile stehen. Folgen weitere Zeichen, werden diese ignoriert.

Der "user code"-Abschnitt

Alle Zeilen bis zur ersten %%-Zeile werden direkt und ohne Änderung an den Anfang der zu erzeugenden Java-Datei kopiert. Hier kann der Entwickler `package`- und `import`-Anweisungen oder auch Kommentare unterbringen. Prinzipiell könnte man hier auch Hilfsklassen unterbringen, die allerdings nicht

public erklärt sein dürfen und daher nur der Scanner-Datei selber zur Verfügung stehen.

In diesem Abschnitt stehen keine Variablen oder Methoden der Klasse bzw. für Objekte der Klasse. Sie würden sonst auf der gleichen Ebene wie `package`- oder `import`-Anweisungen, also nicht innerhalb des Klassen-Körpers, stehen.

JLex Anweisungen

Im mittleren Abschnitt einer *JLex*-Datei, zwischen der ersten und zweiten %%-Zeile, wird das Aussehen und Verhalten der erzeugten Scanner-Klasse durch %-Anweisungen gesteuert. %-Anweisungen sollten immer am Anfang einer Zeile beginnen. Daneben können in diesem Abschnitt Makros für den dritten Abschnitt definiert werden. Auf Makros wird bei der Erklärung des dritten Abschnitts näher eingegangen.

Durch die `%{...%}`-Anweisung wird der Körper der Scanner-Klasse erweitert. Der gesamte Text innerhalb dieser beiden Zeilen wird in den Körper der erzeugten Scanner-Klasse kopiert. In Listing 1 sind so die Klassenmethode *main()* und sechs Instanzmethoden implementiert. An dieser Stelle werden neben Methoden auch Variablen, [statische] Initialisierungs-Blöcke oder auch innere Klassen aufgeschrieben.

Nach Voreinstellung erzeugt *JLex* als Kopf nur `class Yylex`. Mit Anweisungen wie `%public`, `%class` und `%implements` kann der Entwickler den Kopf beeinflussen.

Viele der %-Befehle für *JLex* beschäftigen sich mit dem Aussehen bzw. dem Verhalten der Methode, die den eigentlichen Scanner ausmacht. Diese Methode wird von außen aufgerufen, wenn das nächste Symbol erkannt werden soll. Per Default heißt die Methode *yylex()*, kann aber durch `%function` umbenannt werden. Durch `%yylexthrow` wird der Kopf der Methode um `throws`-Anweisungen erweitert. Per Default liefert die Methode ein Objekt der Klasse *Yytoken*. *Yytoken* ist nicht Teil von *JLex*, sondern muß vom Entwickler implementiert werden. Ist das Resultat ein Objekt einer beliebigen Klasse, liefert *JLex* am Ende der Eingabe `null` als Wert der Methode. Durch `%integer` wird der Resultattyp auf *int* umgestellt und die konstante Instanzenvariable `YYEOF` beim Erreichen des Eingabeendes geliefert. Möchte der Programmierer nicht *Yytoken* und auch nicht *int* als Resultattyp, kann er durch `%type` einen beliebigen Typ auswählen. Wird so ein primitiver Datentyp eingestellt, muß der Entwickler durch `%eofval` einen Block angeben, der am Ende der Eingabe abgefahren wird und muß in diesem Block durch `return` einen Wert liefern, der zum versprochenen Typ paßt (siehe Listing 1).

Zwei Anweisungen dienen zur Verwaltung von Zeichenpositionen innerhalb der Eingabe. Durch `%char` existiert eine Variable *yychar*, durch `%line` eine Variable *yyline*. Beide sind vom Typ *int* und enthalten die absolute Position bzw. die Zeilennummer des ersten Zeichens eines erkannten Tokens.

Der "regular expression rules"-Abschnitt

Der dritte Abschnitt einer *JLex*-Datei besteht aus Regeln, nach denen die Zeichen der Eingabe in die einzelnen *Tokens* zerlegt werden sollen. Eine Regel besteht aus drei Teilen, die durch Zwischenraum getrennt sind: eine optionale Zustandsliste, einem regulären Ausdruck als Muster und einem assoziierten Java-Aktions-Block, der sich über mehrere Zeilen erstrecken kann:

```
[<states>] <expression> { <action> }
```

Wenn ein Stück der Eingabe einem Muster genügt, wird der zugehörige Java-Aktions-Block ausgeführt. Paßt ein Stück der Eingabe zu mehreren Mustern, hat der längste erkannte Text Vorrang und bei gleicher Länge, das erste der Muster. Nicht erkannte Teile der Eingabe werden nicht wie bei *lex* oder *flex* kopiert, sondern ein *Error* wird ausgelöst. Um sich gegen jegliche Eingabe abzusichern, kann der Entwickler als letzte Regel eine Regel der Art

```
.       { java.lang.System.err.println("Unmatched input: " + yytext()); }
```

verwenden.

states bezieht sich darauf, daß man Zustände für den Scanner definiert, in denen dann entsprechend markierte Ausdrücke zusätzlich gelten. Auf diese Weise kann man in verschiedenen Bereichen der Eingabe verschiedene Scanner einsetzen. Ein Beispiel zu dieser sehr mächtigen Technik finden Sie im Paket der Beispiele zu diesem Artikel. Näheres entnehmen Sie sonst bitte der Dokumentation[3,4].

Für reguläre Ausdrücke in *JLex* gelten annähernd die Regeln von *egrep*:

abc...	Zeichen stellen sich selbst dar
"abc..."	jedes Zeichen stellt sich selbst dar; einzige Ausnahme \" steht für "
\n \t \b \f \r	Zeilentrenner, Tab, Backspace, Seitenvorschub, Wagenrücklauf
\ooo \xhh	oktal, hexadezimal
\^C	control-Zeichen
\x	ein zitiertes Zeichen
.	ein beliebiges Zeichen (aber kein Zeilentrenner)
[abd-x...]	ein Zeichen aus einer Klasse
[^abd-x...]	ein Zeichen nicht aus einer Klasse
$	Treffer am Zeilenende
x*	null oder mehrmals
x+	ein oder mehrmals
x?	optional: null oder einmal
xy	nacheinander
x|y	alternativ
(...)	Vorrang steuern
{name}	im zweiten Abschnitt definiertes Makro

In einem Muster kann mit `{name}` ein im zweiten Abschnitt definiertes Makro verwendet werden (siehe Listing 1). Ein regulärer Ausdruck endet vor einem Zwischenraum-Zeichen; der Entwickler kann jedoch bis auf das Newline Zwischenraum zitieren, und in einer Zeichenklasse steht ein Leerzeichen für sich selbst.

Eine weitere mögliche Falle ist, daß die Verwendung eines Makros praktisch einen Textersatz und nicht eine Art Funktionsaufruf darstellt.

```
%%

...
OPT1=   b|c
OPT2=   (x|y)

%%

a{OPT1}d        { System.out.println("found: "+yytext()); }
w{OPT2}z        { System.out.println("found: "+yytext()); }
```

Hier sind `ab` und `cd` passende Zeichenfolgen für das erste Muster, welches nach Einsatz des Makros `ab|cd` lautet. Dies ist sicherlich nicht der gewünschte Effekt. Der Entwickler sollte daher Makros im Zweifel vorsichtshalber immer klammern, siehe Listing 1.

Ist ein Muster erkannt, wird dessen Aktions-Block ausgeführt. Wird in dem Block nicht per `return` die Methode zur lexikalischen Analyse verlassen (siehe Listing 1), beginnt danach die Erkennung des nächsten *Tokens*, so ignoriert man zum Beispiel Zwischenraum.

Der Entwickler sollte den Java-Code in Aktions-Blöcken eher knapp halten, damit bleibt die Scanner-Beschreibung übersichtlich. Eine oft verwendete Technik ist, in dem Aktions-Block eine Methode mit dem erkannten Text als Argument aufzurufen. Die Bearbeitung des erkannten Texts konzentriert sich damit innerhalb der Methode, die auch von mehreren Aktions-Blöcken aufgerufen werden kann. Weiterhin kann die Methode in Unterklassen überschrieben werden. Man bekommt unterschiedliche Funktionalitäten, ohne den Scanner ändern zu müssen.

Der Entwickler sollte den Scanner so einfach wie möglich halten. Es ist effizienter, Schlüsselwörter und Identifier über ein Muster zu erkennen und in der assoziierten Methode den erkannten Text über eine Hashtabelle auf ein Schlüsselwort zu testen.

Installation von JLex
Übersetzung von JLex-Dateien

Die Installation gestaltet sich recht einfach. Zusammen mit den Beispielen dieses Artikels können Sie die *JLex*-Quelle vom ftp-Server der iX unter ftp://www.heise.de/pub/ix/ix_listings/2000_03/jlex.tgz beziehen. Die Quelle bestehen aus einer einzigen Java-Datei *Main.java*. Am einfachsten übersetzen Sie die Datei und erzeugen aus allen *.class*-Dateien ein *jar*- oder *zip*-Archiv. Mit diesem Archiv auf dem Klassenpfad gestaltet sich die Übersetzung einer *JLex*-Quelle recht einfach. Der komplette Vorgang wird im Paket der Beispiele zu diesem Artikel durch *makefiles* geregelt. Im folgenden Listing werden die Schritte von Hand ausgeführt, wobei *Main.java* im Katalog *JLex* lebt.

```
$ CLASSPATH=. javac Main.java
$ cd .. && jar cf JLex/JLex.jar JLex/*.class
```

Das folgende Listing zeigt wie die *JLex*-Datei aus Listing 1 übersetzt wird:

```
$ CLASSPATH=path_to_jlex/JLex/JLex.jar:.. java JLex.Main Scanner.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 84 states.
Creating DFA transition table.
Working on DFA states.............................
Minimizing DFA transition table.
16 states after removal of redundant states.
Outputting lexical analyzer code.
$ mv Scanner.lex.java Scanner.java
$ CLASSPATH=.. javac  Scanner.java
$
```

Mit der *jar*-Datei für *JLex* auf dem Klassenpfad bringt man *JLex.Main* in der JVM zur Ausführung und gibt die Eingabedatei als nachfolgendes Kommandozeilen-Argument mit. *JLex* untersucht die Datei auf ihre Gültigkeit, berechnet einen deterministischen endlichen Automaten zur Klassifizierung der Zeichen aus der Eingabe entsprechend der Tabelle von regulären Ausdrücken und erzeugt abschließend eine *.java*-Datei. Der Name der generierten Datei besteht dabei aus den Dateinamen der bearbeiteten Eingabedatei, verlängert um den Suffix *.java*. Im nächsten Schritt übersetzt man die *java*-Datei wie gewohnt. Dabei ist zu bemerken, daß der erzeugte Java-Text nicht mehr von *JLex* abhängig ist. Die *.jar*-Datei von *JLex* braucht also nicht mehr auf dem Klassenpfad liegen und muß nicht als Runtime-Support dem erzeugten Scanner beigelegt werden.

Fehler (z.B. syntaktische Fehler in den Aktionen) beim Compilieren der Java-Datei des Scanners beziehen sich natürlich auf die Java-Datei und nicht auf die entsprechende *JLex*-Quelle — das ist leider in Java nicht zu ändern. Leider liefert *JLex* auch in Form von Kommentaren keinen Hinweis...

Der erzeugte Quelltext

Die generierte Java-Klasse hat zwei *public* Konstruktoren. Beide bekommen genau ein Argument der Klasse *java.io.Reader* bzw. *java.io.InputStream* als zu untersuchenden Eingabestrom von Zeichen. Im zweiten Fall wird der *InputStream* innerhalb des Konstruktors durch ein *InputStreamReader* mit dem Standard Unicode Encoding in einen *Reader* gewandelt.

Dieses Design hat meines Erachtens mindestens zwei Schwachpunkte. Besteht der vom Scanner zu untersuchende Eingabestrom aus der Verkettung mehrerer *Reader*- und/oder *InputStream*-Objekte, muß der Entwickler sich selber um eine Kapselung der Ströme zu einem *Reader*- bzw. *InputStream*-Objekt kümmern. Es gibt keine *SequenceReader*-Klasse und mehrfaches Lesen von der Standardeingabe mit einem *SequenceInputStream*-Objekt ist ohne Kapselung des `System.in`-Stroms in einen *FilterInputStream*-Objekt, der ein `close()` gegen den Strom ignoriert, nicht möglich. Eine Methode `addInput(Reader)` bzw. `addInput(InputStream)` des erzeugten Scanners könnte eine bequeme

Schnittstelle zum Verlängern der Eingabe sein.

Das zweite Problem besteht darin, daß bei diesem Design die Eingabe bereits bei der Konstruktion des Scanners bekannt sein muß, was aber in einem unserer Projekte nicht der Fall gewesen ist. Es gibt keinen Konstruktor ohne Parameter und ein späteres Setzen der Eingabe über eine Methode wie `addInput()` ist nicht vorgesehen. Ein Entwickler, der in dieses Problem läuft, kann sich durch folgenden Trick helfen: Bei einem Blick in den Text der erzeugten Java-Quelle für einen Scanner sieht man, daß es den Konstruktor ohne Parameter in der Scanner-Klasse gibt. Dieser ist allerdings *private* deklariert. Der Entwickler kann nach dem Erzeugen der Java-Datei durch *JLex* z.B. durch einen *sed*-Aufruf den *private*-Deklarator in *public* abändern. Analog sieht man, daß der Eingabestrom in der Instanzenvariable *yy_reader* vom Typ *BufferedReader* geparkt wird. Auch *yy_reader* ist wieder *private* deklariert.

Vererbung

Will man die Zerlegungsaktivität vererben (kapseln), die Aktionen aber modifizierbar halten, dann bietet sich folgende Architektur an:

```
%%
...
%{
  protected void methode(...) { ... }
%}
%class BaseScanner
...
%%
Muster { methode(...); }
 .           .
 .           .
```

In den Aktionen zu erkannten Mustern wird eine Methode, z.B. mit dem erkannten Text für das Muster, aufgerufen. Die Methode könnte man in der Klasse abstrakt deklarieren oder vorimplementieren. Leider kann man mit *JLex* keine abstrakten Scanner-Klassen generieren...

In Anwendungen für den Scanner erben Entwickler nun die Zerlegungsaktivität, passen aber die Methoden — und damit die Aktionen — der Anwendung an:

```
class RealScanner extends BaseScanner {
  protected void methode(...)  { ... }
  ...
  public RealScanner(java.io.InputStream in) { super(in); }
  public RealScanner(java.io.Reader in) { super(in); }
  ...
}
```

So vermeidet man, daß z.B. die Aktionen zu erkannten Mustern immer ein Objekt liefern, daß aber die Aktivität wieder per switch über diese Objekte bzw. über den Zustand dieser Objekte ausgewählt werden.

Delegierung

Eine andere Technik zur Kapselung der Zerlegungsaktivität für eine Wiederverwendung, stellt die Delegierung der Aktionen an ein Objekt dar. Die Schnittstelle für die Objekte wird durch ein Interface fest vorgegeben. Je nach Implementierung des Interfaces profitieren verschiedenartige Klassen von der Zerlegungsaktivität.

```
...
%%
...
%{
  public interface ActionIF {
    void methode (String s);
    ...
  }
```

```
  protected ActionIF delegate;
  protected void setDelegate(ActionIF delegate) {
    this.delegate = delegate;
  }
  protected ActionIF getDelegate() {
    return delegate;
  }
%}
%class ActionIFScanner
...
%%

Muster { if(delegate!=null)
           delegate.methode(yytext());
       }
 .           .
 .           .
```

Was blieb unerwähnt?

Folgende Punkte wurden in diesem Artikel nicht näher aufgeführt, da dies den Rahmen des Artikels sprengen würde. Bitte lesen Sie die entsprechenden Stellen im *JLex*-Manual[3,4] nach.

- mehrere %-Anweisungen des zweiten Abschnitts.
- Zustände; ein Scanner mit Zuständen finden Sie im Paket mit den Beispielen des Artikels
- *JLex* und Unicode.
- fehlendes Zeilenende bei Verwendung von ^ in regulären Ausdrücken.

Fazit

JLex ist in Java implementiert und folglich auf vielen Plattformen sofort verfügbar und kommt ohne Laufzeit-Support aus. Das Verfahren ist leicht zu erlernen, sehr mächtig und nützlich, nicht nur für Scanner, sondern auch für Filter, die zeilenübergreifend arbeiten sollen. Natürlich ist *JLex* relativ leicht für existente Compiler-Generatoren wie *jay*[5,6] oder *cup*[7] nutzbar.

lex-Programmierer können sehr leicht zu *JLex* umsteigen, da sich am Prinzip und an den regulären Ausdrücken sehr wenig geändert hat. Schwierigkeiten macht am Anfang die Beeinflussung der generierten Klassen- und Methodenköpfen, die über eine Vielzahl von %-Anweisungen erfolgt. Das hätte man eleganter entwerfen können.

Wie immer finden Sie die Beispiele des Artikels zusammen mit *JLex* selber auf dem ftp-Server der *iX* unter *ftp://www.heise.de/pub/ix/ix_listings/2000_03/jlex.tgz*.

Literatur- und Quellenverzeichnis

[1] lex [Lesk 1975] M. E. Lesk, Lex - a lexical analyzer generator, Tech. Rep. Computing Science, Technical Report 39, Bell Laboratories, 1975.

[2] V. Paxson, Flex - Fast lexical analyzer generator, Lawrence Berkeley Laboratory, `ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz`, 1995.

[3] Manual-Seite *JLex*: Sie finden das Manual neben der *JLex*-Quelle im Paket der Beispiel zu diesem Artikel

[4] Homepage *JLex*: `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[5] jay — Compiler bauen mit yacc und Java, iX 10/99, Heise Verlag.

[6] Homepage jay: `http://www.inf.uos.de/jay`

[7] Homepage cup: `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[8] Andrew W. Appel, Modern compiler implementation in Java, Cambridge University Press, 1997.