

Lexer und Parser für eine simple Klammersprache

Die Standardklasse `java.io.StreamTokenizer` soll das Programmieren eines Lexers erleichtern. Die Klasse `Lexer` erweitert die Klasse `StreamTokenizer` und soll das Programmieren eines Lexers noch weiter erleichtern.

Der einzige Konstruktor der Klasse `Lexer` erwartet drei Parameter: Den Pfad der Datei, die geparkt werden soll, ein `HashMap`-Objekt mit Zeichen und ein `HashMap`-Objekt mit Strings, die als Lexeme erkannt und in Token umgewandelt werden sollen. Als Token werden hier `int`-Werte (bzw. `Integer`-Objekte) verwendet.

Ein `Lexer` der Klasse `Lexer` erkennt automatisch *Kommentare* wie in C/C++/Java-Programmen (`/*`-Kommentare und `/* ... */`-Kommentare), *Worte* (die mit einem Buchstaben beginnen und ansonsten aus Buchstaben und Ziffern bestehen und *Zahlen-Literale* (die wie `double`-Literale in Java aussehen müssen).

Ein `Lexer`-Objekt stellt seinen Benutzern im Wesentlichen vier Elemente zur Verfügung: zwei Methoden (`match` und `next`) und zwei Attribute (`tokenNr` vom Typ `int` und `lexem` vom Typ `String`). Die Methode `match` prüft das aktuelle Token, `next` liest das nächste Lexem und wandelt es in ein Token um, `tokenNr` enthält das aktuelle Token und `lexem` das aktuelle Lexem.

Es folgt ein Beispiel für einen Parser, der nach dem Verfahren des *rekursiven Abstiegs* arbeitet und ein Objekt der Klasse `Lexer` als `Lexer` verwendet.

```

1 // ParserTst01.java
2 /* -----
3 Dieses Programm benutzt (und testet) die Klasse Lexer und ist ein
4 Parser fuer eine simple "Klammersprache". Hier ein Beispiel fuer
5 ein korrektes Wort dieser Sprache:
6
7 anfang
8   anna01, (bert02, carl03), 123, (dora04, (3.14, emil)), frida06
9 ende
10
11 Dabei sind anfang und ende Schluesselworte. Die anderen Worte und
12 Zahlenliterale koennen weitgehend beliebig gewaehlt werden (Worte
13 muessen mit einem Buchstaben anfangen und duerfen nur aus Buchstaben
14 und Ziffern bestehen, Zahlenliterale muessen die uebliche Form haben).
15 Zwischen den Worten, Literalen, Klammern und Kommas duerfen beliebig
16 viele (oder wenige) transparente Zeichen (white space, Blank-Zeichen,
17 Tab-Zeichen und Zeilenwechsel) stehen.
18 -----*/
19 import java.util.Map;
20 import java.util.HashMap;
21
22 class ParserTst01 {
23 // -----
24 // Getestet wird das Lexer-Objekt, auf das die Variable lex
25 // spaeter zeigen wird:
26 static Lexer lex;
27
28 // Fuer jedes Schluesselwort ("Wort mit eigener Token-Nr") wird
29 // eine int-Konstante mit seiner Token-Nr vereinbart. Diese
30 // Token-Nr sollte groesser als das groesste ASCII-Zeichen
31 // sein (um Konflikte mit den Sonderzeichen auszuschliessen) und
32 // sich von allen anderen Token-Nrn unterscheiden.
33 static int nr = 256;

```

```

34 static final int ANFANG = nr++; // Fuer das Schluesselwort "anfang"
35 static final int ENDE = nr++; // Fuer das Schluesselwort "ende"
36 // -----
37 static void init() {
38 // Erzeugt und initialisiert das Lexer-Objekt lex:
39
40 // Der Konstruktor der Klasse Lexer erwartet einen String
41 // und 2 Map-Objekte:
42 String pfad = "ParserTst01.txt"; // Die zu parsende Datei
43 Map<String, Integer> worte =
44   new HashMap<String, Integer>(); // Worte mit eigener Token-Nr
45 Map<Character, Integer> zeichen =
46   new HashMap<Character, Integer>(); // Alle erlaubten Sonderzeichen
47
48 worte .put(new String ("anfang"), new Integer(ANFANG));
49 worte .put(new String ("ende"), new Integer(ENDE));
50
51 zeichen.put(new Character('('), new Integer('('));
52 zeichen.put(new Character(')'), new Integer(')'));
53 zeichen.put(new Character(','), new Integer(','));
54
55 lex = new Lexer(pfad, worte, zeichen);
56 lex.next(); // Das erste Token bereitstellen
57 } // init
58 // -----
59 static void Error() {
60   printf("Error: Keine Regel anwendbar in Zeile %d&n", lex.zeilenNr());
61 } // Error
62 // -----
63 // Die Methoden Start, Folge und Rest bilden einen Parser (nach dem
64 // Verfahren des rekursiven Abstiegs) fuer die Sprache mit folgender
65 // Grammatik:
66 //
67 // R01: Start -> ANFANG Folge ENDE
68 // R02: Folge -> '(' Folge ')' Rest
69 // R03: Folge -> WORT Rest
70 // R04: Folge -> ZAHL Rest
71 // R05: Rest -> ',' Folge
72 // R06: Rest -> epsilon
73 //
74 // Dabei sind Start, Folge und Rest Zwischensymbole und alle anderen
75 // Symbole Endsymbole (Token). epsilon bezeichnet die leere Folge.
76 // -----
77 static void Start() {
78   if (lex.tokenNr == ANFANG) {
79     lex.match(ANFANG); // R01
80     Folge();
81     lex.match(ENDE);
82   } else {
83     Error();
84   }
85 } // Start
86 // -----
87 static void Folge() {
88   if (lex.tokenNr == '(') {
89     lex.match('('); // R02
90     Folge();
91     lex.match(')');
92     Rest();
93   } else if (lex.tokenNr == lex.WORT) {
94     lex.match(lex.WORT); // R03

```

```

95     Rest();
96     } else if (lex.tokenNr == lex.ZAHL) {
97         lex.match(lex.ZAHL);           // R04
98         Rest();
99     } else {
100        Error();
101    }
102 } // Folge
103 // -----
104 // Achtung: Die Regel R06 wird dadurch realisiert,
105 // dass in der Methode Rest der if-Befehl *keinen
106 // else-Teil* hat (weil ein Rest auch leer sein
107 // kann).
108 static void Rest() {
109     if (lex.tokenNr == ',') {
110         lex.match(',');               // R
111         Folge();
112     }
113 } // Rest
114 // -----
115 static public void main(String[] _) {
116     printf("ParserTst01: Jetzt geht es los!\n");
117     printf("=====\n");
118
119     init();           // Das Parser-Objekt erzeugen und initialisieren
120     lex.setTrace(true); // Trace-Funktion einschalten
121     Start();         // Das "Startsymbol der Grammatik" aufrufen
122
123     printf("-----\n");
124     if (lex.tokenNr == lex.EOF) {
125         printf("Gut, es wurden alle Eingabezeichen verarbeitet!\n");
126     } else {
127         printf("Schlecht, es sind noch unverarbeitete Zeichen " +
128             "in der Eingabe!\n");
129     } // if
130
131     printf("=====\n");
132     printf("ParserTst01: Das war's erstmal!\n");
133 } // main
134 // -----
135 // Eine Methode mit einem kurzen Namen:
136 static void printf(String f, Object... v) {System.out.printf(f, v);}
137 // -----
138 } // class ParserTst01
139 /* -----
140 Ausgabe des Programms ParserTst01 (fuer die im Anfangskommentar erwaehte
141 Beispieleingabe):
142
143 ParserTst01: Jetzt geht es los!
144 =====
145 Geparst wird die Datei ParserTst01.txt
146 =====
147 1: // Datei ParserTst01.txt
148 2:
149 3: anfang
150 4:  anna01, (bert02, carl03), 123, (dora04, (3.14, emil05)), frida06
151 5: ende
152 =====
153 Token LfdNr:  1, Token-Nr: 256, lexem: anfang
154 Token LfdNr:  2, Token-Nr:  1, lexem: anna01
155 Token LfdNr:  3, Token-Nr: 44, lexem: ,

```

```

156 Token LfdNr:  4, Token-Nr: 40, lexem: (
157 Token LfdNr:  5, Token-Nr:  1, lexem: bert02
158 Token LfdNr:  6, Token-Nr: 44, lexem: ,
159 Token LfdNr:  7, Token-Nr:  1, lexem: carl03
160 Token LfdNr:  8, Token-Nr: 41, lexem: )
161 Token LfdNr:  9, Token-Nr: 44, lexem: ,
162 Token LfdNr: 10, Token-Nr:  2, lexem: 123.0
163 Token LfdNr: 11, Token-Nr: 44, lexem: ,
164 Token LfdNr: 12, Token-Nr: 40, lexem: (
165 Token LfdNr: 13, Token-Nr:  1, lexem: dora04
166 Token LfdNr: 14, Token-Nr: 44, lexem: ,
167 Token LfdNr: 15, Token-Nr: 40, lexem: (
168 Token LfdNr: 16, Token-Nr:  2, lexem: 3.14
169 Token LfdNr: 17, Token-Nr: 44, lexem: ,
170 Token LfdNr: 18, Token-Nr:  1, lexem: emil05
171 Token LfdNr: 19, Token-Nr: 41, lexem: )
172 Token LfdNr: 20, Token-Nr: 41, lexem: )
173 Token LfdNr: 21, Token-Nr: 44, lexem: ,
174 Token LfdNr: 22, Token-Nr:  1, lexem: frida06
175 Token LfdNr: 23, Token-Nr: 257, lexem: ende
176 -----
177 Gut, es wurden alle Eingabezeichen verarbeitet!
178 =====
179 ParserTst01: Das war's erstmal!
180 ----- */

```

In den Zeilen 67 bis 72 stehen die Regeln einer (kontextfreien) Grammatik. Jedem *Zwischensymbol* der Grammatik entspricht eine gleichnamige *Methode* des Parsers (Start, Folge und Rest). Zwischen diesen Methoden und den Zwischensymbolen der Grammatik besteht folgender Zusammenhang:

Die Methode *Start* versucht, eine Zeichenkette einzulesen, die man aus dem Zwischensymbol *Start* ableiten kann (und gibt eine Fehlermeldung aus, wenn das misslingt).

Die Methode *Folge* versucht, eine Zeichenkette einzulesen, die man aus dem Zwischensymbol *Folge* ableiten kann (und gibt eine Fehlermeldung aus, wenn das misslingt).

Die Methode *Rest* versucht, eine Zeichenkette einzulesen, die man aus dem Zwischensymbol *Rest* ableiten kann (und gibt eine Fehlermeldung aus, wenn das misslingt).

Weil man aus dem Zwischensymbol *Rest* (unter anderem) die *leere Symbolfolge* *epsilon* ableiten kann, unterscheidet sich die Methode *Rest* ein bisschen von den anderen Methoden: Ihr *if*-Befehl hat *keinen else-Error()*-Teil.