Primi passi con JFlex

Luca Chiodini

Abstract

Ci proponiamo di muovere i primi passi nel mondo dell'analisi lessicale e della produzione automatica di un analizzatore lessicale. Verrà mostrato l'uso di JFlex attraverso semplici esempi.

1 Analisi lessicale e generatori di lexer

L'analisi lessicale è svolta da un analizzatore lessicale (detto *lexer*). Tale processo è, di fatto, il primo passo svolto da un compilatore. Lo scopo di questo processo è, data in ingresso una sequenza di caratteri, il riconoscimento di pattern attraverso espressioni regolari. Solitamente, l'output è una sequenza di *token*. Definiamo un token come una stringa con una ben precisa interpretazione.

Tuttavia, questa non è la sola modalità in cui può operare un lexer. Infatti, le azioni che esso può intraprendere al riconoscimento di un pattern sono arbitrarie.

Si tratta quindi di scrivere un lexer, che non è altro che un programma realizzato in un certo linguaggio di programmazione (supponiamo, Java). Questo compito però può rivelarsi tedioso e arduo. Ci viene incontro una categoria di software detta *generatori di analizzatori lessicali*. Il loro scopo è generare automaticamente il codice sorgente dell'analizzatore lessicale a partire dalle "descrizioni" (che sono espressioni regolari) dei token.

2 JFlex

JFlex è uno strumento che appartiene alla categoria menzionata in precedenza e produce sorgenti in linguaggio Java. Descriviamo con precisione input e output per JFlex.

L'input consiste in un file `.jflex` che contiene la descrizione delle strutture dei token, specificate mediante le espressioni regolari, e informazioni su come "processare" ciascun token (ovvero quale "azione" intraprendere).

L'output consiste in un sorgente `.java`. Tale programma implementa un automa a stati finiti che riconosce i token e li processa (eseguendo le azioni ad essi associate).

La struttura di un file `.jflex` è la seguente:

```
    /* Codice definito dall'utente */
%%
    /* Opzioni e dichiarazioni */
%%
    /* Regole lessicali */
```

dove

- Il codice definito dall'utente contiene, per esempio, istruzioni di `import` da essere messe all'inizio del sorgente Java generato. Può essere lasciato vuoto.
- Le opzioni contengono:
  - Macro, ovvero espressioni regolari a cui viene assegnato un nome;
  - Codice da includere nella classe Java che verrà generata
- Le regole lessicali sono un'insieme di coppie pattern-azione, dove il pattern è una espressione regolare e l'azione è un pezzo di codice Java da eseguirsi quando l'input è "matchato" dalla espressione regolare.

3 Primo esempio

Vogliamo scrivere un analizzatore lessicale che riconosca le stringhe che consistono in 'a' oppure 'b' e terminano in 'aab'. Definiamo il file `search.flex`:

```
1 %%
2 %standalone
3 %%
4 (a|b)*aab   { System.out.println("Pattern trovato!"); }
5 \n          { /* non fare niente */ }
6 .           { /* non fare niente */ }
```

Il pattern principale in questo caso è `(a|b)*aab`, mentre l'azione è il metodo Java `println()`. Quindi, ogni volta che viene rilevato un match tra la stringa e il pattern, l'azione (quindi `println()`) verrà eseguita.

Per impostazione predefinita, i frammenti di input che non vengono "matchati" da nessun pattern vengono stampati così come sono in output. Le righe 5 e 6 servono a evitare questo comportamento, perché ogni carattere non riconosciuto dal pattern principale verrà riconosciuto dal pattern `.` (ad eccezione di `\n`).

In ultimo, osserviamo che l'opzione `%standalone%` include un metodo `main` nel codice generato, che può così essere compilato ed eseguito.

I seguenti tre comandi

```
1 $ jflex search.flex
2 $ javac Yylex.java
3 $ java YYlex provasearch.txt
```

nell'ordine:

- eseguono JFlex producendo `Yylex.java`;
- compilano `Yylex.java` usando il consueto compilatore `javac`;
- eseguono `YYlex` usando come input `provasearch.txt`.

4 Secondo esempio

Prima di continuare assicurati di aver compreso e provato il primo esempio!

Miglioriamo l'esempio precedente creando il file `search2.jflex`:

```
%%
%class Search
%standalone
%%
(a|b)*aab   { System.out.println("Pattern trovato: " + yytext()); }
\n          { /* non fare niente */ }
.           { /* non fare niente */ }
```

Rispetto alla versione precedente abbiamo aggiunto:

- L'opzione `%class Search` che definisce il nome della classe (e - di riflesso - del file) che JFlex crea;
- Nell'azione viene usato il metodo `yytext()` che restituisce una stringa contenente il frammento di testo che è stato "matchato" dal pattern.

Naturalmente sono disponibili diversi metodi per ricavare informazioni che si possono trovare descritti nella documentazione di JFlex.

Possiamo generare questo nuovo analizzatore lessicale ed eseguirlo:

```
$ jflex search2.flex
$ javac Search.java
$ java Search provasearch.txt
```

5 Ulteriori dettagli su JFlex

- La sintassi delle espressioni regolari usate da JFlex è equivalente alla sintassi UNIX, che si può trovare ampiamente descritta sulla rete.
- Quando più regole possono essere applicate, si preferisce quella con il matching più lungo. A parità di lunghezza, l'ordine con cui sono specificate le regole determina quale viene applicata.

6 Terzo esempio

Prima di continuare assicurati di aver compreso e provato i primi due esempi!

Presentiamo ora un terzo esempio che chiamiamo `ContaLinee.jflex`:

```
1  %%
2
3  %class ContaLinee
4  %standalone
5
6  %{
7  int linee = 1;
8  %}
9
10 LINEA = .*\n
11
12 %%
13
14 {LINEA} { System.out.print("Nuova linea trovata: " + yytext());
15           System.out.println("Numero linea: " + linee);
16           linee++;
17         }
```

Abbiamo definito una macro di nome `LINEA` che corrisponde alla espressione regolare `.*\n` (qualsiasi numero di caratteri terminanti con un fine riga). Il pattern dell'unica regola definita in questo file (riga 14) richiama quindi la macro rendendo molto pulito il codice.

C'è un'altra novità in questo esempio: alla riga 7 abbiamo definito una variabile che viene poi utilizzata nell'azione. Questo è possibile grazie al fatto che JFlex copia all'interno della classe quanto racchiuso tra `%{` e `%}`.

Si può generare l'analizzatore lessicale, compilarlo e testarlo in analogia a quanto visto negli esempi precedenti.

7 Quarto esempio

> Prima di continuare assicurati di aver compreso e provato i primi tre esempi!

Presentiamo ora un ultimo esempio che fa uso del concetto di stati. L'analizzatore lessicale che viene generato da JFlex non è altro che l'implementazione di un automa a stati finiti. Finora abbiamo completamente ignorato gli stati, lasciando l'automa sempre nell'unico stato iniziale presente per impostazione predefinita. Tale stato si chiama `YYINITIAL`.

È possibile definire nuovi stati (in realtà di due tipi diversi, ma la loro trattazione separata va oltre le finalità di questa breve introduzione).

Analizziamo il file `TrovaCommenti.jflex`, che stampa i commenti nella forma `/* commento */` estratti da un ipotetico sorgente:

```
%%

%class TrovaCommenti
%standalone
%state COMMENTO

INIZIOCOMMENTO = "/*"
FINECOMMENTO = "*/"
```

```
9
10 %%
11
12 <YYINITIAL> {
13     {INIZIOCOMMENTO} { yybegin(COMMENTO); }
14     [^] { }
15 }
16
17 <COMMENTO> {
18     .* {FINECOMMENTO} {
19         System.out.println("Trovato commento: " + yytext().substring
               (0, yylength() - 2));
20         yybegin(YYINITIAL);
21     }
22 }
```

È stato definito un nuovo stato di nome `COMMENTO`. Quando l'automa si trova nello stato iniziale e il frammento di input viene rilevato essere l'inizio di un commento, l'automa cambia stato usando il metodo `yybegin(NUOVO_STATO)`.

Nel caso di esempio, si entra nello stato `COMMENTO` da cui si esce dopo una sequenza qualsiasi di caratteri terminante con il pattern di fine commento. A quel punto si eseguono le due azioni associate, ovvero viene stampato il commento trovato (riga 19) eliminando gli ultimi due caratteri che rappresentano il fine commento e viene riportato l'automa nello stato iniziale con `yybegin(YYINITIAL)`.