Capitolo 2

Analisi Lessicale

L'analisi lessicale scompone un sorgente Kitten in una sequenza di *token*. Ogni token rappresenta un insieme di stringhe. In questo capitolo vedremo come creare un analizzatore lessicale che riconosce un dato insieme di token, specificato da una lista di espressioni regolari. Useremo a tal fine il generatore `JLex` di analizzatori lessicali, che è una versione Java del programma `lex` inizialmente sviluppato per il linguaggio C.

2.1 I token Kitten

Abbiamo detto che un token rappresenta un insieme di stringhe. Per esempio, il token `THEN` rappresenta l'insieme di stringhe {`then`}, mentre il token `ID` rappresenta l'insieme (potenzialmente infinito) delle stringhe che sono *identificatori* Kitten, cioè nomi di variabili, classi, campi e metodi. I token che rappresentano più di una stringa, come `ID`, hanno associato un *valore lessicale*. Si consideri per esempio la classe `Led.kit` in Figura 1.4. Il risultato della sua analisi lessicale è in Figura 2.1. Si noti come le parole chiave del linguaggio, come `class`, `field`, `void`, `boolean`, sono rappresentate da un token specifico. Gli identificatori sono invece rappresentati dall'unico token `ID` con associato un valore lessicale, cioè la stringa dell'identificatore. Si sarebbe potuto rappresentare anche le parole chiave come identificatori con associato un valore lessicale. Per esempio, si poteva usare `ID(method)` piuttosto che `METHOD`. La scelta dei token è in parte arbitraria, ma è in effetti pensata per semplificare la successiva ricostruzione della struttura grammaticale del testo, nella fase di analisi sintattica. A quel punto, sarà più semplice avere a che fare con `METHOD` piuttosto che `ID(method)`. In Figura 2.1 si noti come anche le parentesi tonde sono rappresentate da due appositi token (`LPAREN` ed `RPAREN`), così come le parentesi graffe (`LBRACE` ed `RBRACE`). Lo stesso accade per le parentesi quadre (`LBRACK` ed `RBRACK`), non mostrate in figura. Alla fine è presente il token fittizio `EOF` (Sezione 2.6), che segnala la fine del file sorgente. Si noti che spazi, tabulazioni e commenti sono assenti in Figura 2.1. Essi vengono infatti scartati dall'analizzatore lessicale (Sezione 2.5).

Per rappresentare un token, usiamo la classe in Figura 2.2, che fa parte del programma `JavaCup`. Esso è un generatore di analizzatori sintattici. Il motivo per cui usiamo tale classe è che così facendo sarà più semplice interfacciare il nostro analizzatore lessicale con l'analizzatore sintattico che costruiremo in futuro, dal momento che entrambi utilizzano la stessa classe. I codici `sym` dei token sono enumerati dentro la classe `syntactical/sym.java`. Anche tale file fa parte del generatore di analizzatori sintattici in modo che sia l'analizzatore lessicale che quello sintattico usino gli stessi codici per i token. La Figura 2.2 mostra che di ogni token è possibile

```
CLASS from 0 to 4
ID(Led) from 6 to 8
LBRACE from 10 to 10
FIELD from 14 to 18
BOOLEAN from 20 to 26
ID(state) from 28 to 32
CONSTRUCTOR from 37 to 47
LPAREN from 48 to 48
RPAREN from 49 to 49
LBRACE from 51 to 51
RBRACE from 52 to 52
METHOD from 57 to 62
VOID from 64 to 67
ID(on) from 69 to 70
LPAREN from 71 to 71
RPAREN from 72 to 72
ID(this) from 78 to 81
DOT from 82 to 82
ID(state) from 83 to 87
ASSIGN from 89 to 90
TRUE from 92 to 95
METHOD from 100 to 105
VOID from 107 to 110
ID(off) from 112 to 114
LPAREN from 115 to 115
RPAREN from 116 to 116
ID(this) from 122 to 125
DOT from 126 to 126
ID(state) from 127 to 131
ASSIGN from 133 to 134
FALSE from 136 to 140
METHOD from 145 to 150
BOOLEAN from 152 to 158
ID(isOn) from 160 to 163
LPAREN from 164 to 164
RPAREN from 165 to 165
RETURN from 171 to 176
ID(this) from 178 to 181
DOT from 182 to 182
ID(state) from 183 to 187
METHOD from 192 to 197
BOOLEAN from 199 to 205
ID(isOff) from 207 to 211
LPAREN from 212 to 212
RPAREN from 213 to 213
RETURN from 219 to 224
NOT from 226 to 226
ID(this) from 227 to 230
DOT from 231 to 231
ID(state) from 232 to 236
RBRACE from 238 to 238
EOF from 239 to 239
```

Figura 2.1: Il risultato dell'analisi lessicale della classe in Figura 1.4.

conoscere la posizione `left` a cui inizia, espressa in numero di caratteri dall'inizio del file, commenti inclusi, e la posizione `right` a cui finisce. Nonché l'eventuale valore lessicale `value`, se c'è. Per esempio, il token `ID(Led)` in Figura 2.1 ha `left` pari a 6, `right` pari a 8 e `value` legato alla stringa `Led`. I token che non hanno valore lessicale avranno `value` pari a `null`.

2.2 La generazione dell'analizzatore lessicale

Quello che vogliamo ottenere è un analizzatore lessicale per i token Kitten. Al posto di generare tutta la sequenza di token, come in Figura 2.1, e poi passarli all'analizzatore sintattico, è molto più economico generare un token alla volta e

```
public class Symbol {
  public int sym;       // il codice che identifica il token
  public int left;      // il carattere a cui inizia
  public int right;     // il carattere a cui finisce
  public Object value;  // il valore lessicale associato, se esiste
  ...
}
```

Figura 2.2: La classe `java_cup.runtime.Symbol.java`, che rappresenta un token Kitten.

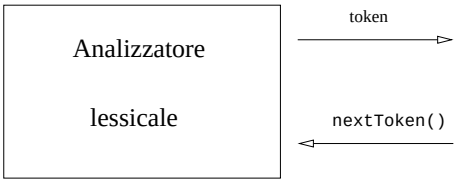

Figura 2.3: L'interfaccia dell'analizzatore lessicale

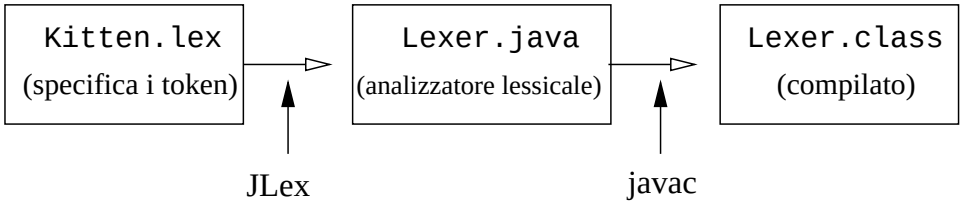

Figura 2.4: La generazione dell'analizzatore lessicale per Kitten

passarlo all'analizzatore sintattico. Anche quest'ultimo però dovrà essere capace di lavorare con un token alla volta. L'interfaccia dell'analizzatore lessicale sarà quindi quella mostrata in Figura 2.3. Il metodo `nextToken()` restituisce un token alla volta.

Genereremo l'analizzatore lessicale per Kitten in modo automatico, a partire dalla specifica dell'analizzatore lessicale e dei token che deve riconoscere. A tal fine useremo un programma Java di nome `JLex`. La Figura 2.4 mostra il modo in cui generiamo l'analizzatore lessicale usando `JLex`. Inizialmente, scriviamo dentro il file `lexical/Kitten.lex` una specifica dell'analizzatore lessicale e dei token Kitten. Tale specifica è scritta in un linguaggio comprensibile dal programma `JLex`. Per esempio, essa contiene la specifica della funzione `nextToken()` che vogliamo generare (Figura 2.3) e il suo tipo di ritorno:

```
%function nextToken
%type java_cup.runtime.Symbol
```

Inoltre, come vedremo fra poco, dovrà contenere una sequenza di espressioni regolari con associato un pezzo di codice Java che viene eseguito quando viene riconosciuto il token corrispondente a ciascuna espressione regolare. Normalmente, tale codice Java non fa altro che sintetizzare il token opportuno (cioè un oggetto della classe `java_cup.runtime.Symbol` in Figura 2.2) e restituirlo. Fornendo al programma `JLex` la specifica `lexical/Kitten.lex` dei token, otteniamo un programma Java di nome `lexical/Lexer.java`. Tale programma è l'analizzatore lessicale e include al suo interno una traduzione in Java dell'automa che riconosce i token enumerati dentro `lexical/Kitten.lex`. Quando ciascun token viene riconosciuto, esso esegue il codice Java che avevamo associato al token dentro `lexical/Kitten.lex`. Il programma `lexical/Lexer.java` viene quindi compilato come un qualsiasi programma Java, ottenendo l'analizzatore lessicale che cercavamo. Programmi generati in maniera automatica, come `lexical/Lexer.java`, sono normalmente di difficile lettura. Fidiamoci quindi del risultato, e descriviamo invece il contenuto del file `lexical/Kitten.lex`.

2.3 La specifica dei token

Abbiamo detto che il file `lexical/Kitten.lex` contiene la descrizione dei token Kitten, sotto forma di espressioni regolari con associata un'azione di sintesi del token corrispondente. Per esempio esso contiene le seguenti coppie espressione regolare/azione:

```
<YYINITIAL>"while"      {return tok(sym.WHILE, null);}
<YYINITIAL>"for"        {return tok(sym.FOR, null);}
<YYINITIAL>"break"      {return tok(sym.BREAK, null);}
<YYINITIAL>"continue"   {return tok(sym.CONTINUE, null);}
....
<YYINITIAL>"+"          {return tok(sym.PLUS, null);}
<YYINITIAL>"-"          {return tok(sym.MINUS, null);}
<YYINITIAL>"*"          {return tok(sym.TIMES, null);}
<YYINITIAL>"/"          {return tok(sym.DIVIDE, null);}
<YYINITIAL>"="          {return tok(sym.EQ, null);}
<YYINITIAL>"<>"         {return tok(sym.NEQ, null);}
<YYINITIAL>"<"          {return tok(sym.LT, null);}
<YYINITIAL>"<="         {return tok(sym.LE, null);}
<YYINITIAL>">="         {return tok(sym.GE, null);}
<YYINITIAL>">"          {return tok(sym.GT, null);}
<YYINITIAL>"&"          {return tok(sym.AND, null);}
<YYINITIAL>"|"          {return tok(sym.OR, null);}
<YYINITIAL>"!"          {return tok(sym.NOT, null);}
<YYINITIAL>":="         {return tok(sym.ASSIGN, null);}
....
```

che riconoscono rispettivamente i token `WHILE`, `FOR`, `BREAK`, `CONTINUE`, il segno di addizione ecc. La sintassi ‘‘`while`’’ è un'espressione regolare che va intesa come `w` · `h` · `i` · `l` · `e`, cioè la concatenzazione sequenziale di cinque caratteri. La notazione `<YYINITIAL>` specifica che queste regole sono attive quando l'analizzatore lessicale è nella *modalità* di default `YYINITIAL`. Parleremo più tardi delle modalità (Sezione 2.5). Per adesso ci basta sapere che, all'inizio, l'analizzatore lessicale è in modalità `YYINITIAL`, per cui le regole precedenti sono attive. L'azione corrispondente a ciascuna regola, che viene eseguita quando il token corrispondente è stato riconosciuto, è fra parentesi graffe. Si tratta di codice Java. Per adesso, esso sintetizza il token corrispondente alle espressioni regolari, usando `null` come valore lessicale. Il metod `tok` non fa altro che costruire un oggetto della classe in Figura 2.2:

```
private java_cup.runtime.Symbol tok(int kind, Object value) {
  return new java_cup.runtime.Symbol
    (kind,yychar,yychar + yylength(),value);
}
```

La variabile `yychar` contiene il numero di caratteri tra l'inizio del file e l'inizio del token. Il metodo `yylength()` ritorna la lunghezza del token riconosciuto. Metodi di ausilio, come quello precedente, sono inseriti in `lexical/Kitten.lex` fra i delimitatori `%{` e `}%` e vengono ricopiati testualmente da `JLex` dentro `lexical/Lexer.java`.

I token che hanno un valore lessicale sono specificati in maniera appena più complicata:

```
<YYINITIAL>[a-zA-Z][a-zA-Z0-9_]*
    {return tok(sym.ID,yytext());}
<YYINITIAL>[0-9]+
```

```
public class ErrorMsg {
  /* costruttore: si chiede il nome del file che si sta compilando */
  public ErrorMsg(String fileName) { ... }

  /* chiamata quando si incotra un newline in fileName */
  public newline(int pos) { ... }

  /* segnala un errore Msg alla posizione pos dall'inizio di fileName */
  public error(int pos, String Msg) { ... }

  /* dice se si e' verificato almeno un errore */
  public boolean anyErrors() { ... }
}
```

Figura 2.5: La classe `errorMsg.ErrorMsg.java` per la segnalazione di errori.

```
    {return tok(sym.INTEGER,new Integer(yytext()));}
<YYINITIAL>[0-9]*"."[0-9]+
    {return tok(sym.FLOATING,new Float(yytext()));}
```

Si noti come il valore lessicale degli identificatori `ID` sia la stringa che rappresenta l'identificatore, mentre per interi e numeri a virgola mobile si tratti, rispettivamente, di un oggetto di classe `java.lang.Integer` e `java.lang.Float`. Il programma `JLex` accorda maggiore priorità alle regole specificate prima in `lexical/Kitten.lex`. Al fine di non fare riconoscere, per esempio, la parola chiave `while` come un identificatore, occorre quindi mettere la regola per il `while` prima di quella per gli identificatori.

Esiste infine una regola che riconosce qualsiasi carattere ma che, essendo messa alla fine, viene eseguita solo quando nessun'altra regola è applicabile. Tale regola segnala un'errore lessicale, cioè la lettura di un carattere che non è associabile ad alcun token:

```
<YYINITIAL>.            {errorMsg.error(yychar,"Unmatched input");}
```

2.4 La segnalazione di errori

Abbiamo appena visto che l'analizzatore lessicale può aver bisogno di segnalare un errore all'utente di Kitten. Lo stesso (e molto più spesso) accadrà con l'analizzatore sintattico e con quello semantico. Tutti questi analizzatore usano la stessa classe `errorMsg.ErrorMsg.java` per segnalare errori. La sua interfaccia è in Figura 2.5. Il metodo `error()` segnala un errore all'utente. La posizione dell'errore è indicata all'utente con la notazione `riga:colonna`. Ma il metodo `error()` richiede solo il numero `pos` di caratteri passati dall'inizio del file che si sta compilando. Per potere trasformare `pos` in `riga:colonna`, occorre che l'oggetto di segnalazione di errori sia al corrente di dove, nel file sorgente, si trovano i caratteri di newline. Ecco perché, ogni volta che si incontra tale carattere, l'analizzatore lessicale chiama il metodo `newline()`:

```
<YYINITIAL>\n           {errorMsg.newline(yychar);}
```

Il campo `errorMsg` contiene la struttura di segnalazione di errore dell'analizzatore lessicale. Essa è creata dal costruttore dell'analizzatore lessicale a partire dal nome del file che si sta compilando. Si noti che la regole precedente scarta il carattere newline, poiché non vogliamo i caratteri di spaziatura nel risultato dell'analisi lessicale (Figura 2.1).

2.5 Modalità lessicali: commenti e stringhe

Le regole contenute in `lexical/Kitten.lex` hanno come prefisso una *modalità* che indica quando esse sono attive. Normalmente, l'analizzatore lessicale è nella modalità di default `YYINITIAL`. È possibile però cambiare modalità tramite il metodo `yybegin()`. Occorre per prima cosa dichiarare le nuove modalità dentro `lexical/Kitten.lex`:

```
%state COMMENT
%state STRING
```

La scelta di queste due modalità è finalizzata a semplificare la gestione di commenti e stringhe. Non è strettamente necessario usare le modalità, ma il loro uso è in molti casi di grande ausilio. Per esempio, la modalità `COMMENT` si attiva quando incontriamo la sequenza di caratteri `/*`:

```
<YYINITIAL>"/*"         {commentCount++; yybegin(COMMENT);}
```

La variabile `commentCount` conta il livello di annidamento dei commenti visti fino a questo momento. Essa è dichiarata fra i delimitatori `%{` e `%}` e inizializzata a 0.

Le uniche regole attive in modalità `COMMENT` hanno come scopo di scartare tutti i caratteri letti fino alla chiusura dell'ultimo commento, tenendo conto dell'annidamento. Non dobbiamo però dimenticare di registrare le posizioni dei caratteri di newline:

```
<COMMENT>"*/"           {commentCount--;
                         if (commentCount == 0) yybegin(YYINITIAL);}
<COMMENT>"/*"           {commentCount++;}
<COMMENT>\n             {newline();}
<COMMENT>.              {}
```

La modalità `STRING` si attiva quando si incontra un carattere di doppio apice. Essa riconosce la stringa fra doppi apici e processa le sequenze di escape. Memorizza il valore lessicale dentro una variabile `myString` che viene usata come valore lessicale del token `STRING`:

```
<YYINITIAL>"\"" {myString = ""; yybegin(STRING);}
<STRING>\\n     {myString += "\n";}
<STRING>\\t     {myString += "\t";}
... altre sequenze di escape ...
<STRING>"\""    {yybegin(YYINITIAL); return tok(sym.STRING,myString);}
<STRING>\n      {errorMsg.newline(yychar); myString += "\n";}
<STRING>.       {myString += yytext();}
```

La seconda e la terza regola inseriscono un carattere di escape dentro la stringa quando viene riconosciuto la corrispondente espressione di escape dentro il file sorgente. Si noti che l'espressione regolare `\\n` è formata da *due* caratteri `\\` e `n`. Il primo è a sua volta una sequenza di escape di `JLex` per esprimere il carattere `\`. Esistono altre espressioni di escape per inserire, per esempio, un carattere a partire dal suo codice ASCII. Esse sono esaminabili dentro `lexical/Kitten.lex`. La terzultima regola riporta l'analizzatore in modalità `YYINITIAL` quando si vede il secondo carattere di doppio apice. Le ultime due regole accumulano tutti caratteri dentro `myString`.

2.6 La fine del file sorgente

Quando l'analizzatore lessicale giunge alla fine del file sorgente, viene eseguito il codice specificato, dentro `lexical/Kitten.lex`, fra i delimitatori `%eofval{` e `%eofval}`. Nel nostro caso abbiamo scelto di eseguire quanto segue:

```
%eofval{
        {
          if (commentCount != 0) err("Unclosed comment");
          return tok(sym.EOF, null);
        }
%eofval}
```

ovvero controlliamo che il file non termini con un commento ancora aperto e poi restituiamo comunque il token fittizio `EOF`.

2.7 L'uso di JLex

Una volta inserita dentro `lexical/Kitten.lex` la specifica dell'analizzatore lessicale che vogliamo generare, non ci rimane che generarlo tramite `JLex`. A tal fine basta eseguire `JLex`:

```
java JLex.Main lexical/Kitten.lex
```

Per default, `JLex` scrive l'analizzatore lessicale nel file `lexical/Kitten.lex.java`, che contiene la dichiarazione di una classe di nome `Yylex`. Basta quindi ridenominare tale file in `lexical/Lexer.java`, dopo avere però sostituito tutte le stringhe `Yylex` al suo interno con `Lexer`. A tal fine usiamo il programma Unix `sed`:

```
cat lexical/Kitten.lex.java | sed s/Yylex/Lexer/g > lexical/Lexer.java
rm lexical/Kitten.lex.java
```

Questi tre comandi sono stati inseriti dentro a un makefile al fine di seplificarne l'uso. Basta quindi scrivere `make lexical` per ottenere lo stesso effetto.

In conclusione, abbiamo ottenuto un analizzatore lessicale `lexical/Lexer.java` che contiene un costruttore

```
public Lexer(fileName)
```

e un metodo

```
public java_cup.runtime.Symbol nextToken()
```

che estrae un token alla volta da `fileName` e lo restituisce.