Integración de JFlex y CUP (analizadores léxico y sintáctico)

Rafael A. Vega Castro

Octubre de 2008

Resumen

El siguiente artículo trata sobre cómo si se tiene por un lado un generador de analizadores léxicos (JFlex) y por otro lado un generador de analizadores sintácticos (CUP), se pueden integrar cada uno de los procesos de dichos metacompiladores para de esta forma obtener un compilador mucho mas completo y personalizado, esto ahorra de manera abismal la generación de código al momento de programar un compilador.

The next article is about how if you have in one hand a lexical analysis (JFlex) and in the other hand a parser generator (CUP), you can integrate the process of the tow analyzers and get a completely compiler software, with these you can save thousands of programming time and lines of algorithms code.

JFlex es un metacompilador que permite generar rapidamente analizadores léxicos que se integran con Java.

1. Características de JFlex

Las características sobre salientes de JFlex son las siguientes:

- Soporte completo con caracteres unicode[1]
- Permite generar analizadores léxicos rápidamente.
- Tiene una sintaxis cómoda de manipular y fácil de interpretar.
- Es independiente de la plataforma debido a que esta diseñado para ser integrado con Java.
- Permite la integración con CUP (Analizador sintáctico)

2. Estructura un archivo de JFlex

Un archivo jflex esta dividido en 3 secciones:

- Opciones y declaraciones
- Código de usuario
- Reglas lexicográficas

2.1. Opciones y declaraciones

La primera parte del archivo es el bloque donde se importaran los paquete que se van a utilizar para nuestro analizador, es decir, si en nuestro programa utilizaremos componentes del paquete *util* debemos importar aquí dicho paquete:

```
import java.util.*;
```

Luego sigue un par de signos de porcentaje (%) para indicar que empezará la definición del bloque de configuración del analizador. El bloque de configuración se define por el conjunto de parámetros que

[1] **Unicode**, es una gran tabla, que en la actualidad asigna un código a cada uno de los más de cincuenta mil símbolos, los cuales abarcan todos los alfabetos europeos, ideogramas chinos, japoneses, coreanos, muchas otras formas de escritura, y más de un millar de símbolos especiales.

se especifican en el archivo para decirle a nuestro a analizador como se debe comportar, cada parámetro empieza con el símbolo % y se escribe solo uno por linea, es decir, uno de bajo del otro.

```
%unicode
%line
%column
```

2.2. Código de usuario

En el siguiente fragmento de código podremos incluir código Java el cual podemos utilizar en el analizador, cabe notar que el código que aquí se escriba será incluido sin ninguna alteración al resultado final del analizador, dicho fragmento ira enmarcado entre las etiquetas %{ al inicio del código y %} al final del mismo.

```
%{

  public static void escribir(String cadena)
  {
    System.out.println(cadena);
  }

%}
```

2.3. Reglas lexicográficas

El siguiente fragmento formará parte esencial dentro del funcionamiento del analizador, en este se definirán el conjunto de expresiones regulares que se utilizarán durante el proceso de análisis, a continuación se presentan unos ejemplos de este tipo de declaraciones:

```
FinDeLinea = \r | \n | \r\n
Variable = [:jletter:][:jletterdigit:]*
Si = “Si”
Entero = 0 | [1-9][0-9]*
```

En la próxima línea de código se especifican los posible estados en

los que se encontrará el analizador, esta funcionalidad es muy útil a la hora de identificar el contexto en el que se esta realizando el análisis.

`%state CADENA` La última parte de código es donde se le especificará al analizador los tokens y que acciones realizar cuando se encuentren dichos tokens, para inicial este fragmento debemos insertar nuevamente un doble símbolo de porcentaje (%%):

```
"Inicio" {System.out.println("Palabra Inicio")}
"+" {System.out.println("Símbolo +")}
{FinDeLinea} {System.out.println("Fin de línea")}
```

2.4. Parámetros y Opciones

Los siguientes son los posibles parámetros y/o opciones que pueden especificar en un archivo jflex.

- **%class Lexer:** Esta opción le dice a JFlex que el archivo .java a generar lleve el nombre de *Lexer*, aquí podremos indicar cualquier nombre.
- **%unicode:** La opción unicode nos permite trabajar con archivos que tienen este tipo de caracteres.
- **%cup:** CUP es un analizador sintactico el cual se mostrará mas adelante, esta opción nos permite integrar JFlex y CUP.
- **%line:** Le dice al analizador que lleve el conteo de la linea que se esta analizando.
- **%line:** Le dice al analizador que lleve el conteo de la columna que se esta analizando.

2.5. Reglas y acciones

La sección de "Reglas léxicas" de JFlex contiene expresiones regulares y acciones (Código Java) que son ejecutadas cuando el analizador encuentra cadenas asociadas a las expresiones regulares. Así como el escáner lee las entradas, también hace seguimiento a todas las expresiones regulares y activa la acción del patrón que tenga la mas grande coincidencia, por ejemplo, para la especificación anterior, la cadena *Sino* coincide en parte con la expresión *Si*, pero como la expresión

regular *Variable* tiene mayor numero de coincidencias (coincide totalmente) entonces es esta ultima quien ejecuta la acción. Para el caso en que dos expresiones regulares coincidan en su totalidad con una cadena entrante entonces es el patrón que se ha especificado primero el que ejecuta la acción.

Estados léxicos, Adicionalmente a las expresiones regulares, se pueden utilizar estados léxicos para hacer las especificaciones mas exactas. Un estado léxico como una condición de inicio, una cadena, entre otros. Si el escáner se encuentra en estado CADENA, entonces solo las expresiones regulares que se estén precedidas por la condición inicial <CADENA>podrán ser evaluados. La condición inicial de una expresión regular puede contener mas de un estado léxico, de ser así, dicha expresión regular se evaluará cuando el escáner se encuentre en cualquiera de esos estados iniciales. El estado léxico inicial del escáner es YYINITIAL y es con este el cual se empieza a escanear, si una expresión regular no tiene condición inicial esto quiere decir que podrá ser evaluada en cualquier estado léxico que se encuentre el escáner.

```
<YYINITIAL>"Inicio" {System.out.println("Palabra Inicio")}
```

Dos métodos importantes que se utilizan en el código de acción de una expresión regular son *yybegin* y *yytext*. **yybegin** se utiliza para decirle al escáner que cambie el estado léxico, por ejemplo: `yybegin(CADENA)`, esto indica al escáner que a partir del llamado de esa función el escáner se encontrará en el estado léxico CADENA, por otro lado tenemos **yytext** la cual devuelve la entrada la cual coincidió con la respectiva expresión regular.

3. Codificación de caracteres

- **%7bit**
  Permite usar entradas de 7 bits, es decir entre 0-127. Si una entrada es mayor que 127 se generará un error en tiempo de ejecución y se lanzará una excepción de tipo `ArrayIndexOutofBoundsException`.
- **%full**
  %8bit
  Permite usar entradas de 8 bits, es decir entre 0-255. Si una entrada es mayor que 255 se generará un error en tiempo de ejecución y

se lanzará una excepción de tipo `ArrayIndexOutofBoundsException`.

- **%unicode**
  %16bit
  Esto significa que la entrada puede ser cualquier tipo de caracter, es decir, entre 0-65535. Para esta directiva no se produce ningún tipo de error en tiempo de ejecución.

4. Reglas léxicas

Las *reglas léxicas* contienen un conjunto de expresiones regulares y acciones.

4.1. Definición de la sintaxis

La sintaxis de las *reglas léxicas* de JFlex esta descrita por la siguiente gramática:

```
LexicalRules ::= Rule+
Rule ::= [StateList] ['∧'] RegExp [LookAhead] Action
         | [StateList] '«EOF»' Action
         | StateGroup
StateGroup ::= StateList '{'Rule+ '}'
StateList ::= '<' Identifier (',' Identifier)* '>'
LookAhead ::= '$'| '/'RegExp
Action ::= '{'JavaCode '}'| '|'
RegExp ::= RegExp '|'RegExp
         | RegExp RegExp
         | '('RegExp ')'
         | ('!'|'∼') RegExp
         | RegExp ('*'|'+'|'?')
         | RegExp "{"Number [","Number] "}"
         | '['['∧'] (Character|Character'-'Character)* ']'
         | PredefinedClass
         | '{' Identifier '}'
         | '"'StringCharacter+ '"'
         | Character
PredefinedClass ::= '[:jletter:]'
            | '[:jletterdigit:]'
            | '[:letter:]'
```

```
| '[:digit:]'
| '[:uppercase:]'
| '[:lowercase:]'
| '.'
```

La gramática usa los siguiente símbolos terminales:

- **JavaCode**
  Es una secuencia que describe las especificaciones del lenguaje Java.
- **Number** Un número entero no negativo.
- **Identifier**
  Una secuencia de letras seguidas de cero o mas letras, digitos o rallas al pie (_);
- **Character**
  Una secuencia de caracteres que no sean ninguno de los siguientes:
  | ( ) { } [ ] < > \ . * + ? ∧ $ / " ∼ -
  !
- **StringCharacter**
  Una secuancia de caracteres que no sean ninguno de los siguientes:
  \ "

4.2. Operadores en las expresiones regulares

Ahora se mostrarán los operadores que se pueden utilizar en la definición de expresiones regulares en el analizador léxico JFlex.
Sean a y b expresiones regulares, entonces:

- **a** | **b** *Unión*
  Es una expresión regular que encuentra todas las entradas que sean validas para *a* ó *b*.
- **a b** *Concatenación*
  Es una expresión regular que encuentra todas las entradas que sean validas para *a* seguida de *b*.
- **a*** *Cerradura de Kleene*
  Es una expresión regular que encuentra todas las entradas que sean validas para cero o mas repeticiones de *a*.

- **a+** *Iteración*
  Es una expresión regular que encuentra todas las entradas que sean validas para una o mas repeticiones de *a*. Es equivalente a `aa*`
- **a?** *Opción*
  Es una expresión regular que encuentra todas las entradas que sean validas para cero o una ocurrencia de *a*.
- **!a** *Negación*
  Es una expresión regular que encuentra todas las entradas que sean validas para cualquier expresión diferente de *a*.
- **a{n}** *Repetición*
  Es una expresión regular que encuentra todas las entradas que sean validas para exactamente n repeticiones de *a*.
- **a{n}{m}**
  Es una expresión regular que encuentra todas las entradas que sean validas para entre n y m repeticiones de *a*.
- **a**
  Es una expresión regular que encuentra todas las entradas que coincidan exactamente con *a*.

4.3. Precedencia de operadores

JFlex usa la siguiente precedencia de operadores estándar para expresiones regulares:

- Operadores unarios posfijos ('*', '+', '?', {n}, {n,m})
- Operadores unarios prefijos ('!', '∼')
- Concatenación (RegExp::= RegExp Regexp)
- Unión (RegExp::= RegExp '|'RegExp)

Entonces la expresión `a | abc | !cd*` terminará convertida en `(a|(abc)) | ((!c)(d*))`.

5. Métodos y atributos de JFlex asequibles en el código de acción

Todos los métodos y atributos asequibles para el código de acción de JFlex tiene el prefijo *yy* para evitar que al momento de que el usuario

los use generen conflicto con otros métodos o atributos de las demás clases. JFlex no tiene métodos ni atributos públicos o privados, por el contrario utiliza los prefijos *yy* y *zz* para indicar que estos pueden ser usados en el código de acción o para el código interno respectivamente. El API actual que se puede utilizar en el código de acción es:

- `String yytext()`
  Devuelve la cadena que coincidió con la respectiva expresión regular.
- `int yylength()`
  Devuelve el tamaño de la cadena que coincidió con la respectiva expresión regular.
- `char yycharat(int pos)`
  Devuelve el carácter que se encuentra en la posición *pos* de la cadena que coincidió con la respectiva expresión regular.
- `void yyclose()`
  Cierra el flujo de la entrada de datos, por tanto a todas las entradas siguientes arrojara fin de archivo.
- `int yystate()`
  Devuelve el estado léxico actual del analizador.
- `void yybegin(int estado)`
  Cambia el estado léxico actual del analizador por el nuevo estado especificado como parámetro.
- `int yyline`
  Contiene el número de la línea en la que se encuentra el analizador. (Solo si se incluyó la directiva %line)
- `int yychar`
  Contiene el número del carácter que se esta analizando.
- `int yycolumn`
  Contiene el número de la columna en la que se encuentra el analizador. (Solo si se incluyó la directiva %column)

CUP es un metacompilador utilizado para generar un analizadores sintácticos ascendentes con algoritmos LALR. CUP es el homólogo para Java del programa YACC utilizado en C.

6. Sintaxis de un archivo .CUP

Un archivo de entrada CUP consta de las siguientes cinco partes:

1. Definición de paquete y sentencias `import`.
2. Sección de código de usuario.
3. Declaración de símbolos terminales y no terminales.
4. Declaraciones de precedencia.
5. Definición del símbolo inicial de la gramática y reglas de producción.

6.1. Sentencias `import` e inclusión de paquete

Las especificaciones comienzan de forma opcional con las directivas `package` y `import` Estas tienen la misma sintaxis y juegan el mismo rol que el `package` y el `import` de un programa normal escrito en Java. La declaración de un paquete tiene la forma:

```
package nombre_del_paquete;
```

donde nombre_del_paquetes es el nombre del paquete al que se esta incluyendo la clase en Java. En general, CUP implementa las convenciones léxicas de Java, como por ejemplo, soporta los dos tipos de comentarios que soporta Java.

Después de la declaración opcional de `package`, luego se pueden importar cero o mas paquetes Java. Al igual que un programa escrito en Java importar un paquete se hace de la forma:

```
import nombre_del_paquete.nombre_de_la_clase;
```

o

```
import nombre_del_paquete.*;
```

En general, la declaración del paquete indica en donde se van a incluir las clases `sym` y `parser` que son generadas por el analizador. Todos los `import` que aparezcan en el archivo fuente parecerán luego en el archivo de la clase `parser` permitiendo de esta forma utilizar las clases incluidas en el código de acción.

6.2. Código del programador

Después de la declaración opcional de `import` y `package`, viene una serie de declaraciones opcionales que permiten al usuario escribir código que luego hará parte del analizador generado como parte del archivo `parser`, pero separado en una clase no-pública que contendrá todo el código escrito por el usuario. Este código va incluido entre la siguiente directiva:

```
action code {: ... :};
```

en donde el código que se incluirá en el archivo `parser` será el que esta incluido entre las etiquetas `{: :}`.

Luego de la declaración del código de acción se puede hacer la declaración opcional del código del analizador el cual se agregará directamente a la clase `parser`, este código es útil cuando se va a personalizar algunos de los métodos del analizador:

```
parser code {: ... :};
```

Otra vez, el código que se copia dentro de la clase `parser` es el que se encuentra entre las etiquetas `{: :}`.

La siguiente declaración opcional es:

```
init with {: ... :};
```

Esta declaración provee el código que se va a ejecutar antes de que el analizador llame al primer token. Esta declaración es usada usualmente para inicializar el escáner con tablas y cierto tipos de datos que luego podrán ser utilizados por las acciones semánticas. En este caso, el código se escribirá en un metodo `void` que se encuentra dentro de la clase `parser`.

La siguiente declaración opcional permite indicar al analizador como debe preguntar por el siguiente token del escáner.

```
scan with {: ... :};
```

Al igual que la declaración `init` el código de este bloque se incluye en un método dentro de la clase `parser`, de cualquier forma este método deberá devolver un objeto de tipo `java_cup.runtime.Symbol`, en consecuencia con esto el código que sea incluido dentro de la declaración `scan with` deberá devolver un objeto de este tipo.

6.3. Símbolos terminales y no terminales

Seguido de las declaraciones de código de usuario viene la primera parte requerida de las especificaciones: la lista de símbolos. Esta declaración es la responsable de listar y asignar un tipo para cada símbolo terminal o no terminal que aparece en la gramática. Como se mostrará, cada símbolo terminal o no terminal esta representado en tiempo real por un objeto de tipo `Symbol`. En el caso de los terminales, estos son retornados por el escáner y colocados en la pila del analizador. En el caso de los no terminales reemplazan una serie de objetos `Symbol` en la pila del analizador siempre y cuando este concuerde con la parte derecha de alguna producción. Para efectos de especificar al analizador que tipo de objeto es cada símbolo terminal o no terminal, se hace de la siguiente forma:

```
terminal Nombre_de_la_clase simbolo1, simbolo2, ... ;
terminal simbolo1, simbolo2, ... ;
non terminal Nombre_de_la_clase simbolo1, simbolo2, ... ;
non terminal simbolo1, simbolo2, ... ;
```

donde Nombre_de_la_clase es la clase a la que pertenece el objeto simbolo$x$.

6.4. Definición de la precedencia de operadores

La siguiente sección que es opcional, es donde se especifica la precedencia y asociatividad de los terminales. Esta herramientas es muy útil a la hora de analizar gramáticas ambiguas. Como se muestra en el siguiente ejemplo existen tres tipos de declaraciones:

```
precedence left     terminal, terminal, ...;
precedence right    terminal, terminal, ...;
precedence nonassoc terminal, terminal, ...;
```

La coma separa los terminales que deben tener asociatividad y en ese nivel de precedencia que se esta declarando. El orden de la precedencia del mayor a menor es de abajo a arriba. En el siguiente ejemplo el producto y la división son asociativos y tiene mayor precedencia que la suma y la resta que son asociativos entre ellos.

```
precedence left  SUMA, RESTA;
```

```
precedence left PRODUCTO, DIVISION;
```

La precedencia resuelve problemas de reducción. Por ejemplo, en la entrada $3 + 5 * 3$, el analizador no sabe si reducir la pila, si por el $+$ o por el $*$, de cualquier forma, utilizando precedencia se tiene que el $*$ tiene mayor precedencia que el $+$ por lo tanto reducirá primero por el $*$.

6.5. Definición del símbolo inicial de la gramática y reglas de producción

Para definir el símbolo inicial de la gramática se utiliza la construcción `start with...;`

```
start with Prog;
```

7. Ejecutando en generador de analizadores sintácticos

Como se ha mencionado, CUP esta escrito en Java. Para invocarlo se necesita un interprete Java para invocar el método estático java_cup.Main(), pasando un arreglo de *string* que contiene las opciones. La forma mas fácil de invocarlo es directamente desde la línea de comando de la siguiente forma:

```
java -jar java-cup-11a.jar opciones erchivo_de_entrada
```

Si todo ha salido bien se deberán generar dos archivos .java, el *sym.java* y el *parser.java*. La siguiente es la lista de opciones que se pueden pasar al archivo que genera el código del analizador:

- **package** *name*: Se le especifica al analizador que las clases `sym` y `parser` serán agregadas al paquete *name*. Por defecto estas clases no serán agregadas a ningún paquete.
- **parser** *name*: Hace que el archivo del analizador se llame *name* en vez de `parser`

- **symbols** *namr*: Hace que el archivo de símbolos se llame *name* en vez de `sym`
- **expect** *number*: Por lo general el analizador resuelve problemas de precedencia, esta opción coloca un limite a ese número de errores que se pueden corregir, una vez exceda este limite el analizador se detendra por si solo.
- **nowarn:** Evita que analizador arroje mensajes de prevención o alertas (En ingles: warnings)
- **nosummary:** Normalmente, el sistema imprime una lista con cierto tipo de cosas como los terminales, no terminales, estados del analizador, etc. al final de cada ejecución. Esta opción elimina esta funcionalidad.
- **progress:** Esta opción causa que el sistema imprima pequeños mensajes indicando varias partes del proceso de la creación del analizador.
- **dump:** Esta opción causa que el sistema imprima pedazos de la gramática, estados del analizador y tablas de analisis con el fin de resolver conflictos en el analisis.
- **time:** Causa que el sistema muestre detalles de las estadisticas sobre los tiempos resultantes del analizador. Muy útil para futuros mantenimientos y optimización del analizador.
- **version:** Causa que CUP imprima la versión actual con la que se esta trabajando.

8. Uso de las clases Java

Para utilizar las clases Java obtenidas con CUP para realizar un análisis sintáctico, se seguirá el proceso descrito a continuación.

Como sabemos, el analizador sintáctico consume los tokens generados por el analizador léxico. En CUP, al crear el analizador sintáctico (que será un objeto de la clase parser creada por CUP al ejecutar java_cup.Main), se le pasa al constructor como argumento un objeto que es el analizador léxico:

```
lexer l= new ...; parser par; par= new parser(l);
```

El analizador léxico (en el código de ejemplo anterior, el objeto de la clase lexer) sólo debe de cumplir el requisito de ser de una clase Java que implemente el siguiente interfaz:

```
public interface Scanner {
    public Symbol next_token() throws java.lang.Exception;
}
```

`next_token()` devuelve un objeto de la clase Symbol que representa el siguiente Token de la cadena de Tokens que será la entrada para realizar el análisis sintáctico. El primer Token que se le pasa al analizador sintáctico al invocar next_token será el de más a la izquierda. Un analizador léxico obtenido con JLex se ajusta a estos requisitos.

Para realizar el análisis sintáctico se invoca el método `parse()`, que devuelve un objeto de la clase Symbol que representa al símbolo no terminal raíz del árbol de derivación que genera la cadena de Tokens de entrada. Si queremos obtener el objeto Java asociado a dicho símbolo no terminal, deberemos acceder al atributo value del objeto de la clase Symbol obtenido:

```
p= par.parse().value;
```

Como el objeto almacenado en el atributo value es de la clase Object, normalmente se realizará un casting para convertirlo a la clase adecuada, como por ejemplo:

```
p= (Prog) par.parse().value;
```