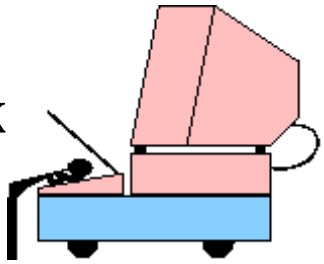
Breve Introducción a JLex



Definición de una gramática en el formato de entrada de JLex

Introducción

NOTA: este documento contiene una breve descripción de algunos de los elementos soportados por JLex. Existen muchas opciones o funcionalidades de JLex que no están descritas aquí. Por lo tanto, se recomienda consultar la documentación de JLex, disponible en la página Web "oficial" de JLex.

Como ejemplo para ilustrar como es un fichero de entrada para JLex (especificación JLex de un analizador léxico), utilizaremos un fichero desarrollado para el lenguaje de programación presentado en clase:

```
package Lexer;

import Parser.sym;
import Traductor.Tipos;

%%
%{
private java_cup.runtime.Symbol tok(int k, Object value) {
//  System.out.println("Token: " + k);
  return new java_cup.runtime.Symbol(k, yyline, 0, value);
}
%}

%public
%cup
%line
%eofval{
{return tok(sym.EOF, null); }
%eofval}

letra= [a-zA-Z]


%%

";"             {return tok(sym.PUNTOCOMA, null); }
":="            {return tok(sym.ASOP, null); }
"+"             {return tok(sym.MAS, null); }
"="             {return tok(sym.IGUALQUE, null); }
"{"             {return tok(sym.ABRELLAVE, null); }
"}"             {return tok(sym.CIERRALLAVE, null); }
","             {return tok(sym.COMA, null); }
"("             {return tok(sym.PAREN, null); }
")"             {return tok(sym.TESIS, null); }
if              {return tok(sym.IF, null); }
```

```
then            {return tok(sym.THEN, null); }
endif           {return tok(sym.ENDIF, null); }
prog            {return tok(sym.PROG, null); }
in              {return tok(sym.IN, null); }
out             {return tok(sym.OUT, null); }
local           {return tok(sym.LOCAL, null); }
true            {return tok(sym.CLOG, new Boolean(true)); }
false           {return tok(sym.CLOG, new Boolean(false)); }
int             {return tok(sym.TIPO, new Boolean(Tipos.tint)); }
bool            {return tok(sym.TIPO, new Boolean(Tipos.tbool)); }
[0-9]+          {return tok(sym.CENT, new Integer(yytext())); }
{letra}({letra}|[0-9])* {return tok(sym.IDENT, yytext()); }
(" "|\n|\t|\r)+ { }
.               { System.out.println("Caracter Ilegal en linea" + yyline);}
```

Un fichero de entrada para JLex consta de 3 secciones:

1. Sección de código de usuario.
2. Directivas JLex.
3. Definición de las expresiones regulares que identifican a cada Token.

Estas secciones están separadas unas de otras por una línea que empieza con los caracteres `%%`, sin que en esa línea pueda haber ningún símbolo adicional.

A continuación se describen cada una de estas partes.

Sección de código de usuario.

El texto anterior a la primera línea que empiece por `%%` se supone que es código Java y se copia exactamente al fichero con el código Java producido por JLex. Aquí se pueden incluir, por ejemplo, construcciones `package` o `import` de Java.

Directivas JLex.

De todas las directivas soportadas por JLex presentamos las siguientes:

- Las directivas `%{` y `%}` permiten encapsular código Java que se va a incluir en la clase Java producida por JLex. La forma correcta de utilizar estas directivas es:

  ```
  %{
  <código_Java>
  %}
  ```

  Es decir, las líneas que contengan las directivas `%{` y `%}` no deberán contener ningún símbolo adicional, y el código Java deberá de encontrarse entre estas líneas. No se debe utilizar identificadores que comiencen por `yy`, para evitar colisiones con los identificadores del código Java producido por JLex.
- `%public`: hace que la clase generada por JLex tenga el modificador `public`
- `%cup`: hace que el analizador léxico que se genere pueda ser utilizado por un analizador sintáctico producido por CUP, es decir, que la clase generada implemente el interfaz `java_cup.runtime.Scanner`.
- `%line`: si se incluye esta directiva, la variable `yyline`, de tipo `int`, indica la línea en la que empieza el Token que está siendo reconocido. La primera línea del programa que se le pasa al analizador léxico es la línea 0.
- Las directivas `%eofval{` y `%eofval}` permiten especificar el Token que devuelve el analizador léxico cuando se llega al final del fichero. Estas directivas encapsulan el código Java que se va a ejecutar cuando se invoque el método que proporciona el analizador léxico para producir el siguiente Token (es decir, si se utiliza la directiva `%cup`, cuando se invoca el método `next_token`) y se ha llegado al final del fichero de entrada al analizador léxico. Por lo tanto, el código Java encapsulado por estas

directivas debe terminar con una sentencia Java `return` que haga que se devuelva el objeto Java correspondiente al Token utilizado cuando se llega al final del fichero (si se utiliza la directiva `%cup`, este objeto Java habrá de ser de la clase `Symbol`). La forma correcta de utilizar estas directivas es:

```
%eofval{
<código_Java>
%eofval}
```

Es decir, las líneas que contengan las directivas `%eofval{` y `%eofval}` no deberán contener ningún símbolo adicional, y el código Java deberá de encontrarse entre estas líneas. Nótese que esta directiva es obligatoria si se utiliza JLex para producir un analizador léxico que va a ser utilizado por un analizador sintáctico generado por CUP, ya que CUP requiere que se le notifique por medio de un Token cuando se llega al final del fichero de entrada, y la acción por defecto de JLex es devolver `null`. Una vez que se ha llegado al carácter fin de línea del fichero de entrada, el analizador léxico continuará ejecutando el código indicado por estas directivas si se piden sucesivas veces un nuevo Token.

En el caso de que se esté usando JLex combinado con CUP, la clase `sym` generada por CUP contendrá un atributo, de nombre `EOF` cuyo valor es el código esperado por el analizador sintáctico generado por CUP para el fin de fichero. Este valor se pasa en un argumento al crear el objeto de clase `Symbol` que se va a devolver, como se puede ver en el ejemplo.

- Si queremos que nuestro analizador léxico lance una excepción cuando se produzca un error léxico, es necesario declarar que excepciones podemos lanzar. Para ello se utiliza las directiva `%yylexthrow{` y `%yylexthrow}`, de la siguiente forma:

  ```
  %yylexthrow{
  , ...,
  %yylexthrow}
  ```

  Es decir, en una línea entre las dos directivas se deben listar la o las excepciones que pueden ser lanzadas (separadas por comas).
- La última construcción que vamos a ver de esta sección es la posibilidad de definir macros. Una macro permite dar un nombre a una cierta expresión regular para poder utilizar ese nombre en lugar de la expresión regular en la definición de las expresiones regulares asociadas a cada Token. Una macro tiene la forma:

  ```
  <nombre>=<definicion>
  ```

  `<nombre>` es el nombre que le vamos a dar a la expresión regular. Ha de estar formado por letras, números o el símbolo "_", empezando por una letra o el símbolo "_". `<definicion>` es la expresión regular que se define utilizando las reglas que se explican más adelante.

Definición de las expresiones regulares que identifican a cada Token.

La tercera parte del fichero de entrada para JLex consiste en una serie de reglas que permiten obtener los Tokens. Cada regla sigue el siguiente formato:

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

La parte `<states>` es opcional y no se explica en este documento. A continuación se explican las otras 2 partes.

Definición de expresiones regulares

`<expression>` especifica la expresión regular que define las cadenas de símbolos de entrada que pueden activar una regla. Cada vez que se solicita un nuevo Token al analizador léxico generado por JLex, éste

consume un cierto número caracteres del fichero de texto que contiene el programa que se le pasa al analizador léxico. Dichos caracteres deben de formar una cadena que pertenezca al lenguaje definido por la expresión regular de alguna de las reglas que permiten obtener los Tokens.

El analizador generado por JLex funciona de forma que se intenta que cada Token producido por el analizador léxico consuma la cadena de caracteres lo más larga que sea posible. Puede ocurrir que dicha cadena de longitud máxima pertenezca a varios de los lenguajes definidos por las expresiones regulares de reglas distintas. En ese caso, se activa la regla que está primero en la especificación JLex. Por esta razón se suele colocar la regla para reconocer identificadores después de las reglas para reconocer las palabras clave del lenguaje de programación, para dar prioridad a éstas.

A continuación se explican algunos de los mecanismos que ofrece JLex para definir expresiones regulares.

Las expresiones regulares no deben contener espacios en blanco, ya que un espacio en blanco se interpreta como el fin de la expresión regular. Si se desea utilizar el espacio en blanco en una expresión regular, se deberá escribir entre comillas:`" "`.

Los siguientes caracteres son metacaracteres, ya que tienen significados especiales en la definición de una expresión regular:

`? * + | ( ) ^ $ . [ ] { } " \`

Los demás caracteres se interpretan en una expresión regular como ellos mismos como símbolos del alfabeto del lenguaje que define la expresión regular.

Dadas dos expresiones regulares *a* y *b* codificadas en el formato de JLex, la expresión regular *ab* representa la concatenación de *a* y *b*.

Dadas dos expresiones regulares *a* y *b* codificadas en el formato de JLex, la expresión regular *a*|*b* representa la unión de *a* y *b*.

Las siguientes secuencias de escape, entre otras, son reconocidas por JLex:

- `\n`: salto de línea.
- `\t`: tabulador.
- `\r`: retorno de carro.

El punto (`.`) es una expresión regular que define el lenguaje de todas las cadenas que no contienen el carácter salto de línea.

Los metacaracteres rodeados de comillas (`"`) pierden su significado especial y se interpretan como ellos mismos como símbolos del alfabeto del lenguaje que define la expresión regular. La única excepción es `\"` que representa al carácter `"`. Por ejemplo la expresión regular `"*\"*"` representa en JLex a la cadena de caracteres `*"*`.

También se puede hacer que un metacarácter pierda su significado especial haciéndole preceder del carácter `\`. Es decir, `\\` y `\"` son equivalentes a `"\"` y `"\""`.

Para utilizar una macro para representar una expresión regular se escribe el nombre de la macro rodeado de los caracteres `{` y `}`.

`*` representa el cierre reflexivo-transitivo. + representa el cierre transitivo.

`(...)` Los paréntesis se utilizan para agrupar expresiones regulares.

`[...]` Los corchetes se utilizan para representar un conjunto de caracteres. Existen varias formas de definir

conjuntos de caracteres dentro de unos corchetes. Aquí sólo explicamos una de ellas. *a-b* indica que el rango de caracteres del *a* al *b*, ambos inclusive, están incluidos en el conjunto de caracteres. Pueden especificarse varios rangos de caracteres dentro de una expresión `[...]`. Por ejemplo, la expresión regular `[a-zA-Z]` es el conjunto de las letras mayúsculas y minúsculas. Finalmente, si el primer carácter después de `[` es el carácter `^` se interpreta como el negado, es decir, se reconocerá a cualquier carácter que no pertenezca al conjunto de caracteres indicado. Por ejemplo, `[^a-z]` representa el conjunto de todos los caracteres que no son letras minúsculas.

Definición de las acciones tomadas cuando se activa una regla

`<action>` es el código Java que se ejecuta cuando se activa la regla a la que pertenece. Normalmente, una acción está formada por una secuencia de sentencias de código Java, terminadas por una sentencia `return` que devuelve el objeto Java que representa el Token que se ha reconocido (si se utiliza la directiva `%cup` deberá ser un objeto de la clase `Symbol`). Si al ejecutar una acción de una regla no se ejecuta ninguna sentencia `return`, el analizador léxico volverá a empezar, consumiendo una nueva cadena de caracteres que pertenezca al lenguaje de la expresión regular de alguna de las reglas de la especificación JLex, para ejecutar una nueva acción.

Dentro del código Java de una acción se pueden utilizar la variable `yyline` de tipo `int` (explicada más arriba) y el método `yytext()`, que devuelve un dato de tipo `String`, que contiene la cadena de carácteres que se ha consumido para activar la regla. Esto nos permite, por ejemplo, recuperar de `yytext` la cadena de caracteres que forma el nombre de un Token identificador.

Por ejemplo, si con el analizador léxico generado por el fichero JLex de ejemplo presentado al principio de este documento se intenta analizar la cadena "`var1 + 12`", la cadena de tokens que se generaría y el valor que devolvería `yytext()` para cada uno se presenta en la siguiente tabla.

Token	Valor devuelto por `yytext()`
IDENT	var1
MAS	+
CENT	12

Cuando se usa JLex combinado con CUP, como ya se ha dicho antes, el analizador léxico devuelve un objeto de la clase `Symbol` por cada token reconocido (ver manual de CUP). Para que una acción devuelva el objeto de la clase `Symbol` adecuado puede ser útil utilizar el método `tok` que se incluye en el ejemplo, pasándole como primer argumento un entero que identifica al token reconocido y como segundo argumento el objeto Java asociado al token o `null` si no procede asociar un objeto Java a dicho token. El método `tok` se encarga de incluir en el objeto de la clase `Symbol` la línea del fichero que está siendo analizado donde se encuentra el token que se ha reconocido, lo que puede ser utilizado posteriormente por el analizador sintáctico generado por CUP para indicar la línea donde se ha producido un error de sintaxis.

Integración de CUP y JLex

Para producir con JLex un analizador léxico que se vaya a integrar con un analizador sintáctico producido con CUP, hay que seguir una serie de reglas en la especificación JLex del analizador léxico. Ya se han indicado algunas de ellas: la directiva `%cup`, así como la necesidad de utilizar las directivas `%eofval{` y `%eofval}` para definir el Token que se genera cuando se llega al final del fichero de entrada.

Se debe de seguir adicionalmente un procedimiento para identificar el Token que devuelve el analizador léxico cuando se ejecuta una acción determinada. Cuando se crea un analizador sintáctico con CUP, este crea varias clases como resultado. Una de las clases producidas por CUP es la clase `sym`. Esta clase, simplemente se limita a definir una constante de tipo `int` por cada símbolo terminal incluido en la especificación CUP correspondiente. Así, en la especificación JLex del ejemplo que está al principio de este documento, a la expresión regular `if` se le asocia el Token `IF` (definido en la especificación CUP que se presenta en el

documento de introducción a CUP). En la clase `sym` producida por CUP se define la constante `IF`, de tipo `int` asociada a dicho Token. Para indicarle al analizador sintáctico producido por CUP que el siguiente Token es `IF`, el atributo `sym` del objeto de la clase `Symbol` que le pasa el analizador léxico al analizador sintáctico debe de tomar el valor `sym.IF`.

Del mismo modo, para indicarle al analizador sintáctico que se ha llegado al final del fichero de entrada, el atributo `sym` del objeto de la clase `Symbol` que le pasa el analizador léxico al analizador sintáctico debe de tomar el valor `sym.EOF`.

Creación de la clase Java

Una vez que hemos creado el fichero JLex, para obtener la clase Java que implementa el analizador léxico correspondiente hay que ejecutar:

```
java JLex.Main fichero_JLex
```

Donde `fichero_JLex` es el fichero en formato JLex que hemos escrito para crear un analizador léxico.

Uso de la clase Java

La clase Java generada por JLex se llama `Yylex`. Esta clase posee dos constructores, ambos con un sólo argumento: o bien de la clase `java.io.InputStream`, o bien de la clase `java.io.Reader`. El stream que se pasa como argumento debe de estar conectado al fichero de texto que contiene el programa a analizar.