

Mi primer proyecto utilizando Jlex y Cup

En este tutorial se desarrolla un ejemplo sencillo de un intérprete que recibe como entrada un archivo de texto que contiene varias expresiones aritméticas que son evaluadas, para ello se hace análisis léxico y sintáctico de dicha entrada, los analizadores se generan con Jlex y Cup. Para desarrollar el proyecto se utilizó Ubuntu 14.04 y Netbeans 8.0, este ejemplo bien se podría adaptar a algún otro sistema operativo basado en GNU/Linux o a windows, aunque en windows el proceso de instalación de Jlex y Cup es un poco distinto, además, tendría que cambiarse el archivo `compilar.sh`, que se reemplazaría con un archivo `.bat`, con instrucciones distintas. El proyecto completo del ejemplo puede descargarse del siguiente enlace:

[Mi primer proyecto utilizando Jlex y Cup](#)

Jlex

Jlex es un generador de analizadores léxicos, escrito en Java, para Java. Jlex fue desarrollado por Elliot Berk en la Universidad de Princeton. Para más información visitar la página oficial de Jlex:

<https://www.cs.princeton.edu/~appel/modern/java/JLex/>

La principal tarea de un analizador léxico es leer los caracteres de entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens.

- Un *token* es un par que consiste en un nombre de token y un valor de atributo opcional.
- Un *lexema* es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de este token.
- Un *patrón* es una descripción de la forma que pueden tomar los lexemas de un token.

En Jlex se definen los patrones de los diferentes tokens que se desean reconocer, estos patrones pueden definirse a través de expresiones regulares. Además Jlex cuenta con múltiples opciones, una muy importante es su capacidad para integrarse con generadores de analizadores sintácticos como Cup.

Cup

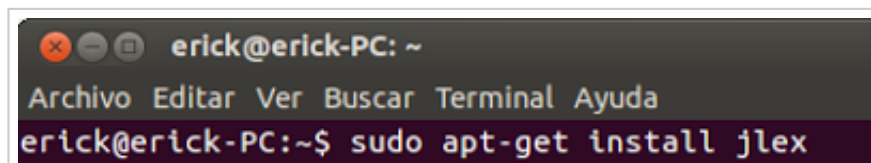
Cup es un generador de analizadores sintácticos de tipo LALR para Java. Para más información visitar la página oficial de Cup:

<http://www2.cs.tum.edu/projects/cup/>

El analizador sintáctico obtiene una cadena de tokens del analizador léxico y verifica que dicha cadena pueda generarse con la gramática para el lenguaje fuente. Una gramática proporciona una especificación precisa y fácil de entender de un lenguaje de programación.

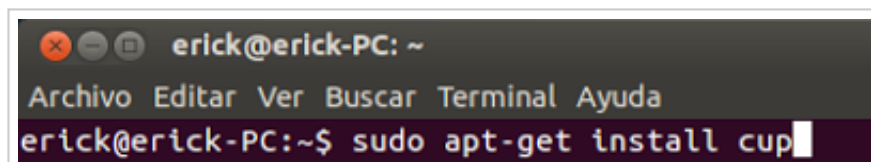
Para desarrollar este ejemplo utilizaremos Netbeans 8.0 en el sistema operativo Ubuntu 14.04.

Lo primero que haremos será instalar Jlex, para ello abrimos una terminal, en Ubuntu puede hacerse con la combinación de teclas `Ctrl+Alt+t` o en Aplicaciones → Accesorios → Terminal, una vez abierta la terminal ingresamos el comando `"sudo apt-get install jlex"`, autenticamos ingresando nuestra contraseña y aceptamos la descarga e instalación, con esto quedará instalado Jlex.



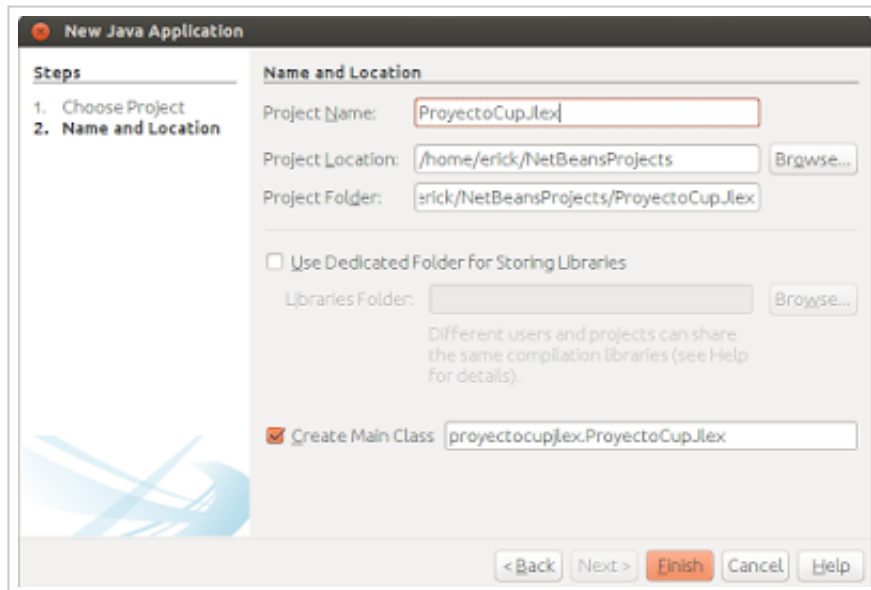
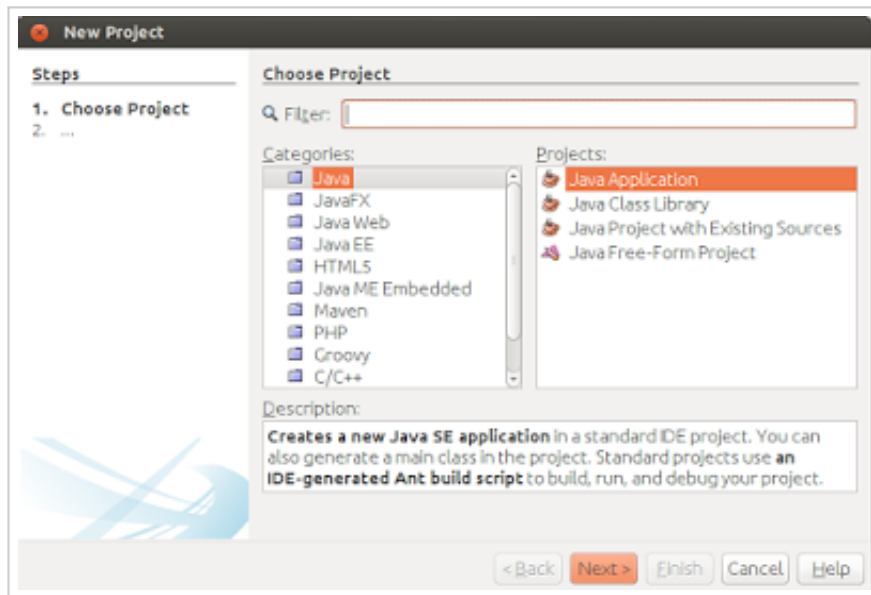
```
erick@erick-PC: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
erick@erick-PC:~$ sudo apt-get install jlex
```

Luego instalamos cup, ejecutando en la terminal el comando `"sudo apt-get install cup"`, autenticamos ingresando nuestra contraseña y aceptamos la descarga e instalación, con esto quedará instalado Cup.

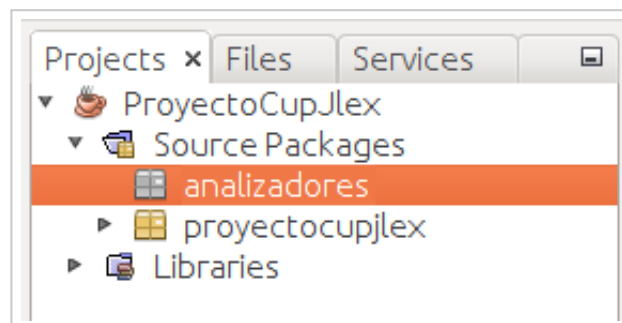
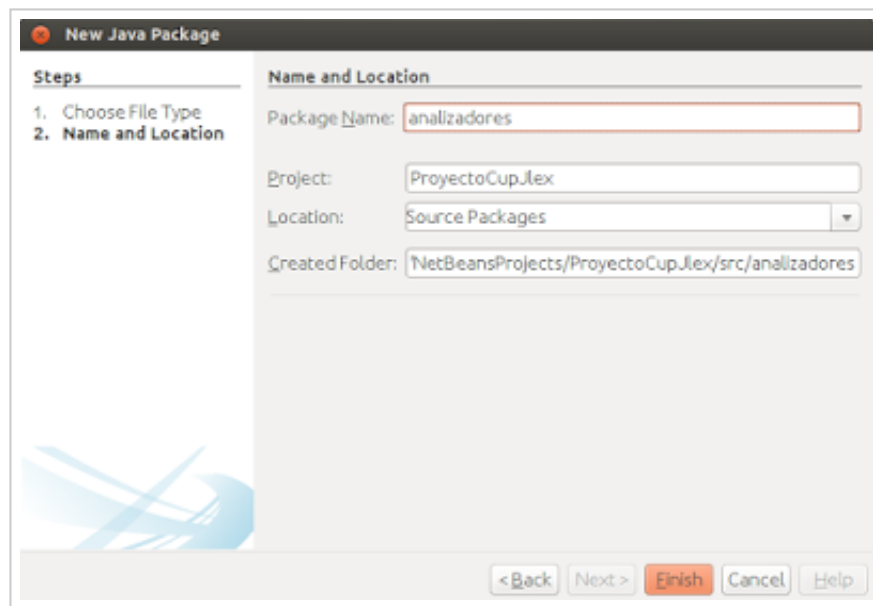


```
erick@erick-PC: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
erick@erick-PC:~$ sudo apt-get install cup
```

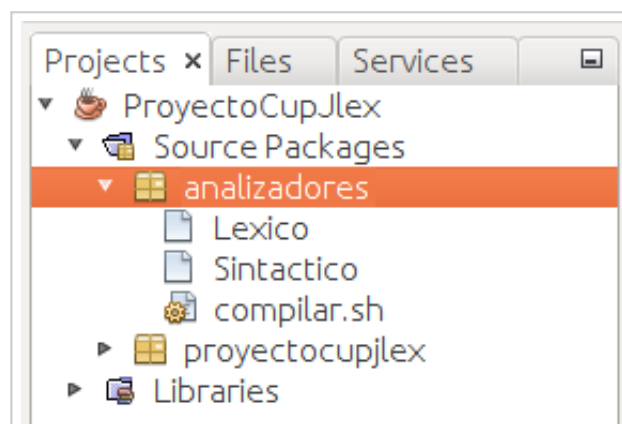
Abrimos Netbeans y creamos un nuevo proyecto de Java (File → New Project).



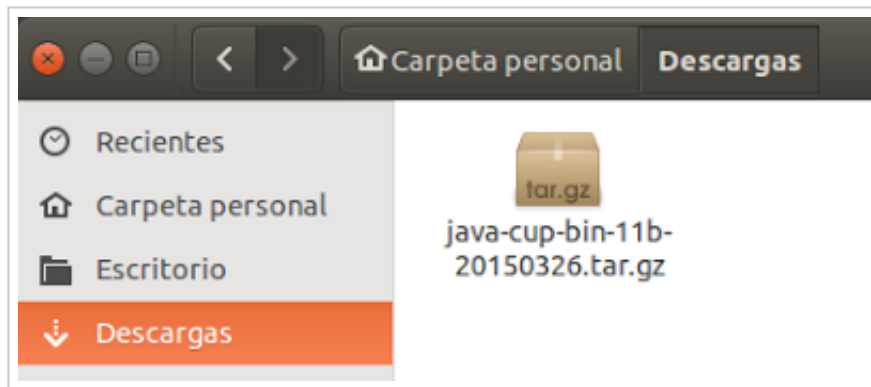
Creamos un paquete llamado analizadores, este almacenará todo el código fuente relacionado con el analizador léxico y sintáctico (Clic derecho en Source Packages → New → Java Package).



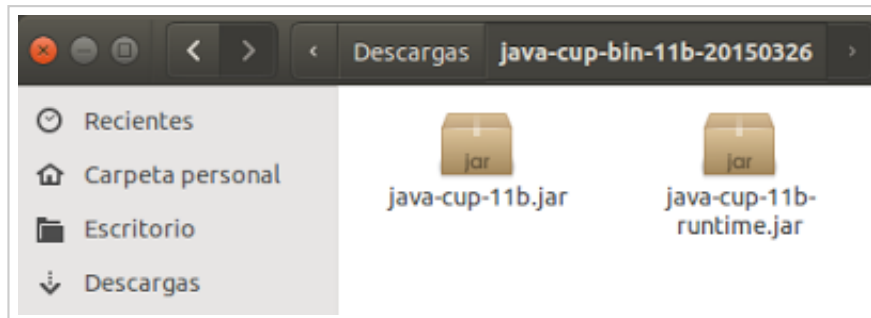
Creamos tres archivos en el paquete analizadores, el primero "Lexico", que almacenará el código fuente con el cual JLex generará el analizador léxico que queremos, el segundo "Sintactico", que almacenará el código fuente con el cual Cup generará el analizador sintáctico que queremos y el tercero "compilar.sh", que guardará los comandos que deben ejecutarse para solicitarle a JLex y a Cup que generen los analizadores. Para crear cada uno de los archivos hacemos clic derecho en el paquete analizadores → New → Other → en la ventana que despliegue, seleccionamos Categories: Other y File Types: Empty File, seleccionamos siguiente, indicamos el nombre para el archivo y finalizamos. De tal modo que al final tendremos los tres archivos.



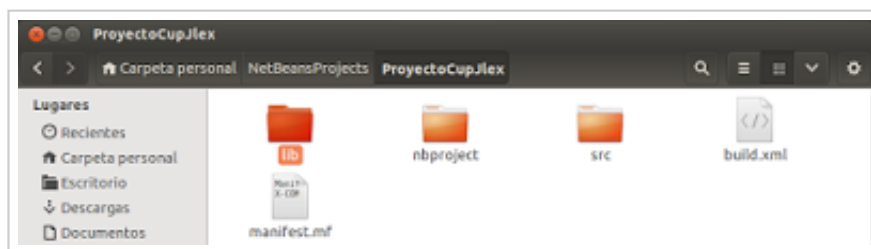
Ahora importaremos la librería de Cup a nuestro proyecto de netbeans, para ello descargamos el archivo "java-cup-bin-11b-<versión>.tar.gz" de la [página oficial de Cup](#). En este caso el archivo descargado fue el "java-cup-bin-11b-20150326.tar.gz".



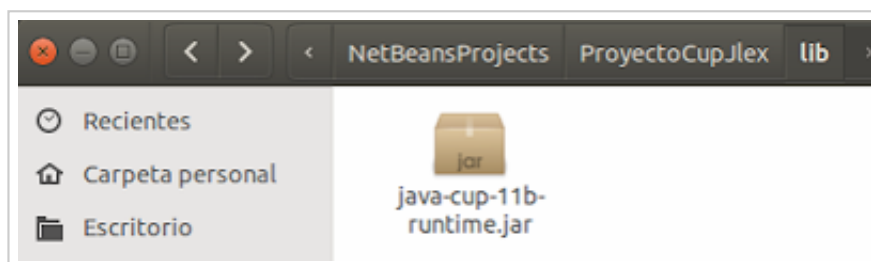
Lo descomprimos y veremos que contiene dos archivos .jar, el que nos interesa es el archivo "java-cup-11b-runtime.jar".



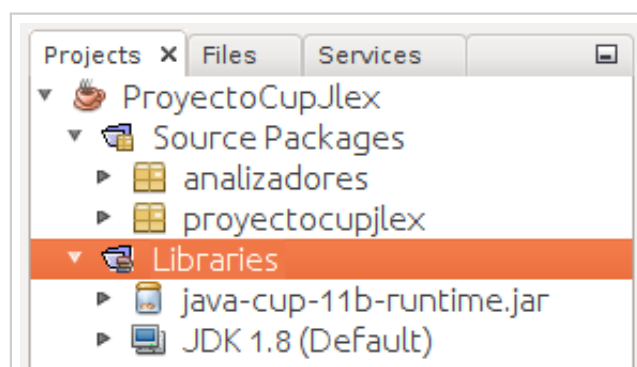
Creamos una carpeta lib dentro de la carpeta de nuestro proyecto.



Dentro de la carpeta lib pegamos el archivo "java-cup-11b-runtime.jar".



Para importar el archivo jar, vamos a Netbeans y damos clic derecho en la pestaña Libraries de nuestro proyecto → Add JAR/Folder... luego buscamos el archivo jar en la carpeta lib que acabamos de crear y lo seleccionamos.



Código fuente para el analizador léxico

En el archivo "Lexico" incluiremos todo el código que le indicará a Jlex lo que debe hacer. El código se muestra a continuación:

```
/*
 * Ejemplo desarrollado por Erick Navarro
 * Blog: e-navarro.blogspot.com
 * Septiembre - 2015
 */

package analizadores;
import java_cup.runtime.Symbol;

%%
%class Lexico
%public
%line
%char
%cup
%unicode
%ignorecase

%init{
    yyline = 1;
```

Explicación del código fuente para el analizador léxico

En las primeras líneas indicamos a Jlex que la clase estará en el paquete analizadores y que es necesario que se importe la clase Symbol.

```
package analizadores;
import java_cup.runtime.Symbol;
```

Posteriormente indicamos a Jlex que:

- La clase del analizador se llamará "Lexico"
- La clase será pública
- Debe llevar el conteo de las líneas
- Debe llevar el conteo de los caracteres reconocidos
- Debe integrarse con cup
- El set de caracteres que debe utilizar es el unicode
- El analizador no será case sensitive, es decir, no le importa si las letras son mayúsculas o minúsculas

```
%%
%class Lexico
%public
%line
%char
%cup
%unicode
%ignorecase
```

Luego viene el bloque init, dentro del init, se ejecutan las acciones de inicialización, es decir, lo que va dentro del constructor del analizador léxico. En este caso indicamos dentro del init que:

- La variable yyline, que lleva la cuenta del número de línea por el que va el analizador valdrá inicialmente 1.
- La variable yychar, que lleva la cuenta del número de carácter por el que va el analizador valdrá inicialmente 1.

```
%init{
    yyline = 1;
    yychar = 1;
}%init}
```

Luego se escriben algunas expresiones regulares que son almacenadas en macros, que básicamente son variables que almacenan los patrones, en este caso se definen las macros: BLANCOS, D y DD. Los patrones para cada una son los siguientes:

- BLANCOS - Expresión regular que reconoce uno o muchos espacios en blanco, retornos de carro o tabuladores.
- D - Expresión regular que reconoce números enteros.
- DD - Expresión regular que reconoce números con punto decimal.

```
BLANCOS=[ \r\t]+
D=[0-9]+
DD=[0-9]+("."[ 0-9])?
%%
```

Por último se definen todas las reglas léxicas, en las que indicamos el patrón que reconocerá y dentro de llaves lo que debe hacer cuando lo reconozca. En la mayoría de los casos se retorna un objeto de tipo Symbol, que vendría siendo un token, este se instancia con el tipo, la fila en la que se encontró, la columna en la que se encontró y el lexema en específico que se reconoció, este se obtiene mediante yytext(). Dentro de las llaves podríamos incluir el código java que quisiéramos. Vemos que al reconocer el patrón BLANCOS no se hace nada porque esperamos que ignore los espacios en blanco. También vemos que al encontrar un salto de línea reinicia la variable yychar, es decir, reinicia el conteo de caracteres para que se lleve la cuenta del número de columna en cada fila.

```
"Evaluar" {return new Symbol(sym.REVALUAR,yyline,yychar,
                             yytext());}

";" {return new Symbol(sym.PTCOMA,yyline,yychar, yytext());}
"(" {return new Symbol(sym.PARIZQ,yyline,yychar, yytext());}
")" {return new Symbol(sym.PARDER,yyline,yychar, yytext());}
"[" {return new Symbol(sym.CORIZQ,yyline,yychar, yytext());}
"]" {return new Symbol(sym.CORDER,yyline,yychar, yytext());}

"+" {return new Symbol(sym.MAS,yyline,yychar, yytext());}
"-" {return new Symbol(sym.MENOS,yyline,yychar, yytext());}
"*" {return new Symbol(sym.POR,yyline,yychar, yytext());}
"/" {return new Symbol(sym.DIVIDIDO,yyline,yychar, yytext());}

\n {yychar=1;}

{BLANCOS} {}
{D} {return new Symbol(sym.ENTERO,yyline,yychar, yytext());}
{DD} {return new Symbol(sym.DECIMAL,yyline,yychar, yytext());}
```

Código fuente para el analizador sintáctico

En el archivo "Sintáctico" incluiremos todo el código que le indicará a Cup lo que debe hacer. El código se muestra a continuación:

```

/*
 * Ejemplo desarrollado por Erick Navarro
 * Blog: e-navarro.blogspot.com
 * Septiembre - 2015
 */

package analizadores;
import java_cup.runtime.*;

parser code
{
    /**
     * Método al que se llama automáticamente ante algún error sintactico.
     */
    public void syntax_error(Symbol s){
        System.out.println("Error Sintáctico en la Línea " + (s.left) +
            " Columna "+s.right+ ". No se esperaba este componente: " +s.value+".");
    }
}

```

Explicación del código fuente para el analizador sintáctico

En las primeras líneas indicamos a Cup que la clase estará en el paquete analizadores y que es necesario que se importe todo el contenido de "java_cup.runtime".

```

package analizadores;
import java_cup.runtime.*;

```

Luego viene la sección "parser code", en la que se programan acciones propias del parser o analizador sintáctico que se va a generar, en este caso se programa lo que se debe hacer ante un error sintáctico y ante un error sintáctico irrecoverable.

```

parser code
{
    /**
     * Método al que se llama automáticamente ante algún error sintactico.
     */
    public void syntax_error(Symbol s){
        System.out.println("Error Sintáctico en la Línea " + (s.left) +
            " Columna "+s.right+ ". No se esperaba este componente: " +s.value+".");
    }

    /**
     * Método al que se llama automáticamente ante algún error sintáctico
     * en el que ya no es posible una recuperación de errores.
     */
    public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception{
        System.out.println("Error sintactico irrecoverable en la Línea " +
            (s.left)+ " Columna "+s.right+ ". Componente " + s.value +
            " no reconocido.");
    }
}
:}

```

Luego se definen los terminales, a estos se les puede indicar un tipo, en este caso todos son de tipo *String*, si no se indicara un tipo, los terminales serían por defecto de tipo *Object*.

```

terminal String PTCOMA, PARIZQ, PARDER, CORIZQ, CORDER;
terminal String MAS, MENOS, POR, DIVIDIDO;
terminal String ENTERO;
terminal String DECIMAL;
terminal String UMENOS;
terminal String REVALUAR;

```

Existe un terminal por cada tipo de token que el analizador léxico devuelve. Todos estos tipos estarán definidos en la clase "sym", que se genera automáticamente y de la que se hablará más adelante.

Luego viene la declaración de los no terminales, a los que también se les puede indicar un tipo específico, si no se les indica un tipo, estos son por defecto de tipo *Object*.

```
non terminal ini;
non terminal instrucciones;
non terminal instruccion;
non terminal Double expresion;
```

Posteriormente podemos indicar la precedencia de los operadores, ya que la gramática escrita es ambigua, es necesario definir una precedencia para que el analizador no entre en conflicto al analizar, en este caso la precedencia es la misma que la de los operadores aritméticos, la precedencia más baja la tienen la suma y la resta, luego están la multiplicación y la división que tienen una precedencia más alta y por último está el signo menos de las expresiones negativas que tendría la precedencia más alta.

```
precedence left MAS,MENOS;
precedence left POR,DIVIDIDO;
precedence right UMENOS;
```

Por último viene el conjunto de reglas de escritura de la gramática o producciones, al final de cada producción puede incluirse código java entre llaves y dos puntos "{:<código java>:}". Podemos ver que en las producciones del no terminal "expresion", se utiliza la variable RESULT, esta variable es propia de Cup y nos permite sintetizar cierto atributo para ese no terminal que se encuentra del lado izquierdo de la producción, recordemos que Cup trabaja con analizadores LALR, que son de tipo ascendente, lo que significa que nos permiten manipular atributos sintetizados. Básicamente eso es RESULT, un atributo sintetizado.

RESULT puede ser cualquier objeto, por ejemplo si quisiéramos que RESULT almacenara varios números enteros hacemos una clase Nodo que contenga muchas variables de tipo entero y declaramos los no terminales para que sean de tipo Nodo, entonces el RESULT que sintetizarán dichos no terminales serán de tipo Nodo.

```
start with ini;

ini::=instrucciones;

instrucciones ::=
    instruccion instrucciones
  | instruccion
  | error instrucciones
;

instruccion ::=
    REVALUAR CORIZQ expresion:a CORDER PTCOMA{:System.out.println("El valor de la expresión
;

expresion ::=
    MENOS expresion:a                {:RESULT=a*-1;:}%prec UMENOS
  | expresion:a MAS                  expresion:b      {:RESULT=a+b;:}
  | expresion:a MENOS                expresion:b      {:RESULT=a-b;:}
  | expresion:a POR                   expresion:b      {:RESULT=a*b;:}
```

El archivo de compilación

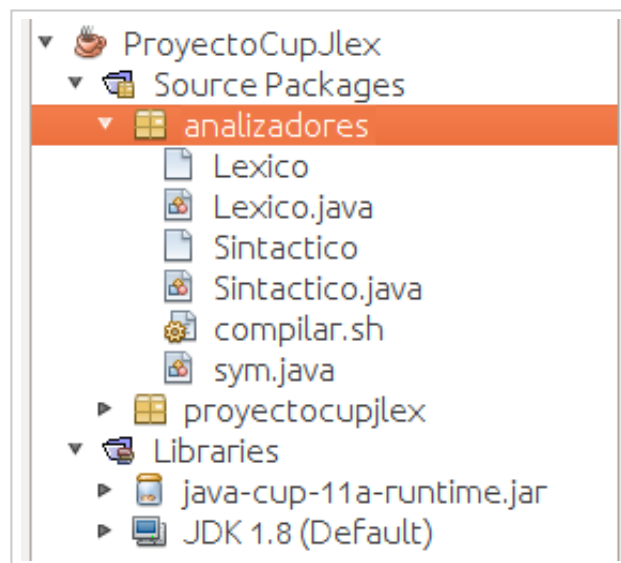
En el archivo "compilar.sh", ejecutamos dos líneas, la primera indica a Jlex que debe generar un analizador léxico en base al código fuente que se encuentra en el archivo "Lexico", la segunda indica a Cup que la clase que debe generar para el analizador sintáctico se llamará "Sintactico" y que debe generarse en base al código fuente que se encuentra en el archivo "Sintactico".

```
jlex Lexico
cup -parser Sintactico Sintactico
```


Para ejecutarlo solo vamos a Netbeans, damos clic derecho sobre el archivo y seleccionamos la opción Run. Al finalizar la ejecución del archivo, veremos en la consola de Netbeans una salida como la siguiente:

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 67 states.
Working on character classes.....
NFA has 24 distinct character classes.
Creating DFA transition table.
Working on DFA states.....
Minimizing DFA transition table.
24 states after removal of redundant states.
Outputting lexical analyzer code.
----- CUP v0.11a beta 20060608 Parser Generation Summary -----
  0 errors and 0 warnings
 15 terminals, 4 non-terminals, and 14 productions declared,
producing 28 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
```

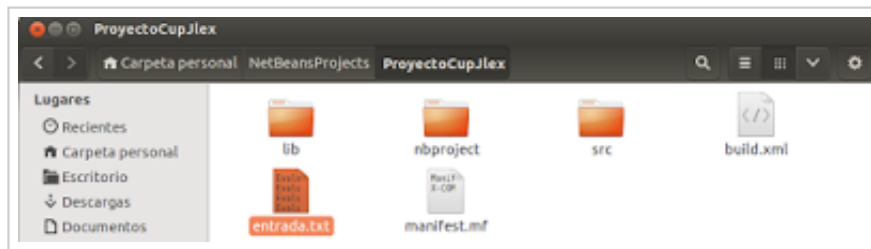
Que nos confirma que la generación del analizador léxico fue exitosa y que la del analizador sintáctico también. Veremos que se han creado tres nuevos archivos en el paquete analizadores. Estos archivos son las clases: Lexico.java, Sintactico.java y sym.java.



La clase sym.java, sirve como puente entre la clase Lexico.java y Sintactico.java, por ejemplo, cuando el analizador léxico reconoce un número entero, instancia un objeto de la clase Symbol e indica que es de tipo número entero por medio de la constante "sym.ENTERO", que se genera dentro de la clase sym.java y esta constante se genera porque en el archivo de entrada para Cup se indicó que existe un terminal llamado ENTERO. Entonces tanto el analizador léxico como el sintáctico hacen referencia a los tokens de tipo número entero con la constante "sym.ENTERO". Básicamente eso es sym.java, una clase con muchas constantes estáticas a las que acceden ambos analizadores para poder integrarse y ejecutar sus tareas exitosamente.

Creando un archivo de entrada para nuestros analizadores

Dentro de la carpeta del proyecto crearé un archivo de entrada llamado "entrada.txt". Que contendrá el archivo de entrada que reconocerán nuestros analizadores.



El archivo de "entrada.txt" contiene lo siguiente:

```
Evaluar[1+1];  
Evaluar[1+1*2];  
Evaluar[-(1+1*6/3-5+7)];  
Evaluar[-(1+1*6/3-5+1*-2)];  
Evaluar[-(1+1)];
```

Clase principal

Dentro de la clase principal solo tenemos el método *main* y el método interpretar que lee el contenido del archivo que se encuentra en el path que se le indica y ejecuta análisis léxico y análisis sintáctico, en el transcurso del análisis sintáctico se mandan a imprimir en consola los resultados de las expresiones aritméticas analizadas, por lo que al final del análisis tendremos todos los resultados de las operaciones en consola. A continuación se muestra el código de la clase principal.

```
/*  
 * Ejemplo desarrollado por Erick Navarro  
 * Blog: e-navarro.blogspot.com  
 * Septiembre - 2015  
 */  
  
package proyectocupjlex;  
import java.io.FileInputStream;  
  
/**  
 * Clase principal de la aplicación  
 * @author Erick  
 */  
public class ProyectoCupJlex {  
  
    /**  
     * @param args argumentos de la linea de comando  
     */  
    public static void main(String[] args) {
```

Ejecutando nuestra aplicación

Al ejecutar la aplicación obtenemos los resultados de las operaciones evaluadas en consola.

