Generadores de analizadores sintácticos: Cup (I)

Jesús Gallardo Casero
Asignatura *Teoría de la Computación*
E.U. Politécnica de Teruel
Universidad de Zaragoza

Contenidos

- Introducción.
- Generadores automáticos de analizadores sintácticos.
- La herramienta Cup.
- Integración de Cup y JFlex.

Introducción

- El **analizador sintáctico** o ***parser*** es un componente de un procesador de lenguajes que suele ejecutarse después del analizador léxico.
- Su función principal es revisar si los *tokens* del código fuente que le proporciona el analizador léxico aparecen en el orden correcto.
  - El orden viene impuesto por una gramática.

Introducción

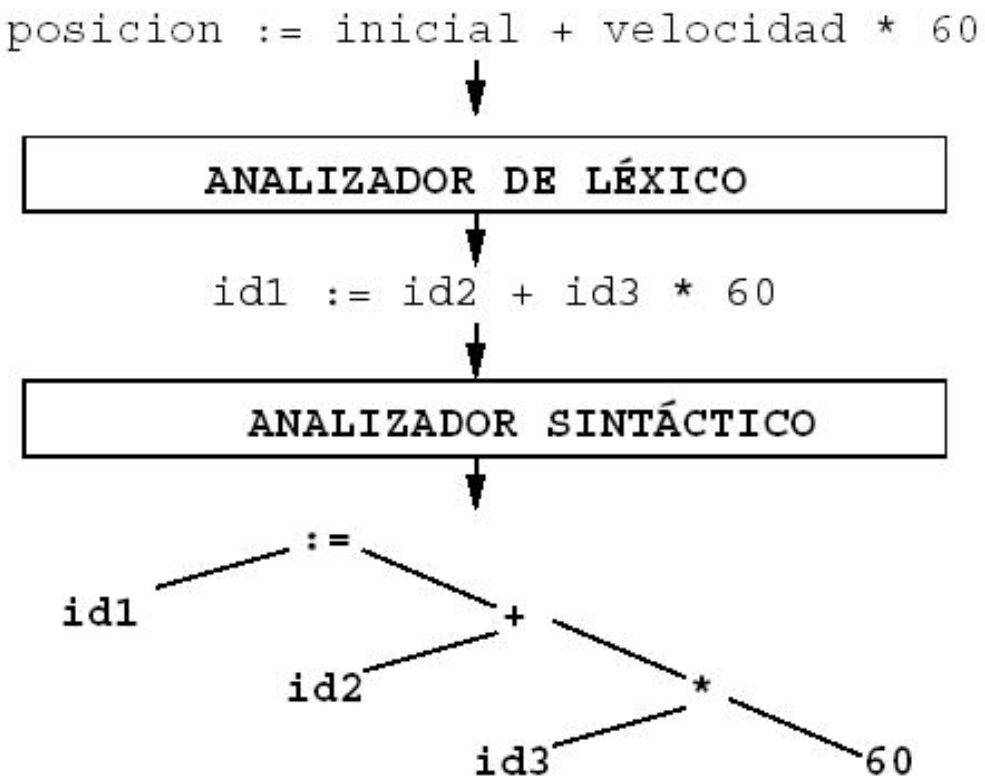
- Si el programa no tiene una estructura sintáctica correcta, el analizador sintáctico no podrá encontrar el árbol de derivación correspondiente y deberá dar un mensaje de error sintáctico.
- Las sintaxis de los lenguajes se describen mediante gramáticas libres de contexto (tipo 2).

Introducción

- Analizar sintácticamente una cadena de *tokens* es encontrar para ella un árbol sintáctico o derivación que tenga como raíz el símbolo inicial de la GLC y mediante la aplicación de sucesiva de sus reglas de derivación se pueda alcanzar dicha cadena como hojas del árbol sintáctico.

Introducción

- Ejemplo:

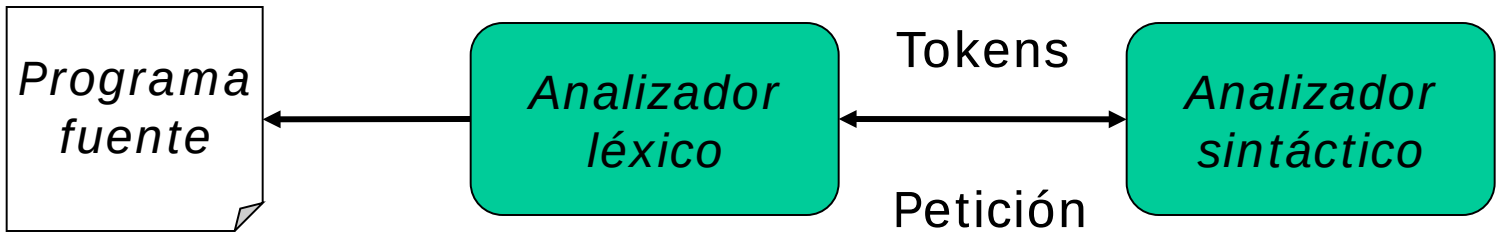


Introducción

- Por lo tanto:
  - El **analizador léxico** valida contra **expresiones regulares.**
  - El **analizador sintáctico** valida contra **gramáticas libres de contexto.**

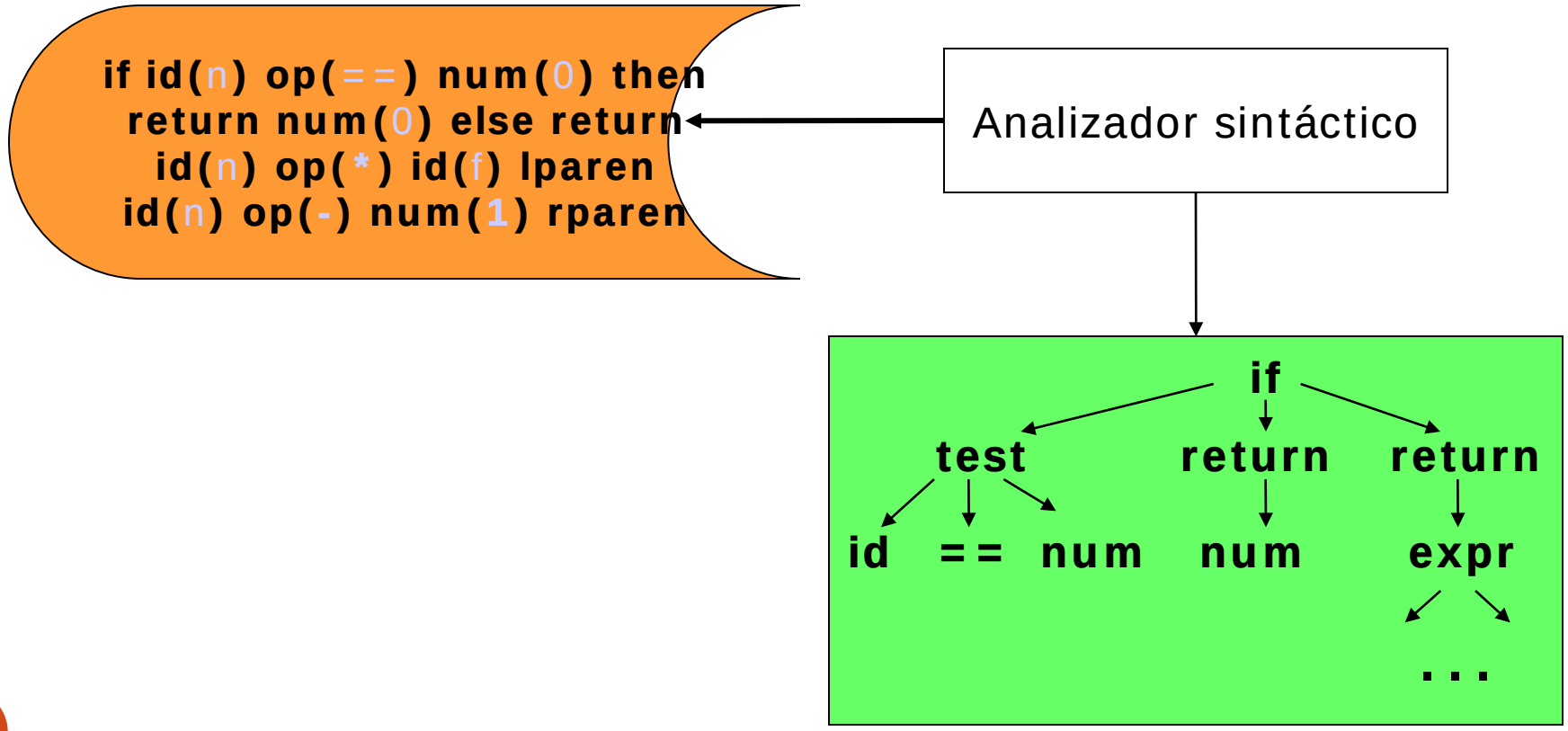
Introducción

- Los analizadores sintácticos se combinan con los analizadores léxicos, recibiendo como entrada la secuencia de *tokens* correspondiente al texto inicial.



Introducción

- Ejemplo de funcionamiento del analizador sintáctico:



Introducción

- Ejemplo de lenguaje de programación definido mediante una GLC:

```
PROGRAM -> program STATEMENT end
STATEMENT -> ASSIGNMENT STATEMENT | LOOP
  STATEMENT | ε
ASSIGNMENT -> (identifier) := EXPRESSION ;
LOOP -> while EXPRESSION do STATEMENT done
EXPRESSION -> VALUE | VALUE + VALUE |
  VALUE <= VALUE
VALUE -> (identifier) | (number)
```

Introducción

- Ejemplo de texto perteneciente al lenguaje anterior:
  - Antes del análisis léxico:

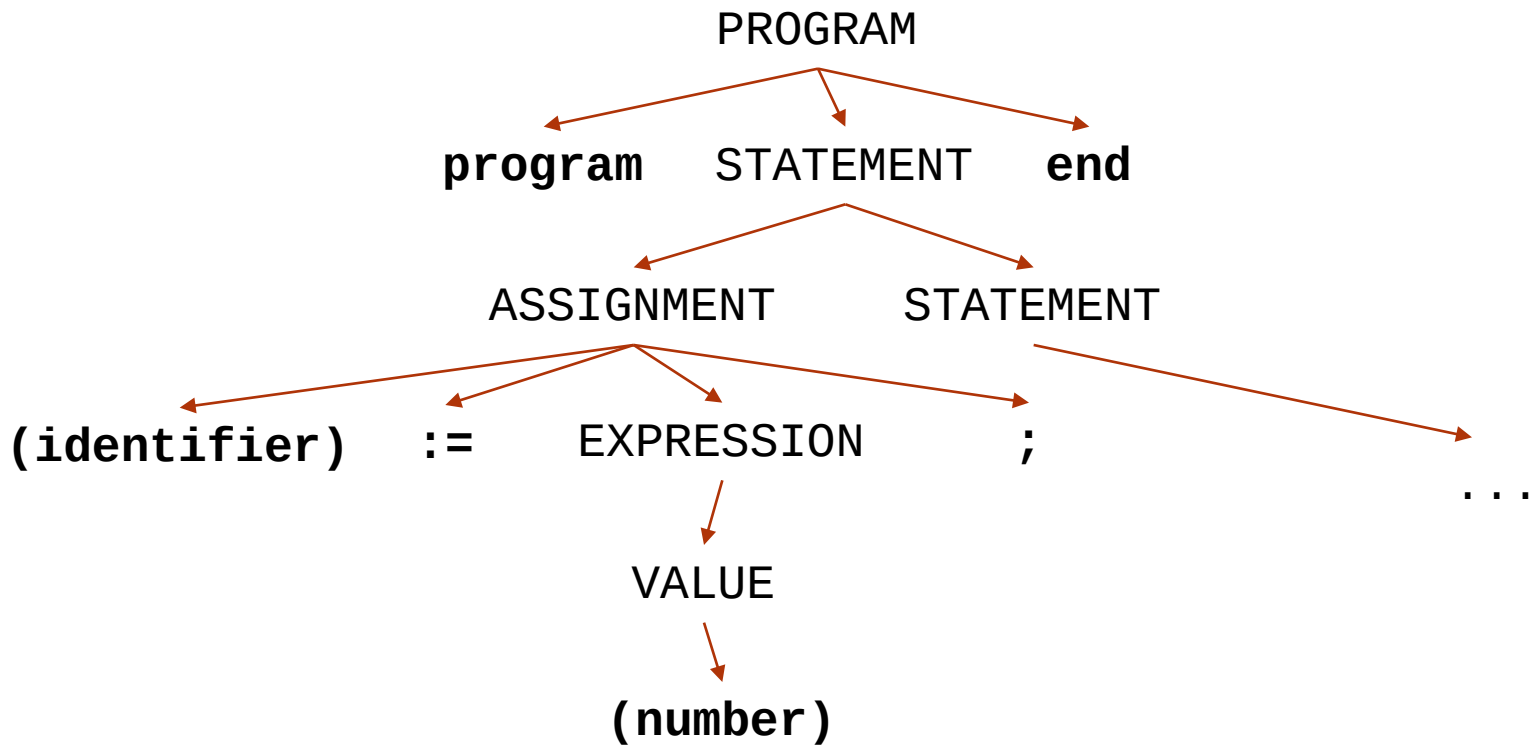
```
program
  miVariable := 0;
  while miVariable <= 10 do
   miVariable := miVariable + 1;
  done
end
```

  - Después del análisis léxico:

```
program
  (identifier) := (number) ;
  while (identifier) <= (number) do
   (identifier) := (identifier) + (number) ;
  done
end
```

Introducción

- Fragmento del árbol sintáctico del ejemplo:



Introducción

- Al definir un lenguaje mediante una gramática libre de contexto, los **terminales** de la GLC se corresponden con los **tokens** del lenguaje.
- Por ejemplo, en el lenguaje de la transparencia anterior, serían *tokens*: **`program, end, while, do, done, := , ; , + , <= , (identifier), (number)`**
- Si se usa un analizador léxico en combinación con el sintáctico, el analizador léxico habrá reconocido anteriormente esos *tokens* y habrá generado la secuencia correspondiente.

Generadores automáticos de analizadores sintácticos

- Al igual que en el caso de los analizadores léxicos, existen herramientas que nos permiten generar analizadores sintácticos de forma automática.
- Estas herramientas reciben una especificación de la GLC de partida e implementan el analizador a partir de ella.
- El analizador se construye implementando alguno de los tipos de análisis sintáctico que existen.

Generadores automáticos de analizadores sintácticos

- Tipos de análisis sintáctico:
  - Descendente:
    - Construye el árbol sintáctico desde la raíz hasta las hojas.
    - Parte del símbolo inicial de la GLC y va aplicando derivaciones hasta llegar a la secuencia de *tokens*.
  - Ascendente:
    - Construye el árbol sintáctico desde las hojas hasta la raíz.
    - Parte de los *tokens* y va aplicando *reducciones* (operación inversa a la derivación) hasta llegar al símbolo inicial de la GLC.
- Normalmente, los generadores automáticos suelen implementar un análisis sintáctico **ascendente**.

La herramienta Cup

- Cup es una herramienta para crear analizadores sintácticos del tipo que hemos visto antes.
- La herramienta Cup está basada en Java:
  - Genera código Java.
  - Sus especificaciones pueden incluir código Java que posteriormente irá en el analizador creado.
- Cup se integra fácilmente con JFlex.

La herramienta Cup

- Para descargarse Cup:
  http://www2.cs.tum.edu/projects/cup/
- Descargarse el archivo .jar
- Para hacerlo funcionar:

```
java -jar java-cup-11b.jar <opciones> <archivo>
```

  - Algunas opciones: *dump*, *debug*.
  - El nombre del archivo .jar podría necesitar de la ruta completa, o de que su carpeta esté en el PATH (variables de entorno).

La herramienta Cup

- Funcionando desde Eclipse:

  1.- Crear un proyecto Java.

  2.- Sobre el proyecto, hacer clic en el botón derecho y elegir "Build Path → Add External Archives…".

  3.- En el cuadro de diálogo, localizar el archivo .jar.

  4.- Crear el archivo .cup en la carpeta *src* del proyecto (para que los fuentes se generen allí).

La herramienta Cup

- Funcionando desde Eclipse (cont.):

  5.- En el menú "Run → Run configurations" crear una configuración nueva con los siguientes parámetros:

  - Project: El nombre del proyecto creado.
  - Main class: `java_cup.Main`.
  - Program arguments: src/*nombre del archivo cup*.
  - Working directory: El directorio src del proyecto.

  6.- Ejecutar la configuración creada. Se generarán las clases con el código fuente.

La herramienta Cup

- Los archivos de entrada de la herramienta Cup tienen cinco partes bien diferenciadas:
  - Especificaciones *package* e *import*.
  - Componentes de código de usuario.
  - Lista de símbolos de la gramática.
  - Declaraciones de precedencia.
  - Especificación de la gramática.
- Sólo la lista de símbolos y la especificación de la gramática son secciones obligatorias.

La herramienta Cup

- Especificaciones *package* e *import*:
  - Sección para declarar el paquete al cual pertenecerá la clase generada y las clases que se necesita importar.
  - Estas sentencias se copian tal cual en la clase generada.
  - Al menos debe incluirse el *import* siguiente:

```
import java_cup.runtime.*;
```

La herramienta Cup

- Componentes de código de usuario:
  - Se puede incluir código Java que el usuario desee incluir en el analizador sintáctico que se va a obtener con CUP.
  - Varios tipos de código de usuario:
    - Declaración de las variables y rutinas utilizadas en el código de usuario embebido.
      - Permite definir variables y procedimientos de asistencia al parser dentro de una clase de ayuda.

        ```
        action code{: //código Java :}
        ```
    - Declaración de métodos y variables que se incluyen dentro del código de la clase del analizador generado.

      ```
      parser code{: //código Java :}
      ```

La herramienta Cup

- Componentes de código de usuario (cont.):
  - Más tipos de código de usuario:
    - Lógica de inicialización que se llamará antes de comenzar el análisis.

```
init with{: //código Java :}
```

    - Sentencias a utilizar para solicitar un nuevo token.

```
scan with{: //código Java :}
```

La herramienta Cup

- Lista de símbolos de la gramática:
  - Símbolos terminales:
    - Se especifican los símbolos terminales de la gramática indicando, opcionalmente, los tipos (clase Java).
    - Ejemplos:

```
terminal PUNTOYCOMA, SIGNO_MAS;
terminal Integer OPERANDO;
```

  - Símbolos no terminales:
    - Se especifican del mismo modo.
    - Ejemplos:

```
non terminal Integer expr, factor, term;
```

La herramienta Cup

- Lista de símbolos de la gramática (cont.):
  - El analizador generado, durante el proceso de análisis sintáctico, guarda internamente una cadena de símbolos que representan el estado en el que se encuentra en cada paso del análisis.
  - Cada uno de los símbolos gramaticales del estado del analizador sintáctico se representa en cup por medio de un objeto de la clase *Symbol*.

La herramienta Cup

- Declaraciones de precedencia:
  - Es útil para el análisis de gramáticas ambiguas y resolver algunos conflictos reducción-desplazamiento.
  - Existen tres tipos de declaraciones de precedenciaasociatividad:

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

  - El orden de precedencia va desde el final al principio, es decir:

```
precedence left ADD, SUBTRACT;
precedence left TIMES, DIVIDE;
```

La herramienta Cup

- Declaraciones de precedencia (cont.):
  - Si un terminal no está en la declaración tendrá el menor orden de precedencia.
  - Las producciones sin terminales tendrán el menor orden de precedencia, y si presentan algún terminal será el correspondiente a este último.
  - La asociatividad es asignada a cada terminal según la declaración realizada (izquierda, derecha y no asociatividad).
    - Por defecto, suele ser de izquierda a derecha.
  - Si un terminal es declarado *nonassoc* servirá para que cuando aparezcan dos ocurrencias del mismo terminal en la misma producción se genere un error.

La herramienta Cup

- Especificación de la gramática:
  - Se introducen las producciones de la gramática:
    - Se usa el separador ::= entre parte izquierda y parte derecha.
    - La barra vertical | separa las partes derechas.
    - Las producciones nulas se indican dejando vacía la parte derecha.
    - Ejemplo:

```
expr_stmt ::= expr PYC | for_stmt PYC | ;
```

Integración de Cup y JFlex

- Habitualmente, Cup se suele utilizar integrado con JFlex.
- La idea es que JFlex reconozca los *tokens* y le pase a Cup la secuencia detectada.
- Cada **terminal** de la GLC especificada en Cup debe corresponder con un **token** reconocido en JFlex.
- Para conseguir esta integración, es necesario hacer algunos cambios en la especificación de entrada a JFlex.
- Posteriormente, desde una clase externa se invocará el análisis léxico y el resultado se pasará al analizador sintáctico.

Integración de Cup y JFlex

- Cambios en la especificación JFlex:
  - En la sección de código de usuario (al principio del archivo), importar las clases necesarias:

```
import java_cup.runtime.Symbol;
import java_cup.runtime.ComplexSymbolFactory;
import
  java_cup.runtime.ComplexSymbolFactory.Location;
```

  - En la sección de opciones y declaraciones, añadir la opción de compatibilidad con Cup:

```
%cup
```

Integración de Cup y JFlex

- Cambios en la especificación JFlex (cont.):
  - Carga de la *fábrica de símbolos*. En el código que se añade a la clase del analizador léxico:

```
public analex(java.io.Reader in, ComplexSymbolFactory sf){
        this(in);
        symbolFactory = sf;
}
ComplexSymbolFactory symbolFactory;
```

Integración de Cup y JFlex

- Cambios en la especificación JFlex (cont.):
  - Crear métodos para crear nuevos objetos *Symbol* a partir de cada token:

```
private Symbol symbol(String name, int sym) {
 return symbolFactory.newSymbol(name, sym, new
  Location(yyline+1,yycolumn+1,yychar), new
  Location(yyline+1,yycolumn+yylength(),yychar+yylength()));
}

private Symbol symbol(String name, int sym, Object val) {
 Location left = new Location(yyline+1,yycolumn+1,yychar);
 Location right= new
  Location(yyline+1,yycolumn+yylength(),yychar+yylength());
 return symbolFactory.newSymbol(name, sym, left, right, val);
}
```

Integración de Cup y JFlex

- Cambios en la especificación JFlex (cont.):
  - Reconocimiento correcto del fin de archivo:

```
%eofval{
 return symbolFactory.newSymbol("EOF", sym.EOF, new
  Location(yyline+1,yycolumn+1,yychar), new
  Location(yyline+1,yycolumn+1,yychar+1));
%eofval}
```

Integración de Cup y JFlex

- Cambios en la especificación JFlex (cont.):
  - Invocación a los métodos que crean los símbolos:

```
[1-9][0-9]*|0        {return symbol("num", sym.NUM,
                            Integer.parseInt(yytext()));}
"+"                  {return symbol("mas",sym.MAS);}
```

Integración de Cup y JFlex

- Código para ejecutar el análisis léxico y el sintáctico:

```
ComplexSymbolFactory csf = new ComplexSymbolFactory();
ScannerBuffer lexer = new ScannerBuffer(new analex(new
  BufferedReader(new FileReader(args[0])),csf));
parser p = new parser(lexer,csf);
p.parse();
```

- Donde *parser* es el nombre de la clase generada por Cup, y *analex* es el nombre de la clase generada por JFlex.

Integración de Cup y JFlex

- Para compilar y ejecutar todo desde línea de comandos:
  - Compilación: La manera más fácil es especificando en la misma orden el CLASSPATH:

```
javac -cp .;java-cup-11b.jar Principal.java
```

  - Ejecución: Se lanza el programa principal pasándole como argumento el archivo a analizar. De nuevo es necesario especificar el CLASSPATH

```
java -cp .;java-cup-11b.jar Principal prueba.txt
```

- Para no tener que poner «`-cp .;java-cup-11b.jar`» cada vez, se puede modificar en la variable de entorno:

```
set CLASSPATH=%CLASSPATH%;.;java-cup-11b.jar
```

Integración de Cup y JFlex

- Para ejecutar en Eclipse:

  1.- Ejecutar JFlex para generar las clases del analizador léxico. Copiar esas clases a la carpeta *src* del proyecto.

  2.- Ejecutar Cup para generar las clases del analizador sintáctico.

  3.- Crear la clase principal que lanzará el análisis en la misma carpeta de fuentes, y que incluye el método *main*.

Integración de Cup y JFlex

- Para ejecutar en Eclipse (cont.):

  4.- Crear el archivo de texto que se analizará y almacenarlo en la carpeta del proyecto.

  5.- Crear una nueva configuración de ejecución con los siguientes parámetros:

  - Project: El nombre del proyecto creado.
  - Main class: La clase que incluye el método *main*.
  - Program arguments: nombre del archivo de texto.

  6.- Ejecutar la configuración creada.