

Procesadores de Lenguaje

→ CUP - Constructor of Useful Parsers



Salvador Sánchez, Daniel Rodríguez
Departamento de Ciencias de la Computación
Universidad de Alcalá

→ CUP

- CUP (Constructor of Useful Parsers)
 - genera analizadores sintácticos LALR.
 - CUP genera código Java que implementa el analizador sintáctico a partir de un fichero con la especificación sintáctica del lenguaje.



→ CUP – Instalación

`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

1. Elegir el directorio. Ej.: `c:\java`
2. Descomprimir `java_cup_v10k.zip`
3. Añadir a la variable de entorno `CLASSPATH`.



→ CUP – Ejecución

1. Para crear el analizador sintáctico, debemos invocar el intérprete de Java y ejecutar el método `java_cup.Main`:

```
java java_cup.Main <opciones> <espec.cup>
```

- Si se omite el nombre del fichero de especificación sintáctica, CUP interpreta que la especificación sintáctica se introducirá por la entrada estándar.

2. Compilar las clases necesarias:

```
javac parser.java sym.java <Otras_Clases>
```



→ CUP - El código generado

- CUP genera dos ficheros con los nombres, por defecto:
 - **sym.java** contiene las definiciones de constantes de la clase sym, que asigna un valor entero a cada terminal y, opcionalmente, a cada no terminal
 - **parser.java** clase pública del analizador sintáctico.



→ Ejemplo – CUP sym.java

```
//-----  
// The following code was generated by CUP v0.10k  
// Sun Mar 11 01:33:31 CET 2007  
//-----  
  
/** CUP generated class containing symbol constants. */  
public class sym {  
    /* terminals */  
    public static final int RPARENT = 10;  
    public static final int DIGITO = 6;  
    public static final int POTENCIA = 7;  
    public static final int SUMA = 2;  
    public static final int MULTIPLICACION = 5;  
    public static final int EOF = 0;  
    public static final int SIGNO = 11;  
    public static final int RESULTADO = 8;  
    public static final int error = 1;  
    public static final int DIVISION = 4;  
    public static final int RESTA = 3;  
    public static final int LPARENT = 9;  
}
```



→ Clase analizador sintáctico `parser.java`

- `parser.java` contiene 2 clases:

- `public class parser`

- `extends java_cup.runtime.lr_parser{ ... }`

- Clase pública del analizador:

- Subclase de `java_cup.runtime.lr_parser` que implementa la tabla de acciones de un analizador LALR.

- `class CUP$parser$actions`

- Clase no pública incluida en el fichero que encapsula las acciones de usuario contenidas en la gramática.

- Contiene el método que selecciona y ejecuta las diferentes acciones definidas en cada regla sintáctica:

- `public final java_cup.runtime.Symbol CUP$parser$do_action`



→ Tablas de `parser.java`

`protected static final short[][] _production_table`

- La tabla de producción contiene el número de símbolo de la izquierda de la producción, para cada producción de la gramática.

`protected static final short[][] _action_table`

- La tabla de acciones, donde se indica qué acción (desplazamiento, reducción o error) debe utilizarse para cada símbolo anticipado (*lookahead*) en función del estado en que se encuentre el análisis.

`protected static final short[][] _reduce_table`

- La tabla de saltos a estados indica estado al que debe irse después de una reducción, en función del no-terminal y el estado en que se encuentra el análisis.



→ Método `parse`

- El análisis es realizado por el método

```
public Symbol parse()
```

- La invocación del analizador puede realizarse con un código como el siguiente:

```
/* declara el objeto parser */  
parser analizer;  
...tratamiento de ficheros y otros...  
/* Creación */  
analizer = new parser ( );  
/* Llamada al método parse */  
analizer.parse();
```



→ Estructura de un fichero CUP

1. **Especificaciones de importación y empaquetamiento.** La primera parte del fichero se utiliza para incluir información que define cómo se generará el analizador y el código de importación de la librería CUP.
2. **Código de usuario.** Después de las especificaciones de importación y empaquetamiento, se puede ubicar una serie de declaraciones opcionales que permiten incluir código en el analizador generado.
3. **Lista de símbolos.** La segunda parte de la especificación declara los terminales y no-terminales utilizados en la gramática y, opcionalmente, especifica el tipo (clase-objeto) de cada uno de ellos. Si se omite el tipo, al terminal o no-terminal correspondiente no se le pueden asignar valores.
4. **Declaraciones de precedencia.** La tercera parte del fichero define la precedencia y asociatividad de los terminales.
5. **Gramática.** La última parte del fichero contiene la gramática del lenguaje. No se utiliza ningún delimitador o separador de secciones, sino que se sitúan secuencialmente en el fichero de especificación sintáctica.

Nota: Estas 5 partes (algunas opcionales) deben introducirse en orden



→ Gramática CUP

- Notación similar a la forma de **Backus-Naur** (BNF).
- Se inicia con una declaración opcional de la forma:

`start with no-terminal`

- La gramática se especifica con producciones del tipo:

```
no-terminal ::= secuencia_de_terminales_y/o_noterminales [ { : acción
                  :} ]
[secuencia_de_terminales_y/o_noterminales[{ :acción:}]]
...
[asignación_contextual_de_precedencia]
  [|secuencia_de_terminales_y/o_noterminales[{ :acción:}]]
]
...
[asignación_contextual_de_precedencia ] ;
```

- La `secuencia_de_terminales_y/o_noterminales` es una lista de terminales o no terminales separados por espacios en blanco.



→ Definición de la gramática

Operación	Expresión regular	CUP
Concatenación	AB	$C ::= AB ;$
Opción	$A \mid B$	$C ::= A \mid B ;$
	$\varepsilon \mid A$	$C ::= \mid A ;$
Clausura de Kleene	A^*	$C ::= \mid AC ;$
	A^+	$C ::= A \mid AC ;$



→ Ejemplo: Calculadora

- En cada sesión, se pueden calcular una o varias ecuaciones. Es a decir:
 - `sesion ::= ecuacion | ecuacion sesion ;`
- Las ecuaciones que acaban con el símbolo (=)
 - El terminal = indica que finde la lectura y se puede evaluar. Ej.:
 - `2 + 5 * 4 =`
 - `2 * 4 / 3 * 5 =`
 - `ecuacion ::= expr IGUAL ;`
- Operaciones binarias:
 - `digito operador_binario digito`. Ej.: `(4+5)`
- Operaciones unarias:
 - `operador_unario digito` Ej.: `(-7)`
- Finalmente, operaciones anteriores, entre paréntesis:
 - `(expresion)`



→ Ejemplo: Calculadora - gramática

```
expr ::= DIGITO
      | expr opBinaria expr
      | opUnaria expr
      | LPARENT expr RPARENT ;
```

```
opUnaria ::= RESTA ;
```

```
opBinaria ::= SUMA
           | RESTA
           | MULTIPLICACION
           | DIVISION
           | POTENCIA ;
```



→ Lista de símbolos

- Sección obligatoria consistente en una o varias listas con los nombres, y el tipo de datos de los terminales y no terminales de la gramática:

```
terminal [clase] nombre1, nombre2, ...;  
non terminal [clase] nombre1, nombre2, ...;
```

- Ej.:

```
terminal SUMA, RESTA, DIVISION, MULTIPLICACION;  
terminal POTENCIA , DIGITO, RESULTADO;  
terminal LPARENT, RPARENT ;  
non terminal sesion, ecuacion, expresion, opbinaria;
```

– Nota: es sección obligatoria y no pueden ser palabras reservadas de CUP



→ Especificación de importación y paquetes

- La 1ª parte del fichero CUP empieza con una declaración opcional de importaciones y paquetes. Estas declaraciones tienen la misma sintaxis y finalidad que cualquier programa Java:

```
package nombre_package;
```

- La declaración de paquete indica el paquete en el que se ubicarán las clases sym y parser generadas por el sistema.

- A continuación de estas, se pueden incluir declaraciones de importación con la misma sintaxis que en Java:

```
import nombre_paquete.nombre_clase;  
import nombre_paquete.*;
```

- Sentencia de importación habitual es la de runtime de CUP:

```
import java_cup.runtime
```



→ Código de usuario

- El código de usuario que puede incluirse en las clases generadas por CUP es:
 1. Código añadido a la clase de las acciones
 2. Código añadido a la clase del analizador
 3. Código previo al inicio del análisis
 4. Código de adquisición de testigo



→ Código de Usuario: Código añadido a la clase de las acciones

- El código con las acciones semánticas se implementa dentro de la clase **CUP\$parser\$actions**. Si se desea añadir código adicional a esta clase, se puede hacer con la declaración siguiente:

```
action code {:  
  ...código añadido a las acciones...  
:};
```

- Como resultado, el código añadido se incluye, literalmente, en la clase **CUP\$parser\$actions**, tal y como sigue:

```
class CUP$parser$actions {  
  ...código añadido a las acciones...  
  <resto del código >  
}
```

- Nota: se incluye código que se necesitará en las acciones semánticas incluidas en la sección de la gramática, y más específicamente, las rutinas de manipulación de la tabla de símbolos del compilador que queremos crear.



→ Código de Usuario: Código añadido a la clase del analizador

- Se puede añadir código en la clase **parser** mediante la declaración:

```
parser code {:  
  ...Código añadido al analizador...  
};
```

- Esto hace que el código definido se copie íntegramente en la clase:

```
public class parser extends java_cup.runtime.lr_parser {  
  <código del analizador>  
  ...Código añadido al analizador...  
}
```

- Se utiliza para particularizar el analizador, incluyendo métodos o, sobrescribiendo métodos de información de error de CUP.



→ Código de Usuario: Código previo al inicio del análisis

- Este código se incluye en la clase parser, para que se ejecute antes de iniciar la exploración:

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

- El código se copia íntegramente en la función **user_init** de la clase parser:

```
public void user_init() throws java.lang.Exception  
{  
...Código Inicial...  
}
```

- Habitualmente se utiliza para inicializar el analizador y, especialmente, tablas y otras estructuras que eventualmente se necesiten para las acciones semánticas.



→ Código de Usuario: Código de adquisición de testigo

- La última sección opcional de código de usuario indica cómo se deben solicitar los testigos al analizador léxico. Tiene la forma:

```
scan with {:  
    ...código adquisición testigos...  
};
```

- El siguiente código se copia íntegramente en el método **scan**, que retorna un objeto de la clase **Symbol**

```
public java_cup.runtime.Symbol scan() throws  
    java.lang.Exception  
{  
    ...código adquisición testigos...  
}
```

- Si se omite, CUP, por defecto, genera:

```
public Symbol scan() throws java.lang.Exception {  
    Symbol sym = getScanner().next_token();  
    return (sym!=null) ? sym : new Symbol(EOF_sym());  
}
```



→ Declaraciones de precedencia y asociatividad

- Hay tres tipos de declaraciones:

```
precedence left terminal11[, terminal12...];  
precedence right terminal21[, terminal22...];  
precedence nonassoc terminal31[, terminal32...];
```

- Precedencia:

- Los terminales en las últimas listas tienen mayor precedencia que las que aparecen en las primeras. El orden de precedencia va de abajo hasta arriba

- En cuanto a asociatividad:

- Los terminales incluidos en una lista de precedencia-asociatividad tiene la asociatividad que se detalla en la declaración, pudiendo ser:

- **right**: asociatividad por la derecha.
- **left**: asociatividad por la izquierda.
- **nonassoc**: sin asociatividad.

- Si un terminal no está incluido en ninguna declaración de precedencia-asociatividad, se le asigna nonassoc.

- La asociatividad de los terminales se utiliza para resolver conflictos de desplazamiento-reducción, pero únicamente se considera para terminales de igual precedencia.



→ Declaraciones de precedencia y asociatividad

- Ejemplo:

`precedence left RESTA, SUMA;`

`precedence left MULTIPLICACION, DIVISION;`

`precedence left MENOSUNARIO;`

`precedence right POTENCIA;`

- Las operaciones suma y resta son las de menor precedencia y la potencia la de mayor.
- La operación potencia es la única con asociatividad por la derecha. La resta lo son por la izquierda.



→ Precedencia contextual

- La asignación contextual de precedencia se sitúa al final de la parte derecha de la producción y se utiliza para modificar la precedencia que inicialmente se ha definido para los terminales.

```
no-terminal ::=
    secuencia_de_terminales_y/o_noterminales [ {: acción
    :} ]
[ secuencia_de_terminales_y/o_noterminales [ {: acción
    :} ] ]
...
[ asignación_contextual_de_precedencia ]
[ | secuencia_de_terminales_y/o_noterminales [ {:
    acción :} ]
]
...
[ asignación_contextual_de_precedencia ] ;
```



→ Precedencia contextual. Ejemplo:

- El signo menos ('-') representa tanto la resta (operador binario), como el cambio de signo (operador unario).
 - Si el signo corresponde al primero de los casos, su precedencia será menor que, por ejemplo, el operador multiplicación. Si se trata del segundo caso, su precedencia será mayor.

```
precedence left RESTA, SUMA;  
precedence left MULTIPLICACION, DIVISION;  
precedence left SIGNO;  
precedence right POTENCIA;  
...  
expr ::= RESTA expr { : ... : }  
      %prec SIGNO  
...
```



→ Precedencia de producciones

- CUP también asigna una precedencia a cada una de las producciones de la gramática.
 - Esta precedencia es igual a la precedencia del último terminal de la producción y, si la producción no contiene terminales, se le asigna la menor. Por ejemplo:
 - `expr ::= expr SUMA expr`
 - misma precedencia que el terminal SUMA
 - `expr ::= expr opBin expr`
 - la menor precedencia entre las producciones
- Para resolver los conflictos de desplazamiento-reducción, el analizador procede como sigue:
 - Se determina que tiene mayor precedencia: terminal a desplazar o la producción a reducir.
 - Si el terminal tiene la precedencia mayor, se desplaza.
 - Si es la producción la que tiene mayor precedencia, se reduce.
 - Si los dos tienen la misma precedencia, la acción se determina a partir de la asociatividad del terminal



→ Particularización del analizador

- Métodos que se pueden definir o sobrescribir:
 - `public void user_init()`. Este método es llamado por el parser antes de solicitar el primer testigo al explorador. El cuerpo del método contiene el código definido con la cláusula `%init{...%}init`.
 - `public java_cup.runtime.Scanner getScanner()`. Este método retorna el explorador por defecto
 - `public java_cup.runtime.Symbol scan()` Este método encapsula el explorador y se llama cada vez que el analizador sintáctico precisa un nuevo terminal. Por defecto, retorna:
 - `getScanner().next_token()`
 - `public void setScanner(java_cup.runtime.Scanner s)`
Establece el explorador que utilizará el analizador.
 - `public void report_error(String message, Object info)`
Este método debería llamarse siempre que haya que mostrar un mensaje de error.



→ Particularización del analizador

- CUP, por defecto, imprime por la salida estándar de error (**System.err**) el 1er parámetro, que contiene el texto del mensaje de error y, si el 2do parámetro es una instancia de la clase **Symbol**, imprime información adicional sobre la situación, dentro del fichero de entrada, donde se ha producido el error. Por defecto es:

```
public void report_error (String message, Object info) {
    System.err.print(message);
    if (info instanceof Symbol)
        if (((Symbol)info).left != -1)
            System.err.println(" at character " +
                               (((Symbol)info).left + " of input");
        else System.err.println("");
    else
        System.err.println("");
}
```



→ Particularización del analizador

- **public void report_fatal_error(String message, Object info)**

- A este método debe llamarse cuando se produce un error irre recuperable. Básicamente realiza tres acciones: llama al método `done_parsing()` para detener el análisis, informa del error con el método anterior y lanza una excepción.

```
public void report_fatal_error( String message, Object
    info) throws java.lang.Exception {
    done_parsing();
    report_error(message, info);
    throw new Exception("Can't recover from previous
                        error");
}
```



→ Particularización del analizador

- **public void syntax_error(Symbol cur_token)**

- Este método es invocado por el analizador al detectar un error sintáctico, y previamente al intento de recuperarlo. En la implementación por defecto proporcionada por CUP únicamente se llama al método report_error.

```
public void syntax_error(Symbol cur_token) {  
    report_error("Syntax error", cur_token);  
}
```



→ Particularización del analizador

- `public void unrecovered_syntax_error(Symbol cur_token)`
 - Invocado por el analizador si es imposible recuperarse de un error sintáctico. La implementación proporcionada por CUP es:

```
public void unrecovered_syntax_error(Symbol cur_token)
    throws java.lang.Exception {
    report_fatal_error("Couldn't repair and continue
                        parse", cur_token);
}
```



→ Particularización del analizador

- **protected int error_sync_size()**

- Este método es llamado por el analizador para determinar cuantos testigos debe analizar correctamente para considerar que la recuperación ha tenido éxito.
- Por defecto, retorna la constante **error_sync_size**, con valor 3.
 - No se recomienda un valor menor de 2.

```
protected final static int _error_sync_size = 3;
protected int error_sync_size( ) {
    return _error_sync_size;
}
```



→ Particularización del analizador

- `debug_parse()`
- `debug_message (String m)`
 - Además del analizador que se ha utilizado hasta el momento (`parse()`), CUP proporciona una versión de depuración. Ésta funciona exactamente igual que la anterior, excepto en que imprime mensajes por la salida de error estándar (`System.err`) informando del proceso de análisis. Esta información es realizada por el método `debug_message (String m)`:

```
public void debug_message (String mess) {  
    System.err.println (mess) ;  
}
```



→ Recuperación de errores

- Cuando el analizador sintáctico generado por CUP recibe una secuencia de testigos que no corresponde a la gramática que ha de analizar, se genera un error y el analizador finaliza su ejecución.
- CUP proporciona un mecanismo de recuperación ante los errores, consistente en definir un no terminal especial (**error**) que encaja toda entrada errónea.
 - El símbolo de error sólo está activo en caso de detectarse un error. En este caso, el analizador intenta reemplazar una secuencia de testigos para este no terminal y continuar el análisis.
- El analizador únicamente considerará que la recuperación ha tenido éxito si, después del no terminal error, puede analizar correctamente un determinado número de testigos (por defecto, 3).



→ Recuperación Errores – Símbolo **error**

```
ecuacion ::= expresion
```

```
    { : System.out.println("Sintaxis correcta"); : }
```

```
    RESULTADO
```

```
| error { : System.out.println("Sintaxis incorrecta"); : }
```

```
    RESULTADO ;
```

- Si se detecta un error en el análisis de una expresión, la recuperación se intentará descartando testigos de la entrada hasta encontrar el testigo **RESULTADO** ('=').



→ Recuperación Errores - Comportamiento

- Con símbolos de error, CUP genera las diferentes tablas normalmente. La diferencia radica en el comportamiento al detectar una entrada errónea:
 - Al encontrar un error, el analizador extraerá símbolos de la pila hasta que encuentre el estado más alto que contenga, dentro de su conjunto de elementos, uno de la forma $A \rightarrow \text{error } \alpha$
 - Entonces, el analizador introducirá un componente ficticio error en la pila, simulando que corresponde a la entrada actual.
 - Si α es la cadena vacía, se produce una reducción a A y se ejecuta la acción asociada a la producción $A \rightarrow \text{error}$ que, presumiblemente, será una rutina de recuperación del error que hemos implementado. Entonces elimina símbolos de entrada hasta que encuentra un símbolo con el que pueda seguir el análisis normal.
 - Si α no es vacía, el analizador buscará en la entrada una cadena que se pueda reducir a α . Así, si α sólo contiene terminales, buscará en esta cadena de terminales en la entrada y los reducirá, desplazándolos a la pila. De esta manera, la cima de la pila contiene error α y se podrá reducir a A ($A \rightarrow \text{error } \alpha$).
 - Después de este proceso seguirá el análisis, sin ejecutar ninguna acción, hasta conseguir analizar correctamente un número `error_sync_size()` testigos de entrada. Si lo consigue, se considera que ha conseguido recuperarse del error; se vuelve al punto anterior y continúa el análisis, ejecutando las acciones asociadas que en el proceso de búsqueda no había ejecutado. Si no consigue analizar este número de testigos, se descarta uno de los testigos de entre los analizados y reinicia la búsqueda más adelante. La recuperación de error falla si se llega al final del fichero de entrada o si, inicialmente, no se encuentra ninguna producción de error en la pila de estado.



→ Recuperación Errores - Ejemplo:

	Lex	Token	Acción	Pila
1	2	DIGITO	Desplazar	DIGITO
2			Reducir por $\text{expr} \rightarrow \text{DIGITO}$	expr
3	+	SUMA	Desplazar	SUMA; exp
4	+	SUMA	ERROR	SUMA; exp
Búsqueda en la pila de estados de uno que tenga una producción de error: $\text{ecuacion} ::= \text{error RESULTADO}$				
5			Introducir error en la pila	error
Buscar RESULTADO (=)				
6	2	DIGITO	Descartar	
7	=	RESULTADO	Introducir en la pila	RESULTADO; error
8			Reducir por $\text{ecuacion} ::= \text{error RESULTADO}$	Ecuacion
9	2	DIGITO	(Búsqueda hacia adelante: 1)	DIGITO; ecuacion
10			Reducir por $\text{expr} \rightarrow \text{DIGITO}$	Expr; ecuacion
11	+	+	(Búsqueda hacia adelante: 2)	SUMA; expr; ecuacion
12	1	DIGITO	(Búsqueda adelante: 3) $\text{expr} \rightarrow \text{DIGITO}$; $\text{expr} \rightarrow \text{expr SUMA expr}$	Expr; ecuacion
Recuperación validada. Restauración y reinicio del análisis.				
8				ecuacion
9	2	DIGITO	Desplazar	DIGITO; ecuacion
...



→ Recuperación Errores. Posición de símbolos **error**

- En general, cuanto más cerca de la regla del símbolo inicial de la gramática (arriba) se sitúa la producción de error, mayor será el número de *tokens* que descartan por el analizador durante la recuperación.
- Si se sitúan lejos de la regla inicial (abajo), tendremos menos posibilidades de realizar una recuperación correcta del error.
 - Además hay que considerar es que su ubicación no genere cadenas de nuevos errores.
- Típicamente se deben situar producciones de error son subconjuntos de los no terminales que generan expresiones, proposiciones, bloques y procedimientos.



→ Opciones en la fase de creación

```
Command Prompt

Usage: java_cup [options] [filename]
and expects a specification file on standard input if no filename is given.
Legal options include:
  -package name    specify package generated classes go in [default none]
  -parser name     specify parser class name [default "parser"]
  -symbols name    specify name for symbol constant class [default "sym"]
  -interface      put symbols in an interface, rather than a class
  -nonterms       put non terminals in symbol constant class
  -expect #       number of conflicts expected/allowed [default 0]
  -compact_red    compact tables by defaulting to most frequent reduce
  -nowarn         don't warn about useless productions, etc.
  -nosummary      don't print the usual summary of parse states, etc.
  -nopositions    don't propagate the left and right token position values
  -noscanner      don't refer to java_cup.runtime.Scanner
  -progress       print messages to indicate progress of the system
  -time          print time usage summary
  -dump_grammar   produce a human readable dump of the symbols and grammar
  -dump_states    produce a dump of parse state machine
  -dump_tables    produce a dump of the parse tables
  -dump          produce a dump of all of the above
  -version        print the version information for CUP and exit

C:\compiladores>
```



→ Ejemplo Calculadora: Explorador para una calculadora

```
import java.io.*;
import java_cup.runtime.Symbol;
class Scan implements java_cup.runtime.Scanner {
    private FileIn;
    Scan (instream) {
        FileIn = new BufferedReader(new java.io.InputStreamReader(instream)); }
    public java_cup.runtime.Symbol next_token () throws java.io.IOException {
        int c=0;
        do {
            try {c=FileIn.read();}
            catch(IOException e) {System.out.println ("ERROR lectura ");}
            switch (c) {
                case '1': case '2': case '3': case '4': case '5':
                case '6': case '7': case '8': case '9': case '0':
                    { return new Symbol (sym.DIGITO); }
                case '+': { return new Symbol (sym.SUMA); }
                case '-': { return new Symbol (sym.RESTA); }
                case '*': { return new Symbol (sym.MULTIPLICACION); }
                case '/': { return new Symbol (sym.DIVISION); }
                case '^': { return new Symbol (sym.POTENCIA); }
                case '(': { return new Symbol (sym.LPARENT); }
                case ')': { return new Symbol (sym.RPARENT); }
                case '=': { return new Symbol (sym.RESULTADO); }
                default: { break; }
            }
        } while (true);
    }
}
```



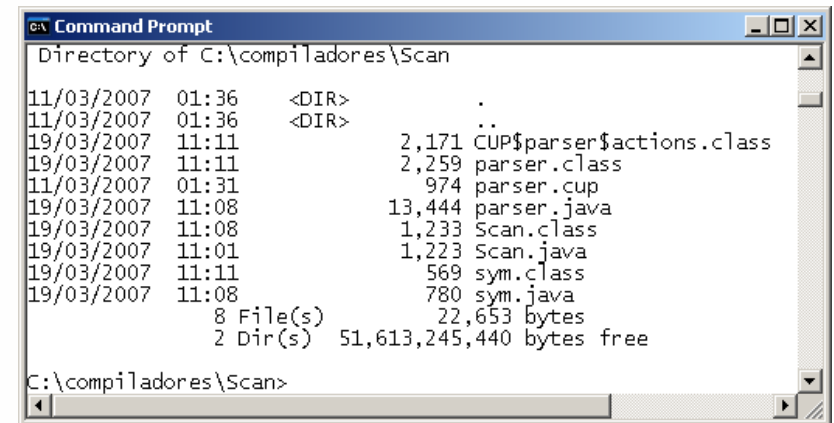
→ Ejemplo Calculadora: Parser

```
import java_cup.runtime.*; import java.io.*;
parser code {
    public static void main (String argv[]) throws Exception {
        parser analizador;
        analizador = new parser(new Scan(System.in));
        analizador.parse();
    } :};
// terminales y no terminales
terminal SUMA, RESTA, DIVISION, MULTIPLICACION, DIGITO;
terminal POTENCIA, RESULTADO, LPARENT, RPARENT, SIGNO;
non terminal sesion, ecuacion, expresion, opbinaria;
// precedencia y asociatividad
precedence left RESTA, SUMA;
precedence left MULTIPLICACION, DIVISION;
precedence left SIGNO;
precedence right POTENCIA;
// gramatica
sesion ::= ecuacion
        | ecuacion sesion;
ecuacion ::= expresion { : System.out.println("Sintaxis correcta"); : } RESULTADO
        | error { : System.out.println("Sintaxis incorrecta"); : } RESULTADO;
expresion ::= DIGITO
        | expresion opbinaria expresion
        | RESTA expresion
%prec SIGNO
        | LPARENT expresion RPARENT ;
opbinaria ::= SUMA
        | RESTA
        | DIVISION
        | MULTIPLICACION ;
```



→ Ejemplo Calculadora: Compilación-ejecución

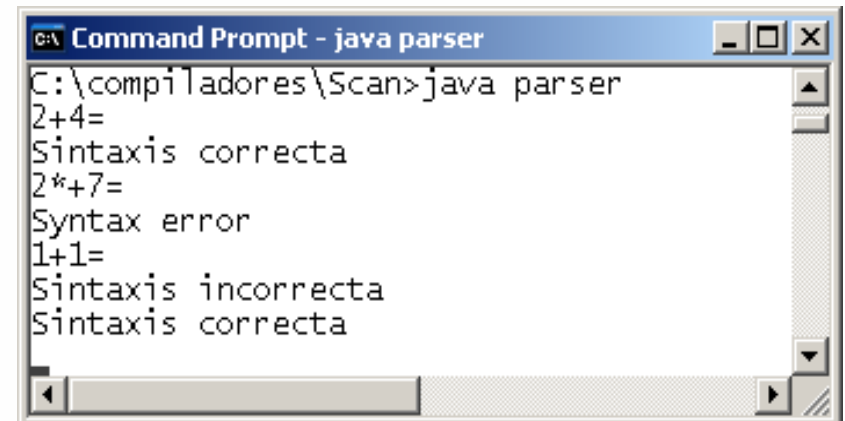
- Generar *parser*:
 `>java java_cup.Main parser.cup`
- Compilar:
 `>javac Scan.java`
 `>javac sym.java parser.java`
- Ejecución:
 `>java parser`



```
Command Prompt
Directory of C:\compiladores\Scan

11/03/2007  01:36    <DIR>          .
11/03/2007  01:36    <DIR>          ..
19/03/2007  11:11                2,171 CUP$parser$actions.class
19/03/2007  11:11                2,259 parser.class
11/03/2007  01:31                974 parser.cup
19/03/2007  11:08            13,444 parser.java
19/03/2007  11:08            1,233 Scan.class
19/03/2007  11:01            1,223 Scan.java
19/03/2007  11:11                569 sym.class
19/03/2007  11:08                780 sym.java
               8 File(s)                22,653 bytes
               2 Dir(s)  51,613,245,440 bytes free

C:\compiladores\Scan>
```



```
Command Prompt - java parser
C:\compiladores\Scan>java parser
2+4=
Sintaxis correcta
2*+7=
Syntax error
1+1=
Sintaxis incorrecta
Sintaxis correcta
```



→ Interconexión JLex-CUP

- Necesitamos configurar JLex para que el explorador que genera sea compatible con el analizador generado por CUP.



→ Estructura de la clase **Symbol**

- El analizador sintáctico representa a los terminales y no terminales con un objeto de la clase **Symbol**. Esta clase contiene los datos:
 - `public Object value`: es el valor léxico del terminal o no terminal.
 - `public int left`: posición izquierda, dentro del fichero.
 - `public int right`: posición derecha, dentro del fichero.
 - `public int sym`; : Este valor se corresponde con el valor entero correspondiente al terminal o no terminal, tal y como se ha enumerado en el fichero `sym.class`.
 - `public int parse_state`: estado del análisis. Este campo es utilizado internamente por el analizador sintáctico y no es recomendable su modificación.



→ Declarando la lista de símbolos

- En la sección de la lista de símbolos se enumeraban todos los terminales y no-terminales que necesitaba la gramática:

```
- terminal [ clase] nombre1, nombre2, ...;  
- non terminal [ clase] nombre1, nombre2, ...;
```

- Si utilizamos todos los datos del objeto **Symbol**, las reglas en el fichero JLex, serán :

```
expresion_regular { return new symbol (  
    sym.cnt_terminal,  
    pos_izquierda,  
    pos_derecha,  
    new objeto ( lista_campos ) ); }
```



→ Ejemplo Símbolos JLex-CUP

```
[\\r\\n\\t ]+ { /*prescindir de blancos*/ }
"+"          { return new Symbol (sym.SUMA); }
"-"          { return new Symbol (sym.RESTA); }
"*"          { return new Symbol (sym.MULTIPLICACION); }
"/"          { return new Symbol (sym.DIVISION); }
"^"          { return new Symbol (sym.POTENCIA); }
"("          { return new Symbol (sym.LPARENT); }
")"          { return new Symbol (sym.RPARENT); }
"="          { return new Symbol (sym.RESULTADO); }
[0-9]+       { return new Symbol (sym.ENTERO,
                                new Integer(yytext())); }

[^0-9\\r\\n\\t \\+\\-\\*\\^\\/]+
    { System.out.println("Error léxico:" + yytext()); }
```



→ Compatibilidad JLex-CUP

- CUP necesita que el explorador que le proporcione los *tokens* conforme a la siguiente interface :

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

- Para cumplir este requisito, el fichero de especificación léxica JLex tiene que contener el código siguiente:

```
%implements java_cup.runtime.Scanner  
%function next_token  
%type java_cup.runtime.Symbol
```



→ Compatibilidad JLex-CUP. Directivas

%implements java_cup.runtime.Scanner

- Hace que la clase generada por JLex implemente el explorador exigido por CUP (“Scanner”):

- `class Yylex implements java_cup.runtime.Scanner{...`

%function next_token

- Cambia el nombre por defecto de la función de análisis léxico, `yylex()`, por `next_token()`.

%type java_cup.runtime.Symbol

- el testigo retornado por la función de exploración sea del tipo `Symbol`.

%cup

- Estas tres directivas se pueden compactar con esta única directiva



→ Ejemplo JLex

```
import java_cup.runtime.Symbol;
%%
%full
%notunix
%cup
%%
[\\r\\n\\t ]+ { /*prescindir de blancos*/ }
"+"      { return new Symbol (sym.SUMA); }
"-"      { return new Symbol (sym.RESTA); }
"*"      { return new Symbol (sym.MULTIPLICACION); }
"/"      { return new Symbol (sym.DIVISION); }
"^"      { return new Symbol (sym.POTENCIA); }
"("      { return new Symbol (sym.LPARENT); }
")"      { return new Symbol (sym.RPARENT); }
"="      { return new Symbol (sym.RESULTADO); }
[0-9]+ { return new Symbol (sym.ENTERO, new Integer (yytext())); }
[^0-9\\r\\n\\t \\+\\-\\*\\^\\/ ]+ {
    System.out.println("Error léxico: "+ yytext() ); }
```



→ Directiva %cup

- Con %cup, el nombre de la clase generada por JLex mantiene su valor por defecto (**Yylex**). Por tanto, la llamada al explorador en el fichero de especificación sintáctica CUP será:

```
/* Creación y inicialización */  
parser aParser = new parser(new Yylex(...));  
/* Iniciar análisis */  
aParser.parse();
```

- O bien con los métodos **setScanner()** y **getScanner()**:

```
/* crear el objeto parser */  
parser un_parser = new parser();  
/* Establecer el explorador */  
un_parser.setScanner( new Yylex( ... ) );  
/* Iniciar análisis */  
un_parser.parse();
```



→ Acceso a los valores semánticos

- Para acceder a los atributos, los símbolos contenidos en las reglas sintácticas pueden referenciarse con etiquetas:

– `noterm1 ::= term1:e1 {accion1} term2:e2 {accion2} ...;`

- Las etiquetas e_i representan instancias del objeto (value) que contiene el símbolo y desde las acciones se puede acceder a los diferentes campos con la referencia convencional de Java:

- `ei.atributon`



→ Ejemplo: Calculadora – parser.cup

```
// terminales y no terminales
terminal SUMA, RESTA, DIVISION, MULTIPLICACION;
terminal Integer ENTERO;
terminal POTENCIA, RESULTADO, LPARENT,
    RPARENT, SIGNO;
non terminal sesion, ecuacion;
non terminal Integer expression;

precedence left RESTA, SUMA;
precedence left MULTIPLICACION, DIVISION;
precedence right SIGNO;

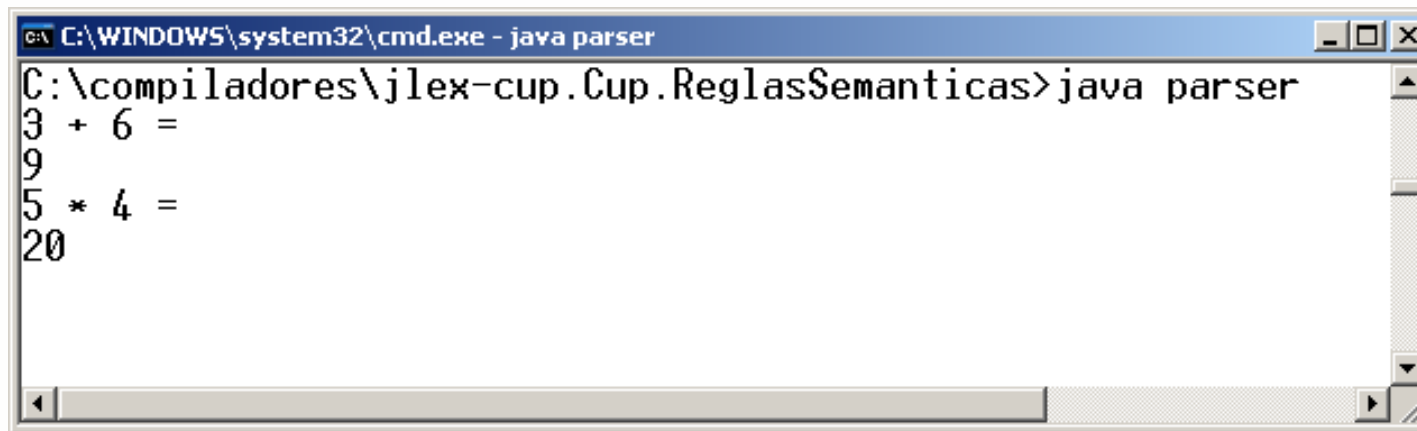
// gramática
sesion ::= ecuacion
| ecuacion sesion
;

ecuacion ::= expresion:E1
    {: System.out.println(E1.intValue()); :}
RESULTADO
;
```

```
expresion ::= ENTERO:E1
    {:RESULT = new Integer (E1.intValue()); :}
| expresion:E1 SUMA expresion:E2
    {:RESULT=new Integer( E1.intValue() +
                        E2.intValue()); :}
| expresion:E1 RESTA expresion:E2
    {:RESULT=new Integer( E1.intValue() -
                        E2.intValue()); :}
| expresion:E1 MULTIPLICACION expresion:E2
    {:RESULT=new Integer( E1.intValue() *
                        E2.intValue()); :}
| expresion:E1 DIVISION expresion:E2
    {:RESULT=new Integer(E1.intValue() /
                        E2.intValue()); :}
| LPARENT expresion:E1 RPARENT
    {:RESULT=new Integer(E1.intValue()); :}
| RESTA expresion:E1
    {: RESULT=new Integer(0-E1.intValue()); :}
%prec SIGNO
;
```



→ Ejemplo: Calculadora – Salida



```
C:\WINDOWS\system32\cmd.exe - java parser
C:\compiladores\jlex-cup.Cup.ReglasSemanticas>java parser
3 + 6 =
9
5 * 4 =
20
```

