Grado en Ingeniería Informática

Procesadores de lenguajes - Análisis sintáctico

Generación de analizadores sintácticos con CUP

Objetivos de la práctica

- Conocer la herramienta CUP para la generación de analizadores sintácticos LR.
- Generar analizadores sintácticos LR con la herramienta CUP y JFlex.

Actividades a realizar

1. Familiarización con la herramienta CUP.
2. Conexión de analizadores léxicos creados con JFlex con analizadores sintácticos creados con CUP.
3. Reconocer conceptos de analizadores LR en los analizadores creados por CUP.

Familiarización con la herramienta CUP

CUP es un generador de analizadores sintácticos ascendentes. El código fuente de los analizadores generados está escrito en el lenguaje Java. La herramienta, de libre distribución, se puede:

- Encontrar en los ordenadores del laboratorio, en el "Escritorio de desarrollo" en la ruta "C:\Cup 011b\" (JFlex está en C:\jflex-1.6.0\)
- Descargar desde la dirección web http://www2.cs.tum.edu/projects/cup/.

CUP se ha desarrollado en java por lo que su instalación y ejecución seguirán los procedimientos típicos del software desarrollado en java. CUP es una herramienta sin interfaz gráfica, por lo que la forma de invocarla es mediante una ventana de línea de comandos o integrándola en las herramientas asociadas a los entornos de programación, por ejemplo ANT. El comando requerido para su ejecución por línea de comandos es:

```
java –jar java-cup-11a.jar especificacionSintactica.cup
```

El esquema de funcionamiento es similar al de la herramienta JFlex vista anteriormente. El usuario proporciona la especificación gramatical, con ella CUP genera el código fuente del analizador sintáctico.

Tras la ejecución de Cup se generan dos ficheros:

- **`parser.java`**: contiene la implementación del analizador sintáctico que se ha definido previamente dentro del fichero especificacionSintactica.cup (el nombre de este fichero es elegido por el usuario).
- **`sym.java`**: contiene el conjunto de símbolos terminales que hemos definido en la especificación sintáctica.

Las especificaciones sintácticas que toma CUP a su entrada se dividen en dos partes principales: preámbulo y especificación gramatical.

En el preámbulo, siempre que sea necesario, se incluyen los siguientes elementos:

- *Importación de paquetes Java requeridos*: como se ha indicado, CUP genera un analizador sintáctico escrito en lenguaje Java. Este analizador, en función de las necesidades que tenga el usuario, puede requerir de otras librerías o clases que el usuario haya empleado. Por ello, lo primero que puede aparecer en un archivo de especificación sintáctica para CUP son declaraciones **`import librerías;`**.
- *Action code*: CUP permite ejecutar código asociado a las distintas producciones gramaticales. Esta sección que permite al usuario introducir código Java (métodos y atributos) que pueda ser utilizado posteriormente en las acciones semánticas.

```
action code
{:
// código Java accesible desde acciones semánticas …
:}
```

- *Parser code*: es una sección que permite al usuario introducir código Java que irá incluído en la clase **`parser`**, a diferencia del código escrito en la sección *Action code*. En esta sección se suelen sobrecargar métodos existentes de las librerías de CUP o crear otros nuevos.

```
parser code
{:
// código Java, integrado en clase 'parser'
:}
```

En la especificación gramatical se definen los siguientes elementos:

- Definición de símbolos terminales: lista de símbolos terminales declarados con **`terminal lista_de_símbolos, …;`**
- Definición de símbolos no terminales: lista de símbolos no terminales declarados con **`non terminal lista_de_símbolos, …;`**
- Especificación de la gramática: lista de producciones gramaticales declaradas con **`ANTECEDENTE ::= CONSECUENTE1 |  CONSECUENTE2 ;`**

Más información en: http://www2.cs.tum.edu/projects/cup/docs.php

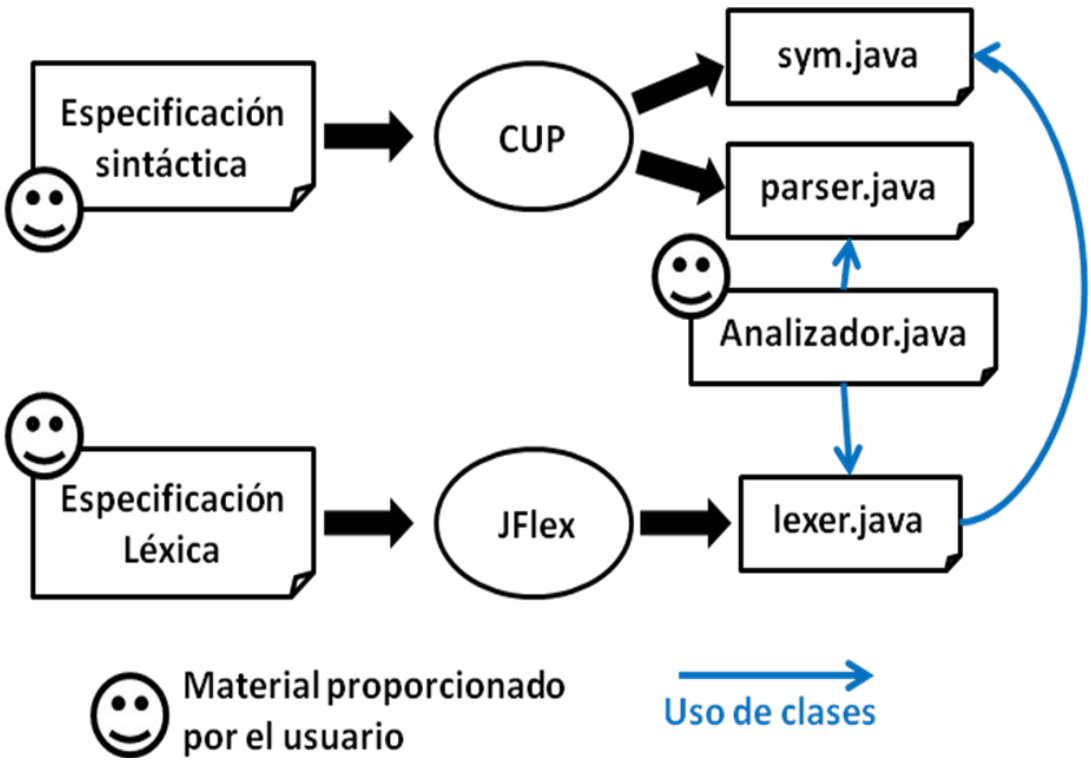
Conexión de analizadores léxicos creados con JFlex con analizadores sintácticos creados con CUP

La unión del analizador léxico generado con JFlex y del analizador sintáctico generado por CUP se realiza teniendo en cuenta las siguientes cuestiones:

- Emplear el **mismo conjunto de símbolos terminales** tanto en la especificación léxica realizada para JFlex como en la especificación sintáctica que se le pretende pasar a CUP.
- Modificar la especificación léxica de JFlex (ver detalles en el anexo 1) para:
  - habilitar la **compatibilidad de JFlex con CUP** mediante la directiva `%cup`.
  - eliminar la directiva `%standalone` si es que existía.
  - devolver los tokens leídos mediante la sentencia "return" de Java dentro de la especificación léxica creando instancias de la clase Symbol, clase que maneja CUP para la gestión interna de los tokens recibidos desde el analizador léxico.
    ```
    return new java_cup.runtime.Symbol(sym.igual);
    ```
- Crear una clase que aglutine las dos especificaciones, léxica y sintáctica (ver ejemplo en el anexo 2). En esta clase se creará una instancia del analizador léxico y se pasará como argumento al analizador sintáctico, de tal forma que éste le vaya proporcionando los tokens leídos desde el fichero de entrada.

A modo de prueba, se puede usar una gramática que reconozca cualquier ristra de tokens devueltos por el analizador léxico, sin estructura sintáctica alguna:

S::= S T | T
T ::= $token_1$ | $token_2$ | … | $token_n$



Reconocer conceptos de analizadores LR en los analizadores creados por CUP

CUP genera analizadores LR, en concreto LALR. CUP muestra los cálculos realizados para la generación del analizador con la opción "**`-dump`**".

El ejercicio consiste en usar las especificaciones del anexo 3 para crear el analizador utilizando la opción "**`-dump`**". Identificar en el resultado producido por CUP los pasos y elementos utilizados en la generación de analizadores LR: items, estados, transiciones y reducciones.

Anexo 1: modificaciones sobre el generador de analizador léxico ya creado con JFLEX para adaptarlo a CUP

1. Como se empleará la clase Symbol para pasar tokens desde el analizador léxico al analizador sintáctico generado por CUP, debe incluirse al comienzo de la especificación léxica la importación de los paquetes necesarios:
   ```
   import java_cup.runtime.*;
   ```
2. Debemos habilitar la compatibilidad de JFLEX con CUP. Por ello, dentro de la especificación léxica habilitamos la opción "cup":
   ```
   %cup
   ```
3. En el apartado de reglas léxicas, para cada tipo de token debemos introducir la devolución de su valor y tipo, con el fin de transmitirlo al analizador sintáctico. En el siguiente ejemplo se ve, para dos tipos de token, cómo serían antes de adaptar la especificación:

   ```
   {id} { System.out.println("Token ID="+ yytext()); }
   "==" { System.out.println("Token COMP_IGUALDAD"); }
   ```

   y después de la adaptación:

   ```
   {id} { return new Symbol(sym.id , yytext()); }
   "==" { return new Symbol(sym.comp_igualdad); }
   ```

Conviene recordar que JFlex tiene varias opciones que puede ser conveniente activar:

- %line: permite acceder al número de fila de los tokens
- %column: permite acceder al número de columna de los tokens
- %class AnalizadorLexico: cambia el nombre de la clase resultante "Ylex" por el nombre "AnalizadorLexico" (se generará, por tanto, la clase "AnalizadorLexico.java" y no "Ylex.java").

Anexo 2: clase que lanza la ejecución (Analizador.java)

Esta clase contiene una instancia del analizador léxico que se ha generado con JFlex (remarcada en amarillo). Es imprescindible que el nombre de la clase del analizador léxico que aparece en esta clase sea idéntico al nombre de la clase Java generada por JFlex.

```
import java.io.IOException;
import java.io.PrintWriter;

/**
*/
public class Analizador{

public static void main(String argv[])
{
	if (argv.length == 0) {
		System.out.println("Inserta nombre de archivo\n"+
		"( Usage : java Analizador <inputfile> )");
	} else {
	for (int i = 0; i < argv.length; i++) {
		AnalizadorLexico lexico = null;
		try {
			lexico =
				new AnalizadorLexico( new java.io.FileReader(argv[i]));
			parser sintactico = new parser(lexico);
			sintactico.parse();
		}
		catch (java.io.FileNotFoundException e) {
			System.out.println("Archivo \""+argv[i]+"\" no encontrado.");
		}
		catch (java.io.IOException e) {
			System.out.println("Error durante la lectura del"
			+ " archivo \""+argv[i]+"\".");
			e.printStackTrace();
		}
		catch (Exception e) {
			System.out.println("Excepcion:");
			e.printStackTrace();
		}
	}
	}
}
}
```

Anexo 3: Ejemplos de especificaciones léxicas y sintácticas

Anexo 3.A: Ejemplo de especificación sintáctica

Gramática	Especificación sintáctica en CUP

S ::= E = E
  | ident
E ::= E + T
  | T
T ::= T * ident
  | ident
	import java_cup.runtime.*;
	
	terminal ident, op_mas, op_mul, igual;
	non terminal E, T, S;
	
	S::= E igual E | ident;
	E::= T | E op_mas T;
	T::= ident | T op_mul ident;

Anexo 3.B: Ejemplo de especificación léxica

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

%class AnalizadorLexico
%unicode
%cup

%%
"="        {return new java_cup.runtime.Symbol(sym.igual);}
"+"        {return new java_cup.runtime.Symbol(sym.op_mas);}
"*"        {return new java_cup.runtime.Symbol(sym.op_mul);}
[a-zA-Z]+ {return new java_cup.runtime.Symbol(sym.ident);}

/*errorfallback*/
.|\n       {;}
```