**ANALIZADOR LÉXICO Y SINTÁCTICO**

Diego Jiménez Fernández-Pacheco GII + GIS Asier Ruano Peñas   GII + GIS

Índice

[Introducción 2](#_Toc479104947)

[Analizador léxico con JFlex 2](#_Toc479104948)

[Analizador sintáctico con Cup 3](#_Toc479104949)

[Conexión entre el Analizador Léxico y el Analizador Sintáctico 4](#_Toc479104950)

[Casos de Prueba 4](#_Toc479104951)

[Corrección de errores con soluciones propuestas 4](#_Toc479104952)

## Introducción

En esta memoria se recogen los aspectos y cuestiones que se consideran más relevantes para la comprensión del trabajo realizado, consistente en las especificaciones léxica y sintáctica para el diseño e implementación de un visualizador de código de un lenguaje de programación similar a C.

En esta primera entrega se incluyen únicamente las partes de código que analizan los elementos del lenguaje (analizador léxico) y su correcta combinación (analizador sintáctico).

La presente memoria se divide en dos partes: una en la que explicaremos el desarrollo del analizador léxico y otra en la que se explicará el del analizador sintáctico. En un tercer apartado se explicará la unión de los dos analizadores para conseguir su colaboración. En un apartado final, se explicará cómo se ha abordado la gestión de errores y se incluirán los casos de prueba utilizados para comprobar el correcto comportamiento del analizador.

Los lenguajes y herramientas que se han empleado para este trabajo han sido JFlex en la parte del analizador léxico y Cup en la parte del analizador sintáctico. Con estas dos herramientas se han construido las expresiones regulares que indican al analizador léxico qué *tokens* debe reconocer y la gramática que debe ser reconocida por el programa.

Todo el trabajo se encuentra en un repositorio de GitHub. Este repositorio es público, por lo que si el profesor desea tener acceso al mismo le enviaremos el enlace.

## Analizador léxico con JFlex

Para poder reconocer los lexemas que cumplían con los patrones de nuestro lenguaje de programación similar a C y generar los *tokens* que se le pasarían posteriormente al analizador sintáctico, necesitábamos generar una especificación léxica a partir de la cual, mediante expresiones regulares, nuestro programa entendiese que una determinada cadena de entrada se correspondía a un determinado patrón.

Para ello nos valimos de la herramienta JFlex, que nos permite generar el autómata que reconocerá nuestros *tokens* a partir de las expresiones regulares apropiadas.

Así, basándonos en el enunciado de la práctica, diseñamos en un fichero *.flex* las expresiones regulares que darían lugar al autómata deseado.

Mediante esta construcción léxica conseguimos que nuestro analizador genere todos los *tokens* correspondientes al programa para que puedan ser posteriormente analizados por el analizador sintáctico.

Se ha decidido poner en primer lugar los tokens de palabras reservadas y símbolos ya que en caso contrario detectaba como identificador una palabra reservada.

Se han añadido en los terminales sentencias del tipo debug para poder mostrar la ejecución del programa cuando se encuentra algún error.

## Analizador sintáctico con Cup

El analizador sintáctico debe comprobar que los elementos reconocidos por el léxico están correctamente combinados.

Para la realización del analizador sintáctico y conseguir la función buscada, hicimos uso de la herramienta Cup. Esta herramienta construye un autómata a pila basándose en la gramática de nuestro lenguaje a reconocer. Así, lo que hicimos para llevar a cabo esta parte de la práctica fue especificar en un archivo .cup la gramática que debían cumplir los diferentes elementos del programa para ser correctos.

Al principio tuvimos algunos problemas con la recursividad en las gramáticas que hacían que el programa no se ejecutara de forma correcta.

La solución fue sencilla: Cup cuenta con una directiva que permite precisamente indicar precedencia entre los símbolos de la gramática (*precedence left [lista de símbolos]*) de modo que el analizador ya podía generar correctamente la tabla de símbolos de la gramática.

Además, las gramáticas de la parte opcional han sido simplificadas, para evitar repeticiones y producciones unitarias.

## Conexión entre el Analizador Léxico y el Analizador Sintáctico

Una vez generados ambos analizadores, debíamos conseguir que se comunicasen entre sí para poder colaborar.

Existen varias formas de hacer comunicarse estos dos componentes, pero nosotros, siguiendo con las pautas marcadas en clase, elegimos que el analizador léxico, al detectar un token, devolviese un elemento de la clase *symbol* (la cual es generada por cup, igual que la clase *parser*) para que de ésta manera pudiese ser utilizada por la clase *parser*. Creamos una clase principal en la cual creamos una instancia del analizador léxico, otra del parser y las unimos de tal manera que al hacer la llamada al método *parser.parse* se procesa todo el archivo que hayamos puesto.

//elegimos que el analizador léxico fuese una subrutina del analizador sintáctico, de manera que cada vez que se procesa un *token*, el analizador sintáctico llama al léxico mediante *nextToken()* para procesar el siguiente.

Esto se consigue mediante una clase en java que nosotros hemos denominado *Parser,* que se encarga precisamente de conectar ambos analizadores.

El programa básicamente lee de un fichero de texto las diferentes cadenas de entrada, se las pasa al analizador léxico para que genere un token o, en su defecto, detecte los errores y posteriormente hace que el sintáctico se inicie y compruebe la combinación de los tokens de una línea, llamando al léxico cuando sea necesario.

## Casos de Prueba

Para las pruebas se han utilizado los ficheros CorrectoSin.txt / FalloSin.txt para pruebas de sintáctico y CorrectoLex.txt / FalloLex.txt para las pruebas de léxico.

Casos de prueba correctos:

## Corrección de errores con soluciones propuestas

Léxico

Cuando entra un número con formato octal pero con un nueve, lo transforma en un número de base decimal.

Cuando aparecen dos operadores seguidos (++, --) devuelve el operador y el tipo de número; ejemplo ++255 devuelve +<decint>.

Sintáctico

Si al ejecutar el código se encuentra un error, el programa muestra por pantalla la secuencia de terminales aceptados y en qué terminal está el error que hace fallar la aplicación.   
Se ha añadido un control de excepciones para que muestre por pantalla el tipo de error.