# MEMORIA PRÁCTICA ANALIZADOR LÉXICO

Grupo 54:

* Marcos Carnerero Blanco
* Rodrigo Elola Torrijos
* Miguel López García

# Índice

* Diseño del Analizador Léxico…………………………………………………2
* Código del Analizador Léxico…………………………………………………4
* Código de la Tabla de Símbolos……………………………………………..5

# Diseño del Analizador Léxico

Para esta práctica hemos implementado los siguientes Tokens:

<EOF,-> End Of File <INPUT,-> Input

<VAR,-> Var <AUTOINCREMENTO,-> ++

<BOOLEAN,-> Boolean <INTEGER,-> Integer

<VOID,-> Void <COMA,-> ,

<BREAK,-> Break <OUTPUT,-> Output

<CteENTERA, Numero> Constante entera <DosPUNTOS,-> :

<FUNCTION,-> Function <STRING,-> String

<ID,Numero> Identificador <EOS,-> ;

<CASE,-> Case <RETURN,-> Return

<CteCADENA,-> Cadena “ <ParentesisABRE,-> (

<ParentesisCIERRA,-> ) <IF,-> if

<SWITCH,-> Switch <ASIG,-> Asignación

<LlaveABRE,-> { <LlaveCIERRA,-> }

<NOT,-> ! <MAYOR,-> >

<MULT,-> Multiplicación \* <DEFAULT,-> Default

<FALSE,-> False <TRUE,-> True

Tras decidir el diseño de los Tokens anteriormente citados, hemos procedido a crear la gramática de la siguiente manera:

S 🡪 del S | /A | lB | dC | “D | +E | ( | ) | { | } | , | ; | : | ! | = | > | ‘ | \* | EOF

A 🡪 \*A’ l: letra

A’ 🡪 c1A’ | \*A’’ d: dígito

A’’ 🡪 /S | c2A’ | \*A’’ c1: Cualquier carácter exceptuando \*

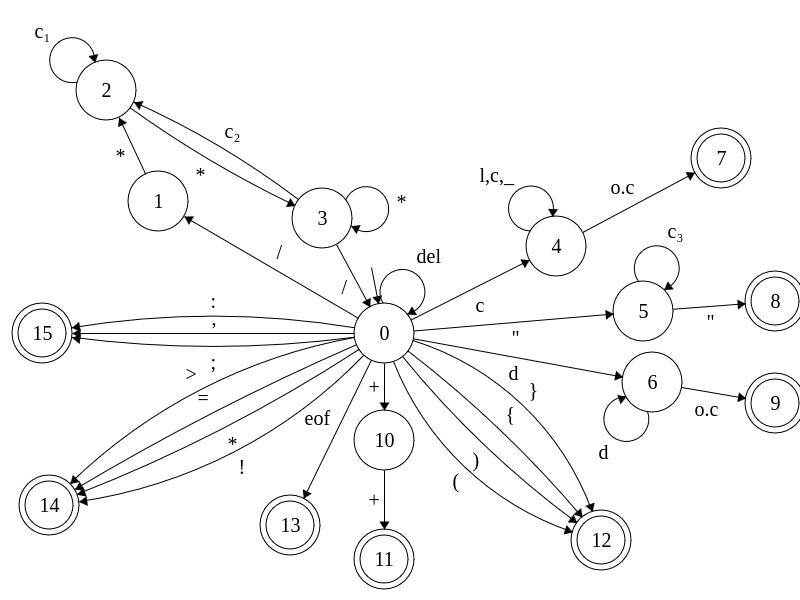
B 🡪 lB | dB | \_B | λ c2: Cualquier carácter menos \* o /

C 🡪 dC | λ c3: Cualquier carácter menos “

D 🡪 c3D | “

E 🡪 +

Con la gramática anterior, construimos el siguiente Autómata Finito Determinista:



Como se puede apreciar, muchos caracteres podrían transitar hacia el mismo estado final, pero los hemos separado para favorecer la limpieza, claridad y comprensión del autómata.

También destacamos las siguientes acciones semánticas:

* Leer🡪 L car := leer() //en todas menos 6:7, 10:11
* Concatenar 🡪 C lex := car //0:6

C’ lex := lex + car //6:6, 8:8

C\* lex := “ //0:8

* Número 🡪 N num := valor (car)

N’ num := num \* 10 + valor

* Generar tokens 🡪 Gen\_token (CteENTERA,num)

Gen\_token (CteCADENA, lex)

# CÓDIGO DEL ANALIZADOR LÉXICO (LEXER)

En cuanto a la programación del Analizador, el lenguaje elegido ha sido Java. A continuación, se listarán las funciones usadas en la clase Lexer.java y los atributos de esa misma clase y se explicarán brevemente:

Atributos:

* **private BufferedReader reader** : lector que nos permitirá identificar un carácter.
* **private TSHandler** **tsHandler**: almacena el gestor de la tabla de símbolos.
* **private int lineCount**: se encarga de guardar el número de la línea en la que se está trabajando.
* **private int c**: almacena el número, en la tabla ASCII, del carácter que se está leyendo.

Funciones:

* **public Lexer** : es el constructor de la clase Lexer, recibe el nombre del archivo y el creador de la tabla de símbolos y lanza las excepciones **FileNotFoundException** e **IOException.** Recibe un string  **fileName** (que contiene un nombre de archivo) y un TSHandler **tsHandler** (el gestor de la tabla de archivos sobre la que vamos a trabajar). Se encarga de inicializar **reader** con el nombre de archivo **fileName**, el atributo de clase **tsHandler** con su homólogo recibido en la función, la variable **lineCount** a 1 ya que empezará a leer por la primera línea y **c** con el primer carácter leído. Puede lanzar las excepciones **IOException** o **LexerException**.
* **private int readSuperflous** :Se encargará de leer cualquier cosa en el archivo que no genere ningún token, tal como un comentario o espacios en blanco.
* **private Pair<Token, Object> words** : Es llamada si el **reader** a leído una letra y sigue leyendo hasta que encuentre cualquier carácter que no sea letra, digito o ‘\_’. También comprueba si la palabra encontrada es una de las reservadas. En caso afirmativo, devuelve el token de dicha palabra reservada, en caso contrario, inserta en el gestor de la tabla de símbolos la palabra leída y la línea en la que se encuentra. Puede lanzar las excepciones **IOException, LexerException** o **TSException**.
* **private Pair<Token, Object> cteEntera** : Es llamada si el **reader** lee un dígito en primera estancia y va a crear el token de la constante entera cuyo atributo va a ser el número completo y devuelve dicho token. Puede lanzar la excepción **IOException**.
* **private Pair<Token, Object> cteCadena** : Es llamada cuando el **reader** lee el carácter “. Se encarga de crear y devolver el token de la constante cadena cuyo atributo es la cadena completa y puede lanzar las excepciones **IOException** o **LexerException** si ocurre algún error en el proceso.
* **public Pair<Token, Object> scan** : La función principal de la clase **Lexer**, se encarga de llamar a las funciones anteriormente desarrolladas según lo que lea el **reader** (excepto **words()**). Si recibe **c**=-1 genera el token EOF (fin del archivo) y si no, devuelve el token devuelto por la función correspondiente o por cualquier otro carácter que genere un token tal como la asignación o el símbolo ‘>’. En caso de no leer nada de lo anterior, lanzará la excepción **LexerException**. También puede lanzar las excepciones **IOException** y **TSException.**

**NOTA**: esta función solo va a crear los tokens que han sido propuestos para esta práctica en concreto, los caracteres no escogidos generarán un error.

# CODIGO DE LA TABLA DE SIMBOLOS

Tras haber explicado la clase **Lexer**, explicaremos la clase **TSHandler** que se encargará de crear y gestionar la tabla de símbolos.

Atributos:

* **private ArrayList<Hashtable<Integer, String>> tsList** : Es una lista de tablas que nos servirá para distinguir las tablas de símbolos (la global y las diferentes locales).
* **private Integer lastPosTS** : Guarda la última posición de la tabla sobre la que estamos trabajando.
* **private int currentTS** : Guarda la posición en la **tsList** de la tabla con la que estamos trabajando.

Funciones:

* **public TSHandler** : Constructor de la clase **TSHandler**. Inicializa **tsList** y le añade una tabla que será nuestra Tabla de Símbolos global. Además, inicializa **lastPosTS** y **currentTS** a 0, ya que estaremos trabajando con la primera tabla de la lista en su última (y única) posición de momento.
* **public void openScope** : Crea una tabla de símbolos local y la añade a **tsList**, cambiando los valores de **lastPosTS** a 0 (última y única posición de la nueva tabla) y **currentTS** a +1 (al crear la tabla aseguramos que estábamos trabajando con la tabla anterior a ella en la lista y así indicamos que trabajaremos con esta ahora).
* **public void closeScope** : Borramos la tabla de símbolos actual y cambiamos los valores de **lastPosTS** al de la última posición de la tabla de símbolos anterior a esta en la lista y **currentTS** a -1 para indicar que la tabla actual es la anterior en la lista.
* **public Pair<Token, Object> insert** : recibe un string **id** y un int **line**. Crea un HashTableMap **actualTS** igual a la TS sobre la que estamos trabajando (mediante **currentTs**) y busca un Token que se llame igual que **id**. Por cada uno que encuentre, crea y devuelve otro Token de tipo ID con atributo igual al que tenía el Token inicial. En caso de que no lo encuentre, añade a la tabla un entry con valor igual a **lastPosTS** y clave igual a **id**, y después incrementa **lastPosTS**. Además, crea un Token de tipo ID cuyo atributo sea el valor de este entry. Puede lanzar la excepción **TSException**.
* **public void toFile** : Recibe un string **fileName**.Inicializa un PrintWriter **writer** con el que va escribiendo los contenidos de cada una de las tablas de forma ordenada, convirtiéndolo en un archivo de texto. Si algo falla durante el proceso, lanza la excepción **IOException**.