



ESCUELA TÉCNICA  
SUPERIOR DE INGENIEROS  
INFORMÁTICOS

UNIVERSIDAD POLITÉCNICA DE MADRID

---

## Procesadores de Lenguajes: Memoria del Proyecto

---

Jose Luis Prado Sierra - 220070  
Alejandro Gragera Serradilla - 22M043  
Antonio Bielza Díez - 22M049

## 1. Analizador Léxico

Durante el desarrollo del Analizador Léxico hemos descrito una serie de objetos y estructuras matemáticas indispensables para que su diseño, desarrollo y función final cumplan con las expectativas de un Procesador de Lenguajes.

### 1.1. Tokens

Los tokens son duplas que se generan cuando el Analizador Léxico encuentra una concatenación de caracteres que identifica como válida.

Contienen la información necesaria para que las reciba el Analizador Sintáctico y se componen de un código que los identifica y un atributo opcional que puede servir para diferenciarlos de otros tokens con el mismo código o aportar información extra.

Hemos definido los siguientes tokens en función de las necesidades de nuestra práctica:

- Boolean: <BOOLEAN, >
- Else: <ELSE, >
- Float: <FLOAT, >
- Function: <FUNCTION, >
- If: <IF, >
- Int: <INT, >
- Let: <LET, >
- Read: <READ, >
- Return: <RETURN, >
- String: <STRING, >
- Void: <VOID, >
- Write: <WRITE, >
- Constante real: <REALCONST, >
- Constante entera: <INTCONST, >
- Cadena: <STR, ç\*”>
- Identificador: <ID, posTS>
- Suma con asignación (+=): <PLUSEQ, >
- Igual (=): <EQ, >
- Coma (,): <COMMA, >
- Punto y coma (;): <SEMICOLON, >
- Paréntesis abierto (): <OPPAR, >
- Paréntesis cerrado (): <CLPAR, >
- Llave abierta ({): <OPBRA, >

- Llave cerrada (}): <CLBRA, >
- Suma (+): <SUM, >
- Y Lógico (&&): <AND, >
- Menor (<): iMINORTHAN, >
- false: <FALSE, >
- true: <TRUE, >
- EOF: <EOF, >

Todos los tokens anteriores conforman todos los obligatorios, los específicos y los opcionales de la práctica.

## 1.2. Gramática

La gramática es la estructura matemática que define el lenguaje a generar, en nuestro caso es para una versión reducida de JS. Definir la gramática es una forma de estructurar por tanto el lenguaje y asegurarnos de que vamos a generar solo lo que queremos.

Hemos definido las siguientes reglas:

$$\begin{aligned}
 S &\rightarrow delS \mid +A \mid \&B \mid 'C \mid dD \mid lF \mid _F \mid /G \mid = \mid , \mid ; \mid < \mid ( \mid ) \mid \{ \mid \} \\
 A &\rightarrow = \mid \lambda \\
 B &\rightarrow \& \\
 C &\rightarrow cC \mid ' \\
 D &\rightarrow dD \mid .E \mid \lambda \\
 E &\rightarrow dE \mid d \\
 F &\rightarrow lF \mid dF \mid _F \mid \lambda \\
 G &\rightarrow /H \\
 H &\rightarrow c_2H \mid \backslash nS
 \end{aligned}$$

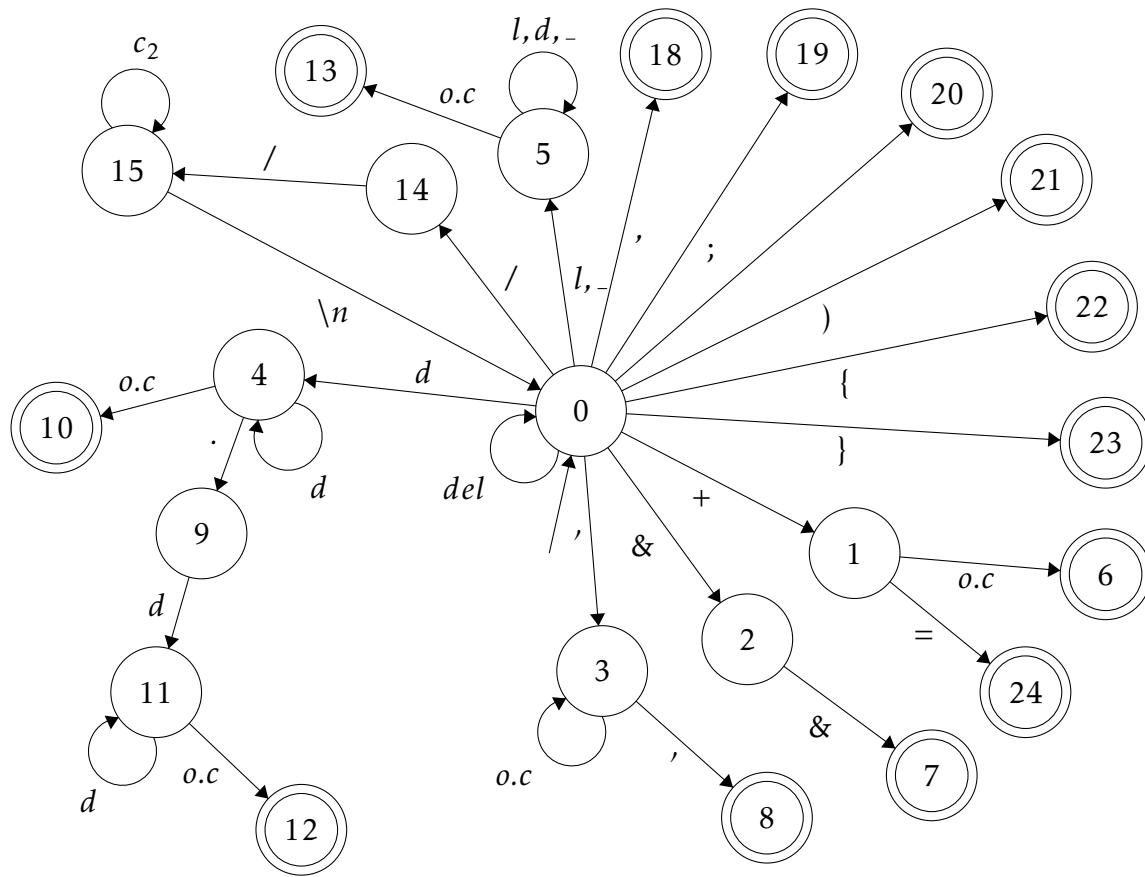
Los conjuntos definidos para el desarrollo de la gramática y del resto de la práctica son:

- Conjunto l: representa cualquier letra.
- Conjunto d: representa cualquier dígito.
- Conjunto c: representa cualquier carácter.
- Conjunto c<sub>2</sub>: representa cualquier carácter sin \n.

### 1.3. Autómata

Un autómata es otro tipo de estructura matemática capaz de, al contrario que una gramática que se centra en generar un lenguaje, un autómata se encarga de comprenderlo.

El autómata que hemos planteado capaz de entender todas las palabras válidas de nuestro lenguaje es el siguiente:



El autómata tiene que ir acompañado de una serie de acciones que realizar mientras se recorre para poder generar los tokens mientras vamos leyendo el archivo, que vamos a plantear en el siguiente punto.

Cuando aparece una transición o.c significa que es cualquier otro carácter distinto a las demás transiciones salientes del vértice.

## 1.4. Acciones Semánticas

Las acciones semánticas son una serie de acciones que se realizan entre las transiciones del autómata que permiten realizar diferentes funciones como por ejemplo ir generando un string para cuando detectamos una cadena.

Las acciones semánticas que hemos definido para nuestro proyecto son:

0:0 Leer  
0:18 Leer, gentoken(<COMMA, >)  
0:19 Leer, gentoken(<SEMICOLON, >)  
0:20 Leer, gentoken(<OPPAR, >)  
0:21 Leer, gentoken(<CLPAR, >)  
0:22 Leer, gentoken(<OPBRA, >)  
0:23 Leer, gentoken(<CLBRA, >)  
0:1 Leer  
1:6 gentoken(<SUM, >)  
1:24 Leer, gentoken(<EQ, >)  
0:2 Leer  
2:7 Leer, gentoken(<AND, >)  
0:3 Leer, *str* =''  
3:3 Leer, *str* = *str* + *c*  
3:8 Leer, gentoken(<STRING, str>)  
0:4 Leer, *num* = *d*  
4:4 Leer, *num* = *num* \* 10 + *d*  
4:10 gentoken(<INTCONST, num>)  
4:9 Leer, *cont* = 1  
9:11 Leer, *num* = *num* \* 10 + *d*, *cont* ++  
11:11 Leer, *num* = *num* \* 10 + *d*, *cont* ++  
11:12 *num* = *num* \*  $10^{-cont}$ , gentoken(<REALCONST, num>)  
0:14 Leer  
0:15 Leer  
15:15 Leer  
15:0 Leer

```
0:5 Leer, id = c
5:5 Leer, id = id + c
5:13

if(id in palabrasReservadas)
    posTS = insertTS(id)
    gentoken(<toUpperCase(id), posTS>)
else
    posTS = insertTS(id)
    gentoken(<ID, posTS>)
```

## 1.5. Errores

Todo buen procesador de lenguajes debe de avisar al usuario de si existe algún error y además se capaz de tratar todos los posibles errores para una ejecución robusta y comprensible.

Hemos detectado los siguientes casos capaces de generar algún error y que por tanto tratamos en nuestro código:



**Figura 1:** A goose.

## **2. Analizador Sintáctico**

### **3. Analizador Semántico**