

# Práctica 1 - Procesadores de Lenguajes

## Grupo 7

Carmen Toribio Pérez, 22M009

Sergio Gil Atienza, 22M046

María Moronta Carrión, 22M111

## 1 Introducción

La primera entrega de esta práctica consiste en la implementación de un Analizador Léxico y una Tabla de Símbolos. Para ello, hemos seguido los siguientes pasos:

1. **Identificar los Tokens** del lenguaje fuente, teniendo en cuenta las especificaciones de nuestro grupo.
2. Construir la **Gramática Regular** que los genera.
3. Diseñar el **Autómata Finito Determinista** equivalente a la gramática. Hemos realizado una representación de este a través de un **diagrama de estados**.
4. Añadir las **Acciones Semánticas**, asociadas a cada una de las transiciones del diagrama de estados.
5. Estudiar los posibles **casos de error**, para poder manejar correctamente la detección y el reporte de errores léxicos.

Por otro lado, hemos comenzado con la implementación de la **Tabla de Símbolos**, concretamente con el diseño de su estructura y su organización.

Finalmente, presentamos una serie de casos de prueba que demuestran el correcto funcionamiento del Procesador, así como su capacidad para manejar errores léxicos.

Como integrantes del **grupo 7**, hemos tenido que cumplir con las siguientes especificaciones:

- Sentencias: Sentencia repetitiva (for)
- Operadores especiales: Asignación con suma (+=)
- Técnicas de Análisis Sintáctico: Descendente Recursivo
- Comentarios: Comentario de bloque (/ \* \*/)
- Cadenas: Con comillas simples ( ' ' )

Hemos decidido usar C++ como lenguaje de programación porque la mayor parte de la infraestructura de compiladores está escrita en C o en C++, incluyendo el proyecto LLVM (Low Level Virtual Machine). También hemos tenido en cuenta su flexibilidad y potencia, junto a la amplia variedad de utilidades que tiene su librería estándar. En comparación con lenguajes como Java o JavaScript, C++ ofrece mayor eficiencia y control sobre los recursos del sistema, algo crucial en proyectos como este (que requieren optimización de bajo nivel). Además, consideramos este proyecto como una gran oportunidad para practicar y aprender un lenguaje completamente distinto a los que estamos acostumbrados.

## 2 Tokens

El primer paso a la hora de construir un Analizador Léxico es la identificación de los tokens. Para hacer la lista nos hemos basado en la actividad práctica de la plataforma Draco. Hemos decidido utilizar el mismo formato en tablas con tal de facilitar su legibilidad.

Tokens obligatorios:

Elemento	Código de Token	Atributo
boolean	bool	-
for	for	-
function	fn	-
if	if	-
input	in	-
int	int	-
output	out	-
return	ret	-
string	str	-
var	var	-
void	void	-
constante entera	cint	Número
Cadena (')	cstr	Cadena ("c*")
Identificador	id	Número (posición en la TS)
+=	cumass	-
=	ass	-
,	com	-
;	scol	-
(	po	-
)	pc	-
{	cbo	-
}	cbc	-

Tokens de operadores aritméticos, lógicos y relacionales:

Grupo de Opciones	Código de Token	Atributo
Grupo Operadores Aritméticos: Suma (+)	sum	-
Grupo Operadores Aritméticos: Resta (-)	sub	-
Grupo Operadores Lógicos: Y lógico (&&)	and	-
Grupo Operadores Lógicos: O lógico (  )	or	-
Grupo Operadores Relacionales: Menor (<)	ls	-
Grupo Operadores Relacionales: Mayor (>)	gr	-

Tokens opcionales:

Grupo de Opciones	Código de Token	Atributo
Menos Unario (-)	sub	-
Más Unario (+)	sum	-
false	cap	-
true	nocap	-
EOF	eof	-

Por tanto, los siguientes tipos de expresiones no serán identificados como tokens: los delimitadores (como los espacios en blanco o las tabulaciones), los comentarios de bloque (/ \* \*/) o los saltos de línea (\n).

### 3 Gramática Regular

En esta sección, describimos la Gramática Regular (gramática de tipo 3 según la jerarquía de Chomsky) que hemos diseñado para identificar y generar los tokens del lenguaje fuente.

#### Símbolos no terminales

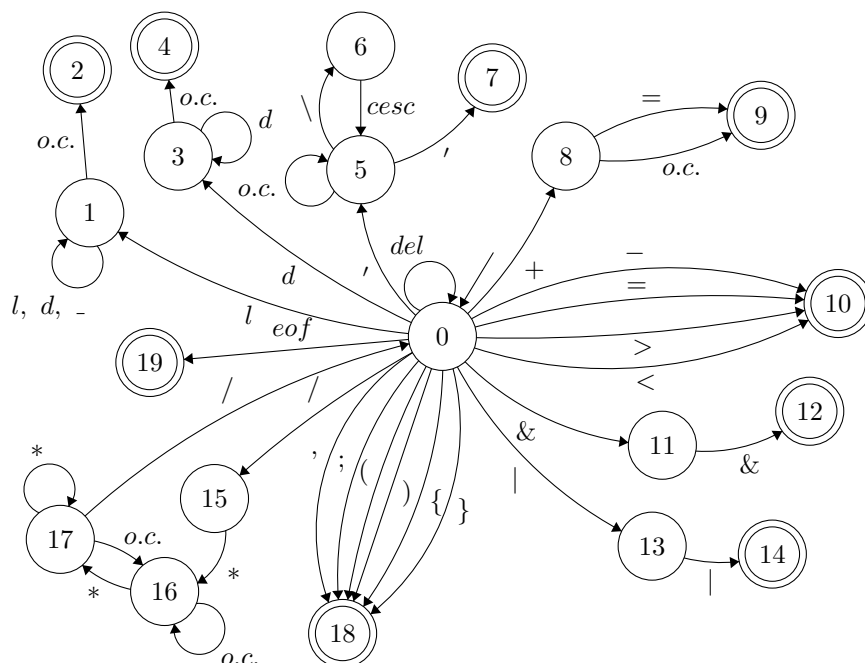
$d := 0...9$   
 $l := a..z, A...Z$   
 $c_1 :=$  espacio o cualquier carácter imprimible menos \  
 $c_{esc} :=$  carácter escapable (' , 0, n, a, t, v, f, r, \  
 $c_2 :=$  cualquier carácter menos \* y eof  
 $c_3 :=$  cualquier carácter menos \*, / y eof

#### Gramática Regular

$S \rightarrow \text{del } S \mid lA \mid dB \mid 'C \mid +D \mid - \mid = \mid > \mid < \mid \&E \mid IF \mid /G \mid \} \mid \{ \mid ) \mid ( \mid ; \mid , \mid \text{eof}$   
 $A \rightarrow lA \mid dA \mid \_A \mid \lambda$   
 $B \rightarrow dB \mid \lambda$   
 $C \rightarrow c_1C \mid \backslash C' \mid '$   
 $C' \rightarrow c_{esc}C$   
 $D \rightarrow = \mid \lambda$   
 $E \rightarrow \&$   
 $F \rightarrow \mid$   
 $G \rightarrow *H$   
 $H \rightarrow c_2H \mid *I$   
 $I \rightarrow /S \mid c_3H \mid *I$

### 4 Autómata Finito Determinista

Una vez definida la gramática regular, el siguiente paso es construir el Autómata Finito Determinista (AFD) correspondiente. A continuación, presentamos su diseño, incluyendo las transiciones entre estados.



## 5 Acciones semánticas

A lo largo de esta sección, describimos las acciones semánticas que hemos añadido a las transiciones del Automata Finito Determinista. Estas acciones permiten que se lleven a cabo acciones como la lectura, la identificación de los tokens o, en su lugar, una correcta gestión de los errores léxicos. A continuación, detallamos las operaciones utilizadas y, por legibilidad en el AFD, las transiciones en las que se realizan:

### Leer

```
car := leer() // Todas las transiciones salvo 0:19, 1:2, 3:4 y 8:9 con un caracter
distinto de '='
```

### Concatenar

```
lex := car // Transición 0:1
lex := ∅ // Transición 0:5
lex := lex ⊕ car // Transición 1:1
```

### Contador

```
cont := 0 // Transición 0:5
cont := cont + 1; // Transiciones 5:5, 6:5
```

### Valor número

```
num := val(car) // Transición 0:3
num := num * 10 + val(car) // Transición 3:3
```

### Generar token

```
// Cadenas y enteros
G1 if (len(lex) > 64) then error (COD_ERROR_STRLEN, lex)
   else Gen_token(cstr, lex) // Transición 1:2
G2 Gen_token(cint, num) // Transición 3:4

// Delimitadores y operadores de control
G3 Gen_token(cumass, -) // Transición 8:9
G4 Gen_token(ass, -) // Transición 8:9
G5 Gen_token(com, -) // Transición 0:18
G6 Gen_token(scol, -) // Transición 0:18
G7 Gen_token(po, -) // Transición 0:18
G8 Gen_token(pc, -) // Transición 0:18
G9 Gen_token(cbo, -) // Transición 0:18
G10 Gen_token(cbc, -) // Transición 0:18

// Operadores Aritméticos, Lógicos y Relacionales
G11 Gen_token(sum, -) // Transición 0:10
G13 Gen_token(sub, -) // Transición 0:10
G16 Gen_token(and, -) // Transición 0:10
G17 Gen_token(or, -) // Transición 0:10
G19 Gen_token(ls, -) // Transición 0:10
G21 Gen_token(gr, -) // Transición 0:10

//EOF
G25 Gen_token(EOF,-) // Transición 0:19

// Palabras reservadas // Transición 1:2
G26 type := keywordToToken(lex) //Función 1, explicada al final
if type == IDENTIFIER then {

    pos := SearchSymbol(lex) //Función 2, explicada al final

    if pos == NULL then pos := AddSymbol(lex)
```

```
Gen_token(type, pos)

} else Gen_token(type, -)
```

Funciones: //Esto lo pondré más bonito, explicando las funciones que se usan

1. keywordToToken devuelve el código de token de la palabra reservada con la que coincida lex, o el código de ID si no lo es
2. SearchSymbol ...

## 6 Gestión de errores

Explicando los casos de error posibles, los códigos de error asignados a cada caso, etc.

## 7 Tabla de Símbolos

Solo tiene identificadores y su posición, de momento. "muchotexto mimimi".

## 8 Funcionamiento del programa

Breve explicación de lo que hace nuestro programa.

## 9 Carmen nini

¿La de las tildes te la sabes? Jejéjèjè.

## 10 Sergio nini

Sí me la sé :( ¿La de venir a clase te la sabes? Ñejejejejeje.  
PUAJAJSJAJAJA se lo dijo