

October 24, 2025

Procesador de MyJS: *jsp*

Memoria del Grupo 59 (Primera Entrega)

Por Andrés Súnico (23M018)

Índice

1. Introducción.....	1
2. Información Adicional	2
3. Opciones de la práctica.....	2
4. Diseño del Lexer.....	2
5. Diseño de la Tabla de Símbolos	5
6. Diseño del Parser	5
7. Diseño del Semanter.....	5
8. Gestor de Errores	5
A. Casos de Prueba	6

1 Introducción

El desarrollo del procesador *jsp* se ha centrado en la experiencia del usuario (*UX*), priorizando tres aspectos clave: una gestión de errores sólida y clara, una interfaz de línea de comandos (*CLI*) intuitiva, y un rendimiento eficiente.

Por ello, se ha elegido *Rust* como el lenguaje de desarrollo. Ofrece una gestión de memoria eficiente, además de integrar *clap*, una de las mejores librerías para desarrollar aplicaciones *CLI*.

Gracias al uso del patrón de *inyección de dependencias* en todo el proyecto, el código fuente es altamente extensible y modular.

2 Información Adicional

El código fuente del procesador se puede encontrar en github.com/suuniquo, así como los tests y las dependencias del proyecto.

3 Opciones de la práctica

Además de las opciones comunes a todos los grupos, se han implementado las opciones:

3.1 Específicas del grupo

- Comentarios de bloque (`/* */`)
- Cadenas con comillas dobles (`" "`)
- Sentencia repetitiva `do-while`
- Asignación con `y` lógico (`&=`)
- Análisis Sintáctico Ascendente

3.2 Adicionales

Para que el procesador esté más completo, se han implementado adicionalmente todos los operadores lógicos, aritméticos, relacionales y unarios, así como los *tokens* booleanos *true* y *false*.

Además se ha escogido implementar el tratamiento de secuencias de escape (`\n` y `|t`).

4 Diseño del Lexer

El Analizador Léxico o *Lexer* es uno de los 3 módulos principales del procesador.

Al ser la primera capa de procesamiento, es el encargado de manejar el fichero fuente y convertirlo en una lista de *tokens* para el Analizador Sintáctico.

4.1 Tokens

Con el fin de lograr un procesamiento eficiente, tanto en memoria como en complejidad, se han minimizado el número de *tokens* con atributos.

De este modo sólo 4 de un total de 41 *tokens* van a utilizar un atributo.

Cabe notar, además, que se ha decidido no hacer uso del *token* fin de fichero (*EOF*). Esto es porque el *Lexer* se ha implementado como un iterador de *tokens*, de modo que el final del flujo se detecta naturalmente cuando se consume el iterador.

Table 1: Listado de *tokens*

Elemento	Código	Atributo
boolean	Bool	-
do	Do	-
float	Float	-
function	Func	-
if	If	-
int	Int	-
let	Let	-
read	Read	-
return	Ret	-
string	Str	-
void	Void	-
while	While	-
write	Write	-
constante real	FloatLit	Número
constante entera	IntLit	Número
Cadena	StrLit	Cadena
Identificador	Id	Posición
<code>&=</code>	AndAssign	-
<code>=</code>	Assign	-
<code>,</code>	Comma	-
<code>;</code>	Semi	-
<code>(</code>	LParen	-
<code>)</code>	RParen	-
<code>{</code>	LBrack	-
<code>}</code>	RBrack	-
Suma (<code>+</code>)	Sum	-
Por (<code>*</code>)	Mul	-
Resta (<code>-</code>)	Sub	-
División (<code>/</code>)	Div	-
Módulo (<code>%</code>)	Mod	-
<code>Y</code> lógico (<code>&&</code>)	And	-
<code>O</code> lógico (<code> </code>)	Or	-
Negación (<code>!</code>)	Not	-
Menor (<code><</code>)	Lt	-
Menor o igual (<code><=</code>)	Le	-
Mayor (<code>></code>)	Gt	-
Mayor o igual (<code>>=</code>)	Ge	-
Relacionales: Distinto (<code>!=</code>)	Ne	-
Igual (<code>==</code>)	Eq	-
Menos Unario (<code>-</code>)	Sub	-
Más Unario (<code>+</code>)	Sum	-
false	False	-
true	True	-

4.2 Errores

El diseño de los errores y del gestor está orientado a la UX. Cada tipo de error consta de un mensaje diferente y de una severidad, distinguiéndose *errors* de *warnings* (que no impedirían la compilación del programa).

El procesador genera mensajes claros con número de línea y columna, muestra la línea afectada y subraya en color (que depende de la severidad) la parte errónea.

El *Lexer* sólo genera un warning, *Invalid Escape Sequence*. Como se muestra en Acciones Semánticas, al detectar una secuencia de escape inválida no se descartara el *token* cadena, sino que se conserva literalmente (por ejemplo, la secuencia `|q`, se sustituye por esos dos mismos caracteres).

Table 2: Listado de errores del *Lexer*

Error	Severidad
Carácter inválido	<i>error</i>
Comentario inacabado	<i>error</i>
Cadena inacabada	<i>error</i>
Overflow de Cadena	<i>error</i>
Overflow de Entero	<i>error</i>
Overflow de Real	<i>error</i>
Formato de Real Inválido	<i>error</i>
Secuencia de Escape Inválida	<i>warning</i>

4.3 Gramática

Se define la gramática del *Lexer* como $G = (T, N, S, P)$, dónde:

$T = \{\text{Todo carácter ASCII}\} \cup \{EOF\}$

$N = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$

P se compone de las reglas:

$$\begin{aligned}
 S &\rightarrow \text{del}S \mid , \mid ; \mid (\mid) \mid \{ \mid \} \mid + \mid * \mid \% \mid = A \mid !B \mid < C \mid > D \mid \&E \mid |F \mid dG \mid "H \mid c_1I \mid /J \mid EOF \\
 A &\rightarrow = \mid \lambda \\
 B &\rightarrow = \mid \lambda \\
 C &\rightarrow = \mid \lambda \\
 D &\rightarrow = \mid \lambda \\
 E &\rightarrow = \mid \lambda \\
 F &\rightarrow \mid \\
 G &\rightarrow dG \mid .K \mid \lambda \\
 K &\rightarrow dK \mid \lambda \\
 H &\rightarrow c_2H \mid \backslash L \mid " \\
 L &\rightarrow nH \mid tH \\
 I &\rightarrow c_3I \mid \lambda \\
 J &\rightarrow *M \mid \lambda \\
 M &\rightarrow c_4M \mid *N \\
 N &\rightarrow c_5M \mid *N \mid /S
 \end{aligned}$$

Y se definen:

$$\begin{aligned}
 d &:= \{0, 1, \dots, 9\} \\
 l &:= \{a, b, \dots, z, A, B, \dots, Z\} \\
 c_1 &:= l \cup \{_ \} \\
 c_2 &:= T \setminus (\{ \backslash, ", EOF \} \cup \{\text{Todos los caracteres ASCII no gráficos (Es decir, con código menor que 32)}\}) \\
 c_3 &:= c_1 \cup d \\
 c_4 &:= T \setminus \{*, EOF\} \\
 c_5 &:= T \setminus \{*, /, EOF\}
 \end{aligned}$$

4.4 Autómata

A continuación se muestra el autómata finito determinista que reconoce el lenguaje generado por la gramática G . Una transición "o.c." ocurre al leer un carácter que no corresponda a otra transición del estado.

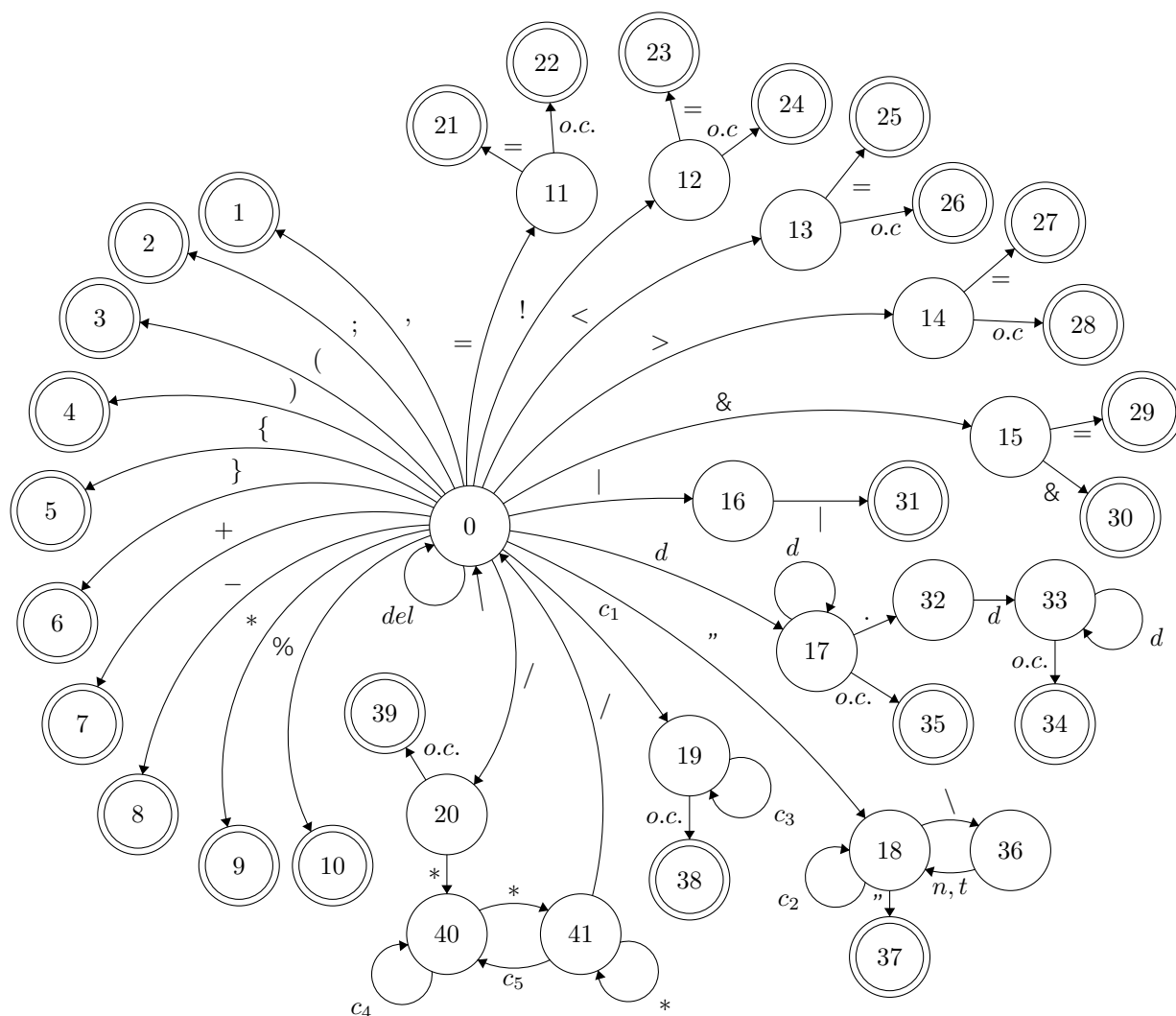


Figure 1: Autómata que reconoce el lenguaje $L(G)$

4.5 Acciones Semánticas

```

1  {
2    "city": [
3    {
4      "id": 1,
5      "name": "Toronto",
6      "country": "Canada",
7      "population": 6200000
8    },
9    {
10     "id": 2,
11     "name": "New York",
12     "country": "United States of America",
13     "population": 8800000
14   }
15 ]
16 }
```

5 Diseño de la Tabla de Símbolos

5.1 Estructura y Organización

6 Diseño del Parser

7 Diseño del Semanter

8 Gestor de Errores

Appendices

A Casos de Prueba

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget euismod pretium.