October 25, 2025

Procesador de MyJS: *jsp*

Memoria del Grupo 59 (Primera Entrega)

Por Andrés Súnico (23M018)

Índice

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 desarollar 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/suuniqo, 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
&=	AndAssign	-
=	Assign	-
,	Comma	-
;	Semi	-
(	LParen	-
)	RParen	-
{	LBrack	-
}	RBrack	-
Suma (+)	Sum	-
Por (*)	Mul	-
Resta (-)	Sub	-
División (/)	Div	-
Módulo (%)	Mod	-
Y lógico (&&)	And	-
O lógico (||)	Or	-
Negación (!)	Not	-
Menor (<)	Lt	-
Menor o igual (<=)	Le	-
Mayor (>)	Gt	-
Mayor o igual (>=)	Ge	-
Relacionales: Distinto (!=)	Ne	-
Igual (==)	Eq	-
Menos Unario (-)	Sub	-
Más Unario (+)	Sum	-
false	False	-
true	True	-

4.2 Errores

Cada tipo de error consta de un mensaje diferente y de una severidad, distinguiéndose *error* de *warning* (que no impediría la compilación del programa).

El procesador genera mensajes claros con número de línea y columna, muestra la línea afectada y subraya en color 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	*error*
Comentario inacabado	*error*
Cadena inacabada	error
Overflow de Cadena	*error*
Overflow de Entero	*error*
Overflow de Real	*error*
Formato de Real Inválido	*error*
Secuencia de Escape Inválida	*warning*

4.3 Gramática

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

$T = \{\text{Todo carácter } \textit{ASCII}\} \cup \{\textit{EOF}\}$
$N = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$

P se compone de las reglas:

$S \to delS \mid , \mid ; \mid (\mid) \mid \{ \mid \} \mid + \mid * \mid \% \mid =A \mid !B \mid <C \mid >D \mid \&E \mid |F \mid dG \mid "H \mid c_1 I \mid /J \mid \textit{EOF}$
$A \to = \mid \lambda$
$B \to = \mid \lambda$
$C \to = \mid \lambda$
$D \to = \mid \lambda$
$E \to = \mid \lambda$
$F \to \mid$
$G \to dG \mid .K \mid \lambda$
$K \to dK \mid \lambda$
$H \to c_2 H \mid \backslash L \mid "$
$L \to \mathsf{n}H \mid \mathsf{t}H$
$I \to c_3 I \mid \lambda$
$J \to *M \mid \lambda$
$M \to c_4 M \mid *N$
$N \to c_5 M \mid *N \mid /S$

Y se definen:

$del := \{\text{Todos los caracteres } \textit{ASCII whitespace}\}$
$ngr := \{\text{Todos los caracteres } \textit{ASCII} \text{ no gráficos}\}$
$d := \{0, 1, \ldots, 9\}$
$l := \{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}, \mathsf{A}, \mathsf{B}, \ldots, \mathsf{Z}\}$
$c_1 := l \cup \{\_\}$
$c_2 := T \setminus (\{\backslash, ", \textit{EOF}\} \cup ngr)$
$c_3 := c_1 \cup d$
$c_4 := T \setminus \{*, \textit{EOF}\}$
$c_5 := T \setminus \{*, /, \textit{EOF}\}$

4.4 Autómata

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

Se considera un error y se detiene la ejecución cuando el autómata lee un carácter con el que no puede transitar. Solo se alcanza un estado final cuando se ha reconocido un *token* exitosamente.

Como se explica en el siguiente apartado, un autómata no va a ser un modelo suficientemente potente como para representar las operaciones de un *Lexer*. Va a ser necesario complementarlo con algo más.

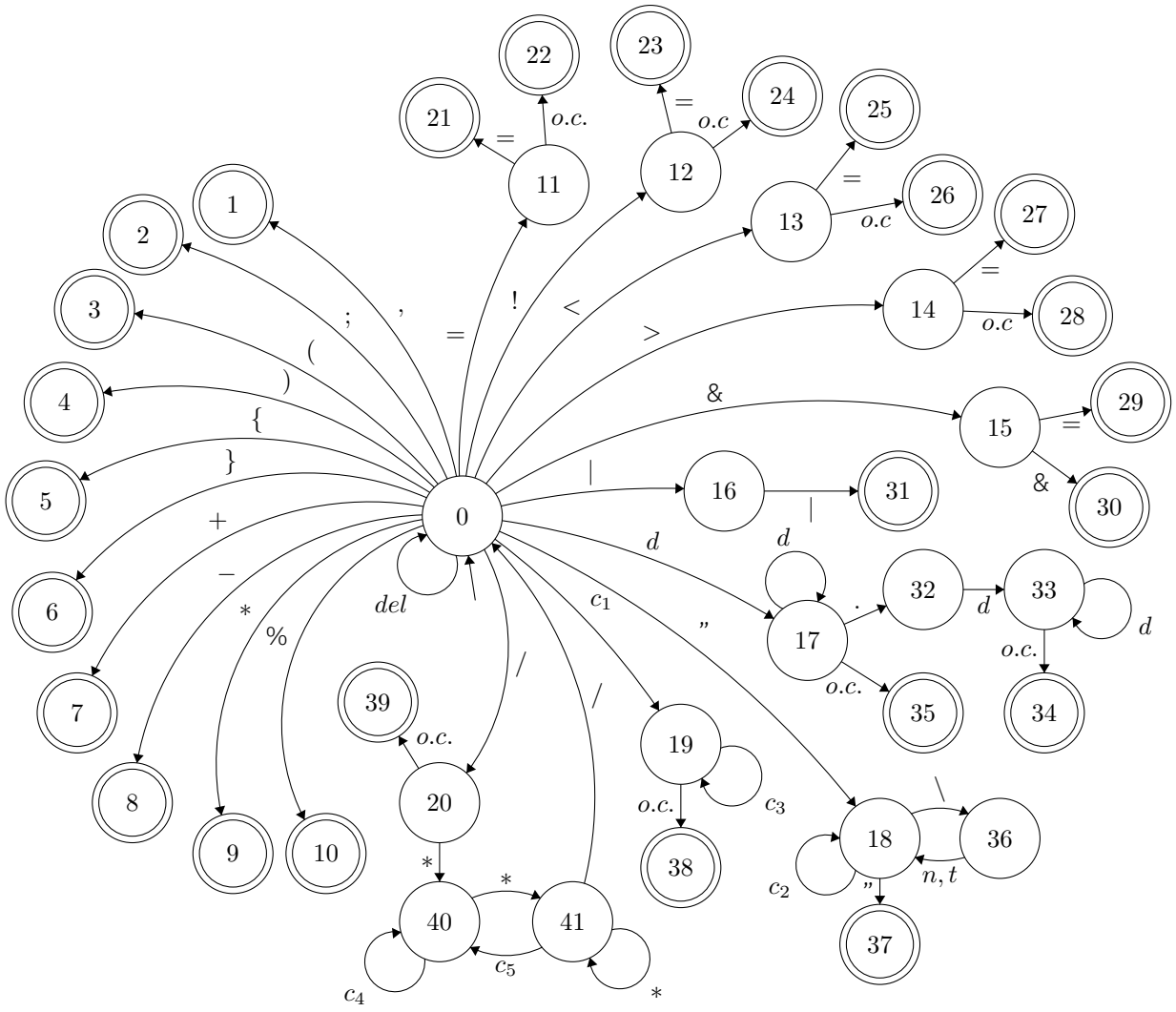

Figure 1: Autómata que reconoce el lenguaje $L(G)$

4.5 Acciones Semánticas

Las acciones semánticas son operaciones adicionales que se ejecutan durante las transiciones del autómata, con el propósito de aumentar la expresividad cuando es necesario. Resultan especialmente útiles para realizar conversiones de tipos o para simplificar la ejecución de otras acciones más complejas.

Por claridad, se dividen en varios grupos:

4.5.1 General

READ Segunda acción de toda transición menos 11:22, 12:24, 13:26, 14:28, 17:35, 33:34, 36:18, 19:38, 20:39

```
chr := read()
```

INV_CHAR Ante cualquier error no manejado en el resto de acciones

```
error("Illegal character")
```

4.5.2 Generación Directa

GEN_COMMA En la transición 0:1

```
gen_token(Comma, -)
```

GEN_SEMI En la transición 0:2

```
gen_token(Semi, -)
```

GEN_LPAREN En la transición 0:3

```
gen_token(LParen, -)
```

GEN_RPAREN En la transición 0:4

```
gen_token(RParen, -)
```

GEN_LBRACK En la transición 0:5

```
gen_token(LBrack, -)
```

GEN_RBRACK En la transición 0:6

```
gen_token(RBrack, -)
```

GEN_SUM En la transición 0:7

```
gen_token(Sum, -)
```

GEN_SUB En la transición 0:8

```
gen_token(Sub, -)
```

GEN_MUL En la transición 0:9

```
gen_token(Mul, -)
```

GEN_MOD En la transición 0:10

```
gen_token(Mod, -)
```

GEN_EQ En la transición 11:21

```
gen_token(Eq, -)
```

GEN_ASSIGN En la transición 11:22

```
gen_token(Assign, -)
```

GEN_NE En la transición 12:23

```
gen_token(Ne, -)
```

GEN_NOT En la transición 12:24

```
gen_token(Not, -)
```

GEN_LE En la transición 13:25

```
gen_token(Le, -)
```

GEN_LT En la transición 13:26

```
gen_token(Lt, -)
```

GEN_GE En la transición 14:27

```
gen_token(Ge, -)
```

GEN_GT En la transición 14:28

```
gen_token(Gt, -)
```

GEN_ANDASSIGN En la transición 15:29

```
gen_token(AndAssign, -)
```

GEN_AND En la transición 15:30

```
gen_token(And, -)
```

GEN_OR En la transición 16:31

```
gen_token(Or, -)
```

GEN_DIV En la transición 20:39

```
gen_token(Div, -)
```

4.5.3 Generación de Números

INIT_NUM En la transición 0:17

```
num := val(chr)
```

INIT_DEC En la transición 32:33

```
if (!is_ascii_digit(chr)) {
    error("Invalid Float Format")
} else {
    dec := 10
    num := num + vald(chr) / dec
}
```

GEN_DEC En la transición 33:34

```
if (size_bytes(num) > 16) {
    error("Float out of range")
} else {
    gen_token(FloatLit, num)
}
```

ADD_INTDIG En la transición 17:17

```
num := num * 10 + val(chr)
```

ADD_DECDIG En las transiciones 33:33

```
dec := dec * 10
num := num + vald(chr) / dec
```

GEN_INT En la transición 17:35

```
if (size_bytes(num) > 16) {
    error("Integer out of range")
} else {
    gen_token(IntLit, num)
}
```

4.5.4 Generación de Cadenas e Identificadores

INIT_STR_ID En la transición 0:18, 0:19

```
lex := ""
```

ADD_CHAR_ID En la transición 0:19, 19:19

```
lex.concat(chr)
```

ADD_CHAR_STR En la transición 18:18

```
if (chr == EOF) {
    error("Unterminated String")
} else {
    lex.concat(chr)
}
```

GEN_STR En la transición 18:37

```
if (lex.len() > 64) {
    error("String is too long")
} else {
    gen_token(StrLit, lex)
}
```

ADD_ESCSEQ En la transición 36:18

```
switch (chr) {
    case 'n' -> lex.concat('\n')
    case 't' -> lex.concat('\t')
    case EOF -> {
        error("Unfinsihed comment")
    }
    default  -> {
        warning("Invalid sequence")
        lex.concat('\\')
        lex.concat(chr)
    }
}
```

GEN_ID En la transición 19:38

```
code := search_keyword(lex)

if (code != null) {
    gen_token(code, -)
} else {
    pos := symtable_search(lex)

    if (pos == null) {
        pos := symtable_insert(lex)
    }
    gen_token(Id, pos)
}
```

4.5.5 Procesado de Comentarios

UNTERM_COMM En las transiciones 40:40, 40:41, 41:40, 41:41

```
if (chr == EOF) {
    error("Unterminated Comment")
}
```

5 Diseño de la Tabla de Símbolos

5.1 Estructura y Organización

6 Diseño del Parser

Se detallará en la próxima entrega.

7 Diseño del Semanter

Se detallará en la entrega final.

8 Gestor de Errores

Se detallará en la entrega final.

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.