

27 de octubre de 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	7
6. Diseño del Parser	7
7. Diseño del Semanter.....	7
8. Diseño del Gestor de Errores.....	7
A. Casos de Prueba	8

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 bibliotecas 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.

Cuadro 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
<i>&=</i>	AndAssign	-
<i>=</i>	Assign	-
<i>,</i>	Comma	-
<i>;</i>	Semi	-
<i>(</i>	LParen	-
<i>)</i>	RParen	-
<i>{</i>	LBrack	-
<i>}</i>	RBrack	-
Suma (<i>+</i>)	Sum	-
Por (<i>*</i>)	Mul	-
Resta (<i>-</i>)	Sub	-
División (<i>/</i>)	Div	-
Módulo (<i>%</i>)	Mod	-
<i>Y</i> lógico (<i>&&</i>)	And	-
<i>O</i> lógico (<i> </i>)	Or	-
Negación (<i>!</i>)	Not	-
Menor (<i><</i>)	Lt	-
Menor o igual (<i><=</i>)	Le	-
Mayor (<i>></i>)	Gt	-
Mayor o igual (<i>>=</i>)	Ge	-
Relacionales: Distinto (<i>!=</i>)	Ne	-
Igual (<i>==</i>)	Eq	-
Menos Unario (<i>-</i>)	Sub	-
Más Unario (<i>+</i>)	Sum	-
<i>false</i>	False	-
<i>true</i>	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)¹.

Cuadro 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:

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

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

P se compone de la regla del axioma:

$S \rightarrow delS \mid , \mid ; \mid (\mid) \mid \{ \mid \} \mid + \mid * \mid \% \mid =A \mid !B \mid <C \mid >D \mid \&E \mid |F \mid dG \mid "H \mid c_1I \mid /J \mid EOF$

Y del resto de reglas:

$A \rightarrow = \mid \lambda$

$B \rightarrow = \mid \lambda$

$C \rightarrow = \mid \lambda$

$D \rightarrow = \mid \lambda$

$E \rightarrow = \mid \lambda$

$F \rightarrow \mid$

$G \rightarrow dG \mid .K \mid \lambda$

$K \rightarrow dK \mid \lambda$

$H \rightarrow c_2H \mid \backslash L \mid "$

$L \rightarrow nH \mid tH$

$I \rightarrow c_3I \mid \lambda$

$J \rightarrow *M \mid \lambda$

$M \rightarrow c_4M \mid *N$

$N \rightarrow c_5M \mid *N \mid /S$

$del := \{\text{ASCII delimitadores}^2\}$
 $ngr := \{\text{ASCII con código } c : 32 \leq c \leq 126\}$
 $d := \{0, 1, \dots, 9\}$
 $l := \{a, b, \dots, z, A, B, \dots, Z\}$
 $c_1 := l \cup \{_ \}$
 $c_2 := T \setminus (\{\backslash, ", EOF\} \cup ngr)$
 $c_3 := c_1 \cup d$
 $c_4 := T \setminus \{*, EOF\}$
 $c_5 := T \setminus \{*, /, EOF\}$

¹ Se ha elegido este comportamiento para que el procesador sea fiel a la documentación oficial de *ECMAScript*.

² La definición de delimitador se toma de la documentación oficial de *ECMAScript*.

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.

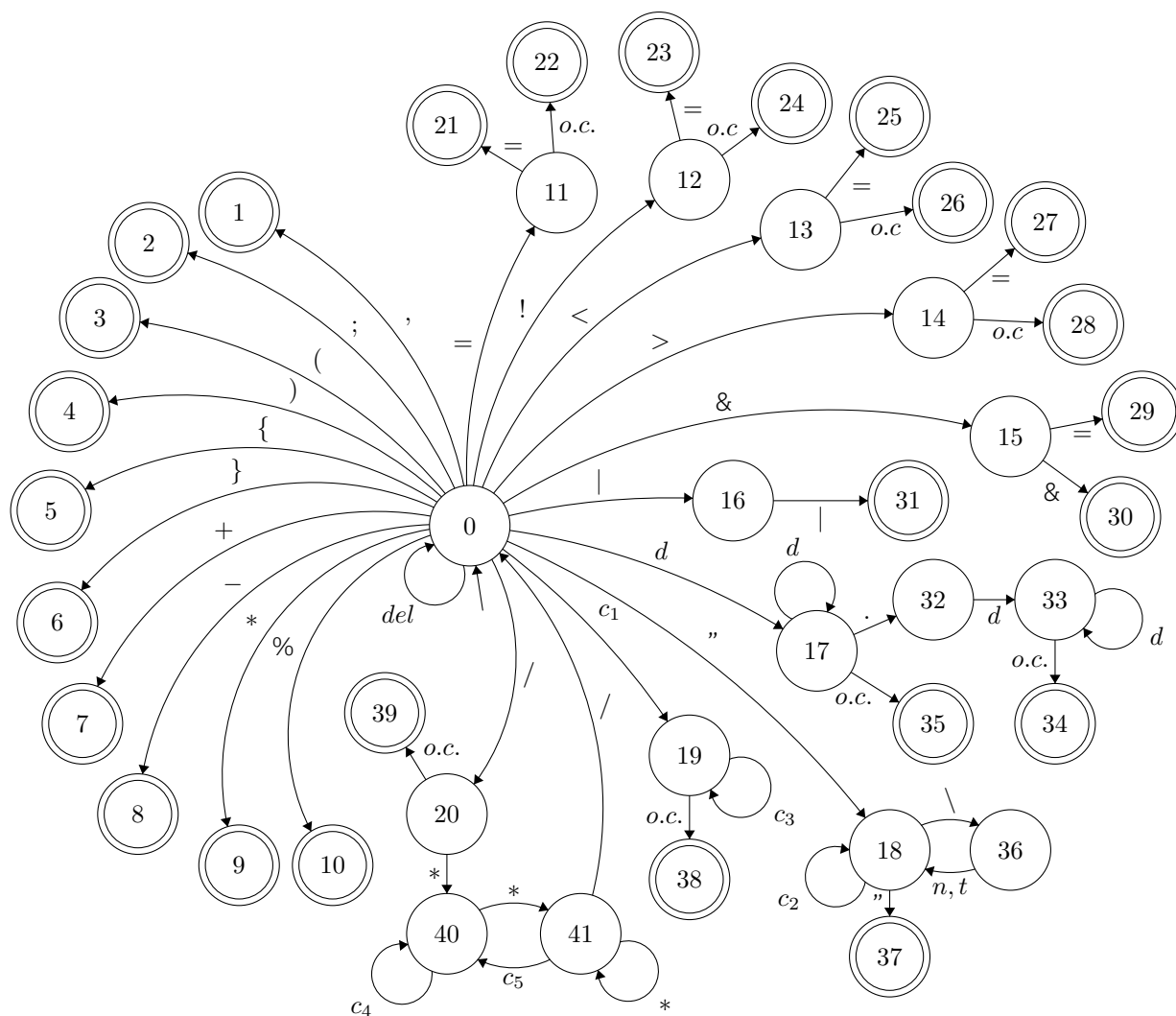


Figura 1: Autómata que reconoce el lenguaje $L(G)$

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

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

INV_CHAR Ante cualquier error no manejado en el resto de acciones

```
1 error("Illegal_character")
```

4.5.2 Generación Directa

GEN_COMMA En la transición 0:1

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

GEN_SEMI En la transición 0:2

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

GEN_LPAREN En la transición 0:3

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

GEN_RPAREN En la transición 0:4

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

GEN_LBRACK En la transición 0:5

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

GEN_RBRACK En la transición 0:6

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

GEN_SUM En la transición 0:7

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

GEN_SUB En la transición 0:8

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

GEN_MUL En la transición 0:9

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

GEN_MOD En la transición 0:10

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

GEN_EQ En la transición 11:21

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

GEN_ASSIGN En la transición 11:22

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

GEN_NE En la transición 12:23

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

GEN_NOT En la transición 12:24

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

GEN_LE En la transición 13:25

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

GEN_LT En la transición 13:26

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

GEN_GE En la transición 14:27

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

GEN_GT En la transición 14:28

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

GEN_ANDASSIGN En la transición 15:29

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

GEN_AND En la transición 15:30

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

GEN_OR En la transición 16:31

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

GEN_DIV En la transición 20:39

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

4.5.3 Generación de Números**INIT_NUM** En la transición 0:17

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

INIT_DEC En la transición 32:33

```
1 if (!is_ascii_digit(chr)) {
2   error("Invalid_Float_Format")
3 } else {
4   dec := 10
5   num := num + vald(chr) / dec
6 }
```

GEN_DEC En la transición 33:34

```
1 if (size_bytes(num) > 16) {
2   error("Float_out_of_range")
3 } else {
4   gen_token(FloatLit, num)
5 }
```

4.5.4 Generación de Cadenas e Identificadores**INIT_STR_ID** En la transición 0:18, 0:19

```
1 lex := ""
```

ADD_CHAR_STR En la transición 18:18

```
1 if (chr == EOF) {
2   error("Unterminated_String")
3 } else {
4   lex.concat(chr)
5 }
```

ADD_ESCSEQ En la transición 36:18

```
1 switch (chr) {
2   case 'n' -> lex.concat('\n')
3   case 't' -> lex.concat('\t')
4   case EOF -> {
5     error("Unfinsihed_comment")
6   }
7   default -> {
8     warning("Invalid_sequence")
9     lex.concat('\\')
10    lex.concat(chr)
11  }
12 }
```

4.5.5 Procesado de Comentarios**UNTERM_COMM** En las transiciones 40:40, 40:41, 41:40, 41:41

```
1 if (chr == EOF) {
2   error("Unterminated_Comment")
3 }
```

ADD_INTDIG En la transición 17:17

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

ADD_DECDIG En las transiciones 33:33

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

GEN_INT En la transición 17:35

```
1 if (size_bytes(num) > 16) {
2   error("Integer_out_of_range")
3 } else {
4   gen_token(IntLit, num)
5 }
```

ADD_CHAR_ID En la transición 0:19, 19:19

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

GEN_STR En la transición 18:37

```
1 if (lex.len() > 64) {
2   error("String_is_too_long")
3 } else {
4   gen_token(StrLit, lex)
5 }
```

GEN_ID En la transición 19:38

```
1 code := search_keyword(lex)
2
3 if (code != null) {
4   gen_token(code, -)
5 } else {
6   pos := symtable_search(lex)
7
8   if (pos == null) {
9     pos := symtable_insert(lex)
10  }
11  gen_token(Id, pos)
12 }
```

5 Diseño de la Tabla de Símbolos

Se trata de un tipo abstracto de datos encargado de gestionar la información relevante a los identificadores del programa. Todos los módulos del procesador van a necesitar acceder a ella con distintos propósitos por lo que es importante que tanto la inserción como la consulta de datos sea eficiente.

5.1 Estructura y Organización

5.1.1 Entradas

La información de los identificadores se va a guardar en la tabla de símbolos en forma de entradas. Como en *MyJS* no existen los *arrays* se distinguen únicamente 2 tipos:

Entrada Básica Para todos los tipos básicos, es decir, *int*, *float*, *string* y *bool*.

Lexema Nombre de la variable.

Tipo Tipo de la variable.

Desplazamiento Desplazamiento en memoria relativo a su ámbito.

Entrada Función Para las funciones. Nótese que 'Tipos Argumentos' es un puntero a una lista de tipos.

Lexema Nombre de la función.

Tipo Retorno Tipo que devuelve la función, pudiendo ser *Void*.

Tipos Argumentos Lista de los tipos de los parámetros en orden.

Etiqueta Etiqueta que se usará para navegar a la función en el código ensamblador.

Cada entrada va a tener una estructura de 'llave-valor', dónde el lexema del identificador actúa como llave, y sus atributos (toda su información relevante) como valor. Como cada llave identifica de forma única cada entrada, se puede optimizar el complejidad de acceso e inserción a $O(1)$ usando *hashmaps*.

5.1.2 Ámbitos

No siempre se puede acceder a cada variable de un programa. Por ejemplo, desde una función no se puede acceder a una variable local de otra. Por ello, por cada ámbito se va a crear una tabla de símbolos distinta. Además, como *MyJS* es un lenguaje sin anidamiento de funciones, en cada momento habrá como máximo 2 tablas de símbolos activas: la global y, opcionalmente, la de una función.

De esta manera, se puede comprender una tabla de símbolos como una *stack* de ámbitos (es decir, tablas de símbolos locales), dónde el Analizador Semántico será el encargado de apilar y desapilar ámbitos al entrar y salir de funciones respectivamente.

6 Diseño del Parser

Se detallará en la próxima entrega.

7 Diseño del Semanter

Se detallará en la entrega final.

8 Diseño del Gestor de Errores

Se detallará en la entrega final.

Anexo

A Casos de Prueba

Se va a probar el funcionamiento del procesador con 6 ficheros fuentes distintos. La mitad de ellos serán correctos y la otra incorrectos. En los casos correctos se volcará el fichero de *tokens* y de la tabla de símbolos; en los incorrectos los diagnósticos generados por el gestor de errores.

A.1 Casos Correctos

fib.javascript Fichero con un estilo limpio y estándar.

```
1  /* This function computes the nth fibonacci number */
2  function int fib(int n) {
3      if (n == 0) {
4          return 0;
5      }
6      if (n == 1) {
7          return 1;
8      }
9
10     let int a = 0;
11     let int b = 1;
12
13     do {
14         let int c = a + b;
15
16         a = b;
17         b = c;
18
19         n = n - 1;
20     } while (n >= 2);
21
22     return b;
23 }
```

Fichero de *tokens*:

```
1  <Func, >
2  <Int, >
3  <Id, 0>
4  <LParen, >
5  <Int, >
6  <Id, 1>
7  <RParen, >
8  <LBrack, >
9  <If, >
10 <LParen, >
11 <Id, 1>
12 <Eq, >
13 <IntLit, 0>
14 <RParen, >
15 <LBrack, >
16 <Ret, >
17 <IntLit, 0>
18 <Semi, >
19 <RBrack, >
20 <If, >
21 <LParen, >
```

```
22 <Id, 1>
23 <Eq, >
24 <IntLit, 1>
25 <RParen, >
26 <LBrack, >
27 <Ret, >
28 <IntLit, 1>
29 <Semi, >
30 <RBrack, >
31 <Let, >
32 <Int, >
33 <Id, 2>
34 <Assign, >
35 <IntLit, 0>
36 <Semi, >
37 <Let, >
38 <Int, >
39 <Id, 3>
40 <Assign, >
41 <IntLit, 1>
42 <Semi, >
43 <Do, >
44 <LBrack, >
```

```
45 <Let, >
46 <Int, >
47 <Id, 4>
48 <Assign, >
49 <Id, 2>
50 <Sum, >
51 <Id, 3>
52 <Semi, >
53 <Id, 2>
54 <Assign, >
55 <Id, 3>
56 <Semi, >
57 <Id, 3>
58 <Assign, >
59 <Id, 4>
60 <Semi, >
61 <Id, 1>
62 <Assign, >
63 <Id, 1>
64 <Sub, >
65 <IntLit, 1>
66 <Semi, >
67 <RBrack, >
```

```
68 <While, >
69 <LParen, >
70 <Id, 1>
71 <Ge, >
72 <IntLit, 2>
73 <RParen, >
74 <Semi, >
75 <Ret, >
76 <Id, 3>
77 <Semi, >
78 <RBrack, >
```

Tabla de símbolos:

```
1  table #0:
2  * 'fib'
3  * 'n'
4  * 'a'
5  * 'b'
6  * 'c'
```


factorial.js Fichero con un código más comprimido y comentarios más raros.

```
1  /* ***** */
2  /*      * / * N FACTORIAL * / *      */
3  /* ***** */
4
5  /*
6   * This function computes n! (n factorial)
7   */
8  function int factorial(int n) {
9      if (n==0){return 1;}
10     if (n<=2){return n;}
11     let int res=n;
12     do{n=n-1;res=res*n;}while(n>2);
13     return res;
14 }
15
16 /* read n and write n! */
17 let int n=0;read(n);let int res=factorial(n);write("n!= ", res);
18
19 /** eof **/
```

Fichero de tokens:

1 <Func, >	32 <Int, >	65 <Id, 1>
2 <Int, >	33 <Id, 2>	66 <Assign, >
3 <Id, 0>	34 <Assign, >	67 <IntLit, 0>
4 <LParen, >	35 <Id, 1>	68 <Semi, >
5 <Int, >	36 <Semi, >	69 <Read, >
6 <Id, 1>	37 <Do, >	70 <LParen, >
7 <RParen, >	38 <LBrack, >	71 <Id, 1>
8 <LBrack, >	39 <Id, 1>	72 <RParen, >
9 <If, >	40 <Assign, >	73 <Semi, >
10 <LParen, >	41 <Id, 1>	74 <Let, >
11 <Id, 1>	42 <Sub, >	75 <Int, >
12 <Eq, >	43 <IntLit, 1>	76 <Id, 2>
13 <IntLit, 0>	44 <Semi, >	77 <Assign, >
14 <RParen, >	45 <Id, 2>	78 <Id, 0>
15 <LBrack, >	46 <Assign, >	79 <LParen, >
16 <Ret, >	47 <Id, 2>	80 <Id, 1>
17 <IntLit, 1>	48 <Mul, >	81 <RParen, >
18 <Semi, >	49 <Id, 1>	82 <Semi, >
19 <RBrack, >	50 <Semi, >	83 <Write, >
20 <If, >	51 <RBrack, >	84 <LParen, >
21 <LParen, >	52 <While, >	85 <StrLit, "n! = ">
22 <Id, 1>	53 <LParen, >	86 <Comma, >
23 <Le, >	54 <Id, 1>	87 <Id, 2>
24 <IntLit, 2>	55 <Gt, >	88 <RParen, >
25 <RParen, >	56 <IntLit, 2>	89 <Semi, >
26 <LBrack, >	57 <RParen, >	
27 <Ret, >	58 <Semi, >	
28 <Id, 1>	59 <Ret, >	
29 <Semi, >	60 <Id, 2>	
30 <RBrack, >	61 <Semi, >	
31 <Let, >	62 <RBrack, >	
	63 <Let, >	
	64 <Int, >	

Tabla de símbolos:

```
1 table #0:
2 * 'factorial'
3 * 'n'
4 * 'res'
```

fuzz.javascript Fichero correcto pero caótico, con el objetivo de obtener *edgecases*.

```

1 let int global=13;
2 function void nothing(void) { read(global) do{global=global-1;}while(global>0);}
3 function int main(float b, string d){let string aux="this is a str";
4 /** this is a comment */ let float foo=1203.123; let int bar=2398;
5 ;;;/* semicolons */;; let oper=2||5;let boolean bool=false;
6 let boolean george=true;george&=bool;}/** / eof */

```

Fichero de *tokens*:

```

1 <Let, >
2 <Int, >
3 <Id, 0>
4 <Assign, >
5 <IntLit, 13>
6 <Semi, >
7 <Func, >
8 <Void, >
9 <Id, 1>
10 <LParen, >
11 <Void, >
12 <RParen, >
13 <LBrack, >
14 <Read, >
15 <LParen, >
16 <Id, 0>
17 <RParen, >
18 <Do, >
19 <LBrack, >
20 <Id, 0>
21 <Assign, >
22 <Id, 0>
23 <Sub, >
24 <IntLit, 1>
25 <Semi, >
26 <RBrack, >
27 <While, >
28 <LParen, >
29 <Id, 0>
30 <Gt, >
31 <IntLit, 0>
32 <RParen, >
33 <Semi, >
34 <RBrack, >
35 <Func, >

```

```

36 <Int, >
37 <Id, 2>
38 <LParen, >
39 <Float, >
40 <Id, 3>
41 <Comma, >
42 <Str, >
43 <Id, 4>
44 <RParen, >
45 <LBrack, >
46 <Let, >
47 <Str, >
48 <Id, 5>
49 <Assign, >
50 <StrLit, "this is a str">
51 <Semi, >
52 <Let, >
53 <Float, >
54 <Id, 6>
55 <Assign, >
56 <FloatLit, 1203.123>
57 <Semi, >
58 <Let, >
59 <Int, >
60 <Id, 7>
61 <Assign, >
62 <IntLit, 2398>
63 <Semi, >
64 <Semi, >
65 <Semi, >
66 <Semi, >
67 <Semi, >
68 <Semi, >
69 <Semi, >
70 <Let, >
71 <Id, 8>
72 <Assign, >

```

```

73 <IntLit, 2>
74 <Or, >
75 <IntLit, 5>
76 <Semi, >
77 <Let, >
78 <Bool, >
79 <Id, 9>
80 <Assign, >
81 <False, >
82 <Semi, >
83 <Let, >
84 <Bool, >
85 <Id, 10>
86 <Assign, >
87 <True, >
88 <Semi, >
89 <Id, 10>
90 <AndAssign, >
91 <Id, 9>
92 <Semi, >
93 <RBrack, >

```

Tabla de símbolos:

```

1 table #0:
2 * 'global'
3 * 'nothing'
4 * 'main'
5 * 'b'
6 * 'd'
7 * 'aux'
8 * 'foo'
9 * 'bar'
10 * 'oper'
11 * 'bool'
12 * 'george'

```

A.2 Casos Incorrectos

unterm.javascript Fichero con errores de sentencias incompletas.

```

1  /* unfinished strings */
2  let string foo = "im not finishing this string
3  let string bar = "im not finishing this one either\
4  let string rep = "last one\q
5
6  /* unfinished floats */
7  let float x = 478234.
8  let float y = 78234..
9  let float z = 2343...
10
11  /** * / * /**/ this comment is finished /* /* */
12  /** this one is not ** ***** */ /*
13
14  function int main {
15      let string useless = "this variable won't be recognised";
16      return 0;
17  }
```

Diagnósticos:

```

1  unterm.javascript:2:18: error: missing terminating character ''' on string literal
2      2 | let string foo = "im not finishing this string
3          |               ^~~~~~
4  unterm.javascript:3:18: error: missing terminating character ''' on string literal
5      3 | let string bar = "im not finishing this one either\
6          |               ^~~~~~
7  unterm.javascript:4:27: warning: unknown escape sequence '\q'
8      4 | let string rep = "last one\q
9          |               ^~
10 unterm.javascript:4:18: error: missing terminating character ''' on string literal
11      4 | let string rep = "last one\q
12          |               ^~~~~~
13 unterm.javascript:7:15: error: expected digit after '.' in float literal
14      7 | let float x = 478234.
15          |               ^~~~~~
16 unterm.javascript:8:15: error: expected digit after '.' in float literal
17      8 | let float y = 78234..
18          |               ^~~~~~
19 unterm.javascript:8:21: error: illegal character '.' in program
20      8 | let float y = 78234..
21          |               ^
22 unterm.javascript:9:15: error: expected digit after '.' in float literal
23      9 | let float z = 2343...
24          |               ^~~~~~
25 unterm.javascript:9:20: error: illegal character '.' in program
26      9 | let float z = 2343...
27          |               ^
28 unterm.javascript:9:21: error: illegal character '.' in program
29      9 | let float z = 2343...
30          |               ^
31 unterm.javascript:12:1: error: unterminated comment
32     12 | /** this one is not ** ***** */ /*
33         | ^~
```

overflow.javascript Fichero con errores de constantes fuera del rango admitido.

```
1 /* string with 64 characters */
2 let string foo = "ffffffffffffffffffffffffffffffffffffffffffffffffffffffff";
3
4 /* string with 65 characters */
5 let string bar = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
6
7 /* max int16 */
8 let int imax = 32767;
9 let int imax_plus_one = 32768;
10
11 /* max float */
12 let float fmax = 340282346638528859811704183484516925440.0;
13 let float fmax_plus_one = 340282346638528859811704183484516925441;
14 let float lots_of_decimals = 0.2347028349820934809218409238845290380928;
```

Diagnósticos:

```
1 overflow.javascript:5:18: error: string literal is too long, length is 65 but the
   maximum is 64
2   5 | let string bar = "
     |      bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
3     |      ^~~~~~
4 overflow.javascript:9:25: error: integer literal out of range for 16-byte type
5   9 | let int imax_plus_one = 32768;
     |                        ^~~~~
6
7 overflow.javascript:13:27: error: integer literal out of range for 16-byte type
8  13 | let float fmax_plus_one = 340282346638528859811704183484516925441;
     |                           ^~~~~~
9
```

noise.javascript Fichero de ruido, repleto de errores diferentes.

```

1 123.45.67 12abc 12.34.56..78
2 -+*/% &&& ||| != == <= >= >< != &== |= <=>
3 "unterminated string" "string with illegal char \q" ""empty""
4 "bad\escape" "no closing quote
5 /* nested comment */ /* comment */ /* unclosed comment
6 @@@ ### $$$ %%% ^^ ^^^ ??? !!! <<< >>> === =====
7 ++ -- ** // %% != &== |= != <=> &&||!!
8 "unterminated string again /* unclosed comment
9 "bad string with \x illegal escape" "" ""empty""
10 /* final comment */ /* another unclosed

```

Diagnósticos:

```

1 noise.javascript:1:7: error: illegal character '.' in program
2   1 | 123.45.67 12abc 12.34.56..78
3     |           ^
4 noise.javascript:1:22: error: illegal character '.' in program
5   1 | 123.45.67 12abc 12.34.56..78
6     |           ^
7 noise.javascript:1:23: error: expected digit after '.' in float literal
8   1 | 123.45.67 12abc 12.34.56..78
9     |           ^~~
10 noise.javascript:1:26: error: illegal character '.' in program
11  1 | 123.45.67 12abc 12.34.56..78
12   |           ^
13 noise.javascript:2:9: error: illegal character '&' in program
14  2 | -+*/% &&& ||| != == <= >= >< != &== |= <=>
15   |           ^
16 noise.javascript:2:13: error: illegal character '|' in program
17  2 | -+*/% &&& ||| != == <= >= >< != &== |= <=>
18   |           ^
19 noise.javascript:2:38: error: illegal character '|' in program
20  2 | -+*/% &&& ||| != == <= >= >< != &== |= <=>
21   |                               ^
22 noise.javascript:3:53: warning: unknown escape sequence '\q'
23  3 | "unterminated string" "string with illegal char \q" ""empty""
24   |                                           ^~
25 noise.javascript:4:5: warning: unknown escape sequence '\e'
26  4 | "bad\escape" "no closing quote
27   |       ^~
28 noise.javascript:4:14: error: missing terminating character "'" on string literal
29  4 | "bad\escape" "no closing quote
30   |           ^~~~~~
31 noise.javascript:10:21: error: unterminated comment
32 10 | /* final comment */ /* another unclosed
33   |               ^~

```