24 de noviembre de 2025

Procesador de MyJS: *jsp*

Memoria del Grupo 59

Andrés Súnico

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 bibliotecas 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 los operadores:

- Aritméticos: suma ($+$) y multiplicación ($*$)
- Relacionales: menor ($<$) e igual ($==$)
- Lógicos: negación (!) e Y lógico (&&)
- Unarios: más ($+$) y menos ($-$)

Además se ha escogido implementar el tratamiento de secuencias de escape (*\n* y *\t*) y de las *keywords true* y *false*;

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 (tan sólo 4 de los 33 *tokens* usarán 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
&=	AndAssign	-
=	Assign	-
,	Comma	-
;	Semi	-
(	LParen	-
)	RParen	-
{	LBrack	-
}	RBrack	-
Suma ($+$)	Sum	-
Por ($*$)	Mul	-
Y lógico (&&)	And	-
Negación (!)	Not	-
Menor ($<$)	Lt	-
Igual ($==$)	Eq	-
Menos ($-$)	Sub	-
Más ($+$)	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.

Por aclarar, una cadena malformada es aquella que contiene caracteres *ASCII* no gráficos.

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	*error*
Comentario inacabado	*error*
Cadena inacabada	*error*
Cadena malformada	*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 la tupla $G = (T, N, S, P)$, dónde:

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

P se compone de la regla del axioma:

$S \rightarrow delS \mid , \mid ; \mid (\mid) \mid \{ \mid \} \mid + \mid * \mid \% \mid =A \mid !B \mid <C \mid >D \mid \&E \mid |F \mid dG \mid "H \mid c_1 I \mid /J \mid \textit{EOF}$

Y del resto de reglas:

$A \rightarrow = \mid \lambda$
$B \rightarrow = \mid \lambda$
$C \rightarrow = \mid \lambda$
$D \rightarrow = \mid \lambda$
$E \rightarrow = \mid \lambda$
$F \rightarrow \mid$
$G \rightarrow dG \mid .K \mid \lambda$
$K \rightarrow dL$
$L \rightarrow dL \mid \lambda$
$H \rightarrow c_2 H \mid \backslash M \mid "$
$M \rightarrow \text{n}H \mid \text{t}H$
$I \rightarrow c_3 I \mid \lambda$
$J \rightarrow *N \mid \lambda$
$N \rightarrow c_4 N \mid *O$
$O \rightarrow c_5 N \mid *O \mid /S$

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

El lenguaje generado por esta gramática, $L(G)$, está compuesto por el conjunto de todos los *tokens* válidos del lenguaje de programación *MyJS*. Por tanto, dada una cadena de símbolos terminales, la gramática G es capaz de detectar si forma o no un *token* válido.

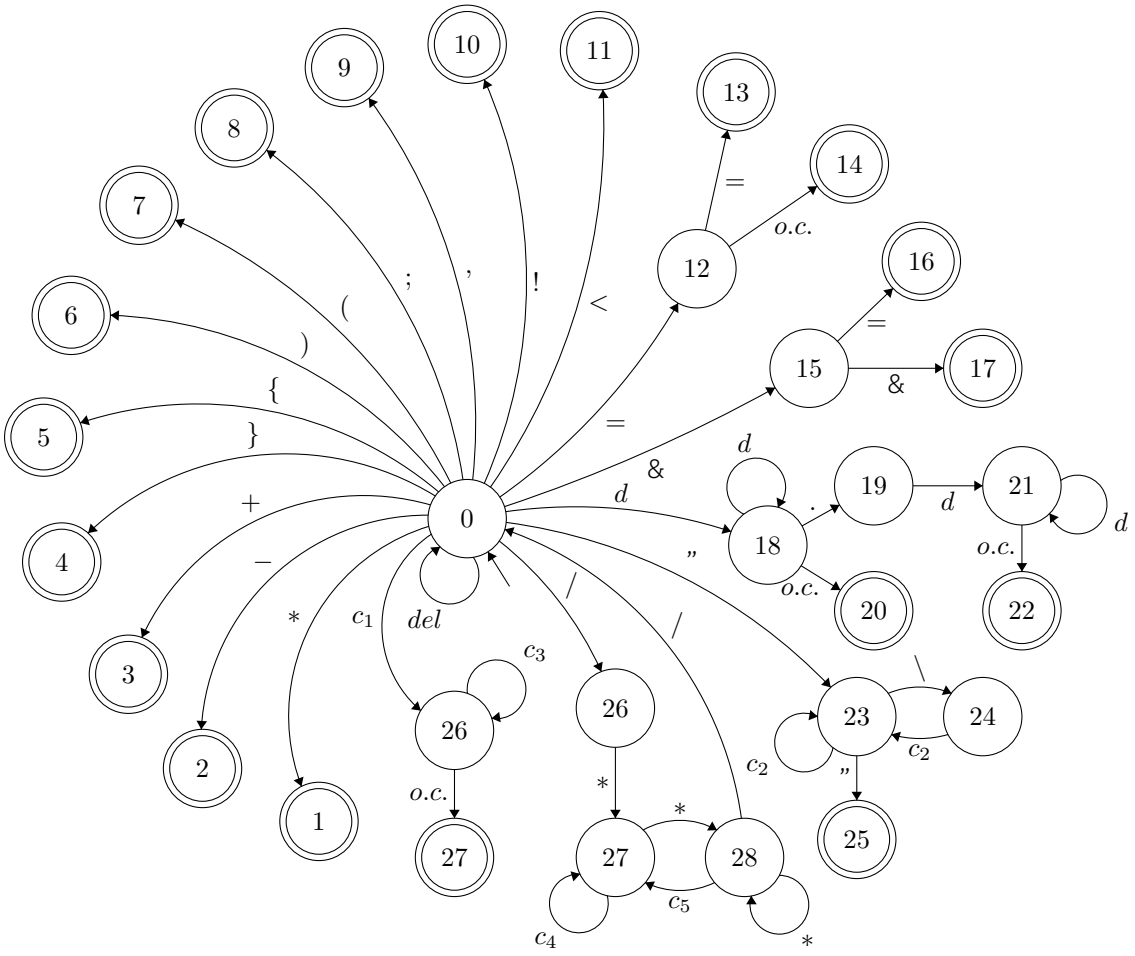
[1] Se ha elegido este comportamiento para que el procesador sea fiel a la documentación oficial de *ECMAScript*.
[2] La definición de delimitador se toma de la documentación oficial de *ECMAScript*.

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 el conjunto de Acciones Semánticas detallado en la siguiente sección.



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 12:14, 18:20, 21:22, 24:23, 29:30

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

4.5.2 Errores

MALFORMED_STR Si en el estado 23 o 24 se recibe un carácter *ASCII* no gráfico

```
error("Malformed␣string␣literal")
```

UNTERM_STR Si en el estado 23 o 24 se recibe *EOF*

```
error("Unterminated␣string␣literal")
```

INV_FLOAT_FMT Si en el estado 19 no se puede transitar

```
error("Invalid␣float␣literal␣format")
```

UNTERM_COMM Si en el estado 27 o 28 no se puede transitar

```
error("Unterminated␣comment")
```

INV_CHAR Ante cualquier error no manejado en el resto de acciones

```
error("Illegal␣character")
```

4.5.3 Generación Directa

GEN_MUL En la transición 0:1

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

GEN_SUB En la transición 0:2

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

GEN_SUM En la transición 0:3

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

GEN_RBRACK En la transición 0:4

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

GEN_LBRACK En la transición 0:5

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

GEN_RPAREN En la transición 0:6

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

GEN_LPAREN En la transición 0:7

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

GEN_SEMI En la transición 0:8

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

GEN_COMMA En la transición 0:9

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

GEN_NOT En la transición 0:10

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

GEN_LT En la transición 0:11

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

GEN_EQ En la transición 12:13

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

GEN_ASSIGN En la transición 12:14

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

GEN_ANDASSIGN En la transición 15:16

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

GEN_AND En la transición 15:17

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

4.5.4 Generación de Números

INIT_NUM En la transición 0:18

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

INIT_DEC En la transición 20:21

```
dec := 10
num := num + val(chr) / dec
```

GEN_DEC En la transición 21:22

```
if (num > 3.4028235e38) {
    error("Float literal out of range")
} else {
    gen_token(FloatLit, num)
}
```

ADD_INTDIG En la transición 18:18

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

ADD_DECDIG En las transiciones 21:21

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

GEN_INT En la transición 18:20

```
if (num > 32767) {
    error("Integer literal out of range")
} else {
    gen_token(IntLit, num)
}
```

4.5.5 Generación de Cadenas e Identificadores

INIT_STR En la transición 0:23

```
lex := ""
len := 0
```

ADD_CHAR_STR En la transición 23:23

```
lex.concat(chr)
len := len + 1
```

ADD_CHAR_ID En la transición 0:26, 26:26

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

ADD_ESCSEQ En la transición 24:23

```
len := len + 1
switch (chr) {
    case 'n' -> lex.concat('\n')
    case 't' -> lex.concat('\t')
    default  -> {
        warning("Invalid sequence")
        lex.concat('\\')
        lex.concat(chr)
        len := len + 1
    }
}
```

INIT_ID En la transición 0:26

```
lex := ""
```

GEN_STR En la transición 23:25

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

GEN_ID En la transición 26:27

```
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)
}
```

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.

6.1 Gramática

Se define la gramática del *Parser* como la tupla $\hat{G} = (T, N, S, P)$, dónde:

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

P se compone de la regla del axioma:

$S \rightarrow delS$ | , | ; | (|) | { | } | + | * | % | =A | !B | <C | >D | &E | |F | dG | "H | $c_1 I$ | /J | *EOF*

Y del resto de reglas:

$A \rightarrow =\ |\ \lambda$
$B \rightarrow =\ |\ \lambda$
$C \rightarrow =\ |\ \lambda$
$D \rightarrow =\ |\ \lambda$
$E \rightarrow =\ |\ \lambda$
$F \rightarrow\ |$
$G \rightarrow dG\ |\ .K\ |\ \lambda$
$K \rightarrow dL$
$L \rightarrow dL\ |\ \lambda$
$H \rightarrow c_2 H\ |\ \backslash M\ |\ "$
$M \rightarrow \text{n}H\ |\ \text{t}H$
$I \rightarrow c_3 I\ |\ \lambda$
$J \rightarrow *N\ |\ \lambda$
$N \rightarrow c_4 N\ |\ *O$
$O \rightarrow c_5 N\ |\ *O\ |\ /S$

El lenguaje generado por esta gramática, $L(G)$, está compuesto por el conjunto de todos los *tokens* válidos del lenguaje de programación *MyJS*. Por tanto, dada una cadena de símbolos terminales, la gramática G es capaz de detectar si forma o no un *token* válido.

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.

```
/* This function computes the nth fibonacci number */
function int fib(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }

    let int a = 0;
    let int b = 1;

    do {
        let int c = a + b;

        a = b;
        b = c;

        n = n - 1;
    } while (n >= 2);

    return b;
}
```

Fichero de *tokens*:

```
<Func, >
<Int, >
<Id, 0>
<LParen, >
<Int, >
<Id, 1>
<RParen, >
<LBrack, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 0>
<RParen, >
<LBrack, >
<Ret, >
<IntLit, 0>
<Semi, >
<RBrack, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 1>
<RParen, >
<LBrack, >
<Ret, >
<IntLit, 1>
<Semi, >
<RBrack, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<IntLit, 0>
<Semi, >
<Let, >
<Int, >
<Id, 3>
<Assign, >
<IntLit, 1>
<Semi, >
<Do, >
<LBrack, >
<Let, >
<Int, >
<Id, 4>
<Assign, >
<Id, 2>
<Sum, >
<Id, 3>
<Semi, >
<Id, 2>
<Assign, >
<Id, 3>
<Semi, >
<Id, 3>
<Assign, >
<Id, 4>
<Semi, >
<Id, 1>
<Assign, >
<Id, 1>
<Sub, >
<IntLit, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Id, 1>
<Ge, >
<IntLit, 2>
<RParen, >
<Semi, >
<Ret, >
<Id, 3>
<Semi, >
<RBrack, >
```

Tabla de símbolos:

```
table #0:
* 'fib'
* 'n'
* 'a'
* 'b'
* 'c'
```

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

```
/*******************************************/
/*          * / * N FACTORIAL * / *          */
/*******************************************/

/*
* This function computes n! (n factorial)
*/
function int factorial(int n) {
    if(n==0){return 1;}
    if(n<=2){return n;}
    let int res=n;
    do{n=n-1;res=res*n;}while(n>2);
    return res;
}

/* read n and write n! */
let int n=0;read(n);let int res=factorial(n);write("n!␣=␣", res);

/*** eof ***/
```

Fichero de *tokens*:

```
<Func, >
<Int, >
<Id, 0>
<LParen, >
<Int, >
<Id, 1>
<RParen, >
<LBrack, >
<If, >
<LParen, >
<Id, 1>
<Eq, >
<IntLit, 0>
<RParen, >
<LBrack, >
<Ret, >
<IntLit, 1>
<Semi, >
<RBrack, >
<If, >
<LParen, >
<Id, 1>
<Le, >
<IntLit, 2>
<RParen, >
<LBrack, >
<Ret, >
<Id, 1>
<Semi, >
<RBrack, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<Id, 1>
<Semi, >
<Do, >
<LBrack, >
<Id, 1>
<Assign, >
<Id, 1>
<Sub, >
<IntLit, 1>
<Semi, >
<Id, 2>
<Assign, >
<Id, 2>
<Mul, >
<Id, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Id, 1>
<Gt, >
<IntLit, 2>
<RParen, >
<Semi, >
<Ret, >
<Id, 2>
<Semi, >
<RBrack, >
<Let, >
<Int, >
<Id, 1>
<Assign, >
<IntLit, 0>
<Semi, >
<Read, >
<LParen, >
<Id, 1>
<RParen, >
<Semi, >
<Let, >
<Int, >
<Id, 2>
<Assign, >
<Id, 0>
<LParen, >
<Id, 1>
<RParen, >
<Semi, >
<Write, >
<LParen, >
<StrLit, "n! = ">
<Comma, >
<Id, 2>
<RParen, >
<Semi, >
```

Tabla de símbolos:

```
table #0:
* 'factorial'
* 'n'
* 'res'
```

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

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

Fichero de *tokens*:

```
<Let, >
<Int, >
<Id, 0>
<Assign, >
<IntLit, 13>
<Semi, >
<Func, >
<Void, >
<Id, 1>
<LParen, >
<Void, >
<RParen, >
<LBrack, >
<Read, >
<LParen, >
<Id, 0>
<RParen, >
<Do, >
<LBrack, >
<Id, 0>
<Assign, >
<Id, 0>
<Sub, >
<IntLit, 1>
<Semi, >
<RBrack, >
<While, >
<LParen, >
<Id, 0>
<Gt, >
<IntLit, 0>
<RParen, >
<Semi, >
<RBrack, >
<Func, >
<Int, >
<Id, 2>
<LParen, >
<Float, >
<Id, 3>
<Comma, >
<Str, >
<Id, 4>
<RParen, >
<LBrack, >
<Let, >
<Str, >
<Id, 5>
<Assign, >
<StrLit, "this is a str">
<Semi, >
<Let, >
<Float, >
<Id, 6>
<Assign, >
<FloatLit, 1203.123>
<Semi, >
<Let, >
<Int, >
<Id, 7>
<Assign, >
<IntLit, 2398>
<Semi, >
<Semi, >
<Semi, >
<Semi, >
<Semi, >
<Semi, >
<Semi, >
<Let, >
<Id, 8>
<Assign, >
<IntLit, 2>
<Or, >
<IntLit, 5>
<Semi, >
<Let, >
<Bool, >
<Id, 9>
<Assign, >
<False, >
<Semi, >
<Let, >
<Bool, >
<Id, 10>
<Assign, >
<True, >
<Semi, >
<Id, 10>
<AndAssign, >
<Id, 9>
<Semi, >
<RBrack, >
```

Tabla de símbolos:

```
table #0:
* 'global'
* 'nothing'
* 'main'
* 'b'
* 'd'
* 'aux'
* 'foo'
* 'bar'
* 'oper'
* 'bool'
* 'george'
```

A.2 Casos Incorrectos

unterm.javascript Fichero con errores de sentencias incompletas.

```
/* unfinished strings */
let string foo = "im not finishing this string
let string bar = "im not finishing this one either\
let string rep = "last one\q

/* unfinished floats */
let float x = 478234.
let float y = 78234..
let float z = 2343...

/*** * / * ///** / this comment is finished /* /* ***/
/*** this one is not ** ****** *** /*

function int main {
    let string useless = "this variable won't be recognised";
    return 0;
}
```

Diagnósticos:

```
unterm.javascript:2:18: error: missing terminating character '"' on string literal
   2 | let string foo = "im not finishing this string
     |                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
unterm.javascript:3:18: error: missing terminating character '"' on string literal
   3 | let string bar = "im not finishing this one either\
     |                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
unterm.javascript:4:27: warning: unknown escape sequence '\q'
   4 | let string rep = "last one\q
     |                           ^~
unterm.javascript:4:18: error: missing terminating character '"' on string literal
   4 | let string rep = "last one\q
     |                  ^~~~~~~~~~
unterm.javascript:7:15: error: expected digit after '.' in float literal
   7 | let float x = 478234.
     |               ^~~~~~
unterm.javascript:8:15: error: expected digit after '.' in float literal
   8 | let float y = 78234..
     |               ^~~~~
unterm.javascript:8:21: error: illegal character '.' in program
   8 | let float y = 78234..
     |                     ^
unterm.javascript:9:15: error: expected digit after '.' in float literal
   9 | let float z = 2343...
     |               ^~~~
unterm.javascript:9:20: error: illegal character '.' in program
   9 | let float z = 2343...
     |                    ^
unterm.javascript:9:21: error: illegal character '.' in program
   9 | let float z = 2343...
     |                     ^
unterm.javascript:12:1: error: unterminated comment
   12 | /*** this one is not ** ****** *** /*
      | ^~
```

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

```
/* string with 64 characters */
let string foo = "ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff";

/* string with 65 characters */
let string bar = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";

/* max int16 */
let int imax = 32767;
let int imax_plus_one = 32768;

/* max float */
let float fmax = 340282346638528859811704183484516925440.0;
let float fmax_plus_more = 3402823466385288598117041834845169254382923892341.0;
let float lots_of_decimals = 0.23470283498209348092184092388452903809 28;
```

Diagnósticos:

```
overflow.javascript:5:18: error: string literal is too long, length is 65 but the
    maximum is 64
   5 | let string bar = "
       bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
     |
         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
overflow.javascript:9:25: error: integer literal out of range for 16-byte type
   9 | let int imax_plus_one = 32768;
     |                         ^~~~~
overflow.javascript:13:27: error: float literal out of range for 32-byte type
  13 | let float fmax_plus_one = 3402823466385288598117041834845169254382923892341.0;
     |                           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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

```
123.45.67 12abc 12.34.56..78
-+*/% &&& ||| != == <= >= >< =!= &== |= <=>
"unterminated string" "string with illegal char \q" ""empty""
"bad\escape" "no closing quote
/* nested comment */ /* comment */ /* unclosed comment
@@@ ### $$$ %%% ^^^ ??? !!! <<< >>> === ====
++ -- ** // %% =!= &== |= != <=> &&|||!
"unterminated string again /* unclosed comment
"bad string with \x illegal escape" "" ""empty""
/* final comment */ /* another unclosed
```

Diagnósticos:

```
noise.javascript:1:7: error: illegal character '.' in program
   1 | 123.45.67 12abc 12.34.56..78
     |       ^
noise.javascript:1:22: error: illegal character '.' in program
   1 | 123.45.67 12abc 12.34.56..78
     |                      ^
noise.javascript:1:23: error: expected digit after '.' in float literal
   1 | 123.45.67 12abc 12.34.56..78
     |                       ^~~
noise.javascript:1:26: error: illegal character '.' in program
   1 | 123.45.67 12abc 12.34.56..78
     |                          ^
noise.javascript:2:9: error: illegal character '&' in program
   2 | -+*/% &&& ||| != == <= >= >< =!= &== |= <=>
     |         ^
noise.javascript:2:13: error: illegal character '|' in program
   2 | -+*/% &&& ||| != == <= >= >< =!= &== |= <=>
     |             ^
noise.javascript:2:38: error: illegal character '|' in program
   2 | -+*/% &&& ||| != == <= >= >< =!= &== |= <=>
     |                                      ^
noise.javascript:3:53: warning: unknown escape sequence '\q'
   3 |     "unterminated string" "string with illegal char \q" ""empty""
     |                                                     ^~
noise.javascript:4:5: warning: unknown escape sequence '\e'
   4 | "bad\escape" "no closing quote
     |     ^~
noise.javascript:4:14: error: missing terminating character '"' on string literal
   4 | "bad\escape" "no closing quote
     |              ^~~~~~~~~~~~~~~~
noise.javascript:10:21: error: unterminated comment
  10 | /* final comment */ /* another unclosed
     |                     ^~
```